C# 温故而知新： 线程篇(二)线程池和异步线程

1 什么是CLR线程池？

在上一章中通过Thread对象创建我们所需要的线程，但是创建线程的开销是很大的，在需要以性能为重的项目中这的确容易导致一些性能问题，其实我们所想象中的线程开销最好如下表示：

 1 尽量少的创建线程并且能将线程反复利用

2 最好不要销毁而是挂起线程达到避免性能损失

3 通过一个技术达到让应用程序一个个执行工作，类似于一个队列

4 如果某一线程长时间挂起而不工作的话，需要彻底销毁并且释放资源

5 如果线程不够用的话能够创建线程，并且用户可以自己定制最大线程创建的数量

 令人欣慰的是微软早就想到了以上几点，于是CLR线程池的概念出现了，说到底线程池就是一个帮助我们开发人员实现多线程的一个方案，就是用来存放“线程”的对象池，利用线程池我们可以开发出性能比较高的对于多线程的应用，同时减低一些不必要的性能损耗，我们不必去手动创建线程，线程池根据给定线程池中的任务队列的队列速度和相关任务执行速度相比较去自己添加或复用线程，关于线程池的细节我会在下文中详细阐述

2 简单介绍下线程池各个优点的实现细节

让我们根据上节中线程池已经实现了5个优点来详细介绍下线程池的功能

**1** **尽量少的创建线程并且能将线程反复利用：**初始化的线程池中是没有线程的，当应用程序区请求线程池时，线程池会制造一个初始线程，一般情况下，线程池会重复使用这个线程来经量少的创建线程,这样线程池就能尽量避免去创建新的线程而减少的创建线程的开销

**2 最好不要销毁而是挂起线程达到避免性能损失：**当一个线程池中的线程工作完毕之后，该线程不会被销毁而是被挂起操作等待，关于线程的挂起大家可以参考第一篇，如果应用程序又一次请求线程池的话，那么这个线程会重新被唤醒，从而是实现了线程的复用并且避免一定的性能损失

**3 通过一个技术达到让应用程序一个个执行工作，类似于一个队列：**多个应用程序请求线程池后，线程池会将各个应用程序排队处理，首先利用线程池中的一个线程对各个应用程序进行操作，如果应用程序的执行速度  超过了队列的排队速度时，线程池会去创建一个新的线程，否则复用原来的线程

4 **如果某一线程长时间挂起而不工作的话，需要彻底销毁并且释放资源：**有可能在多个程序请求线程池执行后，线程池中产生了许多挂起的线程，并且这些线程池中的线程会一直处于空闲状态间接导致的内存的浪费，所以微软为线程池设定了一个超时时间，当挂起的线程超时之后会自动销毁这些线程

5 **如果线程不够用的话能够创建线程：**前面已经提到过，有时候排在队列中的其中一个或多个应用程序工作时间超过了规定的每个应用程序的排队时间，那么线程池不会坐视不管，线程池会创建一个新的线程来帮助另一个需要执行的应用程序

相信大家看完上述5个优点及其细节后，对线程池的目的和优点就豁然开朗了

个人认为CLR线程池最牛的地方就是它能够根据队列中的应用程序执行时间和各个排队应用程序间的排队速度进行比较，从而决定是不是创建或者复用原先的线程，假如一系列的应用程序非常的简单或者执行速度很快的情况下，根本无需创建新的线程，从而这个单一线程可以悠闲的挂起等待排队的下一个应用程序。如果应用程序非常复杂或者层次不齐，那么正好相反，由于这个线程正在忙，所以无暇对排队的下个任务进行处理，所以需要创建一个新的线程处理，这样陆陆续续会创建一些新的线程来完成队列中的应用程序，如果在执行过程中多余线程会超时自动回收，而且CLR线程池允许用户自定义添加最大线程数和最小线程数，但是出于性能的考虑微软不建议开发人员手动更改线程池中的线程数量，对于以上几点大家务必理解

3 线程池ThreadPool的常用方法介绍

如果您理解了线程池目的及优点后，让我们温故下线程池的常用的几个方法：

1. public static Boolean QueueUserWorkItem(WaitCallback wc, Object state);

   WaitCallback回调函数就是前文所阐述的应用程序，通过将一些回调函数放入线程池中让其形成队列，然后线程池会自动创建或者复用线程去执行处理这些回调函数，

   State: 这个参数也是非常重要的，当执行带有参数的回调函数时，该参数会将引用传入，回调方法中，供其使用

3. public static bool SetMaxThreads(int workerThreads,int completionPortThreads);

4. public static bool SetMinThreads(int workerThreads,int completionPortThreads);

   3和4方法 CLR线程池类中预留的两个能够更改，线程池中的工作线程和I/O线程数量的方法。

  使用该方法时有两点必须注意：

   1.不能将辅助线程的数目或 I/O 完成线程的数目设置为小于计算机的处理器数目。

   2.微软不建议程序员使用这两个方法的原因是可能会影响到线程池中的性能

我们通过一个简单的例子来温故下

using System;using System.Threading;

namespace ThreadPoolApplication{

class Program{

//设定任务数量

static int count = 5;

static void Main(string[] args){

//关于ManualResetEvent大伙不必深究，后续章将会详细阐述，这里由于假设

//让线程池执行5个任务所以也为每个任务加上这个对象保持同步

ManualResetEvent[] events=new ManualResetEvent[count];

Console.WriteLine("当前主线程id:{0}",Thread.CurrentThread.ManagedThreadId);

//循环每个任务

for (int i = 0; i < count; i++){

//实例化同步工具

events[i]=new ManualResetEvent(false);

//Test在这里就是任务类，将同步工具的引用传入能保证共享区内每次只有一个线程进入

Test tst = new Test(events[i]);

Thread.Sleep(1000);

//将任务放入线程池中，让线程池中的线程执行该任务  
 ThreadPool.QueueUserWorkItem(tst.DisplayNumber, new { num1=2});

}

//注意这里，设定WaitAll是为了阻塞调用线程（主线程），让其余线程先执行完毕，

//其中每个任务完成后调用其set()方法(收到信号),当所有

//的任务都收到信号后，执行完毕，将控制权再次交回调用线程（这里的主线程）

ManualResetEvent.WaitAll(events);

Console.ReadKey();

}

}

public class Test {

ManualResetEvent manualEvent;

public Test(ManualResetEvent manualEvent) {

this.manualEvent = manualEvent;

}

public void DisplayNumber(object a) {

Console.WriteLine("当前运算结果:{0}",((dynamic)a).num1);

Console.WriteLine("当前子线程id:{0} 的状态:{1}", Thread.CurrentThread.ManagedThreadId,Thread.CurrentThread.ThreadState);

//这里是方法执行时间的模拟，如果注释该行代码，就能看出线程池的功能了

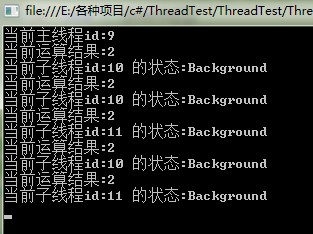
//Thread.Sleep(30000);

//这里是释放共享锁，让其他线程进入 manualEvent.Set();

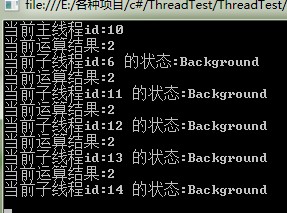
}

}

}

执行结果：

从显示结果能够看出线程池只创建了id为9,10,11这3个线程来处理这5个任务，因为每个任务的执行时间非常短，所以线程池的优势被展现出来了。如果我们去掉DisplayNumber方法中的Thread.Sleep(30000) 的注释的话，会发现由于任务的执行时间远远超于任务在队列中的排队时间，所以线程池开启了5个线程来执行任务



4 简单理解下异步线程

在很多时候例如UI或者IO操作时我们希望将这些很复杂且耗时比较长的逻辑交给后台线程去处理，而不想影响页面的正常运行，而且我们希望后台线程能够触发一个回调事件来提示该任务已经完成，所以基于这种需求越来越多而且在复杂的逻辑下也难以避免一些多线程的死锁，所以微软为我们提供了一个属于微软自己的异步线程的概念，上一章提到了多线程和异步的基本概念和区别大家可以去温故下， 线程异步指的是一个调用请求发送给被调用者，而调用者不用等待其结果的返回，一般异步执行的任务都需要比较长的时间，相信大家理解的异步的概念后都能对异步的根源有个初步的认识，和线程一样，异步也是针对执行方法而设计的，也就是说当我们执行一个方法时，使用异步方式可以不阻碍主线程的运行而独立运行，直到执行完毕后触发回调事件，注意，.net异步线程也是通过内部线程池建立的，虽然微软将其封装了起来，但是我们也必须了解下

5 异步线程的工作过程和几个重要的元素

由于委托是方法的抽象，那么如果委托上能设定异步调用的话，方法也能实现异步，所以本节用异步委托来解释下异步线程的工作过程

前文和前一章节中提到了多线程和异步的区别，对于异步线程来说，这正是体现了其工作方式：

 调用者发送一个请求 ->  调用者去做自己的事情 -> 请求会异步执行 -> 执行完毕可以利用回调函数告诉调用者（也可以不用）

在详细说明这几个过程之前，让我们来了解下下面的几个重要的元素

AsyncCallback 委托：其实这个委托是微软给我们提供的用于异步执行方法体后通知该异步方法已经完成。AsyncCallBack抽象了所有异步方法执行后回调函数（方法）,它规定了回调函数（方法）必须拥有一个IAsyncResult的参数并且没有返回值，

IAsyncResult 接口：让我们先来看下msdn上关于它的解释

1. IAsyncResult 接口由包含可异步操作的方法的类实现。它是启动异步操作的方法的返回类型，也是结束异步操作的方法的第三个参数的类型
2. 当异步操作完成时，IAsyncResult 对象也将传递给由 AsyncCallback 委托调用的方法

      对于第一条的解释，以下两条代码能够直观的理解：

* IAsyncResult result= doSomething.BeginInvoke(null,null);
* doSomething.EndInvoke(asyncResult);
* 其实IAsyncResult贯穿了异步执行的开始和结束
* 委托通过BeginInvoke和EndInvoke来启动异步和结束异步
* 每个委托的BeginInvoke方法都暴露或返回了实现IAsyncResult类型的接口对象的根本目的是可以让该异步方法体自由的代码中控制，有时候主线程需要等待异步执行后才能执行，虽然这违背的异步的初衷但是还是可以纳入可能的需求行列，所以如果我们在beginInoke后立刻使用EndInvoke的话，主线程（调用者）会被阻塞，直到异步线程执行完毕后在启动执行

对于第二条的解释:   结束异步操作时需要使用的回调方法，这里IAsyncResult作为参数被传递进了个这方法，这时IAsyncResult起到了向回调方法传递信息的作用，关于这点会在后文的异步线程的工作过程中详细解释下

    我们最后再来看下IAsyncResult的几个重要属性



   在这里再次强调下IAsyncResult第一个属性AsyncState的作用，就像前面所说，有时我们需要将回调函数的参数传入到回调方法体中，当然传入入口在BeginInvoke的第二个参数中，在回调函数体中我们可以通过将这个属性类型转换成和BeginInvoke第二个参数一摸一样的类型后加以使用

   关于IAsyncResult最后还有一点补充：如果IAsyncResult本身的功能还不能满足你的需要的话，可以自定义实现自己的AsyncResult类，但必须实现这个接口

      理解了以上两个关于异步至关重要的2个元素后，让我们进入一段段代码，在来详细看下异步线程的执行过程

//定义一个委托

public delegate void DoSomething();

static void Main(string[] args){

//1.实例化一个委托，调用者发送一个请求，请求执行该方法体（还未执行）

DoSomething doSomething = new DoSomething(() => {

Console.WriteLine("如果委托使用beginInvoke的话，这里便是异步方法体");

//4，实现完这个方法体后自动触发下面的回调函数方法体 });

//3 。调用者（主线程）去触发异步调用，采用异步的方式请求上面的方法体

IAsyncResult result= doSomething.BeginInvoke(

//2.自定义上面方法体执行后的回调函数

new AsyncCallback(

//5.以下是回调函数方法体

//asyncResult.AsyncState其实就是AsyncCallback委托中的第二个参数

asyncResult => {

doSomething.EndInvoke(asyncResult);

Console.WriteLine(asyncResult.AsyncState.ToString());

}

)

, "BeginInvoke方法的第二个参数就是传入AsyncCallback中的AsyncResult.AsyncState,我们使用时可以强转成相关类型加以使用");

//DoSomething......调用者（主线程）会去做自己的事情 Console.ReadKey();

}

       大家仔细看这面这段非常简单的代码，为了大家理解方便我特意为异步执行过程加上了特有的注释和序列号，这样的话，大伙能直观初步的理解了异步的执行过程。

       让我们根据序列号来说明下：

 1.  实例化一个委托，调用者发送一个请求，请求执行该方法体（还未执行）首先将委实例化并且定义好委托所请求的方法体，但是这个时候方法体是不会运行的

2.  这时候和第一步所相似的是，这里可以将定义好的回调函数AsyncCallback方法体写入BeginInvoke的第一个参数，将需要传入回调方法体的参数放入第二个参数

3．调用者（主线程）去触发异步调用（执行BeginInvoke方法），采用异步的方式执行委托中的方法体

4．实现完这个方法体后自动触发下面的AsyncCallback中的方法体回调函数（可以设定回调函数为空来表示不需要回调）

5 . 执行回调函数方法体，注意使用委托的 EndInvoke方法结束异步操作，并且输出显示传入异步回调函数的参数

    再次强调第五点：(1) 由于使用了回调函数，所以必然异步方法体已经执行过了，所以在回调函数中使用EndInvoke方法是不会阻塞的，  (2) 能通过EndInvoke方法获得一些返回结果，例如FileStream.EndRead()能够返回读取的字节数等等

6 有必要简单介绍下Classic Async Pattern 和Event-based Async Pattern

首先介绍下Classic Async Pattern：其实Classic Async Pattern指的就是我们常见的BeginXXX和EndXXX

IAsyncResult 异步设计模式通过名为 BeginOperationName 和 EndOperationName 的两个方法来实现原同步方法的异步调用

让我们再来回顾下.net中的几个的BeginXXX 和EndXXX

 Stream中的BeginRead,EndRead,BeginWrite,EndWrite

Socket中的BeginReceive，EndReceive

HttpWebRequest的BeginGetRequestStream和EndGetRequestStream...

再来介绍下Event-based Async Pattern：Event-based Async Pattern 值的是类似于 xxxxxxxAsync() 和 类似于event xxxxxCompleteHander。通过一个方法和一个完成事件来处理异步操作。.net中的例子： WebClient.DownloadStringAsync(string uri)和 event DownloadStringCompleteEventHandler

其实Classic Async Pattern和Event-based Async Pattern都是一种异步的设计思路，我们也可以根据这一系列的思路去实现自己的异步方法

7 异步线程的发展趋势以及.net4.5中异步的简化

微软貌似现在把精力放在win8或WinPhone的metro上，而且记得在win 8开发者培训的会议上，着重阐述了微软对于异步的支持将越来越强，而且为了快速响应诸如移动设备的应用程序，微软也在争取为每个方法都实现一个异步版本…..可见异步的重要性，相信异步的发展趋势是个不错的上升曲线，还没反应过来.net4.5的异步新特性便诞生了。首先经历过异步摧残的我们，都会有这样一个感受，往往回调方法和普通方法会搞错，在复杂的项目面前，有时候简直无法维护，到处都是回调函数，眼花缭乱 所以微软为了简化异步的实现过程，甚至大刀阔斧将回调函数做成看起来像同步方法，虽然觉得很诡异，还是让我们初步了解下这种异步的新特性

先看代码

/// <summary>.net 4.5 中 async 和 await 全新的关键字 一起实现异步的简化</summary>

void async ShowUriContent(string uri){

using (FileStream fs = File.OpenRead("你的文件地址")){

using (FileStream fs2 = new FileStream()){

byte[] buffer = new byte[4096];

//FileStream的ReadAsync方法也是net4.5版本出现的，它返回一个Task<int>对象

//而且作用于await后的异步代码会等待阻塞直到异步方法完成后返回

int fileBytesLength = await fs.ReadAsync(buffer,0,buffer.Length).ConfigureAwait(false);

while(fileBytesLength>0){

//FileStream的WriteAsync方法也是net4.5版本出现的

await fs2.WriteAsync(buffer,0,buffer.Length).ConfigureAwait(false);

}

}

}

}

相信看完代码后大家有耳目一新的感觉，不错，原本异步调用的回调函数不见了，取而代之的是await和方法声明上的async关键字，新特性允许我们实现这俩个关键字后便能在方法中实现“同步方式”的异步方法，其实这解决了一些棘手的问题，诸如原本需要在回调事件里才能释放的文件句柄在这里和同步方法一样，使用using便搞定了，还有截获异常等等，都不用像之前那样痛苦了，这里还有一些东东需要关注下，大家先不用去深究ConfigureAwait这个方法，由于ReadAsync和 WriteAsync方法是.net 4.5新加的属于返回Task<int>类型的方法所以使用ConfigureAwait方法能够将数值取到，关于Task泛型类我会在今后的章节中详细阐述

8 本章示例

自定义一个简单的线程池

static void Main(string[] args){

ThreadStart[] startArray ={

new ThreadStart(()=>{Console.WriteLine("第一个任务");}),

new ThreadStart(()=>{Console.WriteLine("第二个任务");}),

new ThreadStart(()=>{Console.WriteLine("第三个任务");}),

new ThreadStart(()=>{Console.WriteLine("第四个任务");}),

};  
  MyThreadPool.SetMaxWorkThreadCount(2);  
 MyThreadPool.MyQueueUserWorkItem(startArray);

Console.ReadKey();

}

/// <summary>自定义一个简单的线程池，该线程池实现了默认开启线程数，当最大线程数全部在繁忙时，循环等待，只到至少一个线程空闲为止，本示例使用BackgroundWorker模拟后台线程，任务将自动进入队列和离开队列</summary>

sealed class MyThreadPool{

//线程锁对象

private static object lockObj = new object();

//任务队列

private static Queue<ThreadStart> threadStartQueue = new Queue<ThreadStart>();

//记录当前工作的任务集合，从中可以判断当前工作线程使用数，如果使用int判断的话可能会有问题，

//用集合的话还能取得对象的引用，比较好

private static HashSet<ThreadStart> threadsWorker = new HashSet<ThreadStart>();

//当前允许最大工作线程数

private static int maxThreadWorkerCount = 1;

//当前允许最小工作线程数

private static int minThreadWorkerCount = 0;

/// <summary>/// 设定最大工作线程数/// </summary>

/// <param name="maxThreadCount">数量</param>

public static void SetMaxWorkThreadCount(int maxThreadCount) {

maxThreadWorkerCount =minThreadWorkerCount>maxThreadCount?

minThreadWorkerCount : maxThreadCount;

}

/// <summary>/// 设定最小工作线程数/// </summary>

/// <param name="maxThreadCount">数量</param>

public static void SetMinWorkThreadCount(int minThreadCount){

minThreadWorkerCount = minThreadCount > maxThreadWorkerCount ?maxThreadWorkerCount : minThreadCount;

}

/// <summary>/// 启动线程池工作/// </summary>

/// <param name="threadStartArray">任务数组</param>

public static void MyQueueUserWorkItem(ThreadStart[] threadStartArray){

//将任务集合都放入到线程池中 AddAllThreadsToPool(threadStartArray);

//线程池执行任务 ExcuteTask();

}

/// <summary>/// 将单一任务加入队列中/// </summary>

/// <param name="ts">单一任务对象</param>

private static void AddThreadToQueue(ThreadStart ts){

lock (lockObj){

threadStartQueue.Enqueue(ts);

}

}

/// <summary>/// 将多个任务加入到线程池的任务队列中/// </summary>

/// <param name="threadStartArray">多个任务</param>

private static void AddAllThreadsToPool(ThreadStart[] threadStartArray){

foreach (var threadStart in threadStartArray)

AddThreadToQueue(threadStart);

}

/// <summary>

/// 执行任务，判断队列中的任务数量是否大于0，如果是则判断当前正在使用的工作线程的，数量是否大于等于允许的最大工作线程数，如果一旦有线程空闲的话， 就会执行ExcuteTaskInQueen方法处理任务。</summary>

private static void ExcuteTask(){

while (threadStartQueue.Count > 0){

if (threadsWorker.Count < maxThreadWorkerCount){ExcuteTaskInQueen();}

}

}

/// <summary>/// 执行出对列的任务，加锁保护/// </summary>

private static void ExcuteTaskInQueen(){

lock (lockObj){

ExcuteTaskByThread(threadStartQueue.Dequeue());

}

}

/// <summary>实现细节，这里使用BackGroudWork来实现后台线程，注册doWork和Completed事件，当执行一个任务前，前将任务加入到工作任务集合（表示工作线程少了一个空闲），一旦RunWorkerCompleted事件被触发则将任务从工作任务集合中移除（表示工作线程也空闲了一个）</summary>

private static void ExcuteTaskByThread(ThreadStart threadStart){

threadsWorker.Add(threadStart);

BackgroundWorker worker = new BackgroundWorker();

worker.DoWork += (o, e) => { threadStart.Invoke(); };

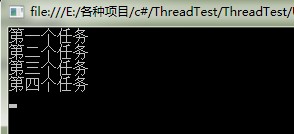
worker.RunWorkerCompleted += (o, e) => { threadsWorker.Remove(threadStart); };

worker.RunWorkerAsync();

}

}

 显示结果：



Asp.net异步IHttpAsyncHandler示例

有时我们需要使用IHttpAsyncHandler来异步实现一些特定的功能，让我用很简单的示例来阐述这个过程

1：首先编写Handler1的逻辑，注意要继承IHttpAsyncHandler接口

/// <summary>异步IHttpHandler,实现了一个简单的统计流量的功能，由于是示例代码，所以没有判断IP或者MAC</summary>

public class Handler1 : IHttpAsyncHandler{

//默认访问量是0

static int visitCount = 0;

/// <summary>这个HttpHandler的同步方法</summary>

/// <param name="context"></param>

public void ProcessRequest(HttpContext context){}

public bool IsReusable{get{return false;}}

/// <summary>实现IHttpAsyncHandler 接口方法</summary>

/// <param name="context">当前HttpContext</param>

/// <param name="cb">回调函数</param>

/// <param name="extraData"></param>

/// <returns></returns>

public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb, object extraData){

//这里可以加上判断IP或mac的方法

visitCount++;

//实例化AsyncUserVisiteCounterResult对象

AsyncUserVisiteCounterResult result = new AsyncUserVisiteCounterResult(cb, visitCount, context);

result.Display();

return result;

}

/// <summary>结束本次IHttpAsyncHandler工作时触发的request方法</summary>

public void EndProcessRequest(IAsyncResult result){}

}

/// <summary>自定义IAsyncResult 实现我们额外的Display方法</summary>

public class AsyncUserVisiteCounterResult : IAsyncResult{

//回调参数

private object \_param;

//是否异步执行完成

private bool \_asyncIsComplete;

//回调方法

private AsyncCallback \_callBack;

//当前上下文

private HttpContext \_context;

public AsyncUserVisiteCounterResult(AsyncCallback callBack, object stateParam, HttpContext context){

this.\_callBack = callBack;

this.\_param = stateParam;

\_asyncIsComplete = false;

this.\_context = context;

}

public object AsyncState{get { return this.\_param; }}

/// <summary>等待句柄用于同步功能，关于等待句柄会在后续章节陈述/// </summary>

public System.Threading.WaitHandle AsyncWaitHandle{get{return null;}}

/// <summary>/// 该属性表示不需要异步任务同步完成/// </summary>

public bool CompletedSynchronously{get { return false; }}

/// <summary>/// 该属性表示异步任务是否已经执行完成/// </summary>

public bool IsCompleted{get{return this.\_asyncIsComplete;}}

/// <summary>/// 自定义的额外功能,需要注意的是，执行完异步功能后要将\_asyncIsComplete设置为true表示任务执行完毕而且 执行回调方法，否则异步工作无法结束页面会卡死</summary>

public void Display(){

//这里先不用waitHandle句柄来实现同步

lock (this){

this.\_context.Response.Write("你是第" + (int)this.\_param + "位访问者，访问时间:"+DateTime.Now.ToString());

this.\_asyncIsComplete = true;

this.\_callBack(this);

}

}

}

2 在web.config中添加相应的配置，注意path指的是.ashx所在的路径，指的是相应的文件类型

    <httpHandlers>  
      <add verb="\*" path="AsyncThreadInAsp.net.Handler1.ashx" type="AsyncThreadInAsp.net.Handler1"/>  
    </httpHandlers>

3 最后在页面中访问

