## 第14章 并发和异步:

大多数应用程序需要处理一次发生的多件事情（并发）。在本章中，我们从基本的先决条件开始，即线程和任务的基础，然后详细描述异步的原理和C#的异步函数。

在第21章中，我们将更详细地回顾多线程，在第22章中，将介绍并行编程的相关主题。

### 简介

下面是最通常的并发场景:

*编写响应式用户界面*

在Windows Presentation Foundation（WPF）、移动和Windows窗体应用程序中，必须与运行用户界面的代码同时运行耗时的任务，以保持响应能力。

*允许同时处理请求*

在服务器上，客户端请求可以并发到达，因此必须并行处理以保持可伸缩性。如果您使用ASP.NET核心或Web API，运行时会自动为您执行此操作。但是，您仍然需要了解共享状态（例如，使用静态变量进行缓存的效果）。

*并行编程*

如果工作负载在内核之间分配，执行密集计算的代码可以在多核多处理器计算机上更快地执行（第22章专门介绍这一点）。

*推测性执行*

在多核计算机上，有时可以通过预测可能需要做的事情，然后提前完成来提高性能。LINQPad使用这种技术来加快新查询的创建速度。一种变体是并行运行许多不同的算法，这些算法都能解决同一任务。无论哪一个先完成，都会“获胜”——当你无法提前知道哪种算法执行得最快时，这是有效的。

程序可以同时执行代码的通用机制称为多线程。CLR和操作系统都支持多线程，这是并发中的一个基本概念。因此，了解线程的基本知识，特别是线程对共享状态的影响是至关重要的。

### 线程(Threading)

线程是一个可以独立于其他线程进行的执行路径

每个线程都在一个操作系统进程中运行，该进程提供了一个程序运行的独立环境。对于单线程程序，只有一个线程在进程的隔离环境中运行，因此线程可以独占访问它。对于多线程程序，多个线程在单个进程中运行，共享相同的执行环境（尤其是内存）。这在一定程度上就是多线程之所以有用的原因：例如，一个线程可以在后台提取数据，而另一个线程在数据到达时显示数据。此数据称为共享状态。

#### 创建线程(Creating a Thread)

客户端程序（Console、WPF、UWP或Windows窗体）在操作系统自动创建的单个线程（“主”线程）中启动。在这里，除非你做了其他事情，通过创建更多的线程（直接或间接）, 它作为一个单线程应用程序度过了它的生命。

您可以通过实例化thread对象并调用其start方法来创建和启动新线程。Thread最简单的构造函数接受ThreadStart委托：一个无参数的方法，指示应该从哪里开始执行。以下是一个示例：

// NB: All samples in this chapter assume the following namespace imports:

using System;

using System.Threading;

Thread t = new Thread (WriteY); // Kick off a new thread

t.Start(); // running WriteY()

// Simultaneously, do something on the main thread.

for (int i = 0; i < 1000; i++) Console.Write ("x");

void WriteY()

{

for (int i = 0; i < 1000; i++) Console.Write ("y");

}

// Typical Output:

xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy

yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

...

主线程创建一个新线程t，在该线程上运行一个重复打印字符y的方法。同时，主线程重复打印字符x，如图14-1所示。在单核计算机上，操作系统必须为每个线程分配“时间片”（在Windows中通常为20ms）来模拟并发性，从而产生重复的x和y块。在多核或多处理器计算机上，这两个线程可以真正并行执行（受计算机上其他活动进程的竞争），尽管在本例中，由于控制台处理并发请求的机制的微妙性，您仍然会得到重复的x或y块。



提示:

一个线程在其执行与另一个线程上的代码执行交叉在一起的点上被称为抢占(preempted)。这个词经常出现在解释为什么出了问题时！

启动后，线程的IsAlive属性将返回true，直到线程结束。当传递给线程的构造函数的委托完成执行时，线程结束。结束后，线程将无法重新启动。

每个线程都有一个Name属性，您可以设置该属性以便于调试。这在Visual Studio中特别有用，因为线程的名称显示在“线程窗口”和“调试位置”工具栏中。您只能设置一次线程的名称；稍后尝试更改它将引发异常。

Thread.CurrentThead静态属性为你提供当前正在执行的线程:

Console.WriteLine (Thread.CurrentThread.Name);

#### 合并和休眠(Join and Sleep)

你可以调用另一个线程的Join方法, 来等待它结束。

Thread t = new Thread (Go);

t.Start();

t.Join();

Console.WriteLine ("Thread t has ended!");

void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }

这会打印1000次“y”，然后立即打印“线程t已结束！”。调用Join时可以包含超时，以毫秒为单位，也可以作为TimeSpan。如果线程结束，则返回true；如果线程超时，则返回false。

Thread.Sleep将暂停当前线一段指定的时间：

Thread.Sleep (TimeSpan.FromHours (1)); // Sleep for 1 hour

Thread.Sleep (500); // Sleep for 500 milliseconds

Thread.Sleep（0）立即放弃线程的当前时间片，自动将CPU移交给其他线程。Thread.Yield（）做同样的事情，只是它只向在同一处理器上运行的线程放弃。

提示：

Sleep（0）或Yield在高级性能调整的生产代码中偶尔有用。它也是一个很好的诊断工具，可以帮助发现线程安全问题：如果插入thread.Yield（）在你的代码中的任何地方破坏了程序，几乎可以肯定你有一个bug。

在等待睡眠或加入时，线程被阻止。

#### 阻塞（Blocking）

当一个线程由于某种原因暂停执行时，例如在休眠或等待另一个线程通过Join结束时，它被视为被阻塞。被阻塞的线程立即产生其处理器时间片，从那时起，在满足其阻塞条件之前，它不消耗处理器时间。您可以通过线程的ThreadState属性测试被阻止的线程：

bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;

提示：

ThreadState是一个标志枚举，以逐位方式组合了三层数据。但是，大多数值都是多余的、未使用的或不推荐使用的。以下扩展方法将ThreadState剥离为四个有用值之一：Unstarted、Running、WaitSleepJoin和Stopped：

public static ThreadState Simplify (this ThreadState ts)

{

return ts & (ThreadState.Unstarted |

ThreadState.WaitSleepJoin |

ThreadState.Stopped);

}

ThreadState属性可用于诊断目的，但不适用于同步，因为线程的状态可能在测试ThreadState和对该信息执行操作之间发生变化。

当线程阻塞或取消阻塞时，操作系统会执行上下文切换。这会产生较小的开销，通常为一到两微秒。

##### I/O-bound versus compute-bound

一个花费大部分时间等待事情发生的操作被称为I/O绑定——例如下载网页或调用Console.ReadLine。（I/O绑定操作通常涉及输入或输出，但这不是硬性要求：Thread.Sleep也被视为I/O绑定。）相比之下，将大部分时间用于执行CPU密集型工作的操作被称为计算机绑定。

##### Blocking versus spinning

I/O绑定操作的工作方式有两种：它要么在当前线程上同步等待，直到操作完成（如Console.ReadLine、thread.Sleep或thread.Join），要么异步操作，在将来操作完成时触发回调（稍后会详细介绍）。

同步等待的I/O绑定操作大部分时间都在阻塞线程。它们还可以周期性地在一个循环中“旋停”：

while (DateTime.Now < nextStartTime)

Thread.Sleep (100);

抛开有更好的方法（如定时器或信号结构）不谈，另一种选择是线程可以连续旋停：

while (DateTime.Now < nextStartTime)

一般来说，这在处理器时间上是非常浪费的：就CLR和OS而言，线程正在执行重要的计算，因此会相应地分配资源。实际上，我们已经将本应是I/O绑定的操作变成了计算绑定的操作。

提示：

关于旋停和阻塞，有一些细微差别。首先，当你期望一个条件很快得到满足时（可能在几微秒内）时，非常短暂的旋转是有效的，因为它避免了上下文切换的开销和延迟。NET提供了特殊的方法和类来提供帮助——请参阅在线补充“SpinLock和SpinWait”。

其次，阻塞不会产生零成本。这是因为每个线程在其生命周期内占用大约1MB的内存，并导致CLR和操作系统的持续管理开销。因此，在需要处理数百或数千个并发操作的I/O绑定严重的程序中，阻塞可能会很麻烦。相反，这样的程序需要使用基于回调的方法，在等待时完全取消线程。这（在一定程度上）就是我们稍后讨论的异步模式的目的。

#### Local Versus Shared State

CLR为每个线程分配自己的内存堆栈，以便将局部变量分开。在下一个示例中，我们使用局部变量定义一个方法，然后在主线程和新创建的线程上同时调用该方法：

new Thread (Go).Start(); // Call Go() on a new thread

Go(); // Call Go() on the main thread

void Go()

{

// Declare and use a local variable - 'cycles'

for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');

}

在每个线程的内存堆栈上创建一个单独的循环变量副本，因此可以预见，输出是10个问号。

如果线程对同一对象或变量有公共引用，则线程共享数据：

bool \_done = false;

new Thread (Go).Start();

Go();

void Go()

{

if (!\_done) { \_done = true; Console.WriteLine ("Done"); }

}

两个线程共享\_done变量，因此“完成”打印一次而不是两次。

lambda表达式捕获的局部变量也可以共享：

bool done = false;

ThreadStart action = () =>

{

if (!done) { done = true; Console.WriteLine ("Done"); }

};

new Thread (action).Start();

action();

不过，更常见的情况是，字段用于在线程之间共享数据。在以下示例中，两个线程都在同一ThreadTest实例上调用了Go（），因此它们共享相同的\_done字段：

var tt = new ThreadTest();

new Thread (tt.Go).Start();

tt.Go();

class ThreadTest

{

bool \_done;

public void Go()

{

if (!\_done) { \_done = true; Console.WriteLine ("Done"); }

}

}

静态字段提供了另一种在线程之间共享数据的方式：

class ThreadTest

{

static bool \_done; // Static fields are shared between all threads

// in the same process.

static void Main()

{

new Thread (Go).Start();

Go();

}

static void Go()

{

if (!\_done) { \_done = true; Console.WriteLine ("Done"); }

}

}

所有四个例子都说明了另一个关键概念：线程安全性（或者更确切地说，缺乏它！）。输出实际上是不确定的：“完成”可能会打印两次（尽管可能性不大）。然而，如果我们在Go方法中交换语句的顺序，则“完成”打印两次的几率会急剧上升：

static void Go()

{

if (!\_done) { Console.WriteLine ("Done"); \_done = true; }

}

问题是，一个线程可能会在另一个线程执行WriteLine语句的同时评估if语句——在它有机会将done设置为true之前。

提示：

我们的示例说明了共享可写状态可能引入间歇性错误的多种方式之一，而多线程正是这种间歇性错误的罪魁祸首。接下来，我们来看看如何通过锁定程序修复；然而，最好尽可能避免共享状态。稍后我们将看到异步编程模式如何帮助实现这一点。

#### 加锁和线程安全（Locking and Thread Safety）

提示:

加锁和线程安全是一个非常大的论题. 全面的讨论, 请查看第890页的”排他锁”和第898页的”加锁与线程安全”

我们可以通过在读取和写入共享字段时获得独占锁来修复前面的示例。C#提供lock语句就是为了这个目的：

class ThreadSafe

{

static bool \_done;

static readonly object \_locker = new object();

static void Main()

{

new Thread (Go).Start();

Go();

}

static void Go()

{

lock (\_locker)

{

if (!\_done) { Console.WriteLine ("Done"); \_done = true; }

}

}

}

当两个线程同时争用一个锁（可以在任何引用类型的对象上；在本例中为\_locker）时，一个线程会等待或阻塞，直到该锁可用。在这种情况下，它确保一次只有一个线程可以进入其代码块，并且“Done”将只打印一次。以这种方式保护的代码——在多线程上下文中不受不确定性的影响——被称为线程安全。

注意:

即使是自动递增变量的行为也不是线程安全的：表达式x++在底层处理器上作为不同的读增量写操作执行。因此，如果两个线程在锁外同时执行x++，那么变量最终可能会增加一次，而不是两次（或者更糟的是，在某些条件下，x可能会被撕裂，最终出现新旧内容的逐位混合）。

锁定并不是线程安全的灵丹妙药——访问字段时很容易忘记锁定，而且锁定本身也会产生问题（例如死锁）。

可以使用锁定的一个很好的例子是ASP.NET应用程序中频繁访问的数据库对象的共享内存缓存。这种类似的应用程序很容易上手，而且不存在死锁的可能性。我们在第901页的“应用程序服务器中的线程安全”中给出了一个例子。

#### 向线程传递数据

有时，您需要将参数传递给线程的启动方法。最简单的方法是使用lambda表达式，该表达式使用所需的参数调用方法：

Thread t = new Thread ( () => Print ("Hello from t!") );

t.Start();

void Print (string message) => Console.WriteLine (message);

使用这种方法，可以向该方法传递任意数量的参数。您甚至可以将整个实现封装在多语句lambda中：

new Thread (() =>

{

Console.WriteLine ("I'm running on another thread!");

Console.WriteLine ("This is so easy!");

}).Start();

另一种（灵活性较低）技术是将参数传递到Thread的Start方法中：

Thread t = new Thread (Print);

t.Start ("Hello from t!");

void Print (object messageObj)

{

string message = (string) messageObj; // We need to cast here

Console.WriteLine (message);

}

这是因为Thread的构造函数被重载以接受两个委托中的任何一个：

public delegate void ThreadStart();

public delegate void ParameterizedThreadStart (object obj);

#### Lambda表达式和捕获的变量

正如我们所看到的，lambda表达式是将数据传递到线程的最方便、最强大的方法。但是，必须小心在启动线程后意外修改捕获的变量。例如，考虑以下内容：

for (int i = 0; i < 10; i++)

new Thread (() => Console.Write (i)).Start();

输出是不确定的！以下是一个典型的结果： 0223557799

问题是，i变量在循环的整个生命周期中都指向相同的内存位置。因此，每个线程都调用Console。写一个变量，它的值可以在运行时更改！解决方案是使用一个临时变量，如下所示：

for (int i = 0; i < 10; i++)

{

int temp = i;

new Thread (() => Console.Write (temp)).Start();

}

然后，数字0到9中的每一个都被精确地写入一次。（由于线程可以在不确定的时间启动，因此顺序仍然未定义。）

提示:

这类似于我们在第434页“捕获的变量”中描述的问题。问题不仅在于多线程，还在于C#在for循环中捕获变量的规则。

变量temp现在是每个循环迭代的本地变量。因此，每个线程捕获不同的内存位置，这是没有问题的。我们可以通过以下示例更简单地说明早期代码中的问题：

string text = "t1";

Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";

Thread t2 = new Thread ( () => Console.WriteLine (text) );

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

因为两个lambda表达式都捕获相同的文本变量，所以t2被打印两次。

#### 异常处理

创建线程时生效的任何try/catch/finaly块在线程开始执行时都与线程无关。考虑以下程序：

try

{

new Thread (Go).Start();

}

catch (Exception ex)

{

// We'll never get here!

Console.WriteLine ("Exception!");

}

void Go() { throw null; } // Throws a NullReferenceException

此示例中的try/catch语句无效，并且新创建的线程将被未处理的NullReferenceException阻塞。当您考虑到每个线程都有一个独立的执行路径时，这种行为是有意义的。

补救方法是将异常处理程序移到Go方法中：

new Thread (Go).Start();

void Go()

{

try

{

...

throw null; // The NullReferenceException will get caught below

...

}

catch (Exception ex)

{

// Typically log the exception and/or signal another thread

// that we've come unstuck

...

}

}

您需要在生产应用程序中的所有线程入口方法上都有一个异常处理程序，就像您在主线程上所做的那样（通常在执行堆栈的更高级别）。一个未经处理的异常会导致整个应用程序关闭——出现一个丑陋的对话框！

提示:

在编写此类异常处理块时，很少会忽略错误：通常，您会记录异常的详细信息。对于客户端应用程序，您可能会显示一个对话框，允许用户自动将这些详细信息提交到web服务器。然后您可能会选择重新启动应用程序，因为意外的异常可能会使程序处于无效状态。

#### 中心化异常处理(Centralized exception handling)

在WPF、UWP和Windows窗体应用程序中，您可以订阅“全局”异常处理事件Application。DispatcherUnhandledException和应用程序。ThreadException。在通过消息循环调用的程序的任何部分中出现未处理的异常后（这相当于应用程序处于活动状态时在主线程上运行的所有代码），就会触发这些异常。这对于记录和报告错误非常有用（尽管它不会为您创建的工作线程上未处理的异常而触发）。处理这些事件可以防止程序关闭，尽管您可以选择重新启动应用程序以避免未处理的异常可能导致的潜在状态损坏。

#### 前台和后台线程(Foreground Versus Background Threads)

默认情况下，显式创建的线程是前台线程。只要其中任何一个线程在运行，前台线程就会使应用程序保持活动状态，而后台线程则不会。在所有前台线程完成后，应用程序结束，并且任何仍在运行的后台线程都会突然终止。

提示:

线程的前台/后台状态与其优先级（执行时间的分配）无关。

您可以使用线程的IsBackground属性查询或更改线程的后台状态：

static void Main (string[] args)

{

Thread worker = new Thread ( () => Console.ReadLine() );

if (args.Length > 0) worker.IsBackground = true;

worker.Start();

}

如果在没有参数的情况下调用此程序，则工作线程将处于前台状态，并将等待ReadLine语句，等待用户按Enter键。同时，主线程退出，但应用程序仍在运行，因为前台线程仍处于活动状态。另一方面，如果一个参数被传递给Main（），则工作线程被分配后台状态，并且程序几乎在主线程结束时立即退出（终止ReadLine）。

当进程以这种方式终止时，将绕过后台线程执行堆栈中的任何finally块。如果您的程序使用finally（或using）块来执行清理工作，例如删除临时文件，您可以通过在退出应用程序时显式地等待这些后台线程来避免这种情况，方法是加入线程或使用信令结构（请参阅第643页的“信令”）。在任何一种情况下，您都应该指定一个超时，这样您就可以在叛徒线程拒绝完成时放弃它；否则，您的应用程序将无法关闭，而用户无需从任务管理器（或Unix上的kill命令）寻求帮助。

前景线程不需要这种处理，但您必须小心避免可能导致线程无法结束的错误。应用程序无法正常退出的一个常见原因是存在活动的前台线程。

#### 线程优化级(Thread Priority)

线程的Priority属性确定相对于操作系统中的其他活动线程分配的执行时间，具体如下：

enum ThreadPriority { Lowest, BelowNormal, Normal,

AboveNormal, Highest }

当多个线程同时处于活动状态时，这一点就变得重要起来。提升线程的优先级时需要小心，因为它可能会使其他线程挨饿。如果希望某个线程的优先级高于其他进程中的线程，则还必须使用System.Diagnostics中的process类来提升进程优先级：

这可以很好地用于非UI进程，这些进程只需做最少的工作，并且在工作中需要低延迟（快速响应的能力）。对于计算机应用程序（尤其是那些具有用户界面的应用程序），提高进程优先级可能会使其他进程挨饿，从而减慢整个计算机的速度。

#### 信号(Signaling)

有时，您需要一个线程等待，直到收到来自其他线程的通知。这被称为信号。最简单的信号结构是ManualResetEvent。在ManualResetEvent上调用WaitOne会阻塞当前线程，直到另一个线程通过调用Set“打开”信号。在下面的示例中，我们启动一个等待ManualResetEvent的线程。它保持阻塞两秒钟，直到主线程发出信号：

var signal = new ManualResetEvent (false);

new Thread (() =>

{

Console.WriteLine ("Waiting for signal...");

signal.WaitOne();

signal.Dispose();

Console.WriteLine ("Got signal!");

}).Start();

Thread.Sleep(2000);

signal.Set(); // “Open” the signal

调用Set后，信号保持打开状态；您可以通过调用Reset再次关闭它。

ManualResetEvent是CLR提供的几个信号结构之一；我们将在第21章中详细介绍所有这些。

#### 富客户端应用程序中的线程

在WPF、UWP和Windows窗体应用程序中，在主线程上执行长时间运行的操作会使应用程序无响应，因为主线程还处理执行渲染和处理键盘和鼠标事件的消息循环。

一种流行的方法是启动“工作”线程进行耗时的操作。工作线程上的代码运行一个耗时的操作，然后在完成后更新UI。但是，所有富客户端应用程序都有一个线程模型，通过该模型，UI元素和控件只能从创建它们的线程（通常是主UI线程）访问。违反此规则会导致不可预测的行为或引发异常。

因此，当您想从工作线程更新UI时，必须将请求转发到UI线程（技术术语是封送）。实现这一点的低级方法如下（稍后，我们将讨论基于这些解决方案的其他解决方案）：

* 在WPF中，对元素的Dispatcher对象上调用BeginInvoke或Invoke。
* 在UWP应用程序中，对Dispatcher对象调用RunAsync
* 在Windows窗体中，调用控件的BeginInvoke或Invoke

所有这些方法都接受引用要运行的方法的委托。BeginInvoke/RunAsync通过将委托排入UI线程的消息队列（与处理键盘、鼠标和计时器事件的队列相同）来工作。Invoke执行相同的操作，但随后会阻塞，直到UI线程读取并处理完消息为止。正因为如此，Invoke允许您从方法中获取返回值。如果不需要返回值，BeginInvoke/RunAsync更可取，因为它们不会阻塞调用程序，也不会引入死锁的可能性（请参阅第896页的“死锁”）。

提示:

您可以想象，当您调用Application.Run时，执行以下伪代码：

while (!thisApplication.Ended)

{

wait for something to appear in message queue

Got something: what kind of message is it?

Keyboard/mouse message -> fire an event handler

User BeginInvoke message -> execute delegate

User Invoke message -> execute delegate & post result

}

正是这种循环使工作线程能够将委托封送到UI线程上执行。

为了进行演示，假设我们有一个WPF窗口，其中包含一个名为txtMessage的文本框，我们希望工作线程在执行耗时的任务（我们将通过调用thread.Sleep来模拟）后更新其内容。以下是我们的做法：

partial class MyWindow : Window

{

public MyWindow()

{

InitializeComponent();

new Thread (Work).Start();

}

void Work()

{

Thread.Sleep (5000); // Simulate time-consuming task

UpdateMessage ("The answer");

}

void UpdateMessage (string message)

{

Action action = () => txtMessage.Text = message;

Dispatcher.BeginInvoke (action);

}

}

多个UI线程

如果多个UI线程各自拥有不同的窗口，那么它们就有可能具有多个。主要情况是，您有一个具有多个顶级窗口的应用程序，通常称为单文档接口（SDI）应用程序，如Microsoft Word。每个SDI窗口通常在任务栏上显示为一个单独的“应用程序”，并且在功能上与其他SDI窗口基本隔离。通过给每个这样的窗口赋予其自己的UI线程，可以使每个窗口相对于其他窗口更具响应性。

运行此操作会立即显示一个响应窗口。五秒钟后，它会更新文本框。代码与Windows窗体类似，只是我们改为调用（Form的）BeginInvoke方法：

void UpdateMessage (string message)

{

Action action = () => txtMessage.Text = message;

this.BeginInvoke (action);

}

#### 同步上下文(Synchronization Contexts)

在System.ComponentModel命名空间中，有一个名为Synchronization Context的类，它可以实现线程通用化的编排(marshaling)。移动和桌面的富客户端API（UWP、WPF和Windows窗体）分别定义和实例化SynchronizationContext这个子类，它可以通过静态属性SynchronizaContext.Current获得（在UI线程上运行的时候）。捕获此属性可以让您稍后从工作线程“post”到UI控件：

partial class MyWindow : Window

{

SynchronizationContext \_uiSyncContext;

public MyWindow()

{

InitializeComponent();

// Capture the synchronization context for the current UI thread:

\_uiSyncContext = SynchronizationContext.Current;

new Thread (Work).Start();

}

void Work()

{

Thread.Sleep (5000); // Simulate time-consuming task

UpdateMessage ("The answer");

}

void UpdateMessage (string message)

{

// Marshal the delegate to the UI thread:

\_uiSyncContext.Post (\_ => txtMessage.Text = message, null);

}

}

这很有用，因为相同的技术适用于所有富客户端用户界面的API

调用Post相当于在Dispatcher或Control上调用BeginInvoke；还有一个与Invoke等效的Send方法。

#### 线程池(The Thread Pool)

无论何时启动一个线程，都要花费几百微秒来组织新的局部变量堆栈。线程池通过具有预先创建的可回收线程池来减少这种开销。线程池对于高效的并行编程和细粒度并发至关重要；它允许短操作运行，而不会被线程启动的开销淹没。

使用池线程时需要注意以下几点：

* + 不能设置池线程的名称，这会使调试更加困难（尽管在Visual Studio的“线程”窗口中调试时可以附加描述）。
  + 线程池的线程总是后台线程
  + 阻塞池线程会降低性能（请参阅第647页的“线程池中的卫生”）。

您可以自由更改池线程的优先级——当释放回池中时，它将恢复正常。

您可以通过Thread.CurrentThread.IsThreadPoolThread属来确定当池线程是否正在执行。

##### 进入线程池（Entering the thread pool）

在池线程上显式运行某些东西的最简单方法是使用Task.Run（我们将在下一节中对此进行更详细的介绍）：

// Task is in System.Threading.Tasks

Task.Run (() => Console.WriteLine ("Hello from the thread pool"));

因为以前NET Framework 4.0不存在Task静态类，一种常见的替代方法是调用ThreadPool.QueueUserWorkItem：

ThreadPool.QueueUserWorkItem(notUsed => Console.WriteLine ("Hello"));

提示：

下面将隐式的使用线程池

1. ASP.NET Core和WebAPI应用服务
2. System.Timers.Timer 和 System.Threading.Timer
3. 并行编程结构（将在第22章详细介绍）
4. 废弃的BackgroundWorker类

##### 线程池中的卫生（Hygiene in the thread pool）

线程池提供了另一个功能，即确保临时过量的计算绑定工作不会导致CPU超额订阅。超额订阅是指活动线程比CPU内核多的情况，操作系统必须对线程进行时间切片。超额订阅会损害性能，因为时间切片需要昂贵的上下文切换，并且可能会使CPU缓存失效，而这些缓存在向现代处理器提供性能方面已经变得至关重要。

CLR通过排队和限制任务的启动来防止线程池中的超额订阅。它首先运行与硬件核心一样多的并发任务，然后通过爬山算法调整并发级别，不断向特定方向调整工作负载。如果吞吐量提高，它将继续朝着相同的方向发展（否则将逆转）。这确保了它始终跟踪最佳性能曲线——即使面对计算机上的竞争过程活动。

如果满足以下两个条件，CLR的策略效果最佳：

* 工作项大多是短时间运行的（<250毫秒，理想情况下<100毫秒），因此CLR有很多机会进行测量和调整。
* 耗时多被阻塞的工作并没有占据主导地位

阻塞很麻烦，因为它给CLR一个错误的想法，即它正在加载CPU。CLR足够聪明，可以检测和补偿（通过向池中注入更多线程），尽管这会使池容易受到后续超额订阅的影响。它还可能引入延迟，因为CLR会抑制注入新线程的速率，尤其是在应用程序生命的早期（在客户端操作系统上更是如此，因为它倾向于降低资源消耗）。

当您想要充分利用CPU时（例如，通过第22章中的并行编程API），在线程池中保持良好的卫生状况尤其重要。

### Tasks

线程是一种用于创建并发的低等级工具，因此，它有局限性，尤其是以下方面：

* 尽管将数据传递到启动的线程中很容易，但要从合并（Join）的线程中获取“返回值”却没有简单的方法。您需要设置某种共享字段。如果线程中操作抛出异常，那么捕获和传播该异常同样痛苦。
* 你不能让线程在它完成时启动其他东西；相反，你必须合并（Join）加入它（在这个过程中是阻塞你自己的线程）。

这些限制阻碍了细粒度并发；换言之，它们使得通过组合较小的并发操作来组合较大的并发操作变得困难（这对于我们将在下面的部分中讨论的异步编程来说是必不可少的）。这反过来又导致了对手动同步（锁定、信号等）的更大依赖以及随之而来的问题。

线程的直接使用也会对性能产生影响，我们在第646页的“线程池”中对此进行了讨论。如果您需要运行数百或数千个并发I/O绑定操作，基于线程的方法纯粹在线程开销中会消耗数百或数千兆字节的内存。

Task类可以帮助解决所有这些问题。与线程相比，Task是更高级的抽象——它表示可能由线程支持也可能不由线程支持的并发操作。任务是组合的（您可以通过使用continuation将它们链接在一起）。他们可以使用线程池来减少启动延迟，通过TaskCompletionSource，他们可以使用回调方法，在等待I/O绑定操作时完全避免线程。

Task类型是在Framework 4.0中作为并行编程库的一部分引入的。然而，它们后来得到了增强（通过使用awaiters），以便在更通用的并发场景中发挥同样的作用，并且是C#异步函数的支持类型。

提示：

在本节中，我们忽略了专门针对并行编程的任务的特征；我们将在第22章中介绍它们。

#### Starting a Task

启动由线程支持的Task的最简单方法是使用静态方法Task.Run（Task类位于System.Threading.Tasks命名空间中）。只需传入一个Action委托：

Task.Run (() => Console.WriteLine ("Foo"));

提示:

默认情况下，任务使用池线程，即后台线程。这意味着当主线程结束时，您创建的任何任务也会结束。因此，要从控制台应用程序运行这些示例，必须在启动任务后阻止主线程（例如，通过等待任务或调用console.ReadLine）：

Task.Run (() => Console.WriteLine ("Foo"));

Console.ReadLine

在本书的LINQPad配套示例中，Console.ReadLine被省略，因为LINQPad进程使后台线程保持活动状态。