器生成的.winmdobj，.pdb 和.xml (doc)文件。WinMDExp.exe 实用程序检查文件的元数据，确保你的类型符合本章开头讨论的 WinRT 类型系统的各种规则。实用程序还会修改.winmdobj 文件中的元数据；它一点儿都不会碰 IL 代码。具体地说，实用程序只是将 CLR 类型映射到等价的 WinRT 类型。例如，对.NET Framework **IList<String>**类型的引用被更改为 WinRT 的 **IVector<String>**类型。WinMDExp.exe 输出的是可供其他语言使用的.winmd 文件。

可用.NET Framework 的 IL 反汇编器工具(ILDasm.exe)检查.winmd 文件的内容。ILDasm.exe 默认显示文件的原始内容。但它支持用**/project** 命令行开关显示将 WinRT 类型投射成.NET Framework 等价类型之后的元数据。



重要提示 托管代码使用同样用托管代码写成的 WinRT 组件时，CLR 将其视为普通的托管组件。也就是说，CLR 不会创建 CCW 和 RCW，不通过这些包装器来调用 WinRT API。这显著增强了性能。但在测试组件时，API 的调用方式有别于从其他语言(如原生 C/C++或 JavaScript)调用时的方式。所以，除了性能和内存消耗不能反映实际情况，托管代码还能向要求一个 **String** 的 WinRT API 传递 **null** 而不引发 **ArgumentNullException**。另外，用托管代码实现的 WinRT API 可操作传入的数组，调用者能在 API 返回时看到更改过的数组内容。而一般情况下，WinRT 类型系统禁止修改传给 API 的数组。还有其他未列出的差异，所以务必小心。

以下代码演示了如何用 C#实现各种 WinRT 组件。组件利用了本章讨论的许多功能，我用大量注释解释了具体发生的事情。用 C#实现 WinRT 组件时，建议把它作为模板使用。

```
/******************************************************************************
Module:  WinRTComponents.cs
Notices: Copyright (c) 2012 by Jeffrey Richter
******************************************************************************/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using System.Threading;
using System.Threading.Tasks;
using Windows.Foundation;
using Windows.Foundation.Metadata;

// The namespace MUST match the assembly name and cannot be "Windows"
namespace Wintellect.WinRTComponents {
   // [Flags]  // Must not be present if enum is int; required if enum is uint
   public enum WinRTEnum : int {    // Enums must be backed by int or uint
      None,
      NotNone
   }


   // Structures can only contain core data types, String, & other structures
   // No constructors or methods are allowed
   public struct WinRTStruct {
      public Int32 ANumber;
      public String AString;
```

```
   public WinRTEnum AEnum;    // Really just a 32-bit integer
}


// Delegates must have WinRT-compatible types in the signature (no BeginInvoke/EndInvoke)
public delegate String WinRTDelegate(Int32 x);


// Interfaces can have methods, properties, & events but cannot be generic.
public interface IWinRTInterface {
   // Nullable<T> marshals as IReference<T>
   Int32? InterfaceProperty { get; set; }
}

// Members without a [Version(#)] attribute default to the class's
// version (1) and are part of the same underlying COM interface
// produced by WinMDExp.exe.
[Version(1)]
// Class must be derived from Object, sealed, not generic,
// implement only WinRT interfaces, & public members must be WinRT types
public sealed class WinRTClass : IWinRTInterface {
   // Public fields are not allowed

   #region Class can expose static methods, properties, and events
   public static String StaticMethod(String s) { return "Returning " + s; }
   public static WinRTStruct StaticProperty { get; set; }

   // In JavaScript 'out' parameters are returned as objects with each
   // parameter becoming a property along with the return value
   public static String OutParameters(out WinRTStruct x, out Int32 year) {
      x = new WinRTStruct { AEnum = WinRTEnum.NotNone, ANumber = 333, AString = "Jeff" };
      year = DateTimeOffset.Now.Year;
      return "Grant";
   }
   #endregion

   // Constructor can take arguments but not out/ref arguments
   public WinRTClass(Int32? number) { InterfaceProperty = number; }

   public Int32? InterfaceProperty { get; set; }

   // Only ToString is allowed to be overridden
   public override String ToString() {
      return String.Format("InterfaceProperty={0}",
         InterfaceProperty.HasValue ? InterfaceProperty.Value.ToString() : "(not set)");
   }

   public void ThrowingMethod() {
      throw new InvalidOperationException("My exception message");

      // To throw a specific HRESULT, use COMException instead
      //const Int32 COR_E_INVALIDOPERATION = unchecked((Int32)0x80131509);
      //throw new COMException("Invalid Operation", COR_E_INVALIDOPERATION);
   }

   #region Arrays are passed, returned OR filled; never a combination
   public Int32 PassArray([ReadOnlyArray] /* [In] implied */ Int32[] data) {
      // NOTE: Modified array contents MAY not be marshaled out; do not modify the array
      return data.Sum();
   }

   public Int32 FillArray([WriteOnlyArray] /* [Out] implied */ Int32[] data) {
      // NOTE: Original array contents MAY not be marshaled in;
      // write to the array before reading from it
      for (Int32 n = 0; n < data.Length; n++) data[n] = n;
      return data.Length;
   }
```

```
public Int32[] ReturnArray() {
   // Array is marshaled out upon return
   return new Int32[] { 1, 2, 3 };
}
#endregion

// Collections are passed by reference
public void PassAndModifyCollection(IDictionary<String, Object> collection) {
   collection["Key2"] = "Value2";  // Modifies collection in place via interop
}

#region Method overloading
// Overloads with same # of parameters are considered identical to JavaScript
public void SomeMethod(Int32 x) { }

[Windows.Foundation.Metadata.DefaultOverload] // Attribute makes this method the default overload
public void SomeMethod(String s) { }
#endregion

#region Automatically implemented event
public event WinRTDelegate AutoEvent;

public String RaiseAutoEvent(Int32 number) {
   WinRTDelegate d = AutoEvent;
   return (d == null) ? "No callbacks registered" : d(number);
}
#endregion

#region Manually implemented event
// Private field that keeps track of the event's registered delegates
private EventRegistrationTokenTable<WinRTDelegate> m_manualEvent = null;

// Manual implementation of the event's add and remove methods
public event WinRTDelegate ManualEvent {
   add {
      // Gets the existing table, or creates a new one if the table is not yet initialized
      return EventRegistrationTokenTable<WinRTDelegate>
         .GetOrCreateEventRegistrationTokenTable(ref m_manualEvent).AddEventHandler(value);
   }
   remove {
      EventRegistrationTokenTable<WinRTDelegate>
         .GetOrCreateEventRegistrationTokenTable(ref m_manualEvent).RemoveEventHandler(value);
   }
}

public String RaiseManualEvent(Int32 number) {
   WinRTDelegate d = EventRegistrationTokenTable<WinRTDelegate>
      .GetOrCreateEventRegistrationTokenTable(ref m_manualEvent).InvocationList;
   return (d == null) ? "No callbacks registered" : d(number);
}
#endregion

#region Asynchronous methods
// Async methods MUST return IAsync[Action|Operation](WithProgress)
// NOTE: Other languages see the DataTimeOffset as Windows.Foundation.DateTime
public IAsyncOperationWithProgress<DateTimeOffset, Int32> DoSomethingAsync() {
   // Use the System.Runtime.InteropServices.WindowsRuntime.AsyncInfo's Run methods to
   // invoke a private method written entirely in managed code
   return AsyncInfo.Run<DateTimeOffset, Int32>(DoSomethingAsyncInternal);
}

// Implement the async operation via a private method using normal .NET technologies
private async Task<DateTimeOffset> DoSomethingAsyncInternal(
   CancellationToken ct, IProgress<Int32> progress) {

   for (Int32 x = 0; x < 10; x++) {
```

```
            // This code supports cancellation and progress reporting
            ct.ThrowIfCancellationRequested();
            if (progress != null) progress.Report(x * 10);
            await Task.Delay(1000); // Simulate doing something asynchronously
         }
         return DateTimeOffset.Now; // Ultimate return value
      }

      public IAsyncOperation<DateTimeOffset> DoSomethingAsync2() {
         // If you don't need cancellation & progress, use
         // System.WindowsRuntimeSystemExtensions' AsAsync[Action|Operation] Task
         // extension methods (these call AsyncInfo.Run internally)
         return DoSomethingAsyncInternal(default(CancellationToken), null).AsAsyncOperation();
      }
      #endregion

      // After you ship a version, mark new members with a [Version(#)] attribute
      // so that WinMDExp.exe puts the new members in a different underlying COM
      // interface. This is required since COM interfaces are supposed to be immutable.
      [Version(2)]
      public void NewMethodAddedInV2() { }
   }
}
```

以下 JavaScript 代码演示了如何访问前面的所有 WinRT 组件和功能。

```
function () {
   // Make accessing the namespace more convenient in the code
   var WinRTComps = Wintellect.WinRTComponents;

   // NOTE: The JavaScript VM projects WinRT APIs via camel casing

   // Access WinRT type's static method & property
   var s = WinRTComps.WinRTClass.staticMethod(null); // NOTE: JavaScript pass "null" here!
   var struct = { anumber: 123, astring: "Jeff", aenum: WinRTComps.WinRTEnum.notNone };
   WinRTComps.WinRTClass.staticProperty = struct;
   s = WinRTComps.WinRTClass.staticProperty; // Read it back

   // If the method has out parameters, they and the return value
   // are returned as an object's properties
   var s = WinRTComps.WinRTClass.outParameters();
   var name = s.value; // Return value
   var struct = s.x; // an 'out' parameter
   var year = s.year; // another 'out' parameter

   // Construct an instance of the WinRT component
   var winRTClass = new WinRTComps.WinRTClass(null);
   s = winRTClass.toString(); // Call ToString()

   // Demonstrate throw and catch
   try { winRTClass.throwingMethod(); }
   catch (err) { }

   // Array passing
   var a = [1, 2, 3, 4, 5];
   var sum = winRTClass.passArray(a);

   // Array filling
   var arrayOut = [7, 7, 7]; // NOTE: fillArray sees all zeros!
   var length = winRTClass.fillArray(arrayOut); // On return, arrayOut = [0, 1, 2]

   // Array returning
   a = winRTClass.returnArray(); // a = [ 1, 2, 3]

   // Pass a collection and have its elements modified
   var localSettings = Windows.Storage.ApplicationData.current.localSettings;
```

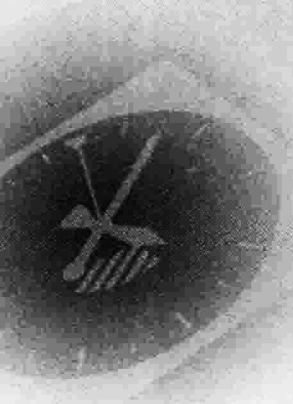
```
    localSettings.values["Key1"] = "Value1";
    winRTClass.passAndModifyCollection(localSettings.values);
    // On return, localSettings.values has 2 key/value pairs in it

    // Call overloaded method
    winRTClass.someMethod(5); // Actually calls SomeMethod(String) passing "5"

    // Consume the automatically implemented event
    var f = function (v) { return v.target; };
    winRTClass.addEventListener("autoevent", f, false);
    s = winRTClass.raiseAutoEvent(7);

    // Consume the manually implemented event
    winRTClass.addEventListener("manualevent", f, false);
    s = winRTClass.raiseManualEvent(8);

    // Invoke asynchronous method supporting progress, cancelation, & error handling
    var promise = winRTClass.doSomethingAsync();
    promise.then(
      function (result) { console.log("Async op complete: " + result); },
      function (error) { console.log("Async op error: " + error); },
      function (progress) {
        console.log("Async op progress: " + progress);
       //if (progress == 30) promise.cancel(); // To test cancelation
      });
}
```

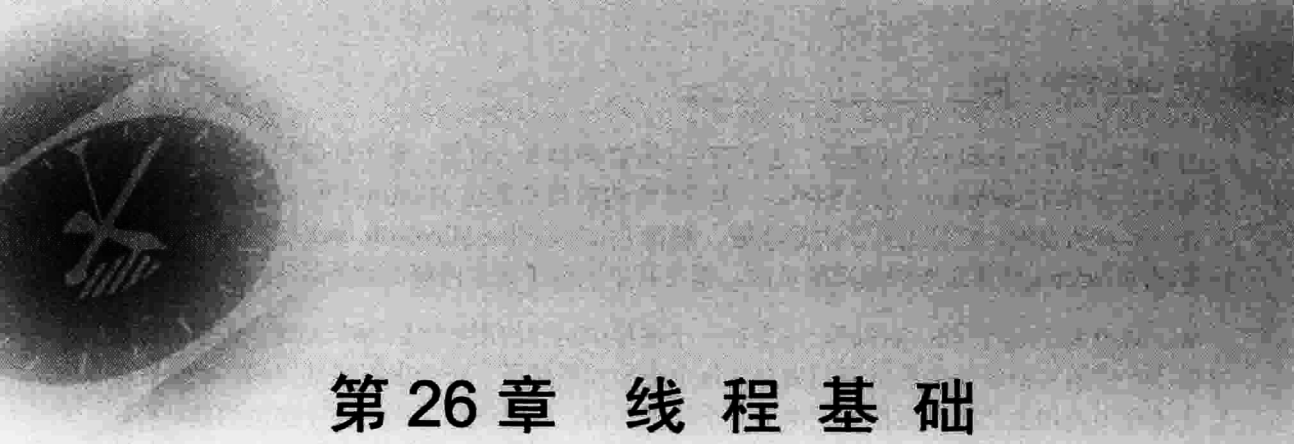


第V部分　线程处理

- 第 26 章　线程基础
- 第 27 章　计算限制的异步操作
- 第 28 章　I/O 限制的异步操作
- 第 29 章　基元线程同步构造
- 第 30 章　混合线程同步构造

數位時代　精選教程

第 35 章 ◀ 第 75 章 ◀ 第 85 章 ◀ 第 93 章 ◀ 第 105 章 ◀



第26章 线程基础

本章内容：

- Windows 为什么要支持线程
- 线程开销
- 停止疯狂
- CPU 发展趋势
- CLR 线程和 Windows 线程
- 使用专用线程执行异步的计算限制操作
- 使用线程的理由
- 线程调度和优先级
- 前台线程和后台线程
- 继续学习

本章将介绍线程的基本概念，帮助开发人员理解线程及其使用。我将解释 Microsoft Windows 为什么引入线程的概念、CPU 发展趋势、CLR 线程和 Windows 线程的关系、线程开销、Windows 如何调度线程以及公开了线程属性的 Microsoft .NET Framework 类。

本书第 V 部分“线程处理”的各个章节将解释 Windows 和 CLR 如何协同提供一个线程处理架构。希望通过这些内容帮你打下一个良好的基础，学会高效使用线程来设计和构建响应灵敏的、可靠的、可伸缩的应用程序和组件。

26.1 Windows 为什么要支持线程

在计算机的早期岁月，操作系统没有线程的概念。事实上，整个系统只运行着一个执行线程，其中同时包含操作系统代码和应用程序代码。只用一个执行线程的问题在于，长时间运行的任务会阻止其他任务执行。例如，在 16 位 Windows 的那些日子，打印文档的应用程序很容易“冻结”整个机器，造成 OS(操作系统)和其他应用程序停止响应。有些应用程序的 bug 会造成死循环，同样会造成整个机器停止工作。

遇到这个问题，用户只好按 Reset 键或电源开关重启计算机。用户对此深恶痛绝(事实上，他们现在一样会)，因为所有正在运行的应用程序都会终止。更重要的是，这些应用程序正在处理的数据都会无端地丢失。Microsoft 明白 16 位 Windows 不是理想的操作系统。随着计算机工业的持续进步，它不足以使 Microsoft 保持领先地位。所以，他们计划构建一个新

的 OS 来满足企业和个人的需要。这个新的 OS 必须健壮、可靠、易于伸缩和安全，而且它必须弥补 16 位 Windows 的许多不足。新的 OS 内核最初通过 Microsoft Windows NT 发布。经过多年的发展，它已进行了大量改进，增加了大量功能。Microsoft 每次发布客户端和服务器 Windows 操作系统的最新版本时，都在其中采用了这个内核的最新版本。

Microsoft 设计这个 OS 内核时，决定在一个**进程**中运行应用程序的每个实例。进程实际是应用程序的实例要使用的资源的集合。每个进程都被赋予了一个虚拟地址空间，确保在一个进程中使用的代码和数据无法由另一个进程访问。这就确保了应用程序实例的健壮性，因为一个进程无法破坏另一个进程使用的代码或数据。此外，进程访问不了 OS 的内核代码和数据；所以，应用程序代码破坏不了操作系统代码或数据。由于应用程序代码破坏不了其他应用程序或者 OS 自身，所以用户的计算体验变得更好了。除此之外，系统变得比以往更安全，因为应用程序代码无法访问另一个应用程序或者 OS 自身使用的用户名、密码、信用卡资料或其他敏感信息。

听起来不错，但 CPU 本身呢？应用程序发生死循环会发生什么？如果机器只有一个 CPU，它会执行死循环，不能执行其他任何东西。所以，虽然数据无法被破坏，而且更安全，但系统仍然可能停止响应。Microsoft 需要修正这个问题，他们拿出的方案就是线程。作为一个 Windows 概念，**线程**的职责是对 CPU 进行虚拟化。Windows 为每个进程都提供了该进程专用的线程(功能相当于一个 CPU)。应用程序的代码进入死循环，与那个代码关联的进程会“冻结”，但其他进程(它们有自己的线程)不会冻结，它们会继续执行！

26.2 线程开销

线程很强大，因为它们使 Windows 即使在执行长时间运行的任务时也能随时响应。另外，线程允许用户使用一个应用程序(比如“任务管理器”)强制终止似乎已经冻结的应用程序(它也有可能正在执行一个长时间运行的任务)。但和一切虚拟化机制一样，线程有空间(内存耗用)和时间(运行时的执行性能)上的开销。

下面更详细地探讨这种开销。每个线程都有以下要素。

- **线程内核对象(thread kernel object)**
  OS 为系统中创建的每个线程都分配并初始化这种数据结构之一。数据结构包含一组对线程进行描述的属性(本章后面讨论)。数据结构还包含所谓的线程上下文(thread context)。上下文是包含 CPU 寄存器集合的内存块。对于 x86，x64 和 ARM CPU 架构，线程上下文分别使用约 700，1240 和 350 字节的内存。

- **线程环境块(thread environment block，TEB)**
  TEB 是在用户模式(应用程序代码能快速访问的地址空间)中分配和初始化的内存块。TEB 耗用 1 个内存页(x86，x64 和 ARM CPU 中是 4 KB)。TEB 包含线程的异常处理链首(head)。线程进入的每个 **try** 块都在链首插入一个节点(node)；线程退出 **try** 块时从链中删除该节点。此外，TEB 还包含线程的“线程本地存储”数据，以及由 GDI(Graphics

Device Interface，图形设备接口)和 OpenGL 图形使用的一些数据结构。

- **用户模式栈(user-mode stack)**
  用户模式栈存储传给方法的局部变量和实参。它还包含一个地址；指出当前方法返回时，线程应该从什么地方接着执行。Windows 默认为每个线程的用户模式栈分配 1 MB 内存。更具体地说，Windows 只是保留 1 MB 地址空间，在线程实际需要时才会提交(调拨)物理内存。

- **内核模式栈(kernel-mode stack)**
  应用程序代码向操作系统中的内核模式函数传递实参时，还会使用内核模式栈。出于对安全的考虑，针对从用户模式的代码传给内核的任何实参，Windows 都会把它们从线程的用户模式栈复制到线程的内核模式栈。一经复制，内核就可验证实参的值。由于应用程序代码不能访问内核模式栈，所以应用程序无法更改验证后的实参值。OS 内核代码开始处理复制的值。除此之外，内核会调用它自己内部的方法，并利用内核模式栈传递它自己的实参、存储函数的局部变量以及存储返回地址。在 32 位 Windows 上运行，内核模式栈大小是 12 KB；64 位 Windows 是 24 KB。

- **DLL 线程连接(attach)和线程分离(detach)通知**
  Windows 的一个策略是，任何时候在进程中创建线程，都会调用进程中加载的所有非托管 DLL 的 **DllMain** 方法，并向该方法传递 **DLL_THREAD_ATTACH** 标志。类似地，任何时候线程终止，都会调用进程中的所有非托管 DLL 的 **DllMain** 方法，并向方法传递 **DLL_THREAD_DETACH** 标志。有的 DLL 需要获取这些通知，才能为进程中创建/销毁的每个线程执行特殊的初始化或(资源)清理操作。例如，C-Runtime 库 DLL 会分配一些线程本地存储状态。线程使用 C-Runtime 库中包含的函数时需要用到这些状态。

在 Windows 的早期岁月，许多进程最多只加载五六个 DLL。但如今，随便一个进程就可能加载几百个 DLL。就拿目前来说，在我的机器上，Microsoft Visual Studio 在它的进程地址空间加载了大约 470 个 DLL！这意味着每次在 Visual Studio 中新建一个线程，都必须先调用 470 个 DLL 函数，然后线程才能开始做它想做的事情。Visual Studio 进程终止时，这 470 个函数还要调用一遍。这严重影响了在进程中创建和销毁线程的性能。[①]

你现在已经知道了创建线程、让它进驻系统以及最后销毁它所需的全部空间和时间开销。但还没完——接着讨论上下文切换。单 CPU 计算机一次只能做一件事情。所以，Windows 必须在系统中的所有线程(逻辑 CPU)之间共享物理 CPU。

Windows 任何时刻只将一个线程分配给一个 CPU。那个线程能运行一个“时间片”(有时也称为“量”或者“量程”，即 quantum) 的长度。时间片到期，Windows 就上下文切换到另一个线程。每次上下文切换都要求 Windows 执行以下操作。

---

① C#和其他大多数托管编程语言生成的 DLL 没有 **DllMain** 函数。所以，托管 DLL 不会收到 **DLL_THREAD_ATTACH** 和 **DLL_THREAD_DETACH** 通知，这提升了性能。此外，非托管 DLL 可调用 Win32 **DisableThreadLibraryCalls** 函数来决定不理会这些通知。遗憾的是，许多非托管开发人员都不知道有这个函数。

1. 将 CPU 寄存器的值保存到当前正在运行的线程的内核对象内部的一个上下文结构中。
2. 从现有线程集合中选出一个线程供调度。如果该线程由另一个进程拥有，Windows 在开始执行任何代码或者接触任何数据之前，还必须切换 CPU“看见”的虚拟地址空间。
3. 将所选上下文结构中的值加载到 CPU 的寄存器中。

上下文切换完成后，CPU 执行所选的线程，直到它的时间片到期。然后发生下次上下文切换。Windows 大约每 30 毫秒执行一次上下文切换。上下文切换是净开销；也就是说，上下文切换所产生的开销不会换来任何内存或性能上的收益。Windows 执行上下文切换，向用户提供一个健壮的、响应灵敏的操作系统。

现在，假如一个应用程序的线程进入死循环，Windows 会定期抢占(preempt)它，将新线程分配给 CPU，让新线程有机会运行。假定新线程是“任务管理器”的线程，用户就可利用“任务管理器”终止包含了死循环线程的进程。之后，进程会终止，它处理的所有数据会被销毁。但系统中的其他所有进程都继续运行，不会丢失它们的数据，用户当然也不需要重启计算机。所以，上下文切换通过牺牲性能换来了好得多的用户体验。

事实上，上下文切换对性能的影响可能超出你的想象。是的，当 Windows 上下文切换到另一个线程时，会产生一定的性能损失。但是，CPU 现在是要执行一个不同的线程，而之前的线程的代码和数据还在 CPU 的高速缓存(cache)中，这使 CPU 不必经常访问 RAM(它的速度比 CPU 高速缓存慢得多)。当 Windows 上下文切换到新线程时，这个新线程极有可能要执行不同的代码并访问不同的数据，这些代码和数据不在 CPU 的高速缓存中。因此，CPU 必须访问 RAM 来填充它的高速缓存，以恢复高速执行状态。但在 30 毫秒之后，一次新的上下文切换又发生了。



重要提示 一个时间片结束时，如果 Windows 决定再次调度同一个线程(而不是切换到另一个线程)，那么 Windows 不会执行上下文切换。相反，线程将继续运行。这显著改进了性能。注意，在设计自己的代码时，上下文切换能避免就要尽量避免。



重要提示 线程可自主提前终止其时间片。这经常发生，因为线程经常要等待 I/O 操作(键盘、鼠标、文件、网络等)结束。例如，“记事本”程序的线程经常都会处于空闲状态，什么事情都不做；这个线程是在等待输入。如果用户按键盘上的 J 键，Windows 会唤醒“记事本”线程，让它处理按键操作。“记事本”线程可能花 5 毫秒处理按键，然后调用一个 Win32 函数，告诉 Windows 它准备好处理下一个输入事件。如果没有更多的输入事件，Windows 使“记事本”线程进入等待状态(时间片剩余的部分就放弃了)，使线程在任何 CPU 上都不再调度，直到发生下一次输入事件。这增强了系统的总体性能，因为正在等待 I/O 操作完成的线程不会在 CPU 上调度，所以不会浪费 CPU 时间，而节省出来的时间则可以供 CPU 调度其他线程。

执行上下文切换所需的时间取决于 CPU 架构和速度。而填充 CPU 缓存所需的时间取决于系统中运行的应用程序、CPU 缓存的大小以及其他各种因素。所以，无法为每一次上下文切换的时间开销给出确定值，甚至无法给出估计值。唯一确定的是，要构建高性能应用程序和组件，就应该尽量避免上下文切换。

此外，执行垃圾回收时，CLR 必须挂起(暂停)所有线程，遍历它们的栈来查找根以便对堆中的对象进行标记[①]，再次遍历它们的栈(有的对象在压缩期间发生了移动，所以要更新它们的根)，再恢复所有线程。所以，减少线程的数量也会显著提升垃圾回收器的性能。每次使用调试器并遇到断点，Windows 都会挂起正在调试的应用程序中的所有线程，并在单步执行或者运行应用程序时恢复所有线程。所以，线程越多，调试体验越差。

根据上述讨论，结论是必须尽量避免使用线程，因为它们要耗用大量内存，而且需要相当多的时间来创建、销毁和管理。Windows 在线程之间进行上下文切换，以及在发生垃圾回收的时候，也会浪费不少时间。然而，根据上述讨论，我们还得出了另一个结论，那就是有时必须使用线程，因为它们使 Windows 变得更健壮，响应更灵敏。

应该指出的是，安装了多个 CPU(或者一个多核 CPU)的计算机可以真正同时运行几个线程，这提升了应用程序的可伸缩性(用更少的时间做更多的工作)。Windows 为每个 CPU 内核都分配一个线程，每个内核都自己执行到其他线程的上下文切换。Windows 确保单个线程不会同时在多个内核上调度，因为这会带来巨大的混乱。如今的许多计算机都配备了多个 CPU、超线程 CPU 或者多核 CPU。但在 Windows 最初设计时，单 CPU 计算机才是主流，所以 Windows 设计了线程来增强系统的响应能力和可靠性。今天，线程还被用于增强应用程序的可伸缩性，但只有在多 CPU(多核)计算机上才有可能。

本书剩余各章将讨论如何利用 Windows 和 CLR 提供的各种机制，当代码在多 CPU(多核)计算机上运行时，创建尽量少的线程并保证代码的响应能力和伸缩性。

26.3　停止疯狂

如果只关心性能，那么任何机器最优的线程数就是那台机器的CPU数目(从现在开始将CPU的每个内核都当作一个 CPU)。所以，安装一个 CPU 的机器最好只有一个线程，安装两个 CPU 的机器最好只有两个线程，以此类推。理由非常明显：如果线程数超过了 CPU 的数目，就会产生上下文切换和性能损失。如果每个 CPU 只有一个线程，就不会有上下文切换，线程将全速运行。

然而，Microsoft 设计 Windows 时，决定侧重于可靠性和响应能力，而非侧重于速度和性能。我拥护这个决定：如果今天的应用程序仍然会造成 OS 和其他应用程序“冻结”，我想没有人愿意使用 Windows 或者.NET Framework。因此，Windows 为每个进程提供了该进程专用的线程来增强系统的可靠性和响应能力。例如，在我的机器上运行“任务管理器”并选

① “标记”是垃圾回收器的第一个阶段，详见 21.1.2 节“垃圾回收算法”。——译注

择“性能”选项卡，会看到如图 26-1 所示的结果。

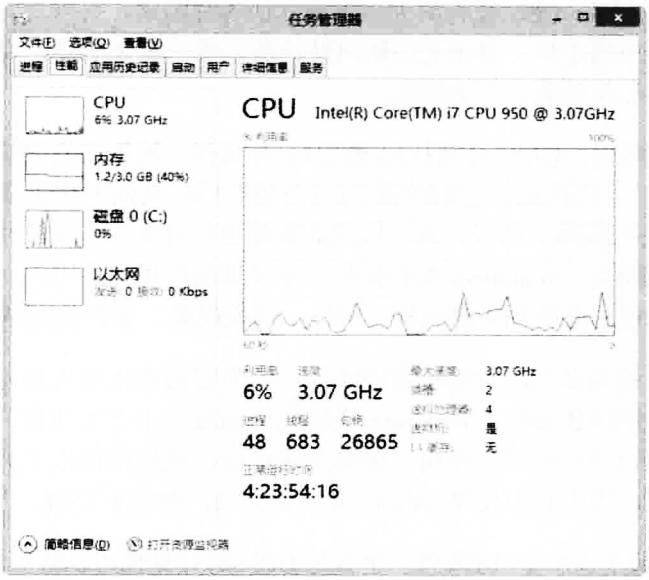


图 26-1　任务管理器的“性能”选项卡

看来我的机器正在运行 48 个进程，所以可认为机器上至少运行着 48 个线程，因为每个进程都至少有一个线程。但“任务管理器”显示我的机器实际有 683 个线程！这意味着仅仅是线程栈就需要分配 683 MB 内存，而我的机器总共才 3 GB RAM！另外，这还意味着每个进程平均有大约 14.2 个线程，而我的 4 核电脑理想情况下不应该是每进程 4 线程吗？

再来看看 CPU 利用率读数：它显示我的 CPU“忙”的时间是 6%。这意味着在 94%的时间里，这 683 个线程什么事情都没做——它们白白霸占着内存；线程未运行时，这些内存是不会使用的。你肯定会问：这些应用程序真的需要这些在 94%的时间里什么事情都不做的线程吗？答案明显是否定的。要知道哪些进程最浪费，请单击“详细信息”标签，添加“线程”列[①]，再按降序对这个列进行排序，如图 26-2 所示。

如你所见，System 创建了 111 个线程，CPU 利用率为 0%。Explorer(资源管理器)创建了 82 个线程，CPU 利用率为 0%。Visual Studio(Devenv.exe)创建了 38 个线程，CPU 利用率为 0%。Microsoft Outlook 创建了 17 个线程，CPU 利用率为 0%；我可以一直列举下去。到底发生了什么事情？

在 Windows 中，进程是十分“昂贵”的。创建一个进程通常要花几秒钟的时间，必须分配大量内存，这些内存必须初始化，EXE 和 DLL 文件必须从磁盘上加载，等等。相反，在 Windows 中创建线程则十分“廉价”。所以，开发人员决定停止创建进程，改为创建线程。这正是我们看到有这么多线程的原因。但是，虽然线程比进程廉价，它们和其他系统资源相比仍然十分昂贵，所以还是应该省着用，而且要用得恰当。

① 右击现有的列，选择“选择列”。



图 26-2　任务管理器显示进程的详细信息

可以肯定地说，刚才讨论的所有这些应用程序都在以效率低下的方式使用线程。所有这些线程在系统中都没有存在的必要。在应用程序中分配资源是十分正常的一件事情，但分配了又不用，这又算是什么呢？这纯属浪费！另外，为线程栈分配这么多内存，意味着一些更重要的数据(比如用户的文档)获得的内存变少了。①

更可怕的是，假如这些进程在用户的“远程桌面服务”会话中运行，而且这台机器有 100 个用户在访问，那么会发生什么？在这种情况下，将会有 100 个 Outlook 实例，每个实例都创建 17 个什么都不干的线程。这 1700 个“坐吃等死”的线程每个都有自己的内核对象、TEB、用户模式栈、内核模式栈等。由此浪费的资源将非常可观。必须停止这种疯狂的举动，尤其是如果 Microsoft 想为笔记本用户提供一个好的使用体验的话(许多笔记本都只有 2~4 GB RAM)。本书剩余各章将讲述如何正确设置应用程序，以一种高效的方式使用尽量少的线程。

26.4　CPU 发展趋势

CPU 厂商过去只知道一味地提高 CPU 速度。所以，在旧机器上运行得慢的程序在新机器上

① 为了让你体会到情况有多糟，我忍不住想和你分享另一个例子。请做个试验：打开“记事本”程序(Notepad.exe)，用“任务管理器”查看它包含了多少个线程(这时看到的是 1)。然后，选择“记事本”的“文件”|“打开”菜单来显示通用的“打开”对话框。保持这个对话框的打开状态，在“任务管理器”中查看现在创建了多少个新线程。在我的机器上，仅仅是因为打开了这个对话框，就创建了 14 个额外的线程！事实上，使用通用“打开”和“保存”对话框的每个应用程序都会在其内部创建大量额外的线程，这些线程大多数时候都处于空闲状态。另外，即使对话框关闭，其中大多数线程也不会销毁。

一般都会快一些。但 CPU 厂商没有延续这个趋势，因为高速运行的 CPU 会产生大量热量。我几年前从一家著名厂商那里购买了一台新款笔记本电脑。这台电脑的固件有一个 bug，造成风扇时转时不转。结果是，用不了多久，CPU 和主板便烧坏了。硬件厂商更换了机器，并“改进”了固件，使风扇能更频繁地工作。遗憾的是，这又造成耗电过大，因为风扇本身也很耗电。

这些问题是今天的硬件厂商必须面对的。由于不能做到一直提升 CPU 的速度，所以又改为侧重于将晶体管做得更小，使一个芯片能容下更多的晶体管。今天，一个硅芯片可以容纳两个或者更多的 CPU 内核。这样一来，如果在写软件时能利用多个内核，软件就能运行得更快。这是怎么做到的呢？我们“**以一种智能的方式**”使用线程。

今天的计算机使用了以下三种多 CPU 技术。

- **多个 CPU**
  有的计算机安装了多个 CPU。主板上有多个 CPU 插座，每个都可安装一个 CPU。由于主板会变得更大，所以计算机机箱也会变得更大。有的时候，这种机器甚至要安装多个电源以提供额外的功率。这种计算机问世已有几十年的历史，但在普通用户中并不流行，因其体积和价格都不太理想。

- **超线程芯片**
  这种技术(Intel 专利)允许一个芯片在操作系统中显示成两个。芯片中包含两组架构状态(比如 CPU 寄存器)，但芯片只有一组执行资源。对于 Windows，这看起来是安装了两个 CPU，所以 Windows 会同时调度两个线程。但芯片一次只能执行一个线程。一个线程由于缓存未命中(cache miss)、分支预测错误(branch misprediction)或者要等待数据(data dependency)而暂停时，芯片将切换到另一个线程。一切都是在硬件中发生的，Windows 对此一无所知；它以为有两个线程正在并发运行。Windows 不知道实际使用的是超线程 CPU。如果一台机器上安装了多个超线程 CPU，Windows 首先在每个 CPU 上都调度一个线程，使线程真正并发运行，然后在已经处于“忙”状态的 CPU 上调度其他线程。Intel 声称超线程 CPU 能提升 10%～30%的性能。

- **多核芯片**
  包含多个内核的 CPU 芯片几年前就已问世。当我写到这里的时候，双核、四核和八核 CPU 已经“遍地开花”。就连我的笔记本电脑都是四核的。不久，也许我们的手机都普遍 4~8 核。Intel 现在甚至在试验集成了 80 个核的处理器!这是多么强大的计算能力!除此之外，Intel 甚至推出了超线程的多核芯片。

26.5 CLR 线程和 Windows 线程

CLR 使用 Windows 的线程处理功能，所以本书第 V 部分实际是围绕 Windows 向开发人员公开的线程处理功能展开的。我将解释线程在 Windows 中如何工作，以及 CLR 如何改变线程的行为(如果能的话)。然而，如果想更多地学习线程，建议阅读我以前就这个主题出

版的著作，比如我和 Christophe Nasarre 合写的 *Windows via C/C++*(Microsoft Press，2007)[①]。



> **注意**　在.NET Framework 的早期岁月，CLR 团队认为有朝一日 CLR 会提供不一定会映射到 Windows 线程的逻辑线程。但大约在 2005 年的时候，这个尝试宣告失败，CLR 团队放弃了在这方面的努力。所以，今天的 CLR 线程完全等价于 Windows 线程。但.NET Framework 仍然留下了一些能看出当年努力成果的一些东西。例如，**System.Environment** 类公开了 **CurrentManagedThreadId** 属性，返回线程的 CLR ID；而 **System.Diagnostics.ProcessThread** 类公开了 **Id** 属性，返回同一个线程的 Windows ID。**System.Thread** 类的 **BeginThreadAffintiy** 和 **EndThreadAffinity** 方法则处理 CLR 线程没有映射到 Windows 线程的情况。



> **注意**　Microsoft 为 Windows Store 应用移除了和线程处理有关的一些 API，因为它们鼓励不好的编程实践(就像 26.3 节“停止疯狂”描述的那样)，或者是因为它们不利于达成 Microsoft 为 Windows Store 应用设立的目标。例如，整个 **System.Thread** 类都不开放给 Windows Store 应用，因为其中有许多不好的 API(比如 **Start**，**IsBackground**，**Sleep**，**Suspend**，**Resume**，**Join**，**Interrupt**，**Abort**，**BeginThreadAffinity** 和 **EndThreadAffinity**)。我个人赞成这个做法，而且认为早就该这么做。所以，第 26 章到第 30 章讨论了适合桌面应用但不适合 Windows Store 应用的一些 API 和功能。阅读过程中能清楚地认识到为何有一些 API 不适合 Windows Store 应用。

26.6　使用专用线程执行异步的计算限制操作

本节展示如何创建线程来执行异步的计算限制(compute-bound)操作。虽然会展示完整过程，但强烈建议避免使用这个技术。而且事实上，由于 **Thread** 类不可用，所以构建 Windows Store 应用时根本用不了这个技术。相反，应尽量使用线程池来执行异步的计算限制操作，这方面的详情将在第 27 章“计算限制的异步操作”讨论。

但极少数情况下，你可能想显式创建线程来专门执行一个计算限制的操作。如果执行的代码要求线程处于一种特定状态，而这种状态对于线程池线程来说是非同寻常的，就可考虑创建专用线程。例如，满足以下任何条件，就可显式创建自己的线程。

- 线程需要以非普通线程优先级运行。所有线程池线程都以普通优先级运行；虽然可以更改这个优先级，但不建议那样做。另外，在不同的线程池操作之间，对优先级的更

① 简体中文版是《Windows 核心编程(第 5 版)》，繁体中文版是《Windows 应用程式开发经典》。分别由清华大学出版社和悦知文化出版。我参与了它的翻译。——译注

改是无法持续的。

- 需要线程表现为一个前台线程，防止应用程序在线程结束任务前终止。欲知详情，请参见本章后面的 26.9 节“前台线程和后台线程”。线程池线程始终是后台线程。如果 CLR 想终止进程，它们就可能完成不了任务。
- 计算限制的任务需要长时间运行。线程池为了判断是否需要创建一个额外的线程，所采用的逻辑是比较复杂的。直接为长时间运行的任务创建专用线程，就可以避免这个问题。
- 要启动线程，并可能调用 **Thread** 的 **Abort** 方法来提前终止它(参见第 22 章“CLR 寄宿和 AppDomain”)。

为了创建专用线程，要构造 **System.Threading.Thread** 类的实例，向构造器传递一个方法名。以下是 **Thread** 的构造器的原型：

```
public sealed class Thread : CriticalFinalizerObject, ... {
    public Thread(ParameterizedThreadStart start);
    // 未列出不常用的构造器
}
```

start 参数标识专用线程要执行的方法，这个方法必须和 **ParameterizedThreadStart** 委托的签名匹配：[①]

```
delegate void ParameterizedThreadStart(Object obj);
```

构造 **Thread** 对象是轻量级的操作，因为它并不实际创建一个操作系统线程。要实际创建操作系统线程，并让它开始执行回调方法，必须调用 **Thread** 的 **Start** 方法，向它传递要作为回调方法的实参传递的对象(状态)。

以下代码演示了如何创建专用线程并让它异步调用一个方法：

```
using System;
using System.Threading;

public static class Program {
    public static void Main() {
        Console.WriteLine("Main thread: starting a dedicated thread " +
            "to do an asynchronous operation");
        Thread dedicatedThread = new Thread(ComputeBoundOp);
        dedicatedThread.Start(5);

        Console.WriteLine("Main thread: Doing other work here...");
        Thread.Sleep(10000);        // 模拟做其他工作(10 秒)

        dedicatedThread.Join();     // 等待线程终止
        Console.WriteLine("Hit <Enter> to end this program...");
```

① 郑重声明，**Thread** 还提供了一个获取 **ThreadStart** 委托的构造器。**ThreadStart** 委托不接受任何实参，返回 **void**。我个人不建议使用这个构造器和委托，因为它们的功能十分有限。如果你的线程方法(要在线程上执行的方法)要获取一个 **Object** 并返回 **void**，可以使用一个专用线程来调用，也可以使用线程池(第 27 章会解释具体做法)。

```
        Console.ReadLine();
    }

    // 这个方法的签名必须和 ParameterizedThreadStart 委托匹配
    private static void ComputeBoundOp(Object state) {
        // 这个方法由一个专用线程执行

        Console.WriteLine("In ComputeBoundOp: state={0}", state);
        Thread.Sleep(1000);         // 模拟做其他任务(1 秒)

        // 这个方法返回后，专用线程将终止
    }
}
```

编译并运行上述代码可能得到以下输出：

```
Main thread: starting a dedicated thread to do an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

但也可能得到以下输出，我搞不定 Windows 对两个线程进行调度的方式：

```
Main thread: starting a dedicated thread to do an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

注意，**Main** 方法调用了 **Join**。**Join** 方法造成调用线程阻塞(暂停)当前执行的任何代码，直到 **dedicatedThread** 所代表的那个线程销毁或终止。

26.7　使用线程的理由

主要是出于两方面的原因使用线程。

- **可响应性(通常是对于客户端 GUI 应用程序)**
  Windows 为每个进程提供它自己的线程，确保发生死循环的应用程序不会妨碍其他应用程序。类似地，在自己的客户端 GUI 应用程序中，可以将一些工作交给一个线程进行，使 GUI 线程能灵敏地响应用户输入。在这个过程中创建的线程数可能超过 CPU 的核数，会浪费系统资源和损害性能。但用户得到了一个响应灵敏的 UI，所以应用程序的总体使用体验增强了。

- **性能(对于客户端和服务器应用程序)**
  由于 Windows 每个 CPU 调度一个线程，而且多个 CPU 能并发执行这些线程，所以同时执行多个操作能提升性能。当然，只有多 CPU(或多核 CPU)才能得到性能的提升。如今这种机器很普遍，所以所以设计应用程序来使用多个内核是有意义的，这将是第 27 章和第 28 章的主题。

我想和你分享我的一个观点。每台计算机其实都包含一个无比强大的资源，即 CPU 自身。既然已经花了钱在计算机上，它就应该一直工作。换言之，我认为计算机所有 CPU 的利用率都应该一直保持在 100%。但有两个例外。首先，如果计算机用电池供电，那么你可能不希望 CPU 一直保持 100%的利用率，否则会快速耗尽电池。其次，有的数据中心可能情愿

让 10 机器以 50%的 CPU 利用率运行，也不愿意让 5 台机器以 100%的利用率运行。这是由于全速运行的 CPU 会变成散热大户，由此带来的散热成本的大幅提升反而不美。虽然数据中心维护多台机器的费用也不算低(每台机器都要定期进行软硬件升级和监视)，但必须把这方面的费用同运行高级散热系统所需的费用比较。

如果你同意我的观点，那么下一步是决定应该让 CPU 做什么。在我给出我自己的想法之前，先来说一些别的东西。过去，开发人员和最终用户总是觉得计算机不够强劲。所以，除非最终用户通过 UI 元素(比如菜单项、按钮和复选框)给开发人员权限，允许应用程序开始消耗 CPU 资源，否则开发人员绝对不会擅自决定让代码开始执行。

但今非昔比。如今的计算机一般都配备了强大的计算能力。而且可以预见，在不久的将来，计算能力还会大幅提高。本章早些时候，我演示了通过“任务管理器”查看我的 CPU 的利用率只有 6%。如果我的计算机使用的是八核 CPU，而不是现在的四核，那么“任务管理器”可能只会报告 3%的利用率。等 80 核处理器问世时，我的机器看起来会在几乎所有时间里都什么都不做！对于计算机的购买者来说，这感觉就像是花了更多的钱购买了更强大的 CPU，但计算机做的工作变得越来越少！

这正是用户升级电脑动力不足的原因：软件没有充分地利用硬件，用户没有从额外的 CPU 中获益。我认为现在的计算能力有很大的富余，而且还会越来越富余，所以开发人员应该大胆消费它。是时候转变思维了——过去，除非最终用户确定想要获得计算结果，否则我们根本不敢提前计算。但现在有了额外的计算能力，所以开发人员完全不必像以前那样畏手畏脚。

举一个例子：在 Visual Studio 的编辑器中停止输入的时候，Visual Studio 会自动运行编译器并编译代码。这极大提高了开发人员的开发效率，因为他们能在输入时看到源代码的警告和错误，并可立即修正问题。事实上，传统的“编辑-生成-调试”模式逐渐变成“编辑-调试”，因为生成(编译)代码的动作一直都在发生。作为最终用户，你是注意不到这一点的，因为有充足的计算资源可供使用，而且编译器的频繁运行一般不会影响你正在做的其他事情。事实上，我希望 Visual Studio 未来的版本完全移除“生成”菜单，因为这个过程变得完全自动化。这不仅能使应用程序的 UI 变得更简单，还能自动向用户提供“答案”来提高他们使用软件的效率。

移除菜单项这样的 UI 元素后，计算机对最终用户来说变得更简单。他们面临的选项变得更少，需要阅读和理解的概念也变得更少。多核革命使我们能够消除菜单项这样的 UI 元素，进而使软件变得更简单好用，让我的奶奶有一天也能舒舒服服地用上计算机。对于开发人员，移除 UI 元素之后，所需的测试次数一般会变得更少。另外，向最终用户提供更少的选项简化了代码库。如果需要对 UI 元素和文档进行本地化(就像 Microsoft 那样)，那么移除 UI 元素意味着需要写的文档变少了，需要本地化的文档也变少了。这一切能为你的公司节省大量时间和金钱。

下面是更多主动消费计算能力的例子：文档拼写检查和语法检查、电子表格重新计算、磁盘文件索引以提高搜索速度以及硬盘碎片整理以提升 I/O 性能。

我愿意生活在一个 UI 得到大幅简化的世界里。我想有更多的屏幕空间来显示我实际处理的数据。应用程序主动提供更多的信息帮我快速和高效地完成工作，而不是每次都由我自己告诉应用程序去获取信息。过去这几年，硬件在那里静静地等待着软件开发人员去发掘它们的潜力。现在，软件是时候开始创造性地使用硬件了！

26.8　线程调度和优先级

抢占式(preemptive)操作系统必须使用算法判断在什么时候调度哪些线程多长时间。本节讨论 Windows 采用的算法。本章早些时候说过，每个线程的内核对象都包含一个上下文结构。上下文(context)结构反映了线程上一次执行完毕后 CPU 寄存器的状态。在一个时间片(time-slice)之后，Windows 检查现存的所有线程内核对象。在这些对象中，只有那些没有正在等待什么的线程才适合调度。Windows 选择一个可调度的线程内核对象，并上下文切换到它。Windows 实际记录了每个线程被上下文切换到的次数。可以使用像 Microsoft Spy++ 这样的工具查看这个数据。[①]图 26-3 展示了一个线程的属性。注意，这个线程已被调度了 31768 次[②]。

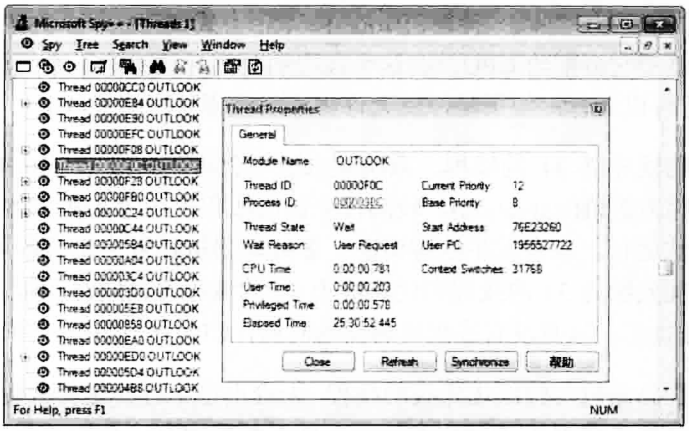


图 26-3　Spy++显示了线程的属性

然后，线程开始执行代码，并在其进程的地址空间处理数据。又过了一个时间片之后，Windows 执行下一次上下文切换。Windows 从系统启动开始便一直执行上下文切换，直到系统关闭为止。

Windows 之所以被称为抢占式多线程(preemptive multithreaded)操作系统，是因为线程可在任何时间停止(被抢占)并调度另一个线程。你在这个方面是有一定控制权的，虽然并不多。

---

① Microsoft Spy++(spyxx.exe)随同 Visual Studio 安装。启动“VS2013 开发人员命令提示”，输入 **spyxx** 即可启动它。——译注

② 补充一点，线程已在系统中存在超过 25 小时了，但它实际只被使用了不到 1 秒的 CPU 时间。可想而知它浪费了多少资源！

记住一点，你不能保证自己的线程一直运行，你阻止不了其他线程的运行。



> **注意** 经常有开发人员问我，怎样保证线程在发生某个事件后的一段时间内开始运行——例如，怎样保证一个线程在网络有数据传来的 1 毫秒内开始运行？对此我的回答总是很干脆：保证不了！
>
> 实时操作系统能做出这样的保证，但 Windows 不是实时操作系统。实时操作系统需要对硬件的运行情况有一个精准的把握，它知道硬盘控制器、键盘以及其他组件的准确延迟时间。但 Microsoft Windows 的设计目标是兼容大范围的硬件，包括不同的 CPU、不同的驱动器、不同的网络等。简单地说，Windows 没有被设计成实时操作系统。补充一句，CLR 使托管代码的行为变得更不“实时”了。之所以要这样设计，是出于许多方面的原因，包括 DLL 的 JIT(just-in-time)加载、代码的 JIT 编译以及垃圾回收器无法预测的介入时间等。

每个线程都分配了从 0(最低)到 31(最高)的优先级。系统决定为 CPU 分配哪个线程时，首先检查优先级 31 的线程，并以一种轮流(round-robin)方式调度它们。如果优先级 31 的一个线程可以调度，就把它分配给 CPU。在这个线程的时间片结束时，系统检查是否有另一个优先级 31 的线程可以运行；如果是，就允许将那个线程分配给 CPU。

只要存在可调度的优先级 31 的线程，系统就永远不会将优先级 0～30 的任何线程分配给 CPU。这种情况称为**饥饿**(starvation)。较高优先级的线程占用了太多 CPU 时间，造成较低优先级的线程无法运行，就会发生这种情况。多处理器机器发生饥饿的可能性要小得多，因为这种机器上优先级为 31 的线程和优先级为 30 的线程可以同时运行。系统总是保持各个 CPU 处于忙碌状态，只有没有线程可调度的时候，CPU 才会空闲下来。

较高优先级的线程总是抢占较低优先级的线程，无论正在运行的是什么较低优先级的线程。例如，如果有一个优先级为 5 的线程在运行，而系统确定有一个较高优先级的线程准备好运行，系统会立即挂起(暂停)较低优先级的线程(即使后者的时间片还没有用完)，将 CPU 分配给较高优先级的线程，该线程将获得一个完整的时间片。

顺便说一下，系统启动时会创建一个特殊的**零页线程**(zero page thread)。该线程的优先级是 0，而且是整个系统唯一优先级为 0 的线程。在没有其他线程需要“干活儿”的时候，零页线程将系统 RAM 的所有空闲页清零。

Microsoft 知道开发人员在为线程分配优先级时很难做到完全合理。这个线程的优先级应该为 10 吗？另一个线程的优先级应该为 23 吗？为了解决这个问题，Windows 公开了优先级系统的一个抽象层。

设计应用程序时，你要决定自己的应用程序需要比机器上同时运行的其他应用程序更大还

是更小的响应能力。然后，选择一个进程**优先级类**(priority class)[①]来反映你的决定。Windows 支持 6 个进程优先级类：Idle，**Below Normal**，**Normal**，**Above Normal**，**High** 和 **Realtime**。默认的 **Normal** 是最常用的优先级类。

在系统什么事情都不做的时候运行的应用程序(比如屏幕保护程序)适合分配 **Idle** 优先级类。注意，即使一台计算机没有被交互地使用[②]，也有可能处于忙的状态(比如作为文件服务器运行)，它不应该和屏幕保护程序竞争 CPU 时间。一些执行统计、跟踪和分析的应用程序只是定期更新与系统有关的状态，它们一般不应妨碍执行更关键的任务。

只有绝对必要的时候才应使用 **High** 优先级类。**Realtime** 优先级类要尽可能地避免。**Realtime** 优先级相当高，它甚至可能干扰操作系统任务，比如阻碍一些必要的磁盘 I/O 和网络传输。除此之外，一个 **Realtime** 进程的线程可能造成不能及时地处理键盘和鼠标输入，用户会感觉“死机”了。简单地说，必须要有很好的理由才能使用 **Realtime** 优先级，比如需要响应延迟(latency)很短的硬件事件，或者要执行一些不能中断的“短命”任务。



注意　为了保持系统总体平稳运行，除非用户有“提高调度优先级”(Increase Scheduling Priority)特权，否则进程不能以 **Realtime** 优先级类运行。管理员和 Power User 默认有这个特权。

选好优先级类之后，就不要再思考你的应用程序和其他应用程序的关系了。现在，应该将所有注意力放在应用程序中的线程上。Windows 支持 7 个相对线程优先级：**Idle**，**Lowest**，**Below Normal**，**Normal**，**Above Normal**，**Highest** 和 **Time-Critical**。这些优先级是*相对*于进程优先级类而言的。同样地，由于 **Normal** 是默认的相对线程优先级，所以是最常用的。

总之，你的进程是一个优先级类的成员。在你的进程中，要为各个线程分配相对优先级。到目前为止，我一直没有提到关于 0～31 的线程优先级的任何事情。应用程序开发人员永远不直接处理这些优先级。相反，系统将进程的优先级类和其中的一个线程的相对优先级映射成一个优先级(0～31)。表 26-1 总结了进程的优先级类和线程的相对优先级与优先级(0～31)的映射关系。

---

① 优先级类和优先级是两个概念。根据定义，每个线程的优先级取决于两个标准：1)它的进程的优先级类；2)在其进程的优先级类中，线程的优先级。优先级类和优先级合并构成了一个线程的“基础优先级”(base priority)。注意，每个线程都有一个动态优先级(dynamic priority)。线程调度器根据这个优先级来决定要执行哪个线程。最初，线程的动态优先级和它的基础优先级是相同的。系统可提升(boost)和降低(lower)动态优先级，以确保它的可响应性，并避免线程在处理器时间内“饥饿”。但是，对于基础优先级 16～31 之间的线程，系统不会提升它们的优先级。只有基础优先级在 0 到 15 之间的线程才会被动态提升(优先级)。——译注

② 所谓交互使用，是指一个用户坐在计算机面前或者以远程连接的方式进行人机交互。——译注

表 26-1 “进程优先级类”和“相对线程优先级”如何映射到“优先级”值

相对线程优先级	进程优先级类					
	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

例如，**Normal** 进程中的一个 **Normal** 线程的优先级是 8。由于大多数进程都是 **Normal** 优先级，大多数线程也是 **Normal** 优先级，所以系统中大多数线程的优先级都是 8。

High 优先级进程中的 **Normal** 线程的优先级是 13。将进程优先级类更改为 **Idle**，线程的优先级变成 4。记住，线程优先级是相对于进程优先级类的。更改一个进程的优先级类，线程的相对优先级不会改变，但它的绝对优先级值会改变。

注意，表中没有值为 0 的线程优先级。这是因为 0 优先级保留给零页线程了，系统不允许其他线程的优先级为 0。而且，以下优先级也不可获得：17, 18, 19, 20, 21, 27, 28, 29 或者 30。以内核模式运行的设备驱动程序才能获得这些优先级；用户模式的应用程序不能。还要注意，**Realtime** 优先级类中的线程优先级不能低于 16。类似地，非 **Realtime** 的优先级类中的线程优先级不能高于 15。

正常情况下，进程根据启动它的进程来分配优先级。大多数进程都由 Windows 资源管理器启动，后者在 **Normal** 优先级类中生成它的所有子进程。托管应用程序不应该表现为拥有它们自己的进程；相反，它们应该*表现*为在一个 AppDomain 中运行。所以，托管应用程序不应该更改它们的进程的优先级类，因为这会影响进程中运行的所有代码。例如，许多 ASP.NET 应用程序都在单个进程中运行，每个应用程序都在它自己的 AppDomain 中。类似的还有 Silverlight 应用程序，它在一个 IE 浏览器进程中运行。还有托管的存储过程，它在 Microsoft SQL Server 进程中运行。



注意 “进程优先级类”的概念容易引起混淆。人们可能以为 Windows 在调度进程。事实上 Windows 永远不会调度进程，它只调度线程。“进程优先级类”是 Microsoft 提出的抽象概念，旨在帮助你理解自己的应用程序和其他正在运行的应用程序的关系，它没有别的用途。



重要提示　最好是降低一个线程的优先级，而不是提升另一个线程的优先级。如果线程要执行长时间的计算限制任务，比如编译代码、拼写检查、电子表格重新计算等，一般应降低该线程的优先级。如果线程要快速响应某个事件，运行短暂时间，再恢复为等待状态，则应提高该线程的优先级。高优先级线程在其生存期的大多数时间里都应处于等待状态，这样才不至于影响系统的总体响应能力。例如，响应用户按 Windows 徽标键的“Windows 资源管理器”线程就是一个高优先级线程。当用户按这个键时，Windows 资源管理器立即抢占其他更低优先级的线程，并显示它的菜单。用户在菜单中上下移动时，Windows 资源管理器的线程会快速响应每一次按键(或鼠标移动)，更新菜单，并停止运行，直到用户继续在菜单中导航。

注意，Windows Store 应用不能创建额外的 AppDomain，不能更改其进程的优先级类，也不能更改其任何线程的优先级。另外，当 Windows Store 应用不再处于前台时，Windows 自动挂起它的所有线程。这是出于两方面的考虑。首先，这样做可防止后台应用从用户当前正在积极交互的应用“窃取”CPU 时间。确保像“轻扫”(swipe)这样的触摸事件能快速和流畅地完成。其次，降低 CPU 利用率有利于省电，PC 充电一次运行得更久。

另一方面，你的应用程序可更改其线程的相对线程优先级，这需要设置 **Thread** 的 **Priority** 属性，向其传递 **ThreadPriority** 枚举类型定义的 5 个值之一：**Lowest**，**BelowNormal**，**Normal**，**AboveNormal** 或者 **Highest**。然而，就像 Windows 为自己保留了优先级 0 和 Realtime 范围一样，CLR 为自己保留了 Idle 和 Time-Critical 优先级。今天的 CLR 还没有以 Idle 优先级运行的线程，但这一点未来可能改变。然而，如第 21 章“托管堆和垃圾回收”讨论的那样，CLR 的终结器线程以 **Time-Critical** 优先级运行。所以，作为托管应用程序的开发人员，你实际只需使用表 26-1 中 5 个加了底纹的相对线程优先级。



重要提示　今天，大多数应用程序都没有利用线程优先级。但但在我设想的理想世界中，CPU 应保持 100%的利用率，一直都在做有用的工作。在这种情况下，为了保证系统响应能力不受影响，线程优先级就显得至关重要。遗憾的是，多年来最终用户已养成了一个习惯：一旦看到太高的 CPU 使用率，就感觉应用程序要失去控制了。在我的新世界中，需要对最终用户进行“知识再普及”，让他们明白高的 CPU 利用率也许是一件好事情——表明计算机正在积极地为用户处理有用的信息。如果所有 CPU 都忙于运行优先级 8 和以上的线程，就真的出问题了。这意味着应用程序在响应最终用户的输入时遇到麻烦。“任务管理器”未来的版本在报告 CPU 利用率时，或许应该将线程优先级考虑在内；诊断有问题的系统时，这种信息是相当有帮助的。

对于桌面应用(非 Windows Store 应用)，我要强调的是 **System.Diagnostics** 命名空间包含 **Process** 类和 **ProcessThread** 类。这两个类分别提供了进程和线程的 Windows 视图。用托管代码写实用程序(utility)，或者建构代码来帮助自己进行调试时，就可以使用这两个类。事实上，这正是为什么这两个类都在 **System.Diagnostics** 命名空间中的原因。应用程序需要以特殊的安全权限运行才能使用这两个类。例如，Silverlight 或 ASP.NET 应用程序便用不了这两个类。

另一方面，应用程序可使用 **AppDomain** 和 **Thread** 类，它们公开了 AppDomain 和线程的 CLR 视图。一般不需要特殊安全权限就可以使用这两个类，虽然某些操作仍需提升权限才可以使用。

26.9 前台线程和后台线程

CLR 将每个线程要么视为前台线程，要么视为后台线程。一个进程的所有前台线程停止运行时，CLR 强制终止仍在运行的任何后台线程。这些后台线程被直接终止；不抛出异常。

所以，应该用前台线程执行确实想完成的任务，比如将数据从内存缓冲区 flush[①]到磁盘。非关键性任务则使用后台线程，比如重新计算电子表格的单元格，或者为记录建立索引等。这是由于这些工作能在应用程序重启时继续，而且如果用户想终止应用程序，就没必要强迫应用程序保持活动。

CLR 需要提供前台和后台线程的概念来更好地支持 AppDomain。我们知道，每个 AppDomain 都可运行一个单独的应用程序，而每个应用程序都有自己的前台线程。如果应用程序退出，造成它的前台线程终止，则 CLR 仍需保持活动并运行，使其他应用程序能继续运行。所有应用程序都退出，它们的所有前台线程都终止后，整个进程就可以被销毁了。

以下代码演示了前台和后台线程之间的差异：

```
using System;
using System.Threading;

public static class Program {
    public static void Main() {
        // 创建新线程(默认为前台线程)
        Thread t = new Thread(Worker);

        // 使线程成为后台线程
        t.IsBackground = true;

        t.Start(); // 启动线程
        // 如果 t 是前台线程，则应用程序大约 10 秒后才终止
        // 如果 t 是后台线程，则应用程序立即终止
        Console.WriteLine("Returning from Main");
    }
```

① flush 在文档中翻译成“刷新”，本书保留原文未译。其实 flush 在技术文档中的意思和日常生活中一样，即“冲(到别处)”。例如，我们会说“冲水”，不会说“刷水”。——译注

```
    private static void Worker() {
        Thread.Sleep(10000); // 模拟做 10 秒钟的工作

        // 下面这行代码只有在由一个前台线程执行时才会显示
        Console.WriteLine("Returning from Worker");
    }
}
```

在线程的生存期中，任何时候都可以从前台变成后台，或者从后台变成前台。应用程序的主线程以及通过构造一个 **Thread** 对象来显式创建的任何线程都默认为前台线程。相反，线程池线程默认为后台线程。另外，由进入托管执行环境的本机(native)代码创建的任何线程都被标记为后台线程。



重要提示　尽量避免使用前台线程。我有次接手了一个顾问工作，有个应用程序就是不终止。花了几小时研究问题后，才发现是一个 UI 组件显式地创建了一个前台线程(默认)，这正是进程一直不终止的原因。后来修改组件使用线程池才解决了问题。执行效率也提升了。

26.10　继续学习

本章讲解了线程的基础知识，我希望我已讲清楚了线程是非常宝贵的资源，必须省着用。为了做到这一点，最好的方式就是使用 CLR 的线程池。线程池自动为你管理线程的创建和销毁。线程池创建的线程将为各种任务而重用，所以你的应用程序其实只需几个线程即可完成全部工作。

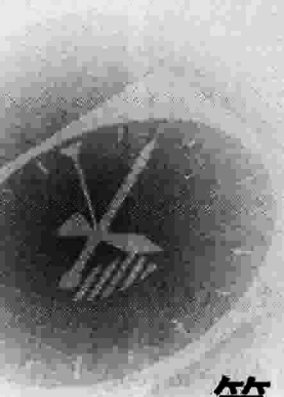
第 27 章重点讲述如何使用 CLR 线程池执行计算限制的操作。第 28 章则要讨论如何使用线程池执行 I/O 限制的操作。许多时候都可以在不需要线程同步的前提下执行异步的计算限制和 I/O 限制操作。但也有的时候必须进行线程同步。第 29 章和第 30 章将讨论线程同步构造的工作方式，以及各种构造之间的区别

结束本章的讨论之前，我想告诉大家，我全面接触并使用线程始于 1992 年 Windows NT 3.1 的第一个 BETA 版本。.NET 开始 BETA 测试时，我开始制作一个类库来简化异步编程和线程同步。这个库称为 Wintellect Power Threading Library，可免费下载和使用。这个库有针对桌面 CLR、Silverlight CLR 和 Compact Framework 的版本。库、文档和示例代码的下载地址是 *http://wintellect.com/Resource-Power-Collections-Library*。网站还包含了到一个支持论坛的链接。还可观看一些视频，它们演示了如何使用库的各个部分。

[illegible]

[illegible]

[illegible]



第 27 章　计算限制的异步操作

本章内容：

- CLR 线程池基础
- 执行简单的计算限制操作
- 执行上下文
- 协作式取消和超时
- 任务
- **Parallel 的静态 For、ForEach 和 Invoke 方法**
- 并行语言集成查询(PLINQ)
- 执行定时的计算限制操作
- 线程池如何管理线程

本章将讨论以异步方式执行操作的各种方式。异步的计算限制操作要用其他线程执行，例子包括编译代码、拼写检查、语法检查、电子表格重计算、音频或视频数据转码以及生成图像的缩略图。在金融和工程应用程序中，计算限制的操作也是十分普遍的。

大多数应用程序都不会花太多时间处理内存数据或执行计算。要验证这一点，可以打开“任务管理器”，选择“性能”标签。如果 CPU 利用率不到 100%(大多数时候都如此)，就表明当前运行的进程没有使用由计算机的 CPU 内核提供的全部计算能力。CPU 利用率低于 100%时，进程中的部分(但不是全部)线程根本没有运行。相反，这些线程正在等待某个输入或输出操作。例如，这些线程可能正在等待一个计时器到期[①]；等待在数据库/Web 服务/文件/网络/其他硬件设备中读取或写入数据；或者等待按键、鼠标移动或鼠标点击等。执行 I/O 限制的操作时，Microsoft Windows 设备驱动程序让硬件设备为你“干活儿”，但 CPU 本身“无所事事”。由于线程不在 CPU 上运行，所以“任务管理器”说 CPU 利用率很低。

但是，即使 I/O 限制非常严重的应用程序也要对接收到的数据执行一些计算，而并行执行这些计算能显著提升应用程序的吞吐能力。本章首先介绍 CLR 的线程池，并解释了和它的工作和使用有关的一些基本概念。这些信息非常重要。为了设计和实现可伸缩的、响应灵敏和可靠的应用程序和组件，线程池是你必须采用的核心技术。然后，本章展示了通过线程池执行计算限制操作的各种机制。

---

① 计时器“到期”(come due)的意思是还有多久触发它。

27.1 CLR 线程池基础

如第 26 章所述，创建和销毁线程是一个昂贵的操作，要耗费大量时间。另外，太多的线程会浪费内存资源。由于操作系统必须调度可运行的线程并执行上下文切换，所以太多的线程还对性能不利。为了改善这个情况，CLR 包含了代码来管理它自己的**线程池**(thread pool)。线程池是你的应用程序能使用的线程集合。每 CLR 一个线程池；这个线程池由 CLR 控制的所有 AppDomain 共享。如果一个进程中加载了多个 CLR，那么每个 CLR 都有它自己的线程池。

CLR 初始化时，线程池中是没有线程的。在内部，线程池维护了一个操作请求队列。应用程序执行一个异步操作时，就调用某个方法，将一个记录项(entry)追加到线程池的队列中。线程池的代码从这个队列中提取记录项，将这个记录项派发(dispatch)给一个线程池线程。如果线程池中没有线程，就创建一个新线程。创建线程会造成一定的性能损失(前面已讨论过了)。然而，当线程池线程完成任务后，线程不会被销毁。相反，线程会返回线程池，在那里进入空闲状态，等待响应另一个请求。由于线程不销毁自身，所以不再产生额外的性能损失。

如果你的应用程序向线程池发出许多请求，线程池会尝试只用这一个线程来服务所有请求。然而，如果你的应用程序发出请求的速度超过了线程池线程处理它们的速度，就会创建额外的线程。最终，你的应用程序的所有请求都能由少量线程处理，所以线程池不必创建大量线程。

如果你的应用程序停止向线程池发出请求，池中会出现大量什么都不做的线程。这是对内存资源的浪费。所以，当一个线程池线程闲着没事儿一段时间之后(不同版本的 CLR 对这个时间的定义不同)，线程会自己醒来终止自己以释放资源。线程终止自己会产生一定的性能损失。然而，线程终止自己是因为它闲得慌，表明应用程序本身就没有做太多的事情，所以这个性能损失关系不大。

线程池可以只容纳少量线程，从而避免浪费资源；也可以容纳更多的线程，以利用多处理器、超线程处理器和多核处理器。它能在这两种不同的状态之间从容地切换。线程池是启发式的。如果应用程序需要执行许多任务，同时有可用的 CPU，那么线程池会创建更多的线程。应用程序负载减轻，线程池线程就终止它们自己。

27.2 执行简单的计算限制操作

要将一个异步的计算限制操作放到线程池的队列中，通常可以调用 **ThreadPool** 类定义的以下方法之一：

```
static Boolean QueueUserWorkItem(WaitCallback callBack);
static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);
```

这些方法向线程池的队列添加一个“工作项”(work item)以及可选的状态数据。然后，所有方法会立即返回。工作项其实就是由 **callBack** 参数标识的一个方法，该方法将由线程池线程调用。可向方法传递一个 **state** 实参(状态数据)。无 **state** 参数的那个版本的 **QueueUserWorkItem** 则向回调方法传递 **null**。最终，池中的某个线程会处理工作项，造成你指定的方法被调用。你写的回调方法必须匹配 **System.Threading.WaitCallback** 委托类型，后者的定义如下：

```
delegate void WaitCallback(Object state);
```



> **注意**　**WaitCallback** 委托、**TimerCallback** 委托(参见本章 27.8 节“执行定时计算限制操作”的讨论)和 **ParameterizedThreadStart** 委托(在第 26 章“线程基础”中讨论)签名完全一致。定义和该签名匹配的方法后，使用 **ThreadPool.QueueUserWorkItem**、**System.Threading.Timer** 和 **System.Threading.Thread** 对象都可调用该方法。

以下代码演示了如何让一个线程池线程以异步方式调用一个方法：

```
using System;
using System.Threading;

public static class Program {
    public static void Main() {
        Console.WriteLine("Main thread: queuing an asynchronous operation");
        ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
        Console.WriteLine("Main thread: Doing other work here...");
        Thread.Sleep(10000); // 模拟其他工作(10 秒)
        Console.WriteLine("Hit <Enter> to end this program...");
        Console.ReadLine();
    }

    // 这个方法的签名必须匹配 WaitCallback 委托
    private static void ComputeBoundOp(Object state) {
        // 这个方法由一个线程池线程执行

        Console.WriteLine("In ComputeBoundOp: state={0}", state);
        Thread.Sleep(1000); // 模拟其他工作(1 秒)

        // 这个方法返回后，线程回到池中，等待另一个任务
    }
}
```

编译并运行上述代码得到以下输出：

```
Main thread: queuing an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

有时也得到以下输出：

```
Main thread: queuing an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

之所以输出行的顺序会发生变化，是因为两个方法相互之间是异步运行的。Windows 调度器决定先调度哪一个线程。如果应用程序在多核机器上运行，可能同时调度它们。



注意　一旦回调方法抛出未处理的异常，CLR 会终止进程(除非宿主强加了它自己的策略)。未处理异常的详情已在第 20 章“异常和状态管理”进行了讨论。



注意　对于 Windows Store 应用，**System.Threading.ThreadPool** 类是没有公开的。但在使用 **System.Threading.Tasks** 命名空间中的类型时，这个类被间接地使用(详情参见本章稍后的 27.5 节“任务”)。

27.3　执行上下文

每个线程都关联了一个执行上下文数据结构。**执行上下文**(execution context)包括的东西有安全设置(压缩栈、**Thread** 的 **Principal** 属性和 Windows 身份)、宿主设置(参见 **System.Threading.HostExecutionContextManager**)以及逻辑调用上下文数据(参见 **System.Runtime.Remoting.Messaging.CallContext** 的 **LogicalSetData 和 LogicalGetData** 方法)。线程执行它的代码时，一些操作会受到线程执行上下文设置(尤其是安全设置)的影响。理想情况下，每当一个线程(初始线程)使用另一个线程(辅助线程)执行任务时，前者的执行上下文应该流向(复制到)辅助线程。这就确保了辅助线程执行的任何操作使用的是相同的安全设置和宿主设置。还确保了在初始线程的逻辑调用上下文中存储的任何数据都适用于辅助线程。

默认情况下，CLR 自动造成初始线程的执行上下文“流向”任何辅助线程。这造成将上下文信息传给辅助线程，但这会对性能造成一定影响。这是因为执行上下文中包含大量信息，而收集所有这些信息，再把它们复制到辅助线程，要耗费不少时间。如果辅助线程又采用了更多的辅助线程，还必须创建和初始化更多的执行上下文数据结构。

System.Threading 命名空间有一个 **ExecutionContext** 类，它允许你控制线程的执行上下文如何从一个线程“流”向另一个。下面展示了这个类的样子：

```
public sealed class ExecutionContext : IDisposable, ISerializable {
    [SecurityCritical] public static AsyncFlowControl SuppressFlow();
    public static void RestoreFlow();
    public static Boolean IsFlowSuppressed();

    // 未列出不常用的方法
}
```

可用这个类阻止执行上下文流动以提升应用程序的性能。对于服务器应用程序，性能的提升可能非常显著。但客户端应用程序的性能提升不了多少。另外，由于 **SuppressFlow** 方法用 **[SecurityCritical]**特性进行了标识，所以在某些客户端应用程序(比如 Silverlight)中是无法调用的。当然，只有在辅助线程不需要或者不访问上下文信息时，才应阻止执行上下文的流动。如果初始线程的执行上下文不流向辅助线程，辅助线程会使用上一次和它关联的任意执行上下文。在这种情况下，辅助线程不应执行任何要依赖于执行上下文状态(比如用

户的 Windows 身份)的代码。

下例展示了向 CLR 的线程池队列添加一个工作项的时候，如何通过阻止执行上下文的流动来影响线程逻辑调用上下文中的数据[①]：

```
public static void Main() {
   // 将一些数据放到 Main 线程的逻辑调用上下文中
   CallContext.LogicalSetData("Name", "Jeffrey");

   // 初始化要由一个线程池线程做的一些工作，
   // 线程池线程能访问逻辑调用上下文数据
   ThreadPool.QueueUserWorkItem(
      state => Console.WriteLine("Name={0}", CallContext.LogicalGetData("Name")));

   // 现在，阻止 Main 线程的执行上下文的流动
  ExecutionContext.SuppressFlow();

   // 初始化要由线程池线程做的工作，
   // 线程池线程不能访问逻辑调用上下文数据
   ThreadPool.QueueUserWorkItem(
      state => Console.WriteLine("Name={0}", CallContext.LogicalGetData("Name")));

   // 恢复 Main 线程的执行上下文的流动，
   // 以免将来使用更多的线程池线程
   ExecutionContext.RestoreFlow();
   ...
   Console.ReadLine();
}
```

编译并运行上述代码得到以下输出：

```
Name=Jeffrey
Name=
```

虽然我们讨论的是在调用 **ThreadPool.QueueUserWorkItem** 时阻止执行上下文的流动，但在使用 **Task** 对象(参见 27.5 节“任务”)时，以及在发起异步 I/O 操作(参见第 28 章“I/O 限制的异步操作”)时，这个技术同样有用。

27.4　协作式取消和超时

Microsoft .NET Framework 提供了标准的**取消操作**模式。这个模式是**协作式**的，意味着要取消的操作必须显式支持**取消**。换言之，无论执行操作的代码，还是试图取消操作的代码，都必须使用本节提到的类型。对于长时间运行的计算限制操作，支持取消是一件很“棒”的事情。所以，你应该考虑为自己的计算限制操作添加取消能力。本节将解释具体如何做。但首先解释一下作为标准协作式取消模式一部分的两个 FCL 类型。

取消操作首先要创建一个 **System.Threading.CancellationTokenSource** 对象。这个类看起来

---

① 添加到逻辑调用上下文的项必须是可序列化的，详情参见第 24 章“运行时序列化”。对于包含了逻辑调用上下文数据项的执行上下文，让它流动起来可能严重损害性能，因为为了捕捉执行上下文，需要对所有数据项进行序列化和反序列化。

像下面这样：

```
public sealed class CancellationTokenSource : IDisposable { // 一个引用类型
    public CancellationTokenSource();
    public void Dispose(); // 释放资源(比如 WaitHandle)

    public Boolean IsCancellationRequested { get; }
    public CancellationToken Token { get; }

    public void Cancel(); // 内部调用 Cancel 并传递 false
    public void Cancel(Boolean throwOnFirstException);
    ...
}
```

这个对象包含了和管理取消有关的所有状态。构造好一个 **CancellationTokenSource**(一个引用类型)之后，可从它的 **Token** 属性获得一个或多个 **CancellationToken**(一个值类型)实例，并传给你的操作，使操作可以取消。以下是 **CancellationToken** 值类型最有用的成员：

```
public struct CancellationToken { // 一个值类型
    public static CancellationToken None { get; } // 很好用

    public Boolean IsCancellationRequested { get; }  // 由非通过 Task 调用的操作调用
    public void    ThrowIfCancellationRequested();   // 由通过 Task 调用的操作调用

    // CancellationTokenSource 取消时，WaitHandle 会收到信号
    public WaitHandle WaitHandle { get; }
    // GetHashCode, Equals, operator== 和 operator!=成员未列出

    public Boolean CanBeCanceled { get; } // 很少使用

    public CancellationTokenRegistration Register(Action<Object> callback, Object state,
        Boolean useSynchronizationContext); // 未列出更简单的重载版本
}
```

CancellationToken 实例是轻量级值类型，包含单个私有字段，即对其 **CancellationTokenSource** 对象的引用。在计算限制操作的循环中，可定时调用 **CancellationToken** 的 **IsCancellationRequested** 属性，了解循环是否应该提前终止，从而终止计算限制的操作。提前终止的好处在于，CPU 不需要再把时间浪费在你对结果不感兴趣的操作上。以下代码将这些概念全部梳理了一遍：

```
internal static class CancellationDemo {
    public static void Go() {
        CancellationTokenSource cts = new CancellationTokenSource();

        // 将 CancellationToken 和“要数到的数”(number-to-count-to)传入操作
        ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));

        Console.WriteLine("Press <Enter> to cancel the operation.");
        Console.ReadLine();
        cts.Cancel(); // 如果 Count 方法已返回，Cancel 没有任何效果
        // Cancel 立即返回，方法从这里继续运行...

        Console.ReadLine();
    }

    private static void Count(CancellationToken token, Int32 countTo) {
        for (Int32 count = 0; count < countTo; count++) {
            if (token.IsCancellationRequested) {
                Console.WriteLine("Count is cancelled");
                break; // 退出循环以停止操作
```

```
            }

            Console.WriteLine(count);
            Thread.Sleep(200); // 出于演示目的而浪费一些时间
        }
        Console.WriteLine("Count is done");
    }
}
```



> **注意**　要执行一个不允许被取消的操作，可向该操作传递通过调用 **CancellationToken** 的静态 **None** 属性而返回的 **CancellationToken**。该属性返回一个特殊的 **CancellationToken** 实例，它不和任何 **CancellationTokenSource** 对象关联(实例的私有字段为 **null**)。由于没有 **CancellationTokenSource**，所以没有代码能调用 **Cancel**。一个操作如果查询这个特殊 **CancellationToken** 的 **IsCancellationRequested** 属性，将总是返回 **false**。使用某个特殊 **CancellationToken** 实例查询 **CancellationToken** 的 **CanBeCanceled** 属性，属性会返回 **false**。相反，对于通过查询 **CancellationTokenSource** 对象的 **Token** 属性而获得的其他所有 **CancellationToken** 实例，该属性(**CanBeCanceled**)都会返回 **true**。

如果愿意，可调用 **CancellationTokenSource** 的 **Register** 方法登记一个或多个在取消一个 **CancellationTokenSource** 时调用的方法。要向方法传递一个 **Action<Object>**委托；一个要通过委托传给回调(方法)的状态值；以及一个 **Boolean** 值(名为 **useSynchronizationContext**)，该值指明是否要使用调用线程的 **SynchronizationContext** 来调用委托。如果为 **useSynchronizationContext** 参数传递 **false**，那么调用 **Cancel** 的线程会顺序调用已登记的所有方法。为 **useSynchronizationContext** 参数传递 **true**，则回调(方法)会被 send(而不是 post[①])给已捕捉的 **SynchronizationContext** 对象，后者决定由哪个线程调用回调(方法)。**SynchronizationContext** 类的详情将在 28.9 节“应用程序及其线程处理模型”讨论。



> **注意**　向被取消的 **CancellationTokenSource** 登记一个回调方法，将由调用 **Register** 的线程调用回调方法(如果为 **useSynchronizationContext** 参数传递了 **true** 值，就可能要通过调用线程的 **SynchronizationContext** 进行)。

多次调用 **Register**，多个回调方法都会调用。这些回调方法可能抛出未处理的异常。如果调用 **CancellationTokenSource** 的 **Cancel** 方法，向它传递 **true**，那么抛出了未处理异常的第一个回调方法会阻止其他回调方法的执行，抛出的异常也会从 **Cancel** 中抛出。如果调用 **Cancel** 并向它传递 **false**，那么登记的所有回调方法都会调用。所有未处理的异常都会添加到一个集合中。所有回调方法都执行好后，其中任何一个抛出了未处理的异常，**Cancel** 就

---

① 简单地说，如果执行 send 操作，要等到在目标线程那里处理完毕之后才会返回。在此期间，调用线程会被阻塞。这相当于同步调用。而如果执行 post 操作，是指将东西 post 到一个队列中便完事儿，调用线程立即返回，相当于异步调用。——译注

会抛出一个 **AggregateException**，该异常实例的 **InnerExceptions** 属性被设为已抛出的所有异常对象的集合。如果登记的所有回调方法都没有抛出未处理的异常，那么 **Cancel** 直接返回，不抛出任何异常。



重要提示 没有办法将 **AggregateException** 的 **InnerExceptions** 集合中的一个异常对象和特的操作对应起来；你只知道某个操作出错，并通过异常类型知道出了什么错。要跟踪错误的具体位置，需要检查异常对象的 **StackTrace** 属性，并手动扫描你的源代码。

CancellationToken 的 **Register** 方法返回一个 **CancellationTokenRegistration**，如下所示：

```
public struct CancellationTokenRegistration :
    IEquatable<CancellationTokenRegistration>, IDisposable {
    public void Dispose();
    // GetHashCode, Equals, operator==和 operator!=成员未列出
}
```

可以调用 **Dispose** 从关联的 **CancellationTokenSource** 中删除已登记的回调；这样一来，在调用 **Cancel** 时，便不会再调用这个回调。以下代码演示了如何向一个 **CancellationTokenSource** 登记两个回调：

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("Canceled 1"));
cts.Token.Register(() => Console.WriteLine("Canceled 2"));

// 出于测试的目的，让我们取消它，以便执行 2 个回调
cts.Cancel();
```

运行上述代码，一旦调用 **Cancel** 方法，就会得到以下输出：

```
Canceled 2
Canceled 1
```

最后，可以通过链接另一组 **CancellationTokenSource** 来新建一个 **CancellationTokenSource** 对象。任何一个链接的 **CancellationTokenSource** 被取消，这个新的 **CancellationTokenSource** 对象就会被取消。以下代码对此进行了演示：

```
// 创建一个 CancellationTokenSource
var cts1 = new CancellationTokenSource();
cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));

// 创建另一个 CancellationTokenSource
var cts2 = new CancellationTokenSource();
cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

// 创建一个新的 CancellationTokenSource，它在 cts1 或 ct2 取消时取消
var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(cts1.Token, cts2.Token);
linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));

// 取消其中一个 CancellationTokenSource 对象(我选择 cts2)
cts2.Cancel();

// 显示哪个 CancellationTokenSource 对象被取消了
Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts={2}",
   cts1.IsCancellationRequested, cts2.IsCancellationRequested,
linkedCts.IsCancellationRequested);
```

运行上述代码得到以下输出：

```
linkedCts canceled
cts2 canceled
cts1 canceled=False, cts2 canceled=True, linkedCts=True
```

在很多情况下，我们需要在过一段时间之后才取消操作。例如，服务器应用程序可能会根据客户端的请求而开始计算。但必须在 2 秒钟之内有响应，无论此时工作是否已经完成。有的时候，与其等待漫长时间获得一个完整的结果，还不如在短时间内报错，或者用部分计算好的结果进行响应。幸好，**CancellationTokenSource** 提供了在指定时间后自动取消的机制。为了利用这个机制，要么用接收延时参数的构造器构造一个 **CancellationTokenSource** 对象，要么调用 **CancellationTokenSource** 的 **CancelAfter** 方法。

```
public sealed class CancellationTokenSource : IDisposable { // 一个引用类型
   public CancellationTokenSource(Int32 millisecondsDelay);
   public CancellationTokenSource(TimeSpan delay);
   public void CancelAfter(Int32 millisecondsDelay);
   public void CancelAfter(TimeSpan delay);
   ...
}
```

27.5　任务

很容易调用 **ThreadPool** 的 **QueueUserWorkItem** 方法发起一次异步的计算限制操作。但这个技术有许多限制。最大的问题是没有内建的机制让你知道操作在什么时候完成，也没有机制在操作完成时获得返回值。为了克服这些限制(并解决其他一些问题)，Microsoft 引入了**任务**的概念。我们通过 **System.Threading.Tasks** 命名空间中的类型来使用任务。

所以，不是调用 **ThreadPool** 的 **QueueUserWorkItem** 方法，而是用任务来做相同的事情：

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);      // 调用 QueueUserWorkItem
new Task(ComputeBoundOp, 5).Start();                  // 用 Task 来做相同的事情
Task.Run(() => ComputeBoundOp(5));                    // 另一个等价的写法
```

第二行代码创建 **Task** 对象并立即调用 **Start** 来调度任务。当然，也可先创建好 **Task** 对象再调用 **Start**。例如，可以创建一个 **Task** 对象，把它传给另一个方法，由后者决定在什么调用 **Start** 来调度任务。由于创建 **Task** 对象并立即调用 **Start** 是常见的编程模式，所以可以像最后一行代码展示的那样调用 **Task** 的静态 **Run** 方法。

为了创建一个 **Task**，需要调用构造器并传递一个 **Action** 或 **Action<Object>**委托。这个委托就是你想执行的操作。如果传递的是期待一个 **Object** 的方法，还必须向 **Task** 的构造器传递最终要传给操作的实参。调用 **Run** 时可以传递一个 **Action** 或 **Func<TResult>**委托来指定想要执行的操作。无论调用构造器还是 **Run**，都可选择传递一个 **CancellationToken**，它使 **Task** 能在调度前取消(详情参见稍后的 27.5.2 节“取消任务”)。

还可选择向构造器传递一些 **TaskCreationOptions** 标志来控制 **Task** 的执行方式。**TaskCreationOptions** 枚举类型定义了一组可按位 OR 的标志。定义如下：

```
[Flags, Serializable]
```

```
public enum TaskCreationOptions {
    None              = 0x0000, // 默认

    // 提议 TaskScheduler 你希望该任务尽快执行
    PreferFairness    = 0x0001,

    // 提议 TaskScheduler 应尽可能地创建线程池线程
    LongRunning       = 0x0002,

    // 该提议总是被采纳：将一个 Task 和它的父 Task 关联(稍后讨论)
    AttachedToParent  = 0x0004,

    // 该提议总是被采纳：如果一个任务试图和这个父任务连接，它就是一个普通任务，而不是子任务
    DenyChildAttach   = 0x0008,

    // 该提议总是被采纳：强迫子任务使用默认调度器而不是父任务的调度器
    HideScheduler     = 0x0010
}
```

有的标志只是“提议”，**TaskScheduler** 在调度一个 **Task** 时，可能会、也可能不会采纳这些提议。不过，**AttachedToParent**，**DenyChildAttach** 和 **HideScheduler** 总是得以采纳，因为它们和 **TaskScheduler** 本身无关。**TaskScheduler** 对象的详情将在 27.5.7 节“任务调度器”讨论。

27.5.1 等待任务完成并获取结果

可等待任务完成并获取结果。例如，以下 **Sum** 方法在 **n** 值很大的时候会执行较长时间：

```
private static Int32 Sum(Int32 n) {
   Int32 sum = 0;
   for (; n > 0; n--)
     checked { sum += n; } // 如果 n 太大，会抛出 System.OverflowException
   return sum;
}
```

现在可以构造一个 **Task<TResult>**对象(派生自 **Task**)，并为泛型 **TResult** 参数传递计算限制操作的返回类型。开始任务之后，可等待它完成并获得结果，如以下代码所示：

```
// 创建一个 Task(现在还没有开始运行)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

// 可以后再启动任务
t.Start();

// 可选择显式等待任务完成
t.Wait(); // 注意：还有一些重载的版本能接受 timeout/CancellationToken 值

// 可获得结果(Result 属性内部会调用 Wait)
Console.WriteLine("The Sum is: " + t.Result); // 一个 Int32 值
```

如果计算限制的任务抛出未处理的异常，异常会被“吞噬”并存储到一个集合中，而线程池线程可以返回到线程池中。调用 **Wait** 方法或者 **Result** 属性时，这些成员会抛出一个 **System.AggregateException** 对象。



重要提示　线程调用 **Wait** 方法时，系统检查线程要等待的 **Task** 是否已开始执行。如果是，调用 **Wait** 的线程会阻塞，直到 **Task** 运行结束为止。但如果 **Task** 还没有开始执行，系统*可能*(取决于 **TaskScheduler**)使用调用 **Wait** 的线程来执行 **Task**。在这种情况下，调用 **Wait** 的线程不会阻塞；它会执行 **Task** 并立即返回。好处在于，没有线程会被阻塞，所以减少了对资源的占用(因为不需要创建一个线程来替代被阻塞的线程)，并提升了性能(因为不需要花时间创建线程，也没有上下文切换)。不好的地方在于，假如线程在调用 **Wait** 前已获得了一个线程同步锁，而 **Task** 试图获取同一个锁，就会造成死锁的线程！

AggregateException 类型封装了异常对象的一个集合(如果父任务生成了多个子任务，而多个子任务都抛出了异常，这个集合便可能包含多个异常)。该类型的 **InnerExceptions** 属性返回一个 **ReadOnlyCollection<Exception>** 对象。不要混淆 **InnerExceptions** 属性和 **InnerException** 属性，后者是 **AggregateException** 类从 **System.Exception** 基类继承的。在本例中，**AggregateException** 的 **InnerExceptions** 属性的元素 **0** 将引用由计算限制方法(**Sum**)抛出的实际 **System.OverflowException** 对象。

为方便编码，**AggregateException** 重写了 **Exception** 的 **GetBaseException** 方法，返回作为问题根源的最内层的 **AggregateException**(假定集合只有一个最内层的异常)。**AggregateException** 还提供了一个 **Flatten** 方法，它创建一个新的 **AggregateException**，其 **InnerExceptions** 属性包含一个异常列表，其中的异常是通过遍历原始 **AggregateException** 的内层异常层次结构而生成的。最后，**AggregateException** 还提供了一个 **Handle** 方法，它为 **AggregateException** 中包含的每个异常都调用一个回调方法。然后，回调方法可以为每个异常决定如何对其进行处理；回调返回 **true** 表示异常已处理；返回 **false** 表示未处理。调用 **Handle** 后，如果至少有一个异常没有处理，就创建一个新的 **AggregateException** 对象，其中只包含未处理的异常，并抛出这个新的 **AggregateException** 对象。本章以后会展示使用了 **Flatten** 和 **Handle** 方法的例子。



重要提示　如果一直不调用 **Wait** 或 **Result**，或者一直不查询 **Task** 的 **Exception** 属性，代码就一直注意不到这个异常的发生。这当然不好，因为程序遇到了未预料到的问题，而你居然没注意到。为了帮助你检测没有被注意到的异常，可以向 **TaskScheduler** 的静态 **UnobservedTaskException** 事件登记一个回调方法。每次当一个 **Task** 被垃圾回收时，如果存在一个没有被注意到的异常，CLR 的终结器线程就会引发这个事件。一旦引发，就会向你的事件处理方法传递一个 **UnobservedTaskExceptionEventArgs** 对象，其中包含你没有注意到的 **AggregateException**。

除了等待单个任务，**Task** 类还提供了两个静态方法，允许线程等待一个 **Task** 对象数组。

其中，**Task** 的静态 **WaitAny** 方法会阻塞调用线程，直到数组中的任何 **Task** 对象完成。方法返回 **Int32** 数组索引值，指明完成的是哪个 **Task** 对象。方法返回后，线程被唤醒并继续运行。如果发生超时，方法将返回**-1**。如果 **WaitAny** 通过一个 **CancellationToken** 取消，会抛出一个 **OperationCanceledException**。

类似地，**Task** 类还有一个静态 **WaitAll** 方法，它阻塞调用线程，直到数组中的所有 **Task** 对象完成。如果所有 **Task** 对象都完成，**WaitAll** 方法返回 **true**。发生超时则返回 **false**。如果 **WaitAll** 通过一个 **CancellationToken** 取消，会抛出一个 **OperationCanceledException**。

27.5.2 取消任务

可用一个 **CancellationTokenSource** 取消 **Task**。首先必须修订前面的 **Sum** 方法，让它接受一个 **CancellationToken**：

```
private static Int32 Sum(CancellationToken ct, Int32 n) {
    Int32 sum = 0;
    for (; n > 0; n--) {

        // 在取消标志引用的 CancellationTokenSource 上调用 Cancel，
        // 下面这行代码就会抛出 OperationCanceledException
        ct.ThrowIfCancellationRequested();

        checked { sum += n; } // 如果 n 太大，会抛出 System.OverflowException
    }
    return sum;
}
```

循环(负责执行计算限制的操作)中调用 **CancellationToken** 的 **ThrowIfCancellationRequested** 方法定时检查操作是否已取消。这个方法与 **CancellationToken** 的 **IsCancellationRequested** 属性相似(27.4 节“协作式取消和超时”已经讨论过这个属性)。但如果 **CancellationTokenSource** 已经取消，**ThrowIfCancellationRequested** 会抛出一个 **OperationCanceledException**。之所以选择抛出异常，是因为和 **ThreadPool** 的 **QueueUserWorkItem** 方法初始化的工作项不同，任务有办法表示完成，任务甚至能返回一个值。所以，需要采取一种方式将已完成的任务和出错的任务区分开。而让任务抛出异常，就可以知道任务没有一直运行到结束。

现在像下面这样创建 **CancellationTokenSource** 和 **Task** 对象：

```
CancellationTokenSource cts = new CancellationTokenSource();
Task<Int32> t = Task.Run(() => Sum(cts.Token, 1000000000), cts.Token);

// 在之后的某个时间，取消 CancellationTokenSource 以取消 Task
cts.Cancel(); // 这是异步请求，Task 可能已经完成了

try {
   // 如果任务已取消，Result 会抛出一个 AggregateException
   Console.WriteLine("The sum is: " + t.Result); // 一个 Int32 值
}
catch (AggregateException x) {
   // 将任何 OperationCanceledException 对象都视为已处理。
   // 其他任何异常都造成抛出一个新的 AggregateException，
   // 其中只包含未处理的异常
   x.Handle(e => e is OperationCanceledException);
```

```
   // 所有异常都处理好之后，执行下面这一行
   Console.WriteLine("Sum was canceled");
}
```

可在创建 **Task** 时将一个 **CancellationToken** 传给构造器(如上例所示)，从而将两者关联。如果 **CancellationToken** 在 **Task** 调度前取消，**Task** 会被取消，永远都不执行[①]。但如果 **Task** 已调度(通过调用 **Start** 方法[②])，那么 **Task** 的代码只有显式支持取消，其操作才能在执行期间取消。遗憾的是，虽然 **Task** 对象关联了一个 **CancellationToken**，但却没有办法访问它。因此，必须在 **Task** 的代码中获得创建 **Task** 对象时的同一个 **CancellationToken**。为此，最简单的办法就是使用一个 lambda 表达式，将 **CancellationToken** 作为闭包变量“传递”(就像上例那样)。

27.5.3　任务完成时自动启动新任务

伸缩性好的软件不应该使线程阻塞。调用 **Wait**，或者在任务尚未完成时查询任务的 **Result** 属性[③]，极有可能造成线程池创建新线程，这增大了资源的消耗，也不利于性能和伸缩性。幸好，有更好的办法可以知道一个任务在什么时候结束运行。任务完成时可启动另一个任务。下面重写了之前的代码，它不阻塞任何线程：

```
// 创建并启动一个 Task，继续另一个任务
Task<Int32> t = Task.Run(() => Sum(CancellationToken.None, 10000));

// ContinueWith 返回一个 Task，但一般都不需要再使用该对象(下例的 cwt)
Task cwt = t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result));
```

现在，执行 **Sum** 的任务完成时会启动另一个任务(也在某个线程池线程上)以显示结果。执行上述代码的线程不会进入阻塞状态并等待这两个任务中的任何一个完成。相反，线程可以执行其他代码。如果线程本身就是一个线程池线程，它可以返回池中以执行其他操作。注意，执行 **Sum** 的任务可能在调用 **ContinueWith** 之前完成。但这不是一个问题，因为 **ContinueWith** 方法会看到 **Sum** 任务已经完成，会立即启动显示结果的任务。

注意，**ContinueWith** 返回对新 **Task** 对象的引用(我的代码是将该引用放到 **cwt** 变量中)。当然，可以用这个 **Task** 对象调用各种成员(比如 **Wait**，**Result**，甚至 **ContinueWith**)，但一般都忽略这个 **Task** 对象，不再用变量保存对它的引用。

另外，**Task** 对象内部包含了 **ContinueWith** 任务的一个集合。所以，实际可以用一个 **Task** 对象来多次调用 **ContinueWith**。任务完成时，所有 **ContinueWith** 任务都会进入线程池的队列中。此外，可在调用 **ContinueWith** 时传递对一组 **TaskContinuationOptions** 枚举值进行按位 OR 运算的结果。前 6 个标志(**None**，**PreferFairness**，**LongRunning**，**AttachedToParent**，**DenyChildAttach** 和 **HideScheduler**)与之前描述的 **TaskCreationOptions** 枚举类型提供的标

---

① 顺便说一句，如果一个任务还没有开始就试图取消它，会抛出一个 **InvalidOperationException**。

② 调用静态 **Run** 方法会自动创建 **Task** 对象并立即调用 **Start**。——译注

③ **Result** 属性内部会调用 **Wait**。——译注

志完全一致。下面是 **TaskContinuationOptions** 类型的定义：

```
[Flags, Serializable]
public enum TaskContinuationOptions {
    None                    = 0x0000, // 默认

    // 提议 TaskScheduler 你希望该任务尽快执行，
    PreferFairness          = 0x0001,
    // 提议 TaskScheduler 应尽可能地创建线程池线程
    LongRunning             = 0x0002,

    // 该提议总是被采纳：将一个 Task 和它的父 Task 关联(稍后讨论)
    AttachedToParent        = 0x0004,

    // 任务试图和这个父任务连接将抛出一个 InvalidOperationException
    DenyChildAttach         = 0x0008,

    // 强迫子任务使用默认调度器而不是父任务的调度器
    HideScheduler           = 0x0010

    // 除非前置任务(antecedent task)完成，否则禁止延续任务完成(取消)
    LazyCancellation        = 0x0020

    // 这个标志指出你希望由执行第一个任务的线程执行
    // ContinueWith 任务。第一个任务完成后，调用
    // ContinueWith 的线程接着执行 ContinueWith 任务①
    ExecuteSynchronously    = 0x80000,

    // 这些标志指出在什么情况下运行 ContinueWith 任务
    NotOnRanToCompletion    = 0x10000,
    NotOnFaulted            = 0x20000,
    NotOnCanceled           = 0x40000,

    // 这些标志是以上三个标志的便利组合
    OnlyOnCanceled          = NotOnRanToCompletion | NotOnFaulted,
    OnlyOnFaulted           = NotOnRanToCompletion | NotOnCanceled,
    OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
}
```

调用 **ContinueWith** 时，可用 **TaskContinuationOptions.OnlyOnCanceled** 标志指定新任务只有在第一个任务被取消时才执行。类似地，**TaskContinuationOptions.OnlyOnFaulted** 标志指定新任务只有在第一个任务抛出未处理的异常时才执行。当然，还可使用 **TaskContinuationOptions.OnlyOnRanToCompletion** 标志指定新任务只有在第一个任务顺利完成(中途没有取消，也没有抛出未处理异常)时才执行。默认情况下，如果不指定上述任何标志，则新任务无论如何都会运行，不管第一个任务如何完成。一个 **Task** 完成时，它的所有未运行的延续任务都被自动取消②。下面用一个例子来演示所有这些概念。

```
// 创建并启动一个 Task，它有多个延续任务
Task<Int32> t = Task.Run(() => Sum(10000));

// 每个 ContinueWith 都返回一个 Task，但这些 Task 一般都用不着了
t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result),
   TaskContinuationOptions.OnlyOnRanToCompletion);
```

---

① **ExecuteSynchronously** 是指同步执行。两个任务都在使用同一个线程一前一后地执行，就称为同步执行。——译注

② 未运行是因为不满足前面说的各种条件。——译注

```
t.ContinueWith(task => Console.WriteLine("Sum threw: " + task.Exception.InnerException),
    TaskContinuationOptions.OnlyOnFaulted);

t.ContinueWith(task => Console.WriteLine("Sum was canceled"),
    TaskContinuationOptions.OnlyOnCanceled);
```

27.5.4　任务可以启动子任务

最后，任务支持父/子关系，如以下代码所示：

```
Task<Int32[]> parent = new Task<Int32[]>(() => {
    var results = new Int32[3]; // 创建一个数组来存储结果

    // 这个任务创建并启动 3 个子任务
    new Task(() => results[0] = Sum(10000), TaskCreationOptions.AttachedToParent).Start();
    new Task(() => results[1] = Sum(20000), TaskCreationOptions.AttachedToParent).Start();
    new Task(() => results[2] = Sum(30000), TaskCreationOptions.AttachedToParent).Start();

    // 返回对数组的引用(即使数组元素可能还没有初始化)
    return results;
});

// 父任务及其子任务运行完成后，用一个延续任务显示结果
var cwt = parent.ContinueWith(
    parentTask => Array.ForEach(parentTask.Result, Console.WriteLine));

// 启动父任务，便于它启动它的子任务
parent.Start();
```

在本例中，父任务创建并启动三个 **Task** 对象。一个任务创建的一个或多个 **Task** 对象默认是顶级任务，它们与创建它们的任务无关。但 **TaskCreationOptions.AttachedToParent** 标志将一个 **Task** 和创建它的 **Task** 关联，结果是除非所有子任务(以及子任务的子任务)结束运行，否则创建任务(父任务)不认为已经结束。调用 **ContinueWith** 方法创建 **Task** 时，可指定 **TaskContinuationOptions.AttachedToParent** 标志将延续任务指定成子任务。

27.5.5　任务内部揭秘

每个 **Task** 对象都有一组字段，这些字段构成了任务的状态。其中包括一个 **Int32** ID(参见 **Task** 的只读 **Id** 属性)、代表 **Task** 执行状态的一个 **Int32**、对父任务的引用、对 **Task** 创建时指定的 **TaskScheduler** 的引用、对回调方法的引用、对要传给回调方法的对象的引用(可通过 **Task** 的只读 **AsyncState** 属性查询)、对 **ExecutionContext** 的引用以及对 **ManualResetEventSlim** 对象的引用。另外，每个 **Task** 对象都有对根据需要创建的补充状态的引用。补充状态包含一个 **CancellationToken**、一个 **ContinueWithTask** 对象集合、为抛出未处理异常的子任务而准备的一个 **Task** 对象集合等。说了这么多，重点在于虽然任务很有用，但它并不是没有代价的。必须为所有这些状态分配内存。如果不需要任务的附加功能，那么使用 **ThreadPool.QueueUserWorkItem** 能获得更好的资源利用率。

Task 和 **Task<TResult>**类实现了 **IDisposable** 接口，允许在用完 **Task** 对象后调用 **Dispose**。如今，所有 **Dispose** 方法所做的都是关闭 **ManualResetEventSlim** 对象。但可定义从 **Task** 和 **Task<TResult>**派生的类，在这些类中分配它们自己的资源，并在它们重写的 **Dispose**

方法中释放这些资源。我建议不要在代码中为 **Task** 对象显式调用 **Dispose**；相反，应该让垃圾回收器自己清理任何不再需要的资源。

每个 **Task** 对象都包含代表 **Task** 唯一 ID 的 **Int32** 字段。创建 **Task** 对象时该字段初始化为零。首次查询 **Task** 的只读 **Id** 属性时，属性将一个唯一的 **Int32** 值分配给该字段，并返回该值。任务 ID 从 1 开始，每分配一个 ID 都递增 1。在 Microsoft Visual Studio 调试器中查看 **Task** 对象，会造成调试器显示 **Task** 的 ID，从而造成为 **Task** 分配 ID。

该 ID 的意义在于每个 **Task** 都可用唯一值进行标识。事实上，Visual Studio 会在“并行任务”和“并行堆栈”窗口中显示这些任务 ID。但由于不能在自己的代码中分配 ID，所以几乎不可能将 ID 和代码正在做的事情联系起来。运行任务的代码时，可查询 **Task** 的静态 **CurrentId** 属性来返回一个可空 **Int32(Int32?)**。调试期间，可在 Visual Studio 的“监视”或“即时”窗口中调用它，获得当前正在调试的代码的 ID。然后，可在“并行任务”或“并行堆栈”窗口中找到自己的任务。如果当前没有任务正在执行，查询 **CurrentId** 属性会返回 **null**。

在一个 **Task** 对象的存在期间，可查询 **Task** 的只读 **Status** 属性了解它在其生存期的什么位置。该属性返回一个 **TaskStatus** 值，定义如下：

```
public enum TaskStatus {
    // 这些标志指出一个 Task 在其生命期内的状态
    Created,                // 任务已显式创建；可以手动 Start()这个任务
    WaitingForActivation, // 任务已隐式创建；会自动开始

    WaitingToRun,           // 任务已调度，但尚未运行
    Running,                // 任务正在运行

    // 任务正在等待它的子任务完成，子任务完成后它才完成
    WaitingForChildrenToComplete,

    // 任务的最终状态是以下三个之一：
    RanToCompletion,
    Canceled,
    Faulted
}
```

首次构造 **Task** 对象时，它的状态是 **Created**。以后，当任务启动时，它的状态变成 **WaitingToRun**。**Task** 实际在一个线程上运行时，它的状态变成 **Running**。任务停止运行，并等待它的任何子任务时，状态变成 **WaitingForChildrenToComplete**。任务完成时进入以下状态之一：**RanToCompletion**(运行完成)，**Canceled**(取消)或 **Faulted**(出错)。如果运行完成，可通过 **Task<TResult>**的 **Result** 属性来查询任务结果。**Task** 或 **Task<TResult>**出错时，可查询 **Task** 的 **Exception** 属性来获得任务抛出的未处理异常；该属性总是返回一个 **AggregateException** 对象，对象的 **InnerExceptions** 集合包含了所有未处理的异常。

为简化编码，**Task** 提供了几个只读 **Boolean** 属性，包括 **IsCanceled**，**IsFaulted** 和 **IsCompleted**。注意当 **Task** 处于 **RanToCompletion**，**Canceled** 或 **Faulted** 状态时，**IsCompleted** 返回 **true**。判断一个 **Task** 是否成功完成最简单的办法是使用如下所示的代码：

```
if (task.Status == TaskStatus.RanToCompletion) ...
```

调用 **ContinueWith**，**ContinueWhenAll**，**ContinueWhenAny** 或 **FromAsync** 等方法来创建的 **Task** 对象处于 **WaitingForActivation** 状态。通过构造 **TaskCompletionSource<TResult>** 对象来创建的 **Task** 也处于 **WaitingForActivation** 状态。该状态意味着 **Task** 的调度由任务基础结构控制。例如，不可显式启动通过调用 **ContinueWith** 来创建的对象. 该 **Task** 在它的前置任务(antecedent task)执行完毕后自动启动。

27.5.6　任务工厂

有时需要创建一组共享相同配置的 **Task** 对象。为避免机械地将相同的参数传给每个 **Task** 的构造器，可创建一个任务工厂来封装通用的配置。**System.Threading.Tasks** 命名空间定义了一个 **TaskFactory** 类型和一个 **TaskFactory<TResult>**类型。两个类型都派生自 **System.Object**；也就是说，它们是平级的。

要创建一组返回 **void** 的任务，就构造一个 **TaskFactory**；要创建一组具有特定返回类型的任务，就构造一个 **TaskFactory<TResult>**，并通过泛型 **TResult** 实参传递任务的返回类型。创建上述任何工厂类时，要向构造器传递工厂创建的所有任务都具有的默认值。具体地说，要向任务工厂传递希望任务具有的 **CancellationToken**，**TaskScheduler**，**TaskCreationOptions** 和 **TaskContinuationOptions** 设置。

以下代码演示了如何使用一个 **TaskFactory**：

```
Task parent = new Task(() => {
    var cts = new CancellationTokenSource();
    var tf = new TaskFactory<Int32>(
        cts.Token,
        TaskCreationOptions.AttachedToParent,
        TaskContinuationOptions.ExecuteSynchronously,
        TaskScheduler.Default);

    // 这个任务创建并启动 3 个子任务
    var childTasks = new[] {
        tf.StartNew(() => Sum(cts.Token, 10000)),
        tf.StartNew(() => Sum(cts.Token, 20000)),
        tf.StartNew(() => Sum(cts.Token, Int32.MaxValue)) // 太大，抛出 OverflowException
    };

    // 任何子任务抛出异常，就取消其余子任务
    for (Int32 task = 0; task < childTasks.Length; task++)
        childTasks[task].ContinueWith(
            t => cts.Cancel(), TaskContinuationOptions.OnlyOnFaulted);

    // 所有子任务完成后，从未出错/未取消的任务获取返回的最大值，
    // 然后将最大值传给另一个任务来显示最大结果
    tf.ContinueWhenAll(
        childTasks,
        completedTasks => completedTasks.Where(
            t => !t.IsFaulted && !t.IsCanceled).Max(t => t.Result),
        CancellationToken.None)
            .ContinueWith(t =>Console.WriteLine("The maximum is: " + t.Result),
                TaskContinuationOptions.ExecuteSynchronously);
});

// 子任务完成后，也显示任何未处理的异常
```

```
parent.ContinueWith(p => {
    // 我将所有文本放到一个 StringBuilder 中，并只调用 Console.WriteLine 一次，
    // 因为这个任务可能和上面的任务并行执行，而我不希望任务的输出变得不连续
    StringBuilder sb = new StringBuilder(
        "The following exception(s) occurred:" + Environment.NewLine);

    foreach (var e in p.Exception.Flatten().InnerExceptions)
        sb.AppendLine(" "+ e.GetType().ToString());
    Console.WriteLine(sb.ToString());
}, TaskContinuationOptions.OnlyOnFaulted);

// 启动父任务，使它能启动子任务
parent.Start();
```

我通过上述代码创建了一个 **TaskFactory<Int32>**对象。该任务工厂创建 3 个 **Task** 对象。我希望所有子任务都以相同方式配置：每个 **Task** 对象都共享相同的 **CancellationTokenSource** 标记，任务都被视为其父任务的子任务，**TaskFactory** 创建的所有延续任务都以同步方式执行，而且 **TaskFactory** 创建的所有 **Task** 对象都使用默认 **TaskScheduler**。

然后创建包含 3 个子任务对象的数组，所有子任务都调用 **TaskFactory** 的 **StartNew** 方法来创建。使用该方法可以很方便地创建并启动子任务。我通过一个循环告诉每个子任务，如果抛出未处理的异常，就取消其他仍在运行的所有子任务。最后，我在 **TaskFactory** 上调用 **ContinueWhenAll**，它创建在所有子任务都完成后启动的一个 **Task**。由于这个任务是用 **TaskFactory** 创建的，所以它仍被视为父任务的一个子任务，会用默认 **TaskScheduler** 同步执行。但我希望即使其他子任务被取消，也要运行这个任务，所以我通过传递 **CancellationToken.None** 来覆盖 **TaskFactory** 的 **CancellationToken**。这使该任务不能取消。最后，当处理所有结果的任务完成后，我创建另一个任务来显示从所有子任务中返回的最大值。



> **注意** 调用 **TaskFactory** 或 **TaskFactory<TResult>**的静态 **ContinueWhenAll** 和 **ContinueWhenAny** 方法时，以下 **TaskContinuationOption** 标志是非法的：**NotOnRanToCompletion**、**NotOnFaulted** 和 **NotOnCanceled**。当然，基于这些标志组合起来的标志(**OnlyOnCanceled**、**OnlyOnFaulted** 和 **OnlyOnRanToCompletion**)也是非法的。也就是说，无论前置任务是如何完成的，**ContinueWhenAll** 和 **ContinueWhenAny** 都会执行延续任务。

27.5.7 任务调度器

任务基础结构非常灵活，其中 **TaskScheduler** 对象功不可没。**TaskScheduler** 对象负责执行被调度的任务，同时向 Visual Studio 调试器公开任务信息。FCL 提供了两个派生自 **TaskScheduler** 的类型：线程池任务调度器(thread pool task scheduler)，和同步上下文任务调度器(synchronization context task scheduler)。默认情况下，所有应用程序使用的都是线程池任务调度器。这个任务调度器将任务调度给线程池的工作者线程，将在本章后面的 27.9

节“线程池如何管理线程”进行更详细的讨论。可查询 **TaskScheduler** 的静态 **Default** 属性来获得对默认任务调度器的引用。

同步上下文任务调度器适合提供了图形用户界面的应用程序，例如 Windows 窗体、Windows Presentation Foundation(WPF)、Silverlight 和 Windows Store 应用程序。它将所有任务都调度给应用程序的 GUI 线程，使所有任务代码都能成功更新 UI 组件(按钮、菜单项等)。该调度器不使用线程池。可执行 **TaskScheduler** 的静态 **FromCurrentSynchronizationContext** 方法来获得对同步上下文任务调度器的引用。

下面这个简单的 Windows 窗体应用程序演示了如何使用同步上下文任务调度器。

```
internal sealed class MyForm : Form {
    private readonly TaskScheduler m_syncContextTaskScheduler;
    public MyForm() {
        // 获得对一个同步上下文任务调度器的引用
        m_syncContextTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext();

        Text = "Synchronization Context Task Scheduler Demo";
        Visible = true; Width = 400; Height = 100;
    }


    private CancellationTokenSource m_cts;

    protected override void OnMouseClick(MouseEventArgs e) {
        if (m_cts != null) { // 一个操作正在进行，取消它
            m_cts.Cancel();
            m_cts = null;
        } else { // 操作没有开始，启动它
            Text = "Operation running";
            m_cts = new CancellationTokenSource();

            // 这个任务使用默认任务调度器，在一个线程池线程上执行
            Task<Int32> t = Task.Run(() => Sum(m_cts.Token, 20000), m_cts.Token);

            // 这些任务使用同步上下文任务调度器，在 GUI 线程上执行
            t.ContinueWith(task => Text = "Result: " + task.Result,
                CancellationToken.None, TaskContinuationOptions.OnlyOnRanToCompletion,
                m_syncContextTaskScheduler);

            t.ContinueWith(task => Text = "Operation canceled",
            CancellationToken.None, TaskContinuationOptions.OnlyOnCanceled,
            m_syncContextTaskScheduler);

            t.ContinueWith(task => Text = "Operation faulted",
                CancellationToken.None, TaskContinuationOptions.OnlyOnFaulted,
                m_syncContextTaskScheduler);
        }
        base.OnMouseClick(e);
    }
}
```

单击窗体的客户区域，就会在一个线程池线程上启动一个计算限制的任务。使用线程池线程很好，因为 GUI 线程在此期间不会被阻塞，能响应其他 UI 操作。但线程池线程执行的代码不应尝试更新 UI 组件，否则会抛出 **InvalidOperationException**。

计算限制的任务完成后执行三个延续任务之一。它们由与 GUI 线程对应的同步上下文任务调度器来调度。任务调度器将任务放到 GUI 线程的队列中，使它们的代码能成功更新 UI 组件。所有任务都通过继承的 **Text** 属性来更新窗体的标题。

由于计算限制的工作(**Sum**)在线程池线程上运行，所以用户可以和 UI 交互来取消操作。在这个简单的例子中，我允许用户在操作进行期间单击窗体的客户区域来取消操作。

当然，如果有特殊的任务调度需求，完全可以定义自己的 **TaskScheduler** 派生类。Microsoft 在 Parallel Extensions Extras 包中提供了大量和任务有关的示例代码，其中包括多个任务调度器源码，下载地址是 *http://code.msdn.microsoft.com/ParExtSamples*。下面是这个包提供的一部分任务调度器。

- IOTaskScheduler
  这个任务调度器将任务排队给线程池的 I/O 线程而不是工作者线程。

- LimitedConcurrencyLevelTaskScheduler
  这个任务调度器不允许超过 *n*(一个构造器参数)个任务同时执行

- OrderedTaskScheduler
  这个任务调度器一次只允许一个任务执行。这个类派生自 **LimitedConcurrencyLevelTaskScheduler**，为 *n* 传递 1。

- PrioritizingTaskScheduler
  这个任务调度器将任务送入 CLR 线程池队列。之后，可调用 **Prioritize** 指出一个 **Task** 应该在所有普通任务之前处理(如果它还没有处理的话)。可以调用 **Deprioritize** 使一个 **Task** 在所有普通任务之后处理。

- ThreadPerTaskScheduler
  这个任务调度器为每个任务创建并启动一个单独的线程；它完全不使用线程池。

27.6 Parallel 的静态 For，ForEach 和 Invoke 方法

一些常见的编程情形可通过任务提升性能。为简化编程，静态 **System.Threading.Tasks.Parallel** 类封装了这些情形，它内部使用 **Task** 对象。例如，不要像下面这样处理集合中的所有项：

```
// 一个线程顺序执行这个工作(每次迭代调用一次 DoWork)
for (Int32 i = 0; i < 1000; i++) DoWork(i);
```

而要使用 **Parallel** 类的 **For** 方法，用多个线程池线程辅助完成工作：

```
// 线程池的线程并行处理工作
Parallel.For(0, 1000, i => DoWork(i));
```

类似地，如果有一个集合，那么不要像这样写：

```
// 一个线程顺序执行这个工作(每次迭代调用一次 DoWork)
foreach (var item in collection) DoWork(item);
```

而要像这样写：

```
// 线程池的线程并行处理工作
Parallel.ForEach(collection, item => DoWork(item));
```

如果既可以使用 **For**，也可以使用 **ForEach**，那么建议使用 **For**，因为它执行得更快。

最后，如果要执行多个方法，那么既可像下面这样顺序执行：

```
// 一个线程顺序执行所有方法
Method1();
Method2();
Method3();
```

也可并行执行，如下所示：

```
// 线程池的线程并行执行方法
Parallel.Invoke(
    () => Method1(),
    () => Method2(),
    () => Method3());
```

Parallel 的所有方法都让调用线程参与处理。从资源利用的角度说，这是一件好事，因为我们不希望调用线程停下来(阻塞)，等线程池线程做完所有工作才能继续。然而，如果调用线程在线程池线程完成自己的那一部分工作之前完成工作，调用线程会将自己挂起，直到所有工作完成。这也是一件好事，因为这提供了和使用普通 **for** 或 **foreach** 循环时相同的语义：线程要在所有工作完成后才继续运行。还要注意，如果任何操作抛出未处理的异常，你调用的 **Parallel** 方法最后会抛出一个 **AggregateException**。

但这并不是说需要对自己的源代码进行全文替换，将 **for** 循环替换成对 **Parallel.For** 的调用，将 **foreach** 循环替换成对 **Parallel.ForEach** 的调用。调用 **Parallel** 的方法时有一个很重要的前提条件：工作项必须能并行执行！所以，如果工作必须顺序执行，就不要使用 **Parallel** 的方法。另外，要避免会修改任何共享数据的工作项，否则多个线程同时处理可能会损坏数据。解决这个问题一般的办法是围绕数据访问添加线程同步锁。但这样一次就只能有一个线程访问数据，无法享受并行处理多个项所带来的好处。

另外，**Parallel** 的方法本身也有开销：委托对象必须分配，而针对每个工作项都要调用一次这些委托。如果有大量可由多个线程处理的工作项，那么也许能获得性能的提升。另外，如果每一项都涉及大量工作，那么通过委托来调用所产生的性能损失是可以忽略不计的。但如果只为区区几个工作项使用 **Parallel** 的方法，或者为处理得非常快的工作项使用 **Parallel** 的方法，就会得不偿失，反而降低性能。

注意 **Parallel** 的 **For**，**ForEach** 和 **Invoke** 方法都提供了接受一个 **ParallelOptions** 对象的重载版本。这个对象的定义如下：

```
public class ParallelOptions{
    public ParallelOptions();

    // 允许取消操作
    public CancellationToken CancellationToken { get; set; } // 默认为CancellationToken.None

    // 允许指定可以并发操作的最大工作项数目
```

```
    public Int32MaxDegreeOfParallelism { get; set; } // 默认为-1 (可用CPU数)

    // 允许指定要使用哪个TaskScheduler
    public TaskSchedulerTaskScheduler { get; set; } // 默认为TaskScheduler.Default
}
```

除此之外，**For** 和 **ForEach** 方法有一些重载版本允许传递 3 个委托。

- 任务局部初始化委托(localInit)，为参与工作的每个任务都调用一次该委托。这个委托是在任务被要求处理一个工作项之前调用的。
- 主体委托(body)，为参与工作的各个线程所处理的每一项都调用一次该委托。
- 任务局部终结委托(localFinally)，为参与工作的每一个任务都调用一次该委托。这个委托是在任务处理好派发给它的所有工作项之后调用的。即使主体委托代码引发一个未处理的异常，也会调用它。

以下代码演示如何利用这三个委托计算一个目录中的所有文件的字节长度。

```
private static Int64 DirectoryBytes(String path, String searchPattern,
    SearchOption searchOption) {
    var files = Directory.EnumerateFiles(path, searchPattern, searchOption);
    Int64 masterTotal = 0;

    ParallelLoopResult result = Parallel.ForEach<String, Int64>(
        files,

        () => { // localInit: 每个任务开始之前调用一次
            // 每个任务开始之前，总计值都初始化为0
            return 0; // 将taskLocalTotal初始值设为0
        },

        (file, loopState, index, taskLocalTotal) => { // body: 每个工作项调用一次
            // 获得这个文件的大小，把它添加到这个任务的累加值上
            Int64 fileLength = 0;
            FileStream fs = null;
            try {
                fs = File.OpenRead(file);
                fileLength = fs.Length;
            }
            catch (IOException) { /* 忽略拒绝访问的任何文件 */ }
            finally { if (fs != null) fs.Dispose(); }
            return taskLocalTotal + fileLength;
        },

        taskLocalTotal => { // localFinally: 每个任务完成时调用一次
            // 将这个任务的总计值(taskLocalTotal)加到总的总计值(masterTotal)上
            Interlocked.Add(ref masterTotal, taskLocalTotal);
        });

    return masterTotal;
}
```

每个任务都通过 **taskLocalTotal** 变量为分配给它的文件维护它自己的总计值。每个任务在完成工作之后，都通过调用 **Interlocked.Add** 方法(参见第 29 章“基元线程同步构造”)，以一种线程安全的方式更新总的总计值(master total)。由于每个任务都有自己的总计值，所以在一个工作项处理期间，无需进行线程同步。由于线程同步会造成性能的损失，所以不

需要线程同步是好事。只有在每个任务返回之后，**masterTotal** 才需要以一种线程安全的方式更新 **masterTotal** 变量。所以，因为调用 **Interlocked.Add** 而造成的性能损失每个任务只发生一次，而不会每个工作项都发生。

注意，我们向主体委托传递了一个 **ParallelLoopState** 对象，它的定义如下：

```
public class ParallelLoopState{
    public void Stop();
    public Boolean IsStopped { get; }

    public void Break();
    public Int64? LowestBreakIteration{ get; }

    public Boolean IsExceptional { get; }
    public Boolean ShouldExitCurrentIteration { get; }
}
```

参与工作的每个任务都获得它自己的 **ParallelLoopState** 对象，并可通过这个对象和参与工作的其他任务进行交互。**Stop** 方法告诉循环停止处理任何更多的工作，未来对 **IsStopped** 属性的查询会返回 **true**。**Break** 方法告诉循环不再继续处理当前项之后的项。例如，假如 **ForEach** 被告知要处理 100 项，并在处理第 5 项时调用了 **Break**，那么循环会确保前 5 项处理好之后，**ForEach** 才返回。但要注意，这并不是说在这 100 项中，只有前 5 项才会被处理，第 5 项之后的项可能在以前已经处理过了。**LowestBreakIteration** 属性返回在处理过程中调用过 **Break** 方法的最低的项。如果从来没有调用过 **Break**，**LowestBreakIteration** 属性会返回 **null**。

处理任何一项时，如果造成未处理的异常，**IsException** 属性会返回 **true**。处理一项时花费太长时间，代码可查询 **ShouldExitCurrentIteration** 属性看它是否应该提前退出。如果调用过 **Stop**，调用过 **Break**，取消过 **CancellationTokenSource**(由 **ParallelOption** 的 **CancellationToken** 属性引用)，或者处理一项时造成了未处理的异常，这个属性就会返回 **true**。

Parallel 的 **For** 和 **ForEach** 方法都返回一个 **ParallelLoopResult** 实例，它看起来像下面这样：

```
public struct ParallelLoopResult{
    // 如果操作提前终止，以下方法返回 false
    public Boolean IsCompleted { get; }
    public Int64? LowestBreakIteration{ get; }
}
```

可检查属性来了解循环的结果。如果 **IsCompleted** 返回 **true**，表明循环运行完成，所有项都得到了处理。如果 **IsCompleted** 为 **false**，而且 **LowestBreakIteration** 为 **null**，表明参与工作的某个线程调用了 **Stop** 方法。如果 **IsCompleted** 返回 **false**，而且 **LowestBreakIteration** 不为 **null**，表明参与工作的某个线程调用了 **Break** 方法，从 **LowestBreakIteration** 返回的 **Int64** 值是保证得到处理的最低一项的索引。如果抛出异常，应捕捉 **AggregateException** 来得体地恢复。

27.7 并行语言集成查询(PLINQ)

Microsoft 的语言集成查询(Language Integrated Query，LINQ)功能提供了一个简捷的语法来查询数据集合。可用 LINQ 轻松对数据项进行筛选、排序、投射等操作。使用 LINQ to Objects 时，只有一个线程顺序处理数据集合中的所有项；我们称之为**顺序查询**(sequential query)。要提高处理性能，可以使用**并行 LINQ**(Parallel LINQ)，它将顺序查询转换成并行查询，在内部使用任务(排队给默认 **TaskScheduler**)，将集合中的数据项的处理工作分散到多个 CPU 上，以便并发处理多个数据项。和 **Parallel** 的方法相似，要同时处理大量项，或者每一项的处理过程都是一个耗时的计算限制的操作，那么能从并行 LINQ 获得最大的收益。

静态 **System.Linq.ParallelEnumerable** 类(在 System.Core.dll 中定义)实现了 PLINQ 的所有功能，所以必须通过 C#的 **using** 指令将 **System.Linq** 命名空间导入你的源代码。尤其是这个类公开了所有标准 LINQ 操作符(**Where**，**Select**，**SelectMany**，**GroupBy**，**Join**，**OrderBy**，**Skip**，**Take** 等)的并行版本。所有这些方法都是扩展了 **System.Linq.ParallelQuery<T>**类型的扩展方法。要让自己的 LINQ to Objects 查询调用这些方法的并行版本，必须将自己的顺序查询(基于 **IEnumerable** 或者 **IEnumerable<T>**)转换成并行查询(基于 **ParallelQuery** 或者 **ParallelQuery<T>**)，这是用 **ParallelEnumerable** 的 **AsParallel** 扩展方法来实现的[①]，如下所示：

```
public static ParallelQuery<TSource>   AsParallel<TSource>(this IEnumerable<TSource> source)
public static ParallelQuery            AsParallel(this IEnumerable source)
```

下面是将顺序查询转换成并行查询的例子。查询返回的是一个程序集中定义的所有过时(obsolete)方法：

```
private static void ObsoleteMethods(Assembly assembly) {
  var query =
     from type in assembly.GetExportedTypes().AsParallel()

     from method in type.GetMethods(BindingFlags.Public |
          BindingFlags.Instance | BindingFlags.Static)

     let obsoleteAttrType = typeof(ObsoleteAttribute)

     where Attribute.IsDefined(method, obsoleteAttrType)

     orderby type.FullName

     let obsoleteAttrObj = (ObsoleteAttribute)
          Attribute.GetCustomAttribute(method, obsoleteAttrType)

          select String.Format("Type={0}\nMethod={1}\nMessage={2}\n",
               type.FullName, method.ToString(), obsoleteAttrObj.Message);

     // 显示结果
     foreach (var result in query) Console.WriteLine(result);
}
```

① **ParallelQuery<T>**类派生自 **ParallelQuery** 类。

虽然不太常见，但在一个查询中，可以从执行并行操作切换回执行顺序操作，这是通过调用 **ParallelEnumerable** 的 **AsSequential** 方法做到的：

```
public static IEnumerable<TSource> AsSequential<TSource>(this ParallelQuery<TSource> source)
```

该方法将一个 **ParallelQuery<T>**转换回一个 **IEnumerable<T>**。这样一来，在调用了 **AsSequential** 之后执行的操作将只由一个线程执行。

通常，一个 LINQ 查询的结果数据是让某个线程执行一个 **foreach** 语句来计算获得的，就像前面展示的那样。这意味着只有一个线程遍历查询的所有结果。如果希望以并行方式处理查询的结果，就应该使用 **ParallelEnumerable** 的 **ForAll** 方法处理查询：

```
static void ForAll<TSource>(this ParallelQuery<TSource> source, Action<TSource> action)
```

这个方法允许多个线程同时处理结果。可修改前面的代码来使用该方法：

```
// 显示结果
query.ForAll(Console.WriteLine);
```

然而，让多个线程同时调用 **Console.WriteLine** 反而会损害性能，因为 **Console** 类内部会对线程进行同步，确保每次只有一个线程能访问控制台窗口，避免来自多个线程的文本在最后显示时乱成一团。希望为每个结果都执行计算时，才使用 **ForAll** 方法。

由于 PLINQ 用多个线程处理数据项，所以数据项被并发处理，结果被无序地返回。如果需要让 PLINQ 保持数据项的顺序，可调用 **ParallelEnumerable** 的 **AsOrdered** 方法。调用这个方法时，线程会成组处理数据项。然后，这些组被合并回去，同时保持顺序。这样会损害性能。以下操作符生成不排序的操作：**Distinct**，**Except**，**Intersect**，**Union**，**Join**，**GroupBy**，**GroupJoin** 和 **ToLookup**。在这些操作符之后要再次强制排序，只需调用 **AsOrdered** 方法。

以下操作符生成排序的操作：**OrderBy**，**OrderByDescending**，**ThenBy** 和 **ThenByDescending**。在这些操作符之后，要再次恢复不排序的处理，只需调用 **AsUnordered** 方法。

PLINQ 提供了一些额外的 **ParallelEnumerable** 方法，可调用它们来控制查询的处理方式：

```
public static ParallelQuery<TSource> WithCancellation<TSource>(
    this ParallelQuery<TSource> source, CancellationToken cancellationToken)

public static ParallelQuery<TSource> WithDegreeOfParallelism<TSource>(
    this ParallelQuery<TSource> source, Int32 degreeOfParallelism)

public static ParallelQuery<TSource> WithExecutionMode<TSource>(
    this ParallelQuery<TSource> source, ParallelExecutionMode executionMode)

public static ParallelQuery<TSource> WithMergeOptions<TSource>(
    this ParallelQuery<TSource> source, ParallelMergeOptions mergeOptions)
```

显然，**WithCancellation** 方法允许传递一个 **CancellationToken**，使查询处理能提前停止。**WithDegreeOfParallelism** 方法指定最多允许多少个线程处理查询。但不是说指定多少个它就创建多少个。它是有原则的，不需要的不会创建。你一般不用调用该方法。另外，默认是每个内核用一个线程执行查询。但如果想空出一些内核做其他工作，可调用 **WithDegreeOfParallelism** 并传递小于可用内核数的一个数字。另外，如果查询要执行同步 I/O 操作，还可传递比内核数大的一个数字，因为线程会在这些操作期间阻塞。这虽然会浪

费更多线程，但可以用更少的时间生成最终结果。同步 I/O 操作在客户端应用程序中没什么问题，但我强烈建议不要在服务器应用程序中执行同步 I/O 操作。

PLINQ 分析一个查询，然后决定如何最好地处理它。有的时候，顺序处理一个查询可以获得更好的性能，尤其是在使用以下任何操作时：**Concat**，**ElementAt(OrDefault)**，**First(OrDefault)**，**Last(OrDefault)**，**Skip(While)**，**Take(While)**或 **Zip**。使用 **Select(Many)** 或 **Where** 的重载版本，并向你的 **selector** 或 **predicate** 委托传递一个位置索引时也是如此。然而，可以调用 **WithExecutionMode**，向它传递某个 **ParallelExecutionMode** 标志，从而强迫查询以并行方式处理：

```
public enum ParallelExecutionMode {
    Default = 0,           // 让并行 LINQ 决定处理查询的最佳方式
    ForceParallelism = 1   // 强迫查询以其并行方式处理
}
```

如前所述，并行 LINQ 让多个线程处理数据项，结果必须再合并回去。可调用 **WithMergeOptions**，向它传递以下某个 **ParallelMergeOptions** 标志，从而控制这些结果的缓冲与合并方式：

```
public enum ParallelMergeOptions {
    Default       = 0,     // 目前和 AutoBuffered 一样(将来可能改变)
    NotBuffered   = 1,     // 结果一旦就绪就开始处理
    AutoBuffered  = 2,     // 每个线程在处理前缓冲一些结果
    FullyBuffered= 3       // 每个线程在处理前缓冲所有结果
}
```

这些选项使你能在某种程度上平衡执行速度和内存消耗。**NotBuffered** 最省内存，但处理速度慢一些。**FullyBuffered** 消费较多的内存，但运行得最快。**AutoBuffered** 介于 **NotBuffered** 和 **FullyBuffered** 之间。说真的，要想知道应该为一个给定的查询选择哪个并行合并选项，最好的办法就是亲自试验所有选项，并对比其性能。也可以“无脑”地接受默认值，它对于许多查询来说都工作得非常好。请参见以下博客文章，进一步了解 PLINQ 如何在 CPU 内核之间分配工作：

- *http://blogs.msdn.com/pfxteam/archive/2009/05/28/9648672.aspx*
- *http://blogs.msdn.com/pfxteam/archive/2009/06/13/9741072.aspx*

27.8 执行定时计算限制操作

System.Threading 命名空间定义了一个 **Timer** 类，可用它让一个线程池线程定时调用一个方法。构造 **Timer** 类的实例相当于告诉线程池：在将来某个时间(具体由你指定)回调你的一个方法。**Timer** 类提供了几个相似的构造器：

```
public sealed class Timer : MarshalByRefObject, IDisposable {
    public Timer(TimerCallback callback, Object state, Int32      dueTime, Int32 period);
    public Timer(TimerCallback callback, Object state, UInt32     dueTime, UInt32 period);
    public Timer(TimerCallback callback, Object state, Int64      dueTime, Int64 period);
    public Timer(TimerCallback callback, Object state, Timespan dueTime, TimeSpan period);
```

```
}
```

4 个构造器以完全一致的方式构造 **Timer** 对象。**callback** 参数标识希望由一个线程池线程回调的方法。当然，你写的回调方法必须和 **System.Threading.TimerCallback** 委托类型匹配，如下所示：

```
delegate void TimerCallback(Object state);
```

构造器的 **state** 参数允许在每次调用回调方法时都向它传递状态数据；如果没有需要传递的状态数据，可以传递 **null**。**dueTime** 参数告诉 CLR 在首次调用回调方法之前要等待多少毫秒。可以使用一个有符号或无符号的 32 位值、一个有符号的 64 位值或者一个 **TimeSpan** 值指定毫秒数。如果希望回调方法立即调用，为 **dueTime** 参数指定 **0** 即可。最后一个参数 **(period)**指定了以后每次调用回调方法之前要等待多少毫秒。如果为这个参数传递 **Timeout.Infinite(-1)**，线程池线程只调用回调方法一次。

在内部，线程池为所有 **Timer** 对象只使用了一个线程。这个线程知道下一个 **Timer** 对象在什么时候到期(计时器还有多久触发)。下一个 **Timer** 对象到期时，线程就会唤醒，在内部调用 **ThreadPool** 的 **QueueUserWorkItem**，将一个工作项添加到线程池的队列中，使你的回调方法得到调用。如果回调方法的执行时间很长，计时器可能(在上个回调还没有完成的时候)再次触发。这可能造成多个线程池线程同时执行你的回调方法。为解决这个问题，我的建议是：构造 **Timer** 时，为 **period** 参数指定 **Timeout.Infinite**。这样，计时器就只触发一次。然后，在你的回调方法中，调用 **Change** 方法来指定一个新的 **dueTime**，并再次为 **period** 参数指定 **Timeout.Infinite**。以下是 **Change** 方法的各个重载版本：

```
public sealed class Timer : MarshalByRefObject, IDisposable {
    public Boolean Change(Int32    dueTime, Int32    period);
    public Boolean Change(UInt32   dueTime, UInt32   period);
    public Boolean Change(Int64    dueTime, Int64    period);
    public Boolean Change(TimeSpan dueTime, TimeSpan period);
}
```

Timer 类还提供了一个 **Dispose** 方法，允许完全取消计时器，并可在当时处于 pending 状态的所有回调完成之后，向 **notifyObject** 参数标识的内核对象发出信号。以下是 **Dispose** 方法的各个重载版本：

```
public sealed class Timer : MarshalByRefObject, IDisposable {
    public Boolean Dispose();
    public Boolean Dispose(WaitHandle notifyObject);
}
```



重要提示　**Timer** 对象被垃圾回收时，它的终结代码告诉线程池取消计时器，使它不再触发。所以，使用 **Timer** 对象时，要确定有一个变量在保持 **Timer** 对象的存活，否则对你的回调方法的调用就会停止。21.1.3 节“垃圾回收与调试”对此进行了详细讨论和演示。

以下代码演示了如何让一个线程池线程立即调用回调方法，以后每 2 秒调用一次：

```
internal static class TimerDemo {
    private static Timer s_timer;
```

```
    public static void Main() {
        Console.WriteLine("Checking status every 2 seconds");

        // 创建但不启动计时器。确保 s_timer 在线程池线程调用 Status 之前引用该计时器
        s_timer = new Timer(Status, null, Timeout.Infinite, Timeout.Infinite);

        // 现在 s_timer 已被赋值，可以启动计时器了，
        // 现在在 Status 中调用 Change，保证不会抛出 NullReferenceException
        s_timer.Change(0, Timeout.Infinite);

        Console.ReadLine(); // 防止进程终止
    }

    // 这个方法的签名必须和 TimerCallback 委托匹配
    private static void Status(Object state) {
        // 这个方法由一个线程池线程执行
        Console.WriteLine("In Status at {0}", DateTime.Now);
        Thread.Sleep(1000); // 模拟其他工作(1 秒)

        // 返回前让 Timer 在 2 秒后再次触发
        s_timer.Change(2000, Timeout.Infinite);

        // 这个方法返回后，线程回归池中，等待下一个工作项
    }
}
```

如果有需要定时执行的操作，可利用 **Task** 的静态 **Delay** 方法和 C#的 **async** 和 **await** 关键字(第 28 章讨论)来编码。下面重写了前面的代码。

```
internal static class DelayDemo {
  public static void Main() {
     Console.WriteLine("Checking status every 2 seconds");
     Status();
     Console.ReadLine(); // 防止进程终止
  }

  // 该方法可获取你想要的任何参数
  private static async void Status() {
     while (true) {
        Console.WriteLine("Checking status at {0}", DateTime.Now);
        // 要检查的代码放到这里...

        // 在循环末尾，在不阻塞线程的前提下延迟 2 秒
        await Task.Delay(2000); // await 允许线程返回
        // 2 秒之后，某个线程会在 await 之后介入并继续循环
     }
  }
}
```

计时器太多，时间太少

遗憾的是，FCL 事实上提供了几个计时器，大多数程序员都不清楚它们有什么独特之处。让我试着解释一下。

- **System.Threading 的 Timer 类**
  这是上一节讨论的计时器。要在一个线程池线程上执行定时的(周期性发生的)后台任务，它是最好的计时器。

- System.Windows.Forms 的 Timer 类
  构造这个类的实例，相当于告诉 Windows 将一个计时器和调用线程关联(参见 Win32 **SetTimer** 函数)。当这个计时器触发时，Windows 将一条计时器消息(**WM_TIMER**)注入线程的消息队列。线程必须执行一个消息泵来提取这些消息，并把它们派发给需要的回调方法。注意，所有这些工作都只由一个线程完成——设置计时器的线程保证就是执行回调方法的线程。还意味着计时器方法不会由多个线程并发执行。

- System.Windows.Threading 的 DispatcherTimer 类
  这个类是 **System.Windows.Forms** 的 **Timer** 类在 Silverlight 和 WPF 应用程序中的等价物。

- Windows.UI.Xaml 的 DispatcherTimer 类
  这个类是 **System.Windows.Forms** 的 **Timer** 类在 Windows Store 应用中的等价物。

- System.Timers 的 Timer 类
  这个计时器本质上是 **System.Threading** 的 **Timer** 类的包装类。计时器到期(触发)会导致CLR将事件放到线程池队列中。**System.Timers.Timer** 类派生自 **System.ComponentModel** 的 **Component** 类，允许在 Visual Studio 中将这些计时器对象放到设计平面(design surface)上。另外，它还公开了属性和事件，使它在 Visual Studio 的设计器中更容易使用。这个类是在好几年前，Microsoft 还没有理清线程处理和计时器的时候添加到 FCL 中的。这个类完全应该删除，强迫每个人都改为使用 **System.Threading.Timer** 类。事实上，我个人从来不用 **System.Timers.Timer** 类，建议你也不要用它，除非真的想在设计平面上添加一个计时器。

27.9　线程池如何管理线程

现在讨论一下线程池代码如何管理工作者线程和 I/O 线程。但我不打算讲太多细节，因为在这么多年的时间里，随着 CLR 的每个版本的发布，其内部的实现已发生了显著变化。未来的版本还会继续变化。最好是将线程池看成一个黑盒。不要拿单个应用程序去衡量它的性能，因为它不是针对某个单独的应用程序而设计的。相反，作为一个常规用途的线程调度技术，它面向的是大量应用程序；它对某些应用程序的效果要好于对其他应用程序。目前，它的工作情况非常理想，我强烈建议你信任它，因为你很难搞出一个比 CLR 自带的更好的线程池。另外，随着时间的推移，线程池代码内部会更改它管理线程的方式，所以大多数应用程序的性能会变得越来越好。

27.9.1　设置线程池限制

CLR 允许开发人员设置线程池要创建的最大线程数。但实践证明，线程池永远都不应该设置线程数上限，因为可能发生饥饿或死锁。假定队列中有 1000 个工作项，但这些工作项全都因为一个事件而阻塞，等第 1001 个工作项发出信号才能解除阻塞。如果设置最大 1000 个线程，第 1001 个工作项就不会执行，全部 1000 个线程会一直阻塞，用户将被迫终止应

用程序，并丢失所有未保存的工作。

事实上，开发人员很少人为限制自己的应用程序使用的资源。例如，你会启动自己的应用程序，并告诉系统你想限制应用程序能使用的内存量，或限制能使用的网络带宽吗？但出于某种心态，开发人员感觉好像有必要限制线程池拥有的线程数量。

由于存在饥饿和死锁问题，所以 CLR 团队一直都在稳步地增加线程池默认拥有的最大线程数。目前默认值是大约 1000 个线程。这基本上可以看成是不限数量，因为一个 32 位进程最大有 2 GB 的可用地址空间。加载了一组 Win32 和 CLR DLLs，并分配了本地堆和托管堆之后，剩余约 1.5 GB 的地址空间。由于每个线程都要为其用户模式栈和线程环境块(TEB)准备超过 1 MB 的内存，所以在一个 32 位进程中，最多能够有大约 1360 个线程。试图创建更多的线程会抛出 **OutOfMemoryException**。当然，64 位进程提供了 8 TB 的地址空间，所以理论上可以创建千百万个线程。但分配这么多线程纯属浪费，尤其是当理想线程数等于机器的 CPU 数(内核数)的时候。CLR 团队应该做的事情是彻底取消限制，但他们现在还不能这样做，否则预期存在线程池限制的一些应用程序会出错。

System.Threading.ThreadPool 类提供了几个静态方法，可调用它们设置和查询线程池的线程数：**GetMaxThreads**，**SetMaxThreads**，**GetMinThreads**，**SetMinThreads** 和 **GetAvailableThreads**。强烈建议不要调用上述任何方法。限制线程池的线程数，一般都只会造成应用程序的性能变得更差，而不是更好。如果认为自己的应用程序需要几百或几千个线程，表明你的应用程序的架构和使用线程的方式已出现严重问题。本章和第 28 章会演示使用线程的正确方式。

27.9.2 如何管理工作者线程

图 27-1 展示了构成作为线程池一部分的工作者线程的各种数据结构。**ThreadPool.QueueUserWorkItem** 方法和 **Timer** 类总是将工作项放到全局队列中。工作者线程采用一个先入先出(first-in-first-out，FIFO)算法将工作项从这个队列中取出，并处理它们。由于多个工作者线程可能同时从全局队列中拿走工作项，所以所有工作者线程都竞争一个线程同步锁，以保证两个或多个线程不会获取同一个工作项。这个线程同步锁在某些应用程序中可能成为瓶颈，对伸缩性和性能造成某种程度的限制。

现在说一说使用默认 **TaskScheduler**[①](查询 **TaskScheduler** 的静态 **Default** 属性获得)来调度的 **Task** 对象。非工作者线程调度一个 **Task** 时，该 **Task** 被添加到全局队列。但每个工作者线程都有自己的本地队列。工作者线程调度一个 **Task** 时，该 **Task** 被添加到调用线程的本地队列。

工作者线程准备好处理工作项时，它总是先检查本地队列来查找一个 **Task**。存在一个 **Task**，工作者线程就从本地队列移除 **Task** 并处理工作项。要注意的是，工作者线程采用后入先出(LIFO)算法将任务从本地队列取出。由于工作者线程是唯一允许访问它自己的本地队列头

① 其他 **TaskScheduler** 派生对象的行为可能和我在这里描述的不同。

的线程，所以无需同步锁，而且在队列中添加和删除 **Task** 的速度非常快。这个行为的副作用是 **Task** 按照和进入队列时相反的顺序执行。

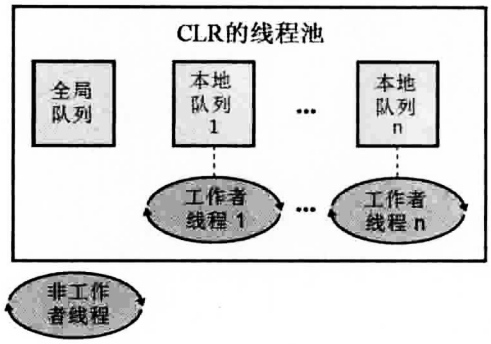


图 27-1　CLR 的线程池

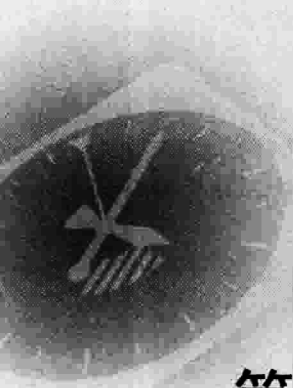


重要提示　线程池从来不保证排队中的工作项的处理顺序。这是合理的，尤其是考虑到多个线程可能同时处理工作项。但上述副作用使这个问题变得恶化了。你必须保证自己的应用程序对于工作项或 **Task** 的执行顺序不作任何预设。

如果工作者线程发现它的本地队列变空了，会尝试从另一个工作者线程的本地队列“偷”一个 **Task**。这个 **Task** 是从本地队列的尾部“偷”走的，并要求获取一个线程同步锁，这对性能有少许影响。当然，希望这种“偷盗”行为很少发生，从而很少需要获取锁。如果所有本地队列都变空，那么工作者线程会使用 FIFO 算法，从全局队列提取一个工作项(取得它的锁)。如果全局队列也为空，工作者线程会进入睡眠状态，等待事情的发生。如果睡眠了太长时间，它会自己醒来，并销毁自身，允许系统回收线程使用的资源(内核对象、栈、TEB 等)。

线程池会快速创建工作者线程，使工作者线程的数量等于传给 **ThreadPool** 的 **SetMinThreads** 方法的值。如果从不调用这个方法(也建议你永远不调用这个方法)，那么默认值等于你的进程允许使用的 CPU 数量，这是由进程的 affinity mask(关联掩码)决定的。通常，你的进程允许使用机器上的所有 CPU，所以线程池创建的工作者线程数量很快就会达到机器的 CPU 数。创建了这么多(CPU 数量)的线程后，线程池会监视工作项的完成速度。如果工作项完成的时间太长(具体多长没有正式公布)，线程池会创建更多的工作者线程。如果工作项的完成速度开始变快，工作者线程会被销毁。





第 28 章　I/O 限制的异步操作

本章内容：

- Windows 如何执行 I/O 操作
- C#的异步函数
- 编译器如何将异步函数转换成状态机
- 异步函数扩展性
- 异步函数和事件处理程序
- FCL 的异步函数
- 异步函数和异常处理
- 异步函数的其他功能
- 应用程序及其线程处理模型
- 以异步方式实现服务器
- 取消 I/O 操作
- 有的 I/O 操作必须同步进行
- I/O 请求优先级

第 27 章重点讲述了如何异步执行计算限制的操作，允许线程池在多个 CPU 内核上调度任务，使多个线程能并发工作，从而高效率地使用系统资源，同时提升应用程序的吞吐能力。本章重点讲述如何异步执行 I/O 限制的操作，允许将任务交由硬件设备处理，期间完全不占用线程和 CPU 资源。然而，线程池仍然扮演了一个重要的角色，因为如同你马上就要看到的那样，各种 I/O 操作的结果还是要由线程池线程来处理的。

28.1　Windows 如何执行 I/O 操作

首先讨论 Windows 如何执行同步 I/O 操作。图 28-1 是连接了几个硬件设备的计算机系统。每个硬件设备都有自己的电路板，每个电路板都集成了一个小型的、特殊用途的计算机，它知道如何控制自己的硬件设备。例如，硬盘驱动器就有一个小的电路板，它知道如何旋转碟片、寻道、在碟片上读写数据以及和计算机内存交换数据。

程序通过构造一个 **FileStream** 对象来打开磁盘文件，然后调用 **Read** 方法从文件中读取数据。调用 **FileStream** 的 **Read** 方法时，你的线程从托管代码转变为本机/用户模式代码，**Read** 内部调用 Win32 **ReadFile** 函数(①)。**ReadFile** 分配一个小的数据结构，称为 I/O 请求包(I/O