http://blog.csdn.net/suifeng3051/article/details/52611310

Java内存模型即**Java** Memory Model，简称JMM。JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。

如果我们要想深入了解Java并发编程，就要先理解好Java内存模型。Java内存模型定义了多线程之间共享变量的可见性以及如何在需要的时候对共享变量进行同步。原始的Java内存模型效率并不是很理想，因此Java1.5版本对其进行了重构，现在的Java8仍沿用了Java1.5的版本。

关于并发编程

在并发编程领域，有两个关键问题：线程之间的**通信**和**同步**。

线程之间的通信

线程的通信是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种**共享内存**和**消息传递**。

在**共享内存**的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过**共享对象**进行通信。

在**消息传递**的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是**wait()**和**notify()**。

关于Java线程之间的通信，可以参考线程之间的通信（thread signal）。

线程之间的同步

同步是指程序用于控制不同线程之间操作发生相对顺序的机制。

在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

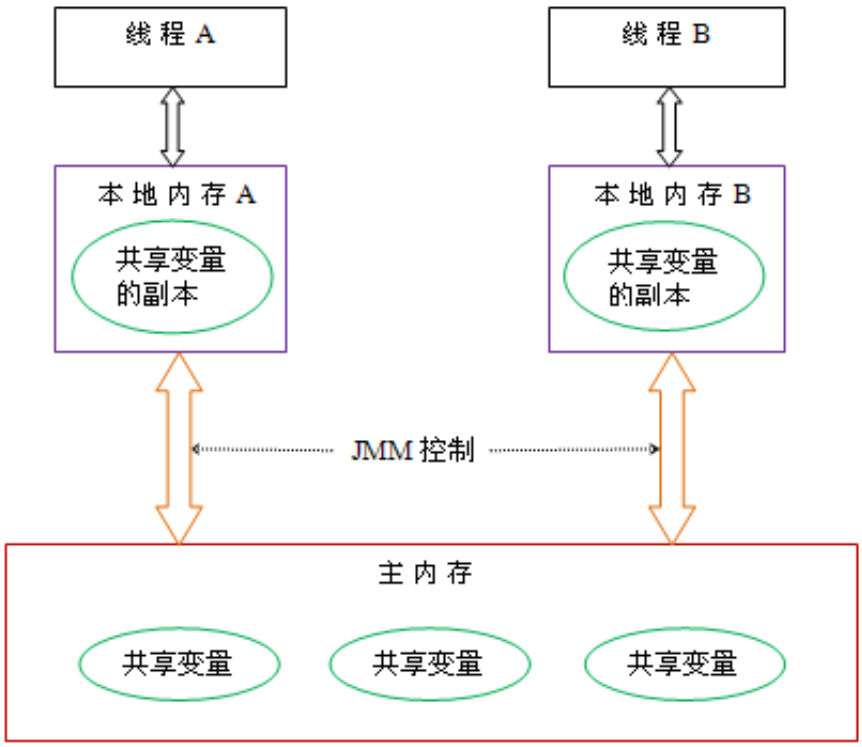
Java的并发采用的是共享内存模型

Java线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。如果编写多线程程序的Java程序员不理解隐式进行的线程之间通信的工作机制，很可能会遇到各种奇怪的内存可见性问题。

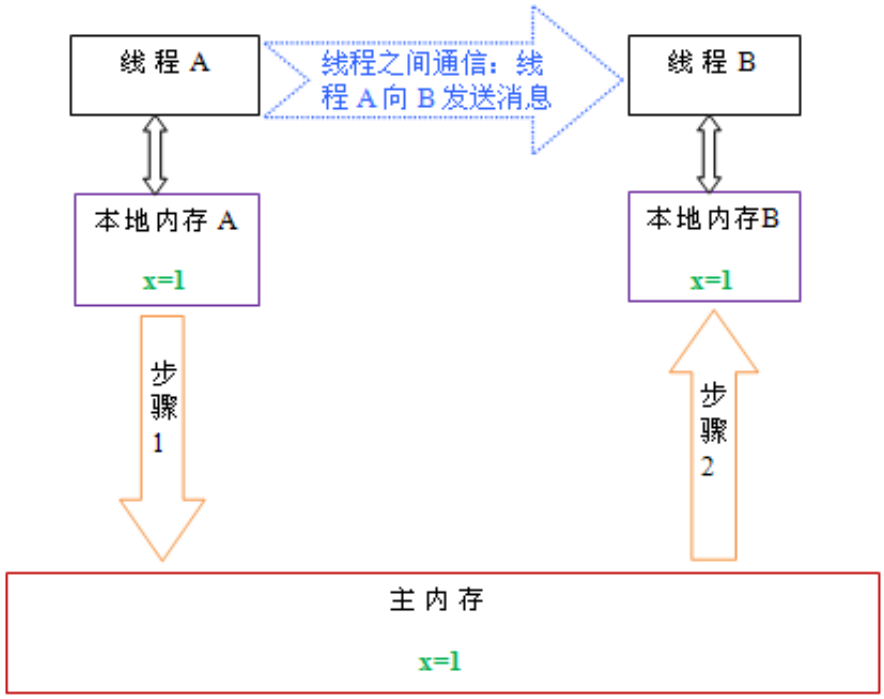
Java内存模型

上面讲到了Java线程之间的通信采用的是过共享内存模型，这里提到的共享内存模型指的就是Java内存模型(简称JMM)，**JMM决定一个线程对共享变量的写入何时对**

另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：**线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。**本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：
1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。
下面通过示意图来说明这两个步骤：



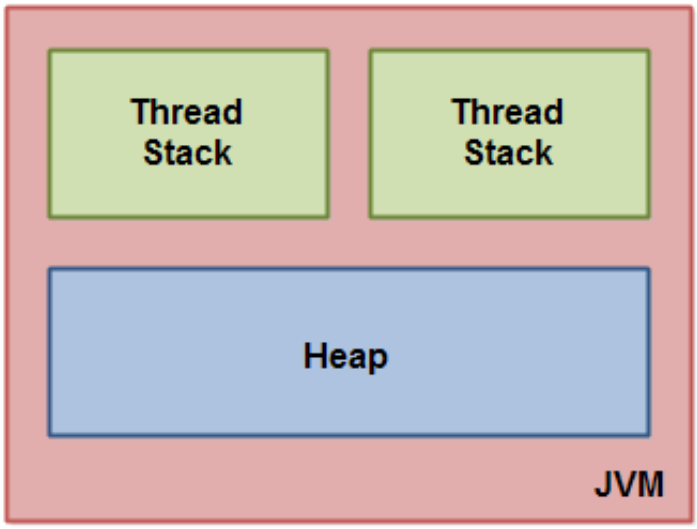
如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

上面也说到了，Java内存模型只是一个抽象概念，那么它在Java中具体是怎么工作的呢？为了更好的理解上Java内存模型工作方式，下面就JVM对Java内存模型的实现、硬件内存模型及它们之间的桥接做详细介绍。

JVM对Java内存模型的实现

在JVM内部，Java内存模型把内存分成了两部分：线程栈区和堆区，下图展示了Java内存模型在JVM中的逻辑视图：



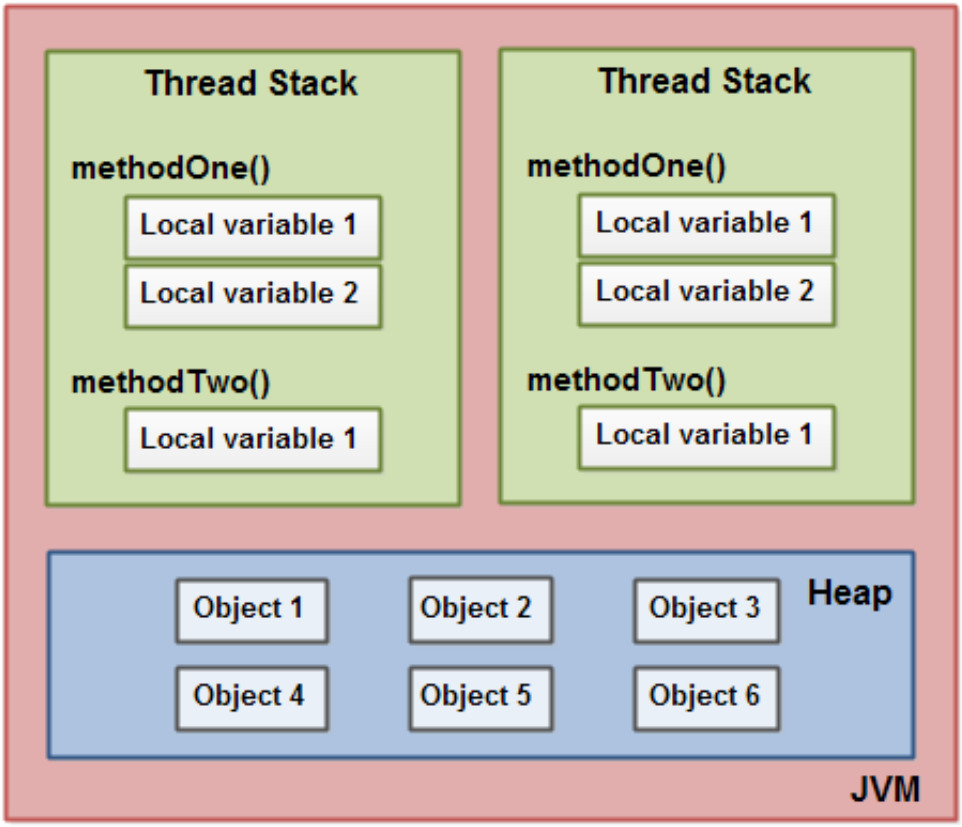
JVM中运行的每个线程都拥有自己的线程栈，线程栈包含了当前线程执行的方法调用相关信息，我们也把它称作调用栈。随着代码的不断执行，调用栈会不断变化。

线程栈还包含了当前方法的所有本地变量信息。一个线程只能读取自己的线程栈，也就是说，线程中的本地变量对其它线程是不可见的。即使两个线程执行的是同一段代码，它们也会各自在自己的线程栈中创建本地变量，因此，每个线程中的本地变量都会有自己的版本。

所有原始类型(boolean,byte,short,char,int,long,float,double)的本地变量都直接保存在线程栈当中，对于它们的值各个线程之间都是独立的。对于原始类型的本地变量，一个线程可以传递一个副本给另一个线程，当它们之间是无法共享的。

堆区包含了Java应用创建的所有对象信息，不管对象是哪个线程创建的，其中的对象包括原始类型的封装类（如Byte、Integer、Long等等）。不管对象是属于一个成员变量还是方法中的本地变量，它都会被存储在堆区。

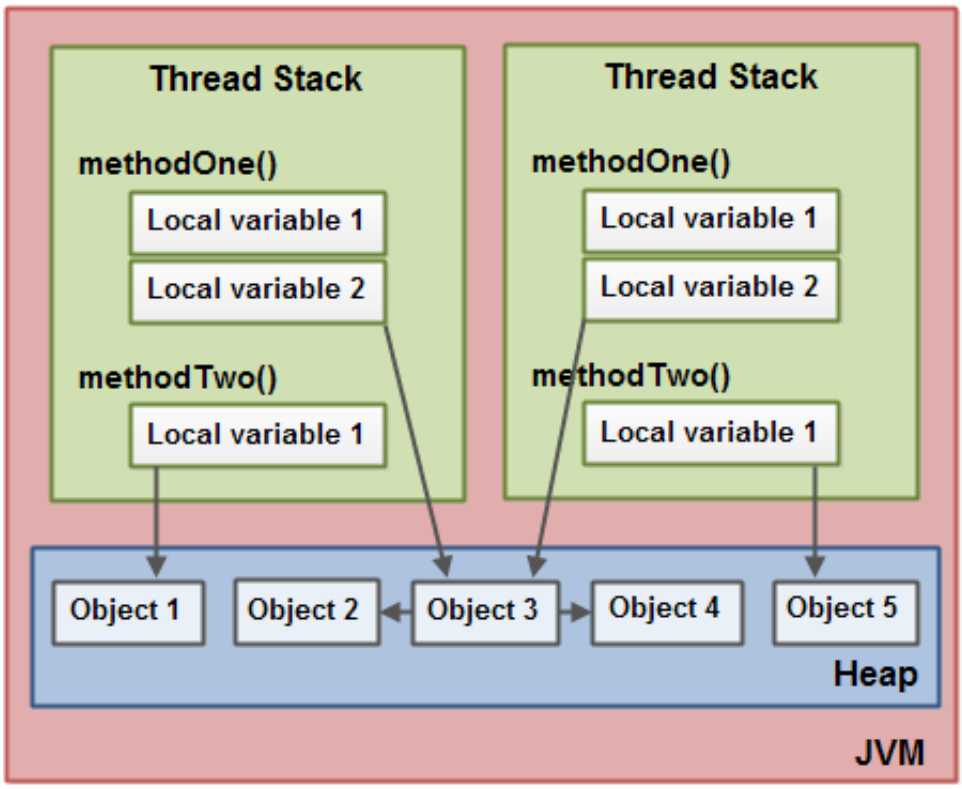
下图展示了调用栈和本地变量都存储在栈区，对象都存储在堆区：



1. 一个本地变量如果是原始类型，那么它会被完全存储到栈区。
2. 一个本地变量也有可能是一个对象的引用，这种情况下，这个本地引用会被存储到栈中，但是对象本身仍然存储在堆区。
3. 对于一个对象的成员方法，这些方法中包含本地变量，仍需要存储在栈区，即使它们所属的对象在堆区。
4. 对于一个对象的成员变量，不管它是原始类型还是包装类型，都会被存储到堆区(方法区)。
5. static类型的变量以及类本身相关信息都会随着类本身存储在堆区。

堆中的对象可以被多线程共享。如果一个线程获得一个对象的应用，它便可访问这个对象的成员变量。如果两个线程同时调用了同一个对象的同一个方法，那么这两个线程便可同时访问这个对象的成员变量，但是对于本地变量，每个线程都会拷贝一份到自己的线程栈中。

下图展示了上面描述的过程:



硬件内存架构

不管是什么内存模型，最终还是运行在计算机硬件上的，所以我们有必要了解计算机硬件内存**架构**，下图就简单描述了当代计算机硬件内存架构：

现代计算机一般都有2个以上CPU，而且每个CPU还有可能包含多个核心。因此，如果我们的应用是多线程的话，这些线程可能会在各个CPU核心中并行运行。

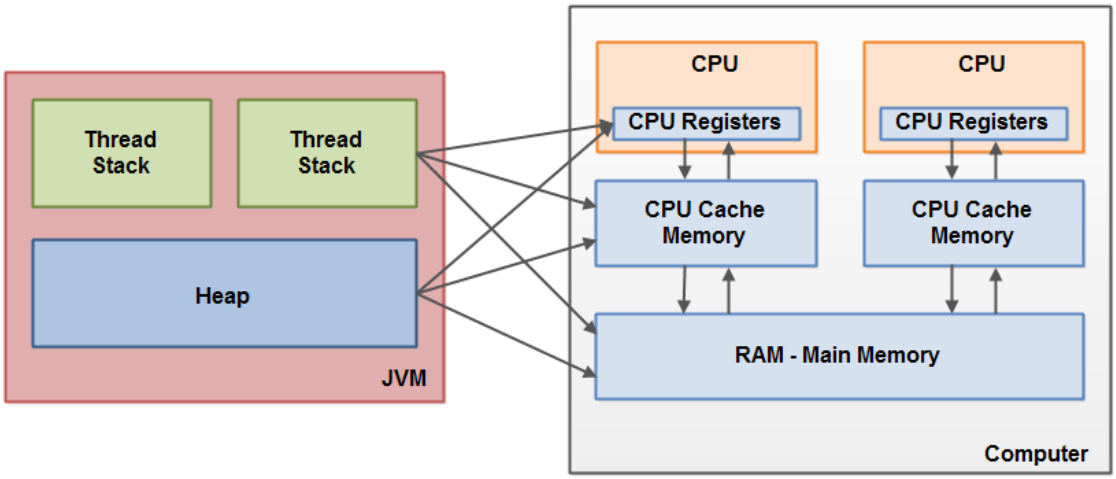
在CPU内部有一组CPU寄存器，也就是CPU的储存器。CPU操作寄存器的速度要比操作计算机主存快的多。在主存和CPU寄存器之间还存在一个CPU缓存，CPU操作CPU缓存的速度快于主存但慢于CPU寄存器。某些CPU可能有多个缓存层（一级缓存和二级缓存）。计算机的主存也称作RAM，所有的CPU都能够访问主存，而且主存比上面提到的缓存和寄存器大很多。

当一个CPU需要访问主存时，会先读取一部分主存数据到CPU缓存，进而在读取CPU缓存到寄存器。当CPU需要写数据到主存时，同样会先flush寄存器到CPU缓存，然后再在某些节点把缓存数据flush到主存。

Java内存模型和硬件架构之间的桥接

正如上面讲到的，Java内存模型和硬件内存架构并不一致。硬件内存架构中并没有区分栈和堆，从硬件上看，不管是栈还是堆，大部分数据都会存到主存中，当然一部分栈和堆的数据也有可能会存到CPU寄存器中，如下图所示，Java内存模型和计算机硬

件内存架构是一个交叉关系：



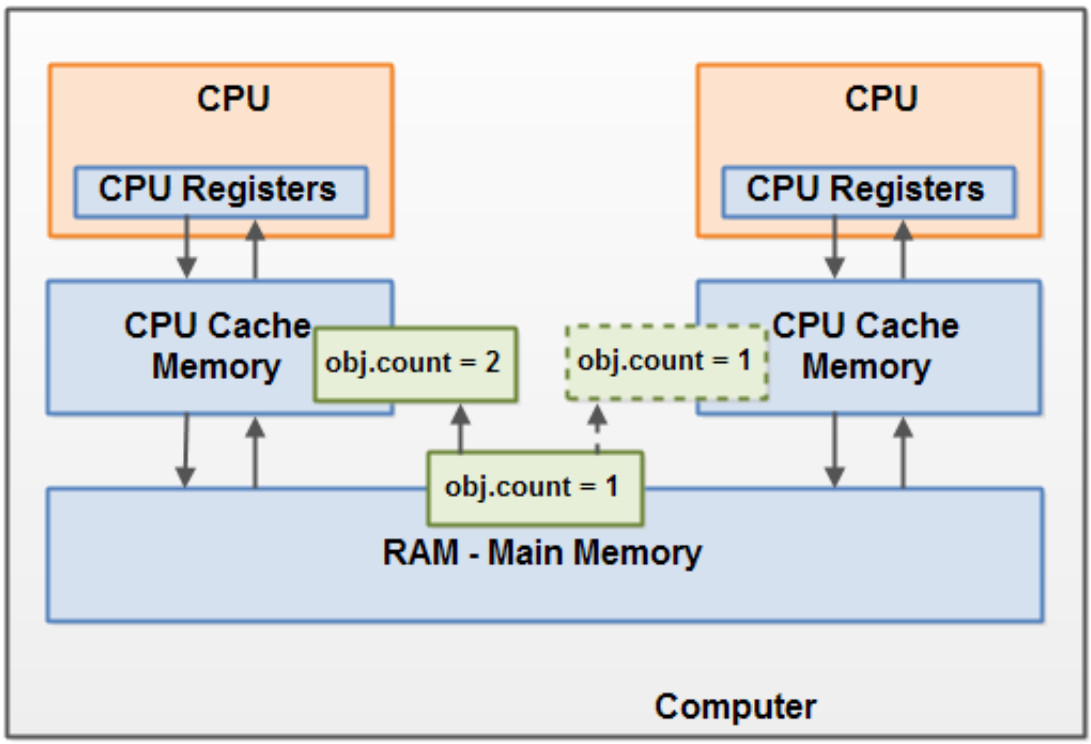
当对象和变量存储到计算机的各个内存区域时，必然会面临一些问题，其中最主要的两个问题是：

1. 共享对象对各个线程的可见性
2. 共享对象的竞争现象

共享对象的可见性

当多个线程同时操作同一个共享对象时，如果没有合理的使用volatile和synchronization关键字，一个线程对共享对象的更新有可能导致其它线程不可见。想象一下我们的共享对象存储在主存，一个CPU中的线程读取主存数据到CPU缓存，然后对共享对象做了更改，但CPU缓存中的更改后的对象还没有flush到主存，此时线程对共享对象的更改对其它CPU中的线程是不可见的。最终就是每个线程最终都会拷贝共享对象，而且拷贝的对象位于不同的CPU缓存中。

下图展示了上面描述的过程。左边CPU中运行的线程从主存中拷贝共享对象obj到它的CPU缓存，把对象obj的count变量改为2。但这个变更对运行在右边CPU中的线程不可见，因为这个更改还没有flush到主存中：



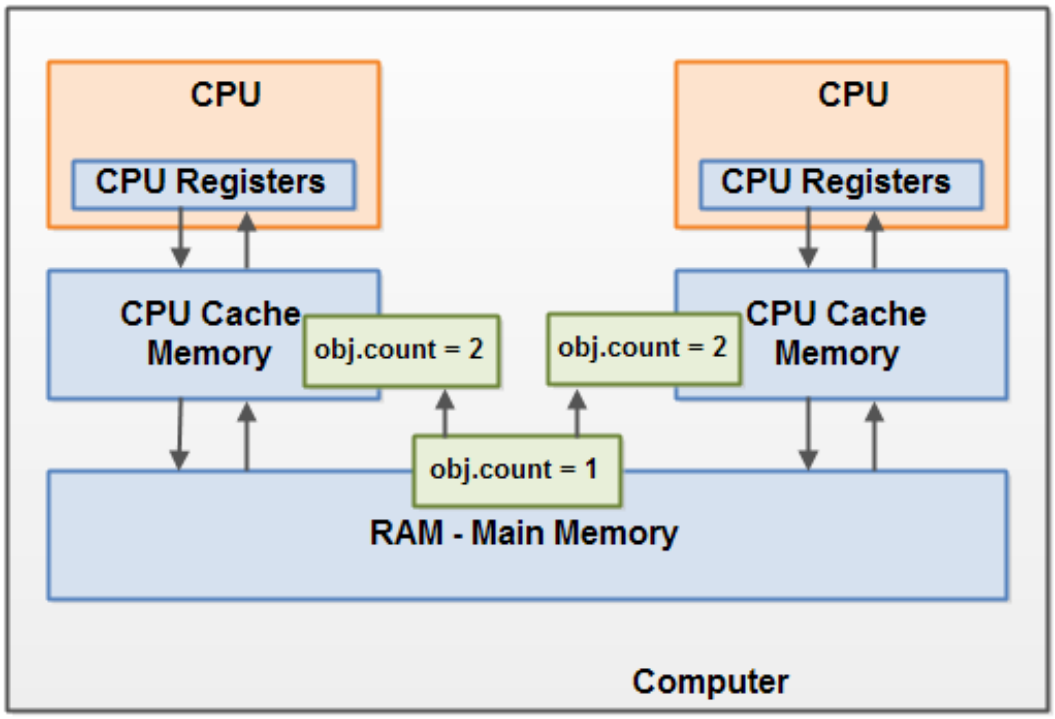
要解决共享对象可见性这个问题，我们可以使用java volatile关键字。 Java’s volatile keyword. volatile 关键字可以保证变量会直接从主存读取，而对变量的更新也会直接写到主存。volatile原理是基于CPU内存屏障指令实现的，后面会讲到。

竞争现象

如果多个线程共享一个对象，如果它们同时修改这个共享对象，这就产生了竞争现象。

如下图所示，线程A和线程B共享一个对象obj。假设线程A从主存读取Obj.count变量到自己的CPU缓存，同时，线程B也读取了Obj.count变量到它的CPU缓存，并且这两个线程都对Obj.count做了加1操作。此时，Obj.count加1操作被执行了两次，不过都在不同的CPU缓存中。

如果这两个加1操作是串行执行的，那么Obj.count变量便会在原始值上加2，最终主存中的Obj.count的值会是3。然而下图中两个加1操作是并行的，不管是线程A还是线程B先flush计算结果到主存，最终主存中的Obj.count只会增加1次变成2，尽管一共有两次加1操作。



要解决上面的问题我们可以使用java synchronized代码块。synchronized代码块可以保证同一个时刻只能有一个线程进入代码竞争区，synchronized代码块也能保证代码块中所有变量都将会从主存中读，当线程退出代码块时，对所有变量的更新将会flush到主存，不管这些变量是不是volatile类型的。

volatile和 synchronized区别

详细请见 volatile和synchronized的区别

支撑Java内存模型的基础原理

指令重排序

在执行程序时，为了提高性能，编译器和处理器会对指令做重排序。但是，JMM确保在不同的编译器和不同的处理器平台之上，通过插入特定类型的Memory Barrier来禁止特定类型的编译器重排序和处理器重排序，为上层提供一致的内存可见性保证。

1. 编译器优化重排序：编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序：如果不存l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：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。

数据依赖性

如果两个操作访问同一个变量，其中一个为写操作，此时这两个操作之间存在数据依赖性。

编译器和处理器不会改变存在数据依赖性关系的两个操作的执行顺序，即不会重排序。

as-if-serial

不管怎么重排序，单线程下的执行结果不能被改变，编译器、runtime和处理器都必须遵守as-if-serial语义。

内存屏障（Memory Barrier ）

上面讲到了，通过内存屏障可以禁止特定类型处理器的重排序，从而让程序按我们预想的流程去执行。内存屏障，又称内存栅栏，是一个CPU指令，基本上它是一条这样的指令：

1. 保证特定操作的执行顺序。
2. 影响某些数据（或则是某条指令的执行结果）的内存可见性。

编译器和CPU能够重排序指令，保证最终相同的结果，尝试优化性能。插入一条Memory Barrier会告诉编译器和CPU：不管什么指令都不能和这条Memory Barrier指令重排序。

Memory Barrier所做的另外一件事是强制刷出各种CPU cache，如一个Write-Barrier（写入屏障）将刷出所有在Barrier之前写入 cache 的数据，因此，任何CPU上的线程都能读取到这些数据的最新版本。

这和java有什么关系？上面java内存模型中讲到的volatile是基于Memory Barrier实现的。

如果一个变量是volatile修饰的，JMM会在写入这个字段之后插进一个Write-Barrier指令，并在读这个字段之前插入一个Read-Barrier指令。这意味着，如果写入一个volatile变量，就可以保证：

1. 一个线程写入变量a后，任何线程访问该变量都会拿到最新值。
2. 在写入变量a之前的写入操作，其更新的数据对于其他线程也是可见的。因为Memory Barrier会刷出cache中的所有先前的写入。

happens-before

从jdk5开始，java使用新的JSR-133内存模型，基于happens-before的概念来阐述操作之间的内存可见性。

在JMM中，如果一个操作的执行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，这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。

与程序员密切相关的happens-before规则如下：

1. 程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。
2. 监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。
3. volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

4. 传递性规则：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，并不意味前一个操作必须要在后一个操作之前执行！仅仅要求前一个操作的执行结果，对于后一个操作是可见的，且前一个操作按顺序排在后一个操作之前。