.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。