Java多线程可见性

在现代操作系统上编写并发程序时，除了要注意线程安全性(多个线程互斥访问临界资源)以外，还要注意多线程对共享变量的可见性，而后者往往容易被人忽略。

在单线程环境中，如果在程序前面修改了某个变量的值，后面的程序一定会读取到那个变量的新值。这看起来很自然，然而当变量的写操作和读操作在不同的线程中时，情况却并非如此。

```
/**
 *《Java并发编程实战》27页程序清单3-1
 */
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while(!ready) {
                Thread.yield();
            }
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start(); //启动一个线程
        number = 42;
        ready = true;
    }
}
```

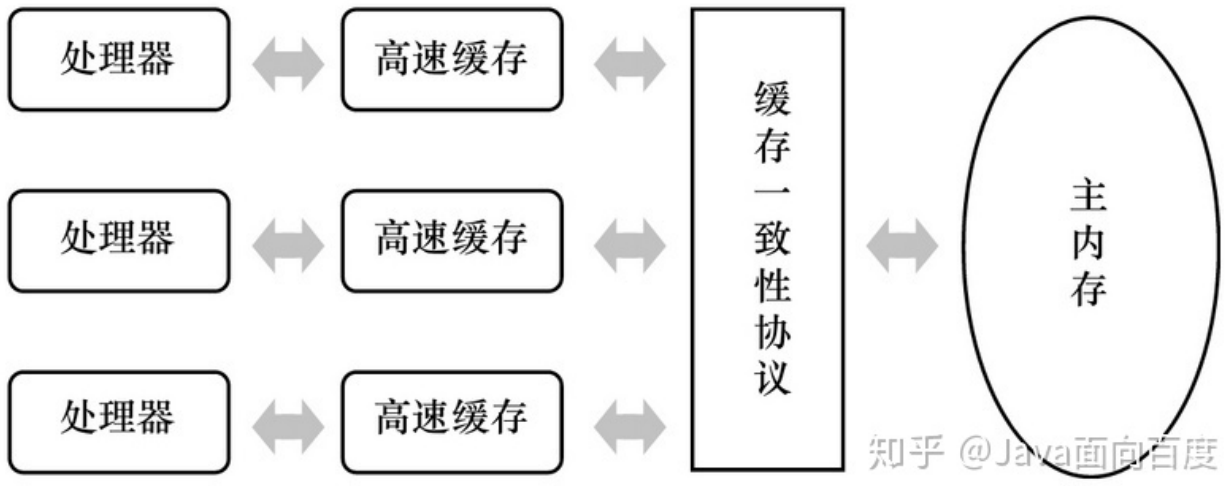
上面的代码中，主线程和读线程都访问共享变量ready和number。程序看起来会输出42，但事实上很可能会输出0，或者根本无法终止。这是因为上面的程序缺少线程间变量可见性的保证，所以在主线程中写入的变量值，可能无法被读线程感知到。

可见性：一个线程对共享变量值的修改，能够及时地被其他线程看到。
共享变量：如果一个变量在多个线程的工作内存中都存在副本，那么这个变量就是几个线程的共享变量。

为什么会出现线程可见性问题

要想解释为什么会出现线程可见性问题，需要从计算机处理器结构谈起。我们都知道计算机运算任务需要CPU和内存相互配合共同完成，其中CPU负责逻辑计算，内存负责数据存储。CPU要与内存进行交互，如读取运算数据、存储运算结果等。由于内存和CPU的计算速度有几个数量级的差距，为了提高CPU的利用率，现代处理器结构都加入了一层读写速度尽可能接近CPU运算速度的高速缓存来作为内存与CPU之间的缓冲：将运算需要使用的数据复制到缓存中，让CPU运算可以快速进行，计算结束后再将计算结果从缓存同步到主内存中，这样处理器就无须等待缓慢的内存读写了。

高速缓存的引入解决了CPU和内存之间速度的矛盾，但是在多CPU系统中也带来了新的问题：缓存一致性。在多CPU系统中，每个CPU都有自己的高速缓存，所有的CPU又共享同一个主内存。如果多个CPU的运算任务都涉及到主内存中同一个变量时，那同步回主内存时以哪个CPU的缓存数据为准呢？这就需要各个CPU在数据读写时都遵循同一个协议进行操作。



参考上图，假设有两个线程A、B分别在两个不同的CPU上运行，它们共享同一个变量X。如果线程A对X进行修改后，并没有将X更新后的结果同步到主内存，则变量X的修改对B线程是不可见的。所以CPU与内存之间的高速缓存就是导致线程可见性问题的一个原因。

重排序

CPU和主内存之间的高速缓存还会导致另一个问题——**重排序**。假设A、B两个线程共享两个变量X、Y，A和B分别在不同的CPU上运行。在A中先更改变量X的值，然后再更改变量Y的值。这时有可能发生Y的值被同步回主内存，而X的值没有同步回主内存的情况，此时对于B线程来说是无法感知到X变量被修改的，或者可以认为对于B线程来说，Y变量的修改被重排序到了X变量修改的前面。上面的程序NoVisibility类中有可能输出0就是这种情况，虽然在主线程中是先修改number变量，再修改ready变量，但对于读线程来说，ready变量的修改有可能被重排序到number变量修改之前。

重排序:代码书写的顺序与实际执行的顺序不同，指令重排序时编译器或处理器为了提高程序性能做的优化。

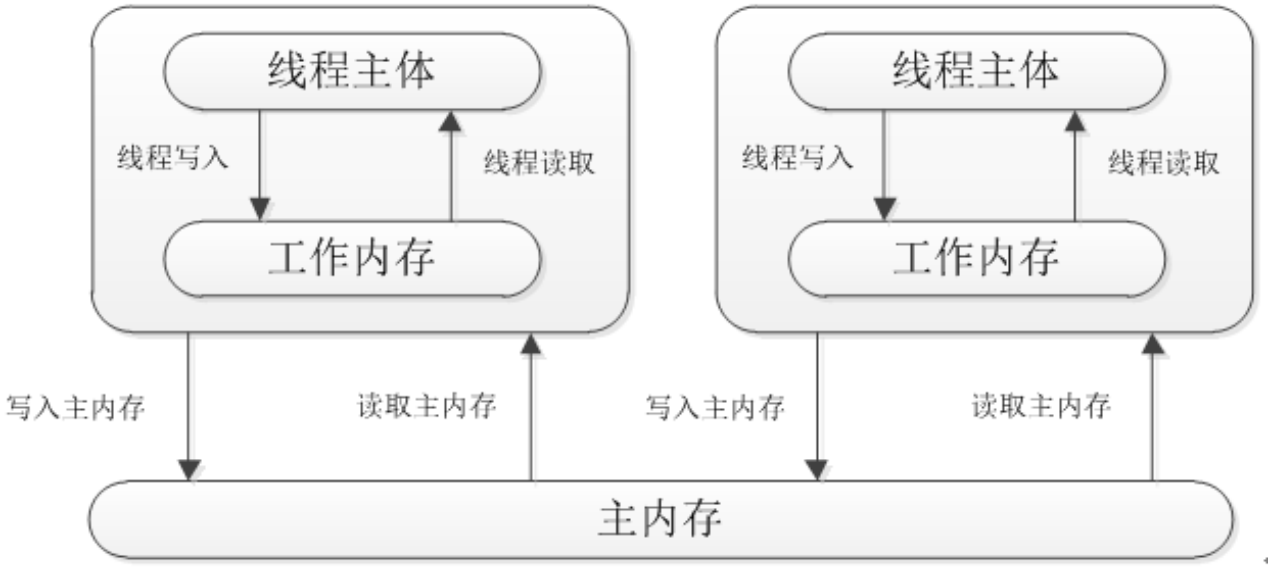
1. 编译器优化的重排序（编译器优化）
2. 指令集并行重排序（处理器优化）
3. 内存系统的重排序（处理器优化）

此外，为了提高程序的执行效率，编译器在生成指令序列时和CPU执行指令序列时，都有可能对指令进行重排序。Java语言规范要求JVM只在单个线程内部维护一种**类似**串行的语义，即只要程序的最终结果与严格串行环境中执行的结果相同即可。所以在单线程环境中，我们无法察觉到重排序，因为程序重排序后的执行结果与严格按顺序执行的结果相同。就像在类NoVisibility的主线程中，先修改ready变量还是先修改number变量对于主线程自己的执行结果是没有影响的，但是如果number变量和ready变量的修改发生重排序，对读线程是有影响的。所以在编写并发程序时，我们一定要注意重排序对多线程执行结果的影响。

到这里大家一定会发现，我们所讨论的CPU高速缓存、指令重排序等内容都是计算机体系结构方面的东西，并不是Java语言所特有的。事实上，很多主流程序语言(如C/C++)都存在多线程可见性的问题，这些语言是借助物理硬件和操作系统的内存模型来处理多线程可见性问题的，因此不同平台上内存模型的差异，会影响到程序的执行结果。Java虚拟机规范定义了自己的内存模型JMM(Java Memory Model)来屏蔽掉不同硬件和操作系统的内存模型差异，以实现让Java程序在各种平台下都能达到一致的内存访问结果。所以对于Java程序员，无需了解底层硬件和操作系统内存模型的知识，只要关注Java自己的内存模型，就能够解决Java语言中的内存可见性问题了。

Java内存模型（JMM）

JMM（java memory model）JAVA 内存模型，描述线程之间如何通过内存(memory)来进行交互。具体说来，JVM中存在一个主存区（Main Memory或Java HeapMemory），对于所有线程进行共享，而每个线程又有自己的工作内存（Working Memory），工作内存中保存的是主存中某些变量的拷贝，线程对所有变量的操作并非发生在主存区，而是发生在工作内存中，而线程之间是不能直接相互访问，变量在程序中的传递，是依赖主存来完成的。具体的如下图所示：



JMM描述了java程序中各种变量的访问规则，以及在JVM中将变量（线程共享变量）存储到内存和从内存中取出变量这样的底层细节。

- 所有的变量都存储在主内存中（分配给进程的内存）
- 每个线程都有自己独立的工作内存，里面保存该线程使用到的变量的副本（主内存中该变量的一份拷贝）

主内存主要对应于java堆中对象的实例数据部分，而工作内存则对应于虚拟机栈中的部分区域；

从更底层来说，主内存就是硬件的内存，而为了获取更好的运行速度，虚拟机及硬件系统可能会让工作内存优先存储于寄存器和高速缓存

JMM有两条规定：

- 线程对共享变量的所有操作都必须在自己的工作内存中进行，不能直接从主内存中读写
- 不同线程之间无法直接访问其他线程工作内存中的变量，线程间变量值的传递需要通过主内存来完成。

共享变量可见性实现的原理

而导致共享变量在线程间不可见的主要原因：

1、线程的交叉执行；

2、重排序结合线程的交叉执行；

3、共享变量更新后的值没有在工作内存与主内存之间及时更新。

要实现共享变量的可见性，必须保证两点：

1、线程修改后的共享变量值能够及时从工作内存刷新到主内存中；

2、其他线程能够及时把共享变量的最新值从主内存更新到自己的工作内存中。

synchronized实现可见性

synchronized 能够实现：
原子性（同步）
可见性

JMM关于synchronized的两条规定:

- 线程解锁前，必须把共享变量的最新值刷新到主内存中
- 线程加锁前，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值（注意：加锁和解锁需要是同一把锁）

线程解锁前对共享变量的修改在下次加锁时对其他线程可见

线程执行互斥代码的过程:

1. 获得互斥锁
2. 清空工作内存
3. 从主内存拷贝变量的最新副本到工作内存
4. 执行代码
5. 将更改后的共享变量的值刷新到主内存
6. 释放互斥锁

代码分析:

```
public class SynchronizedDemo {
    //共享变量
    private boolean ready = false;
    private int result = 0;
    private int number = 1;
    //写操作
    public void write(){
        ready = true;                           //1.1
        number = 2;                             //1.2
    }
    //读操作
    public void read(){
        if(ready){                              //2.1
            result = number*3;                  //2.2
        }
        System.out.println("result的值为：" + result);
    }
```

```
    //内部线程类
    private class ReadWriteThread extends Thread {
        //根据构造方法中传入的flag参数，确定线程执行读操作还是写操作
        private boolean flag;
        public ReadWriteThread(boolean flag){
            this.flag = flag;
        }
        @Override

        public void run() {
            if(flag){
                //构造方法中传入true，执行写操作
                write();
            }else{
                //构造方法中传入false，执行读操作
                read();
            }
        }
    }

    public static void main(String[] args)  {
        SynchronizedDemo synDemo = new SynchronizedDemo();
        //启动线程执行写操作
        synDemo .new ReadWriteThread(true).start();
        //启动线程执行读操作
        synDemo.new ReadWriteThread(false).start();
    }
}
```

可见性分析:

```
//共享变量
    private boolean ready = false;
    private int result = 0;
    private int number = 1;
    //写操作
    public void write(){
        ready = true;                              //1.1
        number = 2;                               //1.2
    }
    //读操作
    public void read(){
        if(ready){                                //2.1
            result = number*3;                     //2.2
        }
        System.out.println("result的值为：" + result);
    }
```

可能执行顺序:

1.1--> 1.2 --> 2.1 --> 2.2 result = 6

1.1--> 2.1 --> 2.2 --> 1.2 result = 3

1.2 --> 2.1 --> 2.2 --> 1.1 result = 0

2.1--> 2.2 --> 1.1 --> 1.2 result = 0

前面说过导致共享变量在线程间不可见的原因:

1. 线程的交叉执行
2. 重排序结合线程交叉执行
3. 共享变量更新后的值没有在工作内存与主内存间及时更新

synchronized解决方案: synchronized实际是实现的一种锁的机制,在一段操作和内存进行加锁，同一时刻只有一个线程才能进入锁内，锁内无论怎么重排序执行结果都是一样的。
synchronized同时可以保证可见性

volatile实现可见性

volatile关键字: 能够保证volatile变量的可见性 不能保证volatile变量复合操作的原子性

volatile如何实现内存可见性: 深入来说:通过加入内存屏障和禁止重排序优化来实现的。

- 对volatile变量执行写操作时，会在写操作后加入一条store屏障指令
- 对volatile变量执行读操作时，会在读操作前加入一条load屏障指令

首先来看如下一个实例:

```
public class VolatileTest1 {
  public static void main(String[] args) {
    try {
      RunThread thread = new RunThread();
      thread.start();
      Thread.sleep(1000);
      thread.setRunning(false);
      System.out.println("已近给  isRunning 赋值为false了！！");

    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
 }
```

```
class RunThread extends Thread {
  private boolean isRunning = true;

  public boolean isRunning() {
    return isRunning;
  }

  public void setRunning(boolean isRunning) {
    this.isRunning = isRunning;
  }

  public void run() {
    System.out.println("进入run了！！");
    while(isRunning) {

    }
    System.out.println("线程被停止了！！");
  }
}
```

代码：“System.out.println("线程被停止了！！");”一直无法得到执行。

在启动RunThread.java线程时，变量privateboolean isRunning = true;存在于公共堆栈以及线程的私有堆栈中。JVM为了线程运行的效率，线程一直在私有堆栈中取得isRunning的值是true。而代码thread.setRunning(false);虽然被执行，更新的却是公共堆栈中的isRunning变量的值false，所以线程一直死循环状态。

这个问题就是私有堆栈（线程工作内存）中的值和公共堆栈（主内存）中的值不同步造成的。解决这样的问题就要使用volatile关键字，它只要的作用就是当前线程访问isRunning这个变量时，强制性从公共堆栈中进行取值。

```
 volatile private boolean isRunning = true;
```

线程写volatile变量的过程：

```
    1、改变线程工作内存中volatile变量副本的值

    2、将改变后的副本的值从工作内存刷新到主内存
```

线程读volatile变量的过程：

```
    1、从主内存中读取volatile变量的最新值到线程的工作内存中

    2、从工作内存中读取volatile变量的副本
```

volatile不能保证volatile变量复合操作的原子性

例如number++操作分为：

1、读取number的值

2、将number的值加1

3、写入最新的number的值

Volatile非原子性的特征实例代码VolatileDemo

```
public class VolatileDemo {

    private Lock lock = new ReentrantLock();
    private volatile int number = 0;

    public int getNumber(){
        return this.number;
    }

    public void increase(){
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        this.number++;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        final VolatileDemo volDemo = new VolatileDemo();
        for(int i = 0 ; i < 500 ; i++){
            new Thread(new Runnable() {

                @Override
                public void run() {
                    volDemo.increase();
                }
            }).start();
        }

        //如果还有子线程在运行，主线程就让出CPU资源，
        //直到所有的子线程都运行完了，主线程再继续往下执行
        while(Thread.activeCount() > 1){
```

```
                Thread.yield();
            }

            System.out.println("number : " + volDemo.getNumber());
        }

    }
```

运行结果： 492,486,492,492,495

保证number自增操作的原子性的解决方案:

- 使用synchronized关键字
- 使用ReentrantLock
- 使用AtomicInteger 后边两个都在java.util.concurrent包下。

```
try {
  this.number++;
} finally {
  lock.unlock();
}
```

volatile使用注意事项

1. 要在多线程中安全的使用volatile变量，必须同时满足：

   1、对变量的写入操作不依赖其当前值

   ```
   不满足：number++, conut = conut *5等

   满足：boolean变量，记录温度变化的变量等
   ```

   2、该变量没有包含在具有其他变量的不等式中

   ```
   不满足：不等式 low < up
   ```

synchronized和volatile比较

- volatile 不需要加锁，比synchronized更轻量级，不会阻塞线程。并且volatile只能修饰变量，而synchronized可以修饰方法，以及代码块。
- 从内存可见性角度，volatile读相当于加锁，volatile写相当于解锁

- synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性。