链滴

[译] 解析 Java 内存模型

作者：valarchie

原文链接：https://ld246.com/article/1576064109186

来源网站：链滴

看到一篇关于Java内存模型的技术文章于是翻译一下供大家学习交流。

原文地址：《Java Memory Model》

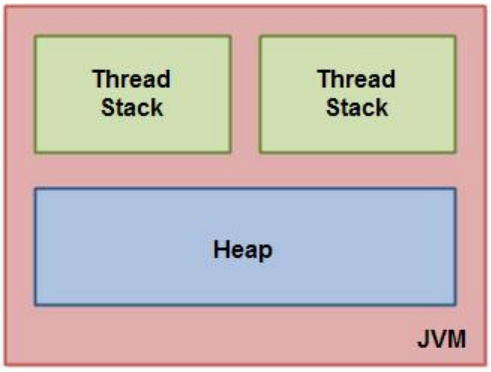
java内存模型指定了java虚拟机如何与计算机的内存（RAM）进行工作。java虚拟机是一整个计算机抽象模型，所以这个模型天然的包括了内存模型，它被称之为java内存模型。

如果你想设计正确运作的并发程序的话，那么理解java内存模型是非常重要的。Java内存模型指定了同线程如何以及何时可以看到其他线程写入共享变量的值，以及在必要时如何同步对共享变量的访问。

在最初版本的Java内存模型是不足的，所以在Java 1.5版本中进行了修正。这个版本的Java内存模型用至Java 8中。

Java内存模型的内部

在JVM的内部使用了Java内存模型将内存划分给线程栈和堆。下图从逻辑视角说明了Java内存模型。



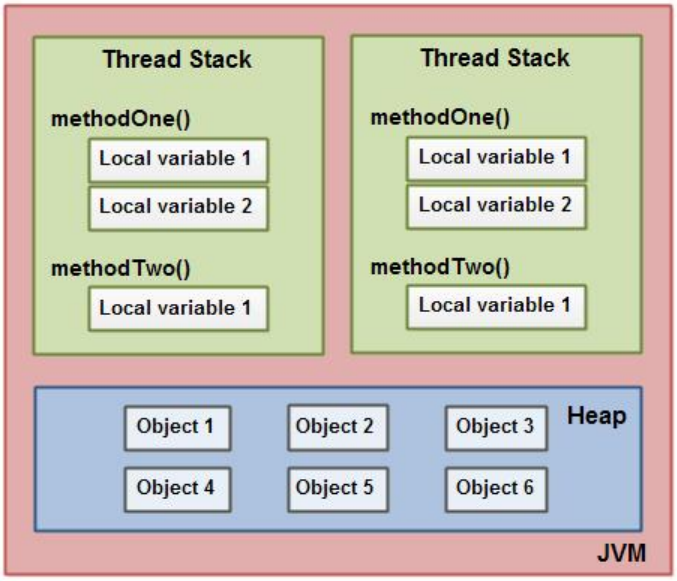
每一个运行在虚拟机的线程都拥有一个它独有的线程栈。线程栈包含了关于线程方法调用中已到达当执行点方法的信息。我们将它称之为调用栈。当线程执行到它的代码，它的调用栈就会改变。

线程栈同时包含了被调用的每一个方法的所有局部变量（所有方法都在调用栈上）。一个线程只能访它自己的线程栈。局部变量除了创建它们的线程之外，对于其他线程都是不可见的。甚至两个线程正执行同一段代码，这两个线程仍然会创建代码中的局部变量在它们各自的线程栈之中。因此，每个线都有每个自己版本的局部变量。

所有基本类型的本地变量（boolean, byte, short, char, int, long, float, double）都存储在线程栈中因此对其他线程都是不可见的。一个线程可能会传递一个基本类型变量的副本给另一个线程，但不能享它自己的局部基本类型变量。

而堆中包含了所有在你的Java应用程序中创建的对象，不管是哪个线程创建的对象。它包含了基本类的包装类型，如Byte、Integer。不管一个对象被创建并赋值给一个局部变量，还是被创建作为另一对象的成员变量，这个对象依然会被存储在堆当中。

下面这张图展示了调用栈和局部变量存储在线程栈之中，对象存储在堆当中。



一个局部变量可能是基本类型，在这种情况下，它们通通保存在线程栈之中。

一个局部变量可能是一个对象的引用，在这种情况下，引用被存储在线程栈之中，但是对象本身被存在堆当中。

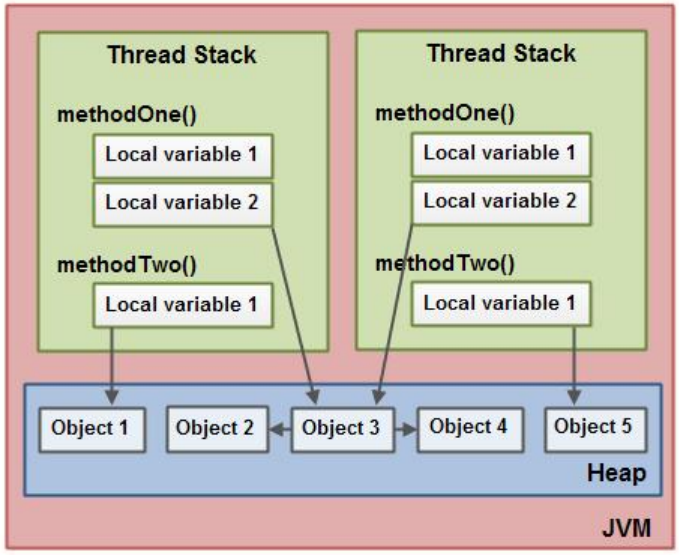
一个对象可能包含方法并且这些方法包含局部变量。这些局部变量也都存储在线程栈中，但方法所属对象存储在堆当中。

一个对象的成员变量与对象本身一样都被存储在堆当中。无论这个成员变量是基本类型还是一个对象引用。

静态类变量跟它的类定义一样也存储在堆当中。

在堆中的对象可以被所有拥有该对象引用的线程所访问。当一个线程访问一个对象，它也可以访问这对象的成员变量。如果两个线程同时调用同一个对象的同一个方法，它们都可以访问该对象的成员变，但每个线程都有自己的本地变量副本。

下面这张图展示了以上几点：



原文链接：[译] 解析 Java 内存模型

两个线程都有自己一组本地变量。其中一个局部变量Local Variable 2指向了堆中的共享对象Object 。两个线程都有一个不同的引用指向同一个对象。（这里不同指的是不同的局部引用地址变量如Local Variable 2）。它们的引用都是局部变量，因此被存储在他们各自的线程栈中。不过这两个不同的引指向了堆中同一个对象。

注意到这个共享对象Object 3拥有Object 2和 Object 4作为成员变量。在图中，Object 3有指向Obje t 2和Object 4的箭头。通过这些在Object 3中成员变量的引用，两个线程可以访问Object 2和Object 4。

在图示中展示了一个局部变量指向了堆中的不同对象。在这种情况下，引用指向了两个不同的对象（ bject 1 和Object 5），而并非同一个。理论上两个线程都可以访问Object 1和Object 5，如果它们有这两个对象的引用。但在图中每个线程值拥有两个对象其中的一个引用。

所以，在什么样的情况下会导致以上的内存图呢？我们用尽可能简单的代码演示。

```
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;
}
```

如果两个线程都执行了run()方法的话，那么结果就是刚才显示的图示。run()方法调用了methodOne(方法，methodOne方法调用了methodTwo()方法。

methodOne()方法声明了一个基本类型的局部变量int类型的localVariable1和一个指向对象的引用型局部变量localVariable2。

每一个执行了methodOne()方法的线程都会创建一份localVarialble1和localVariable2的副本，在它各自的线程栈之中。localVarialble1将会完全各自分离开来，仅存在于各自线程栈之中。一个线程不看到其他线程的localVariable1副本的变化。

每个线程执行methodOne()将会创建一份它们自己的localVariable2的副本。但是，两个不同的localariable2都指向堆中的同一个对象。代码设置localVariable2指向由静态变量引用的对象。静态变量有一份副本，并存储在堆中。因此localVariable2的两个副本都指向了MySharedObject的同一个静变量。MySharedObject也存储在堆中。它对应于上图中的Object 3。

注意MySharedObject也包含了两个成员变量。这两个成员变量本身与对象一同存储在堆中。这两个员变量指向了不同的Integer对象。这两个Integer对象对应于上图中的Object 2和Object 4。

注意到methodTwo()方法创建了一个命名为localVariable1的本地局部变量。这个局部变量是一个指Integer的对象引用。此方法将localVariable1指向一个新的Integer实例对象。每个线程执行methodwo()方法时，都会各自存储一份localVariable1引用副本。实例化后的两个Integer对象将会存储在堆，但是因为在方法每次执行时都会创建一个新的Integer对象，两个线程执行这个方法将会创建单独的Iteger实例。在methodTwo()方法中创建的Integer对象对应了上图中的Object 1和Object 5。

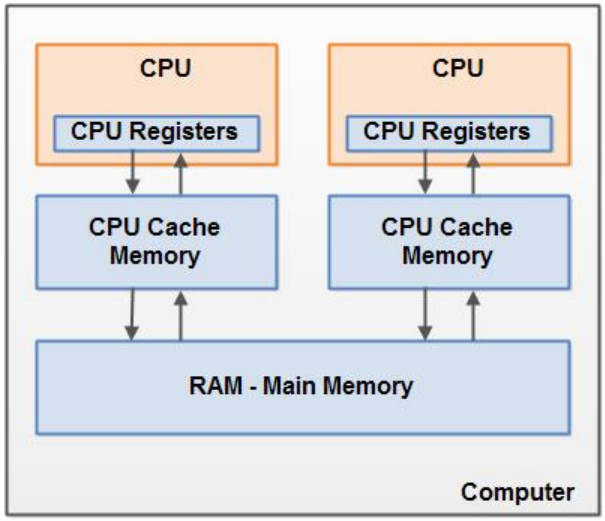
注意MySharedObject对象中的两个long基本类型成员变量，因为这两个都是成员变量，所以它们仍随着对象被存储在堆中。只有局部变量才会存储在线程栈之中。

关于这段描述，大家需要理解本地变量a与a指向的A对象是不同的概念。本地变量a存储在线程栈中，指向的A对象存储在堆中。

硬件内存结构

现代的硬件内存体系结构与Java内存模型存在一些不同。理解硬件内存体系结构也是非常重要的。理它是如何与Java内存模型一同工作的。本节介绍了常见的硬件内存结构，下一节我们将会讲述Java内模型如何与硬件内存结构一同工作。

以下是现代计算机硬件体系结构的简化图：



现代的计算机大多都拥有两个或者更多的cpu。这些cpu可能还是多核的。关键在于，拥有多cpu的现计算机中可以同时运行多个不同的线程。每个cpu都可以在任何时刻运行一个线程。它意味着如果你的ava程序是多线程的，在你的Java程序中，每个cpu可能同时并发运行着一个线程。

每个cpu都包含一组寄存器，这些寄存器基本上都在cpu的内存中。cpu在这些寄存器上执行操作比在存中的变量执行操作要快得多。这是因为cpu访问这些寄存器的速度比访问主存的速度要快得多。

每个cpu还具有一个cpu高速缓冲存储层。大多数现代cpu都有一定大小的缓存层。cpu访问它的缓存比主存要快得多，但通常不会比访问cpu内部寄存器快。cpu缓存的速度介于寄存器和主存的之间。些cpu可能还拥有多层缓存层如：L1和L2。但了解Java内存模型与主存如何交换并不重要。重要的是知道cpu存在某些类型的缓存层。

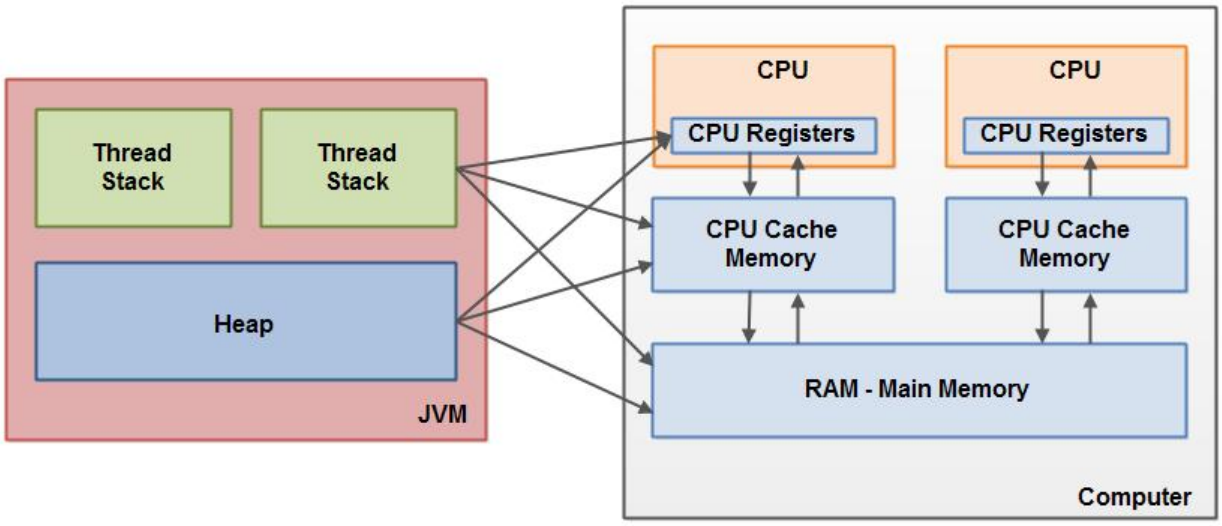
计算机包含一个主存，所有cpu都可以访问主存。主存通常来说会比cpu的缓存层大得多。

通常来说，当一个cpu需要去访问主存的时候，它会先将主存的一部分读入cpu的缓存层。它还会进步将cpu缓存层的一部分读入寄存器，并对其进行操作。当cpu需要将结果写回主存的时候，它会将从寄存器刷回到cpu缓存层，在某个时刻再将值刷回到主存中去。

当cpu缓存层需要存储其他值的时候，它会将原本cpu缓存层中的值刷回到主存中。cpu缓存层可以将据写入一部分内存，也可以刷新一部分内存。它不必每次更新时都读取整个完整的缓存。通常，cpu存层是由称之为cache line的更小的内存块进行更新的。一条或多条cache line可能从主存读进cpu缓层，而一条或多条cache line可能再次被刷回主存中。

Java内存模型和硬件内存模型

正如以上提到的，Java内存模型和硬件内存模型是不同的。硬件内存体系并没有区分线程栈和堆空间在硬件体系中，线程栈和堆都位于主存中。线程栈和堆的部分有时候可能会出现在cpu的缓存层和cp内部的寄存器中。如图所示：



当对象和变量可以被存储在计算机中的不同内存区域时，一些问题可能会发生，两个主要的问题如下：

- 线程对共享变量更新的可见性
- 读取、检查和写入共享变量时的竞争条件

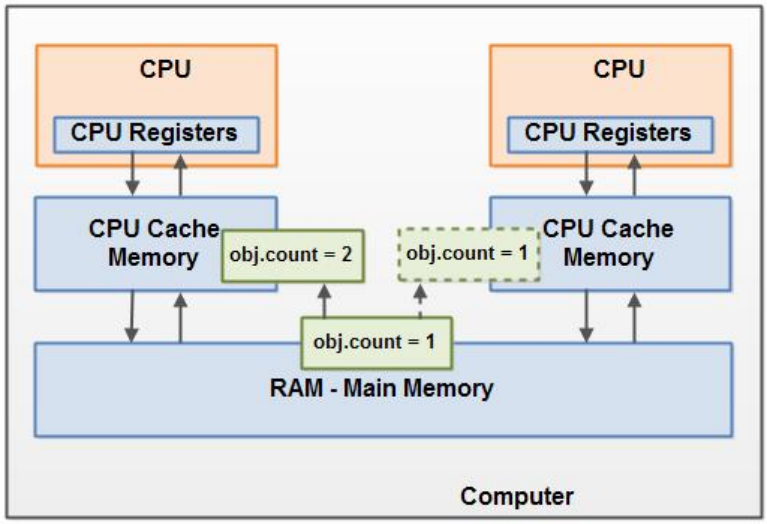
这两个问题我们将会在接下来的两节进行解释：

共享对象的可见性

如果两个或多个线程共享一个对象，在没有正确使用volatile声明或者同步机制的情况下，一个线程共享对象的更新可能对于另外一个线程是不可见的。

思考一下，当共享对象一开始是存储在主存当中。一个跑在cpu1的线程从主存将共享对象读进cpu缓层。它对共享变量做了一些修改。当cpu的缓存层还未将数据刷回主存的时候，已被修改的新共享对对于跑在其他cpu的线程是不可见的。通过这种方式，每个线程都拥有各自关于共享变量的副本，这副本位于不同的cpu缓存当中。

下图说明了大致的情况。一个跑在左边cpu的线程将共享变量拷贝它自己的cpu缓存，并更新它的coun变量为2。这个更新对于跑在右边cpu的线程来说是不可见的，因为对于count变量的更新还未从cpu存刷回主存中。



你可以使用Java中的volatile关键字去解决这个问题。volatile关键字可以确保给定的变量直接从主存读取，并且总是在更新的时候立马从cpu缓存刷回到主存中。

竞争条件

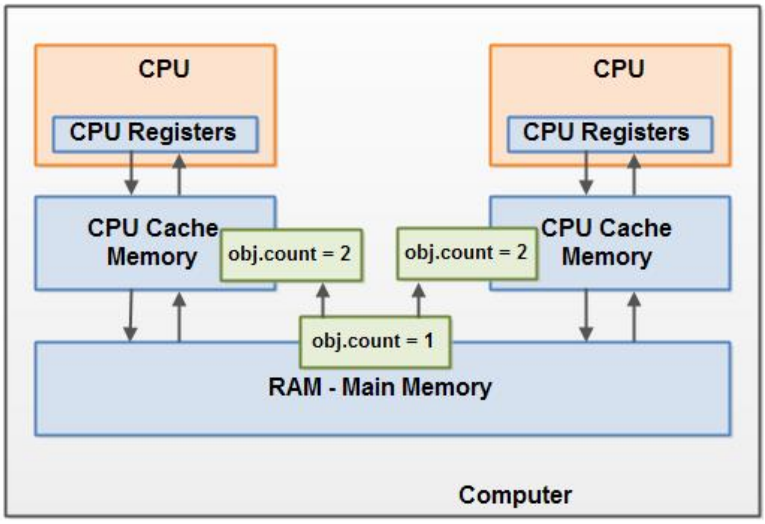
如果两个或多个线程共享同一个对象，并且多个线程同时更新这个共享对象，可能会出现竞争条件。

想象一下这样的场景，当线程A读取共享对象的count变量到cpu缓存时，同时线程B也执行了相同的作，但count被拷贝B线程自己的cpu缓存。此时线程A、线程B对count变量做了+1操作。此时变量cont在两个cpu缓存各加了一次，一共自增了两次。

如果这些自增操作被顺序执行的话，count变量将会被自增两次，并得到正确的结果从cpu缓存刷回主存中。

然而，如果两次自增被并发执行并且没有正确的同步操作时。不管是线程A还是线程B将count变量刷到主存时，主存中更新后的值将只会比原值多1，尽管做了两次自增操作。

下图说明了如上所述竞争条件导致的问题：



你可以使用Java的synchronized同步块去解决这个问题。一个同步块确保在任何时间内只有一个线程以进入给定代码的临界区。同步块还保证在同步块中访问的所有变量都将从主存中读入，并且当线程出同步代码块时，所有被更新的变量都会被刷回到主存中，不管该变量有没有被声明为volatile。

原文链接：[译] 解析 Java 内存模型