**lock和Monitor的区别**

一、lock的底层本身是Monitor来实现的，所以Monitor可以实现lock的所有功能。   
二、Monitor有TryEnter的功能，可以防止出现死锁的问题，lock没有。

**Mutex和其他两者的区别**

个人测试三个都是在限制线程之外的互斥，线程之内，都不限制，同一个线程如果被lock两次。是不会出现死锁的。所以Mutex本身可以实现lock和Monitor所有的操作。至少从功能上讲是这样的。

但是Mutex是内核级别的，消耗较大的资源，不适合频繁的操作，会降低操作的效率。所以一般被调用部分的资源锁，常常用lock或者Monitor，可以提高效率。而线程和线程间的协调，可以用Mutex，因为相互互斥切换的机会会大大的降低，效率就不再那么的重要了。

Mutex本身是可以系统级别的，所以是可以跨越进程的。比如我们要实现一个软件不能同时打开两次，那么Mutex是可以实现的，而lock和monitor是无法实现的

在多线程中，为了使数据保持一致性必须要对数据或是访问数据的函数加锁，在数据库中这是很常见的，但是在程序中由于大部分都是单线程的程序，所以没有加锁的必要，但是在多线程中，为了保持数据的同步，一定要加锁，好在Framework中已经为我们提供了三个加锁的机制，分别是Monitor类、Lock关键字和Mutex类。

        其中Lock关键词用法比较简单，Monitor类和Lock的用法差不多。这两个都是锁定数据或是锁定被调用的函数。而Mutex则多用于锁定多线程间的同步调用。简单的说，Monitor和Lock多用于锁定被调用端，而Mutex则多用锁定调用端。  
例如下面程序：由于这种程序都是毫秒级的，所以运行下面的程序可能在不同的机器上有不同的结果，在同一台机器上不同时刻运行也有不同的结果，我的测试环境为vs2005, windowsXp , CPU3.0 , 1 G monery。  
        程序中有两个线程thread1、thread2和一个TestFunc函数，TestFunc会打印出调用它的线程名和调用的时间(mm级的)，两个线程分别以30mm和100mm来调用TestFunc这个函数。TestFunc执行的时间为50mm。程序如下：  
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.Threading;  
namespace MonitorLockMutex  
{  
    class Program  
    {  
        #region variable  
        Thread thread1 = null;  
        Thread thread2 = null;  
        Mutex mutex = null;  
        #endregion  
        static void Main(string[] args)  
        {  
            Program p = new Program();  
            p.RunThread();  
            Console.ReadLine();  
        }  
        public Program()  
        {  
            mutex = new Mutex();  
            thread1 = new Thread(new ThreadStart(thread1Func));  
            thread2 = new Thread(new ThreadStart(thread2Func));  
        }  
        public void RunThread()  
        {  
            thread1.Start();  
            thread2.Start();  
        }  
        private void thread1Func()  
        {  
            for (int count = 0; count < 10; count++)  
            {  
                TestFunc("Thread1 have run " + count.ToString() + " times");  
                Thread.Sleep(30);  
            }  
        }  
        private void thread2Func()  
        {  
            for (int count = 0; count < 10; count++)  
            {  
                TestFunc("Thread2 have run " + count.ToString() + " times");  
                Thread.Sleep(100);  
            }  
        }  
        private void TestFunc(string str)  
        {  
            Console.WriteLine("{0} {1}", str, System.DateTime.Now.Millisecond.ToString());  
            Thread.Sleep(50);  
        }  
    }  
}  
运行结果如下：

        可以看出如果不加锁的话，这两个线程基本上是按照各自的时间间隔+TestFunc的执行时间(50mm)对TestFunc函数进行读取。因为线程在开始时需要分配内存，所以第0次的调用不准确，从第1~9次的调用可以看出，thread1的执行间隔约是80mm，thread2的执行间隔约是150mm。  
现在将TestFunc修改如下：  
private void TestFunc(string str)  
{  
   lock (this)  
   {  
      Console.WriteLine("{0} {1}", str, System.DateTime.Now.Millisecond.ToString());  
      Thread.Sleep(50);  
   }  
}  
或者是用Monitor也是一样的，如下：  
private void TestFunc(string str)  
{  
      Monitor.Enter(this);  
      Console.WriteLine("{0} {1}", str, System.DateTime.Now.Millisecond.ToString());  
      Thread.Sleep(50);  
      Monitor.Exit(this);  
}  
其中Enter和Exit都是Monitor中的静态方法。  
运行Lock结果如下：

        让我们分析一下结果，同样从第1次开始。相同线程间的调用时间间隔为线程执行时间+TestFunc调用时间，不同线程间的调用时间间隔为TestFunc调用时间。例如：连续两次调用thread1之间的时间间隔约为30+50=80;连续两次调用thread2之间的时间间隔约为100+50=150mm。调用thread1和thread2之间的时间间隔为50mm。因为TestFunc被lock住了，所以一个thread调用TestFunc后，当其它的线程也同时调用TestFunc时，后来的线程即进被排到等待队列中等待，直到拥有访问权的线程释放这个资源为止。  
        这就是锁定被调用函数的特性，即只能保证每次被一个线程调用，线程优先级高的调用的次数就多，低的就少，这就是所谓的强占式。  
        下面让我们看看Mutex类的使用方法，以及与Monitor和Lock的区别。  
将代码修改如下：  
        private void thread1Func()  
        {  
            for (int count = 0; count < 10; count++)  
            {  
                mutex.WaitOne();  
                TestFunc("Thread1 have run " + count.ToString() + " times");  
                mutex.ReleaseMutex();  
            }  
        }  
  
        private void thread2Func()  
        {  
            for (int count = 0; count < 10; count++)  
            {  
                mutex.WaitOne();  
                TestFunc("Thread2 have run " + count.ToString() + " times");  
                mutex.ReleaseMutex();  
            }  
        }  
  
        private void TestFunc(string str)  
        {  
            Console.WriteLine("{0} {1}", str, System.DateTime.Now.Millisecond.ToString());  
            Thread.Sleep(50);  
        }  
运行结果如下：

        可以看出，Mutex只能互斥线程间的调用，但是不能互斥本线程的重复调用，即thread1中waitOne()只对thread2中的waitOne()起到互斥的作用，但是thread1并不受本wainOne()的影响，可以调用多次，只是在调用结束后调用相同次数的ReleaseMutex()就可以了。  
        那么如何使线程按照调用顺序来依次执行呢？其实把lock和Mutex结合起来使用就可以了，改代码如下：  
        private void thread1Func()  
        {  
            for (int count = 0; count < 10; count++)  
            {  
                lock (this)  
                {  
                    mutex.WaitOne();  
                    TestFunc("Thread1 have run " + count.ToString() + " times");  
                    mutex.ReleaseMutex();  
                }  
            }  
        }  
  
        private void thread2Func()  
        {  
            for (int count = 0; count < 10; count++)  
            {  
                lock (this)  
                {  
                    mutex.WaitOne();  
                    TestFunc("Thread2 have run " + count.ToString() + " times");  
                    mutex.ReleaseMutex();  
                }  
            }  
        }