**C#多线程学习(一) 多线程的相关概念**

**什么是进程？**

      当一个程序开始运行时，它就是一个进程，进程包括运行中的程序和程序所使用到的内存和系统资源。而一个进程又是由多个线程所组成的。

**什么是线程？**

      线程是程序中的一个执行流，每个线程都有自己的专有寄存器(栈指针、程序计数器等)，但代码区是共享的，即不同的线程可以执行同样的函数。

**什么是多线程？**

     多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务，也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

**多线程的好处：**

      可以提高CPU的利用率。在多线程程序中，一个线程必须等待的时候，CPU可以运行其它的线程而不是等待，这样就大大提高了程序的效率。

**多线程的不利方面：**

     线程也是程序，所以线程需要占用内存，线程越多占用内存也越多； 多线程需要协调和管理，所以需要CPU时间跟踪线程； 线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题；线程太多会导致控制太复杂，最终可能造成很多Bug；

     接下来将对C#编程中的多线程机制进行探讨。为了省去创建GUI那些繁琐的步骤，更清晰地逼近线程的本质，接下来的所有程序都是控制台程序，程序最后的Console.ReadLine()是为了使程序中途停下来，以便看清楚执行过程中的输出。

**任何程序在执行时，至少有一个主线程。**

一个直观印象的线程示例：

[复制代码](javascript:void(0);)

using System;

using System.Threading;

namespace ThreadTest

{

class RunIt

{

[STAThread]

static void Main(string[] args)

{

//给当前线程起名为"System Thread"

Thread.CurrentThread.Name="System Thread";

Console.WriteLine(Thread.CurrentThread.Name+"'Status:"+

Thread.CurrentThread.ThreadState);

Console.ReadLine();

}

}

}

[复制代码](javascript:void(0);)

**输出如下：**

**System Thread's Status:Running**

     在这里，我们通过Thread类的静态属性CurrentThread获取了当前执行的线程，对其Name属性赋值“System Thread”，最后还输出了它的当前状态（ThreadState）。

      所谓静态属性，就是这个类所有对象所公有的属性，不管你创建了多少个这个类的实例，但是类的静态属性在内存中只有一个。很容易理解CurrentThread为什么是静态的——虽然有多个线程同时存在，但是在某一个时刻，CPU只能执行其中一个。

在程序的头部，我们使用了如下命名空间：

**using System;**

**using System.Threading;**

      在.net framework class library中，所有与多线程机制应用相关的类都是放在System.Threading命名空间中的。如果你想在你的应用程序中使用多线程，就必须包含这个类。

      我们通过其中提供的Thread类来创建和控制线程，ThreadPool类用于管理线程池等。（此外还提供解决了线程执行安排，死锁，线程间通讯等实际问题的机制。）

Thread类有几个至关重要的方法，描述如下：

**Start()：**启动线程；

**Sleep(int)：**静态方法，暂停当前线程指定的毫秒数；

**Abort()：**通常使用该方法来终止一个线程；

**Suspend()：**该方法并不终止未完成的线程，它仅仅挂起线程，以后还可恢复；

**Resume()：**恢复被Suspend()方法挂起的线程的执行。

**C#多线程学习(二) 如何操纵一个线程**

     下面我们就动手来创建一个线程，使用Thread类创建线程时，只需提供线程入口即可。（线程入口使程序知道该让这个线程干什么事）

     在C#中，线程入口是通过ThreadStart代理（delegate）来提供的，你可以把ThreadStart理解为一个函数指针，指向线程要执行的函数，当调用Thread.Start()方法后，线程就开始执行ThreadStart所代表或者说指向的函数。

      打开你的VS.net，新建一个控制台应用程序（Console Application），编写完全控制一个线程的代码示例：

[复制代码](javascript:void(0);)

using System;

using System.Threading;

namespace ThreadTest

{

public class Alpha

{

public void Beta()

{

while (true)

{

Console.WriteLine("Alpha.Beta is running in its own thread.");

}

}

};

public class Simple

{

public static int Main()

{

　 Console.WriteLine("Thread Start/Stop/Join Sample");

Alpha oAlpha = new Alpha();

　　 file://这里创建一个线程，使之执行Alpha类的Beta()方法

　　 Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));

　　 oThread.Start();

　　 while (!oThread.IsAlive)

　　 Thread.Sleep(1);

　　 oThread.Abort();

　　 oThread.Join();

　　 Console.WriteLine();

　　 Console.WriteLine("Alpha.Beta has finished");

　　 try

　　 {

　　　　 Console.WriteLine("Try to restart the Alpha.Beta thread");

　　　　 oThread.Start();

　　 }

　　 catch (ThreadStateException)

　　 {

　　　　 Console.Write("ThreadStateException trying to restart Alpha.Beta. ");

　　　　 Console.WriteLine("Expected since aborted threads cannot be restarted.");

　　　　 Console.ReadLine();

　　 }

　　 return 0;

　　 }

}

}

[复制代码](javascript:void(0);)

      这段程序包含两个类Alpha和Simple，在创建线程oThread时我们用指向Alpha.Beta()方法的初始化了ThreadStart代理（delegate）对象，当我们创建的线程oThread调用oThread.Start()方法启动时，实际上程序运行的是Alpha.Beta()方法：

Alpha oAlpha = new Alpha();

Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));

oThread.Start();

      然后在Main()函数的while循环中，我们使用静态方法Thread.Sleep()让主线程停了1ms，这段时间CPU转向执行线程oThread。然后我们试图用Thread.Abort()方法终止线程oThread，注意后面的oThread.Join()，Thread.Join()方法使主线程等待，直到oThread线程结束。你可以给Thread.Join()方法指定一个int型的参数作为等待的最长时间。之后，我们试图用Thread.Start()方法重新启动线程oThread，但是显然Abort()方法带来的后果是不可恢复的终止线程，所以最后程序会抛出ThreadStateException异常。

**主线程Main()函数**

     所有线程都是依附于Main()函数所在的线程的，Main()函数是C#程序的入口，起始线程可以称之为主线程。如果所有的前台线程都停止了，那么主线程可以终止，而所有的后台线程都将无条件终止。所有的线程虽然在微观上是串行执行的，但是在宏观上你完全可以认为它们在并行执行。

**Thread.ThreadState 属性**

     这个属性代表了线程运行时状态，在不同的情况下有不同的值，我们有时候可以通过对该值的判断来设计程序流程。

**ThreadState 属性的取值如下：**

**Aborted：**线程已停止；

**AbortRequested：**线程的Thread.Abort()方法已被调用，但是线程还未停止；

**Background：**线程在后台执行，与属性Thread.IsBackground有关；

**Running：**线程正在正常运行；

**Stopped：**线程已经被停止；

**StopRequested：**线程正在被要求停止；

**Suspended：**线程已经被挂起（此状态下，可以通过调用Resume()方法重新运行）；

**SuspendRequested：**线程正在要求被挂起，但是未来得及响应；

**Unstarted：**未调用Thread.Start()开始线程的运行；

**WaitSleepJoin：**线程因为调用了Wait(),Sleep()或Join()等方法处于封锁状态；

     上面提到了Background状态表示该线程在后台运行，那么后台运行的线程有什么特别的地方呢？其实后台线程跟前台线程只有一个区别，那就是后台线程不妨碍程序的终止。一旦一个进程所有的前台线程都终止后，CLR（通用语言运行环境）将通过调用任意一个存活中的后台进程的Abort()方法来彻底终止进程。

**线程的优先级**

     当线程之间争夺CPU时间时，CPU 是按照线程的优先级给予服务的。在C#应用程序中，用户可以设定5个不同的优先级，由高到低分别是Highest，AboveNormal，Normal，BelowNormal，Lowest，在创建线程时如果不指定优先级，那么系统默认为ThreadPriority.Normal。

     给一个线程指定优先级，我们可以使用如下代码：

//设定优先级为最低

myThread.Priority=ThreadPriority.Lowest;

      通过设定线程的优先级，我们可以安排一些相对重要的线程优先执行，例如对用户的响应等等。

**C#多线程学习(三) 生产者和消费者**

      前面说过，每个线程都有自己的资源，但是代码区是共享的，即每个线程都可以执行相同的函数。这可能带来的问题就是几个线程同时执行一个函数，导致数据的混乱，产生不可预料的结果，因此我们必须避免这种情况的发生。

      C#提供了一个关键字lock，它可以把一段代码定义为互斥段（critical section），互斥段在一个时刻内只允许一个线程进入执行，而其他线程必须等待。在C#中，关键字lock定义如下：

**lock(expression) statement\_block**

expression代表你希望跟踪的对象，通常是对象引用。

* 如果你想保护一个类的实例，一般地，你可以使用this；
* 如果你想保护一个静态变量（如互斥代码段在一个静态方法内部），一般使用类名就可以了。

而statement\_block就是互斥段的代码，这段代码在一个时刻内只可能被一个线程执行。

下面是一个使用lock关键字的典型例子，在注释里说明了lock关键字的用法和用途。

示例如下：

[复制代码](javascript:void(0);)

using System;

using System.Threading;

namespace ThreadSimple

{

internal class Account

{

int balance;

Random r = new Random();

internal Account(int initial)

{

balance = initial;

}

internal int Withdraw(int amount)

{

if (balance < 0)

{

//如果balance小于0则抛出异常

throw new Exception("Negative Balance");

}

//下面的代码保证在当前线程修改balance的值完成之前

//不会有其他线程也执行这段代码来修改balance的值

//因此，balance的值是不可能小于0 的

lock (this)

{

Console.WriteLine("Current Thread:"+Thread.CurrentThread.Name);

//如果没有lock关键字的保护，那么可能在执行完if的条件判断之后

//另外一个线程却执行了balance=balance-amount修改了balance的值

//而这个修改对这个线程是不可见的，所以可能导致这时if的条件已经不成立了

//但是，这个线程却继续执行balance=balance-amount，所以导致balance可能小于0

if (balance >= amount)

{

Thread.Sleep(5);

balance = balance - amount;

return amount;

}

else

{

return 0; // transaction rejected

}

}

}

internal void DoTransactions()

{

for (int i = 0; i < 100; i++)

Withdraw(r.Next(-50, 100));

}

}

internal class Test

{

static internal Thread[] threads = new Thread[10];

public static void Main()

{

Account acc = new Account (0);

for (int i = 0; i < 10; i++)

{

Thread t = new Thread(new ThreadStart(acc.DoTransactions));

threads[i] = t;

}

for (int i = 0; i < 10; i++)

threads[i].Name=i.ToString();

for (int i = 0; i < 10; i++)

threads[i].Start();

Console.ReadLine();

}

}

}

[复制代码](javascript:void(0);)

**Monitor 类锁定一个对象**

      当多线程公用一个对象时，也会出现和公用代码类似的问题，这种问题就不应该使用lock关键字了，这里需要用到System.Threading中的一个类Monitor，我们可以称之为监视器，Monitor提供了使线程共享资源的方案。

      Monitor类可以锁定一个对象，一个线程只有得到这把锁才可以对该对象进行操作。对象锁机制保证了在可能引起混乱的情况下一个时刻只有一个线程可以访问这个对象。 Monitor必须和一个具体的对象相关联，但是由于它是一个静态的类，所以不能使用它来定义对象，而且它的所有方法都是静态的，不能使用对象来引用。下面代码说明了使用Monitor锁定一个对象的情形：

......

Queue oQueue=new Queue();

......

Monitor.Enter(oQueue);

......//现在oQueue对象只能被当前线程操纵了

Monitor.Exit(oQueue);//释放锁

     如上所示，当一个线程调用Monitor.Enter()方法锁定一个对象时，这个对象就归它所有了，其它线程想要访问这个对象，只有等待它使用Monitor.Exit()方法释放锁。为了保证线程最终都能释放锁，你可以把Monitor.Exit()方法写在try-catch-finally结构中的finally代码块里。

对于任何一个被Monitor锁定的对象，内存中都保存着与它相关的一些信息:

* 其一是现在持有锁的线程的引用;
* 其二是一个预备队列，队列中保存了已经准备好获取锁的线程;
* 其三是一个等待队列，队列中保存着当前正在等待这个对象状态改变的队列的引用。

     当拥有对象锁的线程准备释放锁时，它使用Monitor.Pulse()方法通知等待队列中的第一个线程，于是该线程被转移到预备队列中，当对象锁被释放时，在预备队列中的线程可以立即获得对象锁。

     下面是一个展示如何使用lock关键字和Monitor类来实现线程的同步和通讯的例子，也是一个典型的生产者与消费者问题。

      这个例程中，生产者线程和消费者线程是交替进行的，生产者写入一个数，消费者立即读取并且显示（注释中介绍了该程序的精要所在）。

     用到的系统命名空间如下：

**using System;**

**using System.Threading;**

      首先，定义一个被操作的对象的类Cell，在这个类里，有两个方法：ReadFromCell()和WriteToCell。消费者线程将调用ReadFromCell()读取cellContents的内容并且显示出来，生产者进程将调用WriteToCell()方法向cellContents写入数据。

示例如下：

[复制代码](javascript:void(0);)

public class Cell

{

int cellContents; // Cell对象里边的内容

bool readerFlag = false; // 状态标志，为true时可以读取，为false则正在写入

public int ReadFromCell( )

{

lock(this) // Lock关键字保证了什么，请大家看前面对lock的介绍

{

if (!readerFlag)//如果现在不可读取

{

try

{

//等待WriteToCell方法中调用Monitor.Pulse()方法

Monitor.Wait(this);

}

catch (SynchronizationLockException e)

{

Console.WriteLine(e);

}

catch (ThreadInterruptedException e)

{

Console.WriteLine(e);

}

}

Console.WriteLine("Consume: {0}",cellContents);

readerFlag = false;

//重置readerFlag标志，表示消费行为已经完成

Monitor.Pulse(this);

//通知WriteToCell()方法（该方法在另外一个线程中执行，等待中）

}

return cellContents;

}

public void WriteToCell(int n)

{

lock(this)

{

if (readerFlag)

{

try

{

Monitor.Wait(this);

}

catch (SynchronizationLockException e)

{

//当同步方法（指Monitor类除Enter之外的方法）在非同步的代码区被调用

Console.WriteLine(e);

}

catch (ThreadInterruptedException e)

{

//当线程在等待状态的时候中止

Console.WriteLine(e);

}

}

cellContents = n;

Console.WriteLine("Produce: {0}",cellContents);

readerFlag = true;

Monitor.Pulse(this);

//通知另外一个线程中正在等待的ReadFromCell()方法

}

}

}

下面定义生产者类 CellProd 和消费者类 CellCons ，它们都只有一个方法ThreadRun()，以便在Main()函数中提供给线程的ThreadStart代理对象，作为线程的入口。

public class CellProd

{

　　 Cell cell; // 被操作的Cell对象

　　 int quantity = 1; // 生产者生产次数，初始化为1

　　 public CellProd(Cell box, int request)

　　 {

//构造函数

cell = box;

quantity = request;

　　 }

　　 public void ThreadRun( )

　　 {

for(int looper=1; looper<=quantity; looper++)

　　　　 cell.WriteToCell(looper); //生产者向操作对象写入信息

　　 }

}

public class CellCons

{

　　 Cell cell;

　　 int quantity = 1;

　　 public CellCons(Cell box, int request)

　　 {

//构造函数

cell = box;

quantity = request;

　　 }

　　 public void ThreadRun( )

　　 {

int valReturned;

for(int looper=1; looper<=quantity; looper++)

　　　　 valReturned=cell.ReadFromCell( );//消费者从操作对象中读取信息

　　 }

}

[复制代码](javascript:void(0);)

      然后在下面这个类MonitorSample的Main()函数中，我们要做的就是创建两个线程分别作为生产者和消费者，使用CellProd.ThreadRun()方法和CellCons.ThreadRun()方法对同一个Cell对象进行操作。

[复制代码](javascript:void(0);)

public class MonitorSample

{

　　 public static void Main(String[] args)

　　 {

int result = 0; //一个标志位，如果是0表示程序没有出错，如果是1表明有错误发生

Cell cell = new Cell( );

//下面使用cell初始化CellProd和CellCons两个类，生产和消费次数均为20次

CellProd prod = new CellProd(cell, 20);

CellCons cons = new CellCons(cell, 20);

Thread producer = new Thread(new ThreadStart(prod.ThreadRun));

Thread consumer = new Thread(new ThreadStart(cons.ThreadRun));

//生产者线程和消费者线程都已经被创建，但是没有开始执行

try

{

　　　　 producer.Start( );

　　　　 consumer.Start( );

　　　　 producer.Join( );

　　　　 consumer.Join( );

　　　　 Console.ReadLine();

}

catch (ThreadStateException e)

{

　　　　 //当线程因为所处状态的原因而不能执行被请求的操作

　　　　 Console.WriteLine(e);

　　　　 result = 1;

}

catch (ThreadInterruptedException e)

{

　　　　 //当线程在等待状态的时候中止

　　　　 Console.WriteLine(e);

　　　　 result = 1;

}

//尽管Main()函数没有返回值，但下面这条语句可以向父进程返回执行结果

Environment.ExitCode = result;

　　 }

}

[复制代码](javascript:void(0);)

      在上面的例程中，同步是通过等待Monitor.Pulse()来完成的。首先生产者生产了一个值，而同一时刻消费者处于等待状态，直到收到生产者的“脉冲(Pulse)”通知它生产已经完成，此后消费者进入消费状态，而生产者开始等待消费者完成操作后将调用Monitor.Pulese()发出的“脉冲”。

它的执行结果很简单：

Produce: 1

Consume: 1

Produce: 2

Consume: 2

Produce: 3

Consume: 3

...

...

Produce: 20

Consume: 20

     事实上，这个简单的例子已经帮助我们解决了多线程应用程序中可能出现的大问题，只要领悟了解决线程间冲突的基本方法，很容易把它应用到比较复杂的程序中去。

**C#多线程学习(四) 多线程的自动管理(线程池)**

在多线程的程序中，经常会出现两种情况：

* 一种情况： 应用程序中，线程把大部分的时间花费在等待状态，等待某个事件发生，然后才能给予响应

这一般使用ThreadPool（线程池）来解决；

* 另一种情况：线程平时都处于休眠状态，只是周期性地被唤醒

这一般使用Timer（定时器）来解决；

      ThreadPool类提供一个由系统维护的线程池（可以看作一个线程的容器），该容器需要 Windows 2000 以上系统支持，因为其中某些方法调用了只有高版本的Windows才有的API函数。

将线程安放在线程池里，需使用ThreadPool.QueueUserWorkItem()方法，该方法的原型如下：

//将一个线程放进线程池，该线程的Start()方法将调用WaitCallback代理对象代表的函数

public static bool QueueUserWorkItem(WaitCallback);

//重载的方法如下，参数object将传递给WaitCallback所代表的方法

public static bool QueueUserWorkItem(WaitCallback, object);

     ThreadPool类是一个静态类，你不能也不必要生成它的对象。而且一旦使用该方法在线程池中添加了一个项目，那么该项目将是无法取消的。

     在这里你无需自己建立线程，只需把你要做的工作写成函数，然后作为参数传递给ThreadPool.QueueUserWorkItem()方法就行了，传递的方法就是依靠WaitCallback代理对象，而线程的建立、管理、运行等工作都是由系统自动完成的，你无须考虑那些复杂的细节问题。

**ThreadPool 的用法：**

      首先程序创建了一个ManualResetEvent对象，该对象就像一个信号灯，可以利用它的信号来通知其它线程。

本例中，当线程池中所有线程工作都完成以后，ManualResetEvent对象将被设置为有信号，从而通知主线程继续运行。

**ManualResetEvent对象有几个重要的方法：**

初始化该对象时，用户可以指定其默认的状态（有信号/无信号）；

在初始化以后，该对象将保持原来的状态不变，直到它的Reset()或者Set()方法被调用：

**Reset()方法：**将其设置为无信号状态；

**Set()方法：**将其设置为有信号状态。

**WaitOne()方法：**使当前线程挂起，直到ManualResetEvent对象处于有信号状态，此时该线程将被激活。然后，程序将向线程池中添加工作项，这些以函数形式提供的工作项被系统用来初始化自动建立的线程。当所有的线程都运行完了以后，ManualResetEvent.Set()方法被调用，因为调用了ManualResetEvent.WaitOne()方法而处在等待状态的主线程将接收到这个信号，于是它接着往下执行，完成后边的工作。

**ThreadPool 的用法示例：**

[复制代码](javascript:void(0);)

using System;

using System.Collections;

using System.Threading;

namespace ThreadExample

{

//这是用来保存信息的数据结构，将作为参数被传递

public class SomeState

{

　　 public int Cookie;

　　 public SomeState(int iCookie)

　　 {

Cookie = iCookie;

　　 }

}

public class Alpha

{

　　public Hashtable HashCount;

　　public ManualResetEvent eventX;

　　public static int iCount = 0;

　　public static int iMaxCount = 0;

public Alpha(int MaxCount)

　　{

　 HashCount = new Hashtable(MaxCount);

　 iMaxCount = MaxCount;

　　}

　　//线程池里的线程将调用Beta()方法

　　public void Beta(Object state)

　　{

　　 //输出当前线程的hash编码值和Cookie的值

　 Console.WriteLine(" {0} {1} :", Thread.CurrentThread.GetHashCode(),((SomeState)state).Cookie);

Console.WriteLine("HashCount.Count=={0}, Thread.CurrentThread.GetHashCode()=={1}", HashCount.Count, Thread.CurrentThread.GetHashCode());

lock (HashCount)

{

　　　　 //如果当前的Hash表中没有当前线程的Hash值，则添加之

　　　　 if (!HashCount.ContainsKey(Thread.CurrentThread.GetHashCode()))

　　 　 HashCount.Add (Thread.CurrentThread.GetHashCode(), 0);

　 　　 HashCount[Thread.CurrentThread.GetHashCode()] =

((int)HashCount[Thread.CurrentThread.GetHashCode()])+1;

　 　}

int iX = 2000;

Thread.Sleep(iX);

//Interlocked.Increment()操作是一个原子操作，具体请看下面说明

Interlocked.Increment(ref iCount);

if (iCount == iMaxCount)

{

　 　 Console.WriteLine();

　　　　 Console.WriteLine("Setting eventX ");

　　　　 eventX.Set();

　 }

　　 }

}

public class SimplePool

{

public static int Main(string[] args)

{

Console.WriteLine("Thread Pool Sample:");

bool W2K = false;

int MaxCount = 10;//允许线程池中运行最多10个线程

//新建ManualResetEvent对象并且初始化为无信号状态

ManualResetEvent eventX = new ManualResetEvent(false);

Console.WriteLine("Queuing {0} items to Thread Pool", MaxCount);

Alpha oAlpha = new Alpha(MaxCount);

//创建工作项

//注意初始化oAlpha对象的eventX属性

oAlpha.eventX = eventX;

Console.WriteLine("Queue to Thread Pool 0");

try

{

//将工作项装入线程池

//这里要用到Windows 2000以上版本才有的API，所以可能出现NotSupportException异常

ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta), new SomeState(0));

W2K = true;

}

catch (NotSupportedException)

{

Console.WriteLine("These API's may fail when called on a non-Windows 2000 system.");

W2K = false;

}

if (W2K)//如果当前系统支持ThreadPool的方法.

{

for (int iItem=1;iItem < MaxCount;iItem++)

{

//插入队列元素

Console.WriteLine("Queue to Thread Pool {0}", iItem);

ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta), new SomeState(iItem));

}

Console.WriteLine("Waiting for Thread Pool to drain");

//等待事件的完成，即线程调用ManualResetEvent.Set()方法

eventX.WaitOne(Timeout.Infinite,true);

//WaitOne()方法使调用它的线程等待直到eventX.Set()方法被调用

Console.WriteLine("Thread Pool has been drained (Event fired)");

Console.WriteLine();

Console.WriteLine("Load across threads");

foreach(object o in oAlpha.HashCount.Keys)

Console.WriteLine("{0} {1}", o, oAlpha.HashCount[o]);

}

Console.ReadLine();

return 0;

}

}

}

}

[复制代码](javascript:void(0);)

**程序中应该引起注意的地方：**

      SomeState类是一个保存信息的数据结构，它在程序中作为参数被传递给每一个线程，因为你需要把一些有用的信息封装起来提供给线程，而这种方式是非常有效的。

      程序出现的InterLocked类也是专为多线程程序而存在的，它提供了一些有用的原子操作。

**原子操作：**就是在多线程程序中，如果这个线程调用这个操作修改一个变量，那么其他线程就不能修改这个变量了，这跟lock关键字在本质上是一样的。

[复制代码](javascript:void(0);)

Thread Pool Sample:

Queuing 10 items to Thread Pool

Queue to Thread Pool 0

Queue to Thread Pool 1

Queue to Thread Pool 2

Queue to Thread Pool 3

Queue to Thread Pool 4

Queue to Thread Pool 5

2 0 :

HashCount.Count==0, Thread.CurrentThread.GetHashCode()==2

Queue to Thread Pool 6

Queue to Thread Pool 7

Queue to Thread Pool 8

Queue to Thread Pool 9

Waiting for Thread Pool to drain

4 1 :

HashCount.Count==1, Thread.CurrentThread.GetHashCode()==4

6 2 :

HashCount.Count==1, Thread.CurrentThread.GetHashCode()==6

7 3 :

HashCount.Count==1, Thread.CurrentThread.GetHashCode()==7

2 4 :

HashCount.Count==1, Thread.CurrentThread.GetHashCode()==2

8 5 :

HashCount.Count==2, Thread.CurrentThread.GetHashCode()==8

9 6 :

HashCount.Count==2, Thread.CurrentThread.GetHashCode()==9

10 7 :

HashCount.Count==2, Thread.CurrentThread.GetHashCode()==10

11 8 :

HashCount.Count==2, Thread.CurrentThread.GetHashCode()==11

4 9 :

HashCount.Count==2, Thread.CurrentThread.GetHashCode()==4

Setting eventX

Thread Pool has been drained (Event fired)

Load across threads

11 1

10 1

9 1

8 1

7 1

6 1

4 2

2 2

[复制代码](javascript:void(0);)

      我们应该彻底地分析上面的程序，把握住线程池的本质，理解它存在的意义是什么，这样才能得心应手地使用它。

**C#多线程学习(五) 多线程的自动管理(定时器)**

**Timer类**：设置一个定时器，定时执行用户指定的函数。

定时器启动后，系统将自动建立一个新的线程，执行用户指定的函数。

初始化一个Timer对象：

Timer timer = new Timer(timerDelegate, s,1000, 1000);

// 第一个参数：指定了TimerCallback 委托，表示要执行的方法；

// 第二个参数：一个包含回调方法要使用的信息的对象，或者为空引用；

// 第三个参数：延迟时间——计时开始的时刻距现在的时间，单位是毫秒,指定为“0”表示立即启动计时器；

// 第四个参数：定时器的时间间隔——计时开始以后，每隔这么长的一段时间，TimerCallback所代表的方法将被调用一次，单位也是毫秒。指定 Timeout.Infinite 可以禁用定期终止。

Timer.Change()方法：修改定时器的设置。（这是一个参数类型重载的方法）

使用示例： timer.Change(1000,2000);

**Timer类的程序示例(来源：MSDN)：**

[复制代码](javascript:void(0);)

using System;

using System.Threading;

namespace ThreadExample

{

class TimerExampleState

{

　　 public int counter = 0;

　　 public Timer tmr;

}

class App

{

　　 public static void Main()

　　 {

　　 TimerExampleState s = new TimerExampleState();

　　 //创建代理对象TimerCallback，该代理将被定时调用

　　 TimerCallback timerDelegate = new TimerCallback(CheckStatus);

//创建一个时间间隔为1s的定时器

Timer timer = new Timer(timerDelegate, s,1000, 1000);

s.tmr = timer;

//主线程停下来等待Timer对象的终止

while(s.tmr != null)

Thread.Sleep(0);

Console.WriteLine("Timer example done.");

Console.ReadLine();

　　 }

　　 //下面是被定时调用的方法

　　 static void CheckStatus(Object state)

　　 {

TimerExampleState s =(TimerExampleState)state;

s.counter++;

Console.WriteLine("{0} Checking Status {1}.",DateTime.Now.TimeOfDay, s.counter);

if(s.counter == 5)

{

//使用Change方法改变了时间间隔

(s.tmr).Change(10000,2000);

Console.WriteLine("changed");

}

if(s.counter == 10)

{

Console.WriteLine("disposing of timer");

s.tmr.Dispose();

s.tmr = null;

}

　　 }

}

}

[复制代码](javascript:void(0);)

       程序首先创建了一个定时器，它将在创建1秒之后开始每隔1秒调用一次CheckStatus()方法，当调用5次以后，在CheckStatus()方法中修改了时间间隔为2秒，并且指定在10秒后重新开始。当计数达到10次，调用Timer.Dispose()方法删除了timer对象，主线程于是跳出循环，终止程序。

**C#多线程学习(六) 互斥对象**

      如何控制好多个线程相互之间的联系，不产生冲突和重复，这需要用到互斥对象，即：System.Threading 命名空间中的 Mutex 类。

      我们可以把Mutex看作一个出租车，乘客看作线程。乘客首先等车，然后上车，最后下车。当一个乘客在车上时，其他乘客就只有等他下车以后才可以上车。而线程与Mutex对象的关系也正是如此，线程使用Mutex.WaitOne()方法等待Mutex对象被释放，如果它等待的Mutex对象被释放了，它就自动拥有这个对象，直到它调用Mutex.ReleaseMutex()方法释放这个对象，而在此期间，其他想要获取这个Mutex对象的线程都只有等待。

      下面这个例子使用了Mutex对象来同步四个线程，主线程等待四个线程的结束，而这四个线程的运行又是与两个Mutex对象相关联的。

      其中还用到AutoResetEvent类的对象，可以把它理解为一个信号灯。这里用它的有信号状态来表示一个线程的结束。

// AutoResetEvent.Set()方法设置它为有信号状态

// AutoResetEvent.Reset()方法设置它为无信号状态

Mutex 类的程序示例：

[复制代码](javascript:void(0);)

using System;

using System.Threading;

namespace ThreadExample

{

public class MutexSample

{

　　 static Mutex gM1;

　　 static Mutex gM2;

　　 const int ITERS = 100;

　　 static AutoResetEvent Event1 = new AutoResetEvent(false);

　　 static AutoResetEvent Event2 = new AutoResetEvent(false);

　　 static AutoResetEvent Event3 = new AutoResetEvent(false);

　　 static AutoResetEvent Event4 = new AutoResetEvent(false);

　　 public static void Main(String[] args)

　　 {

Console.WriteLine("Mutex Sample ");

//创建一个Mutex对象，并且命名为MyMutex

gM1 = new Mutex(true,"MyMutex");

//创建一个未命名的Mutex 对象.

gM2 = new Mutex(true);

Console.WriteLine(" - Main Owns gM1 and gM2");

AutoResetEvent[] evs = new AutoResetEvent[4];

evs[0] = Event1; //为后面的线程t1,t2,t3,t4定义AutoResetEvent对象

evs[1] = Event2;

evs[2] = Event3;

evs[3] = Event4;

MutexSample tm = new MutexSample( );

Thread t1 = new Thread(new ThreadStart(tm.t1Start));

Thread t2 = new Thread(new ThreadStart(tm.t2Start));

Thread t3 = new Thread(new ThreadStart(tm.t3Start));

Thread t4 = new Thread(new ThreadStart(tm.t4Start));

t1.Start( );// 使用Mutex.WaitAll()方法等待一个Mutex数组中的对象全部被释放

t2.Start( );// 使用Mutex.WaitOne()方法等待gM1的释放

t3.Start( );// 使用Mutex.WaitAny()方法等待一个Mutex数组中任意一个对象被释放

t4.Start( );// 使用Mutex.WaitOne()方法等待gM2的释放

Thread.Sleep(2000);

Console.WriteLine(" - Main releases gM1");

gM1.ReleaseMutex( ); //线程t2,t3结束条件满足

Thread.Sleep(1000);

Console.WriteLine(" - Main releases gM2");

gM2.ReleaseMutex( ); //线程t1,t4结束条件满足

//等待所有四个线程结束

WaitHandle.WaitAll(evs);

Console.WriteLine(" Mutex Sample");

Console.ReadLine();

　　 }

　　 public void t1Start( )

　　 {

Console.WriteLine("t1Start started, Mutex.WaitAll(Mutex[])");

Mutex[] gMs = new Mutex[2];

gMs[0] = gM1;//创建一个Mutex数组作为Mutex.WaitAll()方法的参数

gMs[1] = gM2;

Mutex.WaitAll(gMs);//等待gM1和gM2都被释放

Thread.Sleep(2000);

Console.WriteLine("t1Start finished, Mutex.WaitAll(Mutex[]) satisfied");

Event1.Set( ); //线程结束，将Event1设置为有信号状态

　　 }

　　 public void t2Start( )

　　 {

Console.WriteLine("t2Start started, gM1.WaitOne( )");

gM1.WaitOne( );//等待gM1的释放

Console.WriteLine("t2Start finished, gM1.WaitOne( ) satisfied");

Event2.Set( );//线程结束，将Event2设置为有信号状态

　　 }

　　 public void t3Start( )

　　 {

Console.WriteLine("t3Start started, Mutex.WaitAny(Mutex[])");

Mutex[] gMs = new Mutex[2];

gMs[0] = gM1;//创建一个Mutex数组作为Mutex.WaitAny()方法的参数

gMs[1] = gM2;

Mutex.WaitAny(gMs);//等待数组中任意一个Mutex对象被释放

Console.WriteLine("t3Start finished, Mutex.WaitAny(Mutex[])");

Event3.Set( );//线程结束，将Event3设置为有信号状态

　　 }

　　 public void t4Start( )

　　 {

Console.WriteLine("t4Start started, gM2.WaitOne( )");

gM2.WaitOne( );//等待gM2被释放

Console.WriteLine("t4Start finished, gM2.WaitOne( )");

Event4.Set( );//线程结束，将Event4设置为有信号状态

　　 }

}

}

[复制代码](javascript:void(0);)

程序的输出结果：

[复制代码](javascript:void(0);)

Mutex Sample

- Main Owns gM1 and gM2

t1Start started, Mutex.WaitAll(Mutex[])

t2Start started, gM1.WaitOne( )

t3Start started, Mutex.WaitAny(Mutex[])

t4Start started, gM2.WaitOne( )

- Main releases gM1

t2Start finished, gM1.WaitOne( ) satisfied

t3Start finished, Mutex.WaitAny(Mutex[])

- Main releases gM2

t1Start finished, Mutex.WaitAll(Mutex[]) satisfied

t4Start finished, gM2.WaitOne( )

Mutex Sample

[复制代码](javascript:void(0);)

      从执行结果可以很清楚地看到，线程t2,t3的运行是以gM1的释放为条件的，而t4在gM2释放后开始执行，t1则在gM1和gM2都被释放了之后才执行。Main()函数最后，使用WaitHandle等待所有的AutoResetEvent对象的信号，这些对象的信号代表相应线程的结束。