1. Why Does Windows Support Threads?

在计算机早期的岁月里，操作系统并没有提供线程的概念。事实上，整个系统只运行着一个线程。其中同时包含操作系统代码和应用程序代码。所以经常会出现“冻机”的情况，造成死循环。

1. 进程的出现。Microsoft 在决定改造16 位Windows系统的时候，他们决定在一个进程中运行应用程序的每一个实例。**进程不过是应用程序的一个实例要使用的资源的集合**。

每个进程都被赋予了一个虚拟地址空间。确保一个进程使用的代码和数据无法有另一个进程访问。这样就保证了程序实例的健壮性。业务一个进程无法使用或者是破坏另一个进程使用的代码或者数据。除此之外，OS的内核代码和数据是进程访问不到的。所以，应用程序代码破坏不了操作系统的代码或者数据。这样系统就变得非常的安全了。

1. 进程的存在的问题。如果一个进程中的程序出现死循环，并且不过假设只有一个CPU的情况下，那么他会一直消耗CPU，不能执行其他任何东西。所以虽然数据无法被破坏，但是仍然难逃死机的情况。
2. Microsoft 提出Thread来解决这个问题。
3. 线程开销。

|  |  |
| --- | --- |
| 要素 | 描述 |
| 线程内核对象(thread kernel object) | OS为系统中船舰的每一个线程都分配并且初始化这种数据结构之一。主要包含一组对线程进行描述的属性。该数据结构中包含对线程上下文的描述。上下文：一个内存块，包含了CPU的寄存器集合。 |
| 线程环境块（TEB） | TEB是在用户模式（应用程序代码能快速访问的地址空间）中分配和初始化的一个内存块。TEB包含线程的异常处理链首（head）.线程进入的每一个try块都在链首插入一个节点，线程退出try块的时候，会从链中删除该节点。除此之外，TEB还包含线程的“线程本地存储数据”，以及有GDI和OPENGL图形使用的一些数据结构。 |
| 用户模式栈(user-mode stack) | 用户模式栈用于存储方法的局部变量和实参，它还包含一个地址；指出当前方法返回时，线程应该从什么地方开始执行。 |
| 内核模式栈(kernel-mode stack) | 应用程序代码向操作系统中的一个内核模式的函数传入实参时候，还会使用内核模式栈；处于安全方面的原因，Windows会先把参数从线程用户模式栈复制到内核模式栈，内核才开始验证实参的值。由于应用程序不能访问内核模式栈，所以应用程序无法修改验证之后的实参值。OS对复制的值进行处理。 |
| DLL线程链接和线程分离通知 | Windows的一个策略是，任何时候在进程中创建一个线程都会调用那个进程中的所有DLL的DllMain方法，并向该方法传递一个Dll\_Thread\_Attach标志，类似，当进程终止，也不行调用所有DLL的Dllmain方法，传递一个Dll\_Thread\_Detach. |

1. **上下文切换**。单CPU的计算机一次只能做一件事情。所以Windows必须在系统中的所有线程(逻辑CPU)之间共享CPU,任何一个给定的时刻，Windows只将一个线程分配给一个CPU.那个线程运行运行一个“时间片（Quantum）”,一旦时间片到期，Windows就上下文切换到另一个线程。切换的过程如下：

Save将CPU寄存器的值 To Current Context

从现有的线程集合中Pick一个进行调度

将所选线程上下文中的值加载到CPU的寄存器中

Windows大约30毫秒切换一次上下文。这个都过程是净开销，不会带来任何内存或者性能上的收益，换来的就是一个**向用户提供一个健壮的、相应灵敏的操作系统**。

1. GC.垃圾回收时，CLR必须挂起（暂停）所有的线程，遍历他们的栈来查找根，一遍对堆中的对象进行标记，再次遍历他们的栈，在恢复所有的线程。所以，减少线程数量会显著提高垃圾回收器的性能。
2. 使用线程的理由：1、响应性。2、提高程序处理性能，因为可以做到并行处理。
3. 线程调度和优先级：Windows被称为“抢占式多线程操作系统（Preemptive multithreaded）”因为线程可能在任何时间被抢占。每个线程都分配了0-31级的一个优先级。Starvation:**只要存在可以调度的优先级31的线程，系统就将永远不会将优先级0-30的任何线程分配给CPU,这种情况称饥饿**。较高优先级的线程总是抢占较低优先级的线程，无论在运行的是什么较低优先级的线程。例如，如果有一个优先级为5的线程在运行，而系统确定有一个较高优先级的线程准备好运行，系统会立即挂起较低优先级的线程（即使后者的时间片还没有用完），将CPU分配给较高优先级的线程，该线程将获得一个完整的时间片。

进程优先级类：6个：Idle, Blow Normal, Normal, Above Normal, High , Real-time.如果一个应用什么都不做的时候就是和分配Idle.只有绝对必要的时候才应使用High, Real-time优先级应该尽可能避免，因为它甚至有可能干扰操作系统的任务，比如阻碍一些 必要的磁盘I/O和键盘鼠标输入。必须要有很充分的理由才能使用Real-time,比如延迟响应很短的硬件事件。

**“进程优先级类”只是一个抽象的概念，宗旨在于帮助理解自己的应用程序和其他正在应用程序的关系，没有别的用途。**

**托管应用程序不应该表现为拥有他们自己的进程；相反，他们应该表现为在一个AppDomain中运行。所以，托管程序不应该更改他们的进程优先级类，业务这样会影响进程中运行的所有代码；例如许多ASP.NET的应用程序都是在单个的进程中运行，每个应用程序都有它自己的AppDomain,类似的还有Silverlight应用程序，他在一个Internet浏览器进程中运行，还有托管存储过程，他在Microsoft SQL Server中运行。**

选好一个优先级类之后，就不用再思考你的应用程序和其他应用程序的关系了。我们再把注意力放到应哟个程序的线程上来。Windows支持7个相对线程优先级：Idle, Lowest ,Below Normal, Normal, Above Normal, Highest, Time-Critical.（CLR的线程终结器以Time-Critical优先级运行）

**大多数线程的优先级都是8.**

1. 前台线程、后台线程。一个进程中所有的前台线程停止运行时，CLR强制性停止仍在运行的后台线程，这些后台线程被终止，并且不会抛出任何异常。

前台线程用于执行确实想完成的任务，比如将数据从内存缓冲区flush到磁盘。

后台线程用于执行非关键性的任务，当程序终止时，就没有必要继续保持活动状态。

应用程序通过主线程以及通过构造一个Thread对象来显示创建的任何线程都是前台线程。

线程池线程默认都是后台线程，此外有托管执行环境的Native代码创建的任何线程都被标识为后台线程。

1. CLR 线程池。由于创建和销毁线程是一个非常昂贵的操作，要消耗大量的时间，另外，大多数线程都会消耗内存资源，由于操作系统必须调度可运行的线程并执行上下文切换，所以太多线程会有损性能。每个CLR有一个线程池，这个线程池由CLR控制的所有APP Domain共享。如果一个进程加载了多个CLR，那么每个CLR都有它自己的线程池。

工作项（workItem）（含有异步操作的方法）

将记录项追加到线程池队列

线程池从队列中取出记录项

Dispatch

线程池中的线程，如果没有就创建一个

当线程池线程完成任务之后，他不会销毁，而是回到线程池，并进入空闲状态，等待另一个请求，由于不会销毁自身，所以不会产生相应的性能损失。（当线程池中的线程一直闲着没有事情做的时候，线程会自己醒过来终止自己，并且释放资源）。

**线程池线程主要分两类：Worker线程和I/O线程。**

1. Why Task ?调用ThreadPool的QueueUserWorkItem方法来发起一次异步、受计算限制的操作是非常简单的，然而，这个技术存在很多限制，最大的问题是，没有一个内建的机制让你知道操作什么时候完成，也没有一个机制在操作完成的时候获取一个返回值。为了解决这个问题Micorosoft引入了**任务（Task）**这个概念。**System.Threading.Tasks.**

**ThreadPool.QueueUserWorkItem(ComputeBoundOp,5);**

**New Task(ComputeBoundOp,5).Start();**

1. 等待任务获取任务结果。Task.Result; Task.Wait,不一定会阻塞线程，因为有两种可能，如果任务已经开始，那么就会阻塞；但是如果没有开始，它会立即开始执行任务并立即返回，不阻塞线程。
2. ContinueWith:调用ContinueWith的时候可以传递一个组**TaskContinuationOptions；默认情况下，一个任务船舰的Task对象就是顶级任务，这些任务与创建他们的那个任务无关。然而，如果使用TaskContinuationOptions.AttachedToParent** 标志将一个Task和创建它的Task关联起来，结果会是的除非所有的子任务都结束了，否则，父任务是不会任务自己已经结束了的。
3. 任务揭秘：

|  |
| --- |
| Task的只读ID属性 |
| 代表Task执行状态的一个Int32 |
| 对父任务的一个引用 |
| 对Task创建的时制定的TaskScheduler的一个引用 |
| 对回调方法的一个引用 |
| 对要传回给回调方法的对象的一个引用，可以通过Task.AsyncState属性查询 |
| 对一个ExecutionContext的一个引用 |
| 对ManualResetEventSlim对象的引用 |
| 对根据需要创建的一些补充状态的一个引用（包含CacellationToken,一个ContinueWithTask的对象集合、为抛出为处理的异常的子任务而准备的一个Task的对象集合） |

如果不需要任务提供的附加功能，其实ThreadPool.QueueUserWorkItem效率会更高。

**Task.Status**

Created

WaitingToRun

WaitingForChildrenToComplete

Canceled

Faulted

RunToCompleted

IsCompleted

var task=new Task

task.Start()

1. TaskFactory:创建一组参数统一的Task.

ContinueWhenAll;ContinueWhenAny

1. **TaskScheduler:任务调度器:**主要分两类，线程池任务调度器（Thread Pool Task Scheduler）、同步上下文任务调度器（Synchronization context task scheduler）.默认情况下都使用线程池任务调度器。同步UI线程的时候会使用后者。
2. 应用程序以及线程处理模型

.Net 支持几种不同的应用程序模型，每个应用程序模型都可以引入它自己的线程处理模型。每个应用程序都可能引入它自己的线程处理模型。控制台应用程序和Windows服务没有引入任何类的县城处理模型；也就是说任何线程可以在它喜欢的任何时候做她爱做的任何事情。然而GUI应用程序如Win Form，WPF, Silverlight都引入了一个线程处理模型。在这个模型中，创建窗口的线程是唯一能够对那个窗口进行更新的线程。在GUI线程中，经常需要生成一个异步操作，使GUI线程不至于阻塞并停止用户相应输入，然而异步操作完成时候，他是用一个线程池线程完成的，而线程池线程不能负责显示结构的UI,因此，线程池线程必须采用某种方式让GUI线程更新UI.

类似于控制台应用程序，ASP.NET Web窗体和XML Web服务应用程序运行任何线程做它爱做的事情。线程池线程开始处理一个有客户的请求时，可以对客户端语言文化进行做出假定，从而允许Web服务器对返回的数字，日期和时间进行语言文化特定的格式处理。除次之外，Web服务器还可以对客户端的身份表示做出假定，是服务器只能访问这个客户端有权访问的资源。当线程池线程生成一个异步操作的时候，，这个异步操作将由另外一个线程池线程完成，该线程池线程处理异步操作的结果；虽然这个一部操作代表原来的客户端而执行的，但是语言文化和身份标志信息不会流向新的线程池。

SynchornizationContext的派生类可以解决这个问题。因为一个入栈的客户但请求开始运行的线程池线程关联了一个SyncronizationContext派生对象，这个对象包含了客户端文化和身份标志信息，为了获取这个对象的一个引用，可以将线程池线程查询SynchornizationContext的静态Current属性。然后对这个对象的引用保存到负责处理客户端的请求的一个成员字段中。另外一个线程池线程回调类中的一个方法是，可以让线程池线程引用保存的SynchornizationContext派生对象。并调用Post方法，传递要用客户但的语言文化和表标志信息调用方法。

1. Primitive Constructs也就是基元构造，最简单的构造。

|  |  |  |
| --- | --- | --- |
|  | 优点 | 缺点 |
| 用户模式构造  User-mode constructs | 更快速，使用特殊CPU指令协调线程。也就是说直接发生在硬件层面，同时也说明，Windows OS用户模式从来检测不到线程是阻塞的，这样，线程池就不会创建新的线程去替代暂时阻塞的线程。 | 独占CPU，一直不释放，会导致别的想使用资源的线程没有办法获取资源。 |
| 内核模式构造  Kernel-mode constructs | 由Windows OS自身提供，需要你的程序的线程调用操作系统内核实现的函数;当以线程使用内核模式去获取另一个线程锁拥有的资源，Windows会阻塞这个线程，以至于不会浪费CPU时间。Windows阻塞当前线程。Windows阻塞线程，不浪费CPU时间 | 线程从用户模式转换到内核模式需要消耗较大性能。 |

如果一个线程占据一个**构造**一直不释放，这样的情况下会使得等待该**构造**的线程一直Block住，如果该构造是一个用户模型的构造，那么占据构造的线程会一直使用CPU，我们称这种情况为**“活锁”**，类似于“**While (true)**”的效果。如果这个构造是一个内核模式构造，那么该线程将会一直block，这种情况我们称之为“**死锁”。**死锁相比来说要比活锁好，因为活锁即浪费CPU时间，又浪费内存。然而死锁只浪费内存。

用户模式构造：

CLR保证在读取Boolean, Char, (S)Byte, (U)Int16, (U)Int32, (U)IntPtr, Single以及引用类型都是原子性的。举个反面的例子，比如对于Int64，

internal static class SomeType {

public static Int64 x = 0;

}

如果有一个线程写出下面的Code:

SomeType.x = 0x0123456789abcdef;

可能会出现在另外一个线程里读出0x0123456700000000 或者是0x0000000089abcdef；这种成为：Torn Read.

1. Hybrid Constructs也就是混合构造，顾名思义，也就是综合上面两种基元构造的一种实现。
2. Volatile.Write 保证所有在他之前的代码必须完成之后再执行，当然它之前的代码段本身是可以再优化的，也就是说可能是无序的。Volatile.Read保证所有的变量在这之后的必须在它之后执行。但是这个两个方法对于程序来说使用起来有些麻烦，所以，C#提供了Volatile，可以适应于：Boolean, (S)Byte, (U)Int16, (U)Int32, (U)IntPtr, Single, or Char. reference types .
3. 内核模式构造。WaitHandle抽象基类。WaitHandle类是一个非常简单的类，它唯一包装一个Windows内核对象句柄。FCL提供了几个WaitHandle的派生类。

EventWaitHandle

Mutex

Semaphore

ManualResetEvent

AutoResetEvent

WaitHandle

WaitHandler基类内部有一个SafeWaitHandler字段，它包含了Win32内核对象句柄。

Demo中，在没有锁竞争的状态，SimpleWaitLock的Enter和Leave每一个调用都会抢播调用线程从托管代码转换为内核代码，在转换回来，这是它的弊端，然而，如果在有锁竞争的状态下，输掉的线程不会被内核组设，不会在那里“自旋”，然非CPU的时间。构造AutoResetEvent对象在Disopose时候，也会造成托管向内核转换，从而带来性能上的负面影响。