## 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）