链滴

《深入理解 Java 虚拟机》读书笔记：线程安全与锁优化

作者：jingqueyimu

原文链接：https://ld246.com/article/1586274333221

来源网站：链滴

<h2 id="正文">正文</h2>
<h2 id="一-线程安全">一、线程安全</h2>
当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结，那这个对象是线程安全的。
<h3 id="1-Java-语言中的线程安全">1、Java 语言中的线程安全</h3>
按线程安全的“安全程度”由强至弱排序，可以将多个线程的共享数据分为 5 类：不可变、绝对程安全、相对线程安全、线程兼容和线程对立。
<h4 id="-1-不可变">（1）不可变</h4>
不可变的对象一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再采取任的线程安全保障措施。

如果共享数据是一个基本数据类型，那么只要在定义时使用 final 修饰就可以保证它是不可变的
如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响。最简单的方法是把对象中带有状态的变量都声明为 final，这样在构造函数结束之后，它就是不可变的。

<h4 id="-2-绝对线程安全">（2）绝对线程安全</h4>
必须满足“不管运行时环境如何，调用者都不需要任何额外的同步措施”。
实现代价非常大。
<h4 id="-3-相对线程安全">（3）相对线程安全</h4>
就是我们通常意义上所讲的线程安全，它需要保证对一个对象单独的操作是线程安全的，调用时需要额外的保障措施，但是对于一些特定顺序的连续调用，可能需要在调用端使用额外的同步手段来证调用的正确性。
Java 语言中，大部分的线程安全类都属于这种类型。
<h4 id="-4-线程兼容">（4）线程兼容</h4>
指对象本身并不是线程安全的，但可以通过在调用端正确地使用同步手段来保证对象在并发环境可以安全地使用。
我们平常说一个类不是线程安全的，绝大多数时候指的是这一种情况。
<h4 id="-5-线程对立">（5）线程对立</h4>
指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。
Java 语言天生就具备多线程特性，线程对立这种排斥多线程的代码是很少出现的。
<h3 id="2-线程安全的实现方法">2、线程安全的实现方法</h3>
<h4 id="-1-互斥同步-阻塞同步-">（1）互斥同步（阻塞同步）</h4>
同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一个（使用信号量时以是多个）线程使用。而互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现式。
从处理问题的方式上说，互斥同步属于一种悲观的并发策略，总是认为只要不去做正确的同步措（例如加锁），那就肯定会出现问题。
synchronized：
synchronized 关键字是 Java 语言最基本的互斥同步手段，经过编译后，它会在同步块的前后分形成 monitorenter 和 monitorexit 两个字节码指令，这两个字节码都需要一个 reference 类型的参来指明要锁定和解锁的对象。

synchronized 同步块对同一条线程来说是可重入的，不会出现自己把自己锁死的问题。
synchronized 同步块在已进入的线程执行完之前，会阻塞后面其他线程的进入。

ReentrantLock（重入锁）：
java.util.concurrent 包中的 ReentrantLock 也可以用来实现同步，但相比 synchronized，ReenrantLock 增加了一些高级功能：

等待可中断：当持有锁的线程长期不释放锁时，正在等待的线程可以选择弃等待，改为处理其他事情。
可实现公平锁：多个线程在等待同一个锁时，必须按照申请锁的时间顺序

依次获得锁。synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但可以过带布尔值的构造函数要求使用公平锁。
锁可以绑定多个条件：一个 ReentrantLock 对象可以同时绑定多个 Condition 对象。而在 synchronized 中，锁对象的 wait() 和 notify()（或 notifyAll()）方法只能实现一个含的条件。

<h4 id="-2-非阻塞同步">（2）非阻塞同步</h4>
非阻塞同步是一种乐观的并发策略，它在进行同步操作时，不需要把线程挂起，而是先进行操作如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采取他的补偿措施（最常见的补偿措施就是不断地重试，直到成功为止）。
CAS（比较并交换）：
CAS 指令有 3 个操作数：变量的内存地址 V、旧的预期值 A、新值 B。当且仅当内存地址 V 的与预期值 A 相等时，将内存地址 V 的值更新为新值 B，否则不执行更新。上述的处理过程是一个原操作。
CAS 的 ABA 问题：
如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然为 A 值，CAS 操会认为它从来没有被改变过。但实际上，在这段期间它的值有可能曾经被改成了 B，后来又被改回为。这个漏洞称为 CAS 操作的“ABA”问题。
<h4 id="-3-无同步方案">（3）无同步方案</h4>
如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性，因此会有一代码天生就是线程安全的。
可重入代码（纯代码）：
如果一个方法，它的返回结果是可以预测的，只要输入了相同的数据，就都能返回相同的结果，它就满足可重入性的要求，当然也就是线程安全的。
可重入代码有一些共同的特征，例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量由参数中传入、不调用非可重入的方法等。
线程本地存储：
如果能保证使用共享数据的代码在同一个线程中执行，那么就可以把共享数据的可见范围限制在一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题。
<h2 id="二-锁优化">二、锁优化</h2>
<h3 id="1-自旋锁与自适应自旋">1、自旋锁与自适应自旋</h3>
如果物理机有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求的那个线程“稍等一下”，但不放弃处理器的执行时间，然后看看持有锁的线程是否很快就会释放锁为了让线程等待，我们只需让线程执行一个忙循环（自旋），这就是所谓的自旋锁。
自适应自旋：
通过前一次在同一个锁上的自旋时间以及锁的拥有者的状态来确定自旋的时间。如果在同一个锁象上，自旋等待刚刚成功获得过锁，那么虚拟机将允许自旋等待持续相对更长的时间。如果对于某个，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。
<h3 id="2-锁消除">2、锁消除</h3>
锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享据竞争的锁进行消除。
锁消除的主要判定依据来源于逃逸分析的数据支持，如果判断在一段代码中，堆上的所有数据都会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同加锁自然就无须进行。
<h3 id="3-锁粗化">3、锁粗化</h3>
如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中的，那虚拟机将会把加锁同步的范围扩展（粗化）到整个操作序列的外部，这样只需要加锁一次就可以了。
<h3 id="4-轻量级锁">4、轻量级锁</h3>
轻量级锁并不是用来代替重量级锁的，而是在没有多线程竞争的前提下，减少传统的重量级锁使操作系统互斥量产生的性能消耗。
对象头的 Mark Word 实现轻量级锁和偏向锁的关键。Mark Word 有一个 2bit 的锁标志位，用标识对象的锁状态。

轻量级锁的加锁过程：

进入同步块时，如果同步对象没有被锁定（锁标志位为“01”状态），虚拟机将在当前线程的栈中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的 Mark Word 拷贝。
虚拟机使用 CAS 操作将对象的 Mark Word 更新为指向 Lock Record 的指针。
如果 CAS 操作成功，那么线程就拥有了该对象的锁，并且对象 Mark Word 的锁标志位将转变“00”，表示此对象处于轻量级锁定状态。
如果 CAS 操作失败，虚拟机会检查对象的 Mark Word 是否指向当前线程的栈帧，如果是说明前线程已经拥有了该对象的锁，那就可以直接进入同步块继续执行，否则说明这个锁对象已经被其他程抢占了。
如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁，锁标志的状值变为“10”，Mark Word 中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要入阻塞状态。

轻量级锁的解锁过程：

如果对象 Mark Word 仍然指向线程的锁记录，那就用 CAS 操作把对象的 Mark Word 更新为程中的 Mark Word 拷贝。
如果 CAS 操作成功，整个同步过程就完成了。
如果 CAS 操作失败，说明有其他线程尝试过获取该锁，那就要在释放锁的同时，唤醒被挂起的程。

<h3 id="5-偏向锁">5、偏向锁</h3>
如果说轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在竞争的情况下把整个同步都消除掉，连 CAS 操作都不做了。
偏向锁是指这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他程获取，则持有偏向锁的线程将永远不需要再进行同步。
偏向锁锁的执行过程：

锁对象第一次被线程获取时，虚拟机把对象 Mark Word 中的标志位设为“01”，即偏向模式。/li>
使用 CAS 操作把获取到这个锁的线程 ID 记录在对象的 Mark Word 中。
如果 CAS 操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都不再进任何同步操作。
当有另外一个线程尝试获取这个锁时，则结束偏向模式。根据锁对象目前是否处于被锁定的状态撤销偏向后恢复到未锁定（标志位为“01”）或轻量级锁定（标志位为“00”）的状态。

<h2 id="结语">结语</h2>
妈呀！终于写完了~
讲道理，当初花三个月把这本书啃完的时候，本来还打算至少在过年的时侯把笔记整理完的。没想，只因为看了一眼《半小时漫画中国史》，顿时就觉得硬邦邦的技术书籍一点也不香了。直到我看了三本《半小时漫画中国史》才恍然惊醒——
唉？我只是个假的文艺青年啊，看这种书不太好吧？身为假程序员的我，真是羞愧！
后来呢，因为疫情的原因，过年的假期延长了一周。那段时间我就一直在想：我是不是忘记了什？然而，因为我告诉我：做任何事情都要专注，开黑的时候不能分心。我觉得我说得挺有道理的，所我也就没太深究。后来我才知道，原来我得了失忆症，这种失忆症的名字叫做“事后自我欺骗型失忆”。
唉，真是羞愧！