线程，任务和同步

siki 微信：devsiki QQ:804632564

线程

对于所有需要等待的操作，例如移动文件，数据库和网络访问都需要一定的时间，此时就可以启动一个新的线程，同时完成其他任务。一个进程的多个线程可以同时运行在不同的CPU上或多核CPU的不同内核上。

线程是程序中独立的指令流。在VS编辑器中输入代码的时候，系统会分析代码，用下划线标注遗漏的分号和其他语法错误，这就是用一个后台线程完成。Word文档需要一个线程等待用户输入，另一个线程进行后台搜索，第三个线程将写入的数据存储在临时文件中。运行在服务器上的应用程序中等待客户请求的线程成为侦听器线程。

进程包含资源，如Window句柄，文件系统句柄或其他内核对象。每个进程都分配的虚拟内存。一个进程至少包含一个线程。

一个应用程序启动，一般会启动一个进程，然后进程启动多个线程。

进程和线程的一个简单解释

1，计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。

2，如果工厂的电力有限一次只能供给一个车间使用。也就是说一个车间开工的时候，其他车间就必须停工。背后的含义就是。单个CPU一次只能运行一个任务。（多核CPU可以运行多个任务）

3，进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。

4，一个车间里，可以有很多工人，他们协同完成一个任务。

5，线程就好比车间里的工人。一个进程可以包括多个线程。

6，车间的控件是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享空间。

7，进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。

8，一个防止他人进入的简单方法，就是门口加一把锁（厕所）。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。

9，还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。

10，这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做"信号量"（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex是semaphore的一种特殊情况（n=1时）。也就是说，完全可以用后者替代前者。但是，因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。

11，操作系统的设计，因此可以归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

资料来源-阮一峰

异步委托

创建线程的一种简单方式是定义一个委托,并异步调用它。 委托是方法的类型安全的引用。Delegate类 还支持异步地调用方法。在后台,Delegate类会创建一个执行任务的线程。

接下来定义一个方法，使用委托异步调用（开启一个线程去执行这个方法）

```
static int TakesAWhile(int data,int ms){
        Console.WriteLine("TakesAWhile started!");
        Thread.Sleep(ms);//程序运行到这里的时候会暂停ms毫秒,然后继续运行下一语句
        Console.WriteLine("TakesAWhile completed");
        return ++data;
}
public delegate int TakesAWhileDelegate(int data,int ms);// 声明委托
static void Main(){
        TakesAWhileDelegate d1 = TakesAWhile;
        IAsyncResult ar = d1.BeginInvoke(1,3000,null,null);
        while(ar.IsCompleted ==false ){
                Console.Write(".");
                Thread.Sleep(50);
        }
        int result = d1.EndInvoke(ar);
        Console.WriteLine("Res:"+result);
}
```

等待句柄IAsyncResult.AsyncWaitHanlde

当我们通过BeginInvoke开启一个异步委托的时候，返回的结果是IAsyncResult，我们可以通过它的AsyncWaitHandle属性访问等待句柄。这个属性返回一个WaitHandler类型的对象，它中的WaitOne（）方法可以等待委托线程完成其任务，WaitOne方法可以设置一个超时时间作为参数（要等待的最长时间），如果发生超时就返回false。

```
static void Main(){
TakesAWhileDelegate d1 = TakesAWhile;
IAsyncResult ar = d1.BeginInvoke(1,3000,null,null);
while(true){
                Console.Write(".");
                if(ar.AsyncWaitHanle.WaitOne(50,false)){
                        Console.WriteLine("Can get result now");
                        break;
                }

}
int result = d1.EndInvoke(ar);
Console.WriteLine("Res:"+result);
}
```

异步回调-回调方法

等待委托的结果的第3种方式是使用异步回调。在BeginInvoke的第三个参数中，可以传递一个满足AsyncCallback委托的方法，AsyncCallback委托定义了一个IAsyncResult类型的参数其返回类型是void。对于最后一个参数，可以传递任意对象，以便从回调方法中访问它。（我们可以设置为委托实例，这样就可以在回调方法中获取委托方法的结果）

```
static void Main(){
TakesAWhileDelegate d1 = TakesAWhile;
d1.BeginInvoke(1,3000,TakesAWhileCompleted,d1);
while(true){
                Console.Write(".");
                Thread.Sleep(50);
}
}
static void TakesAWhileCompleted(IAsyncResult ar){//回调方法是从委托线程中调用的，并不是从主线程调用的，可以认为是委托线程最后要执行的程序
if(ar==null) throw new ArgumentNullException("ar");
TakesAWhileDelegate d1 = ar.AsyncState as TakesAWhileDelegate;
int result = d1.EndInvoke(ar);
Console.Write("Res:"+result);
}
```

异步回调-Lambda表达式

等待委托的结果的第3种方式是使用异步回调。在BeginInvoke的第三个参数中，可以传递一个满足AsyncCallback委托的方法，AsyncCallback委托定义了一个IAsyncResult类型的参数其返回类型是void。对于最后一个参数，可以传递任意对象，以便从回调方法中访问它。（我们可以设置为委托实例，这样就可以在回调方法中获取委托方法的结果）

```
static void Main(){
TakesAWhileDelegate d1 = TakesAWhile;
d1.BeginInvoke(1,3000,ar=>{
                                int result = d1.EndInvoke(ar);
                                Console.WriteLine("Res:"+result);
                                },null);
while(true){
                Console.Write(".");
                Thread.Sleep(50);
}
}
```

Thread类

使用Thread类可以创建和控制线程。Thread构造函数的参数是一个无参无返回值的委托类型。

```
static void Main(){
var t1 = new Thread(ThreadMain);
t1.Start();
Console.WriteLine("This is the main thread.");
}
static void ThreadMain(){
Console.WriteLine("Running in a thread.");
}
```

在这里哪个先输出是无法保证了线程的执行有操作系统决定，只能知道Main线程和分支线程是同步执行的。在这里给Thread传递一个方法，调用Thread的Start方法，就会开启一个线程去执行，传递的方法。

Thread类-Lambda表达式

上面直接给Thread传递了一个方法，其实也可以传递一个Lambda表达式。(委托参数的地方都可以使用Lambda表达式)

```
static void Main(){
var t1 = new Thread( ()=>Console.WriteLine("Running in a thread, id : "+Thread.CurrentThread.ManagedThreadId) );
t1.Start();
Console.WriteLine("This is the main Thread . ID : "+Thread.CurrentThread.ManagedThreadId)
}
```

给线程传递数据-通过委托

给线程传递一些数据可以采用两种方式，一种方式是使用带ParameterizedThreadStart委托参数的Thread构造函数，一种方式是创建一个自定义的类，把线程的方法定义为实例方法，这样就可以初始化实例的数据，之后启动线程。

```
public struct Data{//声明一个结构体用来传递数据
        public string Message;
}
static void ThreadMainWithParameters(Object o ){
        Data d=(Data)o;
        Console.WriteLine("Running in a thread , received :"+d.Message);
}
static void Main(){
        var d = new Data{Message = "Info"};
        var t2 = new Thread(ThreadMainWithParameters);
        t2.Start(d);
}
```

给线程传递数据-自定义类

```
public class MyThread{
private string data;
public MyThread(string data){
          this.data = data;
}
public void ThreadMain(){
          Console.WriteLine("Running in a thread , data : "+data);
}
}
var obj = new MyThread("info");
var t3 = new Thread(obj.ThreadMain);
t3.Start();
```

后台线程和前台线程

只有一个前台线程在运行，应用程序的进程就在运行，如果多个前台线程在运行，但是Main方法结束了，应用程序的进程仍然是运行的，直到所有的前台线程完成其任务为止。

在默认情况下，用Thread类创建的线程是前台线程。线程池中的线程总是后台线程。

在用Thread类创建线程的时候，可以设置IsBackground属性，表示它是一个前台线程还是一个后台线程。

看下面例子中前台线程和后台线程的区别：

```
class Program{
static void Main(){
                var t1 = new Thread(ThreadMain){IsBackground=false};
                t1.Start();
                Console.WriteLine("Main thread ending now.");
}
static void ThreadMain(){
                Console.WriteLine("Thread +"+Thread.CurrentThread.Name+" started");
                Thread.Sleep(3000);
                Console.WriteLine("Thread +"+Thread.CurrentThread.Name+" started");
}
}
```

后台线程用的地方：如果关闭Word应用程序，拼写检查器继续运行就没有意义了，在关闭应用程序的时候，拼写检查线程就可以关闭。

当所有的前台线程运行完毕，如果还有后台线程运行的话，所有的后台线程会被终止掉。

线程的优先级

线程有操作系统调度，一个CPU同一时间只能做一件事情（运行一个线程中的计算任务），当有很多线程需要CPU去执行的时候，线程调度器会根据线程的优先级去判断先去执行哪一个线程，如果优先级相同的话，就使用一个循环调度规则，逐个执行每个线程。

在Thead类中，可以设置Priority属性，以影响线程的基本优先级 ,Priority属性是一个ThreadPriority枚举定义的一个值。定义的级别有Highest ,AboveNormal,BelowNormal 和 Lowest。

控制线程

1,获取线程的状态（Running还是Unstarted,,,），当我们通过调用Thread对象的Start方法，可以创建线程，但是调用了Start方法之后，新线程不是马上进入Running状态，而是出于Unstarted状态，只有当操作系统的线程调度器选择了要运行的线程，这个线程的状态才会修改为Running状态。我们使用Thread.Sleep()方法可以让当前线程休眠进入WaitSleepJoin状态。

2,使用Thread对象的Abort()方法可以停止线程。调用这个方法，会在终止要终止的线程中抛出一个ThreadAbortException类型的异常，我们可以try catch这个异常，然后在线程结束前做一些清理的工作。

3,如果需要等待线程的结束，可以调用Thread对象的Join方法，表示把Thread加入进来，停止当前线程，并把它设置为WaitSleepJoin状态，直到加入的线程完成为止。

线程池

创建线程需要时间。 如果有不同的小任务要完成,就可以事先创建许多线程 ， 在应完成这些任务时发出请求。 这个线程数最好在需要更多的线程时增加,在需要释放资源时减少。

不需要 自己创建线程池，系统已经有一个ThreadPool类管理线程。 这个类会在需要时增减池中线程的线程数,直到达到最大的线程数。 池中的最大线程数是可配置的。 在双核 CPU中 ,默认设置为1023个工作线程和 1000个 I/o线程。也可以指定在创建线程池时应立即启动的最小线程数,以及线程池中可用的最大线程数。 如果有更多的作业要处理,线程池中线程的个数也到了极限,最新的作业就要排队,且必须等待线程完成其任务。

线程池示例

```
static void Main(){
int nWorkerThreads;
int nCompletionPortThreads;
ThreadPool.GetMaxThreads(out nWorkerThreads,out nCompletionPortThreads);
Console.WriteLine("Max worker threads : " +nWorkerThreads+" I/O completion threads :"+nCompletionPortThreads );
for(int i=0;i<5;i++){
                ThreadPool.QueueUserWorkItem(JobForAThread);
}
Thread.Sleep(3000);
}
static void JobForAThread(object state){
for(int i=0;i<3;i++){
                Console.WriteLine("Loop "+i+" ,running in pooled thread "+Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(50);
}
}
```

示例应用程序首先要读取工作线程和 I/o线程的最大线程数,把这些信息写入控制台中。接着在 for循 环中,调 用ThreadPool.QueueUserWorkItem方法,传递一个WaitCallBack类型的委托 ,把 JobForAThread方 法赋予线程池中的线程。 线程池收到这个请求后,就会从池中选择一个线程 来调用该方法。 如果线程池还没有运行,就会创建一个线程池,并启动第一个线程。 如果线程池已经在运行,且有一个空闲线程来完成该任务,就把该作业传递给这个线程。

使用线程池需要注意的事项:

线程池中的所有线程都是后台线程 。 如果进程的所有前台线程都结束了,所有的后台线程就会停止。 不能把入池的线程改为前台线程 。

不能给入池的线程设置优先级或名称。

入池的线程只能用于时间较短的任务。 如果线程要一直运行(如 Word的拼写检查器线程),就应使用Thread类创建一个线程。

任务

在.NET4 新的命名空间System.Threading.Tasks包含了类抽象出了线程功能，在后台使用的ThreadPool进行管理的。任务表示应完成某个单元的工作。这个工作可以在单独的线程中运行，也可以以同步方式启动一个任务。

任务也是异步编程中的一种实现方式。

启动任务

启动任务的三种方式：

```
TaskFactory tf = new TaskFactory();
Task t1 = tf.StartNew(TaskMethod);

Task t2 = TaskFactory.StartNew(TaskMethod);

Task t3 = new Task(TaskMethod);
t3.Start();
```

我们创建任务的时候有一个枚举类型的选项TaskCreationOptions

连续任务

如果一个任务t1的执行是依赖于另一个任务t2的，那么就需要在这个任务t2执行完毕后才开始执行t1。

这个时候我们可以使用连续任务。

```
static void DoFirst(){
Console.WriteLine("do  in task : "+Task.CurrentId);
Thread.Sleep(3000);
}
static void DoSecond(Task t){
Console.WriteLine("task "+t.Id+" finished.");
Console.WriteLine("this task id is "+Task.CurrentId);
Thread.Sleep(3000);
}
```

```
Task t1 = new Task(DoFirst);
Task t2 = t1.ContinueWith(DoSecond);
Task t3 = t1.ContinueWith(DoSecond);
Task t4 = t2.ContinueWith(DoSecond);


Task t5 = t1.ContinueWith(DoError,TaskContinuationOptions.OnlyOnFaulted);
```

任务层次结构

我们在一个任务中启动一个新的任务，相当于新的任务是当前任务的子任务，两个任务异步执行，如果父任务执行完了但是子任务没有执行完，它的状态会设置为WaitingForChildrenToComplete，只有子任务也执行完了，父任务的状态就变成RunToCompletion

```
static void Main(){
var parent = new Task(ParentTask);
parent.Start();
Thread.Sleep(2000);
Console.WriteLine(parent.Status);
Thread.Sleep(4000);
Console.WriteLine(parent.Status);
Console.ReadKey();
}
static void ParentTask(){
Console.WriteLine("task id "+Task.CurrentId);
var child = new Task(ChildTask);
child.Start();
Thread.Sleep(1000);
Console.WriteLine("parent started child , parent end");
}
static void ChildTask(){
Console.WriteLine("child");
Thread.Sleep(5000);
Console.WriteLine("child finished ");
```

线程问题-争用条件

```
public class StateObject{
private int state = 5;
public void ChangeState(int loop){
              if(state==5){
                            state++;//6
                            Console.WriteLine("State==5:"+state==5+"  Loop:"+loop);//false
              }
              state = 5;
}
}

static void RaceCondition(object o ){
StateObject state = o as StateObject;
int i = 0;
while(true){
              state.ChangeState(i++);
}
}
static void Main(){
var state = new StateObject();
for(int i=0;i<20;i++){
              new Task(RaceCondition,state).Start();
}
Thread.Sleep(10000);
```

使用lock（锁）解决争用条件的问题

```
static void RaceCondition(object o ){
StateObject state = o as StateObject;
int i = 0;
while(true){
                lock(state){
                                state.ChangeState(i++);
                }

}
}
```

另外一种方式是锁定StateObject中的state字段，但是我们的lock语句只能锁定个引用类型。因此可以定义一个object类型的变量sync，将它用于lock语句，每次修改state的值的时候，都使用这个一个sync的同步对象。就不会出现争用条件的问题了。下面是改进后的ChangeState方法

```
private object sync = new object();
public void ChangeState(int loop){
lock(sync){
                if(state==5){
                                state++;
                                Console.WriteLine("State==5:"+state==5+"  Loop:"+loop);
                }
                state = 5;
}
}
```

线程问题-死锁

```
public class SampleThread{
private StateObject s1;
private StateObject s2;
public SampleThread(StateObject s1,StateObject s2){
                this.s1= s1;
                this.s2 = s2;
}
public void Deadlock1(){
                int i =0;
                while(true){
                                lock(s1){
                                                lock(s2){
                                                                s1.ChangeState(i);
                                                                s2.ChangeState(i);
                                                                i++;
                                                                Console.WriteLine("Running i : "+i);
                                                }
                                }
                }
}
public void Deadlock2(){
                int i =0;
                while(true){
                                lock(s2){
                                                lock(s1){
                                                                s1.ChangeState(i);
                                                                s2.ChangeState(i);
                                                                    i + + ;
```

如何解决死锁

在编程的开始设计阶段，设计锁定顺序

关注公众号

发布最新的视频，文章和教学资源

THANK

@siki 微信：devsiki QQ:804632564