JVM内存模型详解

可见性问题

1. 首先我们先来看一个多线程的可见性问题，如下代码在jvm以server的模式运行的情况下会进行jit及时编译优化，导致指令重排序，造成了线程thread1无法看到主线程更新标志位而导致的死循环。

```
// 1、 jre/bin/server  放置hsdis动态链接库
//  测试代码 将运行模式设置为-server， 变成死循环  。 没加默认就是client模式，就是正常（可见性问题）
// 2、 通过设置JVM的参数，打印出jit编译的内容 （这里说的编译非class文件），通过可视化工具jitwatch进行查看
// -server -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:+LogCompilation -XX:LogFile=jit.log
//  关闭jit优化-Djava.compiler=NONE
public class VisibilityDemo {
    private boolean flag = true;

    public static void main(String[] args) throws InterruptedException {
        VisibilityDemo demo1 = new VisibilityDemo();
        Thread.sleep(20000L);
        System.out.println("代码开始了");
        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                // class ->  运行时jit编译  -> 汇编指令 -> 重排序
                while (demo1.flag) { // 指令重排序
                    i++;
                }
                System.out.println(i);
            }
        });
        thread1.start();

        TimeUnit.SECONDS.sleep(2);
        // 设置is为false，使上面的线程结束while循环
        demo1.flag = false;
        System.out.println("被置为false了.");
    }
}
```

2. CPU等其他缓存，导致可见性(短时间)。
3. 重排序可能导致可见性问题。

```
Thread thread1 = new Thread(new Runnable() {
    public void run() {
        int i = 0;
        while (demo1.flag) {
            i++;
        }
        System.out.println(i);
    }
});
```

Thread1 只有读
JIT优化器 -- 循环读 --> 读一次

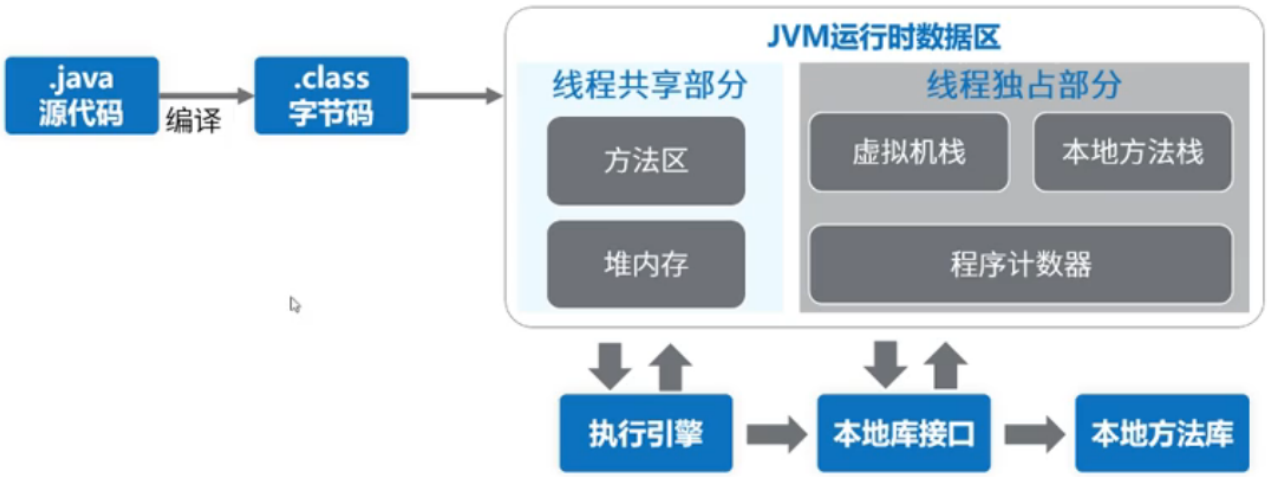
优化后代码

```
boolean f = demo1.flag;
if(f){
  while(true) {
    i++;
  }
}
```

多线程中的问题

1. 所见非所得
2. 无法用肉眼去检测程序的准确性
3. 不同的运行平台有不同的表现
4. 错误很难重现

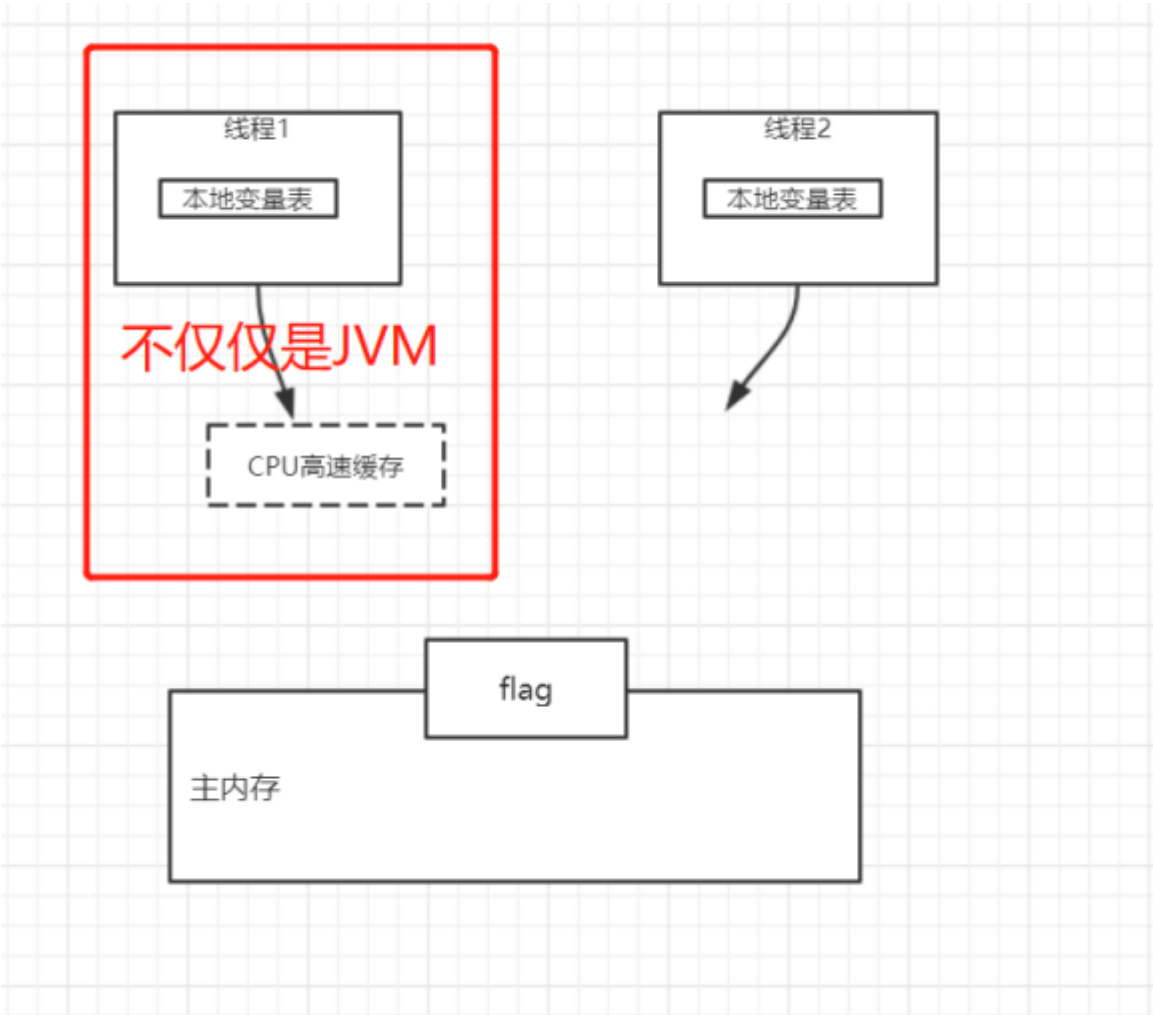
JVM运行时数据区



线程独占：每个线程都会有它独立的空间，随线程生命周期而创建和销毁
线程共享：所有线程能访问这块内存数据，随虚拟机或者GC而创建和销毁

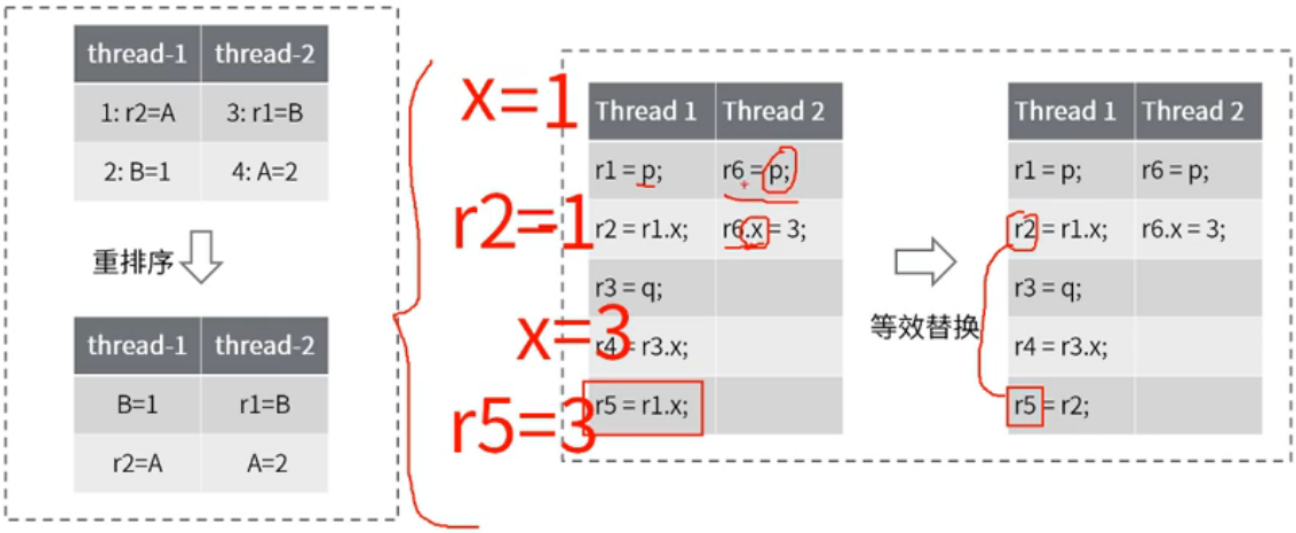
从内存结构到内存模型

因为这种内存结构，在多线程下数据交互会有各种情况出现



指令重排序

Java编程语言的语义允许编译器和处理器执行优化，这些优化可以与不正确的同步代码交互，从而产生看似矛盾的问题。



内存模型的含义

1. 内存模型描述程序的可能行为。
2. Java编程语言内存模型通过检查执行跟踪中的每个读操作，并根据某些规则检查该读操作观察到的写操作是否有效来工作。
3. 只要程序的所有执行产生的结果都可以由内存模型预测。具体的实现者任意实现，包括操作的重新排序和删除不必要的同步。
4. **内存模型决定了在程序的每个点上可以读取什么值。**

Shared Variables共享变量描述

1. 可以在线程之间共享的内存成为共享内存或堆内存。
2. 所有实例字段，静态字段和数组元素都存储在堆内存中。
3. 如果至少有一个访问是写的，那么对同一个变量的两次访问(读或写)是冲突的。

线程操作的定义

1. write要写的变量以及要写的值。
2. read要读的变量以及可见的写入值。
3. lock要锁定的管程(监视器monitor)。
4. unlock要解锁的管程。
5. 外部操作(socket等等)。
6. 启动和终止。

   程序顺序：如果一个程序没有数据竞争，那么程序的所有执行看起来都是顺序一致的。

对于同步的规则定义

1. 对于监视器m的解锁与所有后续操作对于m的加锁同步。

2. 对volatile变量v的写入，与所有其他线程后续对v的读同步。
3. 启动线程的操作与线程中的第一个操作同步。
4. 对于每个属性写入默认值与每个线程对其进行的操作同步。
5. 线程T1的最后操作与线程T1已将结束同步。(isAlive，join可以判断线程是否终结)
6. 线程T1中断了T2，那么线程T1的中断操作与其他所有线程发现T2被中断了同步。通过抛出InterruptedException异常，或者调用Thread.interrupted或Thread.isInterrupted。

Happens-before先行发生原则

happens-before关系主要用于强调两个有冲突的动作之间的顺序，以及定义数据争用发生的时机。

具体由虚拟机实现，有必要确保一下原则的成立。

1. 某个线程中的每个动作都happens-before该线程中该动作之后的动作。
2. 某个管程上的unlock动作happens-before同一个管程上后续的lock动作。
3. 对某个volatile字段的写操作happens-before每个后续对该字段的读操作。
4. 在某个线程对象上调用start()方法happens-before该启动了的线程中的任意动作。
5. 某个线程中的所有动作happens-before任意其它线程成功从该线程对象上的join()中返回。
6. 如果某个动作a happens-before动作b，且b happens-before动作c，则有a happens-before c。

volatile关键字

可见性问题：让一个线程对共享变量的修改，能够及时的被其他线程看到。

根据JMM中规定的happen before和同步原则：

1. 对某个volatile字段的写操作happens-before每个后续对该volatile字段的读操作。
2. 对volatile变量v的写入，与所有其他线程后续对v的读同步。

要满足这些条件，所volatile关键字就有这些功能：

1. 禁止缓存，volatile变量的访问控制符会加ACC_VOLATILE
2. 对volatile变量的相关指令不做重排序。

fianl在JMM中的处理

1. final在该对象的构造函数中设置对象的字段，当线程看到该对象时，将始终看到该对象的final字段的正确构造版本。f = new FinalDemo();读取到的f.x一定最新，x为final字段。
2. 如果在构造函数中设置字段后发生读取，则会看到该final字段分配的值，否则它将看到默认值。public FinalDemo(){x=1;y=x;};y会等于1。
3. 读取该共享对象的final成员变量之前，先读取共享对象。r = new Reference(); k = r.f;这两个操作不能重排序。
4. 通常static final时不可以修改的字段。然而System.in，System.out和System.err是static final字段，遗留原因，必须允许通过set方法改变，我们将这些字段称为写保护，以区别于普通final字段。

Word Tearing字节处理

1. 一个字段或元素的更新不得与其他任何字段或元素的读取或更新交互。特别是分别更新字节数组的相邻元素的两个线程不得干涉或交互，也不需要同步以确保顺序一致性。
2. 有些处理器(尤其是早期的Alphas处理器)没有提供写单个字节的功能。在这样的处理器上更新byte数组，若只是简单的读取整个内容，更新对应的字节，然后将真个内容再写回内存，将是不合法的。
3. 这个问题被称为"字分裂"，在单独更新某个字节有难度的处理器上，就需要寻求其他方式了。

double和long的特殊处理

1. 虚拟机规范中，写64位的double和long分成了两次32位值得操作。由于不是原子操作，可能导致读取到某次写操作64位的前32位，以及另一次操作的后32位。
2. 读写volatile的long和double总是原子的。读写引用也是原子的。商业JVM不会存在这个问题，虽然规范没要求实现原子性，但考虑到时间应用，大部分都实现了原子性。

参考资料

https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html

https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html

https://docs.oracle.com