[**归纳一下:C#线程同步的几种方法**](http://www.cnblogs.com/michaelxu/archive/2008/09/20/1293716.html)

　　我们在编 程的时候，有时会使用多线程来解决问题，比如你的程序需要在后台处理一大堆数据，但还要使用户界面处于可操作状态；或者你的程序需要访问一些外部资源如数 据库或网络文件等。这些情况你都可以创建一个子线程去处理，然而，多线程不可避免地会带来一个问题，就是线程同步的问题。如果这个问题处理不好，我们就会 得到一些非预期的结果。

　　在网上也看过一些关于线程同步的文章，其实线程同步有好几种方法，下面我就简单的做一下归纳。

**一、volatile关键字**

　　volatile是最简单的一种同步方法，当然简单是要付出代价的。它只能在变量一级做同步，volatile的含义就是告诉处理器， 不要将我放入工作内存， 请直接在主存操作我。（【转自[www.bitsCN.com](http://www.bitsCN.com) 】）因此，当多线程同时访问该变量时，都将直接操作主存，从本质上做到了变量共享。

　　能够被标识为volatile的必须是以下几种类型：（摘自MSDN）

 Any reference type.

 Any pointer type (in an unsafe context).

 The types sbyte, byte, short, ushort, int, uint, char, float, bool.

 An enum type with an enum base type of byte, sbyte, short, ushort, int, or uint.

　　如：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gif Code   
public class A  
{  
private volatile int \_i;  
public int I  
{  
get { return \_i; }  
set { \_i = value; }  
}  
}

[复制代码](javascript:void(0);)

　　但volatile并不能实现真正的同步，因为它的操作级别只停留在变量级别，而不是原子级别。如果是在单处理器系统中，是没有任何问题的，变量在主存中没有机会被其他人修改，因为只有一个处理器，这就叫作processor Self-Consistency。但在多处理器系统中，可能就会有问题。 每个处理器都有自己的data cach，而且被更新的数据也不一定会立即写回到主存。所以可能会造成不同步，但这种情况很难发生，因为cach的读写速度相当快，flush的频率也相当高，只有在压力测试的时候才有可能发生，而且几率非常非常小。

**二、lock关键字**

　　lock是一种比较好用的简单的线程同步方式，它是通过为给定对象获取互斥锁来实现同步的。它可以保证当一个线程在关键代码段的时候，另一个线程不会进来，它只能等待，等到那个线程对象被释放，也就是说线程出了临界区。用法：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gif Code   
public void Function()   
{  
object lockThis = new object ();   
lock (lockThis)  
{  
// Access thread-sensitive resources.   
}  
}

[复制代码](javascript:void(0);)

　　lock的参数必须是基于引用类型的对象，不要是基本类型像bool,int什么 的，这样根本不能同步，原因是lock的参数要求是对象，如果传入int，势必要发生装箱操作，这样每次lock的都将是一个新的不同的对象。最好避免使 用public类型或不受程序控制的对象实例，因为这样很可能导致死锁。特别是不要使用字符串作为lock的参数，因为字符串被CLR“暂留”，就是说整 个应用程序中给定的字符串都只有一个实例，因此更容易造成死锁现象。建议使用不被“暂留”的私有或受保护成员作为参数。其实某些类已经提供了专门用于被锁 的成员，比如Array类型提供SyncRoot，许多其它集合类型也都提供了SyncRoot。

　　所以，使用lock应该注意以下几点：

　　１、如果一个类的实例是public的，最好不要lock(this)。因为使用你的类的人也许不知道你用了lock，如果他new了一个实例，并且对这个实例上锁，就很容易造成死锁。

　　２、如果MyType是public的，不要lock(typeof(MyType))

　　３、永远也不要lock一个字符串

**三、System.Threading.Interlocked**

　　对于整数数据类型的简单操作，可以用 Interlocked 类的成员来实现线程同步，存在于System.Threading命名空间。Interlocked类有以下方法：Increment , Decrement , Exchange 和CompareExchange 。使用Increment 和Decrement 可以保证对一个整数的加减为一个原子操作。Exchange 方法自动交换指定变量的值。CompareExchange 方法组合了两个操作：比较两个值以及根据比较的结果将第三个值存储在其中一个变量中。比较和交换操作也是按原子操作执行的。如：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gif Code   
int i = 0 ;  
System.Threading.Interlocked.Increment( ref i);  
Console.WriteLine(i);  
System.Threading.Interlocked.Decrement( ref i);  
Console.WriteLine(i);  
System.Threading.Interlocked.Exchange( ref i, 100 );  
Console.WriteLine(i);  
System.Threading.Interlocked.CompareExchange( ref i, 10 , 100 );

[复制代码](javascript:void(0);)

Output:



**四、Monitor**

　　Monitor类提供了与lock类似的功能,不过与lock不同的是,它 能更好的控制同步块,当调用了Monitor的Enter(Object o)方法时,会获取o的独占权,直到调用Exit(Object o)方法时,才会释放对o的独占权,可以多次调用Enter(Object o)方法,只需要调用同样次数的Exit(Object o)方法即可,Monitor类同时提供了TryEnter(Object o,[int])的一个重载方法,该方法尝试获取o对象的独占权,当获取独占权失败时,将返回false。

　　但使用 **lock** 通常比直接使用 **Monitor** 更可取，一方面是因为 **lock** 更简洁，另一方面是因为 **lock** 确保了即使受保护的代码引发异常，也可以释放基础监视器。这是通过 **finally** 中调用Exit来实现的。事实上，**lock** 就是用 Monitor 类来实现的。下面两段代码是等效的：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gif Code   
lock (x)  
{  
DoSomething();  
}

等效于  
  
object obj = ( object )x;  
System.Threading.Monitor.Enter(obj);  
try   
{  
DoSomething();  
}  
finally   
{  
System.Threading.Monitor.Exit(obj);  
}

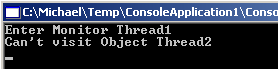
[复制代码](javascript:void(0);)

关于用法，请参考下面的代码：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gif Code   
private static object m\_monitorObject = new object ();  
[STAThread]  
static void Main( string [] args)  
{  
Thread thread = new Thread( new ThreadStart(Do));  
thread.Name = " Thread1 " ;  
Thread thread2 = new Thread( new ThreadStart(Do));  
thread2.Name = " Thread2 " ;  
thread.Start();  
thread2.Start();  
thread.Join();  
thread2.Join();  
Console.Read();  
}  
static void Do()  
{  
if ( ! Monitor.TryEnter(m\_monitorObject))  
{  
Console.WriteLine( " Can't visit Object " + Thread.CurrentThread.Name);  
return ;  
}  
try   
{  
Monitor.Enter(m\_monitorObject);  
Console.WriteLine( " Enter Monitor " + Thread.CurrentThread.Name);  
Thread.Sleep( 5000 );  
}  
finally   
{  
Monitor.Exit(m\_monitorObject);  
}  
}

[复制代码](javascript:void(0);)

　　当线程1获取了m\_monitorObject对象独占权时，线程2尝试调用TryEnter(m\_monitorObject)，此时会由于无法获取独占权而返回false，输出信息如下：  


　　另外，Monitor还提供了三个静态方法 Monitor.Pulse(Object o)，Monitor.PulseAll(Object o)和Monitor.Wait(Object o ) ，用来实现一种唤醒机制的同步。关于这三个方法的用法，可以参考MSDN，这里就不详述了。

**五、Mutex**

　　在使用上,Mutex与上述的Monitor比较接近，不过Mutex不具 备Wait，Pulse，PulseAll的功能，因此，我们不能使用Mutex实现类似的唤醒的功能。不过Mutex有一个比较大的特点，Mutex是 跨进程的，因此我们可以在同一台机器甚至远程的机器上的多个进程上使用同一个互斥体。尽管Mutex也可以实现进程内的线程同步，而且功能也更强大，但这 种情况下，还是推荐使用Monitor，因为Mutex类是win32封装的，所以它所需要的互操作转换更耗资源。

**六、ReaderWriterLock**

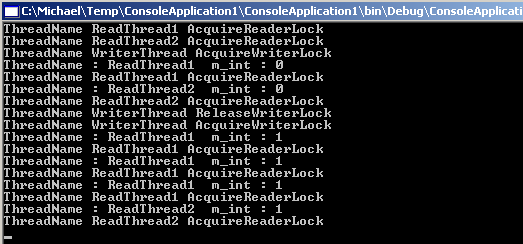
　　在考虑资源访问的时候,惯性上我们会对资源实施lock机制,但是在某些情 况下,我们仅仅需要读取资源的数据,而不是修改资源的数据,在这种情况下获取资源的独占权无疑会影响运行效率,因此.Net提供了一种机制,使用 ReaderWriterLock进行资源访问时,如果在某一时刻资源并没有获取写的独占权,那么可以获得多个读的访问权,单个写入的独占权,如果某一时 刻已经获取了写入的独占权,那么其它读取的访问权必须进行等待,参考以下代码：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gifCode  
private static ReaderWriterLock m\_readerWriterLock = new ReaderWriterLock();  
private static int m\_int = 0;  
[STAThread]  
static void Main(string[] args)  
{  
Thread readThread = new Thread(new ThreadStart(Read));  
readThread.Name = "ReadThread1";  
Thread readThread2 = new Thread(new ThreadStart(Read));  
readThread2.Name = "ReadThread2";  
Thread writeThread = new Thread(new ThreadStart(Writer));  
writeThread.Name = "WriterThread";  
readThread.Start();  
readThread2.Start();  
writeThread.Start();  
readThread.Join();  
readThread2.Join();  
writeThread.Join();  
  
Console.ReadLine();   
}  
private static void Read()  
{  
while (true)  
{  
Console.WriteLine("ThreadName " + Thread.CurrentThread.Name + " AcquireReaderLock");  
m\_readerWriterLock.AcquireReaderLock(10000);  
Console.WriteLine(String.Format("ThreadName : {0} m\_int : {1}", Thread.CurrentThread.Name, m\_int));  
m\_readerWriterLock.ReleaseReaderLock();  
}  
}  
  
private static void Writer()  
{  
while (true)  
{  
Console.WriteLine("ThreadName " + Thread.CurrentThread.Name + " AcquireWriterLock");  
m\_readerWriterLock.AcquireWriterLock(1000);  
Interlocked.Increment(ref m\_int);  
Thread.Sleep(5000);  
m\_readerWriterLock.ReleaseWriterLock();  
Console.WriteLine("ThreadName " + Thread.CurrentThread.Name + " ReleaseWriterLock");  
}  
}

[复制代码](javascript:void(0);)

在程序中,我们启动两个线程获取m\_int的读取访问权,使用一个线程获取m\_int的写入独占权,执行代码后,输出如下：



可以看到，当WriterThread获取到写入独占权后，任何其它读取的线程 都必须等待，直到WriterThread释放掉写入独占权后，才能获取到数据的访问权，应该注意的是，上述打印信息很明显显示出，可以多个线程同时获取 数据的读取权，这从ReadThread1和ReadThread2的信息交互输出可以看出。

**七、SynchronizationAttribute**

　　当我们确定某个类的实例在同一时刻只能被一个线程访问时，我们可以直接将类标识成Synchronization的，这样，CLR会自动对这个类实施同步机制,实际上，这里面涉及到同步域的概念，当类按如下设计时，我们可以确保类的实例无法被多个线程同时访问  
　　1). 在类的声明中,添加System.Runtime.Remoting.Contexts.SynchronizationAttribute属性。  
    2). 继承至System.ContextBoundObject  
    需要注意的是，要实现上述机制，类必须继承至System.ContextBoundObject，换句话说,类必须是上下文绑定的。  
    一个示范类代码如下：

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gifCode  
[System.Runtime.Remoting.Contexts.Synchronization]  
public class SynchronizedClass : System.ContextBoundObject  
{  
  
}

**八、MethodImplAttribute**

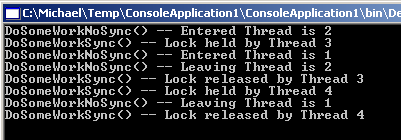
　　如果临界区是跨越整个方法的，也就是说，整个方法内部的代码都需要上锁的 话，使用MethodImplAttribute属性会更简单一些。这样就不用在方法内部加锁了，只需要在方法上面加上 [MethodImpl(MethodImplOptions.Synchronized)] 就可以了，MehthodImpl和MethodImplOptions都在命名空间System.Runtime.CompilerServices 里面。但要注意这个属性会使整个方法加锁，直到方法返回，才释放锁。因此，使用上不太灵活。如果要提前释放锁，则应该使用Monitor或lock。我们来看一个例子：

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gif Code   
[MethodImpl(MethodImplOptions.Synchronized)]  
public void DoSomeWorkSync()  
{  
Console.WriteLine( " DoSomeWorkSync() -- Lock held by Thread " +   
Thread.CurrentThread.GetHashCode());  
Thread.Sleep( 1000 );  
Console.WriteLine( " DoSomeWorkSync() -- Lock released by Thread " +   
Thread.CurrentThread.GetHashCode());  
}  
public void DoSomeWorkNoSync()  
{  
Console.WriteLine( " DoSomeWorkNoSync() -- Entered Thread is " +   
Thread.CurrentThread.GetHashCode());  
Thread.Sleep( 1000 );  
Console.WriteLine( " DoSomeWorkNoSync() -- Leaving Thread is " +   
Thread.CurrentThread.GetHashCode());  
}  
  
[STAThread]  
static void Main( string [] args)  
{  
MethodImplAttr testObj = new MethodImplAttr();  
Thread t1 = new Thread( new ThreadStart(testObj.DoSomeWorkNoSync));  
Thread t2 = new Thread( new ThreadStart(testObj.DoSomeWorkNoSync));  
t1.Start();  
t2.Start();  
Thread t3 = new Thread( new ThreadStart(testObj.DoSomeWorkSync));  
Thread t4 = new Thread( new ThreadStart(testObj.DoSomeWorkSync));  
t3.Start();  
t4.Start();  
  
Console.ReadLine();   
}

[复制代码](javascript:void(0);)

这里，我们有两个方法，我们可以对比一下，一个是加了属性MethodImpl 的DoSomeWorkSync()，一个是没加的DoSomeWorkNoSync()。在方法中Sleep(1000)是为了在第一个线程还在方法中 时，第二个线程能够有足够的时间进来。对每个方法分别起了两个线程，我们先来看一下结果：



可以看出，对于线程１和２，也就是调用没有加属性的方法的线程，当线程２进入方法后，还没有离开，线程１有进来了，这就是说，方法没有同步。我们再来看看线程３和４，当线程３进来后，方法被锁，直到线程３释放了锁以后，线程４才进来。

**九、同步事件和等待句柄**

　　用lock和Monitor可以很好地起到线程同步的作用，但它们无法实现线程之间传递事件。如果要实现线程同步的同时，线程之间还要有交互，就要用到同步事件。同步事件是有两个状态（终止和非终止）的对象，它可以用来激活和挂起线程。

　　同步事件有两种：AutoResetEvent和 ManualResetEvent。 它们之间唯一不同的地方就是在激活线程之后，状态是否自动由终止变为非终止。AutoResetEvent自动变为非终止，就是说一个 AutoResetEvent只能激活一个线程。而ManualResetEvent要等到它的Reset方法被调用，状态才变为非终止，在这之 前，ManualResetEvent可以激活任意多个线程。

　　可以调用WaitOne、WaitAny或WaitAll来使线程等待事件。它们之间的区别可以查看MSDN。当调用事件的 Set方法时，事件将变为终止状态，等待的线程被唤醒。

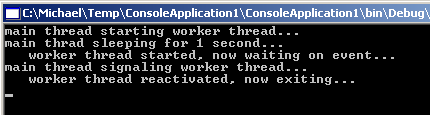
　　来看一个例子，这个例子是MSDN上的。因为事件只用于一个线程的激活，所以使用 **AutoResetEvent** 或 **ManualResetEvent** 类都可以。

[复制代码](javascript:void(0);)

C:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\ExpandedBlockStart[1].gifCode  
static AutoResetEvent autoEvent;  
  
static void DoWork()  
{  
Console.WriteLine(" worker thread started, now waiting on eventC:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\dot[1].gif");  
autoEvent.WaitOne();  
Console.WriteLine(" worker thread reactivated, now exitingC:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\dot[1].gif");  
}  
  
[STAThread]  
static void Main(string[] args)  
{  
autoEvent = new AutoResetEvent(false);  
  
Console.WriteLine("main thread starting worker threadC:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\dot[1].gif");  
Thread t = new Thread(new ThreadStart(DoWork));  
t.Start();  
  
Console.WriteLine("main thrad sleeping for 1 secondC:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\dot[1].gif");  
Thread.Sleep(1000);  
  
Console.WriteLine("main thread signaling worker threadC:\Users\wyz\Documents\My Knowledge\temp\6f545109-2ebc-4630-a4bd-c200cbf365be_4_files\dot[1].gif");  
autoEvent.Set();  
  
Console.ReadLine();   
}

[复制代码](javascript:void(0);)

我们先来看一下输出：



在主函数中，首先创建一个AutoResetEvent的实例，参数false 表示初始状态为非终止，如果是true的话，初始状态则为终止。然后创建并启动一个子线程，在子线程中，通过调用AutoResetEvent的 WaitOne方法，使子线程等待指定事件的发生。然后主线程等待一秒后，调用AutoResetEvent的Ｓｅｔ方法，使状态由非终止变为终止，重新 激活子线程。

参考：

1/MSDN([http://msdn.microsoft.com/zh-cn/library/ms173179(VS.80).aspx](http://msdn.microsoft.com/zh-cn/library/ms173179%28VS.80%29.aspx) )

2/http://www.cnblogs.com/VincentWP/archive/2008/06/25/1229104.html

来源： <<http://www.cnblogs.com/michaelxu/archive/2008/09/20/1293716.html>>