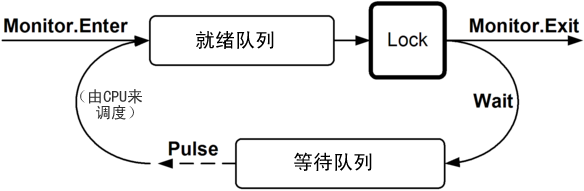
C# 温故而知新： 线程篇（四）线程同步篇 （中）：同步工具类的介绍

1 上篇回顾

让我们开始上一篇向大家介绍了下线程同步中的一些重要概念包括：基元内核模式，基元用户模式，原子性，然后由陆续介绍了基元用户模式中的Validated,Interloced 和ReaderWriterLock 类,同时也简单介绍了下基元内核模式中的lock关键字，本章再让我们继续深入了解其他的一些同步工具类

2 继续介绍基元内核模式中的Monitor类



Monitor类也是同步机制中比较重要的一个类，它属于基元内核模式中的一种，也是上一章中与lock关键字有着密切关系，Monitor类采取排他锁来进行对共享区的同步，当一个线程进入共享区时，会取得排他锁的控制权，其他线程则必须等待，大伙注意，这里有2个重要的线程状态需要在说明下

1：等待队列：  等待进入共享区的线程会首先进入到等待队列中，等待持有排他锁的线程通知某个等待线程进入到就绪队列中，注意（只有拥有排他锁的线程才能进行互换通知功能,甚至该线程能够唤醒一堆的等待线程进入到就绪队列中）

2：就绪队列:  等待队列中的某个线程被持有排他锁的线程唤醒放入到就绪队列中，等待获取排他锁的机会，这样一个周期便可以连接起来，线程从等待到被唤醒到就绪状态，然后获取排他锁进如共享区操作，然后交出排他锁等待或者睡眠，直到再次被唤醒。

在这里强调下Monitor是个十分容易产生死锁的同步类，其原因是：

**1.当一个线程试图去请求锁对象时，它不是处在等待队列，而是就绪队列,如果需要让其进入等待队列，则必须使用Wait方法**

**2.当一个线程释放锁对象时是不会通知等待队列中的线程进入到就绪队列，需要通过Palse方法**

**3.线程启动的时候，线程是处于就绪状态的**

**4.就算处于就绪状态的线程被cpu选中，但是一旦数据被锁定，那个线程还是无法获取到控制权**

其实大家了解原因后对于monitor已经算了解，这正是monitor的机制了解了上述机制后，大家可以开始理解该类比较重要的几个方法：

 Monitor. Enter(Object);   该方法旨在宣布当前线程进入了临界区，持有了排他锁，其他线程继续等待，直到该线程离开共享区交出排他锁给就绪队列中的一个线程

 Monitor. Exit(Object);      当持有共享锁的线程执行完任务之后，该线程变通过这个方法离开共享区，临走前可以操作其唤醒一个或一堆的等待线程

Monitor.Palse(Object)和Monitor.Palse(Object): 这两个方法比较复杂和相似，也就是唤醒（改变）其他线程状态的方法，持有排他锁的线程利用这两个方法通知其他线程进入到就绪队列，离开等待队列

Monitor.Wait(Object):这个方法也是非常的重要，假如在共享区内的线程执行了wait方法后，该线程会被放入等待队列中，从而失去了排他锁的控制权，就绪队列中的下一个线程就进入到了临界区

Monitor.TryEnter(Object, Boolean)):有时其他线程希望通过进行尝试的方式主动去争取排他锁的控制权，这个方法便能实现这个功能，同时通过一个BOOL参数来指示会否占有了排他锁

文字介绍为了让大伙更好的理解Monitor类的本质，接下来就上很简单代码让我们更深入的了解

/// <summary>开启2个写线程，2个读线程演示</summary> /// <param name="args"></param>

static void Main(string[] args){

Thread t1 = new Thread(new ThreadStart(WriteInShareArea));

Thread t3 = new Thread(new ThreadStart(WriteInShareArea));

Thread t2 = new Thread(new ThreadStart(Read));

Thread t4= new Thread(new ThreadStart(Read));

Console.WriteLine("t1's id={0}", t1.ManagedThreadId);

Console.WriteLine("t2's id={0}", t2.ManagedThreadId);

Console.WriteLine("t3's id={0}", t3.ManagedThreadId);

t1.Start();

t3.Start();

t2.Start();

t4.Start();

Console.Read();

}

/// <summary>读数据，首先是先让读线程等待，当写线程写完执行 Pulse 方法后，唤醒读线程继续工作</summary>

private static void Read(){

while (true){

Monitor.Enter(lockObj);  
 //读线程启动时默认就绪队列

//读线程执行到这里时会挂起进入等待队列

Monitor.Wait(lockObj, 4000);

Console.WriteLine("Thread{0} can read nao", Thread.CurrentThread.ManagedThreadId);

Console.WriteLine("Thread{0} reading...............", Thread.CurrentThread.ManagedThreadId);

Thread.Sleep(3000);

Monitor.Exit(lockObj);

}

}

/// <summary>写数据，当写线程写完执行 Pulse 方法后，唤醒读线程继续工作</summary>

private static void WriteInShareArea(){

while (true){

Thread.Sleep(1000);

Monitor.Enter(lockObj);

Console.WriteLine("Thread{0} change data nao", Thread.CurrentThread.ManagedThreadId);

Console.WriteLine("Thread{0} changing...............", Thread.CurrentThread.ManagedThreadId);

Thread.Sleep(1000);

//唤醒读线程进入就绪队列 Monitor.Pulse(lockObj);

Monitor.Exit(lockObj);

}

}

执行结果：



3  同步句柄：WaitHandle

      WaitHandle可以说是本章中许多同步对象的基类包括前后文中的AutoResetEvent，ManualResetEvent，mutex 和Semaphore都是其子类，waitHandle是个抽象类，主要的功能从字面上就能看的出来，等待句柄，不错 WaitHandle非常的神奇，它容纳了一个WIN32的内核对象句柄，线程会等待内核对象的消息，当然内核对象也会等待接收信号，一旦接收到信号则会通知当前线程，这是一种复杂的操作系统调度原理，大家可以参考WINDOWS核心编程等书继续深入了解下，本文旨在说明下waitHandle的概念和一些简单的介绍，这里在为waitHandle打个比方： 当你想穿过马路时，如果信号灯为红色则你只能等待，如果转变成绿灯的话，你就可以通行，其实waitHandle的作用就是红绿灯的作用，聪明的的你肯定想到了，那我们可以将线程同步信号灯放入线程池中么？哈哈当然可以，不仅可以放一根，而且可以放入一堆WaitHadle对象。言归正传：WaitHadle也有2个信号” Signaled” and “NonSignaled” ，前者可以理解为绿灯，绿灯状态时WaitOne方法无效，当前线程不会被阻止，后者可以理解为红灯,当底层内核对象接受到信号时，通知某个当前线程可以进入到共享区（临界区）其余的必须等待，不过基于抽象类的考虑，waitHandle没有唤醒线程的set方法，而让其子类实现了，所以WaitHandle 还是完全体现其”等啊等，直到提示可以通过或者直接奔溃（超时异常）”的特色

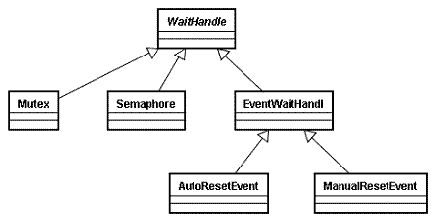
   既然waitHandle只有阻塞等待的作用，那我们来看下它的几个奇妙的方法：

   1 bool WaitOne(): 等待一个通过当前waitHandle指定的内核对象收到信号后返回true,否则返回false

   2 bool WaitAll():等待waitHandle[]中所有的内核对象都收到信号后返回true,否则返回false

   3 bool WaitAny();等待waitHandle[]中内核对象都收到信号后返回true,否则返回false

   4 bool SignalAndWait():这个方法无法用文字表达清楚，但是大伙先可以理解成这样：自动的向内核对象发送信息，等待另一个内核对象接收到信号，如果另一个内核对象接受到信号则返回TRUE，这牵涉到了复杂的”混合锁”的机制，所以本文不再详细说明，后章会详细介绍混合锁机制接着让我们来看下WaitHandle的几个派生类结构，后文中将一一介绍：



最后上一个图来表示下WaitHandle的工作原理



大胖子就是个waitHandle（只负责让当前线程等待，也可以雇佣多个大胖子，内核对象接受到信号后通知大胖子，然后大胖子放人进入club）

4 EventWaitHandle , AutoResetEvent和ManualResetEvent

相信大家对于WaitHandle已经有所了解，接着我们将来介绍下EventWaitHandle ,AutoResetEvent和ManualResetEvent的一些概念和使用方法（用列表的方式）：

 EventWaitHandle：

1具有WaitHandle的一些阻塞线程的wait方法

2具有Set方法来释放被阻塞的当前线程

3具有终止状态和非终止状态

4具有自己的重置模式可以选择：自动重置和手动重置，当eventWaitHandle对象调set方法后，eventWaitHandle会更具重置模式自动重置或手动重置，重置后会立刻阻塞当前线程（当前线程调用Wait方法后阻塞），如果一直不重置，将无法阻塞当前线程

5 重置模式在初始化eventWaitHandle对象的第二个参数中设置

6 终止状态不会导致阻塞线程，非终止装态会导致阻塞当前线程

7 可以调用Reset()方法将状态设置成非终止装态来阻塞线程

 AutoResetEvent

1 具有父类EventWaitHandle 的功能

2 同样具有set方法,**但是每次只能唤醒一个线程**

3 同样具有终止状态和非终止状态

4 重置模式为自动（其实就是一直处于非终止状态，等待新的线程，waitOne方法将不会阻塞线程，直到新的线程进入临界区后，自动将模式设置成非终止模式，阻塞其他线程，如果一直没有新的线程进入，那只能永远处于终止状态）

5 也可以调用Reset()方法将状态设置成非终止装态来阻塞线程

6 可以在构造函数中设置默认的状态(终止状态和非终止状态)

简单示例：

/// <summary>演示waitHandle和AutoResetEvent的同步方法</summary>

class Program{

static WaitHandle[] waitHandle;

static void Main(string[] args){

//委托集合

List<Action> ls = new List<Action>{

()=>{ Console.Write("A");},

()=>{ Console.Write("B");},

()=>{ Console.Write("C");},

()=>{ Console.Write("D");},

()=>{ Console.Write("E");},

()=>{ Console.Write("F");},

()=>{ Console.Write("G");},

};

InvokeAllActions(ls);

}

/// <summary>利用线程池对委托集合中的所有委托进行触发，同时通过AutoResetEvent 进行同步</summary>

static void InvokeAllActions(List<Action> actionList){

waitHandle = new WaitHandle[actionList.Count];

if (actionList == null || actionList.Count <= 0) return;

int i = 0;

actionList.ForEach((action) =>{

//创建AutoResetEvent对象

waitHandle[i] = new AutoResetEvent(false);

//将AutoResetEvent对象作为参数放入线程池

ThreadPool.QueueUserWorkItem(new WaitCallback((handle) => {

//触发委托 action.Invoke();

Thread.Sleep(new Random().Next(1000, 10000));

//委托执行完毕后发出信号，告诉其他线程可以进入共享区

(handle as AutoResetEvent).Set();

}), waitHandle[i]);

if (i == actionList.Count - 1) return;

i++;

});

//注意WaitAll的使用方法，当主线程调用WaitAll方法时，将等待所有的waitHandle接受到信号后才能通过

//信号通过Set()方法发出 WaitHandle.WaitAll(waitHandle);

}

}

 输出结果可能随时发生变化

  ManualResetEvent

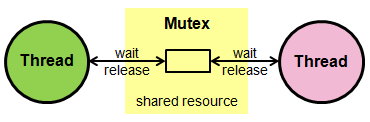
1 具有父类EventWaitHandle 的功能

2 同样具有set方法,**但是每次可以唤醒一个或多个线程**

3 同样具有终止状态和非终止状态

4 重置模式为手动（和AutoResetEvent不同的的是执行waitOne方法后，它不会自动将模式设置成非终止模式，也就是说会一直保持有信号状态，除非程序员手工写上Reset方法让其状态成为非终止状态，阻塞其他线程，同理，一旦执行set方法后ManualResetEvent也就可以同时唤醒多个线程继续执行）

5  同步互斥mutex类



      Mutext 出现的比monitor更早，而且传承自COM，当然，waitHandle也是它的父类，它继承了其父类的功能，有趣的是Mutex的脾气非常的古怪，它允许**同一个线程多次重复访问共享区**，但是对于别的线程那就必须等待，同时，它**甚至支持不同进程中的线程同步**，这点更能体现他的优势，但是劣势也是显而易见的，那就是巨大的性能损耗和容易产生死锁的困扰，所以除非需要在特殊场合，否则 我们尽量少用为妙，这里并非是将mutex的缺点说的很严重，而是建议大家在适当的场合使用更为适合的同步方式,mutex 就好比一个重量型的工具，利用它则必须付出性能的代价。

     接着让我们了解下mutex的工作方式，同样我们用列表的方式更简洁

1．  mutex类通过WaitOne 方法阻止另一个线程的进入共享区，但是对于拥有相同的互斥体的线程，WaitOne方法无效，如果其他线程想获取互斥体的话，必须等待拥有互斥体的线程执行ReleasMutex方法后释放该互斥体后才有可能获取

2．  每次使用WaitOne 方法后都需要使用ReleasMutex方法释放互斥体的所属权，waitOne方法和ReleaMutex方法使用次数必须一致，对于递归锁和混合锁将非常复杂

3.   同样具有终止状态和非终止状态

4．  ReleaMutex 后互斥体的状态设定为终止，直到其他线程占有互斥体，但是如果没有线程拥有互斥体的话，该互斥体的状态便终止了

5.   父类WaitHandle负责接收信号，当接收到信号后方能让其他线程进入临界区或是获取到互斥体的所属权

6.   能否在进程中互相同步取决于该Mutex对象是否有名字，这似乎有点奇怪，但是大家仔细想下，如果跨进程实现同步的话，那么其他进程假如也有一些Mutex的吧，那么根本无法告诉在不同进程中的线程是这个互斥体mutex对象负责这方面的同步工作，或许大家会问如果名字一样怎么办？好问题，微软也想到了，而且它是通过再制造一个新的互斥体替代，这样和不同进程间的互斥体保持唯一互补冲突，但是这个选项在构造函数中，下面的构造函数也会阐述这点

7.  对于没有名字的mutex对象，我们称之为局部互斥体，相反则是全局互斥体

 最后来简单介绍下Mutex类的几个重要方法\*

1.Bool WaitOne():阻止当前线程，直到收到信号后才能继续，如果一直没有接受到信号则永远不会返回而被阻塞，负责接受信号是比较重要的同步类WaitHandle，关于这个类，下文中会有详细阐述

2.void ReleaseMutex():这个方法非常重要，它是配合waitOne方法而存在的，简单的说拥有互斥体mutex控制权的线程如果不需要互斥体的话，则使用该方法释放mutex，再次提醒下WaitOne方法和ReleaseMutex方法的使用次数必须一致

3. Mutex OpenExisting（string mutexName）:该方法相对于mutex来说也是必不可少，它的作用是查找mutex是否存在，如果不存在的话，则会引发WaitHandleCannotBeOpenedException，我们可以再这个捕获异常中实现mutex对象的创建

 Mutex类的构造方法：

1.Mutex():用无参数的构造函数得到的Mutex没有任何名称，而进程间无法通过变量的形式共享数据，所以没有名称的Mutex也叫做局部（Local）Mutex。另外，这样创建出的Mutex，创建者对这个实例并没有拥有权，仍然需要调用WaitOne()去请求所有权。

2.Mutex(Boolean initiallyOwned):与上面的构造函数一样，它只能创建没有名称的局部Mutex，无法用于进程间的同步。Boolean参数用于指定在创建者创建Mutex后，是否立刻获得拥有权，因此Mutex(false)等效于Mutex()。

3.Mutex(Boolean initiallyOwned, String name):在这个构造函数里我们除了能指定是否在创建后获得初始拥有权外，还可以为这个Mutex取一个名字。只有这种命名的Mutex才可以被其它应用程序域中的程序所使用，因此这种Mutex也叫做全局（Global）Mutex。 **如果String为null或者空字符串，那么这等同于创建一个未命名的Mutex**。因为可能有其他程序先于你创建了同名的Mutex，因此返回的 Mutex实例可能只是指向了同名的Mutex而已。但是，这个构造函数并没有任何机制告诉我们这个情况。因此，如果要创建一个命名的Mutex，并且期 望知道这个Mutex是否由你创建，最好使用下面两个构造函数中的任意一个。最后，请注意name是**大小写敏感**的。

4.Mutex(Boolean initiallyOwned, String name, out Boolean createdNew)：头两个参数与上面的构造函数相同，第三个out参数用于表明是否获得了初始的拥有权。这个构造函数应该是我们在实际中使用较多的。

 5.Mutex(Boolean initiallyOwned, String name, out Booldan createdNew, MutexSecurity)：多出来的这个MutexSecurity参数，也是由于全局Mutex的特性所决定的。

同样马上上一个例子:

/// <summary>本示例通过多线程拷贝字符数组</summary>

class Program{

private static char[] testChars = "abcdefg".ToCharArray();

private static char[] backChars = new char[testChars.Length];

private static Mutex mutex = new Mutex(false,"myMutex");

static void Main(string[] args){

try{

//寻找名为myMutex2的互斥体对象

Mutex mutex2= Mutex.OpenExisting("myMutex2", MutexRights.FullControl);

}

catch (WaitHandleCannotBeOpenedException e){

Console.WriteLine("the name of mutex is not exist, error message:{0}",e.Message);

}

for (int i = 0; i < testChars.Length; i++){

ThreadPool.QueueUserWorkItem(new WaitCallback(SetBackString));

}

Console.ReadLine();

}

static void SetBackString(object state=null) {

///阻止当前线程，直到收到信号后才能继续

mutex.WaitOne();

try{

if (testChars.Length > 0){

//每次将test4的第一个字符串放入backChars,原始backChars.Length 减去被删除一个char的testChars.length

Array.Copy(testChars, 0, backChars, backChars.Length - testChars.Length, 1);

Console.WriteLine("ThreadID:{0} the chart:{1} will insert into backChars,the string changed from backChars:{2}", Thread.CurrentThread.ManagedThreadId, testChars[0], string.Join(string.Empty, backChars));

//申明一个临时char数组用来存放testChars,实际作用是删除testChars的一个char

char[] temp = new char[testChars.Length-1];

//将不需要删除的chars拷贝入temp

Array.Copy(testChars,1, temp, 0, testChars.Length-1);

testChars = temp;

}

}

catch{}

finally {

//无论发生什么当前线程最终还是得释放互斥体的控制权

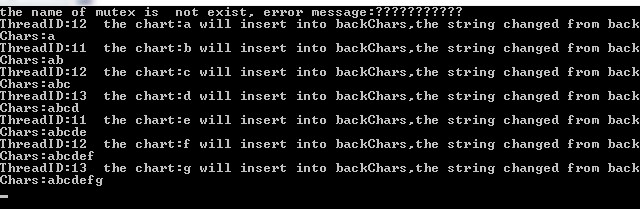
mutex.ReleaseMutex();

}

}

}

显示结果：



6  简单说明下Mutex和Monitor的区别

 相信大家看了上文后对于的Mutex和Monitor已经有所了解，由于这2个工具类可能用法上过于相似，当然区别也是不小

1. Monitor不是waitHandle的子类，它具有等待和就绪队列的实际应用

2  Monitor无法跨进程中实现线程同步，但是Mutex可以

3  相对而言两者有明显的性能差距，mutex相对性能较弱但是功能更为强大，monitor则性能比较好

4  两者都是用锁的概念来实现同步不同的是monitor一般在方法（函数）调用方加锁；mutex一般在方法（函数）内部加锁，即锁定被调用端

以上是比较主要的区别，一些细节大家也可以继续深究

7  选择我们需要的同步工具

有时我们苦于在项目中寻找适合的同步工具，所以一下仅是个人观点，大伙可以讨论或者发表自己的意见

1 处于性能要求考虑：可以考虑用基元用户模式的同步工具，也就是前一篇中的一些同步工具，尽量不要考虑mutex因为其功能强大所以性能损失太多

2 处于功能考虑：如果项目中牵涉到复杂的同步而且不需要严格的性能要求，例如跨进程，混合锁或者递归锁等等，则最好选择基元内核模式中的同步工具

3 分布式开发，在各个服务间实现同步的话，当然mutex是第一考虑

4 中小型项目大家可以随意，根据业务需要决定