.NET Framework定义了两种类型的线程：前台线程和后台线程，默认创建的为前台线程。前台线程与后台线程的唯一区别是：当进程的所有前台线程停止时，后台线程将自动终止。

Thread类：不能继承。

public Thread(ThreadStart start)

其中，start是线程的入口函数，必须具有void返回类型，并且不能带有任何参数。当线程的入口函数返回时，线程自动停止。

例：程序thread\_test1

// Copyright 2016.刘珅珅

// author：刘珅珅

// 多线程编程

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace thread\_test1

{

class MyThread

{

public int count;

public Thread thread;

public MyThread(string name)

{

count = 0;

thread = new Thread(this.RunTest);

thread.Name = name;

thread.Start();

}

// 线程入口函数

void RunTest()

{

Console.WriteLine(thread.Name + " starting.");

do {

Thread.Sleep(500);

Console.WriteLine("In " + thread.Name + ", Count is " + count);

++count;

} while(count < 10);

}

}

class ThreadTest

{

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

MyThread thread = new MyThread("Child #1");

do

{

Console.Write(".");

Thread.Sleep(100);

} while (thread.count != 10);

Console.WriteLine("Main thread ending.");

}

}

}

输出结果：

Main thread starting.

.Child #1 starting.

....In Child #1, Count is 0

.....In Child #1, Count is 1

.....In Child #1, Count is 2

.....In Child #1, Count is 3

.....In Child #1, Count is 4

.....In Child #1, Count is 5

.....In Child #1, Count is 6

.....In Child #1, Count is 7

.....In Child #1, Count is 8

.....In Child #1, Count is 9

Main thread ending.

Thread类确定线程结束：

IsAlive属性判断线程是否在运行

例：程序thread\_test2

// Copyright 2016.刘珅珅

// author：刘珅珅

// 确定线程停止

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace thread\_test2

{

class MyThread

{

public int count;

public Thread thread;

public MyThread(string name)

{

count = 0;

thread = new Thread(this.RunTest);

thread.Name = name;

thread.Start();

}

// 线程入口函数

void RunTest()

{

Console.WriteLine(thread.Name + " starting.");

do

{

Thread.Sleep(500);

Console.WriteLine("In " + thread.Name + ", Count is " + count);

++count;

} while (count < 10);

}

}

class ThreadTest

{

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

// 构造3个线程

MyThread thread1 = new MyThread("Child #1");

MyThread thread2 = new MyThread("Child #2");

MyThread thread3 = new MyThread("Child #3");

// IsAlive只读属性判断线程是否停止

do {

Console.Write(".");

Thread.Sleep(100);

} while (thread1.thread.IsAlive

&& thread2.thread.IsAlive

&& thread3.thread.IsAlive);

Console.WriteLine("Main thread ending.");

}

}

}

Join()方法判断线程是否停止：Join()会阻塞主调线程，直到指定的线程“连接”它，即指定的线程返回主调线程。

例：程序thread\_test3

// Copyright 2016.刘珅珅

// author：刘珅珅

// Join方法判断线程是否停止

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace thread\_test3

{

class MyThread

{

public int count;

public Thread thread;

public MyThread(string name)

{

count = 0;

thread = new Thread(this.RunTest);

thread.Name = name;

thread.Start();

}

// 线程入口函数

void RunTest()

{

Console.WriteLine(thread.Name + " starting.");

do

{

Thread.Sleep(500);

Console.WriteLine("In " + thread.Name + ", Count is " + count);

++count;

} while (count < 10);

}

}

class ThreadTest

{

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

MyThread thread1 = new MyThread("Child #1");

MyThread thread2 = new MyThread("Child #2");

MyThread thread3 = new MyThread("Child #3");

// Join()函数判断线程是否停止

thread1.thread.Join();

Console.WriteLine("Child #1 joined.");

thread2.thread.Join();

Console.WriteLine("Child #2 joined.");

thread3.thread.Join();

Console.WriteLine("Child #3 joined.");

Console.WriteLine("Main thread ending.");

}

}

}

为线程传递实参：

如果希望给线程传递实参，则在构造线程时，必须调用下面的构造方法

public Thread(ParameterizedThreadStart start)

其中start是线程入口方法，其委托类型为：

public delegate void ParameterizedThreadStart(object obj);

通过Thread类的Start()方法将实参传递给线程。

例：程序thread\_test4

// Copyright 2016.刘珅珅

// author：刘珅珅

// 为线程传递实参

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace thread\_test4

{

class MyThread

{

public int count;

public Thread thread;

public MyThread(string name, int num)

{

count = 0;

thread = new Thread(this.Run);

thread.Name = name;

// 传递实参给线程

thread.Start(num);

}

void Run(object num)

{

Console.WriteLine(thread.Name + " starting with count of " + num);

do {

Thread.Sleep(500);

Console.WriteLine("In " + thread.Name + ", Count is " + count);

++count;

} while (count < (int)num);

}

}

class ThreadTest

{

static void Main(string[] args)

{

MyThread thread1 = new MyThread("Child #1", 5);

MyThread thread2 = new MyThread("Child #2", 3);

do {

Thread.Sleep(100);

} while (thread1.thread.IsAlive | thread2.thread.IsAlive);

Console.WriteLine("Main thread ending.");

}

}

}

输出结果：

Child #1 starting with count of 5

Child #2 starting with count of 3

In Child #1, Count is 0

In Child #2, Count is 0

In Child #2, Count is 1

In Child #1, Count is 1

In Child #1, Count is 2

In Child #2, Count is 2

In Child #1, Count is 3

In Child #1, Count is 4

Main thread ending.

前台线程和后台线程：

默认创建的为前台线程，可以通过设置Thread类的IsBackground属性将前台线程改为后台线程。

线程优先级：

线程的优先级在某种程度上决定了线程占用CPU时间的多少。在给定的时间周期内，低优先级的线程占用的CPU时间往往比高优先级的少。

除了线程优先级外，其他因素也能影响到CPU的占用时间，例如，高优先级线程正在等待一些资源（如键盘输入），那么低优先级的线程可以获得比高优先级线程更多的占用时间。

C#中，Thread类的Priority属性设置线程的优先级：

public ThreadPriority Priority {get; set;}

其中，ThreadPriority为枚举类型。定义了下面5种优先级设置：

ThreadPriority.Highest

ThreadPriority.AboveNormal

ThreadPriority.Normal

ThreadPriority.BelowNormal

ThreadPriority.Lowest

线程的默认优先级设置是ThreadPriority.Normal

线程的同步：使用多线程时，有时需要协调两个或更多线程的行为，称为线程的同步，最常见的原因是多个线程访问一个共享资源，而该资源每次只能提供一个线程使用。

lock关键字的使用：

lock(lockObj) {

// 同步的语句

}

其中，lockObj是引用，指向要同步的对象。

锁定的对象是代表同步资源的对象。不可以公有访问锁定的对象。这是因为超出控制范围的另一部分代码可能会锁定该对象，并且永远不会释放它。

较好的方式是为锁定操作创建一个私有的对象。

lock关键字的作用：

1. 对于任意给定的对象，一旦获得锁，该锁定该对象，其他线程不能获得锁；
2. 同一对象上试图获得锁的其他线程将进入等待状态，直到代码解锁；
3. 当线程离开锁定的块时，对象解锁。

例：程序thread\_test6

// Copyright 2016.刘珅珅

// author：刘珅珅

// 多线程同步

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace thread\_test6

{

class SumArray

{

int sum;

// 私有对象用于锁定

object lock\_obj = new object();

public int SumIt(int[] nums)

{

// 锁定一个私有对象

lock(lock\_obj)

{

sum = 0;

for (int i = 0; i < nums.Length; ++i)

{

sum += nums[i];

Console.WriteLine("Running total for " + Thread.CurrentThread.Name

+ " is " + sum);

Thread.Sleep(10);

}

}

return sum;

}

}

class MyThread

{

public Thread thread;

int[] array;

int answer;

// SumArray的静态对象

// 所有的MyThread对象共享这个静态对象

static SumArray sa = new SumArray();

public MyThread(string name, int[] nums)

{

array = nums;

thread = new Thread(this.Run);

thread.Name = name;

thread.Start();

}

void Run()

{

Console.WriteLine(thread.Name + " starting.");

answer = sa.SumIt(array);

Console.WriteLine("Sum for " + thread.Name + " is " + answer);

Console.WriteLine(thread.Name + " terminating.");

}

}

class ThreadTest

{

static void Main(string[] args)

{

int[] array = { 1, 2, 3, 4, 5};

MyThread thread1 = new MyThread("Child #1", array);

MyThread thread2 = new MyThread("Child #2", array);

thread1.thread.Join();

thread2.thread.Join();

}

}

}

输出结果：

Child #1 starting.

Child #2 starting.

Running total for Child #1 is 1

Running total for Child #1 is 3

Running total for Child #1 is 6

Running total for Child #1 is 10

Running total for Child #1 is 15

Sum for Child #1 is 15

Child #1 terminating.

Running total for Child #2 is 1

Running total for Child #2 is 3

Running total for Child #2 is 6

Running total for Child #2 is 10

Running total for Child #2 is 15

Sum for Child #2 is 15

Child #2 terminating.

从输出结果可以看出，所有线程在计算数组求和是都能正确的计算出结果。在MyThread类中，所有对象都共享一个静态SumArray类对象，如果两个线程同时调用SumIt()方法，就会出现错误的计算结果，因此需要考虑同步。在SumIt()方法中，lock语句可避免不同线程同时使用此方法。SumIt()方法的内部代码被锁定，因为同一时间只能有一个线程使用它。

如果注释掉上述程序中SumIt()方法中的lock语句，将会得到下面的结果：

Child #1 starting.

Child #2 starting.

Running total for Child #2 is 1

Running total for Child #1 is 1

Running total for Child #2 is 5

Running total for Child #1 is 5

Running total for Child #2 is 8

Running total for Child #1 is 11

Running total for Child #1 is 15

Running total for Child #2 is 19

Running total for Child #2 is 24

Running total for Child #1 is 29

Sum for Child #2 is 29

Child #2 terminating.

Sum for Child #1 is 29

Child #1 terminating.

可以看出，在同一个时刻，有多个线程同时进行了数组的计算，运算结果都存储在sum中，结果不正确。

实现同步的另一种方式：我们或许希望同步访问并非自己创建的类中的方法，但该方法本身并不是同步的。

例：程序thread\_test7

// Copyright 2016.刘珅珅

// author：刘珅珅

// 实现同步的另一种方式

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace thread\_test7

{

class SumArray

{

int sum;

public int SumIt(int[] nums)

{

sum = 0;

for (int i = 0; i < nums.Length; ++i)

{

sum += nums[i];

Console.WriteLine("Running total for " + Thread.CurrentThread.Name

+ " is " + sum);

Thread.Sleep(10);

}

return sum;

}

}

class MyThread

{

public Thread thread;

int[] array;

int answer;

// SumArray的静态对象

// 所有的MyThread对象共享这个静态对象

static SumArray sa = new SumArray();

public MyThread(string name, int[] nums)

{

array = nums;

thread = new Thread(this.Run);

thread.Name = name;

thread.Start();

}

void Run()

{

Console.WriteLine(thread.Name + " starting.");

// 直接在MyThread类中锁定SumArray的SumIt()方法

// sa为私有变量，可以安全的锁定

lock (sa)

{

answer = sa.SumIt(array);

}

Console.WriteLine("Sum for " + thread.Name + " is " + answer);

Console.WriteLine(thread.Name + " terminating.");

}

}

class ThreadTest

{

static void Main(string[] args)

{

int[] array = { 1, 2, 3, 4, 5 };

MyThread thread1 = new MyThread("Child #1", array);

MyThread thread2 = new MyThread("Child #2", array);

thread1.thread.Join();

thread2.thread.Join();

}

}

}

C#线程间的通信

考虑这样一种情形：

线程T正在lock块内执行，此时需要访问暂时不可用的资源R。如果线程T等待资源R可用，那么可能进行某种形式的轮询，因为T将锁定lock块内的对象，阻止其他线程访问它。更好的方案是让T暂时放弃锁，允许其它线程访问。当资源R可用时，就通知T并恢复执行该线程。

C#利用Wait()、Pulse()和PluseAll()来实现线程间的通信，只能从锁定的代码块内部调用这3个方法。

在暂时中断线程运行时调用Wait()方法，这使得线程暂时进入休眠状态，并释放对象的锁，以允许其他线程获得该锁。然后，当其它线程使用完锁定的对象并调用Pulse()或PluseAll()方法时，唤醒休眠的线程。其中，Pulse()方法恢复等待该锁的线程队列中的第一个线程，而调用PulseAll()方法表示将锁释放给所有正在等待的线程。

public static bool Wait(object obj)

释放对象上的锁并阻塞该线程，直到其重新获得锁的使用权。

public static bool Wait(object, int millisecondsTimeout)

释放对象上的锁并阻塞该线程，直到重新获得锁的使用权或到达指定的毫秒数。

参考：<http://www.cnblogs.com/zhycyq/articles/2679017.html>

如果在同步代码块之外调用Wait()，Pulse()或PulseAll()，会抛出异常。

例：程序thread\_test8

// Copyright 2016.刘珅珅

// author：刘珅珅

// 线程间的通信

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*;

using *System*.*Threading*.*Tasks*;

namespace thread\_test8

{

class TickTock

{

object lock\_obj = new object();

public void Tick(bool running)

{

lock(lock\_obj)

{

if (!running)

{

// 唤醒等待线程队列中的第一个线程

// 函数返回会自动释放锁

*Monitor*.*Pulse*(lock\_obj);

return;

}

*Console*.*Write*("Tick");

*Monitor*.*Pulse*(lock\_obj);

// 释放锁，并阻塞Tick()方法，直到

// 重新锁定lock\_obj对象

*Monitor*.*Wait*(lock\_obj);

// Console.Write("end");

}

}

public void Tock(bool running)

{

lock (lock\_obj)

{

if (!running)

{

// 通知等待线程队列中的第一个线程

*Monitor*.*Pulse*(lock\_obj);

return;

}

*Console*.*WriteLine*("Tock");

*Monitor*.*Pulse*(lock\_obj);

*Monitor*.*Wait*(lock\_obj);

}

}

}

class MyThread

{

public *Thread* thread;

TickTock obj;

public MyThread(string name, TickTock tt)

{

thread = new *Thread*(this.Run);

obj = tt;

thread.*Name* = name;

thread.*Start*();

}

void Run()

{

if (thread.*Name* == "Tick")

{

for (int i = 0; i < 5; ++i)

obj.Tick(true);

obj.Tick(false);

}

else

{

for (int i = 0; i < 5; ++i)

obj.Tock(true);

obj.Tock(false);

}

}

}

class ThreadTest

{

static void Main(string[] args)

{

TickTock tt = new TickTock();

MyThread thread1 = new MyThread("Tick", tt);

MyThread thread2 = new MyThread("Tock", tt);

thread1.thread.*Join*();

thread2.thread.*Join*();

*Console*.*WriteLine*("Clock Stopped.");

}

}

}

输出结果为：

TickTock

TickTock

TickTock

TickTock

TickTock

Clock Stopped.

在上述例程中，Tick()方法根据running的值进行判断，如果running为false，则调用Pulse()方法唤醒正在等待lock\_obj的第一个线程；如果running为true，则打印Tick字符串，然后调用Pulse()方法唤醒其它线程，最后调用Wait()方法释放lock\_obj，并进入休眠状态，直到重新锁定lock\_obj对象。

死锁和竞争条件

死锁：一个线程等待另一个线程占用的资源可用，而另一个线程却等待第一个线程占用的资源可用，从而两个线程都只能挂起，互相等待对方释放资源，导致任何一个线程都不能执行。

互斥锁和信号量

互斥锁Mutex：

互斥锁是一个互斥的同步对象，同一时间有且仅有一个线程可以获取它。

Mutex可以跨应用程序域边界对资源进行独占访问，可以同步不同进程中的线程。

调用Mutex类的WaitOne()会处于等待状态直到可以获取其所调用的互斥锁。因此，该方法将阻塞主调线程的执行直到指定的互斥锁可用。

调用Mutex类的ReleaseMutex()释放互斥锁，调用WaitOne()方法的次数必须和ReleaseMutex()方法的次数相同。

例：程序thread\_test9

// Copyright 2016.刘珅珅

// author：刘珅珅

// 互斥锁

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*;

using *System*.*Threading*.*Tasks*;

namespace thread\_test9

{

class SharedRes

{

public static int count = 0;

public static *Mutex* mutex = new *Mutex*();

}

class IncThread

{

int num;

public *Thread* thread;

public IncThread(string name, int n)

{

thread = new *Thread*(this.Run);

num = n;

thread.*Name* = name;

thread.*Start*();

}

void Run()

{

*Console*.*WriteLine*(thread.*Name* + " is waiting for the mutex.");

// 请求Mutex

SharedRes.mutex.*WaitOne*();

*Console*.*WriteLine*(thread.*Name* + " acquires the mutex.");

do

{

*Thread*.*Sleep*(500);

SharedRes.count++;

*Console*.*WriteLine*("In " + thread.*Name*

+ ", SharedRes.count is " + SharedRes.count);

--num;

} while (num > 0);

*Console*.*WriteLine*(thread.*Name* + " releases the mutex.");

SharedRes.mutex.*ReleaseMutex*();

}

}

class DecThread

{

int num;

public *Thread* thread;

public DecThread(string name, int n)

{

thread = new *Thread*(this.Run);

num = n;

thread.*Name* = name;

thread.*Start*();

}

void Run()

{

*Console*.*WriteLine*(thread.*Name* + " is waiting for the mutex.");

// 请求Mutex

SharedRes.mutex.*WaitOne*();

*Console*.*WriteLine*(thread.*Name* + " acquires the mutex.");

do

{

*Thread*.*Sleep*(500);

SharedRes.count--;

*Console*.*WriteLine*("In " + thread.*Name*

+ ", SharedRes.count is " + SharedRes.count);

--num;

} while (num > 0);

*Console*.*WriteLine*(thread.*Name* + " releases the mutex.");

SharedRes.mutex.*ReleaseMutex*();

}

}

class ThreadTest

{

static void Main(string[] args)

{

IncThread thread1 = new IncThread("Increment Thread", 5);

// 使增加线程启动

*Thread*.*Sleep*(1);

DecThread thread2 = new DecThread("Decrement Thread", 5);

thread1.thread.*Join*();

thread2.thread.*Join*();

}

}

}

输出结果：

Increment Thread is waiting for the mutex.

Increment Thread acquires the mutex.

Decrement Thread is waiting for the mutex.

In Increment Thread, SharedRes.count is 1

In Increment Thread, SharedRes.count is 2

In Increment Thread, SharedRes.count is 3

In Increment Thread, SharedRes.count is 4

In Increment Thread, SharedRes.count is 5

Increment Thread releases the mutex.

Decrement Thread acquires the mutex.

In Decrement Thread, SharedRes.count is 4

In Decrement Thread, SharedRes.count is 3

In Decrement Thread, SharedRes.count is 2

In Decrement Thread, SharedRes.count is 1

In Decrement Thread, SharedRes.count is 0

Decrement Thread releases the mutex.

信号量：当共享资源由“组”或“池”构成时，特别适合信号量。信号量允许多个线程同时访问一个共享资源。

信号量通过使用一个计数器来控制对共享资源的访问。如果计数器大于0，那么就允许访问。如果计数器等于0，就拒绝访问。为了访问某个资源，线程必须从信号量获取一个许可证。

通常，如果信号量的计数器大于0，则线程获取一个许可证并将信号量的计数减1.否则，线程将阻塞直到得到一个许可证。当线程不再需要共享资源时，将释放许可证并将信号量的计数加1.

如果创建一个仅允许一个线程访问共享资源的信号量，信号量就和互斥锁的用法一模一样。

例：程序thread\_test10

// Copyright 2016.刘珅珅

// author：刘珅珅

// 信号量

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*;

using *System*.*Threading*.*Tasks*;

namespace thread\_test10

{

class MyThread

{

public *Thread* thread;

// 信号量许可证的初始值为2，最大值也为2

// 最多允许两个线程同时访问它

static *Semaphore* sem = new *Semaphore*(2, 2);

public MyThread(string name)

{

thread = new *Thread*(this.Run);

thread.*Name* = name;

thread.*Start*();

}

void Run()

{

*Console*.*WriteLine*(thread.*Name* + " is waiting for a permit");

// 等待获取许可证

sem.*WaitOne*();

*Console*.*WriteLine*(thread.*Name* + " acquires a permit.");

for (char ch = 'A'; ch < 'D'; ++ch)

{

*Console*.*WriteLine*(thread.*Name* + " : " + ch + " ");

*Thread*.*Sleep*(500);

}

*Console*.*WriteLine*(thread.*Name* + " release a permit.");

// 释放许可证

sem.*Release*();

}

}

class ThreadTest

{

static void Main(string[] args)

{

MyThread thread1 = new MyThread("Thread #1");

MyThread thread2 = new MyThread("Thread #2");

MyThread thread3 = new MyThread("Thread #3");

thread1.thread.*Join*();

thread1.thread.*Join*();

thread3.thread.*Join*();

}

}

}

事件同步：当一个线程等待另一个线程中的某些事件发生时，就会用到事件同步。

同步事件有两种类型：手动重置和自动重置，分别由ManualResetEvent和AutoResetEvent实现。二者的区别在于事件的重置方式：对于ManualResetEvent，事件将保持已发出信号状态直到调用Reset()方法。而AutoResetEvent类，只要等待该事件的线程获得了事件通知并恢复执行，事件就自动转换为未发出信号的状态，不需要调用Reset()方法。

例：程序thread\_test11

// Copyright 2016.刘珅珅

// author：刘珅珅

// 事件同步

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*;

using *System*.*Threading*.*Tasks*;

namespace thread\_test11

{

class MyThread

{

public *Thread* thread;

*ManualResetEvent* mre;

public MyThread(string name, *ManualResetEvent* evt)

{

thread = new *Thread*(this.Run);

thread.*Name* = name;

mre = evt;

thread.*Start*();

}

void Run()

{

*Console*.*WriteLine*("Inside thread " + thread.*Name*);

for (int i = 0; i < 5; ++i)

{

*Console*.*WriteLine*(thread.*Name*);

*Thread*.*Sleep*(500);

}

*Console*.*WriteLine*(thread.*Name* + " Done!");

// 发送事件信号：将事件置成已发出信号状态

mre.*Set*();

}

}

class ThreadTest

{

static void Main(string[] args)

{

// false：初始状态下，事件未发出信号

// true：初始状态下，事件已发出信号

*ManualResetEvent* evt\_obj = new *ManualResetEvent*(false);

MyThread thread1 = new MyThread("Event Thread 1", evt\_obj);

*Console*.*WriteLine*("Main thread waiting for event.");

// 等待事件发出信号

evt\_obj.*WaitOne*();

*Console*.*WriteLine*("Main thread received first event.");

// 复位事件信号

evt\_obj.*Reset*();

}

}

}

输出结果：

Inside thread Event Thread 1

Event Thread 1

Main thread waiting for event.

Event Thread 1

Event Thread 1

Event Thread 1

Event Thread 1

Event Thread 1 Done!

Main thread received first event.

Interlocked类：可以更改共享变量的值，Interlocked的方法可以确保它们的操作以一种独立的、不被中断的方式进行，不需要其它的同步机制。

例：程序thread\_test12

// Copyright 2016.刘珅珅

// author：刘珅珅

// Interlocked同步

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*;

using *System*.*Threading*.*Tasks*;

namespace thread\_test12

{

class SharedRes

{

public static int count = 0;

}

class IncThread

{

public *Thread* thread;

public IncThread(string name)

{

thread = new *Thread*(this.Run);

thread.*Name* = name;

thread.*Start*();

}

void Run()

{

for (int i = 0; i < 5; ++i)

{

*Interlocked*.*Increment*(ref SharedRes.count);

*Console*.*WriteLine*(thread.*Name* + " count is " + SharedRes.count);

}

}

}

class DecThread

{

public *Thread* thread;

public DecThread(string name)

{

thread = new *Thread*(this.Run);

thread.*Name* = name;

thread.*Start*();

}

void Run()

{

for (int i = 0; i < 5; ++i)

{

*Interlocked*.*Decrement*(ref SharedRes.count);

*Console*.*WriteLine*(thread.*Name* + " count is " + SharedRes.count);

}

}

}

class ThreadTest

{

static void Main(string[] args)

{

IncThread thread1 = new IncThread("Increment Thread");

DecThread thread2 = new DecThread("Decrement Thread");

thread1.thread.*Join*();

thread2.thread.*Join*();

}

}

}