c# 线程同步各类锁

1）原子操作（Interlocked）：所有方法都是执行一次原子读取或一次写入操作。

2）lock()语句：避免锁定public类型，否则实例将超出代码控制的范围，定义private对象来锁定。

3）Monitor实现线程同步

通过Monitor.Enter() 和 Monitor.Exit()实现排它锁的获取和释放，获取之后独占资源，不允许其他线程访问。

还有一个TryEnter方法，请求不到资源时不会阻塞等待，可以设置超时时间，获取不到直接返回false。

4）ReaderWriterLock

当对资源操作读多写少的时候，为了提高资源的利用率，让读操作锁为共享锁，多个线程可以并发读取资源，而写操作为独占锁，只允许一个线程操作。

5）事件（Event）类实现同步

事件类有两种状态，终止状态和非终止状态，终止状态时调用WaitOne可以请求成功，通过Set将时间状态设置为终止状态。

1）AutoResetEvent（自动重置事件）

2）ManualResetEvent（手动重置事件）

6）信号量（Semaphore）

信号量是由内核对象维护的int变量，为0时，线程阻塞，大于0时解除阻塞，当一个信号量上的等待线程解除阻塞后，信号量计数+1。

线程通过WaitOne将信号量减1，通过Release将信号量加1，使用很简单。

7）互斥体（Mutex）

独占资源，用法与Semaphore相似。

8）跨进程间的同步

通过设置同步对象的名称就可以实现系统级的同步，不同应用程序通过同步对象的名称识别不同同步对象。

C#线程锁使用全功略

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.几种同步方法的区别

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。lock（Monitor.Enter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知道能等着看热 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，最简单的情况是，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：

```
class InterLockedTest
{
    public static Int64 i = 0;

    public static void Add()
    {
        for (int i = 0; i < 100000000; i++)
        {
            Interlocked.Increment(ref InterLockedTest.i);
            //InterLockedTest.i = InterLockedTest.i + 1;
        }
    }



    public static void Main(string[] args)
    {
        Thread t1 = new Thread(new ThreadStart(InterLockedTest.Add));
        Thread t2 = new Thread(new ThreadStart(InterLockedTest.Add));

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

        Console.WriteLine(InterLockedTest.i.ToString());
        Console.Read();
    }
}
```

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。

另外InterLockedTest类还有几个常用方法，具体用法可以参考MSDN上的介绍。

4.集合类的同步

.NET在一些集合类，比如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使用lock示例

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关键字进行了同步处理。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，这里只是根据公司的一个项目把相关的知识整理出来，作为工作的一种总结。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。