## 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）

如果一个程序或方法能够在任何多线程场景中正确工作，那么它就是线程安全的。线程安全主要通过锁定和减少线程交互的可能性来实现。

通用类型的整体很少是线程安全的，原因如下：

* 全线程安全的开发负担可能很大，特别是如果一个类型有许多字段（每个字段都有可能在任意多线程上下文中进行交互）。
* 线程安全可能会带来性能成本（在一定程度上，无论该类型是否实际由多个线程使用，都需要支付）。
* 线程安全类型并不一定使使用它的程序成为线程安全的，而且后者所涉及的工作往往会使前者变得多余。

因此，线程安全通常只在需要处理特定多线程场景的地方实现

然而，有几种方法可以“作弊”，并使大型复杂类在多线程环境中安全运行。一种是牺牲粒度，将大部分代码（甚至是对整个对象的访问）封装在一个独占锁中，从而在高级别上强制执行序列化访问。事实上，如果您想在多线程上下文中使用线程不安全的第三方代码（或者大多数.NET类型），这种策略是必不可少的。诀窍是简单地使用相同的独占锁来保护对线程不安全对象上的所有属性、方法和字段的访问。当对象的方法都执行得很快（否则会有很多阻塞），那么这种解决方案就可以很好地工作。

***提示***：

撇开原始类型不谈，少数NET类型在实例化时，对于并发只读访问以外的任何内容都是线程安全的。开发人员有责任叠加线程安全性，通常使用独占锁。（我们在第22章中介绍的System.collections.Concurrent中的集合是一个例外。）

另一种作弊方法是通过最小化共享数据来最小化线程交互。这是一种非常好的方法，并且在“无状态”的中间层应用程序和网页服务器中隐式使用。因为多个客户端请求可以同时到达，所以它们调用的服务器方法必须是线程安全的。无状态设计（由于可伸缩性的原因而流行）本质上限制了交互的可能性，因为类不在请求之间保存数据。然后，线程交互就仅限于您可能选择创建的静态字段这种情形，如用于在内存中缓存常用数据以及提供身份验证和审计等基础设施服务。

另一个解决方案（在富客户端应用程序中）我们只在UI线程上访问共享的状态数据。正如我们在第14章中看到的，异步函数使这一点变得容易。

#### 线程安全和.NET类型（Thread Safety and .NET Types）

您可以使用锁定将线程不安全的代码转换为线程安全的代码。这方面的一个很好的应用是: 当实例化时，.NET中几乎所有的非基础类型都不是线程安全的（除了只读访问之外），但如果通过锁来保护给定对象，则可以在多线程代码中使用它们。以下是一个示例，其中两个线程模拟同时向同一List集合添加一个项，然后枚举该列表：

class ThreadSafe

{

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

static void Main()

{

new Thread (AddItem).Start();

new Thread (AddItem).Start();

}

static void AddItem()

{

lock (\_list) \_list.Add ("Item " + \_list.Count);

string[] items;

lock (\_list) items = \_list.ToArray();

foreach (string s in items) Console.WriteLine (s);

}

}

在这种情况下，我们锁定\_list对象本身。如果我们有两个相互关联的列表，我们需要选择一个共同的对象来锁定（我们可以指定其中一个列表，或者更好：使用一个独立的字段）。

枚举.NET集合也是线程不安全的，因为如果在枚举过程中修改列表，就会引发异常。在本例中，我们首先将项复制到数组中，而不是在枚举期间锁定。如果我们在枚举过程中所做的工作可能很耗时，这就避免了过度持有锁。（另一种解决方案是使用读写器锁；请参阅第907页的“读写器锁定”。）

#### 在线程安全对象周围锁定(Locking around thread-safe objects)

有时，您还需要锁定访问线程安全对象。为了说明这一点，想象一下.NET的List类确实是线程安全的，我们想向列表中添加一个项：

if (!\_list.Contains (newItem)) \_list.Add (newItem);

不管列表是否是线程安全的，这句话肯定不是！整个if语句需要封装在锁中，以防止项在包含测试和添加新项目之间抢占。然后，在我们修改该列表的任何地方都需要使用相同的锁。例如，下面的语句也需要封装在相同的锁中，以确保它不会抢占前面的语句

\_list.Clear();

换句话说，我们需要与线程不安全的集合类完全一样锁定（使List类的假设线程安全性成为冗余）。

提示:

在高度并发的环境中，围绕访问集合的锁定可能会导致过多的阻塞。为此.NET提供了线程安全的队列、堆栈和字典，我们将在第22章中对此进行讨论。

#### 静态成员(Static members)

只有当所有并发线程都知道并使用该锁时，将对对象的访问封装在自定义锁周围才有效。如果对象的作用域很广，则情况可能并非如此。最糟糕的情况是使用公共类型中的静态成员。例如，假设DateTime结构的静态属性DateTime.Now，这不是线程安全的，两个并发调用可能导致混乱的输出或异常。使用外部锁定解决此问题的唯一方法,可能是在调用DateTime.Now之前, 锁定类型本身——lock（typeof（DateTime））。只有在所有程序员都同意的情况下才能工作（这是不可能的）。此外，锁定类型本身也会产生问题。

出于这个原因，DateTime结构上的静态成员经过精心编程，可以实现线程安全。这在.NET是贯穿始终。：静态成员是线程安全的；实例成员不是。

在编写供公众使用的类型时，遵循这种模式也是有意义的，这样就不会造成不可能的线程安全难题。换言之，通过使静态方法成为线程安全的，您进行编程是为了不排除该类型消费者的线程安全。

***提示:***

静态方法中的线程安全性是必须明确编码的：它不会因为方法是静态的而自动发生！

#### 只读线程安全(Read-only thread safety)

使类型对并发只读访问是线程安全的（在可能的情况下）是有利的，因为这意味着消费者可以避免过度锁定。许多.NET类型遵循这一原则：例如，集合对于并发读取器来说是线程安全的。

遵循这一原则很简单：如果您将一个类型记录为并发只读访问的线程安全类型，则不要在方法中写入使用者期望为只读的字段（或锁定这样做）。例如，在集合中实现ToArray（）方法时，可以从压缩集合的内部结构开始。然而，这将使它对期望它是只读的消费者来说是线程不安全的。

只读线程安全性是枚举器与“enumerables”分开的原因之一：两个线程可以同时在一个集合上枚举，因为每个线程都有一个单独的枚举器对象。

***提示:***

在没有文档的情况下，在假设一个方法本质上是否是只读的时要谨慎。一个很好的例子是Random类：当您调用Random.Next（）时，其内部实现要求它更新私有种子值。因此，您必须使用Random类进行锁定，或者为每个线程维护一个单独的实例。

#### 服务应用程序的线程安全(Thread Safety in Application Servers)

应用程序服务器需要是多线程的，以处理同时发生的客户端请求。ASP.NET核心和Web API应用程序是隐式多线程的。这意味着，在服务器端编写代码时，如果处理客户端请求的线程之间存在交互的可能性，则必须考虑线程安全性。幸运的是，这种可能性很少见；典型的服务器类要么是无状态的（没有字段），要么有一个激活模型，为每个客户端或每个请求创建一个单独的对象实例。交互通常只通过静态字段产生，有时用于缓存数据库的内存部分以提高性能。

例如, 假设你有一个RetrieveUser方法向数据查询

如果这个方法被频繁的调用, 你应从一个静态字典中获取缓存结果以改善性能。以下是一个概念简单的解决方案，它将线程安全性考虑在内：

static class UserCache

{

static Dictionary <int, User> \_users = new Dictionary <int, User>();

internal static User GetUser (int id)

{

User u = null;

lock (\_users)

if (\_users.TryGetValue (id, out u))

return u;

u = RetrieveUser (id); // Method to retrieve from database;

lock (\_users) \_users [id] = u;

return u;

}

}

我们必须，用最少的锁定读取和更新字典，以确保线程安全。在这个例子中，我们在锁定的简单性和性能之间选择了一个实用的折衷方案。我们的设计产生了一个小的低效可能性：如果两个线程同时使用相同的以前未检索的id调用此方法，RetrieveUser方法将被调用两次，并且dictionary将不必要地更新。在整个方法上锁定一次可以防止这种情况发生，但这会造成更糟糕的低效率：在调用RetrieveUser期间，整个缓存将被锁定，在此期间，其他线程将被阻止检索任何用户。

一个理想的解决方案是： 需要使用我们在第677页“同步完成”中描述的策略。我们不缓存User，而是缓存Task＜User＞，然后调用者等待它：

static class UserCache

{

static Dictionary <int, **Task<User>**> \_userTasks =

new Dictionary <int, Task<User>>();

internal static **Task<User>** GetUserAsync (int id)

{

lock (\_userTasks)

if (\_userTasks.TryGetValue (id, out var userTask))

return userTask;

else

return **\_userTasks [id] = Task.Run (() => RetrieveUser (id))**;

}

}

请注意，我们现在有一个覆盖整个方法逻辑的锁。我们可以在不损害并发性的情况下做到这一点，因为我们在锁内所做的只是访问字典和（可能）启动异步操作（通过调用Task.Run）。如果两个线程同时用相同的ID调用此方法，它们最终都将等待相同的Task，这正是我们想要的结果。

#### 不可变对象(Immutable Object)

不可变对象是指其状态不能在外部或内部更改的对象。不可变对象中的字段通常被声明为只读，并在构造过程中完全初始化。

不可变性是函数式编程的一个标志——在函数式编程中，您不改变对象，而是创建一个具有不同属性的新对象。LINQ遵循这一范式。不可变性在多线程中也很有价值，因为它通过消除（或最小化）可写状态来避免共享可写状态的问题。

一种模式是使用不可变对象来封装一组相关字段，以最小化锁定持续时间。举一个非常简单的例子，假设我们有两个字段，如下所示：

int \_percentComplete;

string \_statusMessage;

现在让我们假设我们想要以原子的方式读写它们。我们可以定义以下不可变类，而不是锁定这些字段：

class ProgressStatus // Represents progress of some activity

{

public **readonly** int PercentComplete;

public **readonly** string StatusMessage;

// This class might have many more fields...

public ProgressStatus (int percentComplete, string statusMessage)

{

PercentComplete = percentComplete;

StatusMessage = statusMessage;

}

}

然后，我们可以定义该类型的单个字段以及锁定对象：

readonly object \_statusLocker = new object();

ProgressStatus \_status;

我们现在可以读取和写入该类型的值，而无需为多个赋值语句而持有锁：（即加锁只对应一条语句）

var status = new ProgressStatus (50, "Working on it");

// Imagine we were assigning many more fields...

// ...

**lock (\_statusLocker) \_status = status; // Very brief lock 相对简单的锁，它只加一条语句**

要读取对象，我们首先获取对象引用的副本（在锁中）。然后，我们可以读取其值，而无需持有锁：

ProgressStatus status;

lock (\_statusLocker) status = \_status; // Again, a brief lock 一样，也只加锁一条语句

int pc = status.PercentComplete;

string msg = status.StatusMessage;

...

### 非排它锁（Nonexclusive Locking）

非独占锁定构造用于限制并发性。在本节中，我们将介绍信号量和读/写锁，并说明SemaphoreSlim类如何限制异步操作的并发性。

#### 信号（Semaphore）

信号灯就像一个容量有限的夜总会，由保镖强制执行。当俱乐部座无虚席时，其它人都不可以进入，只能在外面排起一个队列，等待信号后准入。

信号量的计数对应于夜总会中的空间数量。释放信号量会增加计数；这通常发生在有人离开俱乐部时（对应于释放的资源），也发生在信号量初始化时（设置其启动容量）。您也可以随时调用Release以增加容量。

等待信号量会减少计数，通常发生在获得资源之前。对当前计数大于0的信号量调用Wait方法会立即完成（返回）。

信号量可以有一个最大计数作为硬限制。计数增加超过此限制会引发异常。在构造信号量时，可以指定初始计数（起始容量），也可以指定最大限制。

初始计数为1的信号量相当于 Mutex或Lock，只是信号量没有“所有者”——它与线程无关。任何线程都可以在信号量上调用Release，而使用Mutex和Lock，只有获得锁的线程才能释放它。

提示：

这儿有两个功能相似的版本：Semaphore和SemaphoreSlim。后者经过优化以满足并行编程的低延迟要求。它在传统的多线程中也很有用，因为它允许您在等待时指定取消令牌（请参阅第681页的“取消”），并且它公开了用于异步编程的WaitAsync方法。但是，您不能将它用于进程间信令。

信号量在调用WaitOne和Release时会花费大约一微秒；SemaphoreSlim引起了大约十分之一的开销。

信号量在限制并发性方面很有用——防止过多的线程同时执行特定的代码。在以下示例中，五个线程试图进入一个一次只允许三个线程进入的夜总会：

class TheClub // No door lists!

{

static SemaphoreSlim \_sem = new SemaphoreSlim (3); // Capacity of 3

static void Main()

{

for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);

}

static void Enter (object id)

{

Console.WriteLine (id + " wants to enter");

**\_sem.Wait();**

Console.WriteLine (id + " is in!"); // Only three threads

Thread.Sleep (1000 \* (int) id); // can be here at

Console.WriteLine (id + " is leaving"); // a time.

**\_sem.Release();**

}

}

1 wants to enter

1 is in!

2 wants to enter

2 is in!

3 wants to enter

3 is in!

4 wants to enter

5 wants to enter

1 is leaving

4 is in!

2 is leaving

5 is in!

实例化初始计数（容量）为0的信号量，然后调用Release来增加其计数也是合法的。以下两个信号量是等效的：

var semaphore1 = new SemaphoreSlim (3);

var semaphore2 = new SemaphoreSlim (0); semaphore2.Release (3);

信号量，如果命名，可以以与Mutex相同的方式跨越进程（命名的信号量仅在Windows上可用，而命名的Mutex也在Unix平台上工作）。

#### 异步信号和锁（Asynchronous semaphores and locks）

如下如示，锁定await语句是非法的：

lock (\_locker)

{

await Task.Delay (1000); // Compilation error

...

}

这样做毫无意义，因为锁是由线程持有的，线程通常在等待返回时会发生变化。锁定也会阻塞，而阻塞可能很长一段时间正是异步函数无法实现的。

然而，有时还是希望异步操作按顺序执行，或者限制并行性，使一次执行的操作不超过n个。例如，考虑一个web浏览器：它需要并行执行异步下载，但它可能希望施加一个限制，一次最多可以进行10次下载。我们可以通过使用SemaphoreSlim来实现这一点：

SemaphoreSlim \_semaphore = new SemaphoreSlim (10);

async Task<byte[]> DownloadWithSemaphoreAsync (string uri)

{

await \_semaphore.WaitAsync();

try { return await new WebClient().DownloadDataTaskAsync (uri); }

finally { \_semaphore.Release(); }

}

将信号量的initialCount减少到1将最大并行度减少到1，从而将其转换为异步锁。

#### Writing an EnterAsync extension method

以下扩展方法通过使用我们在第585页的“匿名处置”中所写的一次性类，简化了SemaphoreSlim的异步使用：

public static async Task<IDisposable> EnterAsync (this SemaphoreSlim ss)

{

await ss.WaitAsync().ConfigureAwait (false);

return Disposable.Create (() => ss.Release());

}

使用此方法，我们可以重写DownloadWithSemaphoreAsync方法，如下所示：

async Task<byte[]> DownloadWithSemaphoreAsync (string uri)

{

using (await \_semaphore.EnterAsync())

return await new WebClient().DownloadDataTaskAsync (uri);

}

Parallel.ForEachAsync

从. NET 6起，另一种限制异步并发的方法是使用Parallel.ForEachAsync方法。假设您希望下载的URI数组中有uri，以下是如何并行下载它们，同时将并发性限制为最多10次并行下载：

await Parallel.ForEachAsync (uris,

new ParallelOptions { MaxDegreeOfParallelism = 10 },

async (uri, cancelToken) =>

{

var download = await new HttpClient().GetByteArrayAsync (uri);

Console.WriteLine ($"Downloaded {download.Length} bytes");

});

Parallel类中的其他方法适用于（计算绑定的）并行编程场景，我们在第22章中对此进行了描述。

#### Reader/Writer Locks

通常情况下，一个类型的实例对于并发读取操作是线程安全的，但对于并发更新（对于并发读取和更新）则不是线程安全的。对于文件等资源，情况也可能如此。尽管使用针对所有访问模式的简单排他锁来保护此类类型的实例通常会起到一定的作用，但如果存在许多读操作并且只是偶尔出现更新操作，它可能会不合理地限制并发。一个可能发生这种情况的例子是在业务应用程序服务器中，为其缓存常用数据，以便在静态字段中快速检索。ReaderWriterLockSlim类设计用于在这种情况下提供最大可用性锁定。

提示:

ReaderWriterLockSlim是旧的“胖”ReaderWriterLock类的替代品。后者在功能上类似，但速度慢了好几倍，并且在处理锁升级的机制中存在固有的设计缺陷。

不过，与普通锁（Monitor.Enter/Exit）相比，ReaderWriterLockSlim的速度仍然是普通锁的两倍。这样做的代价是更少的竞争（当有大量的读操作和最少的写操作时）。

对于这两个类，有两种基本类型的锁——读锁和写锁：

* 写锁通常是独占的
* 读锁与其他读锁兼容

因此，持有写锁的线程会阻止所有试图获得读或写锁的其他线程（反之亦然）。但是，如果没有线程持有写锁，那么任何数量的线程都可以同时获得读锁。

ReaderWriterLockSlim定义了以下获取和释放读/写锁的方法：

public void EnterReadLock();

public void ExitReadLock();

public void EnterWriteLock();

public void ExitWriteLock();

此外，所有EnterXXX方法都有“Try”版本，它们接受Monitor. TryEnter风格的超时参数。（如果资源被严重争用，则很容易发生超时）。ReaderWriterLock提供了类似的方法，名为AcquireXXX和ReleaseXXX。如果发生超时，它们将抛出ApplicationException，而不是返回false。

以下程序演示ReaderWriterLockSlim。三个线程连续枚举列表，另外两个线程每100毫秒向列表附加一个随机数。读锁保护列表读取器，写锁保护列表写入器：

class SlimDemo

{

static ReaderWriterLockSlim \_rw = new ReaderWriterLockSlim();

static List<int> \_items = new List<int>();

static Random \_rand = new Random();

static void Main()

{

new Thread(Read).Start();

new Thread(Read).Start();

new Thread(Read).Start();

new Thread(Write).Start("A");

new Thread(Write).Start("B");

}

static void Read()

{

while (true)

{

\_rw.EnterReadLock();

foreach (int i in \_items) Thread.Sleep(10);

\_rw.ExitReadLock();

}

}

static void Write(object threadID)

{

while (true)

{

int newNumber = GetRandNum(100);

\_rw.EnterWriteLock();

\_items.Add(newNumber);

\_rw.ExitWriteLock();

Console.WriteLine("Thread " + threadID + " added " + newNumber);

Thread.Sleep(100);

}

}

static int GetRandNum(int max) { lock (\_rand) return \_rand.Next(max); }

}

ReaderWriterLockSlim允许比简单锁更多的并发读取活动。我们可以通过在Write方法中while循环的开头处插入以下行来说明这一点：

Console.WriteLine (\_rw.CurrentReadCount + " concurrent readers");

这几乎总是输出“3个并发读取器”（Read方法的大部分时间都花在foreach循环中）。除了CurrentReadCount，ReaderWriter LockSlim还提供了以下用于监视锁的属性：

public bool IsReadLockHeld { get; }

public bool IsUpgradeableReadLockHeld { get; }

public bool IsWriteLockHeld { get; }

public int WaitingReadCount { get; }

public int WaitingUpgradeCount { get; }

public int WaitingWriteCount { get; }

public int RecursiveReadCount { get; }

public int RecursiveUpgradeCount { get; }

public int RecursiveWriteCount { get; }

#### Upgradeable locks

有时，在单个原子操作中将读锁交换为写锁是很有用的。例如，假设您只想在项不存在的情况下将该项添加到列表中。理想情况下，您希望最大限度地减少持有（独占）写锁所花费的时间，因此可以按以下方式进行操作：

1. 获取一个读锁
2. 测试 “项”是否已在列表中, 如果存在, 释放锁然后返回
3. 释放读锁
4. 获取写锁
5. 添加”项”到列表

问题是，另一个线程可能会在步骤3和4之间偷偷进入并修改列表（例如，添加相同的项目）。ReaderWriterLockSlim通过第三种称为可升级锁的锁来解决这一问题。可升级锁与读锁类似，只是后来可以在原子操作中将其升级为写锁。以下是使用方法：

* 1. 调用EnterUpgradeableReadLock.
  2. 执行read-based活动（例如，测试项目是否已存在于列表）。
  3. 调用EnterWriteLock(这将转换可更新的锁为一个写锁)
  4. 调用ExitWriteLock(这将转换写锁为一个可更新的锁)
  5. 执用其它read-base活动
  6. 调用ExitUpgradeableReadLock

从调用者的角度来看，这更像是嵌套或递归锁定。不过，从功能上讲，在步骤3中，ReaderWriterLockSlim会自动释放您的读锁并获得新的写锁。

可升级锁和读取锁之间还有另一个重要区别。尽管可升级锁可以与任意数量的读取锁共存，但一次只能取出一个可升级锁。这可以通过序列化竞争转换来防止转换死锁——就像更新锁在SQL Server中所做的那样：

SQL Server ReaderWriterLockSlim

Share lock Read lock

Exclusive lock Write lock

Update lock Upgradeable lock

我们可以通过更改上例中的Write方法来演示可升级锁，使其仅在列表中不存在数字的情况下才添加数字：

while (true)

{

int newNumber = GetRandNum (100);

\_rw.EnterUpgradeableReadLock();

if (!\_items.Contains (newNumber))

{

\_rw.EnterWriteLock();

\_items.Add (newNumber);

\_rw.ExitWriteLock();

Console.WriteLine ("Thread " + threadID + " added " + newNumber);

}

\_rw.ExitUpgradeableReadLock();

Thread.Sleep (100);

}

提示:

ReaderWriterLock也可以进行锁转换，但不可靠，因为它不支持可升级锁的概念。这就是ReaderWriterLockSlim的设计者不得不用一个新类重新开始的原因。

#### Lock recursion

通常，ReaderWriterLockSlim禁止嵌套或递归锁定。因此，以下抛出一个异常：

var rw = new ReaderWriterLockSlim();

rw.EnterReadLock();

rw.EnterReadLock();

rw.ExitReadLock();

rw.ExitReadLock();

如果像下面构造ReaderWriterLockSlim, 它会无错的运行.

var rw = new ReaderWriterLockSlim (**LockRecursionPolicy.SupportsRecursion**);

这确保了递归锁定只有在您计划的情况下才能发生。递归锁定可能会产生不必要的复杂性，因为可能会获得多种类型的锁：

rw.EnterWriteLock();

rw.EnterReadLock();

Console.WriteLine (rw.IsReadLockHeld); // True

Console.WriteLine (rw.IsWriteLockHeld); // True

rw.ExitReadLock();

rw.ExitWriteLock();

基本规则是，在获得锁后，后续的递归锁可以更少，但不能更大，其规模如下：

Read Lock→Upgradeable Lock→Write Lock

不过，将可升级锁升级为写锁的请求始终是合法的。