**C#编程总结（五）多线程带给我们的一些思考**

**如有不妥之处，欢迎批评指正。**

**1、什么时候使用多线程？**

     这个问题，对于系统架构师、设计者、程序员，都是首先要面对的一个问题。

     在什么时候使用多线程技术？

     在许多常见的情况下，可以使用多线程处理来显著提高应用程序的响应能力和可用性。

     上一章，我们讲了几个多线程的应用案例，主要的应用场景也做了介绍。这里不再赘述。

<http://www.cnblogs.com/yank/p/3232955.html>

**2、如何才能保证线程安全？**

     使用多线程，这是一个必须要弄清的问题。只有了解了多线程对结构和程序的影响，才能真正会使用多线程，使其发挥应有的效果。

     为什么应用多线程就不安全了呢？

     线程安全的一个判定指标，线程之间有没有临界资源，如果有临界资源，且没有采用合理的同步机制，就会出现多个线程竞争一个资源，如若多个线程都在为得不到所需的资源，则会发生死锁。死锁，线程就会彼此僵持，系统停滞不前，如果后果严重，则直接导致系统崩溃。常见的案例有：生产者与消费者问题、哲学家就餐问题等。

      咱就根据哲学家就餐问题做个简化：两个人去餐馆吃饭，由于资源紧张，只有一双筷子，每个人都饿了，都想吃饭，且同时去抢筷子，势均力敌，两人每人抢到一根筷子，只有使用一双筷子才能吃饭。这时你会说了，我可以用手抓着吃，呵呵。如果是刚出锅的饺子，怕你抓不起来。两个人只能面面相觑，大眼瞪小眼，就是吃不上。如果如果僵持个一年半载，都饿死了。哈哈。如果我们给一个约定，在拿筷子时，一下拿到一双，且吃完就交换给对方。则两个人都高高兴兴吃上饭了。筷子就是临界资源。当然，在两个人僵持的时候，可以进行外部干预，使得两个人都有饭吃。比如：强制一方将筷子空闲出来，则另一方就饭吃了。吃完了筷子空闲出来，则另一个人也有饭吃了。

     只要我们处理好临界资源问题，也就解决了线程安全问题。

     使用多线程，未必必须要做好线程同步，但是如果有临界资源，则必须进行线程同步处理。

**3、 如何能写出线程安全的代码？**

     在OOP中，程序员使用的无非是：变量、对象（属性、方法）、类型等等。

     1）变量

     变量包括值类型和引用类型。

     值类型是线程安全的，但是如果作为对象的属性，值类型就被附加到对象上，需要参考对象的线程安全性。

     引用类型，这里要注意的是，对于引用对象，他包括了引用和对象实例两部分，实例需要通过对其存储位置的引用来访问，对于

       private Object o = new Object()，

     其实可以分解为两句话：

        private Object o；

        o = new Object()；

      其中private Object o是定义了对象的引用，也就是记录对象实例的指针，而不是对象本身。这个引用存储于堆栈中，占用4个字节；当没有使用o = new Object()时，引用本身的值为null，也就是不指向任何有效位置；当o = new Object()后，才真正根据对象的大小，在托管堆中分配空间给对象实例，然后将实例的指针位置赋值给前面的引用。这才完成一个对象的实例化。

      引用类型的安全性，在于：可以由多个引用，同时指向一个内存地址。如果一个引用被修改，另一个也会修改。

[复制代码](javascript:void(0);)

using System;

namespace VariableSample

{

class Program

{

static void Main(string[] args)

{

Box b1 = new Box();

b1.Name = "BigBox";

Console.WriteLine("Create Box b1.");

Console.WriteLine("Box: b1'Name is {0}.", b1.Name);

Console.WriteLine("Create same Box b2.");

Box b2 = b1;

b2.Name = "LittleBox";

Console.WriteLine("Box: b2's Name is {0}.",b2.Name);

Console.WriteLine("Box: b1's Name is {0}.", b1.Name);

Console.ReadKey();

}

}

/// <summary>

/// 盒子

/// </summary>

public class Box

{

/// <summary>

/// 名称

/// </summary>

public string Name

{

get;

set;

}

}

}

[复制代码](javascript:void(0);)

输出结果：

Create Box b1.

Box: b1'Name is BigBox.

Create same Box b2.

Box: b2's Name is LittleBox.

Box: b1's Name is LittleBox.

    这里对盒子名字修改，是对两个引用对象修改，其实我们可以将其设计为两个多线程对对象的修改。这里必然存在线程安全性问题。

**总之，变量的线程安全性与变量的作用域有关。**

     2）对象

     对象是类型的实例

     在创建对象时，会单独有内存区域存储对象的属性和方法。所以，一个类型的多个实例，在执行时，只要没有静态变量的参与，应该都是线程安全的。

这跟我们调试状态下，是不一样的。调试状态下，如果多个线程都创建某实例的对象，每个对象都调用自身方法，在调试是，会发现是访问的同一个代码，多个线程是有冲突的。但是，真正的运行环境是线程安全的。

      以销售员为例，假设产品是充足的，多个销售员，销售产品，调用方法:Sale()，其是线程安全的。

      但是，如果涉及到仓库，必须仓库有足够的产品才能进行销售，这时，多个销售人员就有了临界资源：仓库。

      在这里我们只讨论对象的普通方法。至于方法传入的参数，以及方法内对静态变量操作的，这里需要根据参数和静态变量来判定方法的线程安全性。

      销售员案例：

[复制代码](javascript:void(0);)

using System;

using System.Threading;

namespace MutiThreadSample.Sale

{

/// <summary>

/// 销售

/// </summary>

public class Saler

{

/// <summary>

/// 名称

/// </summary>

public string Name { get; set; }

/// <summary>

/// 间隔时间

/// </summary>

public int IntervalTime { get; set; }

/// <summary>

/// 单位时间销售运量

/// </summary>

public int SaleAmount { get; set; }

/// <summary>

/// 销售

/// </summary>

public void Sale()

{

Console.WriteLine("销售：{0} 于 {1} 销售产品 {2}", this.Name, DateTime.Now.Millisecond, this.SaleAmount);

Thread.Sleep(IntervalTime);

}

/// <summary>

/// 销售

/// </summary>

/// <param name="interval">时间间隔</param>

public void Sale(object obj)

{

WHouseThreadParameter parameter = obj as WHouseThreadParameter;

if (parameter != null)

{

while (parameter.WHouse != null && parameter.WHouse.CanOut(this.SaleAmount))

{

parameter.WHouse.Outgoing(this.SaleAmount);

Console.WriteLine("Thread{0}, 销售：{1} 于 {2} 销售出库产品 {3}", Thread.CurrentThread.Name, this.Name, DateTime.Now.Millisecond, this.SaleAmount);

Thread.Sleep(this.IntervalTime);

}

}

}

}

}

[复制代码](javascript:void(0);)

      3）类型

      已经讲了类的实例--对象的多线程安全性问题。这里只讨论类型的静态变量和静态方法。

      当静态类被访问的时候，CLR会调用类的静态构造器（类型构造器），创建静态类的类型对象，CLR希望确保每个应用程序域内只执行一次类型构造器，为了做到这一点，在调用类型构造器时，CLR会为静态类加一个互斥的线程同步锁，因此，如果多个线程试图同时调用某个类型的静态构造器时，那么只有一个线程可以获得对静态类的访问权，其他的线程都被阻塞。第一个线程执行完 类型构造器的代码并释放构造器之后，其他阻塞的线程被唤醒，然后发现构造器被执行过，因此，这些线程不再执行构造器，只是从构造器简单的返回。如果再一次调用这些方法，CLR就会意识到类型构造器被执行过，从而不会在被调用。

      调用类中的静态方法，或者访问类中的静态成员变量，过程同上，所以说静态类是线程安全的。

      最简单的例子，就是数据库操作帮助类。这个类的方法和属性是线程安全的。

[复制代码](javascript:void(0);)

using System;

namespace MutiThreadSample.Static

{

public class SqlHelper

{

/// <summary>

/// 数据库连接

/// </summary>

private static readonly string ConnectionString = "";

/// <summary>

/// 执行数据库命令

/// </summary>

/// <param name="sql">SQL语句</param>

public static void ExcuteNonQuery(string sql)

{

//执行数据操作，比如新增、编辑、删除

}

}

}

[复制代码](javascript:void(0);)

       但是，对于静态变量其线程安全性是相对的，如果多个线程来修改静态变量，这就不一定是线程安全的。而静态方法的线程安全性，直接跟传入的参数有关。

**总之:**

      针对变量、对象、类型，说线程安全性，比较笼统，在这里，主要是想让大家明白，哪些地方需要注意线程安全性。对于变量、对象（属性、方法）、静态变量、静态方法，其线程安全性是相对的，需要根据实际情况而定。

     万剑不离其宗，**其判定标准：是否有临界资源。**

**4、集合类型是线程安全的吗？**

      常用的集合类型有List、Dictionary、HashTable、HashMap等。在编码中，集合应用很广泛中，常用集合来自定义Cache，这时候必须考虑线程同步问题。

      默认情况下集合不是线程安全的。在System.Collections 命名空间中只有几个类提供Synchronize方法，该方法能够超越集合创建线程安全包装。但是，System.Collections命名空间中的所有类都提供SyncRoot属性，可供派生类创建自己的线程安全包装。还提供了IsSynchronized属性以确定集合是否是线程安全的。但是ICollection泛型接口中不提供同步功能，非泛型接口支持这个功能。

**Dictionary(MSDN解释)**

     此类型的公共静态（在 Visual Basic 中为 Shared）成员是线程安全的。 但不保证所有实例成员都是线程安全的。  
     只要不修改该集合，Dictionary<TKey, TValue> 就可以同时支持多个阅读器。 即便如此，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。 当出现枚举与写访问互相争用这种极少发生的情况时，必须在整个枚举过程中锁定集合。 若允许多个线程对集合执行读写操作，您必须实现自己的同步。

      很多集合类型都和Dictionary类似。默认情况下是线程不安全的。当然微软也提供了线程安全的Hashtable.

**HashTable**

      Hashtable 是线程安全的，可由多个读取器线程和一个写入线程使用。 多线程使用时，如果只有一个线程执行写入（更新）操作，则它是线程安全的，从而允许进行无锁定的读取（若编写器序列化为 Hashtable）。 若要支持多个编写器，如果没有任何线程在读取 Hashtable 对象，则对 Hashtable 的所有操作都必须通过 Synchronized 方法返回的包装完成。

      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。 即使某个集合已同步，其他线程仍可以修改该集合，这会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。

     线程安全起见请使用以下方法声明

/// <summary>

/// Syncronized方法用来创造一个新的对象的线程安全包装

/// </summary>

private Hashtable hashtable = Hashtable.Synchronized(new Hashtable());

    在枚举读取时，加lock，这里lock其同步对象SyncRoot

[复制代码](javascript:void(0);)

/// <summary>

/// 读取

/// </summary>

public void Read()

{

lock(hashtable.SyncRoot)

{

foreach (var item in hashtable.Keys)

{

Console.WriteLine("Key:{0}",item);

}

}

}

[复制代码](javascript:void(0);)

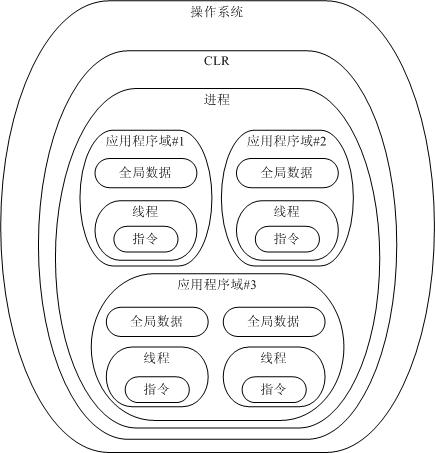
**5、如何进行线程同步？**

在第三章做了具体讲解，并介绍了常用的几种线程同步的方法，具体可见：

<http://www.cnblogs.com/yank/p/3227324.html>

**6、IIS多线程应用**

IIS有多个应用程序池，每个应用程序池对应一个w3wp.exe的进程，每个应用程序池对应多个应用程序，每个应用程序对应一个应用程序域，应用程序域中包含了共享数据和多个线程，线程中有指定操作。由下图我们就能清晰的了解整个结构。



**7、如何有效使用多线程？**

     线程可以大大提高应用程序的可用性和性能，但是多线程也给我们带来一些新的挑战，要不要使用多线程，如何使用多线程，需要根据实际情况而定。

     1）复杂度

     使用多线程，可能使得应用程序复杂度明显提高，特别是要处理线程同步和死锁问题。需要仔细地评估应该在何处使用多线程和如何使用多线程，这样就可以获得最大的好处，而无需创建不必要的复杂并难于调试的应用程序。

     2)数量

     线程不易过多，线程的数量与服务器配置（多核、多处理器）、业务处理具体过程，都有直接关系。线程量过少，不能充分发挥服务器的处理能力，也不能有效改善事务的处理效率。线程量过多，需要花费大量的时间来进行线程控制，最后得不偿失。可以根据实际情况，通过检验测试，设定一个特定的合理的范围。

      3）同步和异步调用之间的选择  
      应用程序既可以进行同步调用，也可以进行异步调用。同步 调用在继续之前等待响应或返回值。如果不允许调用继续，就说调用被阻塞 了。异步或非阻塞 调用不等待响应。异步调用是通过使用单独的线程执行的。原始线程启动异步调用，异步调用使用另一个线程执行请求，而与此同时原始的线程继续处理。

      4）前台线程和后台线程之间的选择  
      .NET Framework 中的所有线程都被指定为前台线程或后台线程。这两种线程唯一的区别是 — 后台线程不会阻止进程终止。在属于一个进程的所有前台线程终止之后，公共语言运行库 (CLR) 就会结束进程，从而终止仍在运行的任何后台线程。  
      在默认情况下，通过创建并启动新的 Thread 对象生成的所有线程都是前台线程，而从非托管代码进入托管执行环境中的所有线程都标记为后台线程。然而，通过修改 Thread.IsBackground 属性，可以指定一个线程是前台线程还是后台线程。通过将 Thread.IsBackground 设置为 true，可以将一个线程指定为后台线程；通过将 Thread.IsBackground 设置为 false，可以将一个线程指定为前台线程。

     在大多数应用程序中，您会选择将不同的线程设置成前台线程或后台线程。通常，应该将被动侦听活动的线程设置为后台线程，而将负责发送数据的线程设置为前台线程，这样，在所有的数据发送完毕之前该线程不会被终止。只有在确认线程被系统随意终止没有不利影响时，才应该使用后台线程。如果线程正在执行必须完成的敏感操作或事务操作，或者需要控制关闭线程的方式以便释放重要资源，则使用前台线程。

**8、何时使用线程池（ThreadPool）？**

      到现在为止，您可能会认识到许多应用程序都会从多线程处理中受益。然而，线程管理并不仅仅是每次想要执行一个不同的任务就创建一个新线程的问题。有太多的线程可能会使得应用程序耗费一些不必要的系统资源，特别是，如果有大量短期运行的操作，而所有这些操作都运行在单独线程上。另外，显式地管理大量的线程可能是非常复杂的。  
      线程池化技术通过给应用程序提供由系统管理的辅助线程池解决了这些问题，从而使得您可以将注意力集中在应用程序任务上而不是线程管理上。  
      在需要时，可以由应用程序将线程添加到线程池中。当 CLR 最初启动时，线程池没有包含额外的线程。然而，当应用程序请求线程时，它们就会被动态创建并存储在该池中。如果线程在一段时间内没有使用，这些线程就可能会被处置，因此线程池是根据应用程序的要求缩小或扩大的。  
      注意：每个进程都创建一个线程池，因此，如果您在同一个进程内运行几个应用程序域，则一个应用程序域中的错误可能会影响相同进程内的其他应用程序域，因为它们都使用相同的线程池。

线程池由两种类型的线程组成：

* **辅助线程**。辅助线程是标准系统池的一部分。它们是由 .NET Framework 管理的标准线程，大多数功能都在它们上面执行。
* **完成端口线程**.这种线程用于异步 I/O 操作（通过使用 IOCompletionPorts API）

对于每个计算机处理器，线程池都默认包含 25 个线程。如果所有的 25 个线程都在被使用，则附加的请求将排入队列，直到有一个线程变得可用为止。每个线程都使用默认堆栈大小，并按默认的优先级运行。

下面代码示例说明了线程池的使用。

private void ThreadPoolExample()

{

WaitCallback callback = new WaitCallback( ThreadProc );

ThreadPool.QueueUserWorkItem( callback );

}

在前面的代码中，首先创建一个委托来引用您想要在辅助线程中执行的代码。.NET Framework 定义了 **WaitCallback** 委托，该委托引用的方法接受一个对象参数并且没有返回值。下面的方法实现您想要执行的代码。

private void ThreadProc( Object stateInfo )

{

// Do something on worker thread.

}

可以将单个对象参数传递给 **ThreadProc** 方法，方法是将其指定为 **QueueUserWorkItem** 方法调用中的第二个参数。在前面的示例中，没有给 **ThreadProc** 方法传递参数，因此 **stateInfo** 参数为空。

在下面的情况下，使用 **ThreadPool** 类：

* 有大量小的独立任务要在后台执行。
* 不需要对用来执行任务的线程进行精细控制。

    Thread是显示来管理线程。只要有可能，就应该使用 **ThreadPool** 类来创建线程。

在下面的情况下，使用 **Thread** 对象：

* 需要具有特定优先级的任务。
* 有可能运行很长时间的任务（这样可能阻塞其他任务）。
* 需要确保只有一个线程可以访问特定的程序集。
* 需要有与线程相关的稳定标识。