Java内存模型英文为**Java Memory Model**，简称为JMM。JMM本身是一个抽象概念，并非真实存在于Java虚拟机中。它的目的仅仅是定义了程序中各种变量（指实例变量、静态字段和构成数组对象的元素，但不包括局部变量和方法参数，因为局部变量和方法参数是线程私有的，不会被线程共享）的访问规范，即关注在虚拟机中把变量存储到内存和从内存中取出变量值这样的底层细节。而在虚拟机中的运算单元就是线程，因此**可以理解为JMM定义的就是线程访问共享变量的方式。** 当然，这么解释JMM可能仍然很抽象和难以理解。关于JMM我们不妨先放一放，先来了解一下计算机的缓存一致性问题，了解缓存一致性问题将更有利于我们认识Java内存模型。

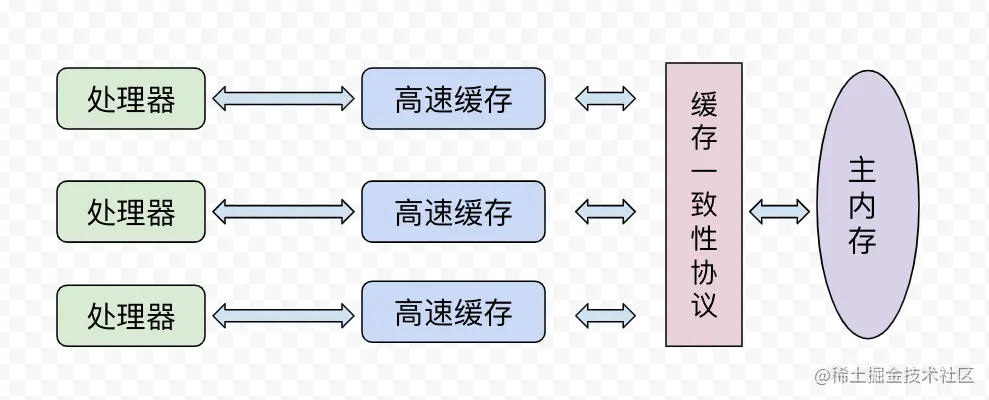
缓存一致性

我们知道，一个简单的计算机可以抽象为CPU、内存以及I/O设备，其中，CPU负责数据的处理与运算，内存则可以理解为存储CPU运算后的数据。CPU在运行时，会首先从内存中取出运算指令，然后解码并确定其类型和操作数，最后执行该指令。在指令执行完毕后，CPU会将计算所得数据写入内存。

然而，在计算机系统中存在一个CPU的运算速度与内存读写速度不匹配的问题，即CPU的运算速度远比内存的读写速度快。由于读写速度缓慢，严重拖累了计算机的运行效率。为了解决这一问题，现代计算机系统在CPU与内存之间加入了一层或多层高速缓存，而高速缓存的读写速度与CPU的运算速度几乎相当。在加入高速缓存后，CPU在执行指令前，需要先将要运算的数据从内存读取（即复制）到高速缓存中，接着CPU对数据进行处理，然后再将运算后的数据写入到高速缓存，最后再从缓存同步回内存中。

可以看到，基于高速缓存的存储交互很好的解决了单CPU与内存读写速度之间的矛盾。但在多CPU的计算机中却引来了新的问题。由于人们对计算机性能的追求，单CPU已经很难维持“摩尔定律”。目前市面上绝大部分都是多CPU的计算机。在多CPU的系统中，每个处理器都有自己的高速缓存，而它们又共享同一个主内存，如下图。当多个处理器的运算任务都涉及到同一块主内存区域时，将可能导致各自缓存数据不一致的问题。例如，处理器1与处理器2都从主内存读取了同一个数据分别存储到自己的高速缓存区域，然后，两个处理器都对这一数据进行了修改。那么再同步回主内存的时候应该以哪条数据为准呢？这一问题就是缓存一致性问题。

为了解决缓存一致性问题，设计者们为CPU制定了一个读写协议，并要求各个CPU在读写缓存时都要遵循这一协议。这类协议有MSI、MESI、MOSI等，被称为缓存一致性协议。只要CPU的读写遵循了缓存一致性协议就能很好的解决缓存一致性问题了。



JMM

JMM规定所有变量都存储在主内存中，每条线程还有自己的工作内存。线程的工作内存中保存了被线程使用的变量的主内存副本，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的数据。不同线程之间也无法直接访问对方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。 也就是说Java线程之间的通信采用的是共享内存。

看到这里是不是觉得似曾相识？没错，这里其实跟上一节缓存一致性中讲到的多CPU共享主内存是类似的。只不过在虚拟机中不是CPU，而是线程。每条线程都有自己的工作空间，而共享变量存储在共享内存中。线程在运行时会首先将共享内存中的数据读取到自己的工作内存，即在线程的工作内存中复制了一个共享变量的副本，然后对其进行计算，计算完成后线程会将自己工作内存中的这个共享变量副本同步回主内存。线程、工作内存、与主内存的关系如下图所示：

