## 线程的高级技术:

我们将论下面这些内容:

排他锁(Semaphore和read/writer locks)

所有的信号结构(AutoResetEvent, ManualResetEvent,CountdownEvent and Barrier)

懒初始化(Lazy<T> 和 LazyInitializer)

线程本地存储(TheadStaticAttribute,ThreadLocal<T>和GetData/SetData)

计时器(Timers)

线程是一个如此庞大的主题，我们在网上添加了额外的材料来完成, 请访问http://albahari.com/threading, 它将论更神秘的下列话题:

Monitor.Wait和Monitor.Pulse 这种更特殊的信号场景:

非阻塞同步技术这种微优化技术(Interlocked, memory barriers, volatile)

自旋锁和自锁等待(SpinLock,SpinWait)这种高并发场景.

### 同步概述:

同步是协调可预测的并发操作的行为结果, 当多个线程访问共享数据时, 同步尤为重要；在这些场景出奇的容易踩中地雷。

可以说，最简单、最有用的同步工具是依次执行和第14章中描述的任务组合. 通过制定并行程序转换为与连续和组合, 把它们串在一起的异步操作，这些减少了你对锁定和信号的需要。然而有时低等级结构开始发挥作用.

同步结构被分为3类:

排他锁(Excluesive locking)

独占锁定构造只允许一个线程执行某些活动或

一次执行一段代码。它们的主要目的是让线程

在不相互干扰的情况下访问共享写入状态。独家

锁定结构有: lock、Mutex和SpinLock

非排他锁(Nonexclusive locking)

非独占锁定可以限制并发性。非排他性锁定

构造有: Semaphore（Slim）和ReaderWriterLock（Slim）

信号(Signaling)

允许线程进行阻止，直到收到来自的一个或多个其他线程的通知。

信号结构包括ManualResetEvent（Slim），

AutoResetEvent、CountdownEvent和Barrier。前三个是参考

作为事件等待句柄。

通过使用非锁定同步结构，在不锁定的情况下操作共享状态是可能(和刺手)的。

这些有: Thread.MemoryBarrier，Thread.VolatileRead，Thread.VolatileWrite，

volatile关键字和Interlocked类。

我们在网上讨论这个话题使用Monitor的Wait/Pulse方法，您可以使用该方法编写自定义信号逻辑

### 排它锁(Exclusive Locking)

这里有3种排它锁结构分别是: lock关键字语句, Mutex和SpinLock. Lock结构初最认为最实用和广泛的包括两个最佳的应用场景:

Mutex允许您跨越多个进程（计算机范围的锁）。

SpinLock实现了一个微处理器优化，可以减少在高并发场景中的上下文切换

#### Lock关键字语句

为说明什么情况需要锁, 请看如下的类:

class ThreadUnsafe

{

static int \_val1 = 1, \_val2 = 1;

static void Go()

{

if (\_val2 != 0) Console.WriteLine (\_val1 / \_val2);

\_val2 = 0;

}

}

这个类不是线程安全的: 如果Go方法被两个线程同时调用, 它可能会导致除零错误, 这是因为一个线程执行到 if判断和 console.Write之间时, 有可能另一个线程把\_val2置为0. 下面是锁来修复此问题的代码:

class ThreadSafe

{

static readonly object \_locker = new object();

static int \_val1 = 1, \_val2 = 1;

static void Go()

{

lock (\_locker)

{

if (\_val2 != 0) Console.WriteLine (\_val1 / \_val2);

\_val2 = 0;

}

}

}

只有一个线程在一个时间点才能锁住这个同步对象(此示例是\_locker), 并且任何争用这个同步对象的线程都会被阻塞，直到锁被释放为止。如果多个线程争用这个锁, 它们将被除队列到”ready queue”并且会遵循先来先得的原则。排它锁有时被认为是强制的按序访问受保护的内容。因为访问lock块内部的代码不是同时的，这样我们就保护了Go方法的内部处理\_val1和\_val2的逻辑。

#### Monitor.Enter和Monitor.Exit

C#的lock关键字语句事实上是Monitor.Enter和Monitor.Exit方法以及Try/finally语句块的语法糖。 前面Go方法真实的情况如同下面的代码：

Monitor.Enter (\_locker);

try

{

if (\_val2 != 0) Console.WriteLine (\_val1 / \_val2);

\_val2 = 0;

}

finally { Monitor.Exit (\_locker); }

如果没有先执行Monitor.Enter而去执行Monitor.Exit将会抛出异常。（也就是说必须先Monitor.Enter再Monitor.Exit）

#### LockTaken重载

我们刚刚演示的代码有一个微妙的漏洞。考虑一下（不太可能）在对Monitor.Enter和Try块的调用之间引发异常的事件。（可能是由于OutOfMemoryException，或者在.NET Framework中，如果线程被中止）。在这种情况下，锁可能会也可能不会被带走。如果锁被拿走了，它就不会松开——因为我们永远执行不到try/finaly块。这将导致锁泄漏。为了避免这种危险，Monitor.Enter定义以下重载：

public static void Enter (object obj, ref bool lockTaken);

当Enter方法执行异常时并且锁没有去除时，lockTaken将会被置为false

下面是更稳健的使用模式（更为精准的把lock关键字语句的转译语句块）

bool lockTaken = false;

try

{

Monitor.Enter (\_locker, ref lockTaken);

// Do your stuff...

}

finally { if (lockTaken) Monitor.Exit (\_locker); }

#### TryEnter

Monitor还提供了TryEnter方法，该方法允许指定超时，以毫秒为单位或作为TimeSpan。如果锁为已获取，如果由于方法超时而未获取锁，则为false。TryEnter也可以在没有参数的情况下调用，这将“测试”锁，并立即超时如果不能立即获得锁。与Enter方法一样，TryEnter是重载以接受lockTaken参数。

#### 选择同步对象

您可以使用每个参与线程可见的任何对象作为同步对象，受一条严格规则约束：它必须是引用类型。同步对象通常是私有的（因为这有助于封装锁定逻辑），并且通常是实例或静态字段。同步对象可以兼作它正在保护的对象，如下面示例中的\_list字段所做的那样

class ThreadSafe

{

List <string> \_list = new List <string>();

void Test()

{

lock (\_list)

{

\_list.Add ("Item 1");

...

专用于锁定的字段（如前面的示例中的\_locker）允许对锁的范围和粒度进行精确控制。您也可以使用包含对象（this）作为同步对象

lock (this) { ... }

甚至是它的类型：

lock (typeof (Widget)) { ... } // For protecting access to statics

以这种方式锁定的缺点是没有封装锁定逻辑，因此防止死锁和过度阻塞变得更加困难

还可以锁定lambda表达式或匿名表达式捕获的局部变量方法。

#### 何时加锁

在最简单的情况下——对单个字段的赋值操作——您必须考虑同步。在下面的类中，Increment和Assign方法都不是线程安全的：

class ThreadUnsafe

{

static int \_x;

static void Increment() { \_x++; }

static void Assign() { \_x = 123; }

}

下面是线程安全的版本

static readonly object \_locker = new object();

static int \_x;

static void Increment() { lock (\_locker) \_x++; }

static void Assign() { lock (\_locker) \_x = 123; }

没有加锁，会产生两个问题：

* 诸如递增变量（或者甚至读取/写入变量，在某些条件下）不是原子的。
* 编译器、CLR和处理器有权改变指令和CPU寄存器中的缓存变量的顺序以提高性能——只要是这样优化不会改变单线程程序（或使用锁的多线程程序）结果。

锁定缓解了第二个问题，因为它在以及锁定之后，内存屏障就像一道“栅栏”使得重新排序和缓存无法穿透。

提示：这不仅适用于锁，而且适用于所有同步构造。因此，例如，如果你使用信号结构，确保一次只有一个线程读取/写入变量，你不需要锁。因此，以下代码是线程安全的而不锁定在x:

var signal = new ManualResetEvent (false);

int x = 0;

new Thread (() => { x++; signal.Set(); }).Start();

signal.WaitOne();

Console.WriteLine (x); // 1 (always)

在“无阻塞同步（Nonblocking Synchronization）”中，我们解释了这种需求是如何产生的，以及内存屏障（memory barriers）和Interlocked类可以提供锁定的替代方案这些情况。

#### 加锁和原子性（Locking and Atomicity）

如果一组变量总是在同一个锁中读写，你可以说变量是以原子方式读取和写入的。假设字段x和y总是在对象锁定器上的锁定中读取和分配

lock (locker) { if (x != 0) y /= x; }

我们可以说，x和y是原子访问的，因为代码块不能被另一个线程的操作分割或抢占，更改x或y并使其结果无效。你永远不会得到零误差除法，前提是x和y总是在同一个独占锁中被访问

***提示***：如果异常在锁块内抛出（无论多线程是否卷入）则锁块提供的原子性将失效，例如，考虑以下内容：

decimal \_savingsBalance, \_checkBalance;

void Transfer (decimal amount)

{

lock (\_locker)

{

\_savingsBalance += amount;

\_checkBalance -= amount + GetBankFee();

}

}

如果GetBankFee（）引发异常，则银行会赔钱。在这种情况下，我们可以通过早些时候调用GetBankFee。更复杂案例的解决方案是在catch或finally中实现“回滚”逻辑块

指令原子性是一个不同但相似的概念：指令是原子，如果它在底层处理器上不可分割地执行。

#### 嵌套锁（Nested Locking）

线程可以以嵌套（可重入）的方式重复锁定同一对象：

lock (locker)

lock (locker)

lock (locker)

{

// Do something...

}

可替换为：

Monitor.Enter (locker); Monitor.Enter (locker); Monitor.Enter (locker);

// Do something...

Monitor.Exit (locker); Monitor.Exit (locker); Monitor.Exit (locker);

在这些场景中，直到最外层的lock语句退出或Monitor.Exit匹配完后，这个对象才解锁。

当一个方法从锁内调用另一个方法时，嵌套锁定很有用：

object locker = new object();

lock (locker)

{

AnotherMethod();

// We still have the lock - because locks are reentrant.

}

void AnotherMethod()

{

lock (locker) { Console.WriteLine ("Another method"); }

}

线程只能阻塞第一个（最外面的）锁

#### 死锁（Deadlocks）

当两个线程各自等待另一个线程持有的资源时，就会发生死锁，因此两者都不能继续。说明这一点的最简单方法是使用两个锁：

object locker1 = new object();

object locker2 = new object();

new Thread (() => {

lock (locker1)

{

Thread.Sleep (1000);

lock (locker2); // Deadlock

}

}).Start();

lock (locker2)

{

Thread.Sleep (1000);

lock (locker1); // Deadlock

}

您可以使用三个或更多线程创建更复杂的死锁链。

***提示：***

CLR在标准托管环境中并不像SQL Server，它不会自动检测和解析死锁，并通过终止其中任何一个线程来解除死锁。一个线程死锁导致所有参与线程无限期地被阻塞，

除非您指定了锁定超时。（在SQL CLR集成宿主下，然而死锁会被自动检测到，并且在其中一个线程上引发[可捕获]异常。）

死锁是多线程中最困难的问题之一，尤其是当存在许多相互关联的对象时。从根本上讲，难点在于你不能确定你需要的锁哪些已被其它线程拿走了。

因此，您可能会在类x中锁定私有字段a，而不知道您的调用者（或调用者的调用者）已经锁定了类y中的字段b。与此同时，另一个线程正在做相反的事情——这就创建死锁。（好的）面向对象的设计模式加剧了这一现象，因为这样的模式创建的调用链直到运行的时候才确定。

流行的建议是“以一致的顺序锁定对象以防止死锁。”尽管在我们最初的例子中很有用，但很难应用于像描述的那样的场景。一个更好的策略是小心的锁定调用对象里的方法周围，那样可能引用回您自己拥有的对象。此外，请考虑是否您确实需要对其他类中方法的调用周围加锁（但有时这种情形你有另外的选项去做同样的事——我们将在第898页的“锁定和线程安全”看到此种方法。更多地依赖于更高级别的同步选项，如任务延续/组合、数据并行性和不可变类型（稍后将介绍章）可以减少对加锁的需要。

***提示：***

这里有一种替代方法去理解这个问题：当你调用其他代码的同时持有一个锁，封装那把锁会微漏。这不是CLR中的故障；是加锁的一个基本限制。加锁的关键点是在程程序中被锁定地址，包括软件内存事务。

当拥有锁的情况下调用Dispatcher.Invoke（在WPF应用程序中）或Control.Invoke（在Windows窗体应用程序中）时会出现另一种死锁情况。如果用户界面恰好在运行另一个方法，该方法正在等待同一个锁，那么死锁就会立即发生。您通常可以通过调用BeginInvoke而不是Invoke（或者在存在同步上下文时依赖于隐式执行此操作的异步函数）来解决此问题。或者，您可以在调用Invoke之前释放您的锁，虽然当您的调用者取出了锁它不起作用。

#### 性能

锁定很快：如果锁定未被控制，您可以在2020时代的计算机上用不到20纳秒的时间获取并释放锁定。如果它被争用，那么相应的上下文切换会使开销更接近微秒等级，尽管在实际重新调度线程之前可能需要更长的时间。

#### 互斥（Mutex）

Mutex就像一个C#锁，但它可以跨多个进程工作。换句话说，Mutex可以是计算机范围的，也可以是应用范围的。获取和释放一个未受控制的Mutex大约需要半微秒，比锁慢20多倍。

使用Mutex类，可以调用WaitOne方法进行锁定，调用ReleaseMutex进行解锁。与lock语句一样，Mutex只能从获得它的同一线程中释放。

***提示***：

如果您忘记调用ReleaseMutex，而只调用Close或Dispose，则当其它线程等待该互斥对象时会抛出bandonedMutexException。

跨进程互斥的一个常见用途是确保一次只能运行程序的一个实例。以下是操作方法：

// Naming a Mutex makes it available computer-wide. Use a name that's

// unique to your company and application (e.g., include your URL).

using var mutex = new Mutex (true, @"Global\oreilly.com OneAtATimeDemo");

// Wait a few seconds if contended, in case another instance

// of the program is still in the process of shutting down.

if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))

{

Console.WriteLine ("Another instance of the app is running. Bye!");

return;

}

try { RunProgram(); }

finally { mutex.ReleaseMutex (); }

void RunProgram()

{

Console.WriteLine ("Running. Press Enter to exit");

Console.ReadLine();

}

提示：

如果您在终端服务下或在单独的Unix控制台中运行，则计算机范围的Mutex通常仅对同一会话中的应用程序可见。要使其对所有终端服务器会话可见，请在其名称前面加上Global\，如示例所示。

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