[浅谈线程池（上）：线程池的作用及CLR线程池](http://www.cnblogs.com/JeffreyZhao/archive/2009/07/22/thread-pool-1-the-goal-and-the-clr-thread-pool.html)

线程池是一个重要的概念。不过我发现，关于这个话题的讨论似乎还缺少了点什么。作为资料的补充，以及今后文章所需要的引用，我在这里再完整而又简单地谈一下有关线程池，还有.NET中各种线程池的基础。更详细的内容就不多作展开了，有机会我们再详细讨论这方面的细节。这次，还是一个“概述”性质的，希望可以说明白这方面问题的一些概念。

线程池的作用

其实“线程池”就是用来存放“线程”的对象池。

在程序中，如果某个创建某种对象所需要的代价太高，同时这个对象又可以反复使用，那么我们往往就会准备一个容器，用来保存一批这样的对象。于是乎，我们想要用这种对象时，就不需要每次去创建一个，而直接从容器中取出一个现成的对象就可以了。由于节省了创建对象的开销，程序性能自然就上升了。这个容器就是“池”。很容易理解的是，因为有了对象池，因此在用完对象之后必须有一个“归还”的动作，这样便可以把对象放回池中，下次需要的时候就可以再次拿出来使用了。

例如，我们在使用ADO.NET连接SQL Server时，.NET框架就会自动帮我们维护一个连接池，这就是因为重新创建一个连接的代价相对比较高昂，“复用”就显得比较划算了。不过有些朋友可能会说，我们明明是每次都创建一个SqlConnection对象，哪里有“复用”啊？这是因为.NET框架中把“连接池”做透明了，对于程序员完全隐藏了这个概念。每次我们虽然创建的是新的SqlConnection对象，但是这个对象内部占用的“数据库连接”还是会复用的。为什么总是强调用完SqlConnection对象后要及时“关闭”（Dispose或Close）呢？其实这里并没有断开数据库连接，只是把这个连接放回了连接池。等到下次创建新的SqlConnection对象时，这个连接又可以拿出来用了。

既然我们每次都是从池中获取对象，那么这些对象是由谁来创建，又是什么时候创建的呢？这个就要根据不同情况由各对象池来自行实现了。例如，可以在创建对象池的时候指定池内对象数量，并且一下子全部创建好，当然您也可以在得到请求时，如果发现池中已经没有剩余对象时创建。您也可以“事前”先准备一部分，“事中”根据需要再继续补充。还可以做得“智能”一些，例如，根据实际情况添加或删除一些对象，甚至对需求“走势”进行“预测”，在空闲时便创建更多的对象以备“不时之需”。各中变化难以言尽。

当然，它们的原理和目的是类似的。相信上面这段文字也已经讲清了“线程池”的作用：因为创建一个线程的代价较高，因此我们使用线程池设法复用线程。就是这么简单。

CLR线程池

在.NET中，CLR线程和操作系统线程对应，您可以简单地认为.NET中的Thread对象便封装了一个操作系统线程，并附带一些托管环境下所需要的数据（如GC Handle）1。而CLR线程池便是存放这些CLR线程的对象池。

我们在编写程序的时候，可以使用ThreadPool类的两个静态方法：QueueUserWorkItem和UnsafeUserQueueWorkItem向CLR线程池中添加任务（一个WorkCallback委托对象），这两个方法的区别，在于前者会收集调用方的ExecutionContext，也就是保留了的当前线程的执行信息（如认证或语言文化等），使任务最终会在“创建”时刻的环境中执行2——后者就不会。因此，如果比较两个方法的绝对性能，Unsafe方法会略胜一筹。但是平时还是建议使用QueueUserWorkItem方法，因为保留执行上下文会避免很多麻烦事情，且这点性能损耗其实算不上什么。

CLR线程池在.NET框架中的作用很大，除了让程序员使用之外，其他一些功能也会依赖CLR线程池。如ThreadPool.RegisterWaitForSingleObject方法，或是System.Threading.Timer组件——还有更重要可能也是更隐藏的：ASP.NET在得到一个请求后，也会将这个请求处理的任务交由CLR线程池去执行——请注意，它们最多只是添加任务而已，并不表示任务会立即执行。所有添加到CLR线程池的任务都会在合适的时候得以执行——可能马上，也可能要稍等片刻，甚至更久。

向CLR线程池添加任务时，任务会被临时放到一个队列中，并在合适的时候执行。那么怎么样才算是“合适的时候”？简单的概括说来，便是线程池内有空闲的线程，或线程池所管理的线程数量还没有达到上限的时候。如果有空闲的线程，线程池就会立即让它领取一个任务执行。如果是第二种情况，线程池便会创建新的Thread对象。由于让操作系统管理太多线程反而会造成性能下降，因此CLR线程池会有一个上限。不同的托管环境会设置不同的上限。如在.NET 2.0 SP1之后，普通的Windows应用程序（如控制台或WinForm/WPF），会将其设置为“处理器数 \* 250”。也就是说，如果您的机器为2个2核CPU，那么CLR线程池的容量默认上限便是1000，也就是说，它最多可以管理1000个线程同时运行——很多情况下这已经是一个很可怕的数字了，如果您觉得这还不够，那么就应该考虑一下您的实现方式是否可以改进了。

对于ASP.NET应用程序来说，CLR线程池容量代表了应用程序最多可以同时执行的请求数量。对于托管在IIS上的ASP.NET执行环境来说，这个值由全局配置决定。这个配置在machine.config文件中system.web/processModel节点中，为maxWorkerThreads属性，它决定了为单个处理器分配的线程数。如果这个值为40，且机器上拥有4个处理器（2 \* 2CPU），那么这台机器目前的配置表示在同一时刻，ASP.NET可以同时处理160个请求。某些参考资料建议您将其修改为每处理器80-100个线程，这时您只要修改相应的属性值就可以了。

既然有最大值，也就相应有了最小值，它代表了CLR线程池“总是会保留”的最少线程数量。由于线程会占用资源，如在默认情况下，每个线程将获得1MB大小的栈空间3。所以如果在系统中保留太多空闲线程对资源也是一种浪费。因此，CLR线程池在使用大量线程处理完大量任务之后，也会逐步地释放线程，直至到达最小值。CLR线程池的最小线程数量确保了在任务数量较少的情况下，新来的任务可以立即执行，从而省去了创建新线程的时间。在普通应用程序中这个值为“处理器数 \* 1”，而在ASP.NET应用程序中这个值配置在machine.config文件中system.web/processModel节点的minWorkerThreads属性中4。

在某些时候可能会遇到这样的情况：在一个瞬间忽然来大量任务，每个任务的执行时间说长不长说短不短，不过足以导致线程池快速分配数百个线程。如果这个峰值之后就一片平静，那么势必造成大量空闲的线程，这种开销对性能的损耗也非常明显。因此，CLR线程池限制了线程的创建速度不超过每秒2个。这样，即使在某个瞬时获得了大量的任务，CLR线程池也可以使用相对较少的线程来完成所有工作5。

但是，还有一种情况也值得考虑。例如，对于一个比较繁忙的Web应用程序来说，一打开便会涌入大量的连接。由于线程的创建速度有限，因此可以执行的请求数量也只能慢慢增加。对于这种您预料到会产生大量线程，而且忙碌状况会持续一段时间的情况，限制线程的创建速度反而会带来损伤效率。这时，您就可以手动设置CLR线程池的最小线程数量。如果此时CLR线程池中拥有的线程数量较少，那么系统就会立即创建一定数量的线程来达到这个最小值。设置和获取CLR线程池最小线程数量的接口为：

public static class ThreadPool

{

public static void GetMinThreads(out int workerThreads, out int completionPortThreads);

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

}

这两个接口的作用和使用方式应该足够明显了（不理解的话可以查阅MSDN），其中workerThreads参数便是CLR线程池的最小线程数，而completionPortThreads涉及到我们下次要讨论IO线程池，在此就不多作展开了。除了设置和读取CLR最小线程数的方法之外，ThreadPool还包含这些接口：

public static class ThreadPool

{

public static void GetMaxThreads(out int workerThreads, out int completionPortThreads);

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

public static void GetAvailableThreads(out int workerThreads, out int completionPortThreads);

}

值得注意的是，无论是设置还是获取到的这些数值，都与处理器数量没有任何关系了。也就是说，在一台2 \* 2CPU的机器上运行一个普通的.NET应用程序时：

* 调用GetMaxThreads方法将获得1000，表示CLR线程池最大容量为1000（250 \* 4），而不是250。
* 调用SetMinThreads并传入100，表示CLR线程池所拥有的最小线程数量为100，而不是400（100 \* 4）。

对于CLR线程池的简单描述就暂时先到这里了。如果您还有什么疑问请提出，我会加以补充。

**注1：**严格说来，Thread对象和系统线程对应关系还有些细节上的考虑。例如，Thread对象只有当真正Start了之后，CLR才会创建一个操作系统线程与它绑定。

**注2：**ExecutionContext是个很重要且很有用的对象，例如，WinForms或WPF的异步任务中操作界面元素抛出异常该怎么办呢？

**注3：**使用Windows API或Thread类创建线程时可以指定它的栈空间大小，但是CLR线程池中的线程只能使用默认值——不过这个默认值也和托管环境有关，如普通应用程序默认为1MB，而ASP.NET为250KB，这意味着ASP.NET应用程序相对更容易产生Stack Overflow异常。

**注4：**可惜的是，对于processModel节点的数据，ASP.NET只会读取machine.config中的全局配置信息，这意味着我们不能使用web.config为不同应用程序配置不同的参数。如果我们要实现应用程序级别的配置，那么必须使用ThreadPool类中提供的API进行设置，这点稍后便会提到。

**注5：**对于这点，您不妨来做一个算术题：线程池内一下子涌入了500个任务，每个任务阻塞或暂停5秒，每个线程占用1MB内存，假设线程池目前为空，且有着足够的容量，此外线程创建速度也足够快，那么在限制及不限制线程创建速度的情况下，完成这些任务需要多少时间和内存空间？

[浅谈线程池（中）：独立线程池的作用及IO线程池](http://www.cnblogs.com/JeffreyZhao/archive/2009/07/24/thread-pool-2-dedicate-pool-and-io-pool.html)

在[上一篇文章](http://www.cnblogs.com/JeffreyZhao/archive/2009/07/22/thread-pool-1-the-goal-and-the-clr-thread-pool.html)中，我们简单讨论了线程池的作用，以及CLR线程池的一些特性。不过关于线程池的基本概念还没有结束，这次我们再来补充一些必要的信息，有助于我们在程序中选择合适的使用方式。

独立线程池

上次我们讨论到，在一个.NET应用程序中会有一个CLR线程池，可以使用ThreadPool类中的静态方法来使用这个线程池。我们只要使用QueueUserWorkItem方法向线程池中添加任务，线程池就会负责在合适的时候执行它们。我们还讨论了CLR线程池的一些高级特性，例如对线程的最大和最小数量作限制，对线程创建时间作限制以避免突发的大量任务消耗太多资源等等。

那么.NET提供的线程池又有什么缺点呢？有些朋友说，一个重要的缺点就是功能太简单，例如只有一个队列，没法做到对多个队列作轮询，无法取消任务，无法设定任务优先级，无法限制任务执行速度等等。不过其实这些简单的功能，倒都可以通过在CLR线程池上增加一层（或者说，通过封装CLR线程池）来实现。例如，您可以让放入CLR线程池中的任务，在执行时从几个自定义任务队列中挑选一个运行，这样便达到了对多个队列作轮询的效果。因此，在我看来，CLR线程池的主要缺点并不在此。

我认为，CLR线程池的主要问题在于“大一统”，也就是说，整个进程内部几乎所有的任务都会依赖这个线程池。如前篇文章所说的那样，如Timer和WaitForSingleObject，还有委托的异步调用，.NET框架中的许多功能都依赖这个线程池。这个做法是合适的，但是由于开发人员对于统一的线程池无法做到精确控制，因此在一些特别的需要就无法满足了。举个最常见例子：控制运算能力。什么是运算能力？那么还是从线程讲起吧1。

我们在一个程序中创建一个线程，安排给它一个任务，便交由操作系统来调度执行。操作系统会管理系统中所有的线程，并且使用一定的方式进行调度。什么是“调度”？调度便是控制线程的状态：执行，等待等等。我们都知道，从理论上来说有多少个处理单元（如2 \* 2 CPU的机器便有4个处理单元），就表示操作系统可以同时做几件事情。但是线程的数量会远远超过处理单元的数量，因此操作系统为了保证每个线程都被执行，就必须等一个线程在某个处理器上执行到某个情况的时候，“换”一个新的线程来执行，这便是所谓的“上下文切换（context switch）”。至于造成上下文切换的原因也有多种，可能是某个线程的逻辑决定的，如遇上锁，或主动进入休眠状态（调用Thread.Sleep方法），但更有可能是操作系统发现这个线程“超时”了。在操作系统中会定义一个“时间片（timeslice）”2，当发现一个线程执行时间超过这个时间，便会把它撤下，换上另外一个。这样看起来，多个线程——也就是多个任务在同时运行了。

值得一提的是，对于Windows操作系统来说，它的调度单元是线程，这和线程究竟属于哪个进程并没有关系。举个例子，如果系统中只有两个进程，进程A有5个线程，而进程B有10个线程。在排除其他因素的情况下，进程B占有运算单元的时间便是进程A的两倍。当然，实际情况自然不会那么简单。例如不同进程会有不同的优先级，线程相对于自己所属的进程还会有个优先级；如果一个线程在许久没有执行的时候，或者这个线程刚从“锁”的等待中恢复，操作系统还会对这个线程的优先级作临时的提升——这一切都是牵涉到程序的运行状态，性能等情况的因素，有机会我们在做展开。

现在您意识到线程数量意味着什么了没？没错，就是我们刚才提到的“运算能力”。很多时候我们可以简单的认为，在同样的环境下，一个任务使用的线程数量越多，它所获得的运算能力就比另一个线程数量较少的任务要来得多。运算能力自然就涉及到任务执行的快慢。您可以设想一下，有一个生产任务，和一个消费任务，它们使用一个队列做临时存储。在理想情况下，生产和消费的速度应该保持相同，这样可以带来最好的吞吐量。如果生产任务执行较快，则队列中便会产生堆积，反之消费任务就会不断等待，吞吐量也会下降。因此，在实现的时候，我们往往会为生产任务和消费任务分别指派独立的线程池，并且通过增加或减少线程池内线程数量来条件运算能力，使生产和消费的步调达到平衡。

使用独立的线程池来控制运算能力的做法很常见，一个典型的案例便是[SEDA架构](http://www.eecs.harvard.edu/~mdw/proj/seda/)：整个架构由多个Stage连接而成，每个Stage均由一个队列和一个独立的线程池组成，调节器会根据队列中任务的数量来调节线程池内的线程数量，最终使应用程序获得优异的并发能力。

在Windows操作系统中，[Server 2003及之前版本的API](http://msdn.microsoft.com/en-us/library/ms686756(VS.85).aspx)也只提供了进程内部单一的线程池，不过在[Vista及Server 2008的API](http://msdn.microsoft.com/en-us/library/ms686766(VS.85).aspx)中，除了改进线程池的性能之外，还提供了在同一进程内创建多个线程池的接口。很可惜，.NET直到如今的4.0版本，依旧没有提供构建独立线程池的功能。构造一个优秀的线程池是一件相当困难的事情，幸运的是，如果我们需要这方面的功能，可以借助著名的[SmartThreadPool](http://www.codeproject.com/KB/threads/smartthreadpool.aspx)，经过那么多年的考验，相信它已经足够成熟了。如果需要，我们还可以对它做一定修改——毕竟在不同情况下，我们对线程池的要求也不完全相同。

IO线程池

IO线程池便是为异步IO服务的线程池。

访问IO最简单的方式（如读取一个文件）便是阻塞的，代码会等待IO操作成功（或失败）之后才继续执行下去，一切都是顺序的。但是，阻塞式IO有很多缺点，例如让UI停止响应，造成上下文切换，CPU中的缓存也可能被清除甚至内存被交换到磁盘中去，这些都是明显影响性能的做法。此外，每个IO都占用一个线程，容易导致系统中线程数量很多，最终限制了应用程序的伸缩性。因此，我们会使用“异步IO”这种做法。

在使用异步IO时，访问IO的线程不会被阻塞，逻辑将会继续下去。操作系统会负责把结果通过某种方法通知我们，一般说来，这种方式是“回调函数”。异步IO在执行过程中是不占用应用程序的线程的，因此我们可以用少量的线程发起大量的IO，所以应用程序的响应能力也可以有所提高。此外，同时发起大量IO操作在某些时候会有额外的性能优势，例如磁盘和网络可以同时工作而不互相冲突，磁盘还可以根据磁头的位置来访问就近的数据，而不是根据请求的顺序进行数据读取，这样可以有效减少磁头的移动距离。

Windows操作系统中有多种异步IO方式，但是性能最高，伸缩性最好的方式莫过于传说中的“IO完成端口（I/O Completion Port，IOCP）”了，这也是.NET中封装的唯一异步IO方式。大约一年半前，老赵写过一篇文章《[正确使用异步操作](http://www.cnblogs.com/JeffreyZhao/archive/2008/02/24/use-async-operation-properly.html)》，其中除了描述计算密集型和IO密集型操作的区别和效果之外，还简单地讲述了IOCP与CLR交互的方式，摘录如下：

当我们希望进行一个异步的IO-Bound Operation时，CLR会（通过Windows API）发出一个IRP（I/O Request Packet）。当设备准备妥当，就会找出一个它“最想处理”的IRP（例如一个读取离当前磁头最近的数据的请求）并进行处理，处理完毕后设备将会（通过Windows）交还一个表示工作完成的IRP。CLR会为每个进程创建一个IOCP（I/O Completion Port）并和Windows操作系统一起维护。IOCP中一旦被放入表示完成的IRP之后（通过内部的ThreadPool.BindHandle完成），CLR就会尽快分配一个可用的线程用于继续接下去的任务。

不过事实上，使用Windows API编写IOCP非常复杂。而在.NET中，由于需要迎合标准的APM（异步编程模型），在使用方便的同时也放弃一定的控制能力。因此，在一些真正需要高吞吐量的时候（如编写服务器），不少开发人员还是会选择直接使用Native Code编写相关代码。不过在绝大部分的情况下，.NET中利用IOCP的异步IO操作已经足以获得非常优秀的性能了。使用APM方式在.NET中使用异步IO非常简单，如下：

static void Main(string[] args)

{

WebRequest request = HttpWebRequest.Create("http://www.cnblogs.com");

request.BeginGetResponse(HandleAsyncCallback, request);

}

static void HandleAsyncCallback(IAsyncResult ar)

{

WebRequest request = (WebRequest)ar.AsyncState;

WebResponse response = request.EndGetResponse(ar);

// more operations...

}

BeginGetResponse将发起一个利用IOCP的异步IO操作，并在结束时调用HandleAsyncCallback回调函数。那么，这个回调函数是由哪里的线程执行的呢？没错，就是传说中“IO线程池”的线程。.NET在一个进程中准备了两个线程池，除了上篇文章中所提到的CLR线程池之外，它还为异步IO操作的回调准备了一个IO线程池。IO线程池的特性与CLR线程池类似，也会动态地创建和销毁线程，并且也拥有最大值和最小值（可以参考上一篇文章列举出的API）。

只可惜，IO线程池也仅仅是那“一整个”线程池，CLR线程池的缺点IO线程池也一应俱全。例如，在使用异步IO方式读取了一段文本之后，下一步操作往往是对其进行分析，这就进入了计算密集型操作了。但对于计算密集型操作来说，如果使用整个IO线程池来执行，我们无法有效的控制某项任务的运算能力。因此在有些时候，我们在回调函数内部会把计算任务再次交还给独立的线程池。这么做从理论上看会增大线程调度的开销，不过实际情况还得看具体的评测数据。如果它真的成为影响性能的关键因素之一，我们就可能需要使用Native Code来调用IOCP相关API，将回调任务直接交给独立的线程池去执行了。

我们也可以使用代码来操作IO线程池，例如下面这个接口便是向IO线程池递交一个任务：

public static class ThreadPool

{

public static bool UnsafeQueueNativeOverlapped(NativeOverlapped\* overlapped);

}

NativeOverlapped包含了一个IOCompletionCallback回调函数及一个缓冲对象，可以通过Overlapped对象创建。Overlapped会包含一个被固定的空间，这里“固定”的含义表示不会因为GC而导致地址改变，甚至不会被置换到硬盘上的Swap空间去。这么做的目的是迎合IOCP的要求，但是很明显它也会降低程序性能。因此，我们在实际编程中几乎不会使用这个方法3。

**注1：**如果没有加以说明，我们这里谈论的对象默认为XP及以上版本的Window操作系统。

**注2：**timeslice又被称为quantum，不同操作系统中定义的这个值并不相同。在Windows客户端操作系统（XP，Vista）中时间片默认为2个clock interval，在服务器操作系统（2003，2008）中默认为12个clock interval（在主流系统上，1个clock interval大约10到15毫秒）。服务器操作系统使用较长的时间片，是因为一般服务器上运行的程序比客户端要少很多，且更注重性能和吞吐量，而客户端系统更注重响应能力——而且，如果您真需要的话，时间片的长度也是可以调整的。

**注3：**不过，如果程序中多次复用单个NativeOverlapped对象的话，这个方法的性能会略微好于QueueUserWorkItem，据说WCF中便使用了这种方式——微软内部总有那么些技巧是我们不知如何使用的，例如老赵记得之前查看ASP.NET AJAX源代码的时候，在MSDN中不小心发现一个接口描述大意是“预留方法，请不要在外部使用”。对此，我们又能有什么办法呢？

[浅谈线程池（下）：相关试验及注意事项](http://www.cnblogs.com/JeffreyZhao/archive/2009/10/20/thread-pool-3-lab.html)

三个月，整整三个月了，我忽然发现我还有三个月前的一个小系列的文章没有结束，我还欠一个试验！线程池是.NET中的重要组件，几乎所有的异步功能依赖于线程池。之前我们讨论了线程池的作用、独立线程池的存在意义，以及对CLR线程池和IO线程池进行了一定说明。不过这些说明可能有些“抽象”，于是我们还是要通过试验来“验证”这些说明。此外，我认为针对某个“猜想”来设计一些试验进行验证是非常重要的能力，如果您这方面的能力略有不足的话，还是尽量加以锻炼并提高吧。

CLR线程的使用与创建

首先，我们准备这样一段代码：

public static void ThreadUseAndConstruction()

{

ThreadPool.SetMinThreads(5, 5); // set min thread to 5

ThreadPool.SetMaxThreads(12, 12); // set max thread to 12

Stopwatch watch = new Stopwatch();

watch.Start();

WaitCallback callback = index =>

{

Console.WriteLine(String.Format("{0}: Task {1} started", watch.Elapsed, index));

Thread.Sleep(10000);

Console.WriteLine(String.Format("{0}: Task {1} finished", watch.Elapsed, index));

};

for (int i = 0; i < 20; i++)

{

ThreadPool.QueueUserWorkItem(callback, i);

}

}

这段代码很简单。首先将线程池最小和最大线程数量设为5和12，然后向线程池中连续推入20个任务，每个任务都是打印出执行时的当前时间，然后等待10秒钟。那么请您思考一下，这段代码的输出是什么样的呢？

00:00:00.0028309: Task 0 started

00:00:00.0079552: Task 1 started

00:00:00.0080033: Task 2 started

00:00:00.0081628: Task 3 started

00:00:01.0058442: Task 4 started

00:00:01.5039911: Task 5 started

00:00:02.0048392: Task 6 started

00:00:02.5051786: Task 7 started

00:00:03.0051154: Task 8 started

00:00:04.0048998: Task 9 started

00:00:05.0053109: Task 10 started

00:00:06.0068503: Task 11 started

00:00:10.0079897: Task 1 finished

00:00:10.0084587: Task 12 started

00:00:10.0087316: Task 2 finished

00:00:10.0079939: Task 3 finished

00:00:10.0090849: Task 14 started

00:00:10.0088292: Task 0 finished

00:00:10.0101870: Task 15 started

00:00:10.0089327: Task 13 started

00:00:11.0059534: Task 4 finished

00:00:11.0063235: Task 16 started

00:00:11.5040658: Task 5 finished

00:00:11.5044820: Task 17 started

00:00:12.0051271: Task 6 finished

00:00:12.0064219: Task 18 started

00:00:12.5061198: Task 7 finished

00:00:12.5074655: Task 19 started

00:00:13.0052512: Task 8 finished

00:00:14.0052185: Task 9 finished

00:00:15.0064023: Task 10 finished

00:00:16.0074270: Task 11 finished

00:00:20.0085632: Task 12 finished

00:00:20.0094829: Task 14 finished

00:00:20.0104810: Task 13 finished

00:00:20.0119368: Task 15 finished

00:00:21.0066450: Task 16 finished

00:00:21.5045454: Task 17 finished

00:00:22.0116638: Task 18 finished

00:00:22.5107089: Task 19 finished

高位的零我们就直接忽略了，我们只观察“秒”及以下精度的时间。对这个数据进行简单观察之后，我们发现可以把时间精确到0.5秒来描述每个时刻所发生的事情：

1. 0秒：任务0至任务3，共计4个任务开始执行。
2. 1至3秒：任务4至任务8依次执行，间隔为0.5秒。
3. 3至6秒：任务8至任务11依次执行，间隔为1秒。
4. 10秒：任务0至任务3执行完成，任务12至任务15开始执行。
5. 11至12.5秒：每执行完一个旧任务（4至7)，便立即开始一个新任务（16至19）。
6. 13至22.5秒：剩余任务（8至19）依次结束。

您猜对了吗？我没有猜对，因为有两点：

* 原来最小线程数量为5时，只有4个线程可以立即执行。经过进一步尝试，最小线程数量为10时，也只有9个线程可以立即执行。
* 原来线程池创建线程的速度并非永远是“每秒2个”，而一些资料上写着“每秒不超过2个”的确是确切的说法。

但是，我们还是验证了以下几个结论：

* 在线程池最小线程数量的范围之内，尽可能多的任务立即执行。
* 线程池使用使用每秒不超过2个的频率创建线程（1秒一个或0.5秒一个）。
* 当达到线程池最大线程数时（第6秒），停止创建新线程。
* 在旧任务执行完毕后，新任务立即执行。

当然，由于我们在这之前已经“了解”了线程池是如何工作的，因此这里得到的结果可能会有“自圆其说”的倾向在里面。要减少这个可能性，则需要设计更完整的试验来“解释”问题。您也可以顺着这一点进行更深入的探索。

线程池中的线程是“公用”的

我们没有独立创建线程，而是选择使用线程池一定有其原因。不过，我们既然使用了线程池，就有一些额外的东西值得注意。

首先，我们要明确一个观念：线程并不“属于”任何一个任务，或者说任务并不“拥有”线程。我们只是借用一个线程来做事，用完以后便会还回。也就是说，任务在执行时修改线程的信息（名称，优先级，语言文化等等）是没有意义的，此外，任务也不应该依赖线程的这些状态。还记得上篇文章中谈到的QueueUserWorkItem和UnsafeQueueUserWorkItem之间的区别吗？如果您的任务需要依赖什么东西，也请自行准备。线程池中的线程状态是不可靠的。当然，也尽量不要直接对当前线程进行其他操作。

其次，由于线程池有大小限制，在某些时候还可能出现死锁的情况：

static void WaitCallback(object handle)

{

ManualResetEvent waitHandle = (ManualResetEvent)handle;

for (int i = 0; i < 10; i++)

{

ThreadPool.QueueUserWorkItem(state =>

{

int index = (int)state;

if (index == 9)

{

waitHandle.Set(); // release all

}

else

{

waitHandle.WaitOne(); // wait

}

}, i);

}

}

public static void DeadLock()

{

ManualResetEvent waitHandle = new ManualResetEvent(false);

ThreadPool.SetMaxThreads(5, 5);

ThreadPool.QueueUserWorkItem(WaitCallback, waitHandle);

waitHandle.WaitOne();

}

在上面的代码中，waitHandle将永远阻塞。因为我们放入线程池的10个任务，只有最后一个会将waitHandle打开，其余任务也统统阻塞在这个waitHandle上。但是请注意，我们使用SetMaxThreads方法把最大线程数限制为5，这样第10个任务根本无法执行，从而进入了死锁。避免这个问题最简单的做法是增加最大线程数，但是这还是会产生许多无法工作的线程，造成资源的浪费。因此，最好的做法是重新设计并行算法，并且时刻记住：“不要阻塞线程池里的线程”。

如何合理而有效的使用线程（既不多也不少还不阻塞），这是并行算法中最常见的课题之一。例如，让您设计一个并行计算[斐波那契数列](http://zh.wikipedia.org/wiki/%E6%96%90%E6%B3%A2%E9%82%A3%E5%A5%91%E6%95%B0%E5%88%97)的算法，如果您每次计算Fib(n)时，都创建两个新的任务来并行计算Fib(n - 1)和Fib(n - 2)，并等待它们结束，就会造成上述的死锁（或大量线程）。如何解决这个问题？您可以观察一下.NET 4.0中新增的Task并行类库，它提供了丰富而易用的并行运算API，帮我们省去了大量的工作1。

最后，便是时刻记得系统中哪些功能依赖线程池。例如ASP.NET中的请求也会使用CLR线程池，那么您是否应该使用ThreadPool？是否应该直接使用委托的异步调用？您是否应该调整线程池的最大和最小线数？这些问题没有确定答案，这需要您根据实际情况自己做判断。

CLR线程池与IO线程池

当第一次了解到.NET准备了一个CLR线程池和一个IO线程池的时后，我在想，这两者真的是没有关系的吗？他们会互相影响吗？于是我做了这么一个试验：

public static void IoThread()

{

ThreadPool.SetMinThreads(5, 3);

ThreadPool.SetMaxThreads(5, 3);

ManualResetEvent waitHandle = new ManualResetEvent(false);

Stopwatch watch = new Stopwatch();

watch.Start();

WebRequest request = HttpWebRequest.Create("http://www.cnblogs.com/");

request.BeginGetResponse(ar =>

{

var response = request.EndGetResponse(ar);

Console.WriteLine(watch.Elapsed + ": Response Get");

}, null);

for (int i = 0; i < 10; i++)

{

ThreadPool.QueueUserWorkItem(index =>

{

Console.WriteLine(String.Format("{0}: Task {1} started", watch.Elapsed, index));

waitHandle.WaitOne();

}, i);

}

waitHandle.WaitOne();

}

得到的结果是这样的：

00:00:00.0923543: Task 0 started

00:00:00.1152495: Task 2 started

00:00:00.1153073: Task 3 started

00:00:00.1152439: Task 1 started

00:00:01.0976629: Task 4 started

00:00:01.5235481: Response Get

从中可以看出，我们将CLR线程池的最大线程数量设为了5，并使用与上一例类似的做法故意“阻塞”了线程池（而只有5个任务被执行了，说明线程池的确被阻塞了），其目的便是观察在这种情况下一个IO异步请求是否能够得到正确的回复。答案是肯定的，IO异步请求的回调函数正常执行了。这意味着，虽然CLR线程池被用完了，但是似乎的确还是有一个额外的IO线程池在处理IO的异步回调。这样看来，CLR线程池和IO线程池两者并没有影响。此外，从.NET框架所设计的类库来看，的确将两者作了区分，例如：

public static class ThreadPool

{

public static bool GetAvailableThreads(out int workerThreads, out int completionPortThreads);

}

不过，这并不意味着CLR线程池中线程被用完之后，还是可以发起异步IO请求。例如，您可以尝试着将这个例子中的WebRequest操作放到for循环后面（确保CLR线程池中线程已经被用完了），这是您会发现BeginGetRequest方法的调用抛出了一个异常，提示您说线程池中没有多余的线程了。从这个角度这样看来，CLR线程池的确还是可能影响异步IO操作的（多谢xiongli大哥指出“这是由具体实现决定的”）——虽然这在普通应用程序中一般不会出现这个问题。

其实在IO线程池方面还可以进行其他一些试验。例如，您可以缩小IO线程池的最大线程数量，然后一下子发起多个异步IO请求，观察一下它们的回调函数执行时刻。这些不如就由您来自行完成了？

注1：.NET 4.0在多线程方面进行了明显的增强，除了Task并行类库之外，也将Parallel Library并入框架之内。此外，.NET 4.0还提供了许多线程安全的并行容器，以及轻量级的CountDownLatch、SemaphoreSlim、SpinWait等常用组件，无论是学习还是使用都是绝佳的范例。