并发基础

CPU多级缓存

CPU设置缓存意义：1）**时间局部性**：如果某个数据被访问，那么在不久的将来也可能被访问。2）**空间局部性**：如果某个数据被访问，那个与它相邻的数据很快也可能访问。

CPU多级缓存之缓存一致性（MESI）

在一个有多核的系统中，每一个核都有自己的缓存来共享主存总线，目的是为了解决**CPU运算速度与内存读写速度不匹配**的矛盾

缓存一致性（MESI）是为了保证**CPU多级缓存的共享一致性**。MESI定义了四种状态，也就是CPU对四种操作产生不一致的状态。缓存控制器监听到本地操作和远程操作的时候，需要对cache做出修改，从而保证数据在cache之间的一致性。

M: Modified 修改，指的是**该缓存行只被缓存在该CPU的缓存中，并且是被修改过**的，因此他与主存中的数据是不一致的，该缓存行中的数据需要在未来的某个时间点（允许其他CPU读取主存相应中的内容之前）写回主存，然后状态变成E（独享）。

E：Exclusive **独享缓存行只被缓存在该CPU的缓存中，是未被修改过的**，与主存的数据是一致的，可以在任何时刻当有其他CPU读取该内存时，变成S（共享）状态，当CPU修改该缓存行的内容时，变成M（被修改）的状态。

S：Share 共享，意味着该**缓存行可能会被多个CPU进行缓存，并且该缓存中的数据与主存数据是一致**的，当有一个CPU修改该缓存行时，其他CPU是可以被作废的，变成I(无效的)。

I：Invalid 无效的，代表这个缓存是无效的，可能是有其他CPU修改了该缓存行。

|  |  |
| --- | --- |
| Local read | 读本地缓存中的数据 |
| Local write | 将数据写到本地的缓存中 |
| Remote read | 将内存的数据读取过来 |
| Remote write | 将数据写回到主存中 |

1. 一个缓存除了在I状态之外都可以满足CPU的读请求。
2. 一个写请求只有在M或E状态下才能执行。如果处在S状态，必须先将该缓存中缓存行变成I状态。通常通过广播方式执行。不允许多个CPU修改同一个缓存行，即使修改该缓存行不同的数据也是不允许的。
3. 一个处于M状态的缓存行，必须监听所有试图读缓存行的操作，这种操作必须在缓存将缓存行写回到主存，并将状态变回S状态之前被延迟执行。
4. 一个处于S状态的缓存行，也必须监听其他缓存使改缓存行无效或者独享该缓存行的请求并将缓存行变成无效。
5. 一个处于E状态的缓存行，要监听其他缓存读缓存中该缓存行的操作，一旦有该缓存行的操作，它就会变成S状态。
6. 对于M和E两种状态而言，数据总是精确的，和缓存行的真正状态是一致的，S状态可能是非一致的，如果一个缓存将处于S状态的缓存行作废，
7. 一个缓存可能已经独享了该缓存行，但是该缓存却不会将缓存行升为E状态，因为其他缓存不会广播它们作废掉该缓存行的通知。由于缓存并没有保存该缓存行的copy数量，因此也没办法确定自己是否独享了该缓存行。
8. E更像一个投机性的优化，因为一个CPU想修改一个S状态的缓存行，总线事务需要将所有该缓存行copy的值变成I状态，但修改E状态的缓存，却不需要总线事务。

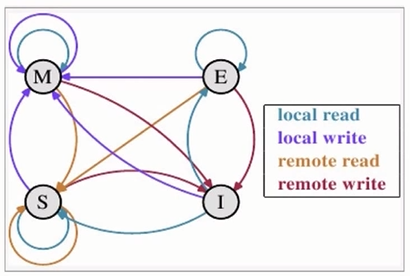


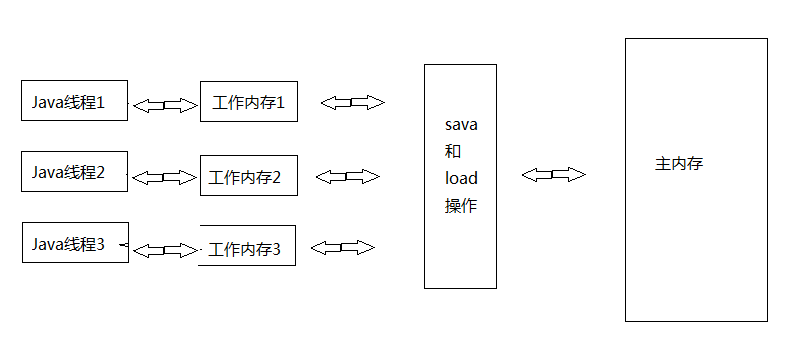
图 1 CPU缓存状态转换表

JAVA内存模型

主内存与工作内存

　　Java内存模型的主要目标是定义**程序中各个变量的访问规则**，即在JVM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。此处的变量与Java编程里面的变量有所不同步，它包含了**实例字段、静态字段和构成数组对象的元素，但不包含局部变量和方法参数**，因为后者是线程私有的，不会共享，当然不存在数据竞争问题（如果局部变量是一个reference引用类型，它引用的对象在Java堆中可被各个线程共享，但是reference引用本身在Java栈的局部变量表中，是线程私有的）。为了获得较高的执行效能，Java内存模型并没有限制执行引起使用处理器的特定寄存器或者缓存来和主内存进行交互，也没有限制即时编译器进行调整代码执行顺序这类优化措施。

JMM规定了**所有的变量都存储在主内存（Main Memory）中。每个线程还有自己的工作内存（Working Memory）,线程的工作内存中保存了该线程使用到的变量的主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）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（volatile变量仍然有工作内存的拷贝**，但是由于它特殊的操作顺序性规定，所以看起来如同直接在主内存中读写访问一般）。不同的线程之间也无法直接访问对方工作内存中的变量，线程之间值的传递都需要通过主内存来完成。



线程1和线程2要想进行数据的交换一般要经历下面的步骤：

　　1.线程1把工作内存1中的更新过的共享变量刷新到主内存中去。

　　2.线程2到主内存中去读取线程1刷新过的共享变量，然后copy一份到工作内存2中去。

内存间相互操作

关于主内存与工作内存之间的具体交互协议，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到主内存之间的实现细节，Java内存模型定义了以下八种操作来完成：

1. lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
2. 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。
3. read（读取）：作用于主内存变量，把一个变量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
4. load（载入）：作用于工作内存的变量，它把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。
5. use（使用）：作用于工作内存的变量，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时将会执行这个操作。
6. assign（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值给工作内存的变量，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。
7. store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
8. write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行read和load操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。Java内存模型只要求上述操作**必须按顺序执行，而没有保证必须是连续执行**。也就是read和load之间，store和write之间是可以插入其他指令的，如对主内存中的变量a、b进行访问时，可能的顺序是read a，read b，load b， load a。Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

* 不允许read和load、store和write操作之一单独出现
* 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。
* 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。
* 一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量
* 一个变量在同一个时刻只允许一条线程对其执行lock操作，但lock操作可以被同一个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
* 如果对一个变量执行lock操作，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。
* 如果一个变量实现没有被lock操作锁定，则不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。
* 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和write操作）。

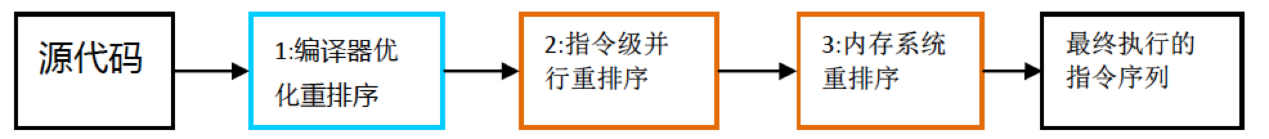
### 重排序

在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

1.编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。

2.指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3.内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。



为了保证内存的可见性，Java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。Java内存模型把内存屏障分为LoadLoad、LoadStore、StoreLoad和StoreStore四种：

|  |  |  |
| --- | --- | --- |
| 屏蔽类型 | 指令示例 | 说明 |
| LoadLoad Barriers | Load1;LoadLoad;Load2 | 确保Load1数据的装载，之前于Load2及所有后续装载指令的装载 |
| StoreStore Barriers | Store1;StoreStore;Store2 | 确保Store1数据对其他处理器可见（刷新到内存），之前于Store2及所有后续存储指令的存储。 |
| LoadStrore Barriers | Load1;LoadStore;Store2 | 确保Load1数据装载，之前于Store2及所有后续的存储指令刷新到内存 |
| StoreLoad Barrirers | Store1;StoreLoad;Load2 | 确保Store1数据对其他处理器可见（刷新到内存），之前于Load2及所有后续装载指令的装载，StroeLoad Barrirers会使该屏蔽之前的所有内存访问指令（存储和装载指令）完成之后，才执行该屏障之后的内存访问指令 |

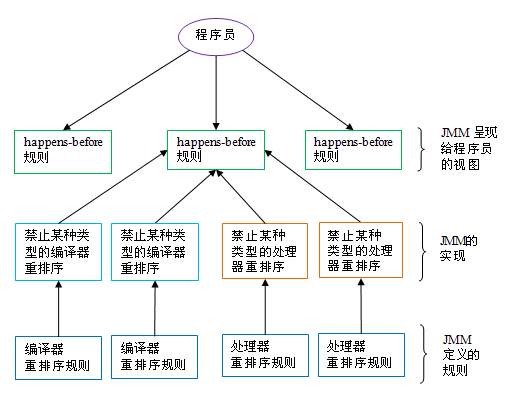
### happens-before

在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系。这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

与程序员密切相关的happens-before规则如下：

* 程序顺序规则：一个线程中的每个操作，happens- before 于该线程中的任意后续操作。
* 监视器锁规则：对一个监视器锁的解锁，happens- before 于随后对这个监视器锁的加锁。
* volatile变量规则：对一个volatile域的写，happens- before 于任意后续对这个volatile域的读。
* 传递性：如果A happens- before B，且B happens- before C，那么A happens- before C。

注意，两个操作之间具有happens-before关系，并不意味着前一个操作必须要在后一个操作之前执行！happens-before仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前



### Java内存区

根据 JVM 规范，JVM 内存共分为**虚拟机栈、堆、方法区、程序计数器、本地方法栈**五个部分，这些区域都有各自的用途、创建时间、销毁时间。

1. PC寄存器/程序计数器：严格来说是一个数据结构，用于保存当前正在执行的程序的内存地址，为了线程切换后能恢复到正确的执行位置，**每个线程都需要有一个独立的程序计数器，各个线程之间计数器互不影响，独立存储**，我们称这类内存区域为“线程私有”的内存,这在某种程度上有点类似于“ThreadLocal”，是线程安全的。
2. Java栈 Java Stack：Java栈总是与线程关联在一起的，每当**创建一个线程，JVM就会为该线程创建对应的Java栈，在这个Java栈中又会包含多个栈帧(Stack Frame)，这些栈帧是与每个方法关联起来的**，每运行一个方法就创建一个栈帧，每个栈帧会含有一些局部变量、操作栈和方法返回值等信息。每当一个方法执行完成时，该栈帧就会弹出栈帧的元素作为这个方法的返回值，并且清除这个栈帧，Java栈的栈顶的栈帧就是当前正在执行的活动栈，也就是当前正在执行的方法，PC寄存器也会指向该地址。只有这个活动的栈帧的本地变量可以被操作栈使用，当在这个栈帧中调用另外一个方法时，与之对应的一个新的栈帧被创建，这个新创建的栈帧被放到Java栈的栈顶，变为当前的活动栈。同样现在只有这个栈的本地变量才能被使用，当这个栈帧中所有指令都完成时，这个栈帧被移除Java栈，刚才的那个栈帧变为活动栈帧，前面栈帧的返回值变为这个栈帧的操作栈的一个操作数。由于Java栈是与线程对应起来的，Java栈数据不是线程共有的，所以不需要关心其数据一致性，也不会存在同步锁的问题。在Java虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。
3. 堆 Heap: 堆是JVM所管理的内存中国最大的一块，是被所有Java线程锁共享的，不是线程安全的，在JVM启动时创建。**堆是存储Java对象的地方**，这一点Java虚拟机规范中描述是：所有的对象实例以及数组都要在堆上分配。Java堆是GC管理的主要区域，从内存回收的角度来看，由于现在GC基本都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代；新生代再细致一点有Eden空间、From Survivor空间、To Survivor空间等
4. 方法区Method Area: 方法区存放了要加载的类的信息（名称、修饰符等）、类中的静态常量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当在程序中通过Class对象的getName.isInterface等方法来获取信息时，这些数据都来源于方法区。方法区是被Java线程锁共享的，不像Java堆中其他部分一样会频繁被GC回收，它存储的信息相对比较稳定，在一定条件下会被GC，当方法区要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。方法区也是堆中的一部分，就是我们通常所说的Java堆中的永久区 Permanet Generation，大小可以通过参数来设置,可以通过-XX:PermSize指定初始值，-XX:MaxPermSize指定最大值。
5. 常量池Constant Pool: 常量池本身是方法区中的一个数据结构。常量池中存储了如字符串、final变量值、类名和方法名常量。常量池在编译期间就被确定，并保存在已编译的.class文件中。一般分为两类：字面量和应用量。字面量就是字符串、final变量等。类名和方法名属于引用量。引用量最常见的是在调用方法的时候，根据方法名找到方法的引用，并以此定为到函数体进行函数代码的执行。引用量包含：类和接口的权限定名、字段的名称和描述符，方法的名称和描述符。

本地方法栈Native Method Stack: 本地方法栈和Java栈所发挥的作用非常相似，区别不过是Java栈为JVM执行Java方法服务，而本地方法栈为JVM执行Native方法服务。本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。



图 2 java内存堆和栈

存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的**私有拷贝**。

### 线程安全