Java内存模型

原文地址 作者：Jakob Jenkov 译者：张坤

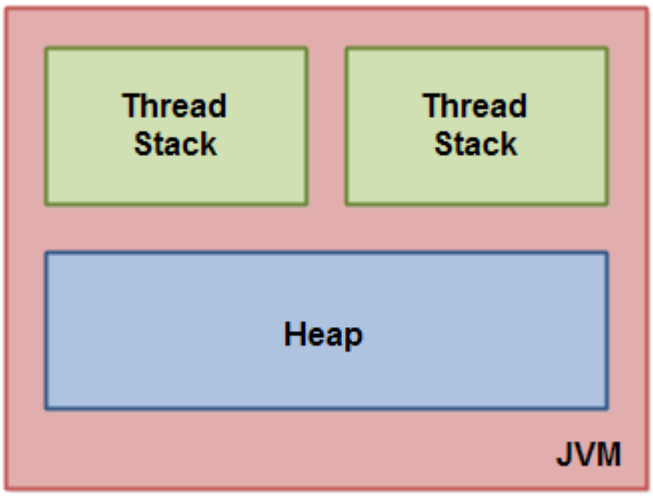
Java内存模型规范了Java虚拟机与计算机内存是如何协同工作的。Java虚拟机是一个完整的计算机的一个模型，因此这个模型自然也包含一个内存模型——又称为Java内存模型。

如果你想设计表现良好的并发程序，理解Java内存模型是非常重要的。Java内存模型规定了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。

原始的Java内存模型存在一些不足，因此Java内存模型在Java1.5时被重新修订。这个版本的Java内存模型在Java8中人在使用。

Java内存模型内部原理

Java内存模型把Java虚拟机内部划分为线程栈和堆。这张图演示了Java内存模型的逻辑视图。

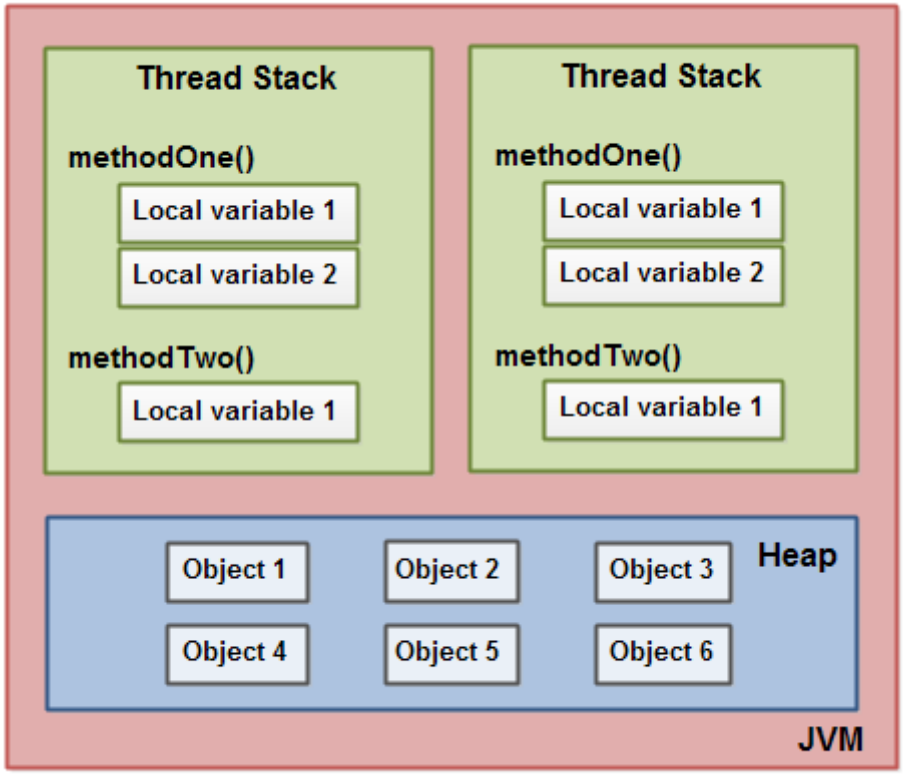


每一个运行在Java虚拟机里的线程都拥有自己的线程栈。这个线程栈包含了这个线程调用的方法当前执行点相关的信息。一个线程仅能访问自己的线程栈。一个线程创建的本地变量对其它线程不可见，仅自己可见。即使两个线程执行同样的代码，这两个线程任然在在自己的线程栈中的代码来创建本地变量。因此，每个线程拥有每个本地变量的独有版本。

所有原始类型的本地变量都存放在线程栈上，因此对其它线程不可见。一个线程可能向另一个线程传递一个原始类型变量的拷贝，但是它不能共享这个原始类型变量自身。

堆上包含在Java程序中创建的所有对象，无论是哪一个对象创建的。这包括原始类型的对象版本。如果一个对象被创建然后赋值给一个局部变量，或者用来作为另一个对象的成员变量，这个对象任然是存放在堆上。

下面这张图演示了调用栈和本地变量存放在线程栈上，对象存放在堆上。



一个本地变量可能是原始类型，在这种情况下，它总是“呆在”线程栈上。

一个本地变量也可能是指向一个对象的一个引用。在这种情况下，引用（这个本地变量）存放在线程栈上，但是对象本身存放在堆上。

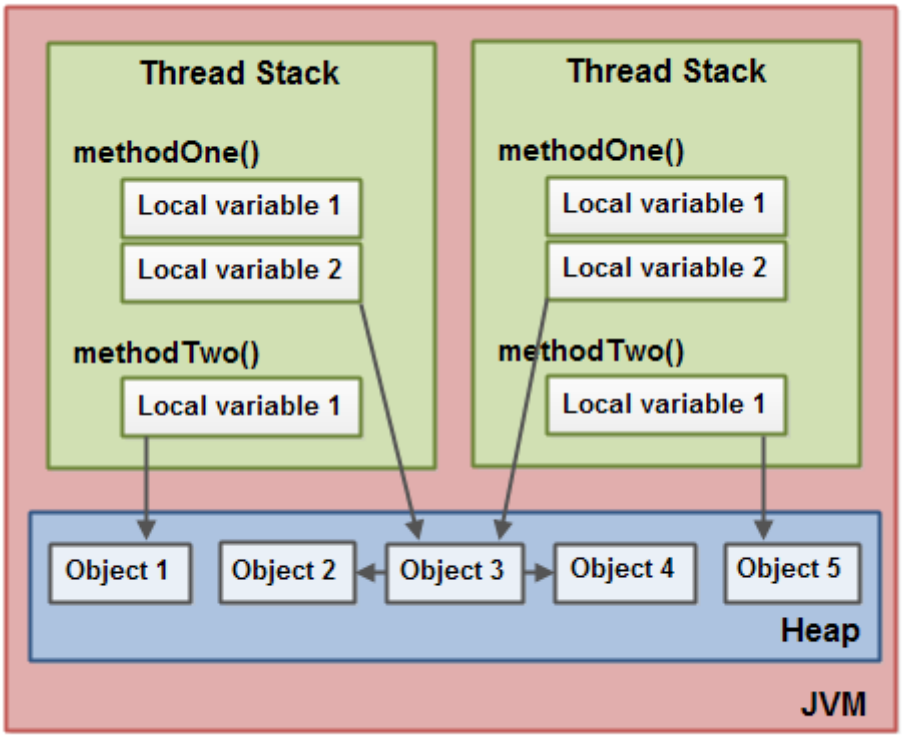
一个对象可能包含方法，这些方法可能包含本地变量。这些本地变量任然存放在线程栈上，即使这些方法所属的对象存放在堆上。

一个对象的成员变量可能随着这个对象自身存放在堆上。不管这个成员变量是原始类型还是引用类型。

静态成员变量跟随着类定义一起也存放在堆上。

存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。

下图演示了上面提到的点：



两个线程拥有一些列的本地变量。其中一个本地变量（Local Variable 2）执行堆上的一个共享对象（Object 3）。这两个线程分别拥有同一个对象的不同引用。这些引用都是本地变量，因此存放在各自线程的线程栈上。这两个不同的引用指向堆上同一个对象。

注意，这个共享对象（Object 3）持有Object2和Object4一个引用作为其成员变量（如图中Object3指向Object2和Object4的箭头）。通过在Object3中这些成员变量引用，这两个线程就可以访问Object2和Object4。

这张图也展示了指向堆上两个不同对象的一个本地变量。在这种情况下，指向两个不同对象的引用不是同一个对象。理论上，两个线程都可以访问Object1和Object5，如果两个线程都拥有两个对象的引用。但是在上图中，每一个线程仅有一个引用指向两个对象其中之一。

因此，什么类型的Java代码会导致上面的内存图呢？如下所示：

```
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;
}
```

如果两个线程同时执行run()方法，就会出现上图所示的情景。run()方法调用methodOne()方法，methodOne()调用methodTwo()方法。

methodOne()声明了一个原始类型的本地变量和一个引用类型的本地变量。

每个线程执行methodOne()都会在它们对应的线程栈上创建localVariable1和localVariable2的私有拷贝。localVariable1变量彼此完全独立，仅“生活”在每个线程的线程栈上。一个线程看不到另一个线程对它的localVariable1私有拷贝做出的修改。

每个线程执行methodOne()时也将会创建它们各自的localVariable2拷贝。然而，两个localVariable2的不同拷贝都指向堆上的同一个对象。代码中通过一个静态变量设置localVariable2指向一个对象引用。仅存在一个静态变量的一份拷贝，这份拷贝存放在堆上。因此，localVariable2的两份拷贝都指向由MySharedObject指向的静态变量的同一个实例。MySharedObject实例也存放在堆上。它对应于上图中的Object3。

注意，MySharedObject类也包含两个成员变量。这些成员变量随着这个对象存放在堆上。这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Object2和Object4.

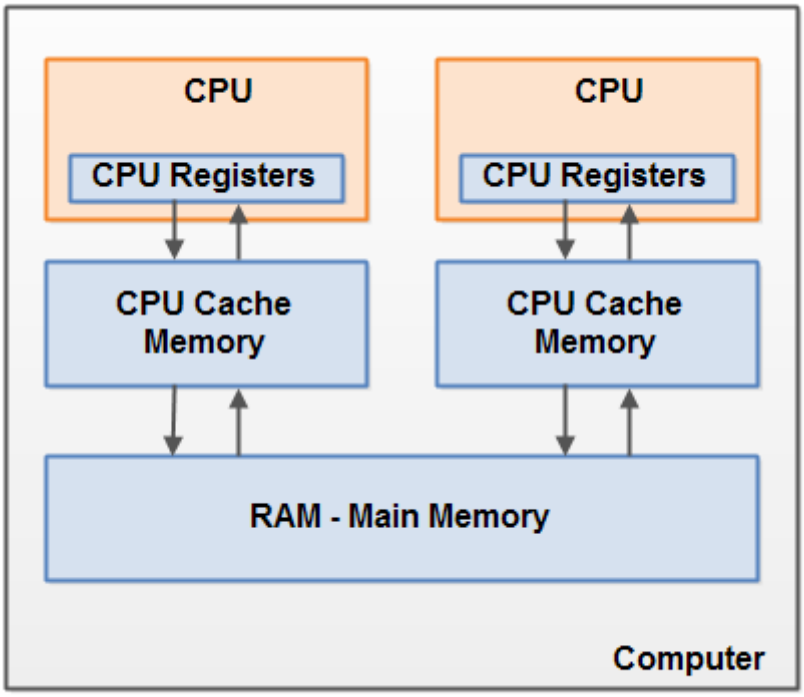
注意，methodTwo()创建一个名为localVariable的本地变量。这个成员变量是一个指向一个Integer对象的对象引用。这个方法设置localVariable1引用指向一个新的Integer实例。在执行methodTwo方法时，localVariable1引用将会在每个线程中存放一份拷贝。这两个Integer对象实例化将会被存储堆上，但是每次执行这个方法时，这个方法都会创建一个新的Integer对象，两个线程执行这个方法将会创建两个不同的Integer实例。methodTwo方法创建的Integer对象对应于上图中的Object1和Object5。

还有一点，MySharedObject类中的两个long类型的成员变量是原始类型的。因为，这些变量是成员变量，所以它们任然随着该对象存放在堆上，仅有本地变量存放在线程栈上。

硬件内存架构

现代硬件内存模型与Java内存模型有一些不同。理解内存模型架构以及Java内存模型如何与它协同工作也是非常重要的。这部分描述了通用的硬件内存架构，下面的部分将会描述Java内存是如何与它“联手”工作的。

下面是现代计算机硬件架构的简单图示：



一个现代计算机通常由两个或者多个CPU。其中一些CPU还有多核。从这一点可以看出，在一个有两个或者多个CPU的现代计算机上同时运行多个线程是可能的。每个CPU在某一时刻运行一个线程是没有问题的。这意味着，如果你的Java程序是多线程的，在你的Java程序中每个CPU上一个线程可能同时（并发）执行。

每个CPU都包含一系列的寄存器，它们是CPU内内存的基础。CPU在寄存器上执行操作的速度远大于在主存上执行的速度。这是因为CPU访问寄存器的速度远大于主存。

每个CPU可能还有一个CPU缓存层。实际上，绝大多数的现代CPU都有一定大小的缓存层。CPU访问缓存层的速度快于访问主存的速度，但通常比访问内部寄存器的速度还要慢一点。一些CPU还有多层缓存，但这些对理解Java内存模型如何和内存交互不是那么重要。只要知道CPU中可以有一个缓存层就可以了。

一个计算机还包含一个主存。所有的CPU都可以访问主存。主存通常比CPU中的缓存大得多。

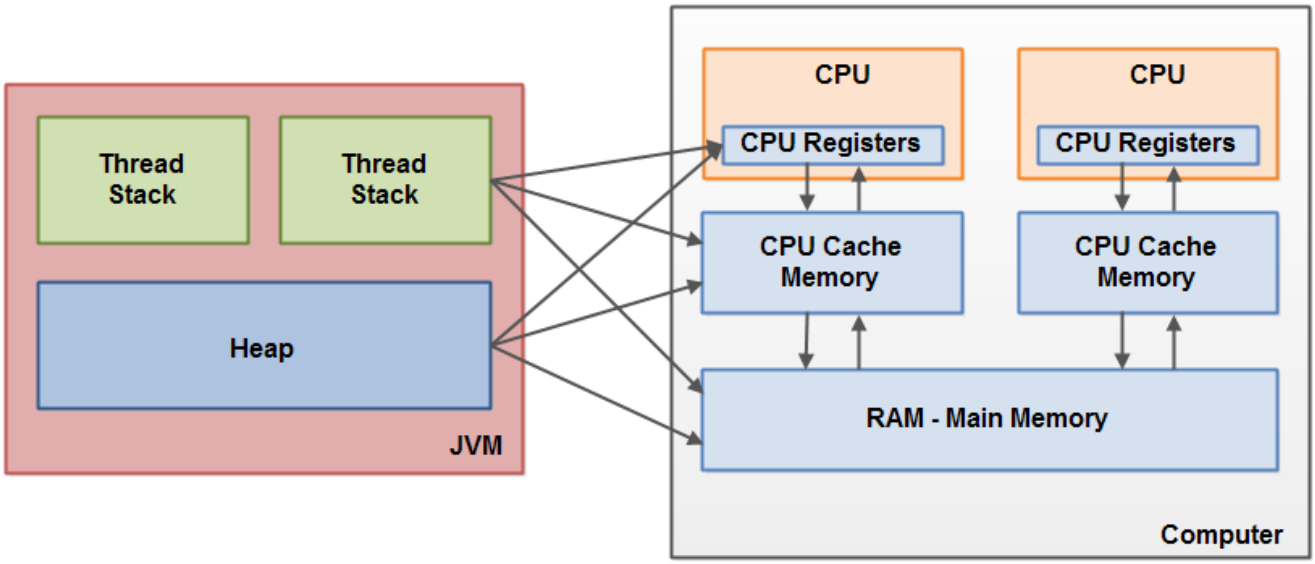
通常情况下，当一个CPU需要读取主存时，它会将主存的部分读到CPU缓存中。它甚至可能将缓存中的部分内容读到它的内部寄存器中，然后在寄存器中执行操作。当CPU需要将结果写回到主存中去时，它会将内部寄存器的值刷新到缓存中，然后在某个时间点将值刷新回主存。

当CPU需要在缓存层存放一些东西的时候，存放在缓存中的内容通常会被刷新回主存。CPU缓存可以在某一时刻将数据局部写到它的内存中，和在某一时刻局部刷新它的内存。它不会再某一时刻读/写整个缓存。

通常，在一个被称作“cache lines”的更小的内存块中缓存被更新。一个或者多个缓存行可能被读到缓存，一个或者多个缓存行可能再被刷新回主存。

Java内存模型和硬件内存架构之间的桥接

上面已经提到，Java内存模型与硬件内存架构之间存在差异。硬件内存架构没有区分线程栈和堆。对于硬件，所有的线程栈和堆都分布在主内中。部分线程栈和堆可能有时候会出现在CPU缓存中和CPU内部的寄存器中。如下图所示：



当对象和变量被存放在计算机中各种不同的内存区域中时，就可能会出现一些具体的问题。主要包括如下两个方面：

-线程对共享变量修改的可见性

-当读，写和检查共享变量时出现race conditions

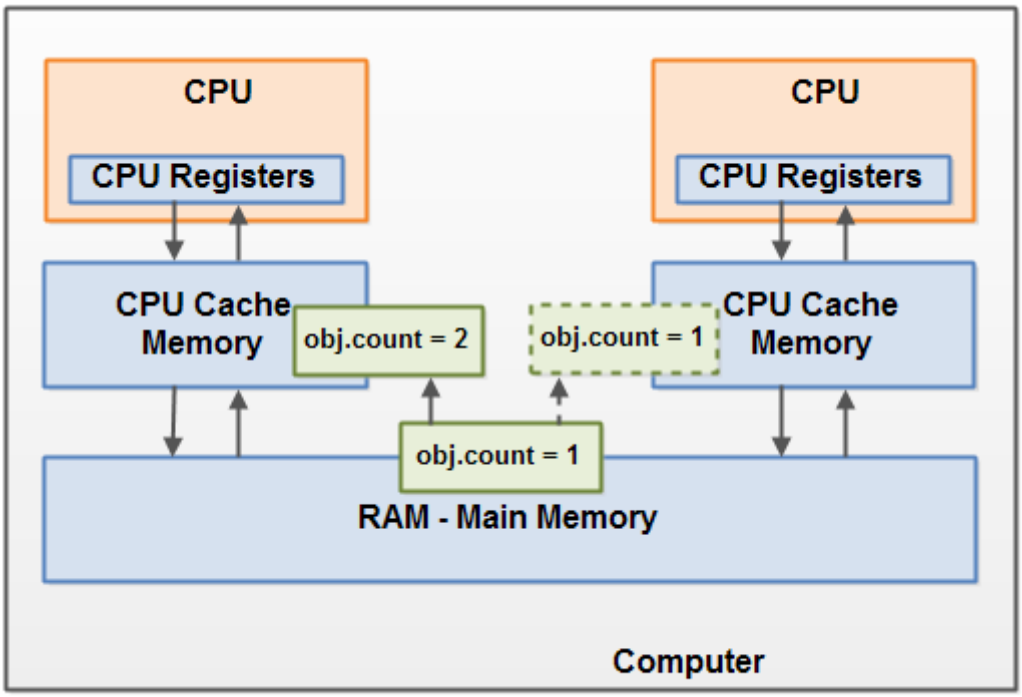
下面我们专门来解释以下这两个问题。

共享对象可见性

如果两个或者更多的线程在没有正确的使用`volatile`声明或者同步的情况下共享一个对象，一个线程更新这个共享对象可能对其它线程来说是不接见的。

想象一下，共享对象被初始化在主存中。跑在CPU上的一个线程将这个共享对象读到CPU缓存中。然后修改了这个对象。只要CPU缓存没有被刷新会主存，对象修改后的版本对跑在其它CPU上的线程都是不可见的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的CPU缓存中。

下图示意了这种情形。跑在左边CPU的线程拷贝这个共享对象到它的CPU缓存中，然后将count变量的值修改为2。这个修改对跑在右边CPU上的其它线程是不可见的，因为修改后的count的值还没有被刷新回主存中去。



解决这个问题你可以使用Java中的volatile关键字。volatile关键字可以保证直接从主存中读取一个变量，如果这个变量被修改后，总是会被写回到主存中去。

Race Conditions

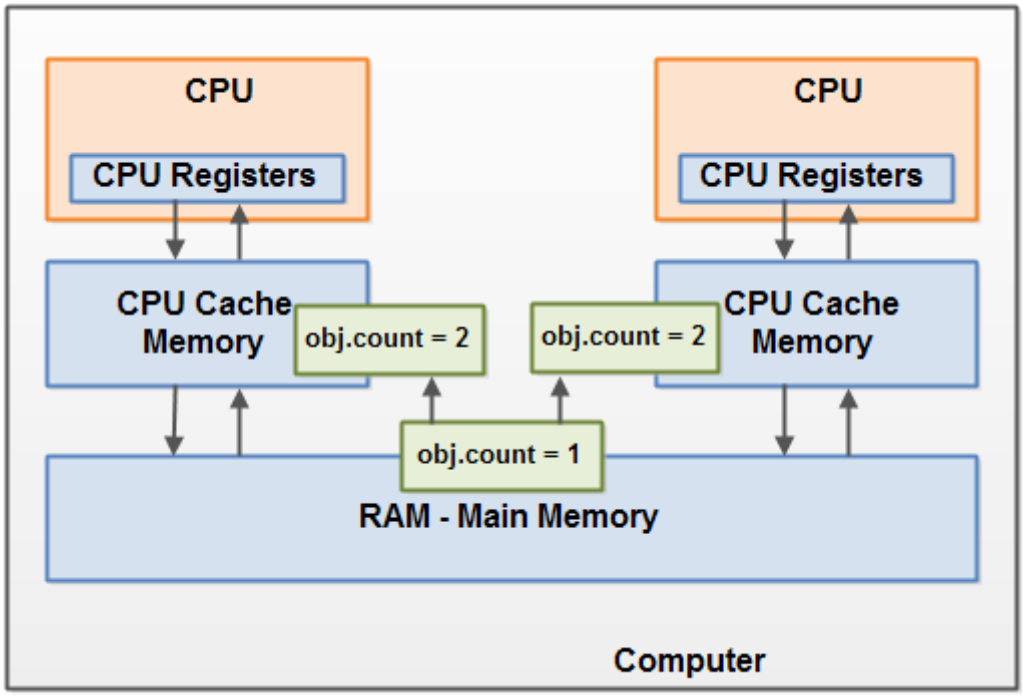
如果两个或者更多的线程共享一个对象，多个线程在这个共享对象上更新变量，就有可能发生race conditions。

想象一下，如果线程A读一个共享对象的变量count到它的CPU缓存中。再想象一下，线程B也做了同样的事情，但是往一个不同的CPU缓存中。现在线程A将count加1，线程B也做了同样的事情。现在count已经被增在了两个，每个CPU缓存中一次。

如果这些增加操作被顺序的执行，变量count应该被增加两次，然后原值+2被写回到主存中去。

然而，两次增加都是在没有适当的同步下并发执行的。无论是线程A还是线程B将count修改后的版本写回到主存中取，修改后的值仅会被原值大1，尽管增加了两次。

下图演示了上面描述的情况：



解决这个问题可以使用Java同步块。一个同步块可以保证在同一时刻仅有一个线程可以进入代码的临界区。同步块还可以保证代码块中所有被访问的变量将会从主存中读入，当线程退出同步代码块时，所有被更新的变量都会被刷新回主存中去，不管这个变量是否被声明为volatile。

木秀林
2015/06/12 1:42下午

并发网总是翻译些不痛不痒的内容，这样下去就把并发网的逼格减低了。

jvm内存模型这样的内容不知道炒了多少次，能不能统一起来？

否则今天这个人挖个坑，明天那个人挖个坑，最终谁也不能真正的挖到一口井，最终大家都没有喝到井水。

BlankKelly
2015/06/12 8:20下午

只是想把这个系列http://ifeve.com/java-concurrency-thread-directory/补充完整。

方 腾飞
2015/06/12 9:04下午

我觉得没有问题，支持你！

方 腾飞
2015/06/12 9:03下午

后续会和JUG合作翻译Java官方文章，另外还计划翻译各技术官网的技术手册

> 木秀林
> 2015/06/15 9:34上午
>
> “翻译各技术官网的技术手册”这个计划不错，表示大力支持。最近在研究freemarker，发现有个金丝燕网，那里的freemarker研究可以说是网络上最全面的，据说后期还会推出源码分析，可以考虑一下和金丝燕的合作。

liuxinglanyue
2015/06/17 3:24下午

“在你的Java程序中每个CPU上一个线程可能同时（并发）执行”

这里是不是说成 并行执行 好些。

鱼丸
2015/08/10 5:05下午

“理论上，两个线程都可以访问Object1和Object5，如果两个线程都拥有两个对象的引用”这句话翻译语序是不是可以改改

dzt
2015/10/15 9:29上午

volatile讲得简单了，能再细点就好了!

BlackLee
2015/10/16 4:56下午

volatile 只能保证每次读取都是从主存中取的，并不能保证是最新的。每个线程从主存取值，修改，写回不是原子性的。

T1取值，T1加一，T2取值，T1写回，T2加一，T2写回。最后还是只加一。

要解决同步问题还是得上锁，或者同步代码。

单个的volatile并不能保证同步。

> javajack
> 2016/02/16 5:51下午

用volatile的话，一般情况下只有一个线程进行写操作。

javajack
2016/02/16 5:52下午

看了之后有些许受益，能像一楼所说的那样，就更完美了~

Tod
2016/04/24 5:49下午

谢谢分享。

有几个问题：

1、我提供的RPC服务中，一个方法里都会new一个对象，这个对象就是放在线程栈里的吧？

2、如果RPC框架会为每一个调用创建一个线程来执行请求，那么这个类型的对象在是线程隔离的，每一次调用就会new一个对象。如果RPC框架用的是线程池，对象肯定还是线程隔离，但是还会每一次调用都new一个对象吗？

3、如果在声明一个ThreadLocal，将这个对象set到ThreadLocal里，会减少性能开销吗？

走召礻 申了
2016/06/01 8:46下午

Race Conditions是不是也可以利用volatile解决？

listenchina
2016/06/30 9:11上午

当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。

————————————

每一个线程都拥有这个本地变量的私有拷贝？“这个本地变量”是指哪个？对象的成员变量？

listenchina
2016/06/30 9:29上午

> listenchina :
>
> 当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。
>
> ————————————
>
> 每一个线程都拥有这个本地变量的私有拷贝？“这个本地变量”是指哪个？对象的成员变量？

哦，“这个本地变量”说的是这个对象的成员变量的引用？

zhijian
2016/07/02 11:22下午

应该是“如果两个线程同时调用同一个对象上的同一个方法”中调用方法的本地变量。

程式猎人
2016/08/13 10:18下午

写的还可以，挺清晰的，非常感谢

Jintao_Ma
2016/11/15 3:20下午

“存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。” 这段话最后一句不对，每个线程都会修改这个对象的成员变量，而不是拥有一份拷贝，就像前文说的，对象的成员变量也是存放在堆中的。

Jintao_Ma
2016/11/15 3:32下午

> Jintao_Ma :
>
> “存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的

> 成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。” 这段话最后一句不对，每个线程都会修改这个对象的成员变量，而不是拥有一份拷贝，就像前文说的，对象的成员变量也是存放在堆中的。
>
> ”

或者改为”但是每一个线程都拥有指向这个对象的引用”

cenyol
2017/02/26 3:41下午

感觉“原始类型”翻译为“基本类型”会更好

sttddy
2017/03/08 2:46下午

“这个版本的Java内存模型在Java8中人在使用。” 人在使用 应为 仍在使用；

“这个线程栈包含了这个线程调用的方法当前执行点相关的信息”,应翻译为”线程栈包含有关线程调用以到达当前执行点的方法的信息”

sttddy
2017/03/08 2:48下午

原文中有大量的“A thread”不应该翻译成“一个线程”，直接翻译成“线程”更好些

zhanwangls
2018/02/25 11:38下午

方法区为什么没有提

Angzk3488
2019/05/28 3:50下午

这两个线程任然在在自己的线程栈中的代码来创建本地变量。

仍然 任然。

erlking
2020/12/04 4:03下午

原始的Java内存模型存在一些不足，因此Java内存模型在Java1.5时被重新修订。这个版本的Java内存模型在Java8中人在使用。

这句话是应该是有个错别字吧 ， “Java8中人在使用”，“Java8中仍在使用”