1. Java并发的底层原理
2. 概述

Java使用的并发机制依赖于JVM的实现和CPU的指令。

1. volatile
2. 概述

volatile是轻量级的锁，保证了共享变量的可见性，它的使用成本比synchronized要低，因为它不会引起线程上下文的切换和调度。

1. CPU术语

内存屏障：一组处理器命令，用于实现对内存操作的顺序限制；

缓冲行：CPU高速缓存中可以分配的最小存储单位，CPU在填写缓冲行时会加载整个缓存行；

缓存行填充：当CPU识别到从内存中读取操作数是可缓存的，CPU读取整个高速缓存行到适当的缓存中（L1，L2，L3）；

1. volatile原理

volatile是如何保证可见性的？

对于有volatile修饰的共享变量会多出一行汇编代码，Lock前缀的指令，该指令在多核CPU中会引发两件事：

1. 将当前CPU缓存行的数据写回到系统内存中；
2. 这个写回内存的操作会使其他CPU缓存了该内存地址的数据无效；

理解：CPU为了提高处理速度，会先将内存中的数据读到CPU的缓存中，对于多核CPU就会读到每个CPU的缓存中，当对声明了volatile的变量进行写操作时，JVM会向CPU发送一条Lock前缀的命令，将这个变量所在的缓存行的数据写回系统内存中。但是此时其他CPU的缓存中依然是旧值，再执行计算的话就会有问题，为了保证各个CPU的缓存一致，出现了缓存一致性协议，该协议会阻止其他的CPU修改被多个CPU缓存的内存地址的数据。 每个CPU通过嗅探在总线上传播的数据来检查自己缓存的值是否过期，当CPU发现自己缓存行对应的内存地址被修改了，就会设置当前CPU的缓存行为无效状态，当CPU对这个数据进行修改操作时，会重新从系统内存中将数据读取到该CPU的缓存中（缓存行填充）。

1. volatile的使用优化

在JDK7中新增了一个队列集合类LinkedTransferQueue，它在使用volatile变量时，用一种追加字节的方式优化队列出队和入队的性能。它将共享变量的字节数追加到64个字节，对于高速缓存行是64个字节的高速缓存行来说，不支持部分填充缓存行，如果队列的头结点和尾结点不足64字节的话，CPU会将它们都读入同一个高速缓存行中，在多CPU下每个CPU会缓存相同的头，尾结点，当一个CPU视图修改头节点时，会将整个缓存行锁定，那么在缓存一致性机制的作用下，其他CPU不能访问自己高速缓存的尾结点，而队列的出队和入队会修改头尾结点，影响效率。因此使用追加到64个字节的方式来填满高速缓存行，避免头结点和尾结点加载到同一个缓存行中，使头尾节点在修改时不会相互锁定。

1. synchronized
2. 概述

synchronized是一种重量级锁，因为在获得锁和释放锁的过程中性能消耗很大。

1. 对象锁

对于普通同步方法，锁是当前的实例对象；

对于静态同步方法，锁是当前类的Class对象；

对于同步方法块，锁是Synchronized括号中的对象；

当一个线程试图访问同步代码块时，首先必须获得锁，退出或抛出异常时必须释放锁。JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，代码块同步和方法同步都可以使用monitorenter和moniterexit指令完成。一个在编译后插入开始位置，一个插入方法结束和异常位置，这两个指令必须是配对出现的。每一个对象都有一个monitor与之关联，当一个monitor被持有后，将处于锁定状态，线程执行到monitorenter指令时，尝试获取对象对应的monitor所有权，即获得对象锁。

1. 锁的升级

为了减少获得锁和释放锁的性能消耗，引入了偏向锁和轻量级锁，锁一共有4个状态，级别从低到高是无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，随着竞争情况逐渐升级，锁只能升级不能降级。

1. 偏向锁
2. 偏向锁的获取

大多数情况下，锁不存在多线程竞争，且总是由同一线程获得，为了让线程获得锁的代价更低引入了偏向锁。当一个线程获得锁时，会在对象头和栈帧中的锁记录中存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需测试对象头的Mark Word里是否存储着指向当前线程的偏向锁即可。如果测试成功，说明该线程已经获得了锁，如果测试失败，则需要再测试Mark Word中偏向锁的标识是否设置为偏向锁1，如果没有设置，则使用CAS竞争锁，如果设置了则用CAS将对象投的偏向锁指向当前线程。

1. 偏向锁的撤销

偏向锁只有在其他线程来竞争该锁时才会释放锁，撤销只有在等待到全局安全点（这个时间点没有正在执行的字节码）的时候，才会暂停拥有偏向锁的线程，然后检查该线程是否存活，如果不存活，则将对象头设置为无锁状态。

1. 关闭偏向锁

偏向锁在Java6和Java7中是默认启动的，但是是有延迟的，一般是几秒钟，必要时可以使用JVM参数关闭延迟：-XX:BiasedLockingStartupDelay=0。关闭偏向锁的JVM参数为：-XX:-UseBiasedLocking=false，则会进入轻量级锁状态。

1. 轻量级锁
2. 加锁

线程在执行同步块之前，JVM会先在当前线程的栈帧中创建用于存储所记录的空间，并将对象头的Mark Word复制到锁记录中，为Displaced Mark Word，然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针，如果成功，则当前线程获得锁，否则其他线程竞争锁，当前线程尝试使用自旋来获得锁。

1. 解锁

轻量级锁解锁时，使用原子CAS操作将Displaced Mark Word替换到对象头中，如果成功，则表示没有竞争，如果失败，表示当前锁存在竞争，锁就会膨胀 为重量级锁，然后释放锁，唤醒等待的线程。因为自旋会消耗CPU，为了避免无用的自旋，一旦轻量级锁升级为重量级锁，无法恢复。

1. 原子操作的实现
2. CAS

比较并交换（CAS）：需要输入两个数值，一个是旧值，一个是新值，在操作期间如果旧值没有变化，则将旧值变换为新值，否则不交换；

1. 处理器实现原子操作

通过总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

1. 总线锁定

如果多个CPU同时对共享变量进行修改操作，则某个CPU会发送一个#Lock信号锁定总线，其他CPU的请求将会被阻塞，则该CPU可以独占共享内存。

1. 缓存锁定

同一时刻，其实我们只需要保证对某个内存地址的操作是原子性的即可，但是总线锁定将CPU和内存之间的通信都锁住了，性能开销过大。目前的处理器使用缓存锁定来优化。内存区域如果被缓存在CPU的缓存行中，并且在Lock期间被锁定，那么当它执行锁操作回写到内存时，CPU不会发送#Lock信号，而是修改内部的内存地址，允许缓存一致性机制来保证操作的原子性，当其他CPU回写已经被锁定的缓存行的数据时，会使缓存行无效。

1. Java中的原子操作

Java使用锁和循环CAS来实现原子操作，自旋CAS的思路就是循环进行CAS操作直到成功为止，从JDK1.5开始，并发包中提供了一些原子类，如AtomicInteger, AtomicLong等，还有一些工具类，如自增操作。

1. CAS实现原子操作的问题

CAS虽然高效地解决了原子操作，但是仍然存在三个问题。

1. ABA问题

CAS检测的是旧值是否发生了变化，如果旧值A先变为B，再变为A，此时检测结果是没有变，但是实际上已经改变了2次。因此在变量前面追加版本号，每次变量更新时将版本号+1，那么就会变为1A -> 2B -> 3C，问题解决。AtomicStampedReference类解决了这个问题，其中的compareAndSet方法首先检查当前应用是否等于预期的引用，然后检查版本号。

1. 循环时间过长

自旋CAS的自旋操作会对CPU造成较大的影响。

1. 只能保证一个共享变量的原子操作

循环CAS无法对多个共享变量进行操作，这是只能用锁。或者将多个变量合并为一个共享变量来操作，AtomicReference类保证了引用对象之间的原子性，就可以将多个变量放在一个对象中进行CAS操作。