**\* 第12章 Java内存模型与线程**

衡量一个服务性能的高低好坏，**每秒事务处理数（TPS）**是最重要的指标之一。它代表着**一秒内服务端平均能响应的请求总数**，而TPS值与程序的并发能力又有着非常密切的关系。对于计算量相同的任务，程序线程并发协调得越有条不紊，效率自然就会越高；反之，线程之间频繁阻塞甚至死锁，将会大大降低程序的并发能力。

“高效并发”是本书讲解JVM的最后一部分，将会介绍虚拟机如何实现多线程、多线程之间由于共享和竞争数据而导致的一系列问题及解决方案。

一、硬件的效率与一致性

**高速缓存**

**高速缓存**

**高速缓存**

**缓**

**存**

**一**

**致**

**性**

**协**

**议**

**处理器**

**处理器**

**处理器**

**二、Java内存模型（JMM）**

JVM规范中定义一种Java内存模型来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。

**（1）主内存与工作内存**

JMM的主要目标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。这里的变量（Variables）不同于Java编程中所描述的**变量**，它**包括实例字段、静态字段**和构成**数组对象的元素，**但**不包括局部变量与方法参数，因为后者是线程私有**的，不会被共享，自然就不会存在竞争问题。

JMM规定**所有的变量都存储在主内存**（Main Memory，此处指虚拟机内存的一部分，并非物理硬件的主存）中。**每条线程还有自己的工作内存**（Working Memory），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**，线程对变量的所有操作（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的变量，线程间变量值的传递均需要通过主内存来完成。

需要注意：

（1）如果局部变量是一个reference类型，它引用的对象在Java堆中可被各个线程共享，但是reference本身在Java栈的局部变量表中，它是线程私有的。

（2）另外关于工作内存的副本拷贝，如果线程访问一个对象，只会将该对象的引用、对象中某个在线程访问到的字段进行拷贝，但不会有虚拟机实现成把整个对象拷贝一次。

（3）根据JVM规范规定，volatile变量也是有工作内存的拷贝，但是由于其特殊的操作顺序性规定，所以操作起来如同直接在主内存中读写访问一般。

**类比虚拟机栈部分区域**

**类比Java堆中对**

**象实例数据部分**

**工作内存**

**工作内存**

**工作内存**

**Save**

**和**

**Load**

**操作**

**Java线程**

**Java线程**

**Java线程**

线程、主内存、工作内存三者交互关系

**（2）内存间交互操作**

主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存，如何从工作内存同步回主内存之类的实现细节，JMM定义了以下8种操作来完成，每一种操作都是**原子的**、不可再分的。\_

自JSR-133文档后，已经放弃采用这8种操作去定义Java内存模型的访问协议，因为这种定义相当严谨且十分繁琐，实践起来很麻烦。

lock（锁定）：作用于**主内存的变量**，把一个变量表示为一条线程独占状态。

unlock（解锁）：作用于**主内存的变量**，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它线程锁定。

read（读取）：作用于主内存的变量，把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用。

load（载入）：作用于工作内存的变量，它把之前通过read操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。

assigin（赋值）：作用于**工作内存**的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。

write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

JMM模型规定执行上述8种基本操作必须满足以下规则：

（1）把一个变量从主内存复制到工作内存：read->load；

把变量从工作内存同步回主内存：store->write；

JMM模型只要求上述两个操作必须按顺序执行，成对搭配。但没有保证是连续执行。即read与load之间，store与write之间是可插入其他指令的。

（2）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但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。

（3）不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。

（4）不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中。

（5）一个新的变量只能在主内存中“诞生”，且不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说，就是对一个变量实施use、store操作之前，必须先经过了assign和load操作。

（6）一个变量在同一个时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。

（7）如果对一个变量执行lock操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。

（8）如果一个变量事先没有被lock操作锁定，那就不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定住的变量。

（9）对一个变量执行unlock操作之前，必须先把此变量同步回主内存（store->write）

**（3）对于volatile型变量的特殊规则**

关键字volatile是JVM提供的最轻量级的同步机制。当一个变量被定义为volatile之后，该变量将具备两种特性。

**一是保证此变量对所有线程的“可见性”**。

这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。而普通变量无法做到这一点，因为普通变量的值在线程间传递均需要通过主内存来完成。例如，线程A在工作内存中修改了一个普通变量的值，然后向主内存进行回写，另外一条线程B等线程A回写完成后再从主内存进行读取操作，新变量值才会对线程B可见。

然而，基于volatile变量的运算在并发下并不是安全的。虽然volatile变量在各个线程的工作内存中不存在一致性问题，但是由于Java里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。因此在**不符合以下两条规则**的运算场景中，仍然**需要通过加锁**（使用synchronized或java.util.concurrent中的原子类）来保证原子性。

（1）运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。

（2）变量不需要与其他的状态变量共同参与不变约束。

**二是禁止指令重排序优化。**

普通变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致。

最后总结一下JMM对volatile变量定义的特殊规则：

设T为线程，V和W分别表示两个volatile型变量，那么在进行read、load、use、assign、store和write操作时需要满足如下规则：

（1）只有当线程T对变量V执行的**前**一个动作是**load**时，线程T才能对变量V**后**执行**use**操作；并且，只有当线程T对变量V执行的后一个动作是use的时候，线程T才能对变量V执行load操作。线程T对变量V的use动作可以认为是和线程T对变量V的load、read动作相关联，必须连续一起出现（这条规则要求在工作内存中，每次使用V前都必须先从主内存刷新最新的值，用于保证能看见其他线程对变量V所做的修改后的值）

（2）只有当线程T对变量V执行的**前**一个动作是**assign**时，线程T才能**后**对变量V执行**store**动作；并且，只有当线程T对变量V执行的后一个动作是store时，线程T才能对变量V执行assign动作。线程T对变量V的assign动作可以认为是和线程T对变量V的store、write动作相关联，必须连续一起出现（这条规则要求在工作内存中，每次修改V后都必须同步回主内存中，用于保证其他线程可以看到自己对变量V所做的修改）。

（3）假定动作A是线程T对变量V实施的use或assign动作，假定动作F是和动作A相关联的load或store动作，假定动作P是和动作F相应的对变量V的read或write动作；类似的，假定动作B是线程T对变量W实施的use或assign动作，假定动作G是和动作B相关联的load或store动作，假定动作Q是和动作G相应的对变量W的read或write动作。如果A先于B，那么P先于Q（这条规则要求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与程序的顺序相同）

**（4）对于long和double型变量的特殊规则**

JMM规定了8个原子操作，但对于64位数据类型（long和double），在模型中特别定义了一条相对宽松的规则：

允许虚拟机将没有被volatile修饰的64位数据读写操作划分为两次32位的操作来进行，即允许虚拟机实现选择可以不保证64位数据类型的load、store、read、write这4个操作的原子性。这就是所谓long和double的非原子协定。

如果有多个线程共享一个未被声明为volatile的long或double类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读取到一个既非原值，也不是其他线程修改值的代表了“半个变量”的数值。但是目前**主流商用虚拟机**都选择把**64位数据的读写操作作为原子操作来对待**，所以程序员在编写代码时一般**不需要特别将用到的long和double变量专门声明为volatile**。

**（5）原子性、可见性与有序性**

JMM是围绕着在并发过程中如何处理原子性、可见性和有序性这3个特征来建立的。

原子性（Atomicity）：原子性变量操作包括read、load、assign、use、store和write，所以基本数据类型的访问读写是具备原子性的（除long和double非原子性协定外）。JMM还提供了lock和unlock来应对一个更大范围的原子性保证，虽然这两个操作没有直接对用户开放，但是JVM提供了更高层次的字节码指令monitorenter和monitorexit来隐式使用这两个操作，这两个字节码指令反映到Java代码中就是同步块——**synchronized**关键字，因此在synchronized块之间的操作也具备原子性。

可见性（Visibility）：可见性是指一个线程修改了共享变量的值，其他线程能够立即得知这个修改。JMM是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变量值这种依赖主内存作为传递媒介的方式来实现可见性的。除**volatile**外，Java还有两个关键字能实现可见性，即**synchronized**和**final**。synchronized同步块的可见性是由“对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store、write操作）”这条规则获得的。final关键字的可见性是指：被final修饰的字段在构造器一旦初始化完成，并且构造器没有把“this”的引用传递出去（注：发生this引用逃逸是一件很危险的事情，因为其他线程有可能通过这个引用访问到了“初始化了一半”的对象），那在其他线程中就能看见final字段的值。

**有序性（Ordering）**：Java程序中天然的有序性可以总结为：“如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的”。前半句指“线程内部表现为串行的语义”，后半句指“指令重排序”现象和“工作内存与主内存同步延迟”现象。Java语言提供了**volatile**和**synchronized**两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一个时刻只允许一条线程对其进行lock操作”这条规则获得的，这条规则决定了持有同一个锁的两个同步块只能串行地进入。

因此，synchronized关键字都满足这三个特性，所以大部分的并发控制都可以用synchronized来完成。但是synchronized的万能也间接造就了它被程序员滥用的局面，带来巨大的性能影响。

**（6）先行发生原则**

Java语言中有一个“先行发生”原则，它是判断数据是否存在竞争、线程是否安全的主要依据。依靠这个规则，可以通过定义几条规则来解决并发环境下两个操作之间是否可能存在冲突的所有问题。

先行发生是指JMM中定义的两项操作之前的偏序关系，如果说操作A先行发生于操作B，也就是说发生操作B之前，操作A产生的影响能被操作B观察到。“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。

JMM下有一些“天然的”先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作的关系不在以下关系列表中，或者无法从下列关系中推导出来，它们就没有顺序性保障，虚拟机可以对它们随意地进行重排序。

（1）程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生与书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。

**（2）管程锁定规则**：一个unlock操作先行发生于后面对**同一个锁**的lock操作。这里必须强调的是同一个锁，而“后面”指时间上的先后顺序（Synchronized同步块）。

（3）volatile变量规则：对一个被volatile关键字修饰的变量的写操作先行发生于后面对这个变量的读操作。

（4）线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。

（5）线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，用户可以通过Thread.join()方法来使得当前线程等待被调用线程执行结束、Thread.isAlive()的返回值等手段检测线程已经终止运行。

（6）线程中断规则，对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。

（7）对象终结规则：一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize()方法的开始。

（8）传递性：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论了。

三、Java与线程

（1）线程的实现

Java语言中，每一个已经执行start()且还未结束的java.lang.Thread类的实例就代表了一个线程。Thread类中的所有关键方法都是声明为Native的。

实现线程主要有3种方式：

（1）使用内核线程（KLT）实现

（2）使用用户线程（UT）实现

**（3）使用用户线程加轻量级进程（LWP）混合实现**

（2）Java线程调度

一般线程的主要调度方式有两种，分别是 协同式线程调度 和 **抢占式线程调度**。

协同式线程调度：线程的执行时间由线程本身来控制，线程把自己的工作执行完了之后，要主动通知系统切换到另外一个线程上。这种方式虽然实现简单，但是会导致线程执行时间不可控制，有可能导致某个线程长时间阻塞不让入CPU执行时间而导致系统崩溃。

**抢占式线程调度**：每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定（在Java中，Thread.yield()可以让出执行时间，但是线程本身是没有办法获取执行时间的）。在这种实现线程调度的方式下，线程执行时间是系统可控的，不存在一个线程导致整个进程阻塞的问题，所以**Java所采用的线程调度方式就是抢占式调度**。

虽然Java线程调度是由系统自动完成的，但是我们可以通过设置线程优先级来实现对某些执行紧迫性任务的线程多分配执行时间，另外的一些线程少分配一些执行时间。Java语言一共定义了**10个级别的线程优先级**（Thread.MIN\_PRIORITY至Thread.MAX\_PRIORITY），当两个线程同时处于Ready状态时，优先级越高的线程越容易被系统选择执行。但需要了解的是，虽然Java线程是通过映射到系统的原生线程上来实现的，但是线程调度最终还是取决于操作系统。

（3）状态转换

**Java语言中定义了5种线程状态**，在任意一个时间点，一个线程有且只有其中的一种状态，这5种状态分别如下：

**新建（New）**：创建后尚未启动（Thread.start）的线程处于这种状态；

**运行（Runable）**：Runable状态包括了操作系统线程状态中的Running（运行中）和Ready（就绪），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（obj.wait()，释放CPU执行权的同时也释放同步锁），它们要等待被其他线程显式地唤醒（obj.notify()）。以下方法会让线程陷入无限期的等待状态：

（1）没有设置Timeout参数的Object.wait()方法

**等待**

（2）没有设置Timeout参数的Thread.join()方法

（3）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：

（1）Thread.sleep()方法 —— 只释放CPU执行权，不释放同步锁

（2）设置了Timeout参数的Object.wait()方法

（3）设置了Timeout参数的Thread.join()方法

（4）LockSupport.parkNanos()方法

（5）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（等待状态的前提是拥有锁）。

**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时会相互转换

run()结束

wait()

synchronized()

sleep()

Timed Waiting

Timed Waiting

notify()/notifyAll()

Waiting

Terminated

start()

Runnable

Blocked

New