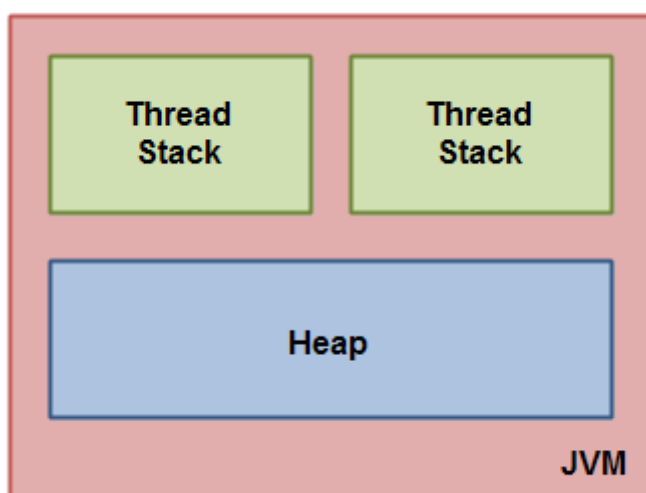


JVM 基础 - Java 内存模型引入

JMM引入

从堆栈说起

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

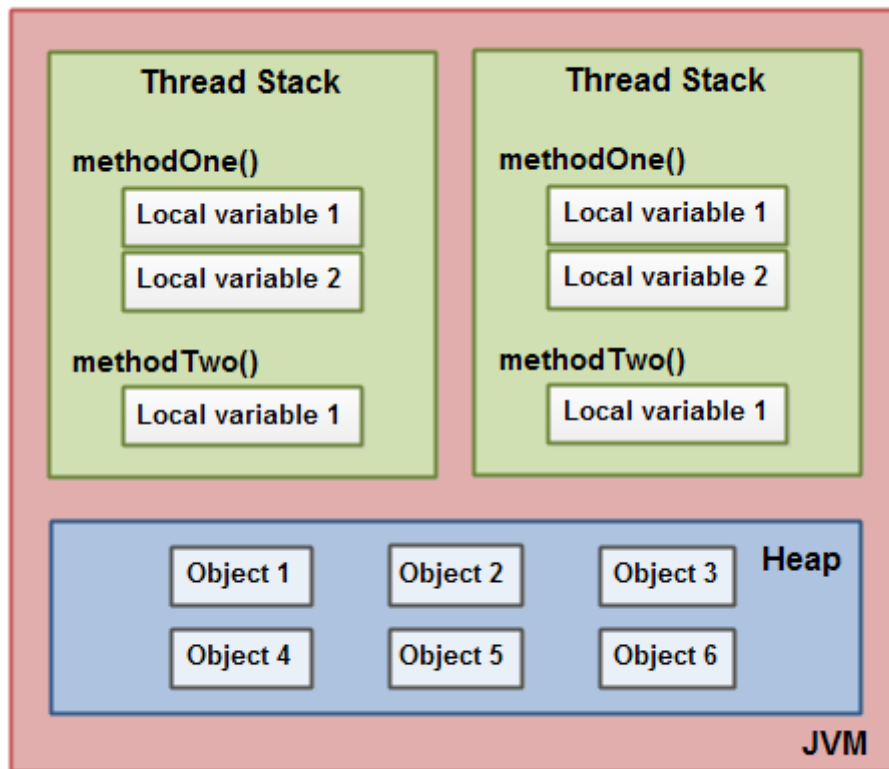


堆栈里面放了什么？

线程堆栈还包含正在执行的每个方法的所有局部变量(调用堆栈上的所有方法)。线程只能访问它自己的线程堆栈。由线程创建的局部变量对于创建它的线程以外的所有其他线程是不可见的。即使两个线程正在执行完全相同的代码，两个线程仍将在每个自己的线程堆栈中创建该代码的局部变量。因此，每个线程都有自己的每个局部变量的版本。

基本类型的所有局部变量(boolean, byte, short, char, int, long, float, double)完全存储在线程堆栈中，因此对其他线程不可见。一个线程可以将一个基本类型变量的副本传递给另一个线程，但它不能共享原始局部变量本身。

堆包含了在Java应用程序中创建的所有对象，无论创建该对象的线程是什么。这包括基本类型的包装类(例如 Byte, Integer, Long等)。无论是创建对象并将其分配给局部变量，还是创建为另一个对象的成员变量，该对象仍然存储在堆上。



局部变量可以是基本类型，在这种情况下，它完全保留在线程堆栈上。

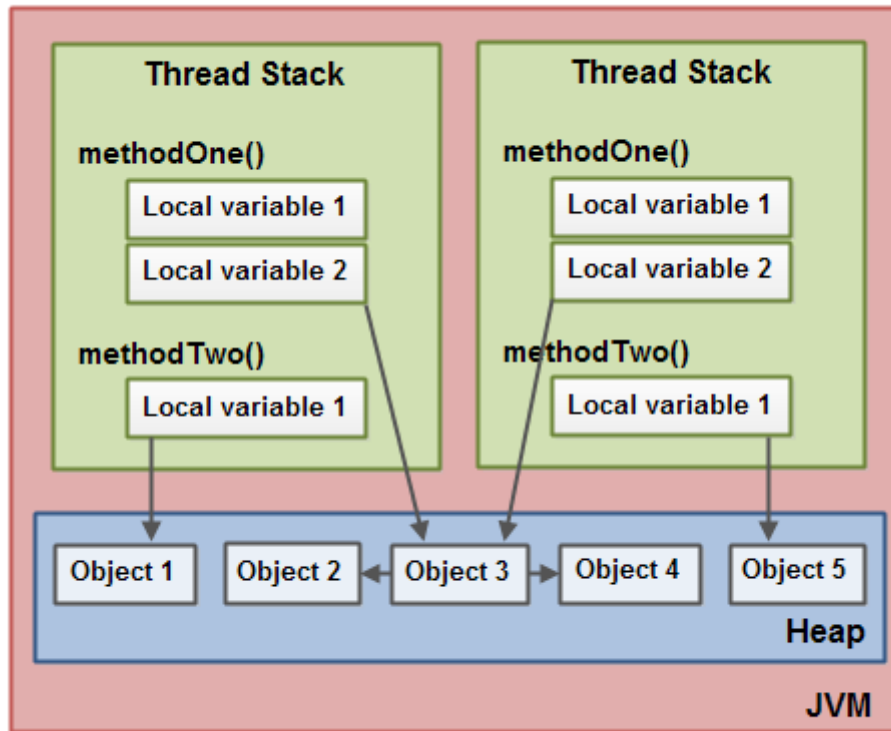
局部变量也可以是对象的引用。在这种情况下，引用(局部变量)存储在线程堆栈中，但是对象本身存储在堆(Heap)上。

对象的成员变量与对象本身一起存储在堆上。当成员变量是基本类型时，以及它是对象的引用时都是如此。

静态类变量也与类定义一起存储在堆上。

线程栈如何访问堆上对象？

所有具有对象引用的线程都可以访问堆上的对象。当一个线程有权访问一个对象时，它也可以访问该对象的成员变量。如果两个线程同时在一个对象上调用一个方法，它们都可以访问该对象的成员变量，但每个线程都有自己的局部变量副本。



两个线程有一组局部变量。其中一个局部变量(局部变量2)指向堆上的共享对象(对象3)。两个线程各自对同一对象具有不同的引用。它们的引用是局部变量，因此存储在每个线程的线程堆栈中(在每个线程堆栈上)。但是，这两个不同的引用指向堆上的同一个对象。

注意共享对象(对象3)如何将对象2和对象4作为成员变量引用(由对象3到对象2和对象4的箭头所示)。通过对象3中的这些成员变量引用，两个线程可以访问对象2和对象4。

该图还显示了一个局部变量，该变量指向堆上的两个不同对象。在这种情况下，引用指向两个不同的对象(对象1和对象5)，而不是同一个对象。理论上，如果两个线程都引用了两个对象，则两个线程都可以访问对象1和对象5。但是在上图中，每个线程只引用了两个对象中的一个。

线程栈访问堆示例

那么，什么样的Java代码可以导致上面的内存图？好吧，代码就像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，则前面显示的图表将是结果。run()方法调用methodOne()，methodOne()调用methodTwo()。

methodOne()声明一个局部基本类型变量(类型为int的localVariable1)和一个局部变量，它是一个对象引用(localVariable2)。

执行methodOne()的每个线程将在各自的线程堆栈上创建自己的localVariable1和localVariable2副本。localVariable1变量将完全相互分离，只存在于每个线程的线程堆栈中。一个线程无法看到另一个线程对其localVariable1副本所做的更改。

执行methodOne()的每个线程也将创建自己的localVariable2副本。但是，localVariable2的两个不同副本最终都指向堆上的同一个对象。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，此副本存储在堆上。因此，localVariable2的两个副本最终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。它对应于上图中的对象3。

注意MySharedObject类还包含两个成员变量。成员变量本身与对象一起存储在堆上。两个成员变量指向另外两个Integer对象。这些Integer对象对应于上图中的Object 2和Object 4。

另请注意methodTwo()如何创建名为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引用设置为指向新的Integer实例。localVariable1引用将存储在执行methodTwo()的每个线程的一个副本中。实例化的两个Integer对象将存储在堆上，但由于该方法每次执行该方法时都会创建一个新的Integer对象，因此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对象对应于上图中的Object 1和Object 5。

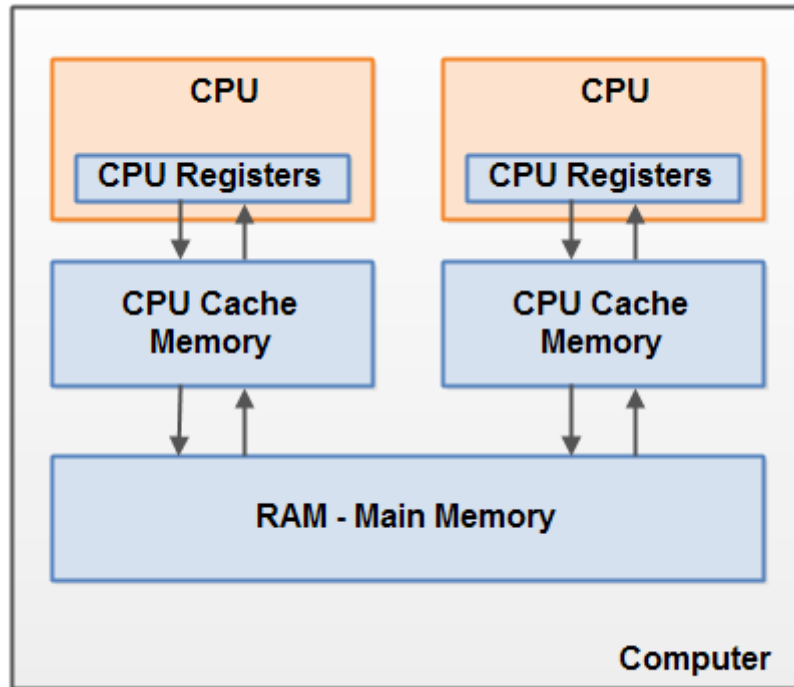
另请注意类型为long的MySharedObject类中的两个成员变量，它们是基本类型。由于这些变量是成员变量，因此它们仍与对象一起存储在堆上。只有局部变量存储在线程堆栈中。

JMM与硬件内存结构关系

硬件内存结构简介

现代硬件内存架构与内部Java内存模型略有不同。了解硬件内存架构也很重要，以了解Java内存模型如何与其一起工作。本节介绍了常见的硬件内存架构，后面的部分将介绍Java内存模型如何与其配合使用。

这是现代计算机硬件架构的简化图：



现代计算机通常有2个或更多CPU。其中一些CPU也可能有多个内核。关键是，在具有2个或更多CPU的现代计算机上，可以同时运行多个线程。每个CPU都能够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，线程真的在可能同时运行。

每个CPU基本上都包含一组在CPU内存中的寄存器。CPU可以在这些寄存器上执行的操作比在主存储器中对变量执行的操作快得多。这是因为CPU可以比访问主存储器更快地访问这些寄存器。

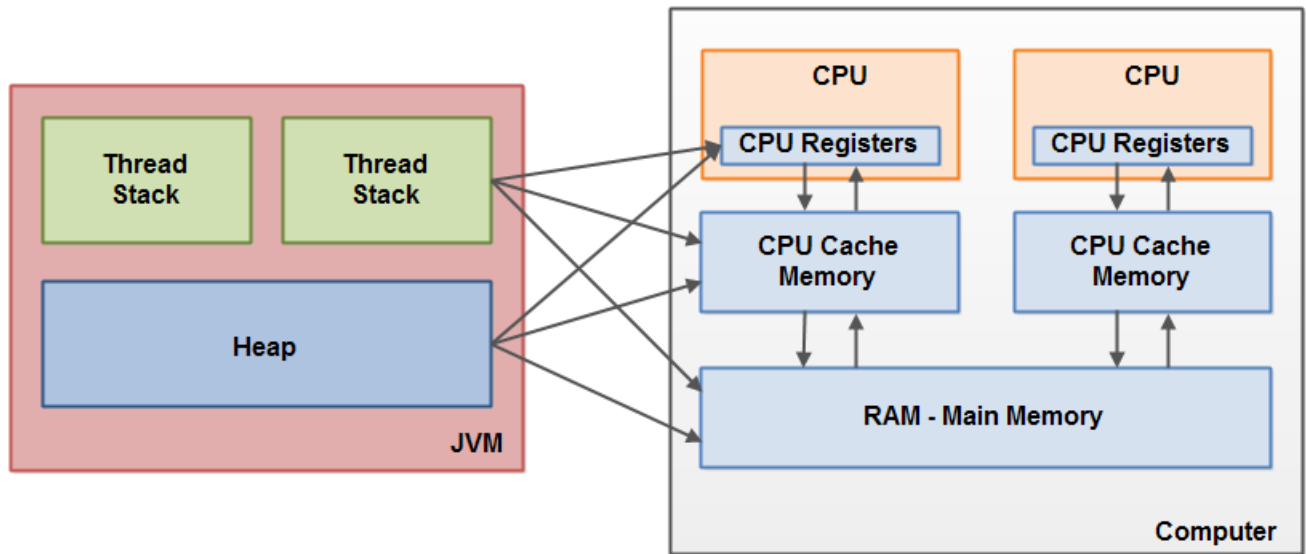
每个CPU还可以具有CPU高速缓存存储器层。事实上，大多数现代CPU都有一些大小的缓存存储层。CPU可以比主存储器更快地访问其高速缓存存储器，但通常不会像访问其内部寄存器那样快。因此，CPU高速缓存存储器介于内部寄存器和主存储器的速度之间。某些CPU可能有多个缓存层(级别1和级别2)，但要了解Java内存模型如何与内存交互，这一点并不重要。重要的是要知道CPU可以有某种缓存存储层。

计算机还包含主存储区(RAM)。所有CPU都可以访问主内存。主存储区通常比CPU的高速缓存存储器大得多。同时访问速度也就较慢。

通常，当CPU需要访问主存储器时，它会将部分主存储器读入其CPU缓存。它甚至可以将部分缓存读入其内部寄存器，然后对其执行操作。当CPU需要将结果写回主存储器时，它会将值从其内部寄存器刷新到高速缓冲存储器，并在某些时候将值刷新回主存储器。

JMM与硬件内存连接 - 引入

如前所述，Java内存模型和硬件内存架构是不同的。硬件内存架构不区分线程堆栈和堆。在硬件上，线程堆栈和堆都位于主存储器中。线程堆栈和堆的一部分有时可能存在于CPU高速缓存和内部CPU寄存器中。这在图中说明：



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

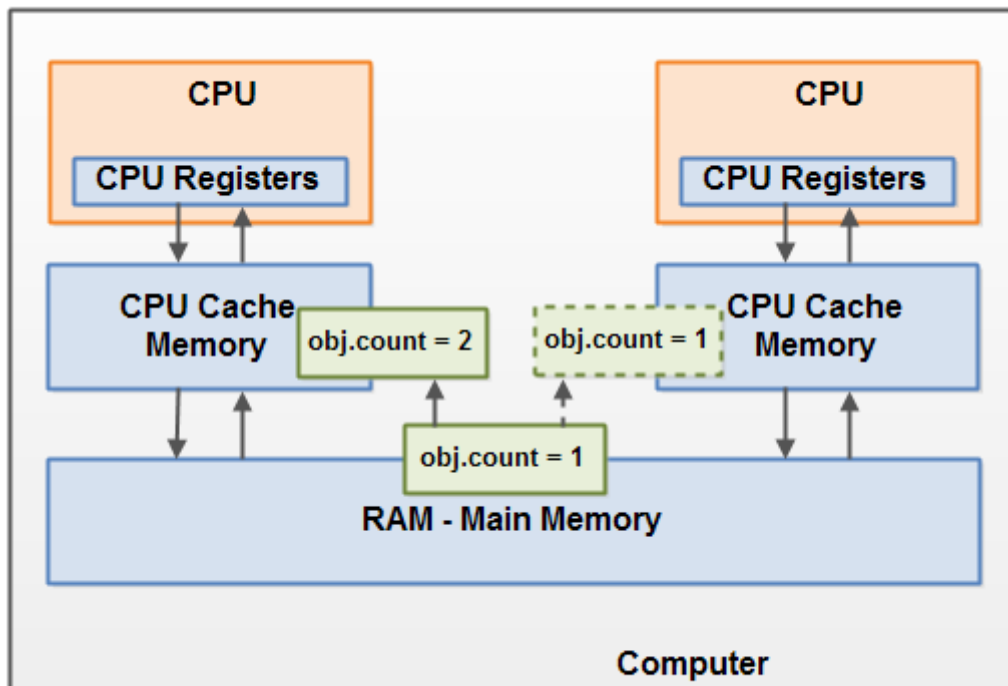
- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables. 以下各节将解释这两个问题。

JMM与硬件内存连接 - 对象共享后的可见性

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，则一个线程对共享对象的更新可能对其他线程不可见。

想象一下，共享对象最初存储在主存储器中。然后，在CPU上运行的线程将共享对象读入其CPU缓存中。它在那里对共享对象进行了更改。只要CPU缓存尚未刷新回主内存，共享对象的更改版本对于在其他CPU上运行的线程是不可见的。这样，每个线程最终都可能拥有自己的共享对象副本，每个副本都位于不同的CPU缓存中。

下图描绘了该情况。在左CPU上运行的一个线程将共享对象复制到你CPU缓存中，并将其count变量更改为2.对于在右边的CPU上运行的其他线程，此更改不可见，因为计数更新尚未刷新回主内存中。



要解决此问题，您可以使用Java的volatile关键字。volatile关键字可以确保直接从主内存读取给定变量，并在更新时始终写回主内存。

JMM与硬件内存连接 - 竞态条件

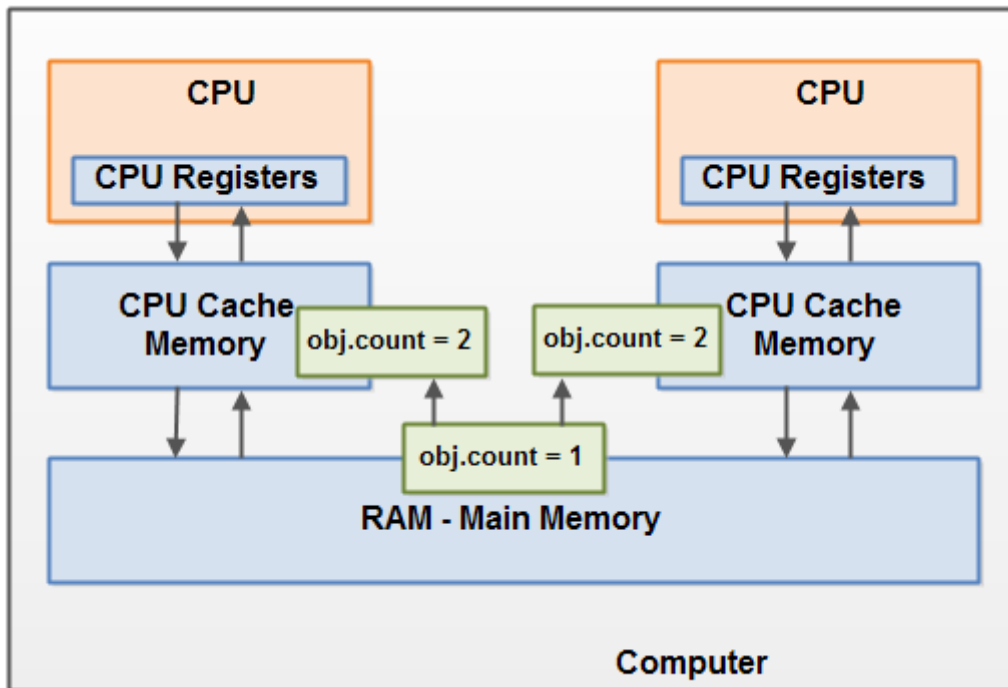
如果两个或多个线程共享一个对象，并且多个线程更新该共享对象中的变量，则可能会出现竞态。

想象一下，如果线程A将共享对象的变量计数读入其CPU缓存中。想象一下，线程B也做同样的事情，但是进入不同的CPU缓存。现在，线程A将一个添加到count，而线程B执行相同的操作。现在var1已经增加了两次，每个CPU缓存一次。

如果这些增量是按先后顺序执行的，则变量计数将增加两次并将原始值+ 2写回主存储器。

但是，两个增量同时执行而没有适当的同步。无论线程A和B中哪一个将其更新后的计数版本写回主存储器，更新的值将仅比原始值高1，尽管有两个增量。

该图说明了如上所述的竞争条件问题的发生：



要解决此问题，您可以使用Java `synchronized`块。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。同步块还保证在同步块内访问的所有变量都将从主存储器中读入，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，无论变量是不是声明为`volatile`