# [**C# 线程资源同步方式总结**](http://www.cnblogs.com/VincentWP/archive/2008/06/25/1229104.html)

        在现代的程序开发中,资源的同步是一个比较重要的课题,在.Net中,对这部分有很丰富类库供我们使用,现在总结一下在各种情况下对资源同步的

机制。

     1.将字段声明为volatile

        当一个字段被声明为volatile时,CLR中一些管理代码和内存的内部机制将负责对字段进行同步,并且总能保证读取到的字段信息都为最新的值,被声明为

volatile的字段必须具备以下特征之一ds that are declared volatile are not subject to compiler optimizations.

        1.为引用类型

        2.一个指针(在不安全代码中)

        3.sbyte,byte,short,ushort,int,uint,char,float,bool

        4.一个使用底层类型的枚举

Fiel类型

     2**.使用System.Threading.Interlocked 类**

         在许多增对整数的操作中,我们都比较容易忽视线程的问题,例如执行下列代码

         i = i + 1;

         实际上,上述代码分为3步骤

              1).  从内存中,读取i的值

              2).  将读取出来的值加1

              3).  将新的值写入内存中

         在单线程上,这个操作不会有任何问题,但是当i被多个线程访问时,问题就出现了,对i进行修改的线程,在上述的任何一部都有可能被其它读取线程打断,想象一下,

当操作线程执行完第二步,准备将新的值写入内存中时,此时其它读取线程获得了执行权,这时读取到的i的值并不是我们想要的,因此,这个操作不具备原子性,在.Net中,使用Interlocked类能确保操作的原子性,Interlocked类有以下的方法

          Increment

          Decrement

          Exchange

         上述的方法的参数都为带ref 标识的参数,因此我们说,这些方法保证了数据的原子性,在多线程中,涉及到整数的操作时,数据的原子性值得考虑,Interlocked的操作代码如下

            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);

          输出信息如下



       3.**使用lock关键字**

          地球人都知道,使用lock关键字可以获取一个对象的独占权,任何需要获取这个对象的操作的线程必须等待以获取该对象的线程释放独占权,lock提供了简单的同步资源的方法,与Java中的synchronized关键字类似。

          lock关键字的使用示例代码如下

object o = new object();

            lock (o)

            {

                Console.WriteLine("O");

            }

       4.**使用System.Theading.Monitor类进行同步**

          System.Threading.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,查看如下代码

{

    class Program

    {

        private static object m\_monitorObject = new object();

        static void Main(string[] args)

        {

            Thread thread = new Thread(Do);

            thread.Name = "Thread1";

            Thread thread2 = new Thread(Do);

            thread2.Name = "Thread2";

            thread.Start();

            thread2.Start();

            thread.Join();

            thread2.Join();



        }



        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);

            }

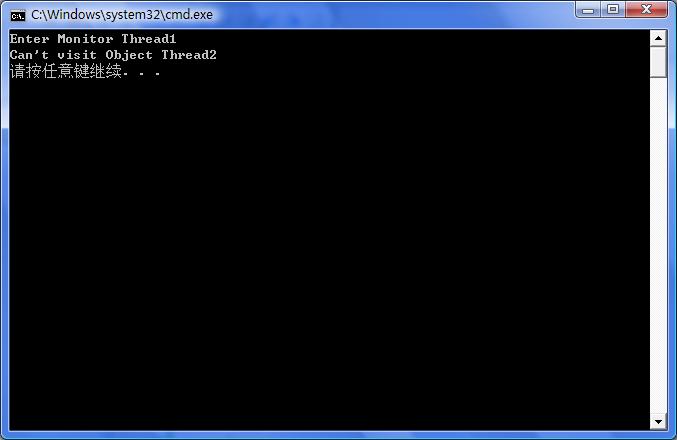
        }

    }

}



 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



可以看到线程2无法获取到m\_monitorObject的独占权,因此输出了一条错误信息.

Monitor类比lock类提供了一种更优秀的功能,考虑一下如下的场景

      1.当你进入某家餐馆时,发现餐馆里坐满了客人,但是你又不想换地方,因此,你选择等待。

      2.当餐馆内的服务员发现有空位置时,他通知前台的服务生,服务生给你安排了一个座位,于是你开始享受国际化的大餐。

使用Monitor类就可以实现如上的应用需求,在Monitor类中,提供了如下三个静态方法

       Monitor.Pulse(Object o)

       Monitor.PulseAll(Object o)

       Monitor.Wait(Object o ) // 重载函数

当调用Wait方法时,线程释放资源的独占锁,并且阻塞在wait方法直到另外的线程获取资源的独占锁后,更新资源信息并调用Pulse方法后返回。

我们模拟上述代码如下

{

    class Program

    {

        private static  object m\_isFull = new object();

        static void Main(string[] args)

        {

            Thread thread = new Thread(HaveLunch);

            thread.Name = "HaveLunchThread";

            Thread thread2 = new Thread(SeatChange);

            thread2.Name = "SeatChangeThread";

            thread.Start();

            System.Threading.Thread.Sleep(2000);

            thread2.Start();

            thread.Join();

            thread2.Join();

        }



        private static void HaveLunch()

        {

            lock (m\_isFull)

            {

                Console.WriteLine("Wati for seta");

                Monitor.Wait(m\_isFull);

                Console.WriteLine("Have a good lunch!");

            }

        }



        private static void SeatChange()

        {

            lock (m\_isFull)

            {

                Console.WriteLine("Seat was changed");

                Monitor.Pulse(m\_isFull);

            }

        }

    }

}



输出信息如下



可见,**使用Monitor,我们能实现一种唤醒式的机制,相信在实际应用中有不少类似的场景**。

          5.**使用System.Threading.Mutex(互斥体)类实现跨进程同步，每次释放一个进程**

          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。

           考虑如下的代码,代码通过获取一个称为ConfigFileMutex的互斥体,修改配置文件信息,写入一条数据,我们同时开启两个相同的进程进行测试

{

    class Program

    {

        static void Main(string[] args)

        {

            Mutex configFileMutex = new Mutex(false, "configFileMutex");

            Console.WriteLine("Wait for configFileMutex Process Id : " + Process.GetCurrentProcess().Id);

            configFileMutex.WaitOne();

            Console.WriteLine("Enter configFileMutex Process Id : " + Process.GetCurrentProcess().Id);

            System.Threading.Thread.Sleep(10000);

            if (!File.Exists(@".\config.txt"))

            {

                FileStream stream = File.Create(@".\config.txt");

                StreamWriter writer = new StreamWriter(stream);

                writer.WriteLine("This is a Test!");

                writer.Close();

                stream.Close();

            }

            else

            {

                String[] lines = File.ReadAllLines(@".\config.txt");

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

                    Console.WriteLine(lines[i]);

            }

            configFileMutex.ReleaseMutex();

            configFileMutex.Close();

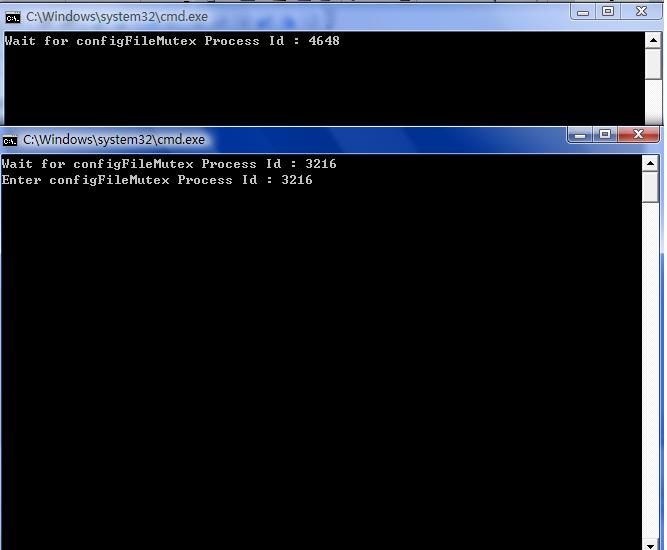
        }

    }

}

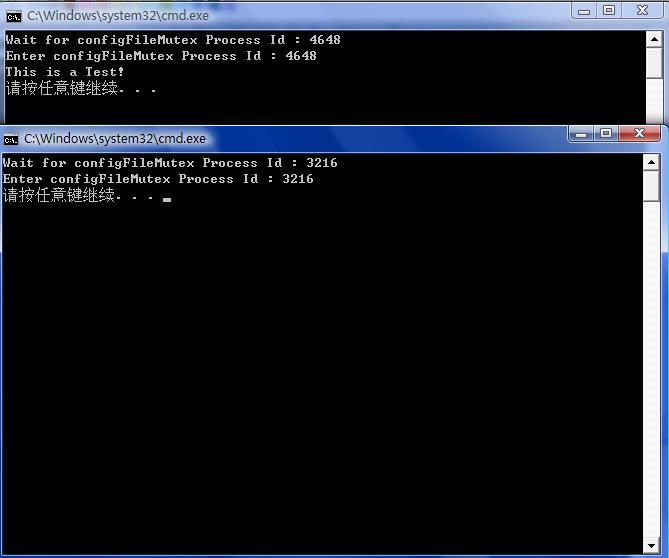


运行截图如下



此时,PID 为 4628的进程正在等待PID 为 3216的进程释放configFileMutex的互斥体,因此它阻塞在WaitOne函数中,当PID为3216的进程添加并写入了一条信息

至config.txt文件后,释放configFileMutex的独占权,PID为4628的进程获取configFileMutex的独占权,并从config.txt文件中读取输出PID3216进程写入的信息,截图如下



可见使用Mutex进行同步,同步的互斥体是存在于多个进程间的。

            6. **使用System.Threading.ReaderWriterLock类实现多用户读/单用户写的同步访问机制**

                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

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading;

using System.IO;

using System.Diagnostics;

namespace MonitorApplication

{

    class Program

    {

        private static ReaderWriterLock m\_readerWriterLock = new ReaderWriterLock();

        private static int m\_int = 0;

        static void Main(string[] args)

        {

            Thread readThread = new Thread(Read);

            readThread.Name = "ReadThread1";

            Thread readThread2 = new Thread(Read);

            readThread2.Name = "ReadThread2";

            Thread writeThread = new Thread(Writer);

            writeThread.Name = "WriterThread";

            readThread.Start();

            readThread2.Start();

            writeThread.Start();

            readThread.Join();

            readThread2.Join();

            writeThread.Join();



        }



        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");

            }

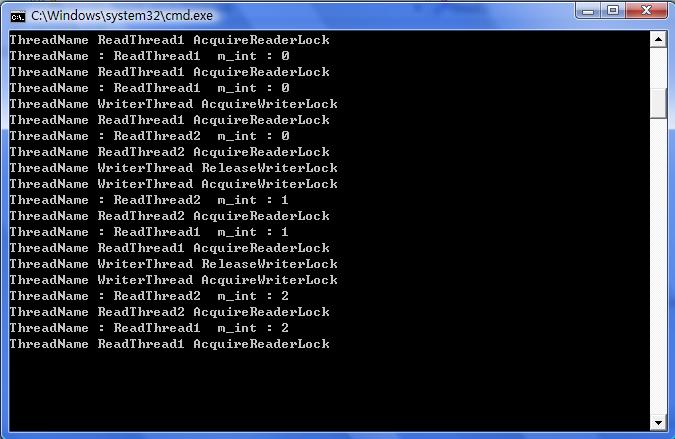
        }

    }

}



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。

**7.使用System.Runtime.Remoting.Contexts.SynchronizationAttribute对类对象进行同步控制**

            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

 [System.Runtime.Remoting.Contexts.Synchronization]

 public class SynchronizedClass : System.ContextBoundObject

 {



 }

         8.**使用System.Threading.Semaphore控制并发访问的数量**

Limits the number of threads that can access a resource or pool of resources concurrently.

using System;

using System.Threading;

public class Example

{

    // A semaphore that simulates a limited resource pool.

    //

    private static Semaphore \_pool;

    // A padding interval to make the output more orderly.

    private static int \_padding;

    public static void Main()

    {

        // Create a semaphore that can satisfy up to three

        // concurrent requests. Use an initial count of zero,

        // so that the entire semaphore count is initially

        // owned by the main program thread.

        //

        \_pool = new Semaphore(0, 3);

        // Create and start five numbered threads.

        //

        for(int i = 1; i <= 5; i++)

        {

            Thread t = new Thread(new ParameterizedThreadStart(Worker));

            // Start the thread, passing the number.

            //

            t.Start(i);

        }

        // Wait for half a second, to allow all the

        // threads to start and to block on the semaphore.

        //

        Thread.Sleep(500);

        // The main thread starts out holding the entire

        // semaphore count. Calling Release(3) brings the

        // semaphore count back to its maximum value, and

        // allows the waiting threads to enter the semaphore,

        // up to three at a time.

        //

        Console.WriteLine("Main thread calls Release(3).");

        \_pool.Release(3);

        Console.WriteLine("Main thread exits.");

    }

    private static void Worker(object num)

    {

        // Each worker thread begins by requesting the

        // semaphore.

        Console.WriteLine("Thread {0} begins " +

            "and waits for the semaphore.", num);

        \_pool.WaitOne();

        // A padding interval to make the output more orderly.

        int padding = Interlocked.Add(ref \_padding, 100);

        Console.WriteLine("Thread {0} enters the semaphore.", num);

        // The thread's "work" consists of sleeping for

        // about a second. Each thread "works" a little

        // longer, just to make the output more orderly.

        //

        Thread.Sleep(1000 + padding);

        Console.WriteLine("Thread {0} releases the semaphore.", num);

        Console.WriteLine("Thread {0} previous semaphore count: {1}",

            num, \_pool.Release());

    }

}

   在.Net多线程编程中,AutoResetEvent和ManualResetEvent这两个类经常用到, 他们的用法很类似，但也有区别。Set方法将信号置为发送状态，Reset方法将信号置为不发送状态,WaitOne等待信号的发送。可以通过构造函数的 参数值来决定其初始状态，若为true则非阻塞状态，为false为阻塞状态。如果某个线程调用WaitOne方法,则当信号处于发送状态时,该线程会得 到信号, 继续向下执行。其区别就在调用后,**AutoResetEvent.WaitOne()每次只允许一个线程进入,当某个线程得到信号 后,AutoResetEvent会自动又将信号置为不发送状态,则其他调用WaitOne的线程只有继续等待.也就是说,AutoResetEvent 一次只唤醒一个线程;而ManualResetEvent则可以唤醒多个线程,因为当某个线程调用了ManualResetEvent.Set()方法 后,其他调用WaitOne的线程获得信号得以继续执行,而ManualResetEvent不会自动将信号置为不发送**。也就是说,除非手工调用了 ManualResetEvent.Reset()方法,则ManualResetEvent将一直保持有信号状态,ManualResetEvent也 就可以同时唤醒多个线程继续执行。

            还有AutoResetEvent,ManualResetEvent,EventWaitHandle,Semaphore等可以实现资源的控制,不过它们更多是是基于一种事件唤醒的机制,

如果有兴趣可以查阅MSDN相关的文档。