最近，由于基础框架的整体升级，因此需要更新所有相关项目的DLL文件。这个过程存在不小的风险，因此也对发布后的生产服务器进行了密切的监控，结果还是出现了个别应用出现异常的情况，很快的占用了大量的服务器内存和CPU等资源。通过研究dump，初步发现是由于配置服务器出现单点故障，然后应用通过多线程调用相关SOA服务时出现异常，引发了ThreadAbortException异常，而且由于原有异常处理代码不够严谨，而且与异步发送报警邮件紧密结合在一起，造成线程数量的几何级增加，最终使得整个服务器不可用。这儿介绍的不算太清楚，而且相关原因虽然都有一定说服力，但证据不足，所幸最后通过重构，拿掉不需要的多线程操作，服务恢复正常。但不管如何，也确实要好好学习.NET CLR下的多线程相关知识。身边的一个资深架构师给我们的建议是，尽可能不要创建线程，如果确实需要一定要控制线程的数量，并且要可追溯。此外，如果是在IIS中托管的CLR，线程池的限制很多，而且是CLR中所有的appdomain共享，容易出现意料不到的错误，推荐使用.NET新的异步模型TPL。

在CLR一书中，将与线程有关的内容主要分成了5部分：线程相关基础知识；计算限制的异步操作；I/O限制的一步操作；基本线程同步变量；混合线程同步变量。本文虽然不会使用这个分类，但是这个分类对于相关概念在脑海建立一个有机的整体很有帮助。

进程（Process）是操作系统中的一个基本概念，它包含着一个运行程序所需要的全部资源。进程间相互独立，有自己的内存区域，可以认为是程序独立运行的基本单位。Windows在设计时，通过赋予每个进程独立的虚拟地址空间，确保一个进程不能访问另一个进程的代码，保证程序的健壮性。其使用时间片的方式处理进程（线程）对CPU的争用，Windows是一种抢占式(preempt)的多线程操作系统。

应用程序域（AppDomain）是一个Windows系统下的概念，是一个程序运行的逻辑区域，.NET的程序集正是在应用程序域中运行的，一个进程可以包含有多个应用程序域。

线程（Thread）是进程中的基本执行单元，在进程入口执行的第一个线程被视为这个进程的主线程。在.NET应用程序中，都是以Main()方法作为入口的，当调用此方法时系统就会自动创建一个主线程。线程主要是由CPU寄存器、调用栈和线程本地存储器（Thread Local Storage，TLS）组成的。CPU寄存器主要记录当前所执行线程的状态，调用栈主要用于维护线程所调用到的内存与数据，TLS主要用于存放线程的状态信息。线程可以看做是对CPU的虚拟化，线程主要包含5个要素：

1. 线程内核对象，该数据结构中包含一组对线程进行描述的属性以及线程上下文；
2. 线程环境块，包含线程异常处理head，线程进入的每个try块都在head插入一个节点，这也就是为什么ThreadAbortException这个特殊异常会在每个catch结尾处再次抛出的根源；
3. 用户模式栈，存储传给方法的局部变量和实参，默认分配的空间为1MB，最大的部分
4. 内核模式栈，当调用内核API时会使用
5. DLL线程连接和分离通知，windows每创建一个线程就会加载所有DLL中的入口方法，并传递一个dll\_thread\_attach的方法，当加载dll很多是，这个操作会造成很大的性能消耗。

此外，CLR在执行垃圾回收时，CLR必须挂起所有线程，并且遍历他们的栈来对堆中对象进行标记，因为大量线程对于垃圾回收的性能影响也非常的大，建立费资源，回收也费资源，因此需要非常慎重的考虑，当然多核情况下的并行计算确实非常的吸引人哈。Windows下的线程优先级有32级，但我们通常使用简化的5级优先级处理，实际默认都是Normal级别。

* System.Threading.Thread类

System.Threading.Thread是用于控制线程的基础类，通过Thread可以控制当前应用程序域中线程的创建、挂起、停止、销毁。

它包括以下常用公共属性：

|  |  |
| --- | --- |
| 属性 | 解释 |
| CurrentContext | 获取线程正在其中执行的当前上下文。 |
| CurrentThread | 获取当前正在运行的线程。 |
| ExecutionContext | 获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息。 |
| IsAlive | 获取一个值，该值指示当前线程的执行状态。 |
| IsBackground | 获取或设置一个值，该值指示某个线程是否为后台线程。 |
| IsThreadPoolThread | 获取一个值，该值指示线程是否属于托管线程池。 |
| ManagedThreadId | 获取当前托管线程的唯一标识符。 |
| Name | 获取或设置线程的名称。 |
| Priority | 获取或设置一个值，该值指示线程的调度优先级。 |
| ThreadState | 获取一个值，该值包含当前线程的状态。 |

一个应用程序域中可能包括多个上下文，而通过CurrentContext可以获取线程当前的上下文，CurrentThread是最常用的一个属性，它是用于获取当前运行的线程。

通过ThreadState可以检测线程是处于Unstarted、Sleeping、Running 等等状态，它比 IsAlive 属性能提供更多的特定信息，可以通过如下方式改变线程的状态：

1. 挂起线程：Sleep()和Suspend()，前者挂起指定的时间，后者在恢复前始终挂起，请谨慎使用Suspend和Resume的组合。因为一旦某个线程占用了已有的资源，再使用Suspend（）使线程长期处于挂起状态，当在其他线程调用这些资源的时候就会引起死锁！所以在没有必要的情况下应该避免使用这两个方法。此外，当你无法预知异步线程需要运行的时间，通过Thread.Sleep（int）阻塞主线程并不是一个好的解决方法，而应该使用thread.Join()，以保证主线程在异步线程thread运行结束后才会终止。
2. 终止线程：若想终止正在运行的线程，可以使用Abort()方法。在使用Abort()的时候，将引发一个特殊异常ThreadAbortException。若想在线程终止前恢复线程的执行，可以在捕获异常后 ,在catch（ThreadAbortException ex）{...} 中调用Thread.ResetAbort()取消终止。而使用Thread.Join()可以保证应用程序域等待异步线程结束后才终止运行。

* ThreadStart、ParameterizedThreadStart委托类。

通过ThreadStart来创建一个新线程是最直接的方法，但这样创建出来的线程比较难管理，如果创建过多的线程反而会让系统的性能下降（过多的线程上下问切换），因此需要谨慎使用。

CLR初始化时，线程池中是没有线程的，其内部维护了一个操作请求队列，应用程序想执行一个异步操作时，就调用某个方法，将一个记录项(entry)追加到线程池的队列中。线程池代码从这个队列提取记录项，并派遣给一个线程。如果木有线程则创建，在完成任务以后，该线程不会自行销毁，而是以挂起的状态返回到线程池。直到应用程序再次向线程池发出请求时，线程池里挂起的线程就会再度激活执行任务。这样既节省了建立线程所造成的性能损耗，也可以让多个任务反复重用同一线程，从而在应用程序生存期内节约大量开销。

线程池将自己的线程划分为工作者线程(Worker)和IO线程(CompletionPortThread)，前者主要用作管理CLR内部对象的运作，后者用于与外部系统交换信息，简单线程池方法如下：

|  |  |
| --- | --- |
| 方法 | 解释 |
| QueueUserWorkItem(WaitCallback callback, object state) | 向线程池队列添加一个工作项，参数1为回调委托，参数2为该委托的参数 |
| GetMaxThreads（out int workerThreads,out int completionPortThreads ) | 获取最大线程数 |
| SetMaxThreads( int workerThreads, int completionPortThreads) | 设置最大线程数 |

通过Get/SetMaxThreads两个方法可以分别读取和设置CLR线程池中工作者线程与I/O线程的最大线程数。在Framewok4.0中最大线程数默认为250\*CPU数，一般在1000左右，本机情况如下:



线程池使用需要注意：

通过CLR线程池所建立的线程总是默认为后台线程，优先级数为ThreadPriority.Normal。

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。

如果公共语言运行库是被承载的，例如被 IIS 或 SQL Server 承载，主机可能会限制或禁止更改线程池大小。

更改线程池中的最大线程数时需谨慎。虽然这类更改可能对您的代码有益，但对您使用的代码库可能会有不利的影响。

将线程池大小设置得太大可能导致性能问题。如果同时执行的线程太多，任务切换开销就成为影响性能的一个主要因素。

ThreadAbortException

在调用[Abort](https://msdn.microsoft.com/zh-cn/library/5b50fdsz.aspx)方法以销毁线程时，公共语言运行时将引发ThreadAbortException。ThreadAbortException是一种可捕获的特殊异常，但在**catch**块的结尾处它将自动被再次引发。引发此异常时，运行时将在结束线程前执行所有finally块。由于线程可以在finally块中执行未绑定计算或调用[Thread.ResetAbort](https://msdn.microsoft.com/zh-cn/library/system.threading.thread.resetabort.aspx)来取消中止，所以不能保证线程将完全结束。如果您希望一直等到被中止的线程结束，可以调用[Thread.Join](https://msdn.microsoft.com/zh-cn/library/95hbf2ta.aspx)方法。[Join](https://msdn.microsoft.com/zh-cn/library/95hbf2ta.aspx)是一个阻塞调用，它直到线程实际停止执行时才返回。

在错误的使用

* 执行上下文

每个线程都关联了一个执行上下文数据结构，该结构中包括有安全设置（Principal属性和windows身份）、宿主设置(HostExecutionContextManager)以及逻辑调用上下文数据(CallContext)的LogicalSetData和LogicGetData方法，我们可以通过设置使得线程的上下文内容不能流转，以减少资源的开销，接下来通过一个简单例子来理解。

|  |
| --- |
| public void Test()  {  CallContext.LogicalSetData("name", "xionger");  ThreadPool.QueueUserWorkItem(s => Console.WriteLine("name; {0}", CallContext.LogicalGetData("name")));  //阻止线程上下文的流动  ExecutionContext.SuppressFlow();  ThreadPool.QueueUserWorkItem(s => Console.WriteLine("name; {0}", CallContext.LogicalGetData("name")));  //恢复线程上下文的流动  ExecutionContext.RestoreFlow();  } |

* 完成端口模型（一个很老的Win32概念，可以无视）

之前可以看到I/O线程的名称叫CompletionPortThreads完成端口线程，这其实是Windows下的一种异步IO模型，其实可以把完成端口看成系统维护的一个队列，操作系统把重叠IO操作完成的事件通知放到该队列里，由于是暴露 “操作完成”的事件通知，所以命名为“完成端口”（Completion Ports）。一个socket被创建后，可以在任何时刻和一个完成端口联系起来。

一般来说，一个应用程序可以创建多个工作线程来处理完成端口上的通知事件。工作线程的数量依赖于程序的具体需要。但是在理想的情况下，应该对应一个CPU创建一个线程。因为在完成端口理想模型中，每个[线程](http://baike.baidu.com/view/1053.htm" \t "_blank)都可以从系统获得一个“原子”性的时间片，轮番运行并检查完成端口，线程的切换是额外的开销。在实际开发的时候，还要考虑这些线程是否牵涉到其他堵塞操作的情况。如果某线程进行堵塞操作，系统则将其挂起，让别的线程获得运行时间。因此，如果有这样的情况，可以多创建几个线程来尽量利用时间。

总之，开发一个可扩展的Winsock服务器并非十分困难的。主要是开始一个监听socket,接收连接，并且进行重叠发送和接收的IO操作。最大的挑战就是管理系统资源，限制重叠Io的数量，避免内存危机。遵循这几个原则，就能帮助你开发高性能，可扩展的服务程序。socket的接收缓冲，因为接收事件仅仅在[AcceptEx](http://baike.baidu.com/view/2860141.htm" \t "_blank)调用中发生。保证每个socket都有一个接收缓冲不会造成什么危害。一旦客户端/服务器在最初的一次请求（由AcceptEx完成）之后进行交互，发送更多的数据，那么取消接收缓冲更是一个很不好的做法。除非你能保证这些数据都是在每个连接的重叠IO接收里完成的 。

参考资料：

1. Jeffrey, Richter. CLR via C#[M]. 北京:清华大学出版社, 2010.
2. 风尘浪子. 细说多线程[EB/OL]. http://www.cnblogs.com/leslies2/archive/2012/02/07/2310495.html.