JAVA 并发编程的艺术

一、volatile 的两条实现原则
1、lock 前缀指令会引起处理器缓存回写到内存
2、一个处理器的缓存回写到内存会导致其它处理器的缓存无效

二、为什么追加到64字节能提高并发编程效率
多种处理器的L1/L2/L3缓存的高速**缓存行**是64字节，当处理器修改行头时，会将整个缓存行都锁住。空间换时间

三、偏向锁
当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储偏向的线程ID，该线程进出同步块时不需要进行CAS操作来加锁解锁。

四、轻量级锁
线程同步前，JVM会在线程栈帧中创建存储锁记录的空间，并将对象头中的Mark Word 复制到锁记录中。线程尝试用CAS将对象头中的Mark Word 替换为指向锁记录的指针（Displaced Mark Word）。如果成功，则获得锁。
解锁时，CAS会将Displaced Mark Word 换回到对象头。若失败，则锁会升级为重量级锁（不可降级）

五、原子操作实现原理
1、使用总线锁保证原子性：处理器提供一个LOCK#信号，当一个处理器在总线上输出此信号，其它处理器请求将被阻塞
2、通过缓存锁保证原子性：内存区域如果被缓存在处理器缓存行中，并且上锁时，a、当它回写内存时，会修改内部的内存地址，使缓存行无效（其它处理器的缓存则无效）b、cpu1修改了缓存行中X使用了缓存锁定，则CPU2就不能同时缓存X的缓存行

六、指令重排序
1、编译器优化重排：编译器在不改变单线程程序语义的前提下，可重排序
2、指令级并行重排：现代处理器采用了指令级别并行技术可以改变指令执行顺序
3、内在系统的重排：由于处理器使用缓存和读/写缓冲区，使得加载和存储操作看上去可能是乱序

源代码 ---> 编译器重排 ---> 指令级重排 ---> 内存系统重排 ---> 最终执行指令

七、内存屏障类型
1、LoadLoad Barriers：Load1；LoadLoad；Load2
2、StoreStore Barriers：Store1；StoreStore；Store2
3、LoadStore Barriers：Load1；LoadStore；Store2
4、StoreLoad Barriers：Store1；StoreLoad；Load2 同时具体其它3个屏障的效果，该屏障开销很昂贵

八、happens-before (JDK1.5开始)
1、每个操作 happens-before 该线程中任意后续操作
2、一个锁的解锁 happens-before 该锁的加锁
3、volatile写 happens-before 对于对它的读
4、传递性：A happens-before B B happens-before C 则 A happens-before C

九、volatile
1、可见性。volatile变量的读，总是能看到对这个变量最后的写入
2、原子性。对任意单个volatile变量的读、写具有原子性，但类似volatile++这种复合操作不具有原子性

volatile 变量的写-读与锁的释放-获取有相同的内存语义。写和锁的释放、读和锁的获取有相同的内存语义
写的内存语义：JMM会把该线程对应的本地内存中的共享变量值刷新到主内存
读的内存语义：JMM会把该线程对应的本地内存置为无效，再从主内存读取共享变量
内存主义 实现：在每个读、写操作后插入一个内存屏障

十、锁的内存语义
获取锁时：JMM会把该线程对应的本地内存中的共享变量值刷新到主内存
释放锁时：JMM会把该线程对应的本地内存置为无效，再从主内存读取共享变量

volatile/CAS 在锁的实现中，起了重要作用
CAS（compareAndSet）同时具有volatile读和写的内存语义

十一、线程状态
NEW： 初始状态，线程被构建，还没调用start()方法
RUNNINT： 运行状态，java线程将操作系统中的就绪和运行两种状态统称 运行中
BLOCKED： 阻塞状态，阻塞于锁
WAITING： 等待状态，需要等待其它线程做出一些特定动作，通知或中断
TIME_WAITING：超时等待，可以在指定的时间自行返回
TERMINATED：终止状态，当前线程已经执行完毕

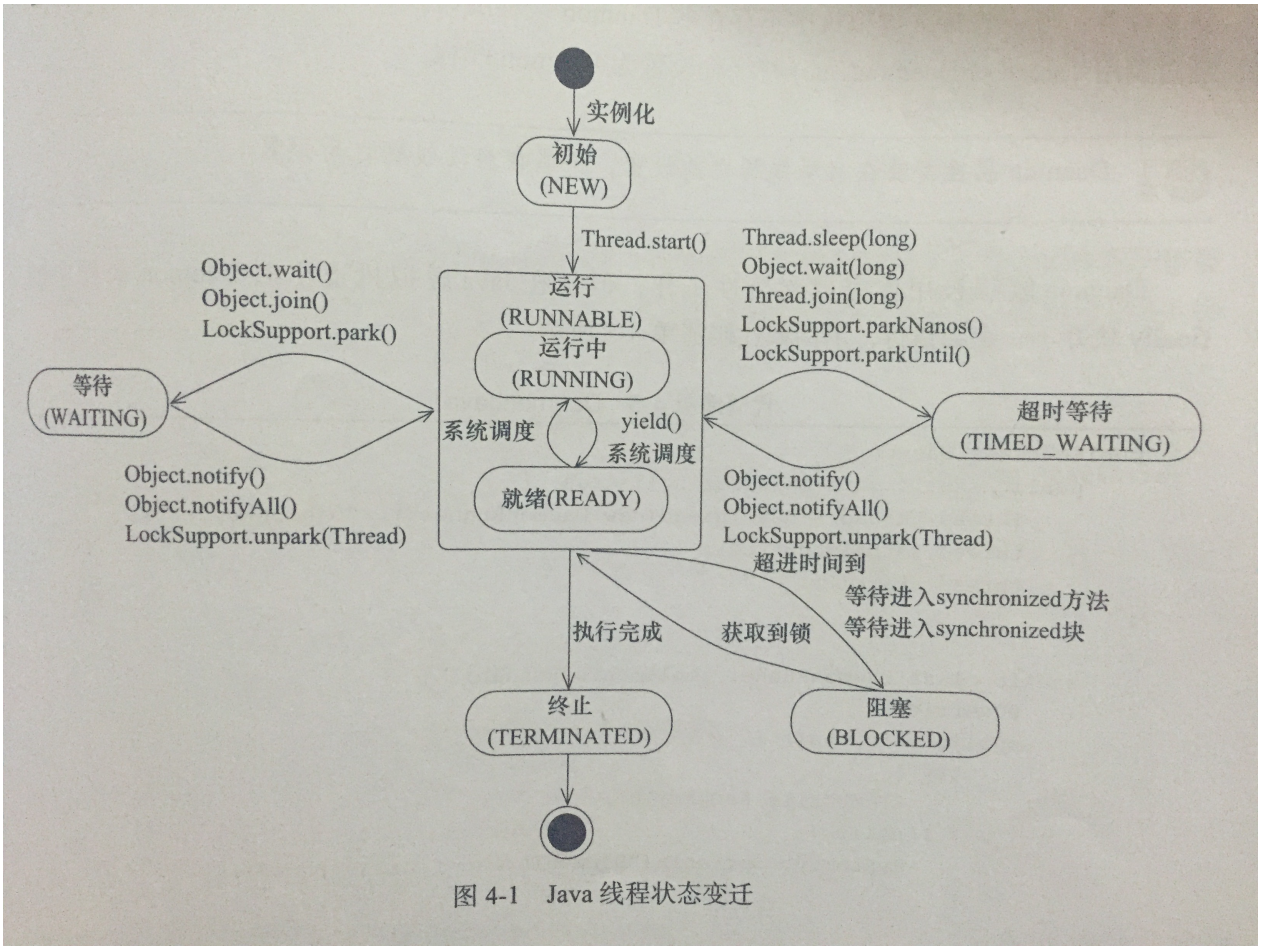

图 4-1 Java 线程状态变迁

十二、理解中断

isInterrupted() 判断是否被中断过

Thread.interrupted() 对当前线程的中断标识位进行复位（清除中断标记）

许多声明抛出InterruptedException的方法，在抛出异常前，java虚拟机会先清除中断标记，此时调用isInterrupted() 返回false

十三、suspend(), resume(), stop()

过期方法，暂停，恢复，停止。过期原因：线程不会释放已经占有的资源（如锁）容易死锁

十四、wait(), notify(), notifyAll()

1、使用wait(), notify(), notifyAll()时需要先对调用对象加锁

2、wait()方法后，线程状态由RUNNING变为WAITING，并将线程放置到等待队列

3、notify(), notifyAll() 调用后，等待线程依旧不会从wait()返回，需要调用方法的线程释放锁后，等待线程才有机会从wait()返回

4、notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列。而notifyAll()将等待队列中所有线程从等待队列中移到同步队列，被移动线程状态出WAITING变为BLOCKED

5、从wait()方法返回的前提是获得了调用对象的锁

线程间数据传输流，传输媒介为内存：PipedOutputStream、PipedInputStream、PipedReader、PipedWriter

前两种面向字节，后两种面向字符

thread.join()：如果一个线程A执行了thread.join()，A要等待thread线程终止后，A线程才能继续执行

十五、synchronized 关键字会隐式地获取锁。Lock 显示的操作锁。

1、队列同步器：面向锁，简化了锁的实现方式。锁和同步器很好的隔离了使用者和锁实现者所需关注的领域。

它使用一个int成员变量表示同步状态，通过内置的FIFO队列完成资源的获取线程的排队工作。当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成一个节点并将其加入同步队列，同时会阻塞当前线程。当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。

2、独占式同步状态获取与释放（同一时，只能有一个线程获取到同步状态）

获取同步状态：成功则返回退出

失败，生成队列节点，CAS设置为节点尾部

自省观察是否为节点头时，自旋获取同步状态

获取到同步状态后，执行相应逻辑退出，唤醒后继节点

十六、重入锁ReentrantLock：排它锁

已经获取到锁的线程，再次调用lock()方法获取锁而不被阻塞。Synchronized隐式 支持重入

公平锁：FIFO 非公平锁：只要CAS设置同步状态成功

读写锁ReentrantReadWriteLock
在同步状态上（一个整型变量）上维护多个读写线程和一个写线程的状态
按位切割使用：高16位表示读，低16位表示写

十七、LockSupport 构建同步组件的基础工具。用于阻塞或唤醒一个线程。
Condition 接口提供了监视器方法，与Lock配合可以实现等待/通知模式

十八、阻塞队列
1、插入：当队列满时，队列会阻塞插入元素的线程，直到队列不满
2、移除：队列为空时，队列会阻塞获取元素的线程，直到队列非空

插入：add 抛出异常，offer 返回特殊值或超时退出，put 一直阻塞
移除：remove 抛出异常，poll 返回特殊值或超时退出，take 一直阻塞

ArrayBlockingQueue: 一个由数组结构组成的有界阻塞队列
LinkedBlockingQueue: 一个由链表结构组成的有界阻塞队列
PriorityBlockingQueue: 一个支持优先级排序的无界阻塞队列
DelayQueue: 一个使用优先级队列实现的界阻塞队列
SynchronousQueue: 一个不存储元素的阻塞队列
LinkedTransferQueue: 一个由数组结构组成的无界阻塞队列
LinkedBlockingDueue: 一个由链表结构组成的双向阻塞队列

十九、Fork/Join框架
把大任务分割成小任务，最终汇总每个小任务结果后得到大任务结果的框架
工作窃取算法，干完活的线程去其它队列里领取任务（从队尾领取），充分利用资源

二十、并发中的工具类
CountDownLatch:传入一个N，等待N个点完成，countDown方法N会减1，await方法会阻塞当前线程，直到N变成零。计数器只能使用一次，不能重置，可用于结果等待汇总。
CyclicBarrier:让一组线程到达一个屏障时被阻塞，直到最后一个屏障时，所有屏障才会被打开 调用 await 方法。计数器能重置，可用于结果等待汇总。
Semaphore:控制并发线程数量。 acquire 得到后可执行，release 释放资源
Exchange:两个线程交换彼此的数据。一个线程先执行exchange 它会等待另一个线程也执行exchange方法交换数据