C# 温故而知新： 线程篇（三）线程同步篇 （上）

1.线程同步中的一些重要概念

1.1**临界区(共享区)的概念**

       在多线程的环境中，可能需要共同使用一些公共资源，这些资源可能是变量，方法逻辑段等等，这些被多个线程共用的区域统称为临界区(共享区)，聪明的你肯定会想到，临界区的资源不是很安全，因为线程的状态是不定的，所以可能带来的结果是临界区的资源遭到其他线程的破坏，我们必须采取策略或者措施让共享区数据在多线程的环境下保持完成性不让其受到多线程访问的破坏

**1.2基元用户模式**

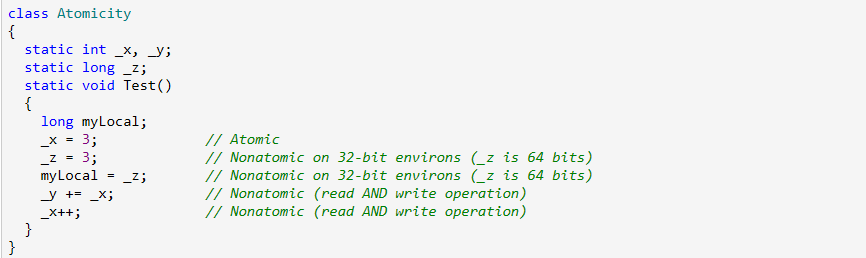
       可能大家觉得这个很难理解，的确如果光看概念解释的话，会让人抓狂的，因为这个模式牵涉到了深奥的底层cup内核和windows的一些底层机制，所以我用最简单的理解相信大家一定能理解，因为这对于理解同步也很重要回到正题，基元用户模式是指使用cpu的特殊指令来调度线程，所以这种协调调度线程是在硬件中进行的所以得出了它第一些优点：速度特别快、 线程阻塞时间特别短。但是由于该模式中的线程可能被系统抢占，导致该模式中的线程为了获取某个资源，而浪费许多cpu时间，同时如果一直处于等待的话会导致”活锁”，也就是既浪费了内存，又浪费了cpu时间，这比下文中的死锁更可怕，那么如何利用强大的cpu时间做更多的事呢？那就引出了下面的一个模式

**1.3基元内核模式**

该模式和用户模式不同，它是windows系统自身提供的，使用了操作系统中内核函数，所以它能够阻塞线程提高了cpu的利用率，同时也带来了一个很可怕的bug,死锁，可能线程会一直阻塞导致程序的奔溃,常用的内核模式的技术例如Monitor,Mutex,等等会在下一章节介绍。本章将详细讨论锁的概念，使用方法和注意事项

**\*1.4原子性操作**

如果一个语句执行一个单独不可分割的指令，那么它是原子的。严格的原子操作排除了任何抢占的可能性（这也是实现同步的一个重要条件，也就是说没有一个线程可以把这个美女占为己有，更方便的理解是这个值永远是最新的），在c#中原子操作如下图所示：



其实要符合原子操作必须满足以下条件

1. c#中如果是32位cpu的话，**为一个少于等于32位字段赋值是原子操作**，其他（自增，读，写操作）的则不是
2. 对于64位cpu而言，**操作32或64位的字段赋值**都属于原子操作
3. 其他读写操作都不能属于原子操作

相信大家能够理解原子的特点，下文中的Volatil和interlocked会详细模拟原子操作来实现线程同步，所以在使用原子操作时也需要注意当前操作系统是32位或是64位cpu或者两者皆要考虑

**1.5非阻止同步**

非阻止同步说到底，就是利用**原子性**操作实现线程间的同步，不刻意阻塞线程，减少相应线程的开销，下文中的VolatileRead,VolatileWrite,Volatile关键字，interlocked类便是c#中非阻止同步的理念所产生的线程同步技术

**1.6阻止同步**

       阻止同步正好相反，其实阻止同步也是基元内核模式的特点之一，例如c# 中的锁机制，及其下几章介绍的mutex,monitor等都属于阻止同步，他们的根本目的是，以互斥的效果让同一时间只有一个线程能够访问共享区，其他线程必须阻止等待，直到该线程离开共享区后，在让其他一个线程访问共享区，阻止同步缺点也是容易产生死锁，但是阻止同步提高了cpu时间的利用率

2.详解Thread类中的VolatileRead和VolatileWrite方法和Volatile关键字

     前文中，我们已经对原子操作和非阻止同步的概念已经有了大概的认识，接着让我们从新回到Thread类来看下其中比较经典的VolatileRead和VolatileWrite方法

 VolatileWrite： 该方法作用是，当线程在共享区（临界区）传递信息时，通过此方法来**原子性的写入最后**一个值

VolatileRead:  该方法作用是，当线程在共享区（临界区）传递信息时，通过此方法来**原子性的读取第一个值**。

 可能这样的解释会让大家困惑，老规矩，直接上例子让大家能够理解：

/// <summary>本例利用VolatileWrite和VolatileRead来实现同步，来实现一个计算的例子，每个线程负责运算1000万个数据，共开启10个线程计算至1亿，而且每个线程都无法干扰其他线程工作</summary>

class Program{

static Int32 count;//计数值，用于线程同步 （注意原子性，所以本例中使用int32）

static Int32 value;//实际运算值，用于显示计算结果

static void Main(string[] args){

//开辟一个线程专门负责读value的值，这样就能看见一个计算的过程

Thread thread2 = new Thread(new ThreadStart(Read));

thread2.Start();

//开辟10个线程来负责计算，每个线程负责1000万条数据

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

Thread.Sleep(20);

Thread thread = new Thread(new ThreadStart(Write));

thread.Start();

}

Console.ReadKey();

}

/// <summary>实际运算写操作</summary>

private static void Write(){

Int32 temp = 0;

for (int i = 0; i < 10000000; i++){

temp += 1;

}

value += temp;

//注意VolatileWrite 在每个线程计算完毕时会写入同步计数值为1，告诉程序该线程已经执行完毕

//所以VolatileWrite方法类似与一个按铃，往往在原子性的最后写入告诉程序我完成了

Thread.VolatileWrite(ref count, 1);

}

/// <summary>显示计算后的数据，使用该方法的线程会死循环等待写操作的线程发出完毕信号后显示当前计算结果</summary>

private static void Read(){

while (true){

//一旦监听到一个写操作线执行完毕后立刻显示操作结果和VolatileWrite相反，VolatileRead类似一个门禁，只有原子性的最先读取他，才能达到同步效果，同时count值保持最新

if (Thread.VolatileRead(ref count) > 0){

Console.WriteLine("累计计数:{1}", Thread.CurrentThread.ManagedThreadId, value);

//将count设置成0，等待另一个线程执行完毕

count = 0;

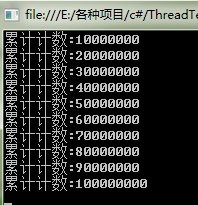
}

}

}

}

显示结果：



例子中我们可以看出当个线程调用Read方法时，代码会先判断Thread. VolatileRead**先读取计数值**是否返回正确的计数值，如果正确则显示结果，不正确的话继续循环等待，而这个返回值是通过其他线程操作Write方法时**最后**写入的，也就是说对于Thread. VolatileWrite方法的作用便一目了然了，在实现Thread. VolatileWrite**前**写入其他的数据或进行相应的逻辑处理，在我们示例代码中我们会先去加运算到10000000时，通过thread. VolatileWrite原子性的操作写入计数值告诉那个操作Read方法的线程有一个计算任务已经完成，于是死循环中的Thread. VolatileRead方法接受到了信号，你可以显示计算结果了，于是结果便会被显示，同时计数值归零，这样便起到了一个非阻塞功能的同步效果，同样对于临界区（此例中的Write方法体和Read方法体）起到了保护的作用。当然由于使用上述两个方法在复杂的项目中很容易出错，往往这种错误是很难被发现，所以微软为了让我们更好使用，便开发出了一个新的关键字**Volatile**：

Volatile关键字的作用：Volatile关键字的本质含义是告诉编译器，声明为Volatile关键字的变量或字段都是提供给多个线程使用的，当然不是每个类型都可以声明为Volatile类型字段，msdn中详细说明了那些类型可以声明为Volatile 所以不再陈述，但是有一点必须注意，Volatile无法声明为局部变量。作为原子性的操作，Volatile关键字具有原子特性，所以线程间无法对其占有，它的值永远是最新的。那我们就对上文的那个例子简化如下：

/// <summary>本例利用volatile关键字来实现同步，来实现一个计算的例子，每个线程负责运算1000万个数据，共开启10个线程计算至1亿， 而且每个线程都无法干扰其他线程工作 </summary>

class Program{

static volatile Int32 count;//计数值，用于线程同步 （注意原子性，所以本例中使用int32）

static Int32 value;//实际运算值，用于显示计算结果

static void Main(string[] args){

//开辟一个线程专门负责读value的值，这样就能看见一个计算的过程

Thread thread2 = new Thread(new ThreadStart(Read));

thread2.Start();

//开辟10个线程来负责计算，每个线程负责1000万条数据

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

Thread.Sleep(20);

Thread thread = new Thread(new ThreadStart(Write));

thread.Start();

}

Console.ReadKey();

}

/// <summary>/// 实际运算写操作/// </summary>

private static void Write(){

Int32 temp = 0;

for (int i = 0; i < 10000000; i++){

temp += 1;

}

value += temp;

//注意VolatileWrite 在每个线程计算完毕时会写入同步计数值为1，告诉程序该线程已经执行完毕，将count值设置成1，效果等同于Thread.VolatileWrite

count = 1;

}

/// <summary>显示计算后的数据，使用该方法的线程会死循环等待写操作的线程发出完毕信号后显示当前计算结果</summary>

private static void Read(){

while (true){

//一旦监听到一个写操作线执行完毕后立刻显示操作结果,效果等同于Thread.VolatileRead

if (count==1){

Console.WriteLine("累计计数:{1}", Thread.CurrentThread.ManagedThreadId, value);

//将count设置成0，等待另一个线程执行完毕

count = 0;

}

}

}

}

            从例子中大家可以看出Volatile关键字的出现替代了原先VolatileRead 和VolatileWrite方法的繁琐，同时原子性的操作更加直观透明

3.详解Interlocked

     相信大家理解了Volatile后对于非阻止同步和原子操作有了更深的认识，接下来的Interlocked虽然也属于非阻止同步但是而后Volatile相比也有着很大的不同，interlocked 利用了一个**计数值**的概念来实现同步，当然这个计数值也是属于原子性的操作，每个线程都有机会通过Interlocked去递增或递减这个计数值来达到同步的效果,同时Interlocked比Volatile更加适应复杂的逻辑和并发的情况首先让我们了解下Interlocked类的一些重要方法

static long Read() ：以原子操作形式读取计数值，该方法能够读取当前计数值，但是如果是64位cpu的可以不需要使用该方法读取.

\***但是如果是32位的cpu则必须使用interlocked类的方法对64位的变量进行操作来保持原子操作**，否则就不是原子操作

static int or long Increment(Int32 Or Int64)：该方法已原子操作的形式递增指定变量的值并存储结果，也可以理解成以原子的操作对计数器加1。Increment有2个返回类型的版本，分别是int 和 long

static int or long  Decrement(Int32 Or Int64)：和Increment方法相反，该方法已原子操作的形式递减指定变量的值并存储结果，也可以理解成以原子的操作对计数器减1，同样，Decrement也有2个返回类型的版本，分别是int 和 long

static int Add(ref int location1,int value)：该方法是将Value的值和loation1中的值相加替换location1中原有值并且存储在locaion1中，注意，该方法不会抛出溢出异常，如果location中的值和Value之和大于int32.Max则，location1中的值会变成int32.Min和Value之和

Exchange(double location1,double value)：Exchange方法有多个重载，但是使用方法是一致的，以原子操作的形式将Value的值赋值给location1

看完了概念性的介绍后，让我们马上进入很简单的一个示例，来深刻理解下Interlocked的使用方法

/// <summary>本示例通过Interlocked实现同步示例，通过Interlocked.Increment和Interlocked.Decrement来实现同步，此例有2个共享区，一个必须满足计数值为0，另一个满足计数值为1时才能进入 </summary>

class Program{

//声明计数变量（注意这里用的是long是64位的，所以在32位机子上一定要通过Interlocked来实现原子操作）

static long \_count = 0;

static void Main(string[] args){

//开启6个线程，3个执行Excution1，三个执行Excution2

for (int i = 0; i < 3; i++){

Thread thread = new Thread(new ThreadStart(Excution1));

Thread thread2 = new Thread(new ThreadStart(Excution2));

thread.Start();

Thread.Sleep(10);

thread2.Start();

Thread.Sleep(10);

}

//这里和同步无关，只是简单的对Interlocked方法进行示例

Interlocked.Add(ref \_count, 2);

Console.WriteLine("为当前计数值加上一个数量级:{0}后，当前计数值为:{1}", 2, \_count);

Interlocked.Exchange(ref \_count, 1);

Console.WriteLine("将当前计数值改变后，当前计数值为:{0}", \_count);

Console.Read();

}

static void Excution1(){

//进入共享区1的条件

if (Interlocked.Read(ref \_count) == 0){

Console.WriteLine("Thread ID:{0} 进入了共享区1", Thread.CurrentThread.ManagedThreadId);

//原子性增加计数值，让其他线程进入共享区2

Interlocked.Increment(ref \_count);

Console.WriteLine("此时计数值Count为:{0}", Interlocked.Read(ref \_count));

}

}

static void Excution2(){

//进入共享区2的条件

if (Interlocked.Read(ref \_count) == 1){

Console.WriteLine("Thread ID:{0} 进入了共享区2", Thread.CurrentThread.ManagedThreadId);

//原子性减少计数值，让其他线程进入共享区1

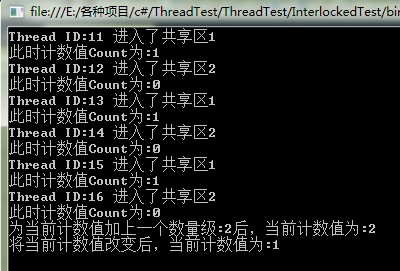
Interlocked.Decrement(ref \_count);

Console.WriteLine("此时计数值Count为:{0}", Interlocked.Read(ref \_count));

}

}

}



在本例中，我们使用和上文一样的思路，通过不同线程来原子性的操作计数值来达到同步效果，大家可以仔细观察到，通过Interlocked对计数值进行操作就能够让我们非常方便的使用非阻止的同步效果了，但是在复杂的项目或逻辑中，可能也会出错导致活锁的可能，大家务必当心

4.介绍下Lock关键字

Lock关键字是用来对于多线程中的共享区进行阻止同步的一种方案，当某一个线程进入临界区时，lock关键字会锁住共享区,同样可以理解为互斥段，互斥段在某一时刻内只允许一个线程进入，同时编译器会把这个关键字编译成Monitor.Entery和Monitor.Exit 方法，关于Monitor类会在下章详细阐述。既然有Lock关键字，那么它是如何工作的？到底锁住了什么，怎么高效和正确的使用lock关键字呢？

   其实锁的概念还是来自于现实生活，共享区就是多个人能够共同拥有房间，当其中一个人进入房间后，他把锁反锁，直到他解锁出门后将钥匙交给下个人，可能房间的门可能有问题，或者进入房间的人因为某种原因出不来了，导致全部的人都无法进去，这些问题也是我们应该考虑到的，好，首先让我们讨论下我们应该Lock住什么，什么材料适合当锁呢？虽然说lock关键字可以锁住任何object类型及其派生类，但是尽量不要用public 类型的，因为public类型难以控制有可能大伙对上面的有点疑问，为什么不能用public类型的呢，为什么会难以控制呢？好，以下3个例子是比较经典的例证

 1.Lock(this):大伙肯定会知道this指的是当前类对象,Lock(this) 、意味着将当前类对象给锁住了，假设我需要同时使用这个类的别的方法，那么某一线程一旦进入临界区后，那完蛋了，该类所有的成员（方法）都无法访问，这可能在某些时刻是致命的错误

2.同理Lock(typeof(XXX)) 更厉害，一方面对锁的性能有很大影响，因为一个类型太大了，其次，当某一线程进入临界区后，包括所有该类型的type都可能会被锁住而产生死锁

3.最严重的某过于锁住字符串对象，lock(“Test”),c#中的字符串对象很特殊，string  test=”Test” 和 string test2=”Test” 其实是一个对象，假如你使用了lock(“Test”)那么，所有字符串值为"Test"的字符串都有可能被锁住，甚至造成死锁，所以有些奇怪的bug都是因为一些简单的细节导致

接着这个例子便是lock（this）的一个示例，既能让大伙了解如何使用Lock关键字，更是让大伙了解，lock(this)的危害性

/// <summary>/// 本例展示下如何使用lock关键字和lock(this)时产生死锁的情况/// </summary>

class Program{

static void Main(string[] args){

//创建b对象，演示lock

B b = new B();

Console.ReadKey();

}

}

/// <summary>A类构造中初始化一个线程并且启动，线程调用的方法内放入死循环，并且在死循环中放入lock(this)，</summary>

public class A{

public A(){

Thread th = new Thread(new ThreadStart(() =>{

while (true){

lock (this){

Console.WriteLine("进入a类共享区");

Thread.Sleep(3000);

}

}

}

));

th.Start();

}

}

/// <summary>B类在构造中创建A的对象，并且还是锁住a对象，这样就创建的死锁的条件,因为初始化A类对象时，A类的构造函数会锁住自身对象，这样在A类死循环间隔期，一旦出了 A类中的锁时进入B的锁住的区域内，A 对象永远无法进入a类共享区，从而产生了死锁</summary>

public class B{

public B(){

A a = new A();

lock (a){

Console.WriteLine(@"将a类对象锁住的话，a类中的lock将进入死锁，

直到3秒后B类中的将a类对象释放锁,如果我不释放，那么 a类中将永远无法进入a类共享区");

//计时器

Timer timer = new Timer(new TimerCallback((obj) => { Console.WriteLine(DateTime.Now); }), this, 0, 1000);

//如果这里运行很长时间， a类中将永远无法进入a类共享区

Thread.Sleep(3000000);

}

}

}

结果计时器会一直滚动，因为a对象被锁住，除非完成Thread.Sleep(3000000)后才能进入到a共享区



由于以上的问题，微软还是建议我们使用一个**私有的变量**来锁定，由于私有变量外界无法访问，所以锁住话死锁的可能性大大下降了。这样我们就能选择正确的“门”来进行锁住，但是可能还有一种可能也会造成死锁，就是在lock内部出现了问题，由于死锁非常复杂，我将在今后的文章中专门**写一篇关于死锁**的文章来深入解释下死锁，所以这里就对死锁不深究了，这里大伙了解下lock的使用方法和注意事项就行了。

5.ReaderWriterLock

由于lock关键字对临界区（共享区）保护的非常周密，导致了一些功能可能会无法实现，假设我将某个查询功能放置在临界区中时，可能当别的线程在查询临界区中的数据时，可能我的那个线程被阻塞了，所以我们期望锁能够达到以下功能

1 首先锁能细分为读锁和写锁

2 能够保证同时可以让多个线程读取数据

3 能保证同一时刻只有一个线程能进行写操作，也就是说，对于写操作，它必须拥有独占锁

4 能保证**一个线程同一时刻只能拥有写锁或读锁中的一个**

显然lock关键字无法满足我们的需求，还好微软想到了这点，ReaderWriterLock便隆重登场了ReaderWriterLock能够达到的效果是：

1. 同一时刻，它允许多个读线程同时访问临界区，或者允许单个线程进行写访问

2. 在读访问率很高，而且写访问率很低的情况下，效率最高，

3．它也满足了同一时刻只能获取写锁或读锁的要求。

4. 最为关键的是，ReaderWriterLock能够保证读线程锁和写线程锁在各自的读写队列中，当某个线程释放了写锁了，同时读线程队列中的所有线程将被授予读锁，同样，当所有的读锁被释放时，写线程队列中的排队的下一个线程将被授予写锁,更直观的说，ReaderWriterLock就是在这几种状态间来回切换

5 使用时注意每当你使用**AcquireXXX方法获取锁时，必须使用ReleaseXXX方法来释放锁**

6 ReaderWriterLock 支持递归锁，关于递归锁会在今后的章节详细阐述

7 在性能方面ReaderWriterLock做的不够理想，和lock比较差距明显，而且该类库中还隐藏些bug,有于这些原因，微软又专门重新写了个新

类**ReaderWriterLockSilm**来弥补这些缺陷。

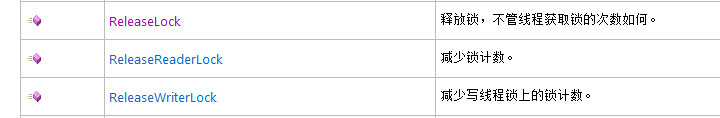
8 处理死锁方面ReaderWriterLock为我们提供了超时的参数这样我们便可以有效的防止死锁

9 对于一个个获取了读锁的线程来说，在写锁空闲的情况下可以升级为写锁

接着让我们了解下ReaderWriterLock的重要成员



上述4个方法分别是让线程获取写锁和读锁的方法，它利用的计数的概念，当一个线程中调用此方法后，该类会给该线程拥有的锁计数加1（每次加1，但是一个线程可以拥有多个读锁，所以计数值可能更多，但是对于写锁来说同时一个一个线程可以拥有）。后面的参数是超时时间，我们可以自己设置来避免死锁。同样调用上述方法后我们必须使用ReleaseXXX 方法来让计数值减1，直到该线程拥有锁的计数为0，释放了锁为止。



 最后我们用一个简单的例子来温故下上述的知识点(请注意看注释)

/// <summary>该示例通过ReaderWriterLock同步来实现Student集合多线程下 的写操作和读操作</summary>

class Program{

static ReaderWriterLock \_readAndWriteLock = new ReaderWriterLock();

static List<Student> demoList = new List<Student>();

static void Main(string[] args){

InitialStudentList();

Thread thread=null;

for (int i = 0; i <5; i++){

//让第前2个个线程试图掌控写锁,

if (i < 2){

thread = new Thread(new ParameterizedThreadStart(AddStudent));

Console.WriteLine("线程ID:{0}, 尝试获取写锁 ", thread.ManagedThreadId);

thread.Start(new Student { Name = "Zhang" + i });

}else {

//让每个线程都能访问DisplayStudent 方法去获取读锁

thread = new Thread(new ThreadStart(DisplayStudent));

thread.Start();

}

Thread.Sleep(20);

}

Console.ReadKey();

}

static void InitialStudentList() {

demoList = new List<Student> { new Student{ Name="Sun"}, new Student{Name="Zheng"} };

}

/// <summary>当多个线程试图使用该方法时，只有一个线程能够透过AcquireSWriterLock获取写锁，同时其他线程进入队列中等待，直到该线程使用ReleaseWriterLock后下个线程才能进入拥有写锁</summary>

static void AddStudent(object student){

if (student == null|| !(student is Student)) return;

if (demoList.Contains(student)) return;

try{

//获取写锁

\_readAndWriteLock.AcquireWriterLock(Timeout.Infinite);

demoList.Add(student as Student);

Console.WriteLine("当前写操作线程为{0},写入的学生是：{1}", Thread.CurrentThread.ManagedThreadId,(student as Student).Name);

}

catch (Exception){}

finally{\_readAndWriteLock.ReleaseWriterLock();}

}

/// <summary>对于读锁来所，允许多个线程共同拥有，所以这里同时可能会有多个线程访问Student集合，使用try catch是为了

/// 一定要让程序执行finally语句块中的releaseXXX方法，从而保证能够释放锁</summary>

static void DisplayStudent(){

try{

\_readAndWriteLock.AcquireReaderLock(Timeout.Infinite);

demoList.ForEach(student=>{Console.WriteLine("当前集合中学生为:{0},当前读操作线程为{1}", student.Name, Thread.CurrentThread.ManagedThreadId);});

}

catch (Exception){}

finally{\_readAndWriteLock.ReleaseReaderLock();}

}

}

internal class Student{

public string Name { get; set; }

}

 运行结果：



从例子可以看出有2个线程试图尝试争取写锁，但是同时只有一个线程可以获取到写锁，同时对于读取集合的线程可以同时获取多个读锁

7.  参考资料

 CLR via c#

 msdn