C#中异步与多线程的运用研究

姜礼盟　彭向梅
四川大学锦城学院　计算机与软件学院　四川　成都　611730

【摘要】随着C#应用程序中一些复杂和耗时操作的增多,多线程并行操作成为了解决难题的一个途径,同时创建线程的一种简单方式就是定义一个委托,并通过异步来调用它。通过多线程的异步委托,使主线程在执行的同时,能够执行子程序并得到子线程的运行结果。本文要通过分析介绍多线程异步委托的调用方法来解释C#多线程异步委托的并行操作。

【关键词】C#;多线程;委托;异步委托;

1　引　言

C#程序中,在主线程执行的时候,如果主线程需要调用一个其他方法,则需要阻塞主线程,并等待调用方法执行完成后主线程才能继续执行。为了使主线程在执行的同时,能够执行子程序并得到子线程的运行结果,我们采用了多线程的异步委托方法来实现这一情况。本文主要通过分析介绍多线程异步委托的调用方法来解释C#多线程异步委托的并行操作。

2　多线程异步委托基本概念

2.1　委托

C#中的委托类似于C或C++中函数的指针,它是用户自定义的类,定义了方法的类型。委托中存储的是一系列具有相同参数和返回类型方法的地址列表,调用委托时,此委托列表的所有方法都将被执行。相当于委托即是一种存有对某个方法的引用的一种引用类变量,能在程序运行的时候调用这一种方法。

2.2　多线程

线程是程序中独立的指令流,当程序需要同时完成多个任务的时候,就需要进入其他线程。使用线程可以将代码同其他代码进行隔离,提高程序的可靠性,也可以简化代码,并使用线程来实现并发执行。设置不同的线程执行路径往往能够解决一些涉及复杂和耗时操作的应用程序,为了同时执行多个任务,多线程并行操作就起到了良好的作用。

2.3　异步并行

在主线程执行的时候,打开一个子线程,主线程不会像同步执行的那样等待子线程的结果返回后在执行,主线程会继续执行,当主线程需要子线程的运行结果时,主线程直接调用子线程的运行结果,如果子线程的运行结果还没有出来,那么主线程等待,直到子线程执行结束,主线程拿到子线程的运行结果,主线程在继续。如下图1所示。

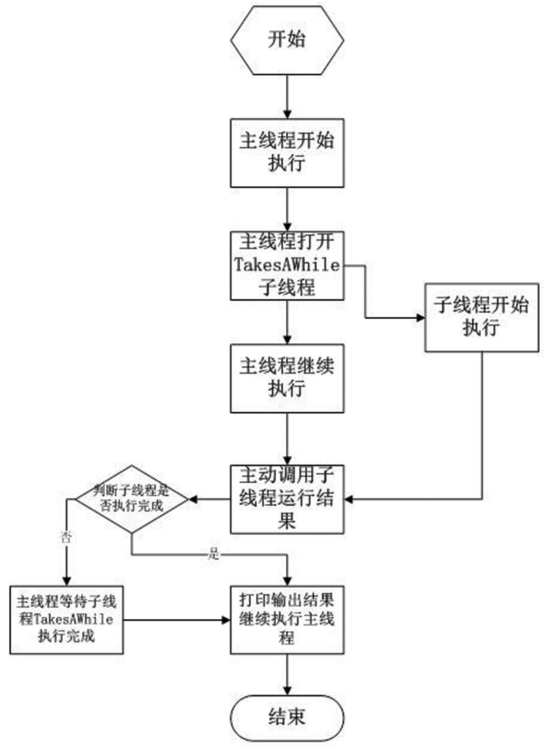

图1　线程的异步执行图

3　异步委托调用的方法

3.1　程序执行延迟

通过利用Thread.Sleep()方法可以使程序延迟一定的毫秒数时间,在此之前我们需要引入using System.Threading命名空间。如果要从委托中调

用这个方法,就必须定义一个有相同参数和返回类型的委托。

示例

```
static int TakesAWhile(int data,int ms)
{
Console. WriteLine("TakesAWhile started");
Thread. Sleep(ms);
Console. Writeline ("TakesAWhile completed.");
return++data;
}
public delegate int TakeAWhileDelegate (int data,int ms);
```

3.2 Invoke 方式

要想检查委托是否完成了任务,需要使用委托的 BeginInvoke 方法,该方法中可以调用委托类型定义的输入参数。该方法有 AsyncCallback 和 object 类型的两个参数,并且该方法的返回类型为 IAsyncResult,从 IAsyncResult 中可以获得委托的信息,并通过其中的 IsCompleted 属性验证委托是否完成了任务。最后用 EndInvoke 方法接受结果,它会一直等待,直到委托完成任务为止。

示例

```
static void Main()
{
TakeAWhileDelegate d = TakesAWhile;
IAsyncResult d1 = d. BeginInvoke(3000,null,null);
while(! d1. IsCompleted)
{
Console. Write(".");
Thread. Sleep(50);
}
int result = d. EndInvoke(d1);
Console. Writeline("result:{0}",result);
}
```

运行应用程序时,可以看到主线程和委托线程同时运行,在委托线程执行完毕之后,主线程就停止循环。

3.3 等待句柄(AsyncWaitHandle)

AsyncWaitHandle 属性可以访问等待句柄,返回一个 WaitHandle 类型的对象,同时也可以等待委托线程完成任务。方法 WaitOne 将一个超时时间作为可选的第一个参数,定义需要等待的最大时间。如果发生超时,该方法会返回 false。依然用 BeginInvoke 方法开始异步调用,用 EndInvoke 方法接受结果。

示例

```
static void Main()
{
TakeAWhileDelegate d1 = TakesAWhile;
IAsyncResult ar = d1. BeginInvoke (3000, null,null);
while(true)
{
Console. Write(".");
if(ar. AsyncWaitHandle. WaitOne(50))
{
Console. WriteLine ("Can get the result now");
break;
}
}
int result = d1. EndInvoke(ar);
Console. WriteLine("result:{0}",result);
}
```

3.4 异步回调

BeginInvoke 方法的第三个参数,一个 AsyncCallback 委托类型的方法。AsyncCallback 委托定义了一个 IAsyncResult 类型的参数,返回类型为 void。BeginInvoke 方法的第四个参数,可以传递任意对象,以便从回调方法中访问它。一般传递委托实例,提供回调函数使用它来获得异步方法的结果。使用回调方法,不需要在主线程中等待结果。在委托线程的任务未完成之前,不能停止主线程,除非主线程结束时停止的委托线程没有问题。

示例 1:利用委托创建回调

```
static void Main()
{
TakeAWhileDelegate d1 = TakesAWhile;
d1. BeginInvoke(1,300,TakesAWhileCompleted,d1);
Console. ReadKey();
static void TakesAWhileCompleted(IAsyncResult ar)
```

```
{
if(ar == null)throw new ArguementException("ar is null");
TakeAWhileDelegate d1 = ar.AsyncState as TakeAWhileDelegate;
if(d1 == null)throw new Exception("Invalid object type");
int result = d1.EndInvoke(ar);
Console.WriteLine("result:{0}",result);
}
}
```

示例 2:利用 Lambda 表达式创建回调

```
TakeAWhileDelegate d1 = TakesAWhile;
d1.BeginInvoke(1,300,ar =>
{
int result = d1.EndInvoke(ar);
Console.WriteLine("result:{0}",result);
},null);
TakeAWhileDelegate d2 = Functions.TakeAWhile2;
d2.BeiginInvoke(20,2000,ar =>
{
int result = d2.EndInvoke(ar);
Console.WriteLine("result2:{0}",result);
},null);
Console.ReadKey();
```

3.5 其他异步调用模式

HttpWebRequest 类的 BeginGetResponse(),异步发送 HTTP Web 请求;SqlCommand 类的 BeginExecuteReader()方法,给数据库发送异步请求。

4 Thread 类的应用

4.1 ThreadStart 委托应用

使用 Thread 类可以创建和控制线程,一个 Thread 实例表示一个线程。

Thread oneThread = new Thread(entryPoint);Thread 构造函数需要一个参数,用于指定线程的入口,使用 ThreadStrat 委托为线程传递工作方法。ThreadStart 委托表示一个返回类型为 void 的无参数方法。可使用 Lambda 表达式传递 ThreadStart 委托实例。通过 Thread 实例的 Start()方法开启线程

示例 1:传递委托

```
//主调线程
var t1 = new Thread(ThreadMain);
t1.Strat();
for(int i = 0;i<10;i++)
{
  Thread.Sleep(50);
  Console.WriteLine("Nice to meet you.");
}
//线程任务
static void ThreadMain()
{
for(int i = 0; i<10; i++)
{
Thread.Sleep(50);
Console.WriteLine("Nice to meet you too.");
}
}
```

示例 2:传递 Lambda 表达式

```
var t1 = new Theread(() =>
{
for(int i = 0;i<10;i++)
{
Thread.Sleep(30);
Console.WriteLine("Nice to meet you too.{0}",
Thread.CurrentThread.ManagedThreadId);
}
});
t1.Strat();
for(int i = 0;i<10;i++)
{
Thread.Sleep(50);
Console.WriteLine("Nice to meet you.{0}",
Thread.CurrentThread.ManagedThreadId);
}
```

其中 Thread.CurrentThreadId 获取当前工作线程的系统 ID。

4.2 ParameterizedThreadStart 委托

ThreadStart 委托仅仅指向无返回值,无参数的方法,虽然能满足大多数情况下的要求,但是,如果想把数据传递在给子线程上执行的方法,则需要使用 ParameterizedThreadStart 委托类型,使用带 Pa-

rameterizedThreadSrart 委托类型参数的构造函数创建 Thread 实例,ParameterizedThreadSrart 委托定义了带有一个 object 类型参数,返回类型为 void 的方法。同时创建一个自定义类,把线程的方法定义为实例方法,初始化该类实例数据后启动线程,从而为线程方法传递参数。

示例 1:ParameterizedThreadSrart 参数的构造函数

```
static void DoAction()
{
Thread t1 = new Thread(ThreadMainWithPa-
rameters);
var d = "info";
t1. Start(d);
}
static void ThreadMainWithParameters(object
o)
{
String d = (String)o;
for(int i = 0;i<10;i++)
{
Thread. Sleep(50);
Console. WriteLine("Running in a thread. re-
ceived{0}",
d. Message);
}
}
```

示例 2:自定义类传递参数

```
MyThread mt = new MyThread ("Hello
Thread");
Thread t1 = new Thread(mt. ThreadMain);
t1. Start();
Console. ReadKey();
Class MyThread
{
private string data;
public MyThread(string data)
{this. data = data;}
public void ThreadMain()
{
for(int i = 0;i<10;i++)
{
Thread. Sleep(50);
Console. WriteLine("Running in a thread,
received{0}",
this. data);
}
}
}
```

4.3 后台线程

只要有一个前台线程在运行,应用程序的进程就在运行,后台线程会在主线程结束时立即中止,而前台线程则继续完成后终止。默认情况下,用 Thread 类创建的线程都是前台线程,而线程池中的线程总是后台线程。将线程对象的 IsBackground 属性设置为 true,线程就为后台线程。

示例

```
var t1 = new Thread(ThreadMain)
{
Name = "myNewThread",
IsBackground = true
};
t1. Strat();
Console. WriteLine ("Main thread ending
now");
static void ThreadMain()
{
Console. WriteLine( $ "Thread [{Thread. Cur-
rentThread. Name}]strated");
Thread. Sleep(3000);
Console. WriteLine( $ "Thread[{Thread. Cur-
rentThread. Name}]completed");
}
```

5 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。线程数应在需要更多的线程时增加,在需要释放资源时减少。线程池可以成功地适应于任何需要大量短暂的开销大的资源的情形。我们事先分配一定的资源,将这些资源放入到资源池。每次需要新的资源,只需从池中获取一个,而不用创建一个新的。当该资源不再被使用,时,就将其返回到池中。

5.1 ThreadPool 类用法

ThreadPool 类型拥有一个 QueueUserWorkItem 静态方法。该静态方法接受一个委托,代表用户自定义的一个异步操作。在该方法被调用后,委,托会进入到内部队列中。如果池中没有任何线程,将创建一个新的工作线程(worker thread) 并将队列中第一个委托放入到该工作线程中。

示例

```
int nWorkThreads;
int nCompletionPortThreads;
ThreadPool. GetMaxThread ( out nWorkThreads,out nCompletionPortThreads);
Console. Writeline("Max worker threads:{0}, I/O completion threads:{1}",
nWorkThreads, nCompletionPortThreads);
for(inti = 0;i<5;i++)
{
ThreadPool. QueueUserWorkItem ( JobForAThread);
}
Thread. Sleep(3000);
static voidJobForAThread(object state)
{
for(int i=0;i<3;i++)
{
Console. WriteLine("Loop{0},running inside pooled thread{1}",
i,Thread. CurrentThread. ManagedThreadId);
Thread. Sleep(50);
}
}
```

6 结束语

我们通过对异步与多线程的学习,明白了在 C#编程中,当我们遇到程序要处理大量任务时需要通过异步委托或者是多线程并行的方式来进行操作,比如服务器要处理大量访问和下载请求的时候、获取大量的网页内容的时候、或者是从数据库查询大量数据的时候等情况。多线程并行操作往往能够为我们节约大量的时间和资源,同时通过异步委托,与主线程并行进行操作。

C#语言本就是一个简单的、现代的、通用的、面向对象的编程语言,通过对 C#语言的深入学习,不仅可以大大提高我们的编程能力,还可以节约更多的时间与资源。正如我们本文所学的多线程异步委托的编程方法,就是一个解决复杂程序的好途径。

作者简介

第一作者:姜礼盟(1997—),男,藏,四川省康定市,本科,四川大学锦城学院,研究方向:.NET 方向。

第二作者:彭向梅(1974—),女,汉,四川省邻水,硕士研究生,四川大学锦城学院,研究方向:软件工程。