**Java Memory Model**

* [The Internal Java Memory Model](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#javas-logic-memory-model)
* [Hardware Memory Architecture](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#hardware-memory-architecture)
* [Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#bridging-the-gap-between-the-java-memory-model-and-the-hardware-memory-architecture)
  + [Visibility of Shared Objects](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#visibility-of-shared-objects)
  + [Race Conditions](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#race-conditions)

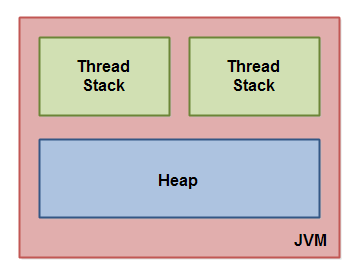
Java内存模型指定Java虚拟机如何与计算机的内存（RAM）一起工作。Java虚拟机是整个计算机的一个模型，所以这个模型自然包含一个内存模型-亦称Java内存模型。

如果想正确地设计并发程序，那么理解Java内存模型是非常重要的。Java内存模型指定如何以及何时不同线程可以看到其他线程写入共享变量的值，以及如何在必要时同步对共享变量的访问。

原来的Java内存模型不足，因此Java内存模型在Java 1.5中进行了修改。Java内存模型的这个版本仍在Java 8中使用。

**The Internal Java Memory Model**

JVM内部使用的Java内存模型在线程栈和堆之间划分内存。这个图从逻辑的角度说明了Java内存模型：



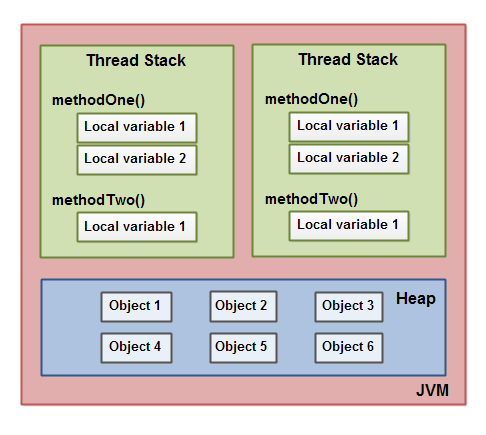
在Java虚拟机中运行的每个线程都有自己的线程栈。线程栈包含关于线程需要调用哪些方法以达到当前执行点的信息。我将称之为“调用栈”。当线程执行其代码时，调用栈会发生变化。

线程栈还包含正在执行的每个方法(调用栈上的所有方法)的所有局部变量。线程只能访问它自己的线程栈。由线程创建的局部变量对所有其他线程(而不是创建线程的线程)都是不可见的。即使两个线程执行完全相同的代码，这两个线程仍然会在各自的线程栈中创建该代码的局部变量。因此，对于每个局部变量，每个线程都有自己的版本。

所有基本类型(boolean、byte、short、char、int、long、float、double)的局部变量都完全存储在线程栈中，因此其他线程不可见。一个线程可以将基本类型变量的副本传递给另一个线程，但是它不能共享基本类型局部变量本身。

堆包含Java应用程序中创建的所有对象，而不管什么线程创建了这个对象。这包括基本类型的对象版本(例如Byte、Integer、Long等)。创建对象并将其分配给本地变量或创建为其他对象的成员变量并不重要，对象仍然存储在堆中。

下面是一个图，说明存储在线程栈上的调用栈和局部变量，以及存储在堆上的对象：



局部变量可以是基本类型，在这种情况下，它完全保持在线程栈上。

局部变量也可以是对象的引用。在这种情况下，引用（局部变量）存储在线程栈上，但是对象本身存储在堆上。

对象可以包含方法，这些方法可以包含局部变量。这些本地变量也存储在线程栈中，即使该方法所属的对象存储在堆上。

对象的成员变量与对象本身一起存储在堆中。当成员变量是基本类型，或它是对对象的引用时，这都是正确的。

静态类变量也与类定义一起存储在堆中。

堆上的对象可以由具有对象引用的所有线程访问。当线程访问对象时，它也可以访问该对象的成员变量。如果两个线程同时调用同一对象上的方法，那么它们都将访问该对象的成员变量，但是每个线程都有自己的局部变量副本。

下面是一个说明以上观点的图表：



两个线程有一组局部变量。局部变量之一(Local Variable 2)指向堆(Object3)上的共享对象。这两个线程每个都对同一个对象有不同的引用。它们的引用是局部变量，因此存储在每个线程的线程堆栈上(在每个线程上)。不过两个不同的引用指向堆上的相同对象。

注意共享对象(Object3)如何将Object2和Object4作为成员变量(由从Object3到Object2和Object4的箭头说明)的引用。通过Object3中的这些成员变量引用，两个线程可以访问Object2和Object4。

该图还显示了一个指向堆上的两个不同对象的局部变量。在这种情况下，引用指向两个不同的对象(Object1和Object5)，而不是同一对象。理论上，两个线程都可以访问Object1和Object5，如果两个线程都引用了这两个对象。但是在上面的图中，每个线程只对两个对象中的一个引用。

那么，什么样的Java代码可以导致上面的内存图？代码和下面的代码一样简单：

public class MyRunnable implements Runnable() {

public void run() {

methodOne();

}

public void methodOne() {

int localVariable1 = 45;

MySharedObject localVariable2 =

MySharedObject.sharedInstance;

//... do more with local variables.

methodTwo();

}

public void methodTwo() {

Integer localVariable1 = new Integer(99);

//... do more with local variable.

}

}

public class MySharedObject {

//static variable pointing to instance of MySharedObject

public static final MySharedObject sharedInstance =

new MySharedObject();

//member variables pointing to two objects on the heap

public Integer object2 = new Integer(22);

public Integer object4 = new Integer(44);

public long member1 = 12345;

public long member1 = 67890;

}

如果两个线程正在执行run()方法，那么前面显示的图表将是输出。run()方法调用methodOne()，methodOne()调用方法methodTwo()。

methodOne()声明一个基本类型局部变量(类型为int的localVariable1)和一个作为对象引用的局部变量(localVariable2)。

每个执行methodOne()的线程将在各自的线程堆栈上创建自己的localVariable1和localVariable2的副本。localVariable1变量将彼此完全分离，只存在于每个线程的线程栈上。一个线程无法看到另一个线程对它的localVariable1的副本所做的更改。

执行方法methodOne()的每个线程也将创建它们自己的localVariable2副本。但是，两个不同的localVariable2副本最终指向堆上的相同对象。该代码将localVariable2设置为指向静态变量引用的对象。静态变量只有一个副本，这个副本存储在堆上。因此，localVariable2的两个副本都指向静态变量指向的MySharedObject的同一个实例。MySharedObject实例也存储在堆上。它对应于上面的图表中的Object3。

请注意MySharedObject 类也包含两个成员变量。成员变量本身与对象一起存储在堆中。两个成员变量指向两个其他Integer 对象。这些Integer 对象对应于上述图中的Object2和Object4。

还要注意methodTwo()如何创建一个名为localVariable1的局部变量。此局部变量是对Integer对象的对象引用。该方法将localVariable1引用指向新的Integer实例。localVariable1引用将被存储在每个执行methodTwo()的线程的一个副本中。实例化的两个Integer对象将存储在堆中，但由于该方法每次执行时都创建一个新的Integer对象，所以执行此方法的两个线程将创建单独的Integer实例。在methodTwo()中创建的Integer对象对应于上面的图中的Object1和Object5。

还注意类型为long类型(这是一个基本类型)的MySharedObject类中的两个成员变量。因为这些变量是成员变量，所以它们仍然与对象一起存储在堆中。只有局部变量存储在线程堆栈上。

**Hardware Memory Architecture**

现代硬件内存体系结构与内部Java内存模型有些不同。了解硬件内存结构也是非常重要的，以理解Java内存模型是如何工作的。本节描述了常见的硬件内存体系结构，后面的部分将描述Java内存模型是如何工作的。

这里是现代计算机硬件体系结构的简图：



现代计算机通常有2个或更多的CPU。这些CPU中的一些也可能有多个内核。关键在于，在具有2个或更多CPU的现代计算机上，可以同时运行多个线程。每个CPU能够在任何给定的时间运行一个线程。这意味着，如果Java应用程序是多线程的，则每个CPU的一个线程可能同时(并发)在Java应用程序内部运行。

每个CPU都包含一组寄存器，这些寄存器基本上是CPU存储器。CPU在这些寄存器上的运行速度比在主存储器上执行的速度快得多。这是因为CPU可以比访问主存储器更快地访问这些寄存器。

每个CPU还可以具有CPU高速缓冲存储器层。事实上，大多数现代CPU都有某种大小的高速缓冲存储器层。CPU可以比主存储器更快地访问它的高速缓冲存储器，但是通常不像它访问它的内部寄存器那么快。因此，CPU高速缓冲存储器在内部寄存器和主存储器的速度之间。一些CPU可能有多个缓存层（1级和2级），但是了解Java内存模型如何与内存交互是不那么重要的。重要的是要知道CPU可以有某种类型的缓存存储层。

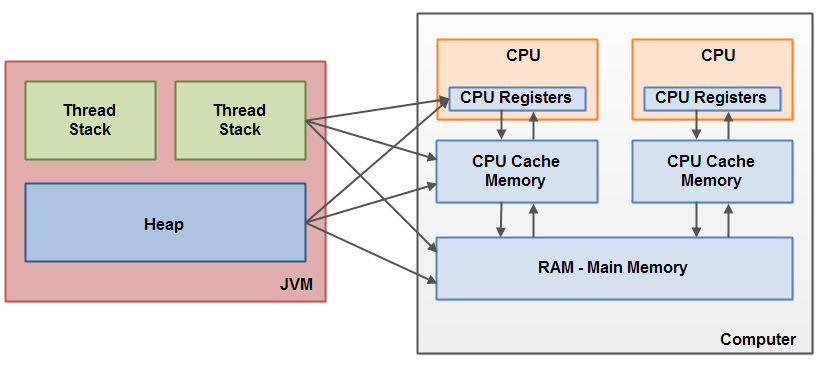
计算机还包含主存储器区域(RAM)。所有CPU都可以访问主内存。主内存区域通常比CPU的高速缓冲存储器大得多。

通常，当CPU需要访问主存储器时，它将读取部分内存到CPU缓存中。它甚至可以读取部分缓存到其内部寄存器，然后对其执行操作。当CPU需要将结果写回主存储器时，它将把值从其内部寄存器刷新到高速缓存存储器，并在某个时候将值刷新回到主存储器。

当CPU需要在高速缓存内存中存储其他内容时，存储在高速缓存内存中的值通常被刷新回主内存。CPU高速缓存可以一次将数据写入其存储器的一部分，并一次刷新其存储器的一部分。它不必在每次更新时读取/写入整个高速缓存。通常，缓存在被称为“缓存行”的较小内存块中更新。可以将一个或多个高速缓存行读入高速缓存存储器，并且可以将一个或多个高速缓存行再次刷新回主存储器。

**Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture**

正如已经提到的，Java内存模型和硬件内存结构是不同的。硬件内存体系结构不区分线程栈和堆。在硬件上，线程栈和堆都位于主内存中。线程栈和堆的某些部分有时可能存在于CPU缓存和内部CPU寄存器中。图中说明了这一点：



当对象和变量可以存储在计算机的各种不同的存储区域中时，可能出现某些问题。这两个主要问题是：

* 线程更新(写入)到共享变量的可见性
* 读取、检查和写入共享变量时的竞争条件

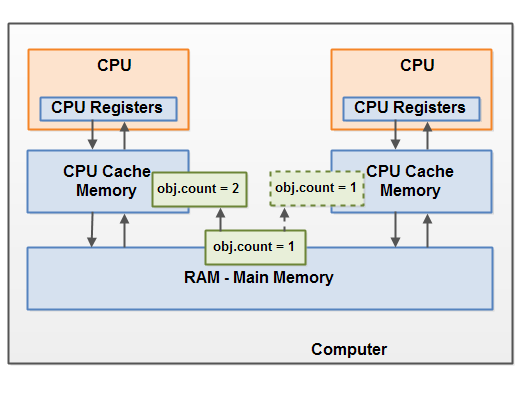
这两个问题将在下面的章节中解释。

**Visibility of Shared Objects**

如果两个或多个线程共享一个对象，而没有正确使用volatile声明或同步，则由一个线程对共享对象的更新对其他线程可能是不可见的。

假设共享对象最初存储在主内存中。在CPU上运行的线程然后将共享对象读入CPU缓存。在那里，它对共享对象进行了更改。只要CPU缓存没有被刷新回主内存，共享对象的更改版本对其他CPU上运行的线程是不可见的。通过这种方式，每个线程都可能有自己的共享对象副本，每个副本都位于不同的CPU缓存中。

下面的图表说明了草图的情况。在左CPU上运行的一个线程将共享对象复制到CPU缓存中，并将其count 变量更改为2。此更改对于在右CPU上运行的其他线程是不可见的，因为count的更新尚未刷新回主内存。



为了解决这个问题，可以使用Java的volatile关键字([**Java's volatile keyword**](http://tutorials.jenkov.com/java-concurrency/volatile.html))。volatile关键字可以确保直接从主存储器读取给定变量，并且在更新时总是写回主存储器。

**Race Conditions**

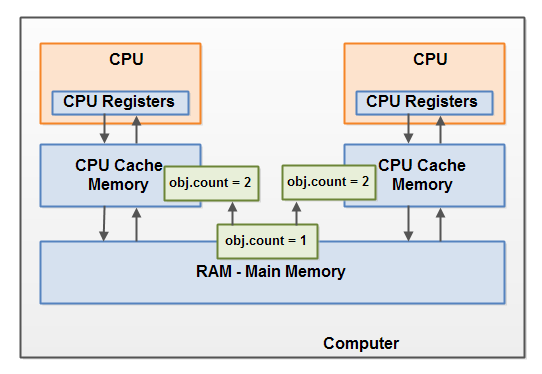
如果两个或多个线程共享一个对象，并且多个线程更新该共享对象中的变量，则可能出现竞争条件([**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html))。

想象一下，如果线程A将共享对象的变量count 读入CPU缓存。想象一下，线程B也一样，但是进入了不同的CPU缓存。现在线程A加上一个数到count，线程B也一样。现在，var1已经增加了两次，在每个CPU缓存中各一次。

如果这些增量是顺序执行的，那么变量计数将增加两次，并将原始值+2写回主存储器。

然而，这两个增量是在没有适当同步的情况下同时进行的。不管哪个线程A和B将更新后的count版本写回主内存，更新后的值将仅比原始值高1，尽管增加了两次。

该图说明了上面所述的竞争条件出现的问题：



为了解决这个问题，可以使用Java同步块([**Java synchronized block**](http://tutorials.jenkov.com/java-concurrency/synchronized.html))。同步块保证在任何给定时间只有一个线程可以进入代码的给定关键部分。同步块还保证从主存储器读入同步块内访问的所有变量，当线程退出同步块时，所有更新的变量将再次刷新回主存储器，而不管变量是否被声明为volatile。