浅谈JMM

## 数据同步

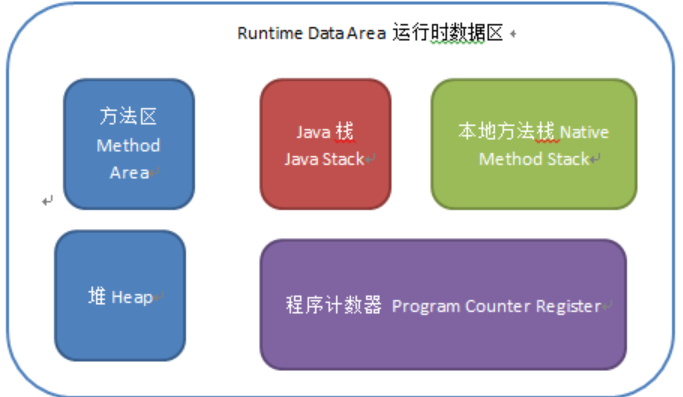
在并发编程中，多个线程之间采取什么机制进行通信（信息交换），什么机制进行数据的同步？

在Java语言中，采用的是共享内存模型来实现多线程之间的信息交换和数据同步的。

线程之间通过共享程序公共的状态，**通过读-写内存中公共状态的方式来进行隐式的通信**。**同步指的是程序在控制多个线程之间执行程序的相对顺序的机制，在共享内存模型中，同步是显式的，**程序员必须显式指定某个方法/代码块需要在多线程之间互斥执行。

在说Java内存模型之前，我们先说一下Java的内存结构，也就是运行时的数据区域：

Java虚拟机在执行Java程序的过程中，会把它管理的内存划分为几个不同的数据区域，这些区域都有各自的用途、创建时间、销毁时间。



## 内存区域

Java运行时数据区分为下面几个内存区域：

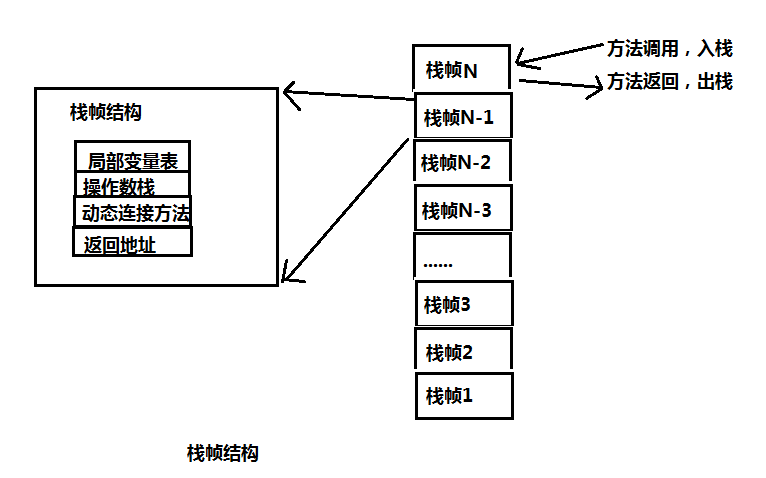
### PC寄存器/程序计数器：

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**，由于Java是支持多线程执行的，所以程序执行的轨迹不可能一直都是线性执行。当有多个线程交叉执行时，被中断的线程的程序当前执行到哪条内存地址必然要保存下来，以便用于被中断的线程恢复执行时再按照被中断时的指令地址继续执行下去。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**。

### 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**。在Hot Spot虚拟机中，可以**使用-Xss参数来设置栈的大小**。栈的大小直接决定了函数调用的可达深度。

### 堆 Heap:

堆是JVM所管理的内存中最大的一块，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。

### 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**。

### 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。

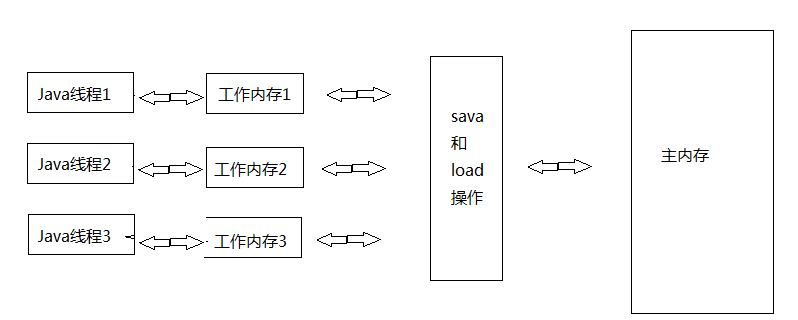
### 本地方法栈Native Method Stack:

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。

## 主内存和工作内存：

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(类似于数据库)。**



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。

### 内存间交互操作

内存间交互主要指工作内存（本地内存）与主内存之间的交互，即一个变量如何从主内存拷贝到工作内存，如何从工作内存刷新到主内存的一些实现细节。

Java 内存模型**对主内存与工作内存之间的具体交互协议定义了八种操作**，具体如下：

lock（锁定）：作用于主内存变量，把一个变量标识为一条线程独占状态。

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）：作用于工作内存变量，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。

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值的字节码指令时执行此操作。

assign（赋值）：作用于工作内存变量，把一个从执行引擎接收的值赋值给工作内存的变量，每当虚拟机遇到一个需要给变量进行赋值的字节码指令时执行此操作。

store（存储）：作用于工作内存变量，把工作内存中一个变量的值传递到主内存中，以便后续 write 操作。

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

对于这8个操作，有如下的一个原则：  
1、不允许read和load，store和write操作单独出现。  
2、不允许一个线程丢弃它最近的assign操作，即变量在工作内存中的更新需要同步到主内存中。  
3、不允许线程无原因地（没有发生过任何assign操作）把数据同步到主内存。  
4、一个新的变量只能在主内存中产生，不能在工作内存中直接使用未被初始化的变量。  
5、一个变量在同一时刻只能被一个线程lock，并且lock和unlock需要成对出现。  
6、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执行load或者assgin操作。  
7、对一个变量执行unclock之前，必须把此变量同步到主内存中。

## 特性

 Java内存模型是围绕着并发编程中原子性、可见性、有序性这三个特征来建立的，那我们依次看一下这三个特征：

### 原子性（Atomicity）

一个操作不能被打断，要么全部执行完毕，要么不执行。在这点上**有点类似于事务操作，要么全部执行成功，要么回退到执行该操作之前的状态**。

基本类型数据的访问大都是原子操作，long 和double类型的变量是64位，但是在32位JVM中，32位的JVM会将64位数据的读写操作分为2次32位的读写操作来进行，这就导致了**long、double类型的变量在32位虚拟机中是非原子操作**，数据有可能会被破坏，也就意味着多个线程在并发访问的时候是线程非安全的。

32位的JVM中，要想保证对long、double类型数据的操作的原子性，可以对访问该数据的方法进行同步，就像下面的：参考Atomicity.java

### 可见性

**一个线程对共享变量做了修改之后，其他的线程立即能够看到（感知到）该变量这种修改（变化）**。

Java内存模型是通过将在工作内存中的变量修改后的值同步到主内存，在读取变量前从主内存刷新最新值到工作内存中，这种依赖主内存的方式来实现可见性的。

无论是普通变量还是volatile变量都是如此，区别在于：volatile的特殊规则保证了volatile变量值修改后的新值**立刻**同步到主内存，每次使用volatile变量前立即从主内存中刷新，因此volatile保证了多线程之间的操作变量的可见性，而普通变量则不能保证这一点(效果相当于回调钩子函数)。

**除了volatile关键字能实现可见性之外，还有synchronized,Lock，final也是可以的**。

使用synchronized关键字，在同步方法/同步块开始时（Monitor Enter）,使用共享变量时会从主内存中刷新变量值到工作内存中（即从主内存中读取最新值到线程私有的工作内存中），在同步方法/同步块结束时(Monitor Exit),会将工作内存中的变量值同步到主内存中去（即将线程私有的工作内存中的值写入到主内存进行同步）。

使用Lock接口的最常用的实现ReentrantLock(重入锁)来实现可见性：当我们在方法的开始位置执行lock.lock()方法，这和synchronized开始位置（Monitor Enter）有相同的语义，即使用共享变量时会从主内存中刷新变量值到工作内存中（即从主内存中读取最新值到线程私有的工作内存中），在方法的最后finally块里执行lock.unlock()方法，和synchronized结束位置（Monitor Exit）有相同的语义,即会将工作内存中的变量值同步到主内存中去（即将线程私有的工作内存中的值写入到主内存进行同步）。

final关键字的可见性是指：被final修饰的变量，在构造函数数一旦初始化完成，并且在构造函数中并没有把“this”的引用传递出去（**“this”引用逃逸是很危险的，其他的线程很可能通过该引用访问到只“初始化一半”的对象）**，那么其他线程就可以看到final变量的值。

### 有序性

对于一个线程的代码而言，我们总是以为代码的执行是从前往后的，依次执行的。这么说不能说完全不对，在单线程程序里，确实会这样执行；但是在多线程并发时，程序的执行就有可能出现乱序。用一句话可以总结为：在本线程内观察，操作都是有序的；如果在一个线程中观察另外一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行语义（WithIn Thread As-if-Serial Semantics）”,后半句是指“指令重排”现象和“工作内存和主内存同步延迟”现象。

Java提供了两个关键字volatile和synchronized来保证多线程之间操作的有序性,volatile关键字本身通过加入**内存屏障来禁止指令的重排序**，而synchronized关键字通过一个变量在同一时间只允许有一个线程对其进行加锁的规则来实现，

在单线程程序中，不会发生“指令重排”和“工作内存和主内存同步延迟”现象，只在多线程程序中出现。

### happens-before原则：

**Java内存模型中定义的两项操作之间的次序关系，如果说操作A先行发生于操作B，操作A产生的影响能被操作B观察到，“影响”包含了修改了内存中共享变量的值、发送了消息、调用了方法等。**

下面是Java内存模型下一些”天然的“happens-before关系，这些happens-before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。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

程序次序规则(Pragram Order Rule)：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于书写在后面的操作。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、循环结构。

管程锁定规则(Monitor Lock Rule)：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“是指时间上的先后顺序。

volatile变量规则(Volatile Variable Rule)：对一个volatile变量的写操作先行发生于后面对这个变量的读取操作，这里的”后面“同样指时间上的先后顺序。

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

线程终于规则(Thread Termination Rule)：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过Thread.join()方法结束，Thread.isAlive()的返回值等作段检测到线程已经终止执行。

线程中断规则(Thread Interruption Rule)：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()方法检测是否有中断发生。

对象终结规则(Finalizer Rule)：一个对象初始化完成(构造方法执行完成)先行发生于它的finalize()方法的开始。

传递性(Transitivity)：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。

时间上的先后顺序与happens-before原则之间基本没有什么关系，所以衡量并发安全问题一切必须以happens-before 原则为准。