Java内存模型与线程

BY CSDN
https://blog.csdn.net/lxiansheng001

12.1硬件的效率与一致性

- 增加高速缓存；为了使得处理器内部的运算单元能尽量被充分利用，处理器可能会对代码进行乱序执行优化，处理器会在计算机之后将乱序执行的结果重组，保证该结果与顺序执行结果是一致的
- JVM的即时编译器也有类似的指令重排序优化
- 永不止步

12.3 Java内存模型

- Java虚拟机规范了一种内存模型来屏蔽掉各种硬件和操作系统的内存访问差异，以实现Java语言在各个平台都能达到一致的内存访问效果
- 12.3.1主内存和工作内存
  - Java内存模型的主要目的是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量和Java编程中所说的变量有所区别，包括实例变量、类变量以及数组对象的元素，但不包括局部变量和方法参数，后者是线程私有，不会被共享，自然不存在竞争的问题
    - 实例对象里面
      成员方法的本地变量为基本数据类型：工作内存栈帧
      成员方法的本地变量为引用类型：引用在工作内存栈帧，对象实例在主内存
      成员变量不管基本还是引用类型：都存在堆区（主内存）
      static变量以及类本身相关的信息将会存在主内存中
  - Java内存模型规定了所有的变量都存储在主内存中（可以类比硬件主内存，但这是虚拟机内存的一部分），每条线程还有自己的工作内存（可以类比高速缓存），线程的工作内存保存了被该线程使用到的变量的主内存副本拷贝。
  - 线程对变量的操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量
  - 不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递都是通过主内存传递
- 12.3.2 内存间交互操作
  - 关于主内存与工作内存交互协议，Java内存模型定义了8中操作完成（每一种操作都是原子的不可再分的 除了long和double）
    - lock(锁定)：作用于主内存变量，把一个变量状态标识一条线程独占的状态
    - unlock(解锁)：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
    - read(读取)：作用于主内存变量，把一个变量的值从主内存传输到工作内存中，以便随后的load动作使用
    - load(载入)：作用于工作内存变量，把read操作从主内存得到的变量值放入工作内存的变量副本中
    - use(使用):作用于工作内存变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令的时候都会执行这个操作
    - assign(赋值):作用于工作内存，把执行引擎返回的结果值赋给工作内存的变量
    - store(存储):作用于工作内存的变量，把工作内存中的一个值传递到主内存中去，以便随后的write操作
    - write(写):作用于主内存变量，把store操作从工作内存中等到的变量值放到主内存的变量中去
  - 对于long和double变量的特殊规则
  - 把一个变量从主内存赋值到工作内存，就要顺序地执行read和load操作；把一个变量从工作内存赋值到主内存，就要顺序地执行store和write操作；只要求顺序执行没有要求连续执行
  - 执行八种操作必须满足的规则
    - 不允许read和load、store和write操作之一单独出现，即不允许一个变量从主内存读取了，在工作内存不接受，或者从工作内存发起回写，主内存不接受的情况
    - 不允许一个线程丢弃掉它最近的assign操作，即变量在工作内存改变以后必须把该变化同步到主内存
    - 不允许一个线程无原因的（没有发生过任何assign）把数据从工作内存同步到主内存
    - 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或者assign）的变量，换句话说，就是对一个变量实施use、store操作之前，必须先执行过assign和load操作
    - 一个变量在同一时刻只能有一个线程对其进行lock操作，但是lock操作可以被同一线程重复执行多次，多次lock执行后，只有执行相同次数的unlock操作，变量才会被解锁
    - 如果对变量执行lock操作，将会清空工作内存此变量的值，在执行引擎使用这个变量钱前，需要重新load或者assign初始化变量的值
    - 如果一个变量事先没有被lock，就不允许执行unlock操作，也不允许unlock一个被其他线程lock的变量
    - 对一个变量unlock之前，必须把此变量同步到主内存
- 12.3.3对于volatile型变量的特殊规则
  - Java虚拟机提供的轻量级同步机制
  - 作用
    - 保证变量对所有线程的可见性
      - 不存在一致性问题，但是Java里面的运算并非原子操作，导致volatile变量的运算在并发下一样是不安全的。例如一个语句是由四个指令组成的，在获取时volatile保证了race值是正确的，但是++时其余线程可能已经把值增大了。
      - 写一个volatile时：先写到本地内存，再将本地内存的值刷新到主内存
      - 读一个volatile变量时，将本地内存中的这个变量置为无效，直接去主内存读取
      - 在这个过程中，根据缓存一致性，每个处理器通过嗅探在总线上传播的数据来检查自身缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改时，会重新从系统内存中把数据读到处理器缓存里。
      - volatile只能保证可见性。遇到不符合以下规则还是要使用锁（synchronized或者concurrent包中的原子类）保证原子性
        运算结果并不依赖变量的当前值（set函），或者能够确保只有单一的线程修改变量的值
        变量不需要与其他的状态变量共同参与不变约束
    - 禁止指令重排序
      - 有volatile修饰的变量，复制后多执行了一个lock add操作，这个操作相当于内存屏障（重排序时不能把后面的指令重排序到内存屏障之前的位置）
      - 指令中add $0*0是一个空操作，这种操作相当于对cache中的变量做了一次Java内存模型中所说的“store” “write”操作，所以可以让前面volatile变量的修饰对其他CPU立即可见
  - 特殊规则
    - 每次使用volatile变量前必须先从主内存刷新最新的值（use和load、read动作相关联必须连续一起出现）
    - 工作内存中修改volatile变量后必须同步到主内存中（只有线程对变量后一个动作时store时，线程才能对变量执行赋值动作；assign和store、write动作相关联，必须连续一起出现）
    - volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序和代码顺序一致
- 12.3.4原子性、可见性与有序性
  - 原子性
    - Java内存模型保证基本数据类型的访问读写是具备原子性的
    - 在更大范围保证原子性，Java内存模型提供了lock和unlock来满足这种需求，jvm提供了高层次的字节码指令monitorenter和monitorexit来隐式的使用这两个操作（），反映到Java代码中就是同步块synchronized关键字。在关键字之间操作也具备原子性
  - 可见性
    - Java内存模型通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新新变量值这种依赖主内存作为传递媒介实现可见性。普通和volatile都是如此
    - 区别在于：volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新，因此volatile保证了多线程操作时变量的可见性，普通无法做到
    - 同步块synchronized 在对变量执行unlock之前，必须先把变量值同步到主内存中（w、r）
      final 可见性
  - 有序性
    - 在本线程内观察，所有操作都是有序的（线程内表现为串行的语义）；在一个线程中观察另一个线程，所有操作都是无序的（指令重排序和工作内存与主内存同步延迟现象）多线程中会导致操作是无序的。仔细想想。
    - 以上要好好体会 发生的场景是多线程环境，好好想想两个线程 执行对数据造成的影响
    - Java语言提供了volatile和synchronized关键字保证线程之间操作的有序性。volatile本身包含了禁止指令重排序的语义，而s则是由一个变量在同意是个只允许一条线程对其进行lock操作（持有一个锁的两个同步块智能串行进入）
- 12.3.6 先行发生原则
  - 程序次序规则：控制流顺序
  - 管程锁定规则：unlock优先于后面对同一个锁的lock操作（先解锁，在上锁）
  - volatile变量规则：写优先于后面对这个变量的读操作
  - 线程启动规则：线程对象的start()方法先行发生于此线程的每个动作
  - 线程终止规则：线程中所有操作先行于对此线程的终止检测
  - 线程中断规则
  - 对象终结规则
  - 传递性
  - 内存屏障是cpu指令，先行发生原则是JSR-122规范之一；前者是实现手段，后者是最终目的

12.4Java与线程

- 12.4.1线程的实现
  - 使用内核线程实现
    - 直接由操作系统内核支持的线程，内核通过操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器上
    - 程序一般不会直接使用内核线程，而是使用内核线程的高级接口-轻量级进程（线程）
    - 轻量级进程和内核线程是一对一的线程模型
    - 轻量级的优缺点
      - 由于内核线程的支持，轻量级进程称为一个独立的调度单元，一个轻量级在系统阻塞不会影响整个进程继续工作
      - 各种线程操作（创建、同步）都需要进行系统调用，系统调用的代价较高，需要用户态和内核态来回切换
        一个系统支持轻量级进程的数量是有限的
  - 用户线程
    - 部分高性能数据库中的多线程就是由用户线程实现的，进程与线程之间称为一对多的线程模型
    - 实现比较复杂
  - 用户线程加轻量级进程
    - 操作系统提供支持轻量级进程作为用户线程和内核线程之间的桥梁
    - 内核提供线程调度功能及处理器映射
  - Java线程的实现
    - 基于操作系统的原生线程模型；一条Java线程就映射到一条轻量级进程之中（window linux）操作系统支持怎样的线程模型，决定了jvm的线程怎么实现
- 12.4.2 Java线程调度
  - 线程调度是指系统为线程分配处理器使用权的过程
  - 协同式调度
  - 抢占式调度：有系统来分配每个线程的执行时间；线程的执行时间是可控的
- Java语言定义了五种线程状态，在任意一个时间点，一个线程只能有且只有一种状态
- 12.4.3 状态转换

❑ 新建（New）：创建后尚未启动的线程处于这种状态。
❑ 运行（Runable）：Runable 包括了操作系统线程状态中的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。
❑ 无限期等待（Waiting）：处于这种状态的线程不会被分配 CPU 执行时间，它们要等待被其他线程显式地唤醒。以下方法会让线程陷入无限期的等待状态：
- 没有设置 Timeout 参数的 Object.wait() 方法。
- 没有设置 Timeout 参数的 Thread.join() 方法。
- LockSupport.park() 方法。

❑ 限期等待（Timed Waiting）：处于这种状态的线程也不会被分配 CPU 执行时间，不过无须等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒。以下方法会让线程进入限期等待状态：
- Thread.sleep() 方法。
- 设置了 Timeout 参数的 Object.wait() 方法。
- 设置了 Timeout 参数的 Thread.join() 方法。
- LockSupport.parkNanos() 方法。
- LockSupport.parkUntil() 方法。

❑ 阻塞（Blocked）：线程被阻塞了，“阻塞状态”与“等待状态”的区别是：“阻塞状态”在等待着获取到一个排他锁，这个事件将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。
❑ 结束（Terminated）：已终止线程的线程状态，线程已经结束执行。

上述 5 种状态在遇到特定事件发生的时候将会互相转换，它们的转换关系