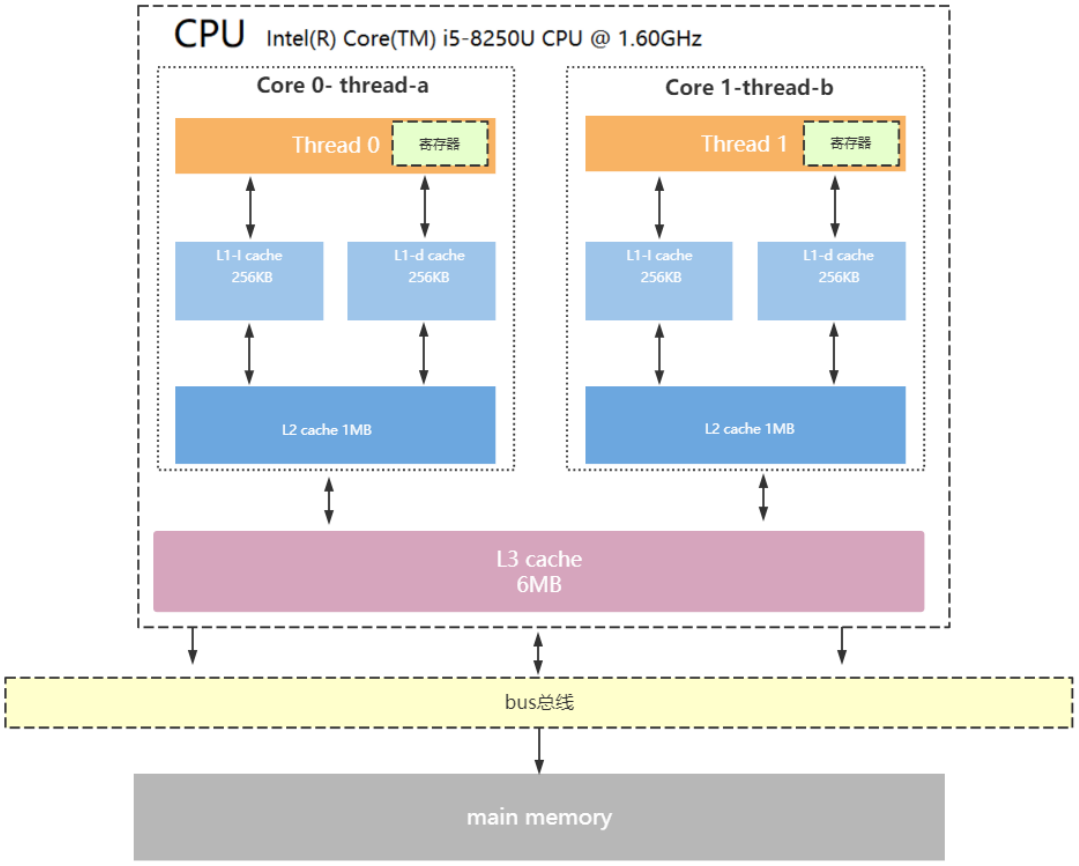
一、JMM内存模型产生的背景?

Java内存模型，又称JMM(Java Memory model)， 是Java 虚拟机规范中所定义的一种内存模型。

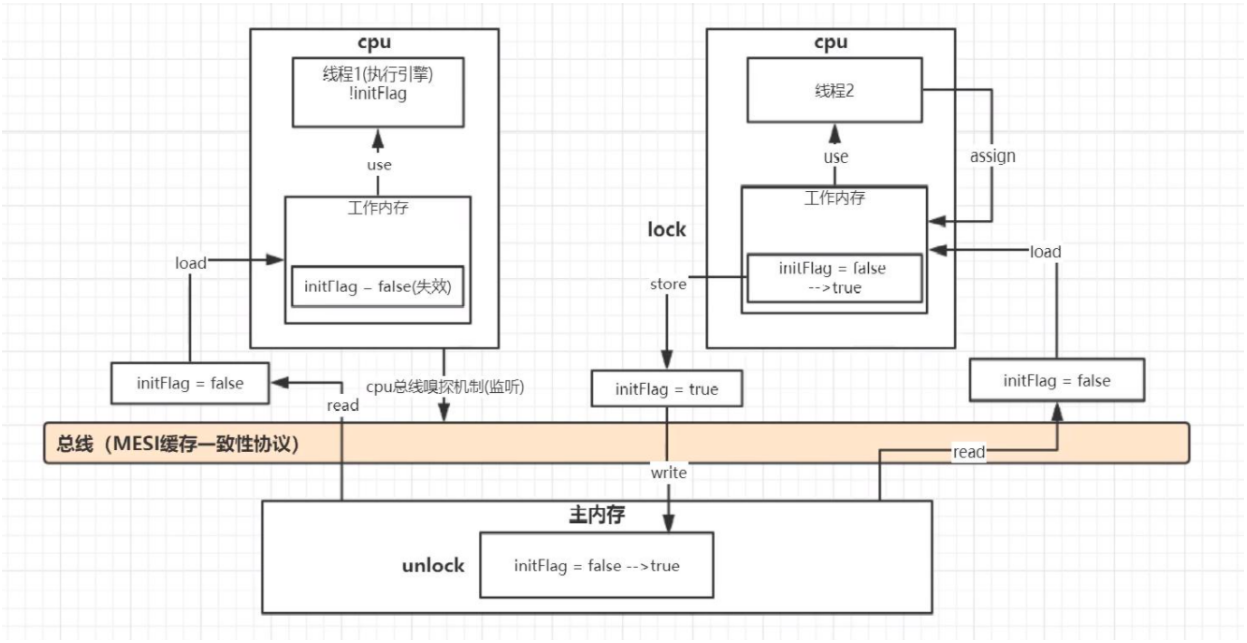
JMM(Java内存模型)源于物理机器CPU架构的内存模型，最初用于解决MP(多处理器架构)系统中的缓存一致性问题，而JVM为了屏蔽各个硬件平台和操作系统对内存访问机制的差异化，提出了JMM的概念。Java内存模型是一种虚拟机规范，JMM规范了Java虚拟机与计算机内存是如何协同工作的：规定了一个线程如何和何时可以看到由其他线程修改过后的共享变量的值，以及在必须时如何同步的访问共享变量。通过这种方式来保证多线程下变量的缓存一致性问题，下图是一个CPU多级缓存图：



二、什么是JMM内存模型?

Java内存模型(Java Memory Model简称JMM)是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内存中的变量副本拷贝，前面说过，工作内存是每个线程的私

有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成。



具体流程解释：

lock（锁定）：将主内存变量加锁，表示为线程独占状态，可以被线程进行read

unlock（解锁）：将主内存的变量解锁，解锁后其他线程可以锁定该变量

read(读取)：线程从主内存读取数据

load(载入)：将上一步线程从主内存中读取的数据，加载到工作内存中

use（使用）：从工作内存中读取数据来进行我们所需要的逻辑计算

assign（复制）：将计算后的数据赋值到工作内存中

store（存储）：将工作内存的数据准备写入主内存

write（写入）：将store过去的变量正式写入主内存

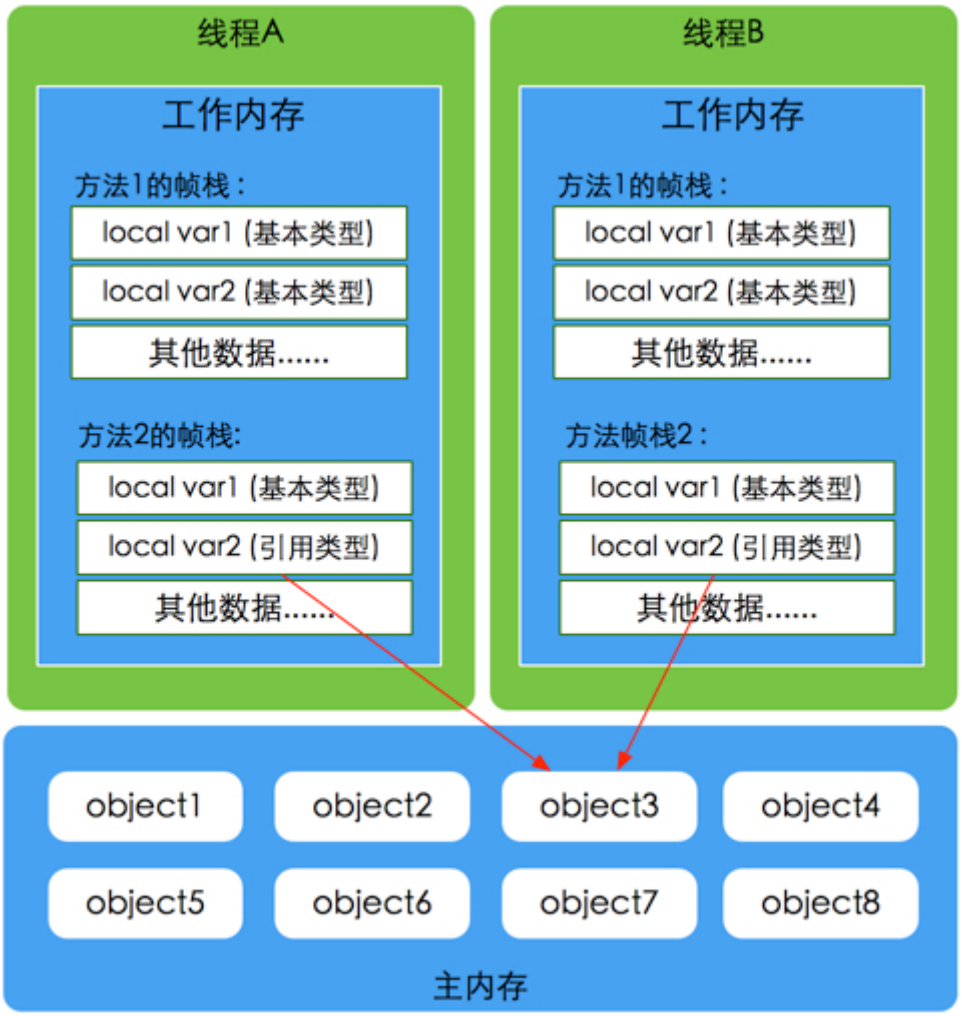
1、主内存

主要存储的是Java实例对象，所有线程创建的实例对象都存放在主内存中，不管该实例对象是成员变量还是方法中的本地变量(也称局部变量)，当然也包括了共享的类信息、常量、静态变量。由于是共享数据区域，多条线程对同一个变量进行访问可能会发生线程安全问题。

2、工作内存

主要存储当前方法的所有本地变量(局部变量)信息(工作内存中存储着主内存中的变量副本拷贝)，每个线程只能访问自己的工作内存，即线程中的本地变量对其它线程是不可见的，就算是两个线程执行的是同一段代码，它们也会各自在自己的工作内存中创建属于当前线程的本地变量，当然也包括了字节码行号指示器、相关Native方法的信息。注意由于工作内存是每个线程的私有数据，线程间无法相互访问工作内存，因此存储在工作内存的数据不存在线程安全问题。

根据JVM虚拟机规范主内存与工作内存的数据存储类型以及操作方式，对于一个实例对象中的成员方法而言，如果方法中包含本地变量是基本数据类型（boolean,byte,short,char,int,long,float,double），将直接存储在工作内存的帧栈结构中，但倘若本地变量是引用类型，那么该变量的引用会存储在功能内存的帧栈中，而对象实例将存储在主内存(共享数据区域，堆)中。但对于实例对象的成员变量，不管它是基本数据类型或者包装类型(Integer、Double等)还是引用类型，都会被存储到堆区。至于static变量以及类本身相关信息将会存储在主内存中。需要注意的是，**在主内存中的实例对象可以被多线程共享，倘若两个线程同时调用了同一个对象的同一个方法，那么两条线程会将要操作的数据拷贝一份到自己的工作内存中，执行完成操作后才刷新到主内存**，模型如下图所示：



三、JMM内存模型用来解决什么问题?

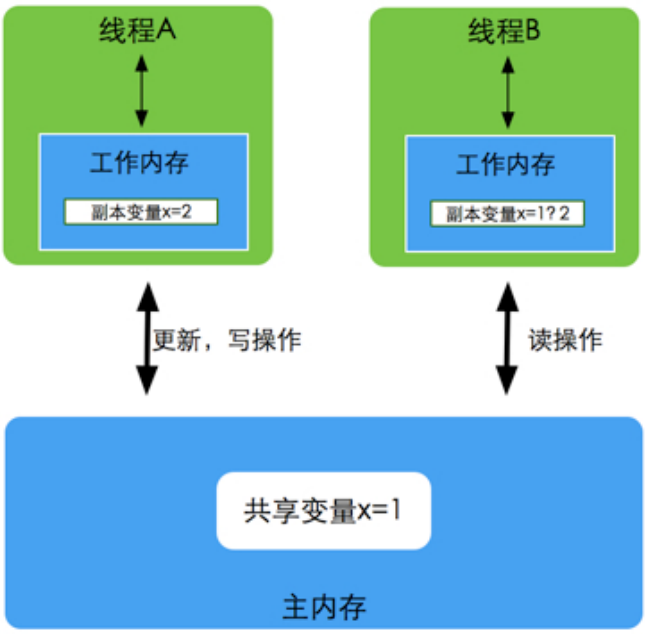
JMM主要解决的问题： 解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题

- 缓存一致性问题其实就是**可见性问题。**
- 处理器优化是可以导致**原子性问题**
- **指令重排即会导致有序性问题**

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，线程与主内存中的变量操作必须通过工作内存间接完成，主要过程是将变量从主内存拷贝到每个线程各自的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，如果存在两个线程同时对一个主内存中的实例对象的变量进行操作就有可能诱发线程安全问题。

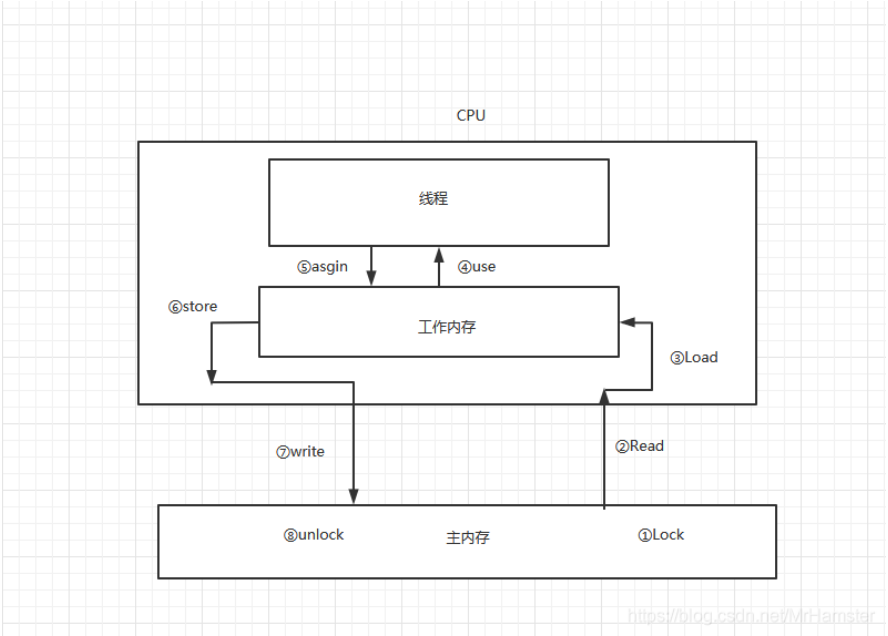
假设主内存中存在一个共享变量x，现在有A和B两条线程分别对该变量x=1进行操作，A/B线程各自的工作内存中存在共享变量副本x。假设现在A线程想要修改x的值为2，而B线程却想要读取x的值，那么B线程读取到的值是A线程更新后的值2还是更新前的值1呢？答案是，不确定，即B线程有可能读取到A线程更新前的值1，也有可能读取到A线程更新后的值2，这是因为工作内存是每个线程私有的数据区域，而线程A操作变量x时，首先是将变量从主内存拷贝到A线程的工作内存中，然后对变量进行操作，操作完成后再将变量x写回主内，而对于B线程的也是类似的，这样就有可能造成主内存与工作内存间数据存在一致性问题，假如A线程修改完后正在将数据写回主内存，而B线程此时正在读取主内存，即将x=1拷贝到自己的工作内存中，这样B线程读取到的值就是x=1，但如果A线程已将x=2写回主内存后，B线程才开始读取的话，那么此时B线程读取到的就是x=2，但到底是哪种情况先发生呢？

如以下示例图所示案例：



以上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。

数据同步八大原子操作



（1）lock(锁定)：作用于主内存的变量，把一个变量标记为一条线程独占状态

（2）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他线程锁定

（3）read(读取)：作用于主内存的变量，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

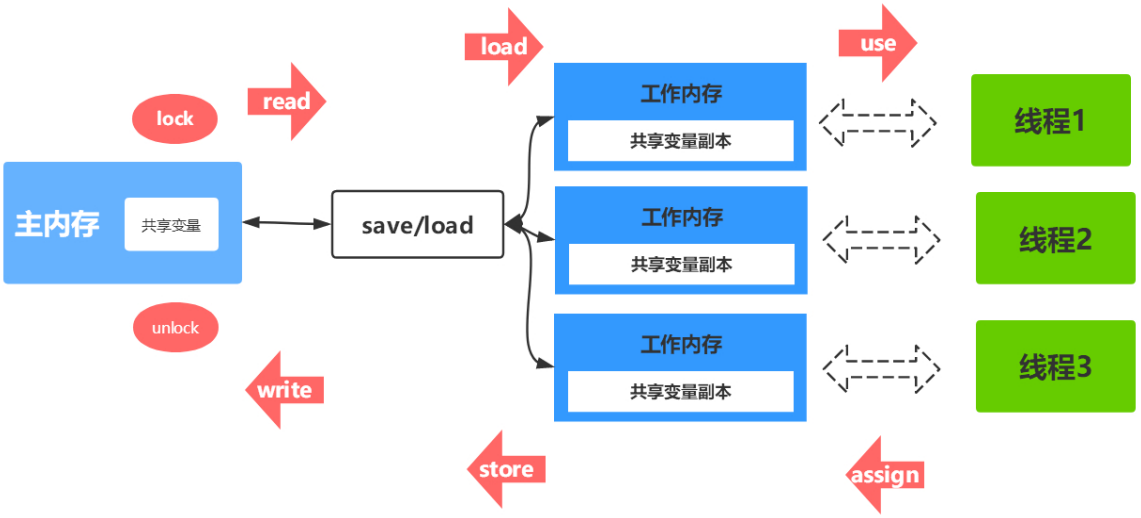
（4）load(载入)：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中

（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

（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给工作内存的变量

（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)：作用于工作内存的变量，它把store操作从工作内存中的一个变量的值传送到主内存的变量中



同步规则分析

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

2）一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或者assign）的变量。即就是对一个变量实施use和store操作之前，必须先自行assign和load操作。

3）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。lock和unlock必须成对出现。

4）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之前需要重新执行load或assign操作初始化变量的值。

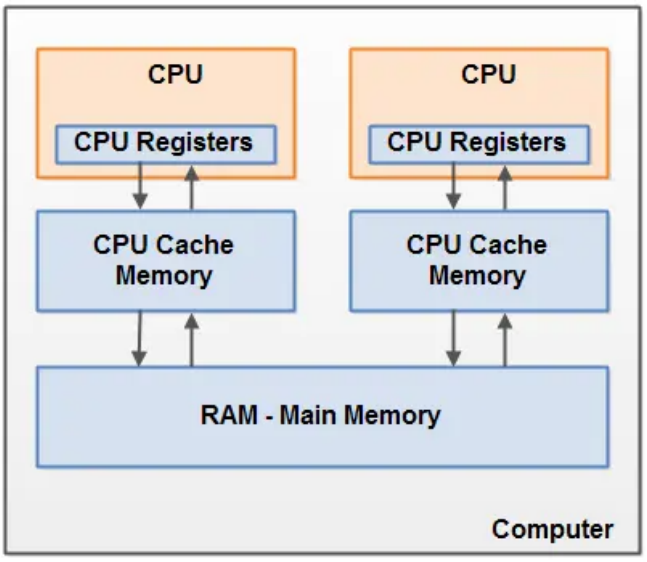
5）如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。

6）对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）

四、JMM内存模型与JVM内存模型有什么关系?

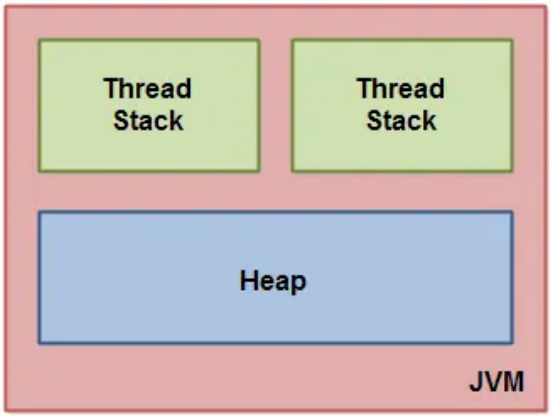
1、JMM是Java物理内存模型

现代计算机通常有 2 个或更多 CPU。其中一些 CPU 也可能具有多个内核。关键是，在具有 2 个或更多 CPU 的现代计算机上，可能同时运行多个线程。每个 CPU 能够在任何给定时间运行一个线程。这意味着如果您的 Java 应用程序是多线程的，则每个 CPU 一个线程可能会在您的 Java 应用程序中同时（并发）运行



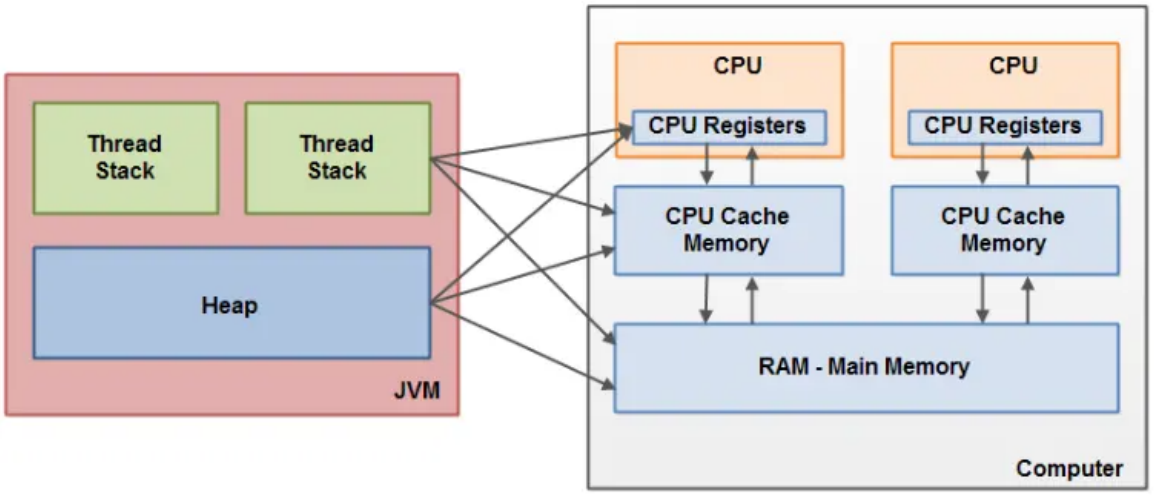
2、JVM是运行时内存模型

在JVM内部使用java内存模型将内存分为线程栈和堆，Java 虚拟机中运行的每个线程都有自己的线程堆栈。线程堆栈包含有关线程调用哪些方法以到达当前执行点的信息。我将其称为“调用堆栈”。当线程执行其代码时，调用堆栈会发生变化。



3、JMM内存模型和JVM运行时内存模型的关系

Java运行时内存模型和计算机物理内存结构是不一样的。计算机物理内存结构并不区分栈和堆。在物理内存结构中，栈和堆都位于主存中。一些线程栈和堆有时候可能在CPU寄存器或缓存器中，像下面这张图这样



当对象和变量可以保存在计算机内存不同区域中，会发生一些严重问题，两个主要方面是：

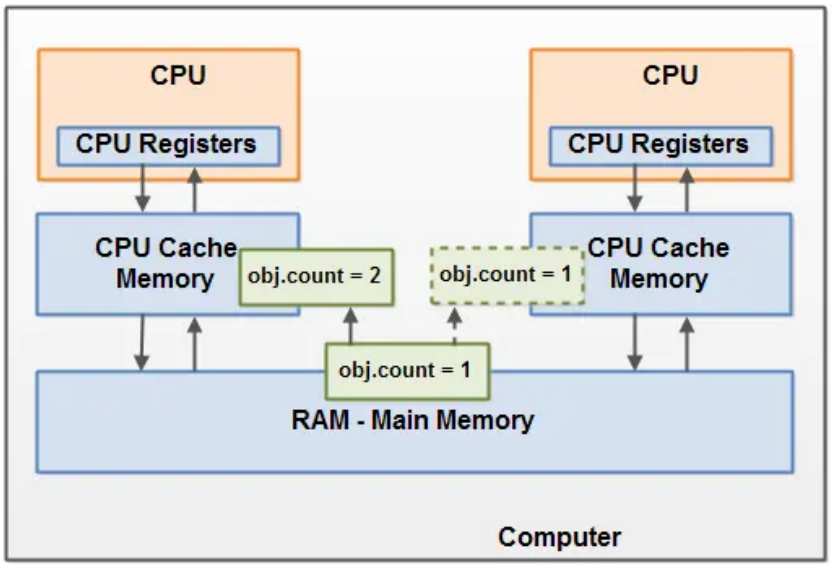
- 线程更新（写）共享变量时的可见性
- 检查和读写共享变量时的竞态条件

共享对象的可见性

如果两个或更多线程共享一个对象，若没有正确地使用volatile或者synchronization的话，一个线程更新了共享变量，但其他线程可能并不可见。

设想共享变量最初存放在主存中。其中一个CPU中的线程将共享对象读取进CPU的缓存器，并修改了共享变量。只要这个CPU寄存器没有将数据写回主存，则共享变量的修改对于在其他CPU中运行的线程不可见。这样每个线程结束时都有自己版本的对象副本，分别保存在各自CPU的缓存器中。

下面这张图描述了上面的情景。左边CPU中的一个线程将共享对象拷贝进自己的缓存器，并且将变量count变成2。这个改变对右边CPU中的线程不可见，因为左边对count的更新还没有回写到主存中。



你可以用Java关键字volatile来解决这个问题。**这个关键字能保证所给的变量总是直接从主存中读取，并总是在更新后马上回写主存。**

竞态条件

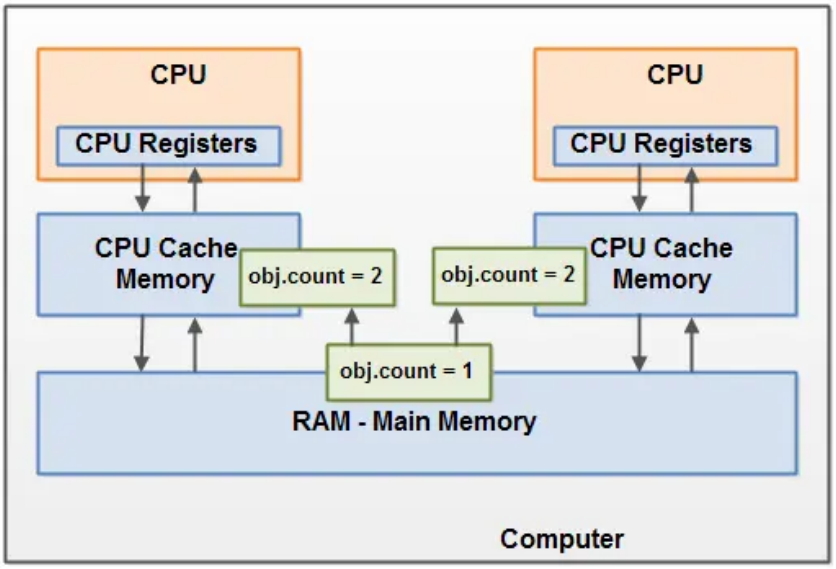
如果两个或者更多线程共享一个对象，并且有超过一个线程更新这个共享对象，就会发生竞争条件。

设想线程A读将共享对象的变量count读入它的CPU缓存器中。同时，线程B也做了相同的事，但是读进了不同CPU的缓存器中。现在线程A将count加1，线程B也加1。现在变量应该被加了两次，每个CPU各一次。

如果加法操作有序进行，变量就会被加两次，最终写回主存的变量应该是被加2.

但是，两次加法同时执行了，而且没有很好地进行同步控制。无论AB哪个线程将自己更新后的变量回写到主存，更新的变量都只会比原来大1，虽然事实上是两个现在一共做了两次加法操作。

下面这张图描述了上面所说的竞态条件：



你可以用Java synchronied代码块解决上面的问题。synchronized代码块可以保证在任何时间都只能有一个线程进入代码块。synchronized代码块还能保证所有在synchronized代码块中的变量都从主存中读取，当线程中存在synchronized代码块时，不管变量是否用volatile关键字修饰，所有的更新都会回写到主存中。