计算机内存体系与Java 内存模型

导语：

并发处理的广泛应用是使得阿姆达尔定律代替摩尔定律成为计算机发展源动力的根本原因，也是人类压榨计算机运算能力的最强有力武器。

*阿姆达尔（Amdahl）定律****通过系统中并行化与串行化的比重****来描述多个处理器系统能获得的运算加速能力。摩尔定律则用于****描述处理器晶体管数量与运行效率之间的发展关系****。这两个定律的更替代表了近年来硬件发展****从追求处理器频率到追求多核心并行处理****的发展过程。*

# 计算机内存相关硬件介绍与缓存一致性：

让计算机并发执行若干个任务与更充分的利用计算机处理器的效能之间的因果关系，非常的复杂，其中一个重要的复杂性来源是**绝大多数的运算任务都不可能只靠处理器计算就能完成的**，处理器至少要与内存交互，如：读取运算数据，存储运算结果等。这个IO操作是很难消除的（无法仅靠寄存器来完成所有的运算任务）。由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不**加入一层读写速度尽可能接近处理器运算速度的高速缓存来作为内存与处理器之间的缓冲**：将运算需要使用到的数据复制到缓冲中，让运算能快速进行，当运算结束后再从缓存同步回主内存之中，这样处理器就无需等待缓慢的内存读写了。

这里真是又不得不搬出大学时枯燥无聊的计算机系统结构了，真心后悔那会听得云里雾里打瞌睡啊~~

觉得有必要先解释一下处理器，寄存器，高速缓存，内存之间的关系：

处理器大家都非常了解，就是负责做逻辑运算的，也就是你写的程序代码最终都会变成一条条指令或计算公式，这些玩意呢到处理器里面就变成了二进制的各种组合，处理器计算后会得到一个结果。

寄存器是离处理器最近的一块存储介质，可以说位于内存模型的顶端，它的速度非常之快，快到可以和处理器相媲美，处理器从里面拿数据，运算完之后又把数据存回去。寄存器是处理器里面的一部分，处理器可能有多个寄存器，比如数据计数器，指令指针寄存器等等。

高速缓存是一个比内存速度快很多接近处理器速度的存储区域，目的是把处理器要用到的一堆数据从主内存中复制进来供处理器享用，处理器享用完了之后又把结果同步回主内存，这样处理器只做自己的事，而高速缓存就成了传话筒。高速缓存有分为一级缓存，二级缓存和三级缓存，离处理器最近的是一级缓存，依次往后排，他们的容量大小也是由一往三 一级比一级大，所以你买电脑看CPU的时候可以关注下这些缓存，数值越大自然性能越好。

主内存是我们通常讲的内存，比如现在的电脑动不动8G，16G啊等等，这些说的就是这个主内存，它比磁盘的读写速度快很多，但是又跟高速缓存没法比，因此，程序启动的时候，程序相关的数据会加载到主内存，然后处理器处理某块逻辑的时候，比较占空间的东西会丢到主内存，比如Java里面的对象，就是存放在堆上面的，而Java虚拟机里面的堆就是放在主内存的。

那么问题来了，学挖掘机到底哪家强？（为啥处理器和内存之间要整的这么繁琐？直接主内存就用高速缓存代替不是简单粗暴高效吗？）

答案很简单：土豪请随意！

高速缓存的造价是非常高昂的，而内存成本要低很多，但是成本低带来的弊病也很明显，速度慢！

其实饶了这么多，说白了就是费尽心思让你花着白菜价享受这法拉利的速度体验。

用一个比较好懂的例子帮助大家记忆他们之间的关系：

处理器好比皇帝，要处理每天的奏折，那么寄存器呢，就好比皇帝身边的太监，负责给皇帝递奏折，收奏折。皇帝身边可能好几个太监，比如专门递紧急事务奏折的太监甲，递一般性奏折的太监乙，递上供美女奏折的太监丙等，寄存器也分很多种，每种都是为了配合CPU进行高速计算和储存的。那么高速缓存呢，就好比朝中大臣，会收集各种重要信息，奏上去，最后这些奏折是通过太监递给皇上的。主内存呢，你可以理解成各地地方官，大臣收集他们的诉求，并反馈给他们。

好了，啰嗦完了，我们现在回到开头讲的，基于高速缓存的存储交互可以很好的解决了处理器与内存之间的矛盾，但是也为计算机系统带来更高的复杂度，因为它引入了一个新的问题：缓存一致性。在多处理器系统中，每个处理器都拥有自己的高速缓存，而它们又共享同一个主内存，当多个处理器的运算任务都设计同一块主内存区域时，将可能导致各自的缓存数据不一致问题。这也是我们为什么要做同步的原因。

# Java内存模型

了解了计算机的一个大致内存模型也很容易了解Java的内存模型。

**Java内存模型规定了所有的变量都存储在主内存中**（Java中的主内存和刚才上面讲的计算机的主内存是一样的，只不过Java是占用分配给虚拟机的那一部分），**每个线程还有自己的工作内存**（可与前面讲的高速缓存类比），**线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝**（如果对象很大怎么办呢，比如10个MB？虚拟机肯定不会那么傻把对象直接拷贝进去啦，会拷贝对象的地址，以及需要用到的字段的值，哪怕值也很大，比如大的字符串，也会有相应的机制保证不会全拷贝，不然不是直接就爆掉了么）。

**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。**本人之前有写过一篇博客《volatile的各种特性》，那么volatile可以保证任何情况下变量的可见性，是不是就是直接读主内存呢，事实上，volatile变量依然有工作内存的拷贝，但是它的操作顺序比较特殊，会每次都从主内存重新加载，所以你会看到每次volatile读取到的都是最新的值。

**不同的线程也无法访问其他线程的工作内存中的变量，线程间变量值的传递需要通过主内存来完成**。

**这里说的Java的主内存，工作内存与Java堆，栈，方法区等并不是同一个层次的划分，这两者基本是没有关系的，如果两者一定要勉强对应起来，那从变量，主内存，工作内存的定义上看，主内存只要对应于Java堆中的对象实例数据部分，工作内存对应于虚拟机栈中的部分区域**。从更低层次上来说，主内存就是直接对应于物理硬件的内存，而为了获取更好的运行速度，**虚拟机可能会让工作内存优先存储于寄存器和高速缓存区中**，因为程序运行时主要访问读写的是工作内存。

# Java主内存与线程工作内存间8种交互操作

Java**一个变量如何从主内存拷贝到工作内存呢，又如何从工作内存同步回主内存**呢？Java内存模型定义了8中操作来完成，虚拟机保证下面的每一种操作都是原子的。

**lock（锁定）**：作用于主内存变量，它把一个变量标识为一条线程独占的状态。

**unlock（解锁）**：作用于主内存变量，它把一个处于锁定状态的变量释放出来，释放后的变量才能被其他线程锁定。

**read（读取）**：作用于主内存变量，它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，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要使用变量的值的字节码指令时，将会执行这个操作。

**assign（赋值）**：作用于工作内存变量，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。

**store（存储）**：作用于工作内存变量，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。

**write（写入）**：作用于主内存变量，它把store操作中从工作内存得到的变量的值放入主内存的变量中。

如果要把一个变量从主内存复制到工作内存，那就要顺序的执行read和load操作，如果要把工作内存的变量同步会主内存，那就要顺序的执行store和write操作。注意，Java内存模型只是要求上述两个操作必须按顺序执行，而没有保证是连续执行。也就是说，read与load之间，store与write之间是可以插入其他指令的。此外，Java内存模型还规定了在执行上述8中操作时必须满足如下规则：

1）**不允许read和load，store和write操作之一单独出现**。即：不允许从主内存读取了，但工作内存不要它，或者从工作内存写出去的，主内存不要。

2）**不允许一个线程丢弃它的最近的assign操作**。即：变量在工作内存改变之后，必须同步回主内存。

3）**不允许一个线程无原因地（没有进行任何assign操作的）把数据从线程的工作内存同步回主内存中**。

4）**一个新的变量必须在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load和assign）的变量**。

5）**一个变量在同一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。

6）**如果对一个变量进行了lock操作，那将会清空工作内存中此变量的值**，因此执行引擎使用这个变量之前，需要重新的进行load和assign操作。

7）**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一个其他线程锁定住的变量。**

8）**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之中**。

# Java多线程：

并发并不一定要依赖多线程，比如“世界上最好的开发语言”PHP就是用多进程实现并发，那么Java线程底层又是如何实现的呢？耐心看到最后哦。

线程是CPU调度的基本单位，主流操作系统都提供了线程实现，Java语言则提供了在不同硬件和操作系统平台下对线程的统一处理（这也是Java敢声称一次编译到处运行的底气之一），Java里面线程对象是Thread类产生的，Thread类的关键方法都是Native的，也就是说这些方法都是针对特定的操作系统去特定实现的。

实现线程主要有三种方式：

**1）使用内核线程实现：**

内核线程就是直接由操作系统支持的线程，这种线程由内核来完成线程的切换，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，每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核叫多线程内核。

然而程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口---轻量级进程。轻量级进程就是我们通常意义上讲的线程，由于每个轻量级进程都有一个内核线程支持，因此只有线支持内核进程，才能支持轻量级进程。这种轻量级进程与内核线程之间是一比一存在的。

轻量级进程可以很好的配合处理器调度，但是也是有一些局限性的，一是由于是基于内核线程实现，因此创建、析构、同步等都需要进行系统调用，而系统调用的代价相对高，需要在用户态和内核态来回切换。二是一个操作系统支持的内核线程是有限的也就造成支持轻量级进程也是有限的。

**2）使用用户线程实现**

可以认为，一个线程只要不是内核线程，那就是用户线程。用户线程的创建，销毁，同步等完全在用户态中完成，不需要内核的帮助，因此操作是非常快且低消耗的，也可以支持更大规模的线程数量，部分高性能数据库就是通过用户线程实现的。这种进程与用户线程之间的关系为1：N的关系成为一对多线程模型。

但是用户线程由于没有内核线程的支持，因此所有的线程操作都需要用户程序自己处理。Java，Ruby等语言都曾经使用过用户线程，最终又都放弃了它，可见，它也并不是好的方案。

**3）使用用户线程加轻量级进程混合实现**

用户线程还是完全建立在用户空间中，因此用户线程的创建，切换，析构等操作依然廉价，并且可以支持大规模的用户线程并发。而操作系统提供支持的轻量级进程则作为用户线程和内核之间的桥梁，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级进程来完成，大大降低了整个进程被完全阻塞的风险。这种混合模式下，用户线程和轻量级进程的数量比是不确定的，即为N:M，这种就是多对多的线程模型。

**4）Java线程的实现**

在JDK1.2以前，是由用户线程实现的，JDK1.2以后，线程模型替换为基于操作系统原生线程模型来实现，因此，在目前的JDK版本中，操作系统支持怎样的线程模型，在很大程度上决定了Java虚拟机的线程是怎样映射的，这点在不同平台上没法达成一致，虚拟机规范也并未要求Java一定要用哪种线程模型来实现。

**对于Sun JDK来说，它的Windows版和Linux版都是使用一对一的线程模型实现的，即：一个Java线程映射到一个轻量级进程之中，因为Windows和Linux提供的线程模型就是一对一的**。

**而在Solaris平台中，由于操作系统支持一对一和多对多的线程模型，因此在Solaris版本的JDK中也对应提供了两个平台专有的虚拟机参数来明确指定虚拟机使用哪种线程模型**。

# Java线程的状态：

Java的线程大致有6种状态：

**NEW：**

新建状态，new了一个线程。

**Runable:**

可执行状态，线程调用了start()或run()方法，或者等待结束，都会进入Runable状态，在操作系统或处理器而言，java的Runable状态可能是他们的就绪态或运行态，因为从就绪态到运行态是处理器决定的，Java无权过问，所以为了可以控制Java能控制的部分，方便理解和管理线程，Java将这种处于就绪队列或正在执行队列的线程统称Runable状态。

**Waiting：**

等待状态，线程调用了wait()方法或者join()方法之后，就会进入等待状态，知道等待被唤醒，才会进入Runable状态。

**Time Waiting：**

有限时间内等待状态，线程调用了wait(time)方法之后，会进入一个有限期的等待态，超过了时间或者在有限期内被唤醒都会进入Runable状态。

**Blocked：**

阻塞状态，线程在争抢某个对象的锁的时候，当一个线程占有了锁，那么其他争抢伙计都会进入阻塞态，知道抢到锁，才会进入Runable状态。

**Terminated：**

终止状态，线程完成run()方法或者调用了stop()方法或者出现异常等被各种原因终止时就会进入终止状态。