

明潮的BLOG

崖下潮起落，捣碎亦开花！

RSS订阅

转 并发危险：解决多线程代码中的 11 个常见的问题

2018年05月09日 10:49:10

阅读数：141

本文将介绍以下内容：

- 基本并发概念
- 并发问题和抑制措施
- 实现安全性的模式
- 横切概念

目录

数据争用
忘记同步
粒度错误
读写撕裂
无锁定重新排序
重新进入
死锁
锁保护
截记
两步舞曲
优先级反转
实现安全性的模式
不变性
纯度
隔离

简要分析：

- Forgotten Synchronization (没有同步)
- Incorrect Granularity (不正确的锁粒度)
- Read and Write Tearing (逻辑上我们看上去读写一次内存，但是底层可能因为不对齐/大小原因读写两次)
- Lock-Free Reordering (里面还提到了recursive lock以及死锁问题)
- Lock Convoys (请求锁的速率高于critical section的处理速率，导致很多请求hang住。解决办法就是减少section大小和提高处理速度。这个问题完全是系统设计处理模型造成的，不过确实会造成性能问题)
- Two-Step Dance(发生的情况大致是如果A占有锁lock,但是触发事件造成B被唤醒，而B也需要lock但是没有办法占有，接着又切换回来，造成不必要的cs.不过我倒是觉得这个问题不是很常见的)
- Priority Inversion (这个完全是因为线程优先级造成的。低优先级占有lock，但是高优先级不断地被调度，但是因为lock问题没有成功，同样造成不必要的cs.感觉和上面问题倒是一样，只不过触发条件不同。个人觉得这个问题完全可以在scheduler上解决的)
- Patterns for Achieving Safety (后面三点都是关于获得thread safety的建议)
 - Immutability.数据不可变性
 - Purity.方法没有side-effect
 - Isolation.每个线程只有自己独立的数据.

并发现象无处不在。服务器端程序长久以来都必须负责处理基本并发编程模型，而随着多核处理器的日益普及，客户端程序也将需要执行一些任务。随着并发操作的不断增加，有关确保安全的问题也浮现出来。也就是说，在面对大量逻辑并发操作和不断变化的物理硬件并行性程度时，程序必须继续保持同样级别的稳定性和可靠性。

与对应的顺序代码相比，正确设计的并发代码还必须遵循一些额外的规则。对内存的读写以及对共享资源的访问必须使用同步机制进行管制，以防发生冲突。另外，通常有必要对线程进行协调以协同完成某项工作。

这些附加要求所产生的直接结果是，可以从根本上确保线程始终保持一致并且保证其顺利向前推进。同步和协调对时间的依赖性很强，这就导致了它们具有不确定性，难于进行预测和测试。

这些属性之所以让人觉得有些困难，只是因为人们的思路还未转变过来。没有可供学习的专门 API，也没有可进行复制和粘贴的代码段。实际上的确有一组基础概念需要您学习和适应。很可能随着时间的推移某些语言和库会隐藏一些概念，但如果您现在就开始了并发操作，则不会遇到这种情况。本文将介绍需要注意的一些较为常见的挑战，并针对您在软件中如何运用它们给出一些建议。

首先我将讨论在并发程序中经常会出错的一类问题。我把它们称为“安全隐患”，因为它们很容易发现并且后果通常比较严重。这些危险会导致您的程序因崩溃或内存问题而中断。

当从多个线程并发访问数据时会发生数据争用（或竞争条件）。特别是，在一个或多个线程写入一段数据的同时，如果有一个或多个线程也在读取这段数据，则会发生这种情况。之所以会出现这种问题，是因为 Windows 程序（如 C++ 和 Microsoft .NET Framework 之类的程序）基本上都基于共享内存概念，进程中的所有线程均可访问驻留在同一虚拟地址空间中的数据。静态变量和堆分配可用于共享。

请考虑下面这个典型的例子：

```
static class Counter {  
    internal static int s_curr = 0;  
    internal static int GetNext() {  
        return s_curr++;  
    }  
}
```



0



写评论



目录



收藏



微信



微博



QQ

```
    }  
}
```

Counter 的目标可能是想为 GetNext 的每个调用分发一个新的唯一数字。但是，如果程序中的两个线程同时调用 GetNext，则这两个线程可能被赋予相同的数字。原因是 s_curr++ 编译包括三个独立的步骤：

1. 将当前值从共享的 s_curr 变量读入处理器寄存器。
2. 递增该寄存器。
3. 将寄存器值重新写入共享 s_curr 变量。

按照这种顺序执行的两个线程可能会在本地从 s_curr 读取了相同的值（比如 42）并将其递增到某个值（比如 43），然后发布相同的结果值。这样一来，GetNext 将为这两个线程返回相同的数字，导致算法中断。虽然简单语句 s_curr++ 看似不可分割，但实际却并非如此。

忘记同步

这是最简单的一种数据争用情况：同步被完全遗忘。这种争用很少有良性的情况，也就是说虽然它们是正确的，但大部分都是因为这种正确性的根基存在问题。这种问题通常不是很明显。例如，某个对象可能是某个大型复杂对象图表的一部分，而该图表恰好可使用静态变量访问，或在创建新线程或将工作排入线程池时通过将某个对象作为闭包的一部分进行传递可变为共享图表。

当对象（图表）从私有变为共享时，一定要多加注意。这称为发布，在后面的隔离上下文中会对此加以讨论。反之称为私有化，即对象（图表）再次从共享变为私有。对这种问题的解决方案是添加正确的同步。在计数器示例中，我可以使用简单的联锁：

```
static class Counter {  
    internal static volatile int s_curr = 0;  
    internal static int GetNext() {  
        return Interlocked.Increment(ref s_curr);  
    }  
}
```

它之所以起作用，是因为更新被限定在单一内存位置，还因为（这一点非常方便）存在硬件指令 (LOCK INC)，它相当于我尝试进行原子化操作的软件语句。或者，我可以使用成熟的锁定：

```
static class Counter {  
    internal static int s_curr = 0;  
    private static object s_currLock = new object();  
    internal static int GetNext() {  
        lock (s_currLock) {  
            return s_curr++;  
        }  
    }  
}
```

lock 语句可确保试图访问 GetNext 的所有线程彼此之间互斥，并且它使用 CLR System.Threading.Monitor 类。C++ 程序使用 CRITICAL_SECTION 来实现相同目的。虽然对这个特定的示例不必使用锁定，但当涉及多个操作时，几乎不可能将其并入单个互锁操作中。

粒度错误

即使使用正确的同步对共享状态进行访问，所产生的行为仍然可能是错误的。粒度必须足够大，才能将必须视为原子的操作封装在此区域中。这将导致在正确性与缩小区域之间产生冲突，因为缩小区域会减少其他线程等待同步进入的时间。

例如，让我们看一看图 1 所示的银行帐户抽象。一切都很正常，对象的两个方法（Deposit 和 Withdraw）看起来不会发生并发错误。一些银行业应用程序可能会使用它们，而且不用担心余额会因为并发访问而遭到损坏。

图 1 银行帐户

```
class BankAccount {  
    private decimal m_balance = 0.0M;  
    private object m_balanceLock = new object();  
    internal void Deposit(decimal delta) {  
        lock (m_balanceLock) { m_balance += delta; }  
    }  
    internal void Withdraw(decimal delta) {  
        lock (m_balanceLock) {  
            if (m_balance < delta)  
                throw new Exception("Insufficient funds");  
            m_balance -= delta;  
        }  
    }  
}
```

但是，如果您想添加一个 Transfer 方法该怎么办？一种天真的（也是不正确的）想法会认为由于 Deposit 和 Withdraw 是安全隔离的，因此很容易就可以合并它们：

```
class BankAccount {  
    internal static void Transfer(  

```

评论

写评论

目录

收藏

微信

微博

QQ

```
BankAccount a, BankAccount b, decimal delta) {
    Withdraw(a, delta);
    Deposit(b, delta);
}

// As before
}
```

这是不正确的。实际上，在执行 Withdraw 与 Deposit 调用之间的一段时间内资金会完全丢失。
正确的做法是必须提前对 a 和 b 进行锁定，然后再执行方法调用：

```
class BankAccount {
    internal static void Transfer(
        BankAccount a, BankAccount b, decimal delta) {
        lock (a.m_balanceLock) {
            lock (b.m_balanceLock) {
                Withdraw(a, delta);
                Deposit(b, delta);
            }
        }
    }
}

// As before
}
```

事实证明，此方法可解决粒度问题，但却容易发生死锁。稍后，您会了解到如何修复它。

读写撕裂

如前所述，良性争用允许您在没有同步的情况下访问变量。对于那些对齐的、自然分割大小的字——例如，用指针分割大小的内容在 32 位处理器中是 32 位的（4 字节），而在 64 位处理器中则是 64 位的（8 字节）——读写操作是原子的。如果某个线程只读取其他线程将要写入的单个变量，而没有涉及任何复杂的不变体，则在某些情况下您完全可以根据这一保证来略过同步。
但要注意。如果试图在未对齐的内存位置或未采用自然分割大小的位置这样做，可能会遇到读写撕裂现象。之所以发生撕裂现象，是因为此类位置的读或写实际上涉及多个物理内存操作。它们之间可能会发生并行更新，并进而导致其结果可能是之前的值和之后的值通过某种形式的组合。

例如，假设 ThreadA 处于循环中，现在需要仅将 0x0L 和 0xaaaabbbbccccdddl 写入 64 位变量 s_x 中。ThreadB 在循环中读取它（参见图 2）。

图 2 将要发生的撕裂现象

```
internal static volatile long s_x;

void ThreadA() {
    int i = 0;
    while (true) {
        s_x = (i & 1) == 0 ? 0x0L : 0xaaaabbbbccccdddl;
        i++;
    }
}

void ThreadB() {
    while (true) {
        long x = s_x;
        Debug.Assert(x == 0x0L || x == 0xaaaabbbbccccdddl);
    }
}
```

您可能会惊讶地发现 ThreadB 的声明可能会被触发。原因是 ThreadA 的写入操作包含两部分（高 32 位和低 32 位），具体顺序取决于编译器。ThreadB 的读取也是如此。因此 ThreadB 可以见证值 0xaaaabbbb00000000L 或 0x00000000aaaabbbbL。

无锁定重新排序

有时编写无锁定代码来实现更好的可伸缩性和可靠性是一种非常诱人的想法。这样做需要深入了解目标平台的内存模型（有关详细信息，请参阅 Vance Morrison 的文章 "Memory Models: Understand the Impact of Low-Lock Techniques in Multithreaded Apps"，网址为 msdn.microsoft.com/magazine/cc163715）。如果不了解或不注意这些规则可能会导致内存重新排序错误。之所以发生这些错误，是因为编译器和处理器在处理或优化期间可自由重新排序内存操作。

例如，假设 s_x 和 s_y 均被初始化为值 0，如下所示：

```
internal static volatile int s_x = 0;
internal static volatile int s_xa = 0;
internal static volatile int s_y = 0;
internal static volatile int s_ya = 0;

void ThreadA() {
    s_x = 1;
    s_ya = s_y;
}

void ThreadB() {
```


0


写评论

目录

收藏

微信





```
s_y = 1;
s_xa = s_x;
}
```

是否有可能在 ThreadA 和 ThreadB 均运行完成后，s_ya 和 s_xa 都包含值 0？看上去这个问题很可笑。或者 s_x = 1 或者 s_y = 1 会首先发生，在这种情况下，其他线程会在开始处理其自身的更新时见证这一更新。至少理论上如此。

遗憾的是，处理器随时都可能重新排序此代码，以便在写入之前加载操作更有效。您可以借助一个显式内存屏障来避免此问题：

```
void ThreadA() {
    s_x = 1;
    Thread.MemoryBarrier();
    s_ya = s_y;
}
```

.NET Framework 为此提供了一个特定 API，C++ 提供了 _MemoryBarrier 和类似的宏。但这个示例并不是想说明您应该在各处都插入内存屏障。它要说明的是在完全弄清内存模型之前，应避免使用无锁定代码，而且即使在完全弄清之后也应谨慎行事。

在 Windows（包括 Win32 和 .NET Framework）中，大多数锁定都支持递归获得。这只是意味着，即使当前线程已持有锁但当它试图再次获得时，其要求仍会得到满足。这使得通过较小的原子操作构成较大的原子操作变得更加容易。实际上，之前给出的 BankAccount 示例依靠的就是递归获得：Transfer 对 Withdraw 和 Deposit 都进行了调用，其中每个都重复获得了 Transfer 已获得的锁定。

但是，如果最终发生了递归获得操作而您实际上并不希望如此，则这可能就是问题的根源。这可能是由于重新进入而导致的，而发生重新进入的原因可能是由于对动态代码（如虚拟方法和委托）的显式调用或由于隐式重新输入的代码（如 STA 消息提取和异步过程调用）。因此，最好不要从锁定区域对动态方法进行调用。

例如，设想某个方法暂时破坏了不变体，然后又调用委托：

```
class C {
    private int m_x = 0;
    private object m_xLock = new object();
    private Action m_action = ...;

    internal void M() {
        lock (m_xLock) {
            m_x++;
            try { m_action(); }
            finally {
                Debug.Assert(m_x == 1);
                m_x--;
            }
        }
    }
}
```

C 的方法 M 可确保 m_x 不发生改变。但会有很短的一段时间，m_x 会先递增 1，然后再重新递减。对 m_action 的调用看起来没有任何问题。遗憾的是，如果它是从 C 类用户接受的委托，则表示任何代码都可以执行它所请求的操作。这包括回调到同一实例的 M 方法。如果发生了这种情况，finally 中的声明可能会被触发；同一堆栈中可能存在多个针对 M 的活动的调用（即使您未直接执行此操作），这必然会导致 m_x 包含的值大于 1。

当多个线程遇到死锁时，系统会直接停止响应。多篇《MSDN 杂志》文章都介绍了死锁的发生原因以及使死锁变得能够接受的一些方法，其中包括我自己的文章 "No More Hangs: Advanced Techniques to Avoid and Detect Deadlocks in .NET Apps"（网址为 msdn.microsoft.com/magazine/cc163618）以及 Stephen Toub 的 2007 年 10 月 .NET 相关问题专栏（网址为 msdn.microsoft.com/magazine/cc163352），因此这里只做简单的讨论。总而言之，只要出现了循环等待链——例如，ThreadA 正在等待 ThreadB 持有的资源，而 ThreadB 反过来也在等待 ThreadA 持有的资源（也许是间接等待第三个 ThreadC 或其他资源）——则所有向前的推进工作都可能会停下来。

此问题的常见根源是互斥锁。实际上，之前所示的 BankAccount 示例遇到的就是这个问题。如果 ThreadA 试图将 \$500 从帐户 #1234 转移到帐户 #5678，与此同时 ThreadB 试图将 \$500 从 #5678 转移到 #1234，则代码可能发生死锁。

使用一致的获得顺序可避免死锁，如图 3 所示。此逻辑可概括为“同步锁获得”之类的名称，通过此操作可依照各个锁之间的某种顺序动态排序多个可锁定的对象，从而使得在以一致的顺序获得两个锁的同时必须维持两个锁的位置。另一个方案称为“锁矫正”，可用于拒绝被认定以不一致的顺序完成的锁获得。

图 3 一致的获得顺序

```
class BankAccount {
    private int m_id; // Unique bank account ID.

    internal static void Transfer(
        BankAccount a, BankAccount b, decimal delta) {
        if (a.m_id < b.m_id) {
            Monitor.Enter(a.m_balanceLock); // A first
            Monitor.Enter(b.m_balanceLock); // ...and then B
        } else {
            Monitor.Enter(b.m_balanceLock); // B first
            Monitor.Enter(a.m_balanceLock); // ...and then A
        }
        try {
            Withdraw(a, delta);
            Deposit(b, delta);
        } finally {
```



0



写评论



目录



收藏



微信



微博



QQ

```
        Monitor.Exit(a.m_balanceLock);  
        Monitor.Exit(b.m_balanceLock);  
    }  
}  
  
// As before ...  
}
```

但锁并不是导致死锁的唯一根源。唤醒丢失是另一种现象，此时某个事件被遗漏，导致线程永远休眠。在 Win32 自动重置和手动重置事件、CONDITION_VARIABLE、CLR Monitor.Wait、Pulse 以及 PulseAll 调用等同步事件中经常会发生这种情况。唤醒丢失通常是一种迹象，表示同步不正确，无法重置等待条件或在 wake-all (WakeAllConditionVariable 或 Monitor.PulseAll) 更为适用的情况下使用了 wake-single 基元 (WakeConditionVariable 或 Monitor.Pulse) 。

此问题的另一个常见根源是自动重置事件和手动重置事件信号丢失。由于此类事件只能处于一个状态 (有信号或无信号) ，因此用于设置此事件的冗余调用实际上将被忽略不计。如果代码认定要设置的两个调用始终需要转换为两个唤醒的线程，则结果可能就是唤醒丢失。

锁保护

当某个锁的到达率与其锁获得率相比始终居高不下时，可能会产生锁保护。在极端的情况下，等待某个锁的线程超过了其承受力，就会导致灾难性后果。对于服务器端的程序而言，如果客户端所需的某些受锁保护的数据结构需求量大增，则经常会发生这种情况。

例如，请设想以下情况：平均来说，每 100 毫秒会到达 8 个请求。我们将八个线程用于服务请求 (因为我们使用的是 8-CPU 计算机) 。这八个线程中的每一个都必须获得一个锁并保持 20 毫秒，然后才能展开实质性的工作。

遗憾的是，对这个锁的访问需要进行序列化处理，因此，全部八个线程需要 160 毫秒才能进入并离开锁。第一个退出后，需要经过 140 毫秒第九个线程才能访问该锁。此方案本质上无法进行调整，因此备份的请求会不断增长。随着时间的推移，如果到达率不降低，客户端请求就会开始超时，进而发生灾难性后果。

众所周知，在锁中是通过公平性对锁进行保护的。原因在于在锁本来已经可用的时间段内，锁被人为封闭，使得到达的线程必须等待，直到所选锁的拥有者线程能够唤醒、切换上下文以及获得和释放该锁为止。为解决这种问题，Windows 已逐渐将所有内部锁都改为不公平锁，而且 CLR 监视器也是不公平的。

对于这种有关保护的基本问题，唯一的有效解决方案是减少锁持有时间并分解系统以尽可能减少热锁 (如果有的话) 。虽然说起来容易做起来难，但这对于可伸缩性来说还是非常重要的。

“蜂拥”是指大量线程被唤醒，使得它们全部同时从 Windows 线程计划程序争夺关注点。例如，如果在单个手动设置事件中有 100 个阻塞的线程，而您设置该事件...嗯，算了吧，您很可能会把事情弄得一团糟，特别是当其中的大部分线程都必须再次等待时。

实现阻塞队列的一种途径是使用手动设置事件，当队列为空时变为无信号而在队列非空时变为有信号。遗憾的是，如果从零个元素过渡到一个元素时存在大量正在等待的线程，则可能会发生蜂拥。这是因为只有一个线程会得到此单一元素，此过程会使队列变空，从而必须重置该事件。如果有 100 个线程在等待，那么其中的 99 个将被唤醒、切换上下文 (导致所有缓存丢失) ，所有这些换来的只是不得不再次等待。

两步舞曲

有时您需要在持有锁的情况下通知一个事件。如果唤醒的线程需要获得被持有的锁，则这可能会很不凑巧，因为它被唤醒后只是发现了它必须再次等待。这样做非常浪费资源，而且会增加上下文切换的总数。此情况称为两步舞曲，如果涉及到许多锁和事件，可能会远远超出两步的范畴。

Win32 和 CLR 的条件变量支持在本质上都会遇到两步舞曲问题。它通常是不可避免的，或者很难解决。

两步舞曲问题在单处理器计算机上情况更糟。在涉及到事件时，内核会将优先级提升应用到唤醒的线程。这几乎可以保证抢占占用线程，使其能够在有机会释放锁之前设置事件。这是在极端情况下的两步舞曲，其中设置 ThreadA 已切换出上下文，使得唤醒的 ThreadB 可以尝试获得锁；当然它无法做到，因此它将进行上下文切换以使 ThreadA 可再次运行；最终，ThreadA 将释放锁，这将再次提升 ThreadB 的优先级，使其优先于 ThreadA，以便它能够运行。如您所见，这涉及了多次无用的上下文切换。

优先级反转

修改线程优先级常常是自找苦吃。当不同优先级的许多线程共享对同样的锁和资源的访问权时，可能会发生优先级反转，即较低优先级的线程实际无限期地阻止较高优先级线程的进度。这个示例所要说明的道理就是尽可能避免更改线程优先级。

下面是一个优先级反转的极端示例。假设低优先级的 ThreadA 获得某个锁 L。随后高优先级的 ThreadB 介入。它尝试获得 L，但由于 ThreadA 占用使得它无法获得。下面就是“反转”部分：好像 ThreadA 被人临时赋予了一个高于 ThreadB 的优先级，这一切只是因为它持有 ThreadB 所需的锁。

当 ThreadA 释放了锁后，此情况最终会自行解决。遗憾的是，如果涉及到中等优先级的 ThreadC，设想一下会发生什么情况。虽然 ThreadC 不需要锁 L，但它的存在可能会从根本上阻止 ThreadA 运行，这将间接地阻止高优先级 ThreadB 的运行。

最终，Windows Balance Set Manager 线程会注意到这一情况。即使 ThreadC 保持永远可运行状态，ThreadA 最终 (四秒钟后) 也将接收到操作系统发出的临时优先级提升指令。但愿这足以使其运行完毕并释放锁。但这里的延迟 (四秒钟) 相当巨大，如果涉及到任何用户界面，则应用程序用户肯定会注意到这一问题。

实现安全性的模式

现在我已经找出了一个又一个的问题，好消息是我这里还有几种设计模式，您可以遵循它们来降低上述问题 (尤其是正确性危险) 的发生频率。大多数问题的关键是由于状态在多个线程之间共享。更糟的是，此状态可被随意控制，可从一致状态转换为不一致状态，然后 (但愿) 又重新转换回来，具有令人惊讶的规律性。

当开发人员针对单线程程序编写代码时，所有这些都非常有用。在您向最终的正确目标迈进的过程中，很可能会使用共享内存作为一种暂存器。多年来 C 语言风格的命令式编程语言一直使用这种方式工作。

但随着并发现象越来越多，您需要对这些习惯密切加以关注。您可以按照 Haskell、LISP、Scheme、ML 甚至 F# (一种符合 .NET 的新语言) 等函数式编程语言行事，即采用不变性、纯度和隔离作为一类设计概念。

不变性

具有不变性的数据结构是指在构建后不会发生改变的结构。这是并发程序的一种奇妙属性，因为如果数据不改变，则即使许多线程同时访问它也不会存在任何冲突风险。这意味着同步并不是一个需要考虑的因素。

不变性在 C++ 中通过 `const` 提供支持，在 C# 中通过只读修饰符支持。例如，仅具有只读字段的 .NET 类型是浅层不变的。默认情况下，F# 会创建固定不变的类型，除非您使用可变修饰符。再进一步，如果这些字段中的每个字段本身都指向字段均为只读（并仅指向深层不可变类型）的另一种类型，则该类型是深层不可变的。这将产生一个保证不会改变的完整对象图表，它会非常有用。

所有这一切都说明不变性是一个静态属性。按照惯例，对象也可以是固定不变的，即在某种程度上可以保证状态在某个时间段不会改变。这是一种动态属性。Windows Presentation Foundation (WPF) 的可冻结功能恰好可实现这一点，它还允许在不同步的情况下进行并行访问（但是无法以处理静态支持的方式对其进行检查）。对于在整个生存期内需要在固定不变和可变之间进行转换的对象来说，动态不变性通常非常有用。

不变性也存在一些弊端。只要有内容需要改变，就必须生成原始对象的副本并在此过程中应用更改。另外，在对象图表中通常无法进行循环（除动态不变性外）。

例如，假设您有一个 `ImmutableStack<T>`，如图 4 所示。您需要从包含已应用更改的对象中返回新的 `ImmutableStack<T>` 对象，而不是一组变化的 `Push` 和 `Pop` 方法。在某些情况下，可以灵活使用一些技巧（与堆栈一样）在各实例之间共享内存。

图 4 使用 `ImmutableStack`

```
public class ImmutableStack<T> {
    private readonly T m_value;
    private readonly ImmutableStack<T> m_next;
    private readonly bool m_empty;
    public ImmutableStack() { m_empty = true; }
    internal ImmutableStack(T value, Node next) {
        m_value = value;
        m_next = next;
        m_empty = false;
    }
    public ImmutableStack<T> Push(T value) {
        return new ImmutableStack(value, this);
    }
    public ImmutableStack<T> Pop(out T value) {
        if (m_empty) throw new Exception("Empty.");
        return m_next;
    }
}
```



0



写评论



目录



收藏



微信



微博



QQ

节点被推入时，必须为每个节点分配一个新对象。在堆栈的标准链接列表实现中，必须执行此操作。但是要注意，当您从堆栈中弹出元素时，可以使用现有的对象。这是因为堆栈中的每个节点是固定不变的。

固定不变的类型无处不在。CLR 的 `System.String` 类是固定不变的，还有一个设计指导原则，即所有新值类型都应是固定不变的。此处给出的指导原则是在可行和合适的情况下使用不变性并抵抗执行变化的诱惑，而最新一代的语言会使其变得非常方便。

纯度

即使是使用固定不变的数据类型，程序所执行的大部分操作仍是方法调用。方法调用可能存在一些副作用，它们在并发代码中会引发问题，因为副作用意味着某种形式的变化。通常这只是表示写入共享内存，但它也可能是实际变化的操作，如数据库事务、Web 服务调用或文件系统操作。在许多情况下，我希望能够调用某种方法，而又不必担心它会导致并发危险。有关这一方面的一些很好示例就是 `GetHashCode` 和 `ToString` on `System.Object` 等简单的方法。很多人都不希望它们带来副作用。

纯方法始终都可以在并发设置中运行，而无需添加同步。尽管纯度没有任何常见语言支持，但您可以非常简单地定义纯方法：

1. 它只从共享内存读取，并且只读取不变状态或常态。
2. 它必须能够写入局部变量。
3. 它可以只调用其他纯方法。

因此，纯方法可以实现的功能非常有限。但当与不变类型结合使用时，纯度就会成为可能而且非常方便。一些函数式语言默认情况下都采用纯度，特别是 `Haskell`，它的所有内容都是纯的。任何需要执行副作用的内容都必须封装到一个被称为 `monad` 的特殊内容中。但是我们中的多数人不使用 `Haskell`，因此我们必须遵照纯度约定。

隔离

前面我们只是简单提及了发布和私有化，但它们却击中了—个非常重要的问题的核心。由于状态通常在多个线程之间共享，因此同步是必不可少的（不变性和纯度也很有趣味）。但如果状态被限制在单个线程内，则无需进行同步。这会导致软件在本质上更具伸缩性。

实际上，如果状态是隔离的，则可以自由变化。这非常方便，因为变化是大部分 C 风格语言的基本内置功能。程序员已习惯了这一点。这需要进行训练以便能够在编程时以函数式风格为主，对大多数开发人员来说这都相当困难。尝试一下，但不要自欺欺人地认为世界会在一夜之间改为使用函数式风格编程。

所有权是一件很难跟踪的事情。对象是何时变为共享的？在初始化时，这是由单线程完成的，对象本身还不能从其他线程访问。将对某个对象的引用存储在静态变量中、存储在已在线程创建或排列队列时共享的某个位置或存储在可从其中的某个位置传递性访问的对象字段中之后，该对象就变为共享对象。开发人员必须特别关注私有与共享之间的这些转换，并小心处理所有共享状态。

Joe Duffy 在 Microsoft 是 .NET 并行扩展方面的开发主管。他的大部分时间都在攻击代码、监督库的设计以及管理梦幻开发团队。他的最新著作是《Concurrent Programming on Windows》。

英文原文如下：

Concurrency Hazards

Solving 11 Likely Problems In Your Multithreaded Code

Joe Duffy

Concurrency is everywhere. Server-side programs have long had to deal with a fundamentally concurrent programming model, and as multicore processors become more commonplace, client-side programs will have to as well. Along with the addition of concurrency comes the responsibility for ensuring safety. In other words, programs must continue to achieve the same level of robustness and reliability in the face of large amounts of logical concurrency and ever-changing degrees of physical hardware parallelism.

Correctly engineered concurrent code must live by an extra set of rules when compared to its sequential counterpart. Reads and writes from memory and access to shared resources need to be regulated using synchronization so that conflicts do not arise. Additionally, it often becomes necessary for threads to coordinate to get a certain job done.

As a direct result of these additional requirements, it becomes nontrivial to ensure that threads are continually making consistent and adequate forward progress. Synchronization and coordination deeply depend on timing, which is nondeterministic and hard to predict and test for.

What makes these attributes difficult is simply that it requires a mindset shift. There isn't a single API that you can learn or a snippet of code to copy and paste. There really is a fundamental set of concepts that you need to learn and become comfortable with. It's likely that certain languages and libraries can hide some concepts over time, but if you're doing concurrency today, you won't have that luxury. This article describes some of the more common challenges to be aware of and presents advice for coping with them in your software.

I'll start by looking at the kinds of things that often go wrong in concurrent programs. I call these "safety hazards" because they are so easy to stumble upon and because their consequences are typically quite undesirable. These hazards cause your programs to break by either crashing or corrupting memory.

A data race—or race condition—occurs when data is accessed concurrently from multiple threads. Specifically, it happens when one or more threads are writing a piece of data while one or more threads are also reading that piece of data. This problem arises because Windows programs (in C++ and the Microsoft .NET Framework alike) are fundamentally based on the concept of shared memory, where all threads in a process may access data residing in the same virtual address space. Static variables and heap allocation can be used to share.

Consider this canonical example:

```
//C#
```

```
static class Counter {  
    internal static int s_curr = 0;  
    internal static int GetNext() {  
        return s_curr++;  
    }  
}
```

```
`Visual Basic
```

```
Friend Class Counter  
    Friend Shared s_curr As Integer = 0  
    Friend Shared Function GetNext() As Integer  
        Return s_curr = s_curr + 1  
    End Function  
End Class
```

The goal of Counter is presumably to hand out a new unique number for each call to GetNext. If two threads in the program both call GetNext simultaneously, however, two threads might be given the same number. The reason is that `s_curr++` compiles into three separate steps:

1. Read the current value from the shared `s_curr` variable into a processor register.
2. Increment that register.
3. Write the register value back to the shared `s_curr` variable.

Two threads executing this same sequence can both read the same value from `s_curr` locally (say, 42), increment it (to, say, 43), and publish the same resulting value. GetNext thus returns the same number for both threads, breaking the algorithm. Although the simple statement `s_curr++` appears to be atomic, this couldn't be farther from the truth.

Forgotten Synchronization

This is the simplest kind of data race: synchronization is forgotten altogether. Races of this kind can be, in rare situations, benign, meaning that they are correct, but most are fundamental problems of correctness.

Problems like this aren't always so obvious. For example, an object may be part of some large complicated object graph that happens to be reachable from a static variable or became shared by passing an object as part of a closure when creating a new thread or queuing work to a thread pool.

It is imperative to pay attention to when an object (graph) transitions from private to shared. This is called publication and will be discussed later in the context of isolation. The reverse is called privatization—that is, when an object (graph) goes from being shared to private again.

The solution is to add proper synchronization. In the example of the counter, I could use a simple interlocked:

```
//C#

static class Counter {
    internal static volatile int s_curr = 0;
    internal static int GetNext() {
        return Interlocked.Increment(ref s_curr);
    }
}
```



0



写评论

```
`Visual Basic
Friend Class Counter
    Friend Shared s_curr As Integer = 0
    Friend Shared Function GetNext() As Integer
        Monitor.Enter(s_curr)
        Try
            Return Interlocked.Increment(s_curr)
        Finally
            Monitor.Exit(s_curr)
        End Try
    End Function
End Class
```



目录



收藏



微信



微博



QQ

This works because the update is confined to a single memory location, and because (conveniently) there is a hardware instruction (LOCK INC) that is equivalent to the software statement I'm trying to make atomic.

Alternatively, I could use a full-fledged lock:

```
//C#

static class Counter {
    internal static int s_curr = 0;
    private static object s_currLock = new object();
    internal static int GetNext() {
        lock (s_currLock) {
            return s_curr++;
        }
    }
}
```

```
`Visual Basic

Friend Class Counter
    Friend Shared s_curr As Integer = 0
    Private Shared s_currLock As New Object()
    Friend Shared Function GetNext() As Integer
        SyncLock s_currLock
            Return s_curr = s_curr + 1
        End SyncLock
    End Function
End Class
```

The lock statement ensures mutual exclusion among all threads trying to access GetNext, and uses the CLR System.Threading.Monitor class. C++ programs would use a CRITICAL_SECTION for the same purpose. There is no need to go with a lock for this particular example, but when multiple operations are involved it's seldom possible to fold them into a single interlocked operation.

Incorrect Granularity

Even if access to shared state occurs with proper synchronization, the resulting behavior may still be incorrect. The granularity must be sufficiently large that the operations that must be seen as atomic are encapsulated within the region. There is a tension between correctness and making the region smaller, because smaller regions reduce the time that other threads will have to wait to concurrently enter.

For example, take a look at the bank account abstraction shown in Figure 1. All is well, and the object's two methods, Deposit and Withdraw, appear to be concurrency-safe. Some banking application can use them without worry that balances will become corrupt due to concurrent access.


Figure 1 A Bank Account


```
//C#


class BankAccount {
    private decimal m_balance = 0.0M;
    private object m_balanceLock = new object();
    internal void Deposit(decimal delta) {
        lock (m_balanceLock) { m_balance += delta; }
    }
    internal void Withdraw(decimal delta) {
        lock (m_balanceLock) {
            if (m_balance < delta)
                throw new Exception("Insufficient funds");
            m_balance -= delta;
        }
    }
}


`Visual Basic


Friend Class BankAccount
    Private m_balance As Decimal = 0D
    Private m_balanceLock As New Object()
    Friend Sub Deposit(ByVal delta As Decimal)
        SyncLock m_balanceLock
            m_balance += delta
        End SyncLock
    End Sub
    Friend Sub Withdraw(ByVal delta As Decimal)
        SyncLock m_balanceLock
            If m_balance < delta Then
                Throw New Exception("Insufficient funds")
            End If
            m_balance -= delta
        End SyncLock
    End Sub
End Class
```



0



写评论


目录


收藏


微信


微博


QQ

What if, however, you'd like to add a Transfer method? A naive (and incorrect) attempt would assume that, because Deposit and Withdraw are safe in isolation, they can be combined easily:

```
//C#

class BankAccount {
    internal static void Transfer(
        BankAccount a, BankAccount b, decimal delta) {
        Withdraw(a, delta);
        Deposit(b, delta);
    }
    // As before
}

`Visual Basic

Friend Class BankAccount
    Friend Shared Sub Transfer(ByVal a As BankAccount, ByVal b As BankAccount, ByVal delta As Decimal)
        Withdraw(a, delta)
    End Sub
End Class
```

```

        Deposit(b, delta)
    End Sub
    ' As before
End Class

```

This is incorrect. There is actually a period of time between the Withdraw and Deposit calls where the money is missing completely.

A correct implementation would need to acquire the locks on both a and b up front and then make the method calls:

```

//C#

class BankAccount {
    internal static void Transfer(
        BankAccount a, BankAccount b, decimal delta) {
        lock (a.m_balanceLock) {
            lock (b.m_balanceLock) {
                Withdraw(a, delta);
                Deposit(b, delta);
            }
        }
    }
}
// As before
}

```



0



写评论



目录



收藏



微信

```

`Visual Basic

```



微博

```

Friend Class BankAccount
    Friend Shared Sub Transfer(ByVal a As BankAccount, ByVal b As BankAccount, ByVal delta As Decimal)
        SyncLock a.m_balanceLock
            SyncLock b.m_balanceLock
                Withdraw(a, delta)
                Deposit(b, delta)
            End SyncLock
        End SyncLock
    End Sub
    ' As before
End Class

```



QQ

It turns out that this approach, while solving the granularity problem, is prone to deadlock. You'll see how to fix it later on.

Read and Write Tearing

As mentioned earlier, benign races allow you to access variables without synchronization. Reads and writes of aligned, naturally sized words—for example, pointer-sized things that are 32 bits (4 bytes) on 32-bit processors and 64 bits (8 bytes) on 64-bit processors—are atomic. If a thread is just reading a single variable that some other thread will write, and there aren't any complex invariants involved, you can sometimes skip the synchronization altogether thanks to this guarantee.

But beware. If you try this on a misaligned memory location, or a location that isn't naturally sized, you can encounter a read or write tearing. Tearing occurs because reading or writing such locations actually involves multiple physical memory operations. Concurrent updates can happen in between these, potentially causing the resultant value to be some blend of the before and after values.

As an example, imagine ThreadA sits in a loop and writes only 0x0L and 0xaaaabbbbccccdddL to a 64-bit variable s_x. ThreadB sits in a loop reading it (see Figure 2).

Figure 2 Tearing about to Happen

```

//C#

internal static volatile long s_x;
void ThreadA() {
    int i = 0;
    while (true) {
        s_x = (i & 1) == 0 ? 0x0L : 0xaaaabbbbccccdddL;
        i++;
    }
}

```

```

void ThreadB() {
    while (true) {
        long x = s_x;
        Debug.Assert(x == 0x0L || x == 0xaaaabbbbccccdddL);
    }
}

```

```
//C#
```

```

internal static volatile long s_x;

void ThreadA() {
    int i = 0;
    while (true) {
        s_x = (i & 1) == 0 ? 0x0L : 0xaaaabbbbccccdddL;
        i++;
    }
}

void ThreadB() {
    while (true) {
        long x = s_x;
        Debug.Assert(x == 0x0L || x == 0xaaaabbbbccccdddL);
    }
}

```

You may be surprised to discover that ThreadB's assert might fire. The reason is that ThreadA's write will consist of two parts, the high 32-bits and the low 32-bits, in some order depending on the compiler. The same goes for ThreadB's read. Thus ThreadB could witness values 0xaaaabbbb00000000L or 0x00000000aaaabbbbL.

Lock-Free Reordering

Sometimes it's tempting to write lock-free code as a way of achieving better scalability and reliability. Doing so requires a deep understanding of the memory model of your target platform (see Vance Morrison's article "Memory Models: Understand the Impact of Low-Lock Techniques in Multithreaded Apps" at msdn.microsoft.com/magazine/cc163715 for details). Failure to understand and heed these rules can lead to memory reordering bugs. These happen because compilers and processors are free to reorder memory operations during the process or making optimizations.

As an example, assume s_x and s_y are both initialized to the value 0, as you see here:

```
//C#
```

```

internal static volatile int s_x = 0;
internal static volatile int s_xa = 0;
internal static volatile int s_y = 0;
internal static volatile int s_ya = 0;

void ThreadA() {
    s_x = 1;
    s_ya = s_y;
}

void ThreadB() {
    s_y = 1;
    s_xa = s_x;
}

```

```
`Visual Basic
```

```

Friend Shared s_x As Integer = 0
Friend Shared s_xa As Integer = 0
Friend Shared s_y As Integer = 0
Friend Shared s_ya As Integer = 0

Private Sub ThreadA()
    s_x = 1;
    Thread.MemoryBarrier();
    s_ya = s_y;
End Sub

```



0



写评论



目录



收藏



微信



微博



QQ

```
Private Sub ThreadA()
    s_y = 1;
    Thread.MemoryBarrier();
    s_xa = s_x;
End Sub
```

Is it possible that, after ThreadA and ThreadB both run to completion, both s_ya and s_xa will contain the value 0? This seems ridiculous on its face. Either s_x = 1 or s_y = 1 will happen first, in which case the other thread will witness the update when it gets around to its update. Or so the theory goes.

Unfortunately, processors are free to reorder this code so that the loads effectively happen before the writes. You can get around this with an explicit memory barrier:

```
//C#

void ThreadA() {
    s_x = 1;
    Thread.MemoryBarrier();
    s_ya = s_y;
}
```

```
`Visual Basic

Private Sub ThreadA()
    s_x = 1
    Thread.MemoryBarrier()
    s_ya = s_y
End Sub
```



0



写评论



目录



收藏



微信



微博



QQ

The .NET Framework offers this particular API, and C++ offers `_MemoryBarrier` and similar macros. But the moral of the story is not that you should insert memory barriers all over the place. The moral is that you should steer clear of lock-free code until you've mastered memory models, and even then tread with care.

In Windows, including Win32 and the .NET Framework, most locks support recursive acquires. That just means that, if the current thread already holds a lock and tries to acquire it again, the request will be satisfied. This makes it easier to compose bigger atomic operations out of smaller ones. In fact, the BankAccount example shown earlier depended on recursive acquires: Transfer called both Withdraw and Deposit, each of which redundantly acquired a lock that Transfer had already acquired.

This can be the source of problems, however, if you end up in a situation where a recursive acquire occurs when you didn't expect it to. This can happen as a result of reentrancy, which is possible due to either explicit callouts to dynamic code (such as virtual methods and delegates) or because of implicitly reentered code (such as STA message pumping and asynchronous procedure calls). For this reason, among others, it's a good idea to never make a call to a dynamic method from within a lock region.

For example, imagine some method that breaks invariants temporarily and then calls a delegate:

```
//C#

class C {
    private int m_x = 0;
    private object m_xLock = new object();
    private Action m_action = ...;

    internal void M() {
        lock (m_xLock) {
            m_x++;
            try { m_action(); }
            finally {
                Debug.Assert(m_x == 1);
                m_x--;
            }
        }
    }
}
```

```

`Visual Basic

Friend Class C
    Private m_x As Integer = 0
    Private m_xLock As New Object()
    Private m_action As Action = ...

    Friend Sub M()
        SyncLock m_xLock
            m_x += 1
            Try
                m_action()
            Finally
                Debug.Assert(m_x = 1)
                m_x -= 1
            End Try
        End SyncLock
    End Sub
End Class

```



0



写评论



目录

C's method M ensures that `m_x` does not change. But there is a brief period of time where `m_x` is incremented by one before being decremented again. The callout to `m_action` looks innocent enough. Unfortunately, if it was a delegate accepted from users of the C class, it represents arbitrary code that can do anything it pleases. That includes calling back into the same instance's M method. And if that happens, the assert within the finally is apt to fire; there could be multiple calls to M active on the same stack, even though you didn't do it directly, which clearly can lead to `m_x` holding a value greater than 1.

When multiple threads encounter a deadlock, the system simply stops responding. Several MSDN Magazine articles have described how deadlocks occur and some ways of tolerating them—including my own article, "No More Hangs: Advanced Techniques to Avoid and Detect Deadlocks in .NET Apps," at msdn.microsoft.com/magazine/cc163618, and Stephen Toub's October 2007 .NET Matters column at msdn.microsoft.com/magazine/cc163352—so the discussion here will be kept light. In summary, whenever a circular wait chain is created—such that some ThreadA is waiting for a resource held by ThreadB, which is also reflexively waiting for a resource held by ThreadA (perhaps indirectly by waiting for a third ThreadC or more)—it is possible for all forward progress to come to a grinding halt.

A common source of this problem is with mutually exclusive locks. In fact, the BankAccount example shown earlier suffers from this problem. If ThreadA tries to transfer \$500 from account #1234 to account #5678 at the same time ThreadB tries to transfer \$500 from #5678 to #1234, the code can deadlock.

Using a consistent acquisition order can avoid the deadlock, as shown in Figure 3. This logic can be generalized to something called simultaneous lock acquisition, whereby multiple lockable objects are sorted dynamically according to some ordering among locks, so that any place two locks must be held at the same time they are acquired in a consistent order. Another scheme called lock leveling can be used to reject lock acquisitions that are provably done in an inconsistent order.

Figure 3 Consistent Acquisition Order

```

//C#

class BankAccount {
    private int m_id; // Unique bank account ID.
    internal static void Transfer(
        BankAccount a, BankAccount b, decimal delta) {
        if (a.m_id < b.m_id) {
            Monitor.Enter(a.m_balanceLock); // A first
            Monitor.Enter(b.m_balanceLock); // ...and then B
        } else {
            Monitor.Enter(b.m_balanceLock); // B first
            Monitor.Enter(a.m_balanceLock); // ...and then A
        }
        try {
            Withdraw(a, delta);
            Deposit(b, delta);
        } finally {
            Monitor.Exit(a.m_balanceLock);
            Monitor.Exit(b.m_balanceLock);
        }
    }
}
// As before ...
}

```

```
'Visual Basic
```

```
Friend Class BankAccount
    Private m_id As Integer ' Unique bank account ID.
    Friend Shared Sub Transfer(ByVal a As BankAccount, ByVal b As BankAccount, ByVal delta As Decimal)
        If a.m_id < b.m_id Then
            Monitor.Enter(a.m_balanceLock) ' A first
            Monitor.Enter(b.m_balanceLock) '...and then B
        Else
            Monitor.Enter(b.m_balanceLock) ' B first
            Monitor.Enter(a.m_balanceLock) '...and then A
        End If
        Try
            Withdraw(a, delta)
            Deposit(b, delta)
        Finally
            Monitor.Exit(a.m_balanceLock)
            Monitor.Exit(b.m_balanceLock)
        End Try
    End Sub
    ' As before ...
End Class
```



0



写评论



目录



收藏



微信



微博



But locks are not the only source of deadlock. Missed wake-ups are another phenomenon, where some event is missed and a thread sleeps forever. This is common with synchronization events like Win32 auto-reset and manual-reset events, `CONDITION_VARIABLEs`, and CLR `Monitor.Wait`, `Pulse`, and `PulseAll` calls. A missed wake-up is usually a sign of improper synchronization, failure to retest a wait condition, or use of a wake-single primitive (`WakeConditionVariable` or `Monitor.Pulse`) when a wake-all (`WakeAllConditionVariable` or `Monitor.PulseAll`) would have been more appropriate.

Another common source of this problem is lost signals with auto- and manual-reset events. Because such an event can only be in one state—signaled or nonsignaled—redundant calls to set the event will effectively be ignored. If code assumes that two calls to set always translates to two threads awakened, the result can be a missed wake-up.

Lock Convoys

When the arrival rate at a lock is consistently high compared to its lock acquisition rate, a lock convoy may result. In the extreme, there are more threads waiting at a lock than can be serviced, leading to a catastrophe. This is more common on server-side programs where certain locks protecting data structures needed by most clients can get unusually hot.

For example, imagine this scenario: On average, eight requests arrive per 100 milliseconds. We use eight threads to service requests (because we're on an 8-CPU machine). Each of those eight threads must acquire a lock and hold it for 20 milliseconds before it can do meaningful work.

Unfortunately, access to this lock must be serialized, so it takes 160 milliseconds for all eight threads to enter and leave the lock. After the first exists, there will be 140 milliseconds of time before a ninth thread can access the lock. This scheme inherently will not scale, and there will be a continuously growing backup of requests. Over time, if the arrival rate does not decrease, client requests are apt to begin timing out, and a disaster will ensue.

Fairness in locks is known to contribute to convoys. The reason is that periods of time where the lock could have been made available are artificially blocked out, so that threads arriving must wait until the chosen lock owner thread can wake up, context switch, and acquire and release the lock. Windows has changed all of its internal locks to be unfair over time in order to combat this problem, and CLR monitors are also unfair.

The only real solution to the fundamental convoy problem is to reduce lock hold times and to factor your system so that there are very few (if any) hot locks. This is easier said than done, but is crucial for scalability.

A stampede is a situation where many threads are awakened, such that they all fight for attention simultaneously from the Windows thread scheduler. If you have 100 threads blocked on a single manual-reset event, for example, and you set that event ... well, you're apt to have a real mess on your hands, particularly if a large portion of those threads will have to wait again.

One way to implement a blocking queue is to use a manual-reset event that gets unsignaled when the queue is empty, and becomes signaled when it is non-empty. Unfortunately, when there are a large number of waiting threads during the transition from zero elements to one element, a stampede can occur. That's because only one thread will take the single element, which transitions the queue back to empty, and necessarily involves resetting the event. If you had 100 threads waiting, then 99 of them will wake up, context switch (and incur all those cache misses), just to realize they have to wait again.

Two-Step Dance

Sometimes you need to signal an event while holding a lock. This can be unfortunate if the waking thread needs to acquire the lock held, because it will be awakened only to find out that it must wait again. This is wasteful and increases the number of overall context switches. This situation is called the two-step dance, and can extend far beyond just two steps if many locks and events are involved.

Both Win32 and the CLR's condition variable support inherently suffers from the two-step dance problem. It is often unavoidable, or prohibitively difficult to work around.

The two-step dance issue is even worse on a single-processor machine. When events are involved, the kernel will apply a priority boost to the awakening thread. This is nearly guaranteed to preempt the thread setting the event before it has a chance to release the lock. This is two-step dance in the extreme, where the setting ThreadA is context switched out so that the awakening ThreadB can attempt to acquire the lock; it of course cannot, and so it will context switch out so that ThreadA can run again; eventually, ThreadA will release the lock, which again will priority boost ThreadB, which preempts ThreadA so that it can run. As you can see, there's a lot of wasteful context switching going on.

Priority Inversion

Modifying thread priorities is usually asking for trouble. Priority inversion can arise when many threads at different priorities share access to the same locks and resources, whereby a lower-priority thread actually indefinitely holds up the progress of a higher-priority thread. The moral of this story is to simply avoid changing thread priorities wherever possible.

Here is an extreme example of priority inversion. Imagine ThreadA at Low priority acquires some lock L. Then comes along ThreadB, which is at High priority. It tries to acquire L, but cannot because ThreadA holds it. This is the "inversion" part: it's as if ThreadA is artificially and temporarily given a higher priority than ThreadB, simply because it holds a lock that ThreadB needs.

The situation will eventually resolve itself when ThreadA releases the lock. Unfortunately, imagine what happens if ThreadC at Medium priority comes into the picture. Although ThreadC doesn't need lock L, its mere presence may prevent ThreadA from running at all, which indirectly prevents the High-priority ThreadB from running.

Eventually the Windows Balance Set Manager thread will notice this situation. Even if ThreadC remains runnable forever, ThreadA will eventually (after four seconds) receive a temporary priority boost by the OS. Hopefully this is enough to allow it to run to completion and release the lock. But the delay here (four seconds) is huge, and if there is any user interface involved, the application's user is certainly going to notice the problem.

Patterns for Achieving Safety

Now that I've chased down issue after issue, the good news is that there are design patterns you can follow to make the occurrence of many of the above issues (particularly the correctness hazards) far less frequent. The crux of most issues is that state is shared among multiple threads. What's worse, this state is freely manipulated, and it transitions from a consistent state to an inconsistent one and then (hopefully) back again with surprising regularity.

As developers write code for single-threaded programs, all of this makes sense. It's easy to use shared memory as a sort of scratch pad as you make your way towards a valid and final destination. C-style imperative programming languages have worked this way for quite a few years.

But with the rise of more and more concurrency, you need to pay closer attention to these habits. And you can take a cue from functional programming languages like Haskell, LISP, Scheme, ML, and even F# (a new .NET-compliant language), by adopting immutability, purity, and isolation as first class design concepts.

Immutability

An immutable data structure is one that doesn't change after construction. This is a wonderful property for concurrent programs because if data isn't being mutated, then there is no risk of conflict if many threads access it at once. That means synchronization isn't a concern.

Immutability has some support in C++ by way of `const`, and in C# by way of the read-only modifier. For example, a .NET type that has only read-only fields is shallow immutable. F# creates shallow immutable types by default, unless you use the mutable modifier. Going a step further, if each of those fields itself refers to another type whose fields are all read-only (and only refers to deeply immutable types), then the type is deeply immutable. This results in an entire object graph that is guaranteed to not change out from underneath you, which is very useful indeed.

All of this describes immutability as a static property. It's also possible for objects to be immutable by convention, meaning that there is some guarantee that state won't change during certain periods of time. This is a dynamic property. The Windows Presentation Foundation (WPF) `Freezable` feature implements precisely this, and it also allows concurrent access without synchronization (although it cannot be checked in the same way that static support can). Dynamic immutability is often useful for objects that transition between immutable and mutable throughout the duration of their lifetime.

Immutability does have some downsides. Namely, whenever something has to change, you will need to make a copy of the original object and apply the changes along the way. Additionally, cycles in an object graph are generally not possible (except with dynamic immutability).

For instance, imagine you have an `ImmutableStack<T>`, as shown in Figure 4. Instead of a set of mutating `Push` and `Pop` methods, you will need to return new `ImmutableStack<T>` objects from them that contain the applied changes. Clever tricks can be used (as with the stack) in some cases to share memory among instances.

Figure 4 Using `ImmutableStack`

```
public class ImmutableStack<T> {
    private readonly T m_value;
    private readonly ImmutableStack<T> m_next;
    private readonly bool m_empty;
    public ImmutableStack() { m_empty = true; }
```



```

internal ImmutableStack(T value, Node next) {
    m_value = value;
    m_next = next;
    m_empty = false;
}
public ImmutableStack<T> Push(T value) {
    return new ImmutableStack(value, this);
}
public ImmutableStack<T> Pop(out T value) {
    if (m_empty) throw new Exception("Empty.");
    return m_next;
}
}

```



0

```

`Visual Basic

```



写评论

```

Public Class ImmutableStack(Of T)
    Private ReadOnly m_value As T
    Private ReadOnly m_next As ImmutableStack(Of T)
    Private ReadOnly m_empty As Boolean
    Public Sub New()
        m_empty = True
    End Sub
    Friend Sub New(ByVal value As T, ByVal [next] As Node)
        m_value = value
        m_next = [next]
        m_empty = False
    End Sub
    Public Function Push(ByVal value As T) As ImmutableStack(Of T)
        Return New ImmutableStack(value, Me)
    End Function
    Public Function Pop(<System.Runtime.InteropServices.Out()> ByRef value As T) As ImmutableStack(Of T)
        If m_empty Then
            Throw New Exception("Empty.")
        End If
        Return m_next
    End Function
End Class

```



目录



收藏



微信



微博



QQ

As nodes are pushed, you have to allocate a new object for each one. In a standard linked-list implementation of a stack, you'd have to do this anyway. Notice that as you pop elements from the stack, however, you can use the existing objects. That's because each node in the stack is immutable.

There are already immutable types running rampant in the wild. The CLR's `System.String` class is immutable, and there is a design guideline that all new value types should be immutable, too. The guidance being given here is to use immutability where it is possible and feels natural, and to resist the temptation to perform mutation simply because it is made convenient by the current generation of languages.

Purity

Even with immutable data types, most operations performed by a program are method calls. And method calls can have side-effects, which are problematic in concurrent code because a side-effect implies mutation of some form. Most often this will be a simple write to shared memory, but it can also be a physically mutating operation like a database transaction, Web service invocation, or file system operation. In many circumstances, I'd like to be able to invoke some method without fear that it will lead to concurrency hazards. Good examples of this are simple methods like `GetHashCode` and `ToString` on `System.Object`. Most people wouldn't expect them to entail side-effects.

A pure method can always be run in a concurrent setting without added synchronization. Although there is no common language support for purity, you can define a pure method very simply:

1. It only reads from shared memory, and it only reads
2. It can, of course, write to local variables.
3. It may only call other pure methods.
4. It only reads from shared memory, and it only reads immutable or constant state.
5. It can, of course, write to local variables.
6. It may only call other pure methods.

The set of things possible within a pure method, thus, is extremely limited. But when combined with immutable types, purity can be made possible and convenient. Some functional languages assume purity by default, most notably Haskell where everything is pure. Anything that must perform a side-effect has to be wrapped in a special thing called a monad. Most of us don't use Haskell, however, so we'll have to get by with purity-by-convention.

Isolation

Publication and privatization were mentioned in passing earlier, but they strike at the heart of a very important issue. Synchronization is necessary—and immutability and purity are interesting—because state is ordinarily shared among multiple threads. But if state is confined within a single thread, then there is no need for synchronization. This leads to more naturally scalable software.

Indeed, if state is isolated, mutation can happen freely. This is convenient because mutation is a fundamental built-in aspect of most C-style languages. Programmers are accustomed to it. It takes discipline to program in a mostly functional style, and is quite difficult for most developers. So give it a try, but don't deceive yourself into thinking the world will become functional overnight.

Ownership is a difficult thing to track down. When does an object become shared? When it is initialized, that is being done by a single thread and the object itself is not reachable from other threads just yet. Once a reference to that object is stored in a static variable, some place that has been shared at thread creation or queuing time, or a field of an object transitively reachable from one of these places, the object becomes shared. It is imperative that developers watch specifically for these transitions between private and shared, and treat all shared state with care.

Joe Duffy is the Development Lead for Parallel Extensions to .NET at Microsoft. He spends most of his time hacking code, overseeing the design of the library, and managing an amazing team of developers. His new book is Concurrent Programming on Windows.

原链接失效：http://msdn.microsoft.com/en-us/magazine/cc817398.aspx

个人分类：多线程

相关热词：并发并发 并发与并发 was并发 并发 强并发和弱并发

上一篇ZeroMQ

下一篇ZeroMQ(java)之Router与Dealer运行原理

想对作者说点什么？

我来说两句

另一种修改在线SWF的方法

这次公开的这种方式有一定的局限性，通过努力可以克服。它也有一定的优越性。与以前所用的内存搜索和局部修改不同，这种方式...

1479

ScheduledExecutorService创建newScheduledThreadPool线程池遇到的问题

最近线程池老是遇到执行两次的情况，故作了分析，情况如下 package test; import java.util.Date; import java.util.concurrent.Execu...

1万

由不同编号生成策略产生的多线程问题及解决 - CSDN博客

最近复习Java多线程时,看到"生产者消费者问题"——这是个多线程并发访问的经典案例,操作系统知识中也讲到过,详细内容就不在此列出了,如果有不明的,可以参考...

5-24

解决多线程并发问题 - CSDN博客

使得每个线程在某一时间访问到的并不是同一个对象,这样就隔离了多个线程对数据...public class SequenceNumber { // ①通过匿名...

7-31

欢迎咨询 友达科技铸造

百度广告



第一次分析 草稿

qq下载地址：http://mobile.qq.com/android/ 用到的工具：wireshrk 改之理 dex2jar jd-gui 一、在手机qq上执行了一次聊...

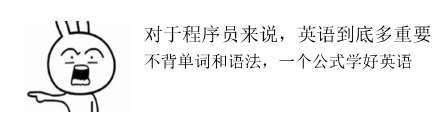
2036

多线程中并发问题 - CSDN博客

多线程的理解更深了,下面是我发的问题和在问题解决...but i am confused with the sequence during i ...访问: 25万+ 积分: 6118 排名: 5513 勋章...

7-22

<div>Spring并发访问的线程安全性问题 - CSDN博客</div> <div>Java Web并发访问的线程安全问题 多线程环境下如果访问单例对象,当对象内部有类...sequence 1篇 soap 1篇 Quartz 2篇 logback 1篇 SLF4J 1篇 mail 1...</div>	7-3
<div>关于自定义TitleBar不能完全显示的问题</div> <div>看了好几篇，自定义TitleBar不能完全显示的解决方案，都不是太全，附上我自己的： 第一步： requestWindowFeature(Window.F...</div>	<div><div></div><div>502</div></div>
<div>Android权限之sharedUserId、签名（实例说明）</div> <div>1. 概述： 权限sharedUserId 与 LOCAL_CERTIFICATE ，在某些时候（较少）需要搭配使用。但若搭配错误，将带来令人费解的...</div>	<div><div></div><div>1.1万</div></div>
<div>ThreadLocal 那点事儿 - CSDN博客</div> <div>一个序列号生成器的程序,可能同时会有多个线程并发访问它,要保证每个线程得到的...ClientThread thread1 = new ClientThread(sequence); ClientThread thread2 = ...</div>	<div><div><div>0</div><div>写评论</div><div>目录</div><div>收藏</div><div>微信</div><div>微博</div><div>QQ</div></div><div>6-7</div></div>
<div>理解多线程 - CSDN博客</div> <div>本实例针对多线程技术在应用中经常遇到的问题,如线程...不同的是它可以同一时刻允许多个线程访问同一个...一个32位的控制台程序,将该程序命名为"sequence"...</div>	<div><div>6-20</div></div>
<div>WIFI万能钥匙协议分析</div> <div>转：https://my.oschina.net/auo/blog/338168 WIFI万能钥匙协议分析 作者：enimey 时间：2014.10.28 0x0 前段时间实验室需求，分...</div>	<div><div><div>2447</div></div></div>
<div>多线程的那点事儿 - CSDN博客</div> <div>多线程创建其实十分简单,在windows系统下面有很多函数可以创建多线程,比如说_begin...当然,事物都有其两面性。这种对公共资源的访问模式也会导致一些问题。什么问题呢...</div>	<div><div>6-16</div></div>
<div>ThreadLocal:多线程共享资源安全访问新思路 - CSDN博客</div> <div>// 代码清单2 SequenceNumber public class SequenceNumber { //1,通过匿名内部...1,相同点:ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题...</div>	<div>5-22</div>
<div>Android多个APK共享数据（ Shared User ID ）</div> <div>Android给每个APK进程分配一个单独的用户空间,其manifest中的userid就是对应一个Linux用户 (Android 系统是基于Linux的). 所以...</div>	<div><div></div><div>607</div></div>
<div>并发危险 [解决多线程代码中的 11 个常见的问题]</div> <div>2012年06月03日 84KB <div>下载</div></div>	<div><div>DOC</div></div>
<div>解决oracle自增长sequence失效的问题 - CSDN博客</div> <div>如果oracle程序没有按照hibernater设置的sequence自增长...等级: 访问: 1037万+ 积分: 8万+ 排名: 19 ...多线程 45篇 算法37篇 编码19篇 分页2篇 ...</div>	<div>6-10</div>
<div>网络编程三---多线程/进程解决并发问题</div> <div>前文列出的代码给大家展示了一个最简单的网络程序，但正如文章末尾所提的，这个最简单的网络程序最大的缺点是服务端一次只能...</div>	<div><div></div><div>2667</div></div>
<div>并发处理中的问题以及解决这些问题的并发模型</div> <div>单机并发是集群并发的基础。本文主要将单机并发问题，和解决单机并发问题解决模型。...</div>	<div><div></div><div>1791</div></div>
<div>JAVA多线程不安全问题解决方案（多线程并发同一资源）。</div> <div>引例：吃苹果比赛，3个人同时吃50个苹果，谁先拿到谁就吃，每个哦ing过都有编号。 问题： 多线程同时执行的时候可能出现不安...</div>	<div><div></div><div>2525</div></div>
<div>C++学习之路（19）--详解C++函数重载</div> <div>写在前面： 函数重载的重要性不言而喻，但是你知道C++中函数重载是如何实现的呢（虽然本文谈的是C++中函数重载的实现，但...</div>	<div><div></div><div>137</div></div>
<div>Android sharedUserId研究记录</div> <div>签名简介： 在Android 系统中，所有安装到系统的应用程序都必有一个数字证书，此数字证书用于标识应用程序的作者和在应用程序...</div>	<div><div></div><div>291</div></div>



对于程序员来说，英语到底多重要
不背单词和语法，一个公式学好英语

查看apk文件中的布局文件源代码

由于apk中的XML布局文件是经过编译处理的，无法直接阅读。因此，需要使用反编译工具处理后再阅读这些文件，例如，可以先将...

多线程并发问题

0. 前言 转载请注明出处：http://blog.csdn.net/seu_calvin/article/details/52370068 面试时很可能遇到这样一个问题：使用volati...

项目中怎么控制多线程高并发访问。

synchronized关键字主要解决多线程共享数据同步问题。 ThreadLocal使用场合主要解决多线程中数据因并发产生不一致问题。 Thr...

JAVA高并发多线程必须懂的50个问题

http://www.importnew.com/12773.html ImportNew 首页所有文章资讯Web架构基础技术书籍教程Java小组工具资源 Java线程面试题...

面试常问问题：银行网上支付项目中怎么控制多线程高并发访问？

银行网上支付项目中怎么控制多线程高并发访问？

家用单反相机哪款好

单反

Java并发：生产者与消费者经典问题（马士兵）

package houy.qing.demo; public class ProducerSummer { public static void main(String[] args) {...

电商实例、业务并发、网站并发及解决办法

电商实例、业务并发、网站并发及解决方法 一、怎么防止多用户同一时间抢购同一商品，防止高并发同时下单同一商品 最近在...

个人资料

明潮

关注

原创

63

粉丝

13

喜欢

11

评论

12

等级： 博客 4 访问： 6万+

积分： 1677 排名： 3万+



最新文章
[快速开发框架二、One框架](#)
[快速开发框架一、XUtils](#)
[OkHttp+Retrofit+Rxjava](#)
[Android快速开发系列 10个常用工具类](#)
[Android 基于libaums实现读写U盘文件](#)

个人分类

android	122篇
Android 5.1 Framework	3篇
Android 6.0 Framework	2篇
C++/C	147篇
java	5篇
展开	

我的栏目

欢迎关注我的原创公众号：



[github地址](#) [每天进步一点](#) 推荐网站：[RGB](#)
[颜色查询对照表](#)

归档

2018年8月	3篇
2018年7月	9篇
2018年6月	31篇
2018年5月	49篇
2018年4月	19篇
展开	

热门文章

[SecureCRT下载安装注册破解](#)
阅读量：7998

[加密jdbc配置文件中的用户名密码](#)
阅读量：5278

[如何分析安卓系统日志](#)
阅读量：2709

[嵌入式没前途，做app更吃香](#)
阅读量：2445

[VS2013 64位 32位|Microsoft Visual Studio Ultimate 2013 \(简体中文旗舰版 \)](#)
阅读量：1896

最新评论

[C语言字符串函数大全及范例](#)
u010144805：[reply]qq_41443458[reply] 两个字
字符串自左向右逐个字符相比（按ASCII...



0



写评论



目录



收藏



微信



微博



QQ

[mbedTLS常用结构体](#)

shi3689476：你好，能分享下SM2 的整个代码吗？

[multiple Lua VMs ...](#)

u011479494：潮升？

[国密杂凑算法SM3](#)

qq_39187545：初始值，这个什么意思

[Android DropboxMa...](#)

u010144805：V/ActivityManager(785): Broadcas
t: Intent { act...

[联系我们](#)

请扫描二维码联系客服

✉ webmaster@csdn.net

☎ 400-660-0108

👤 QQ客服 🗨 客服论坛

[关于](#) [招聘](#) [广告服务](#) [网站地图](#)

©2018 CSDN版权所有 京ICP证09002463号

🔍 百度提供支持

经营性网站备案信息

网络110报警服务

中国互联网举报中心

北京互联网违法和不良信息举报中心



0



写评论



目录



收藏



微信



微博



QQ