# CLR 线程概览 - 文章

托管 vs. 原生线程

托管代码在“托管线程”上执行，（托管线程）与操作系统提供的原生线程不同。原生线程是在物理机器上执行的原生代码序列；而托管线程则是在CLR虚拟机上执行的虚拟线程。

正如JIT解释器将“虚拟的”中间（IL）指令映射到物理机器上的原声指令，CLR线程基础架构将“虚拟的”托管线程映射到操作系统的原生线程上。

在任意时刻，一个托管线程可能会也可能不会被分配到一个原生线程执行。例如，一个已经被创建（通过“new System.Threading.Thread”）但是未启动（通过“System.Threading.Thread.Start”）的托管线程不会被指派到原生线程上执行。类似的，虽然CLR在实际上不会这样做，但是一个托管线程在执行时可被切换到多个原生线程上执行。

托管代码里公开的Thread接口就是用来隐藏其底层原生线程的细节的：

- 托管线程无需绑定到一个原生线程上（甚至有可能根本不映射到原生线程上）。
- 不同操作系统的原生线程不一样。
- 原则上，托管线程是“虚拟的”。

CLR提供并实现了托管线程的抽象。比如说，虽然其不暴露操作系统的线程本地存储（TLS）机制，但是其提供了托管“线程静态”变量。类似的，虽然其不提供原生线程的“线程ID”，但是其提供与操作系统无关的“托管线程ID”。不过为了便于诊断问题，底层原生线程的一些细节可以通过System.Diagnostics命名空间里的类型获得。

托管线程还提供了原生线程通常不用的功能。第一，托管线程在堆栈上使用GC引用，这样CLR必须在GC的时候可以枚举（甚至可能修改）这些GC引用。为了实现这个目的，CLR必须“暂停”每个托管线程（即停止执行以便可以发现所有的GC引用）。第二，当AppDomain卸载时，CLR必须保证没有线程在执行这个AppDomain里的代码。这要求CLR可以强制线程从AppDomain脱离，CLR通过在线程里注入ThreadAbortException来实现这点。

## 数据结构

每个托管线程都跟一个Thread对象关联，其在threads.h里定义。这个对象跟踪CLR关于托管对象所需要了解的所有东西。包括如线程的当前GC模式和堆栈帧链这些必需品，也包括为了性能因素创建的很多元素（如一些快速arena-style分配器）。

所有的Thread对象都保存在ThreadStore中（也在threads.h中定义），其时一个所有已知线程的列表。要遍历所有的托管线程，需要先获取ThreadStoreLock，再使用ThreadStore::GetAllThreadList来枚举

所有的线程对象。这个列表也包含没有被指派原生线程的托管线程（如未启动的线程，或原生线程已经存在了）。

原生线程可以通过一个原生线程本地存储（TLS）槽来获取绑定到该原生线程的托管线程。这允许原生线程上运行的代码可以通过GetThread()获取对应的Thread对象。

另外，许多托管线程有一个与原生Thread对象相区别的 托管 Thread对象（System.Threading.Thread）。托管Thread对象提供了方法以便托管代码与线程交互，其大部分是原生Thread对象功能的封装。通过Thread.CurrentThread可以（在托管代码中）获取到当前的托管线程对象。

在调试器里，“!Threads”这个SOS扩展命令可以用来枚举ThreadStore里的所有Thread对象。

## 线程的生命周期

一个托管线程在下列这些情形中创建：

1. 托管代码通过System.Threading.Thread显式要求CLR创建一个新线程。
2. CLR自己创建的托管线程（参见“特殊线程”一节）。
3. 原生代码在原生线程上调用托管代码，而这个托管代码没有跟托管线程相关联（通过“反向p/invoke”或者COM互交互）。
4. 一个托管进程被启动了（在进程的主线程上调用其Main函数）。

在#1和#2这些情形中，CLR负责创建支撑托管线程的原生线程。这个只会在线程实际上启动了才会发生。在这些情形里，CLR“负责”原生线程；CLR负责原生线程的生命周期，由于CLR创建了它，因此也就知道线程的存在。

在#3和#4这些情形里，原生线程在托管线程之前就存在了，而且由CLR之外的代码负责。CLR不负责这种原生线程的生命周期。CLR只是在其第一次调用托管代码时意识到其存在。

当一个原生线程结束时，CLR通过其DllMain函数获得通知。这在操作系统的“加载锁”中发生，所以在处理这个通知的时候只能做很少（安全）的事情。与其销毁与托管线程关联的数据结构，这个线程只是被简单地标识成“死亡”状态，并启动finalizer线程。finalizer线程会遍历ThreadStore里所有死亡且托管代码不再使用的线程。

## 暂停

CLR必须可以找到托管对象的所有引用以便执行GC。托管代码一直在不停的访问GC堆，操作堆栈和寄存器上的引用。CLR必须保证所有线程停在安全可靠的位置（这样他们不会修改GC堆），以便找到所有的托管对象。它只会停在安全点，这个时候可以在寄存器和堆栈上检查所有可用的引用。

另一个办法就是GC堆、每个线程的堆栈和寄存器状态都是所谓的“共享状态”，可被多个线程访问。正如大多数共享状态一样，需要一些“锁”来保护它们。托管代码在访问堆之前必须要获取锁，并且在安全的时候释放锁。

CLR将这种“锁”称作线程的“GC模式”。当线程获取锁的时候，处于“合作模式（cooperative mode）”；其必须与GC“合作”（通过释放锁）才能允许进行垃圾回收。而线程没有获取锁的时候，处于“优先模式（preemptive mode）” — GC可以“优先”进行垃圾回收，因为其知道线程没有访问GC堆。

GC只有在所有线程都处于“优先”模式（即没有获取锁）时才能进行垃圾回收。将所有线程移到优先模式的过程就称为“GC悬停（GC suspension）”或“暂停执行引擎”。

一个不大成熟的实现“锁”的方案是要求每个托管线程在访问GC堆的时候实际获取和释放保护它的锁。然后GC会向每个线程尝试获取锁，一旦其获取所有线程的锁，就可以安全的进行垃圾回收了。

然而，上面的方案因为两个原因而显得不足。第一，这会要求托管代码耗费大量的时间在于获取和释放锁（或至少是检查GC是否在尝试获取锁 — 也就是“GC轮询 GC poll — 即不停的向GC轮询”）。第二，它要求JIT解释器生成大量的“GC信息代码”，以描述每一行JIT生成的代码后的堆栈的布局和寄存器状态，这些信息会耗费大量的内存。

我们针对上述办法的改进方案是，将JIT后的托管代码区分成“部分可中断”和“全部可中断”的代码。在部分可中断代码中，调用其他函数的地方是唯一的安全点，且JIT生成显式的“GC轮询”点以便检查是否有等待的GC。（JIT）只需要在这些地方生成GC信息。在全部可中断代码里，每个指令都是一个安全点，JIT为每个指令生成GC信息 — 但是其不生成“GC”轮询代码。全部可中断代码而是通过劫持线程（该过程在后文讲解）来进入“中断”状态。JIT基于代码质量，GC信息的大小以及GC悬停的时间延迟这些因素来判定是产生全部或部分可中断代码。

基于上述信息，定义了三个基础操作：进入合作模式，离开合作模式以及暂停执行引擎。

# 进入合作模式

一个线程通过调用Thread::DisablePreemptiveGC进入合作模式。其为当前线程获取“锁”：

1. 如果有GC正在执行（GC拥有这个锁），那么等待GC完成。
2. 标识这个线程将进入合作模式，在这个线程进入“优先模式”之前不能触发GC。

两个步骤实际上是原子操作。

# 进入优先模式

一个线程通过调用Thread::EnablePreemptiveGC来进入优先模式（释放锁）。其通过标识线程不再进入合作模式来完成，并通知GC线程可以启动执行。

# 中断执行引擎

当GC开始运行时，第一步就是中断执行引擎。GCHeap::SuspendEE函数就是用来干这个的：

1. 设置一个全局变量(g_fTrapReturningThreads)来标志GC正在执行，任何想进入合作模式的线程都会

被阻止，直到GC运行完毕。
2. 找出所有处于合作模式的线程，针对每个这样的线程，试图劫持线程并强制其离开合作模式。
3. 重复前面的步骤直到没有线程处于合作模式。

## 劫持

为了GC悬停而进行的劫持操作是通过Thread::SysSuspendForGC函数完成的。这个函数通过强制所有运行在合作模式的托管线程在“安全点”离开合作模式。其通过枚举所有的托管线程（通过遍历ThreadStore），针对每个运行在合作模式中的托管线程：

1. 通过Win32的SuspendThread API来暂停底层的原生线程。这个API强制线程从运行状态停止在任意位置（不一定是一个安全点）。
2. 通过GetThreadContext获取线程的上下文（CONTEXT）。这是一个操作系统的概念；上下文存放了线程的当前寄存器状态。这就允许我们来监视其指令寄存器，并获知正在运行的指令类型。
3. 再次检查线程是否在合作模式，因为其可能在被暂停之前已经离开合作模式了。如果是这样的话，那么线程处于危险地段：线程可能在运行任意的原生代码，必须立即恢复执行以规避死锁。
4. 检查线程是否在运行托管代码。其有可能在合作模式下运行虚拟机（VM）自身的原生代码（参看下面的同步章节），其也需要跟上一步一样立即恢复执行。
5. 那么线程目前是暂停在托管代码上。取决于代码是全部还是部分可中断，采取下面的措施之一：
   - 如果是全部可中断，那么在任意位置GC都是安全的，因为线程按照全部可中断的定义就是在安全点。理论上可以让线程停在这个位置（因为是安全的），但是几个历史性的操作系统Bug妨碍了这点，因为前面获取的线程上下文也许已经损坏了）。于是（CLR）改写线程的指令寄存器，引导线程跳转到一个代码块以便获取更完整的上下文，离开合作模式，等待GC运行完毕，重新进入合作模式，并且还原线程的寄存器。
   - 如果是部分可中断，那么线程按照定义不在一个安全点。但是，其调用者是处于安全点的（函数间切换）。基于这个知识，CLR在堆栈帧上“劫持”起返回地址（即修改堆栈），引导线程跳转到跟“全部可中断”类似的代码块。当函数返回时，其不是返回原来的调用函数那里，而是这个代码块（这个函数可能也会执行JIT在之前注入的GC轮询，导致线程离开合作模式并撤销劫持操作）。

## ThreadAbort / AppDomain-Unload

为了卸载一个应用程序域（AppDomain），CLR需要保证没有线程运行在这个应用程序域中。为了实现这点，所有托管线程都被枚举，而任何堆栈上有属于被卸载应用程序域的帧的线程都被“中断”。一个ThreadAbortException异常被注入正在运行的线程，并导致线程向上展开（一直运行拆除代码）直到没有运行在这个应用程序域当中的堆栈帧，而ThreadAbortException也被转换成一个AppDomainUnloaded异常。

ThreadAbortException是一个很特别的异常。其也许会被用户代码捕捉到，但是CLR确保其在用户的异常处理代码之后再次被抛出。因此ThreadAbortException有时被称作“无法被捕捉”的，尽管严格来说不是这样的。

ThreadAbortException通常通过在托管线程上设置一个标志位标志其“正在终止”来抛出的。CLR很多地方都会检查这个标志位（特别要注意的，每次从p/invoke返回），并且经常有设置这个标志位的目的就

是为了让线程及时终止的情形。

然而，比如说，线程正在运行一个长时间的托管循环，那么它可能根本不会检查这个标志位。为了让这样的线程快速终止，线程就被“劫持”并强制抛出ThreadAbortException异常。劫持过程跟GC悬停很类似，只是线程跳转过去的代码块抛出ThreadAbortException，而不是等待GC运行完毕。

这种劫持意味着ThreadAbortException可能在任意位置发生。这样使得托管代码很难正确处理ThreadAbortException异常。因此除了在卸载应用程序域的时候使用这种机制以外 — 保证由ThreadAbort损坏的状态都跟应用程序域一起被清理，在其他地方使用它都不是很明智的选择。

## 同步：托管代码

托管代码可以访问很多在System.Threading里定义的同步原语。包括操作系统原语的简单封装如：互斥（Mutex），事件（Event）和旗标（Semaphore）对象，也包括类似的栅栏（Barrier）和自旋锁（SpinLock）等抽象。但托管代码用的最多的同步机制是System.Threading.Monitor，其提供了针对 任意托管对象 的高性能同步锁机制，还提供了被其保护的状态发生变化时的通知机制的“条件变量”语义。

Monitor是通过一个“混合锁”来实现的，其有自旋锁和类似互斥（Mutex）这些基于操作系统内核锁的功能。这个思路源自于大部分锁都是短暂获取的，因此自旋等待锁被释放的所耗费的时间比调用内核API从而阻塞线程更少。当然将CPU的时钟周期浪费在自旋上也是很严重的，因此如果锁在一段时间内没有被释放的话，那么CLR则会退回到调用内核API的实现上。

因为任意一个对象都是潜在的锁 / 条件变量，每个对象都需要有一个地方用来保存锁信息。这个就是在“对象头（object headers）”和“同步块（sync blocks）”里完成的。

对象头是一个在每个托管对象前面机器字长大小的字段。它在很多地方会用到，例如保存对象的哈希值。其中一个目的就是保存对象的锁状态。如果对象头需要保存更多的信息，我们通过创建一个“同步块”的方式扩充对象。

同步块保存在同步块表（Sync Block Table）里，通过同步块索引来寻址。对象的同步块索引保存在对象头里。

关于对象头和同步块的细节在syncblk.h/.cpp里定义。

如果对象头里还有空间，Monitor将锁住对象的线程的托管线程ID（如果没有线程锁住对象则是0）保存在其中。在这种情形下，获取锁的过程其实就是自旋等待对象头的线程ID为0，然后原子操作设置其值为当前线程的托管线程ID。

如果自旋一些次数后还不能获取锁，或对象头已经用作其它目的，那么就会为这个对象创建同步块。它包含一些额外数据，包括用来阻塞当前线程的事件对象，这样运行我们停止自旋并等待锁被释放。

一个用来作为条件变量的对象（通过Monitor.Wait 和 Monitor.Pulse）总是会被扩充的，因为同步块里已经没有足够的空间来保存必要的状态。

# 同步：原生情况

CLR的原生部分也必须要有线程意识，因为其可能在多个线程上调用托管代码。这样要求原生的同步机制，例如锁，事件等等。

ITaskHost API 允许一个CLR宿主修改托管线程的很多方面，包括线程的创建、销毁和同步。这种允许宿主修改原生同步机制要求虚拟机的代码不能直接使用原生的同步原语（即临界区，互斥锁，事件等），而是需要使用虚拟机在其上的封装）。

除了上述细节之外，GC悬停是一个特殊的“锁”，而且几乎影响CLR的方方面面。如果必须处理GC堆上的对象，虚拟机的原生代码可能要进入“合作”模式，这样“GC悬停锁”就变成原生虚拟机代码里最重要的同步机制，在托管世界里也一样。

原生虚拟机代码里主要用到的同步机制是GC模式和Crst。

# GC 模式

如上所述，所有托管线程都在合作模式中运行，因为其可能操作GC堆。一般来讲，原生代码不会碰托管对象，因此运行在优先模式。但有些虚拟机里的原生代码需要访问GC堆，需要运行在合作模式。

原生代码通常不会直接操作GC模式，而是通过两个宏：GCX_COOP and GCX_PREEMP 来进入期望的模式，并创建“支持物”以便线程在退出范围的时候返回到之前的模式。

需要注意的是GCX_COOP从GC堆上获取一个锁。在线程处于合作模式时，不能执行GC。而且原生线程也不能像托管线程那样被“劫持”，因此线程在切换回优先模式时都是处于合作模式。

因此在原生代码里进入合作模式是不被鼓励的。如果必须要进入合作模式，那么时间越短越好。线程在此模式时不能被阻塞，而且实际上不能安全的获取锁。

类似的，GCX_PREEMP 释放 线程拥有的锁。在进入优先模式之前必须要万分小心来确保所有GC引用都被妥善保护。

代码规范 文档描述了安全进行GC模式切换的必要原则。

# Crst

正如Monitor对象是托管代码里推荐的锁机制，Crst是虚拟机代码里的推荐机制。与Monitor类似，Crst是一个知道宿主和GC模式的混合锁。Crst通过“层级锁”机制来规避死锁，该实现可参考 BotR的层级锁章节.

虽然有一些必须这么做的异常情况，在合作模式下获取一个Crst锁通常是不合适的。

# 特殊线程

除了托管代码创建的托管线程，CLR自身还创建了一些“特殊”线程。

## 终结者（Finalizer）线程

每个进程都创建了这个线程用来运行托管代码。当GC决定一个可终结（finalizable）的对象不再被引用，其将该对象置于终结队列。当GC结束后，终结者线程会被唤醒并处理队列里的所有终结对象。对象一个一个出列，其终结（finalizer）函数被依次调用。

该线程还用来处理一些CLR内部的清理工作，并等待一些外部事件通知（如低内存情形下，GC会被告知尽量凶悍的回收垃圾）。详情请参见GCHeap::FinalizerThreadStart。

## GC 线程

当运行在“并行”或“服务器”模式时，GC创建一个或多个后台线程来并行执行垃圾回收的不同阶段。这些线程完成由GC管理，而且永远不会执行托管代码。

## 调试器线程

CLR为每个托管进程维护了一个原生线程，其用来在附加到托管调试器时执行多个调试操作。

## 应用程序域卸载线程

这个线程负责卸载应用程序域。其通过一个单独的CLR内部线程，而不是在请求卸载应用程序域的线程里完成。因为 a）为卸载过程提供受保证的堆栈空间，b）在必要时允许请求卸载的线程从应用程序域里向上展开。

## 线程池线程

CLR线程池维护一个托管线程集合用来执行用户的“工作”。这些托管线程都绑定到线程池管理的原生线程。线程池还维护一小部分的原生线程来处理类似“线程注入”，定时器以及“已注册的等待”等等功能。

加入伯乐在线专栏作者。扩大知名度，还能得赞赏！详见《招募专栏作者》
2 赞 1 收藏 评论
合作联系
Email：bd@Jobbole.com
QQ： 2302462408 （加好友请注明来意）

更多频道

小组 - 好的话题、有启发的回复、值得信赖的圈子
头条 - 分享和发现有价值的内容与观点
相亲 - 为IT单身男女服务的征婚传播平台

- 资源 - 优秀的工具资源导航
- 翻译 - 翻译传播优秀的外文文章
- 文章 - 国内外的精选文章
- 设计 - UI,网页，交互和用户体验
- iOS - 专注iOS技术分享
- 安卓 - 专注Android技术分享
- 前端 - JavaScript, HTML5, CSS
- Java - 专注Java技术分享
- Python - 专注Python技术分享