[**C# 多线程(lock,Monitor,Mutex,同步事件和等待句柄)**](http://blog.csdn.net/youarenotme/article/details/77882178)

**原文地址：http://www.cnblogs.com/SkySoot/archive/2012/04/02/2430295.html**

本篇从 Monitor，Mutex，ManualResetEvent，AutoResetEvent，WaitHandler 的类关系图开始，希望通过本篇的介绍能对常见的线程同步方法有一个整体的认识，而对每种方式的使用细节，适用场合不会过多解释。

让我们来看看这几个类的关系图：   
**1. lock 关键字**    lock 是 C# 关键词，它将语句块标记为临界区,确保当一个线程位于代码的临界区时，另一个线程不进入临界区。如果其他线程试图进入锁定的代码，则它将一直等待（即被阻止），直到该对象被释放。方法是获取给定对象的互斥锁，执行语句，然后释放该锁。**MSDN 上给出了使用 lock 时的注意事项通常，应避免锁定 public 类型，否则实例将超出代码的控制范围。**

    常见的结构 lock (this)、lock (typeof (MyType)) 和 lock ("myLock") 。   
        1）如果实例可以被公共访问，将出现 lock (this) 问题。   
        2）如果 MyType 可以被公共访问，将出现 lock (typeof (MyType)) 问题。由于一个类的所有实例都只有一个类型对象（该对象是 typeof 的返回结果），锁定它，就锁定了该对象的所有实例。**微软现在建议不要使用 lock(typeof(MyType)),因为锁定类型对象是个很缓慢的过程，并且类中的其他线程、甚至在同一个应用程序域中运行的其他程序都可以访问该类型对象，因此，它们就有可能代替您锁定类型对象，完全阻止您的执行，从而导致你自己的代码的挂起。**   
        3）由于进程中使用同一字符串的任何其他代码将共享同一个锁，所以出现 lock("myLock") 问题。这个问题和.NET Framework 创建字符串的机制有关系，**如果两个 string 变量值都是"myLock"， 在内存中会指向同一字符串对象。**最佳做法是定义 private 对象来锁定, 或 private static 对象变量来保护所有实例所共有的数据。   
    我们再来通过 IL Dasm 看看 lock 关键字的本质，下面是一段简单的测试代码：   
        lock (lockobject)   
        {   
            int i = 5;   
        }   
    用 IL Dasm 打开编译后的文件，上面的语句块生成的 IL 代码为：   
        IL\_0045: call   
        IL\_004a: nop   
        .try   
        {   
            IL\_004b: nop   
            void [mscorlib]System.Threading.Monitor::Enter(object)   
            IL\_004c: ldc.i4.5 IL\_004d: stloc.1   
        IL\_004e: nop IL\_004f: leave.s }   
        // end .try finally { IL\_0051: ldloc.3 IL\_0052: call IL\_0057: nop IL\_0058: endfinally }   
        // end handler   
    **通过上面的代码我们很清楚的看到：lock 关键字其实就是对 Monitor 类的 Enter()和 Exit()方法的封装。通过 try......catch......finally 语句块确保在 lock 语句块结束后执行 Monitor.Exit()方法，释放互斥锁。**

**2. Monitor 类**        Monitor类通过向单个线程授予对象锁来控制对对象的访问。对象锁提供限制访问临界区的能力。当一个线程拥有对象的锁时，其他任何线程都不能获取该锁。还可以使用 Monitor 来确保不会允许其他任何线程访问正在由锁的所有者执行的应用程序代码节，除非另一个线程正在使用其他的锁定对象执行该代码。通过对 lock 关键字的分析我们知道，lock 就是对 Monitor 的 Enter 和 Exit 的一个封装，而且使用起来更简洁，因此 **Monitor 类的 Enter()和 Exit()方法的组合使用可以用 lock 关键字替代。**   Monitor 类的常用方法：   
        TryEnter()：   
            能够有效的解决长期死等的问题，如果在一个并发经常发生，而且持续时间长的环境中使用 TryEnter，可以有效防止死锁或者长时间的等待。比如我们可以设置一个等待时间 bool gotLock = Monitor.TryEnter(myobject,1000),让当前线程在等待 1000 秒后根据返回的 bool 值来决定是否继续下面的操作。

**Wait() ：**

**释放对象上的锁以便允许其他线程锁定和访问该对象。在其他线程访问对象时，调用线程将等待。脉冲信号用于通知等待线程有关对象状态的更改。**  
        Pulse()：   
        PulseAll()：   
            向一个或多个等待线程发送信号。该信号通知等待线程锁定对象的状态已更改，并且锁的所有者准备释放该锁。等待线程被放置在对象的就绪队列中以便它可以最后接收对象锁。一旦线程拥有了锁，它就可以检查对象的新状态以查看是否达到所需状态。注意：Pulse、PulseAll 和 Wait 方法必须从同步的代码块内调用。

我们假定一种情景：妈妈做蛋糕，小孩有点馋，妈妈每做好一块就要吃掉，妈妈做好一块后，告诉小孩蛋糕已经做好了。**下面的例子用 Monitor 类的 Wait 和 Pulse 方法模拟小孩吃蛋糕的情景。**

**[csharp]** [view plain](http://blog.csdn.net/youarenotme/article/details/77882178) [copy](http://blog.csdn.net/youarenotme/article/details/77882178)

1. **using** System;
2. **using** System.Threading;
4. /// <summary>
5. /// 仅仅是说明 Wait 和 Pulse/PulseAll 的例子
6. /// 逻辑上并不严密，使用场景也并不一定合适
7. /// </summary>
8. **class** MonitorSample
9. {
10. **private** **int** n = 1; // 生产者和消费者共同处理的数据
11. **private** **int** max = 10000;
12. **private** **object** monitor = **new** **object**();
14. **public** **void** Produce() // 生产
15. {
16. **lock** (monitor)
17. {
18. **for** (; n <= max; n++)
19. {
20. Console.WriteLine("妈妈：第" + n.ToString() + "块蛋糕做好了");
22. // Pulse 方法不用调用是因为另一个线程中用的是 Wait(object,int) 方法
23. //**该方法使被阻止线程进入了同步对象的就绪队列**
24. // 是否需要脉冲激活是 Wait 方法一个参数和两个参数的重要区别
25. Monitor.Pulse(monitor);
27. // **调用 Wait 方法释放对象上的锁并阻止该线程（线程状态为 WaitSleepJoin）**
28. // **该线程进入到同步对象的等待队列，直到其它线程调用 Pulse 使该线程进入到就绪队列中**
29. // **线程进入到就绪队列中才有条件争夺同步对象的所有权**
30. // 如果没有其它线程调用 Pulse/PulseAll 方法，该线程不可能被执行
31. Monitor.Wait(monitor);
32. }
33. }
34. }
36. **public** **void** Consume() // 消费
37. {
38. **lock** (monitor)
39. {
40. **while** (**true**)
41. {
42. // 通知等待队列中的线程锁定对象状态的更改，但不会释放锁
43. **// 接收到 Pulse 脉冲后，线程从同步对象的等待队列移动到就绪队列中**
44. // 注意：最终能获得锁的线程并不一定是得到 Pulse 脉冲的线程
45. Monitor.Pulse(monitor);
47. // 释放对象上的锁并阻止当前线程，直到它重新获取该锁
48. // 如果指定的超时间隔已过，则线程进入就绪队列
49. // 该方法只有在线程重新获得锁时才有返回值
50. // 在超时等待时间内就获得了锁,返回结果为 true,否则为 false
51. // 有时可以利用这个返回结果进行一个代码的分支处理
52. Monitor.Wait(monitor, 1000);
54. Console.WriteLine("孩子：开始吃第" + n.ToString() + "块蛋糕");
55. }
56. }
57. }
59. **static** **void** Main(**string**[] args)
60. {
61. MonitorSample obj = **new** MonitorSample();
62. Thread tProduce = **new** Thread(**new** ThreadStart(obj.Produce));
63. Thread tConsume = **new** Thread(**new** ThreadStart(obj.Consume));
64. tProduce.Start();
65. tConsume.Start();
66. Console.ReadLine();
67. }
68. }

   这个例子的目的是要理解 Wait 和 Pulse 如何保证线程同步的，同时要注意 Wait(obeject) 和 Wait(object,int) 方法的区别，理解它们的区别很关键的一点是要理解同步的对象包含若干引用，其中包括对当前拥有锁的线程的引用、对就绪队列（包含准备获取锁的线程）的引用和对等待队列（包含等待对象状态更改通知的线程）的引用。

**.NET 中线程同步的方式多的让人看了眼花缭乱，究竟该怎么理解？其实，抛开.NET 环境看线程同步，无非执行两种操作：   
     1. 互斥/加锁，目的是保证临界区代码操作的"原子性"；   
     2. 信号灯操作，目的是保证多个线程按照一定顺序执行，如生产者线程要先于消费者线程执行。   
.NET 中线程同步的类无非是对这两种方式的封装，目的归根结底都可以归结为实现互斥/加锁或者是信号灯这两种方式，只是它们的适用场合有所不同。**

下面我们根据类的层次结构了解 WaitHandler 及其子类。   
**1. WaitHandler**   
        WaitHandle 是 Mutex，Semaphore，EventWaitHandler，AutoResetEvent，ManualResetEvent 共同的祖先，它封装 Win32 同步句柄内核对象，也就是说是这些内核对象的托管版本。线程可以通过调用 WaitHandler 实例的方法 WaitOne 在单个等待句柄上阻止。此外，WaitHandler 类重载了静态方法，以等待所有指定的等待句柄都已收集到信号 WaitAll，或者等待某一指定的等待句柄收集到信号 WaitAny。这些方法都提供了放弃等待的超时间隔、在进入等待之前退出同步上下文的机会，并允许其它线程使用同步上下文。WaitHandler 是 C# 中的抽象类，不能实例化。   
**2. EventWaitHandler vs. ManualResetEvent vs. AutoResetEvent（同步事件）**   
        我们先看看两个子类 ManualResetEvent 和 AutoResetEvent 在.NET Framework 中的实现：   
        //.NET Framework 中 ManualResetEvent 类的实现   
        [ComVisible(true),HostProtection(SecurityAction.LinkDemand,Synchronization = true, ExternalThreading = true)]   
        public sealed class ManualResetEvent : EventWaitHandle   
        {   
            // Methods   
            public ManualResetEvent(bool initialState) : base(initialState, EventResetMode.ManualReset) {}   
        }

        //.NET Framework 中 AutoResetEvent 类的实现   
        [ComVisible(true), HostProtection(SecurityAction.LinkDemand,Synchronization = true,ExternalThreading = true)]   
        public sealed class AutoResetEvent : EventWaitHandle   
        {   
            // Methods   
            public AutoResetEvent(bool initialState) : base(initialState, EventResetMode.AutoReset) { }   
        }   
        原来 ManualResetEvent 和 AutoResetEvent 都继承自 EventWaitHandler，它们的唯一区别就在于父类 EventWaitHandler 的构造函数参数 EventResetMode 不同，这样我们只要弄清了参数 EventResetMode 值不同时，EventWaitHandler 类控制线程同步的行为有什么不同，两个子类也就清楚了。为了便于描述，我们不去介绍父类的两种模式，而直接介绍子类。

**ManualResetEvent 和 AutoResetEvent 的共同点：**   
    1） Set 方法将事件状态设置为终止状态，允许一个或多个等待线程继续；   
           Reset 方法将事件状态设置为非终止状态，导致线程阻止；   
           WaitOne 阻止当前线程，直到当前线程的 WaitHandler 收到事件信号。   
    2） 可以通过构造函数的参数值来决定其初始状态，若为 true 则事件为终止状态从而使线程为非阻塞状态，为 false 则线程为阻塞状态。   
    3） 如果某个线程调用 WaitOne 方法，则当事件状态为终止状态时，该线程会得到信号，继续向下执行。

**ManualResetEvent 和 AutoResetEvent 的不同点：**   
    1） 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；   
    2）  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。   
    3）  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。

**示例场景：**   
    张三、李四两个好朋友去餐馆吃饭，两个人点了一份宫爆鸡丁，宫爆鸡丁做好需要一段时间，张三、李四不愿傻等，都专心致志的玩起了手机游戏，心想宫爆鸡丁做好了，服务员肯定会叫我们的。服务员上菜之后，张三李四开始享用美味的饭菜，饭菜吃光了，他们再叫服务员过来买单。   
我们可以从这个场景中抽象出来三个线程：   
    张三线程、李四线程、服务员线程他们之间需要同步：   
        服务员上菜 -> 张三、李四开始享用宫爆鸡丁 -> 吃好后叫服务员过来买单。   
    这个同步用什么呢？ManualResetEvent 还是 AutoResetEvent？   
通过上面的分析不难看出，**服务员上完菜需要唤醒张三和李四这2个线程进行消费, 因此我们应该用 ManualResetEvent 进行同步**，下面是程序代码：

**[csharp]** [view plain](http://blog.csdn.net/youarenotme/article/details/77882178) [copy](http://blog.csdn.net/youarenotme/article/details/77882178)

1. **using** System.Threading;
2. **using** System;
4. **public** **class** EventWaitTest
5. {
6. **private** **string** name; //顾客姓名
7. // private static AutoResetEvent eventWait = new AutoResetEvent(false);
8. **private** **static** ManualResetEvent eventWait = **new** ManualResetEvent(**false**);
9. **private** **static** ManualResetEvent eventOver = **new** ManualResetEvent(**false**);
11. **public** EventWaitTest(**string** name)
12. {
13. **this**.name = name;
14. }
16. **public** **static** **void** Product()
17. {
18. Console.WriteLine("服务员：厨师在做菜呢，两位稍等");
19. Thread.Sleep(2000);
20. Console.WriteLine("服务员：宫爆鸡丁好了");
21. eventWait.Set();
22. **while** (**true**)
23. {
24. **if** (eventOver.WaitOne(1000, **false**))
25. {
26. Console.WriteLine("服务员：两位请买单");
27. eventOver.Reset();
28. }
29. }
30. }
32. **public** **void** Consume()
33. {
34. **while** (**true**)
35. {
36. **if** (eventWait.WaitOne(1000, **false**))
37. {
38. Console.WriteLine(**this**.name + "：开始吃宫爆鸡丁");
39. Thread.Sleep(2000);
40. Console.WriteLine(**this**.name + "：宫爆鸡丁吃光了");
41. eventWait.Reset();
42. eventOver.Set();
43. **break**;
44. }
45. **else**
46. {
47. Console.WriteLine(**this**.name + "：等着上菜无聊先玩会手机游戏");
48. }
49. }
50. }
51. }
53. **public** **class** App
54. {
55. **public** **static** **void** Main(**string**[] args)
56. {
57. EventWaitTest zhangsan = **new** EventWaitTest("张三");
58. EventWaitTest lisi = **new** EventWaitTest("李四");
59. Thread t1 = **new** Thread(**new** ThreadStart(zhangsan.Consume));
60. Thread t2 = **new** Thread(**new** ThreadStart(lisi.Consume));
61. Thread t3 = **new** Thread(**new** ThreadStart(EventWaitTest.Product));
62. t1.Start();
63. t2.Start();
64. t3.Start();
65. Console.Read();
66. }
67. }

 编译后查看运行结果，符合我们的预期，控制台输出为：服务员：厨师在做菜呢，两位稍等...... 张三：等着上菜无聊先玩会手机游戏；李四：等着上菜无聊先玩会手机游戏 张三：等着上菜无聊先玩会手机游戏 李四：等着上菜无聊先玩会手机游戏 服务员：宫爆鸡丁好了 张三：开始吃宫爆鸡丁 李四：开始吃宫爆鸡丁 张三：宫爆鸡丁吃光了 李四：宫爆鸡丁吃光了  服务员：两位请买单

       如果改用 AutoResetEvent 进行同步呢？会出现什么样的结果？恐怕张三和李四就要打起来了，一个享用了美味的宫爆鸡丁，另一个到要付账的时候却还在玩游戏。感兴趣的朋友可以把注释的 那行代码注释去掉，并把下面一行代码注释掉，运行程序看会出现怎样的结果。

**3. Mutex(互斥体）**       Mutex 和 EventWaitHandler 有着共同的父类 WaitHandler 类，它们同步的函数用法也差不多，这里不再赘述。**Mutex 的突出特点是可以跨应用程序域边界对资源进行独占访问，即可以用于同步不同进程中的线程，这种功能当然这是以牺牲更多的系统资源为代价的。**前两篇简单介绍了线程同步 lock，Monitor，同步事件 EventWaitHandler，互斥体 Mutex 的基本用法，在此基础上，我们对它们用法进行比较，并给出什么时候需要锁什么时候不需要的几点建议。最后，介绍几个 FCL 中线程安全的类，集合类的锁定方式等，做为对线程同步系列的完善和补充。    
       **1. 什么时候需要同步？**   
           lock 和 Monitor 是.NET 用一个特殊结构实现的，Monitor 对象是完全托管的、完全可移植的，并且在操作系统资源要求方面可能更为有效，同步速度较快，但不能跨进程同步。（Monitor.Enter lock 和 Monitor.Exit 方法的封装），主要作用是锁定临界区，使临界区代码只能被获得锁的线程执行。Monitor.Wait 和 Monitor.Pulse 用于线程同步，类似信号操作，个人感觉使用比较复杂，容易造成死锁。

           互斥体 Mutex 和事件对象 EventWaitHandler 属于内核对象，利用内核对象进行线程同步，线程必须要在用户模式和内核模式间切换，所以一般效率很低，但利用互斥对象和事件对象这样的内核对象，可以在多个进程中的各个线程间进行同步。互斥体 Mutex 类似于一个接力棒，拿到接力棒的线程才可以开始跑，当然接力棒一次只属于一个线程(Thread Affinity)，如果这个线程不释放接力棒(Mutex.ReleaseMutex)，那么没办法，其他所有需要接力棒运行的线程都只能等着看热闹。EventWaitHandle 类允许线程通过发信号互相通信。通常，一个或多个线程在 EventWaitHandle 上阻止，直到一个未阻止的线程调用 Set 方法，以释放一个或多个被阻止的线程。    
   **2. 什么时候需要锁定？**   
           首先要理解锁定是解决竞争条件的，也就是多个线程同时访问某个资源，造成意想不到的结果。比如最简单的情况，一个计数器，两个线程同时加一，后果就是损失了一个计数，但相当频繁的锁定又可能带来性能上的消耗，还有最可怕的情况死锁。   
           那么什么情况下我们需要使用锁，什么情况下不需要呢？   
            1） 只有共享资源才需要锁定，只有可以被多线程访问的共享资源才需要考虑锁定，比如静态变量，再比如某些缓存中的值，而属于线程内部的变量不需要锁定。   
            2） 多使用 lock,少用 Mutex 如果你一定要使用锁定，请尽量不要使用内核模块的锁定机制，比如.NET 的 Mutex，Semaphore，AutoResetEvent 和 ManuResetEvent，使用这样的机制涉及到了系统在用户模式和内核模式间的切换，性能差很多，但是他们的优点是可以跨进程同步线程，所以应该清楚的了解到他们的不同和适用范围。   
            3） 了解你的程序是怎么运行的。实际上在 web 开发中大多数逻辑都是在单个线程中展开的，一个请求都会在一个单独的线程中处理，其中的大部分变量都是属于这个线程的，根本没有必要考虑锁定，当然对于 ASP.NET 中的 Application 对象中的数据，我们就要考虑加锁了。   
            4） 把锁定交给数据库。数据库除了存储数据之外，还有一个重要的用途就是同步，数据库本身用了一套复杂的机制来保证数据的可靠和一致性，这就为我们节省了很多的精力。保证了数据源头上的同步，我们多数的精力就可以集中在缓存等其他一些资源的同步访问上了。通常，只有涉及到多个线程修改数据库中同一条记录时，我们才考虑加锁。   
            5） 业务逻辑对事务和线程安全的要求这条是最根本的东西，开发完全线程安全的程序是件很费时费力的事情，在电子商务等涉及金融系统的案例中，许多逻辑都必须严格的线程安全，所以我们不得不牺牲一些性能，和很多的开发时间来做这方面的工作。而一般的应用中，许多情况下虽然程序有竞争的危险，我们还是可以不使用锁定，比如有的时候计数器少一多一，对结果无伤大雅的情况下，我们就可以不用去管它。    
       **3. InterLocked 类**            Interlocked 类提供了同步对多个线程共享的变量的访问的方法。如果该变量位于共享内存中，则不同进程的线程就可以使用该机制。互锁操作是原子的，即整个操作是不能由相同变量上的另一个互锁操作所中断的单元。这在抢先多线程操作系统中是很重要的，在这样的操作系统中，线程可以在从某个内存地址加载值之后但是在有机会更改和存储该值之前被挂起。   
            我们来看一个 InterLock.Increment() 的例子，该方法以原子的形式递增指定变量并存储结果，示例如下：  

**[csharp]** [view plain](http://blog.csdn.net/youarenotme/article/details/77882178) [copy](http://blog.csdn.net/youarenotme/article/details/77882178)

1. **using** System.Threading;
2. **using** System;
3. **class** InterLockedTest
4. {
5. **public** **static** Int64 i = 0;
7. **public** **static** **void** Add()
8. {
9. **for** (**int** i = 0; i < 100000000; i++)
10. {
11. Interlocked.Increment(**ref** InterLockedTest.i);
12. //InterLockedTest.i = InterLockedTest.i + 1;
13. }
14. }
16. **public** **static** **void** Main(**string**[] args)
17. {
18. Thread t1 = **new** Thread(**new** ThreadStart(InterLockedTest.Add));
19. Thread t2 = **new** Thread(**new** ThreadStart(InterLockedTest.Add));
20. t1.Start();
21. t2.Start();
22. t1.Join();
23. t2.Join();
24. Console.WriteLine(InterLockedTest.i.ToString());
25. Console.Read();
26. }
27. }

输出结果： 200000000    
       如果 InterLockedTest.Add()方法中用注释掉的语句代替 Interlocked.Increment() 方法，结果将不可预知！每次执行结果不同。InterLockedTest.Add() 方法保证了加 1 操作的原子性，功能上相当于自动给加操作使用了 lock 锁。同时我们也注意到 InterLockedTest.Add() 用时比直接用 + 号加 1 要耗时的多，所以说加锁资源损耗还是很明显的。

       **4. 集合类的同步**    
       .NET 在一些集合类，如 Queue、ArrayList、HashTable 和 Stack，已经提供了一个供 lock 使用的对象 SyncRoot。用 Reflector 查看了 SyncRoot 属性（Stack.SynchRoot 略有不同）的源码如下：  
                public virtual object SyncRoot   
                {   
                    get   
                    {   
                        if (this.\_syncRoot == null)   
                        {   
                            //如果\_syncRoot 和 null 相等，将 new object 赋值给\_syncRoot   
                            //Interlocked.CompareExchange 方法保证多个线程在使用 syncRoot 时是线程安全的   
                            Interlocked.CompareExchange(ref this.\_syncRoot, new object (), null);   
                        }   
                        return this.\_syncRoot;   
                    }   
                }    
        这里要特别注意的是 MSDN 提到：从头到尾对一个集合进行枚举本质上并不是一个线程安全的过程。即使一个集合已进行同步，其他线程仍可以修改该集合，这将导致枚举数引发异常。若要在枚举过程中保证线程安全，可以在整个枚举过程中锁定集合，或者捕捉由于其他线程进行的更改而引发的异常。应该使用下面的代码：   
                Queue q = new Queue();   
                lock (q.SyncRoot)   
                {   
                    foreach (object item in q)   
                    {   
                        //do something   
                    }   
                }    
        还有一点需要说明的是，集合类提供了一个是和同步相关的方法 Synchronized， 该方法返回一个对应的集合类的 wrapper 类，该类是线程安全的，因为他的大部分方法都用 lock 进行了同步处理。如HashTable的Synchronized 返回一个新的线程安全的 HashTable 实例，代码如下：

        // 在多线程环境中只要我们用下面的方式实例化 HashTable 就可以了   
            Hashtable ht = Hashtable.Synchronized(new Hashtable());   
        // 以下代码是.NET Framework Class Library 实现，增加对 Synchronized 的认识   
            [HostProtection(SecurityAction.LinkDemand, Synchronization=true)]   
            public static Hashtable Synchronized(Hashtable table)   
            {   
                if (table == null)   
                {   
                    throw new ArgumentNullException("table");   
                }   
                return new SyncHashtable(table);   
            }   
        // SyncHashtable 的几个常用方法，我们可以看到内部实现都加了 lock 关键字保证线程安全   
            public override void Add(object key, object value)   
            {   
                lock (this.\_table.SyncRoot)   
                {   
                    this.\_table.Add(key, value);   
                }   
            }   
            public override void Clear()   
            {   
                lock (this.\_table.SyncRoot)   
                    {   
                        this.\_table.Clear();   
                    }   
            }   
            public override void Remove(object key)   
            {   
                lock (this.\_table.SyncRoot)   
                {   
                    this.\_table.Remove(key);   
                }   
            }   
    线程同步是一个非常复杂的话题，这些同步方法的使用场景是怎样的？究竟有哪些细微的差别？还有待于进一步的学习和实践。

## C#中的lock、Monitor、Mutex学习笔记

线程：线程是进程的独立执行单元，每一个进程都有一个主线程，除了主线程可以包含其他的线程。

多线程的意义：多线程有助于改善程序的总体响应性，提高CPU的效率。

多线程的应用程序域是相当不稳定的，因为多个线程在同一时间内都能运行共享的功能模块。为了保护应用程序的资源不被破坏，为多线程程序提供了三种加锁的机制，分别是：Monitor类、Lock关键字和Mutex类。

**1. lock**

lock实现的功能是：使后进入的线程不会中断当前的线程，而是等待当前线程结束后再继续执行。

应用：

复制代码代码如下:

   private Object thisLock=new object();

   lock(thisLock){  
               //锁定的代码块  
    }

注意：避免锁定 public 类型，否则实例将超出代码的控制范围。

常见的结构 lock (this)、lock (typeof (MyType)) 和 lock ("myLock")  
违反此准则：如果实例可以被公共访问，将出现 lock (this) 问题。  
如果 MyType 可以被公共访问，将出现 lock (typeof (MyType)) 问题。  
由于进程中使用同一字符串的任何其他代码将共享同一个锁，所以出现 lock(“myLock”) 问题。  
最佳做法是定义 private 对象来锁定, 或 private static 对象变量来保护所有实例所共有的数据。  
下面举例说明lock的应用：  
下面的例子中创建了5个次线程，次线程完成的任务就是：输出线程编码，延迟1秒，然后输出当时的时间

Example：

复制代码代码如下:

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading;

namespace ConsoleApplication5 {  
    class Program {  
        static void Main(string[] args) {  
        Console.WriteLine ("程序开始时间："+DateTime.Now .ToString());  
        Example ex=new Example ();  
        Thread []threads=new  Thread[5];  
        for (int i=0;i<5;i++)  
        {  
        threads[i]=new Thread (new ThreadStart(ex.OutPut));  
        threads[i].Name =string.Format ("Worker thread#{0}",i) ;  
        }  
        foreach(Thread t in threads){  
        t.Start();  
        }  
        Console.WriteLine("主线程最后一句代码！"+DateTime.Now.ToString());  
        }  
    }  
    class Example{  
    private static object  thisLock=new object ();  
    public void OutPut()  
    {      
    lock(thisLock){  
    Console .WriteLine("->{0}",Thread.CurrentThread.Name);  
    Thread.Sleep(1000);  
    Console.WriteLine(DateTime.Now);  
    }  
    }  
      
    }  
}

实验结果如下：

           图1：加lock

      图2：没有lock

从实验结果可以看出，加lock后，程序每次只能执行一个线程，只有当前线程执行完了，才会执行下一个线程未加lock，程序执行混乱，容易阻塞。

**2. Monitor**

lock是对Monitor的Enter和Exit的一个封装，因此Monitor类的Enter()和Exit()方法的组合使用可以用lock关键字替代。

Monitor类除了具有lock的功能外，还有以下功能：

TryEnter()解决长期死等的问题，如果一个并发经常发生，并且持续时间很长，使用TryEnter，可以有效防止死锁或者长时间 的等待。

Wait()释放对象上的锁，以便允许其他线程锁定和访问该对象。在其他线程访问对象时，调用线程将等待。脉冲信号用于通知等待线程有关对象状态的更改。

Pulse(),PulseAll()向一个或多个等待线程发送信号。该信号通知等待线程锁定对象的状态已更改，并且锁的所有者准备释放该锁。等待线程被放置在对象的就绪队列中以便它可以最后接收对象锁。一旦线程拥有了锁，它就可以检查对象的新状态以查看是否达到所需状态。

注意：Pulse、PulseAll和Wait方法必须从同步的代码块内调用。

**3. Mutex(互斥体)**

Mutex的突出特点是可以跨应用程序域边界对资源进行独占访问，即可以用于同步不同进程中的线程，这种功能是以牺牲更多的系统资源为代价的。

互斥体Mutex和事件对象EventWaitHandler属于内核对象，利用内核对象进行线程同步，线程必须要在用户模式和内核模式间切换，所以一般效率很低，但利用互斥对象和事件对象这样的内核对象，可以在多个进程中的各个线程间进行同步。

互斥体Mutex类似于一个接力棒，拿到接力棒的线程才可以开始跑，当然接力棒一次只属于一个线程(Thread Affinity)，如果这个线程不释放接力棒(Mutex.ReleaseMutex)，那么其他所有需要接力棒运行的线程只能等着看热闹。