并发编程-volatile使用精讲

第一章 volatile关键字概览

1.1. 多线程下变量的不可见性

1.1.1 概述

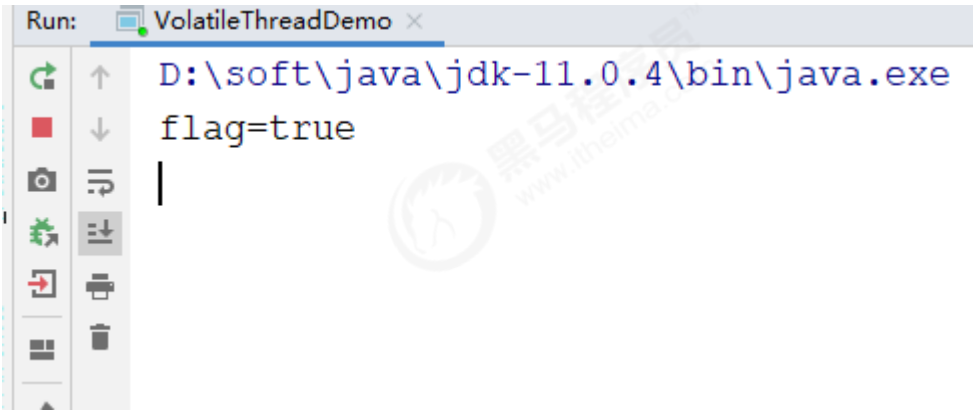
在多线程并发执行下，多个线程修改共享的成员变量，会出现一个线程修改了共享变量的值后，另一个线程不能直接看到该线程修改后的变量的最新值。

1.1.2 案例演示

```
public class MyThread extends Thread {

    // 定义成员变量
    private boolean flag = false ;
    public boolean isFlag() { return flag;}

    @Override
    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 将flag的值更改为true
        this.flag = true ;
        System.out.println("flag=" + flag);

    }
}

public class VolatileThreadDemo {// 测试类

    public static void main(String[] args) {

        // 创建MyThread线程对象
        Thread t = new MyThread() ;
        t.start();

        // main方法
        while(true) {
            if(t.isFlag()) {
                System.out.println("执行了======");
```

```
            }
        }
    }
}
```

1.1.3 执行结果



我们看到，子线程中已经将flag设置为true，但main()方法中始终没有读到修改后的最新值，从而循环没有能进入到fi语句中执行，所以没有任何打印。

1.1.4 小结

多线程下修改共享变量会出现变量修改值后的不可见性。

1.2 变量不可见性内存语义

1.2.1 概述

在介绍多线程并发修改变量不可见现象的原因之前，我们需要了解回顾一下Java内存模型（和Java并发编程有关的模型）：**JMM。**

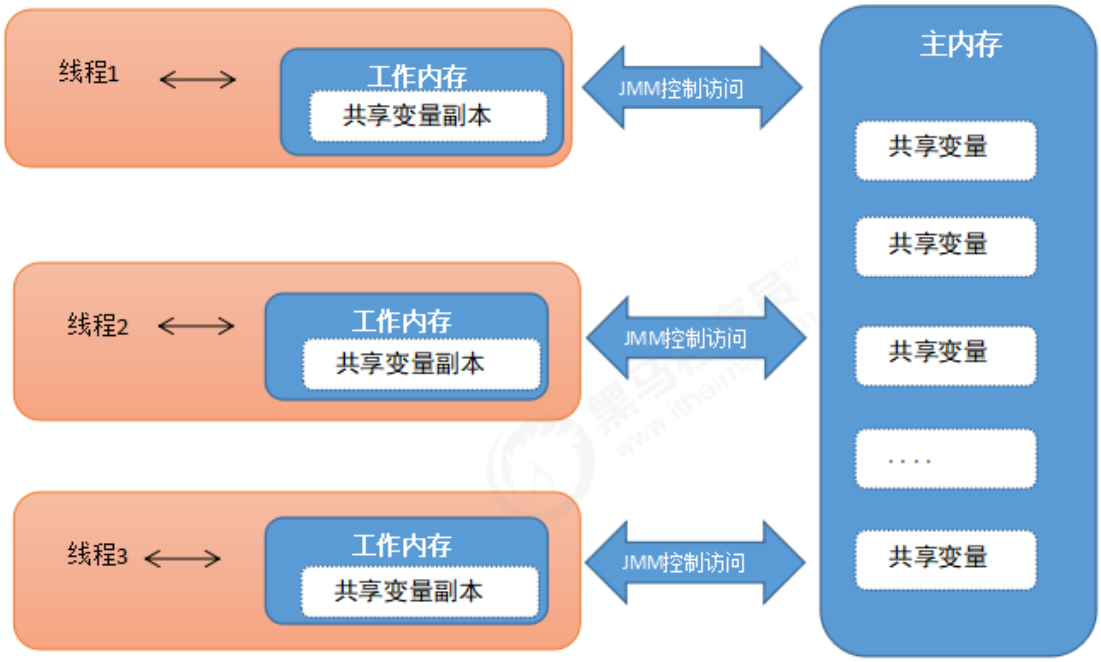
JMM(Java Memory Model)：Java内存模型,是java虚拟机规范中所定义的一种内存模型，Java内存模型是标准化的，屏蔽掉了底层不同计算机的区别。

Java内存模型(Java Memory Model)描述了Java程序中各种变量(线程共享变量)的访问规则，以及在JVM中将变量存储到内存和从内存中读取变量这样的底层细节。

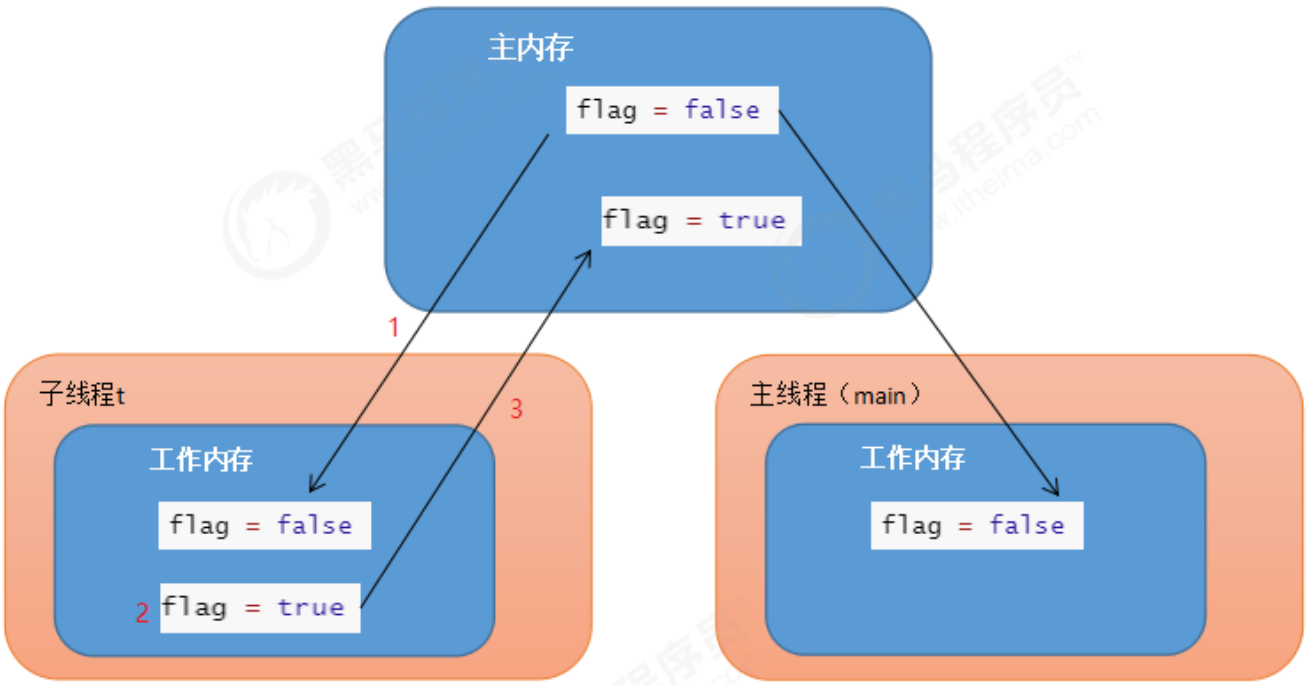
JMM有以下规定：

- 所有的共享变量都存储于主内存。这里所说的变量指的是实例变量和类变量。不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。
- 每一个线程还存在自己的工作内存，线程的工作内存，保留了被线程使用的变量的工作副本。
- 线程对变量的所有的操作(读，取)都必须在工作内存中完成，而不能直接读写主内存中的变量。
- 不同线程之间也不能直接访问对方工作内存中的变量，线程间变量的值的传递需要通过主内存中转来完成。

本地内存和主内存的关系：



1.2.2 问题分析



1. 子线程t从主内存读取到数据放入其对应的工作内存
2. 将flag的值更改为true，但是这个时候flag的值还没有写回主内存
3. 此时main方法读取到了flag的值为false
4. 当子线程t将flag的值写回去后，但是main函数里面的while(true)调用的是系统比较底层的代码，速度快，快到没有时间再去读取主存中的值，

   所以while(true)读取到的值一直是false。(如果有一个时刻main线程从主内存中读取到了主内存中flag的最新值，那么if语句就可以执行，main线程何时从主内存中读取最新的值，我们无法控制)

1.2.3 小结

可见性问题的原因

- 所有共享变量存在于主内存中，每个线程由自己的本地内存，而且线程读写共享数据也是通过本地内存交换的，所以才导致了可见性问题。

1.3. 变量不可见性解决方案

1.3.1 概述

如何实现在多线程下访问共享变量的可见性：也就是实现一个线程修改变量后，对其他线程可见呢？接下来为大家介绍两种方案：

第一种是加锁，第二种是使用**volatile**关键字

1.3.2 解决方案

加锁

```
// main方法
while(true) {
    synchronized (t) {
        if(t.isFlag()){
             System.out.println("主线程进入循环执行~~~~~");
        }
    }
}
```

某一个线程进入synchronized代码块前后，执行过程入如下：

a.线程获得锁

b.清空工作内存

c.从主内存拷贝共享变量最新的值到工作内存成为副本

d.执行代码

e.将修改后的副本的值刷新回主内存中

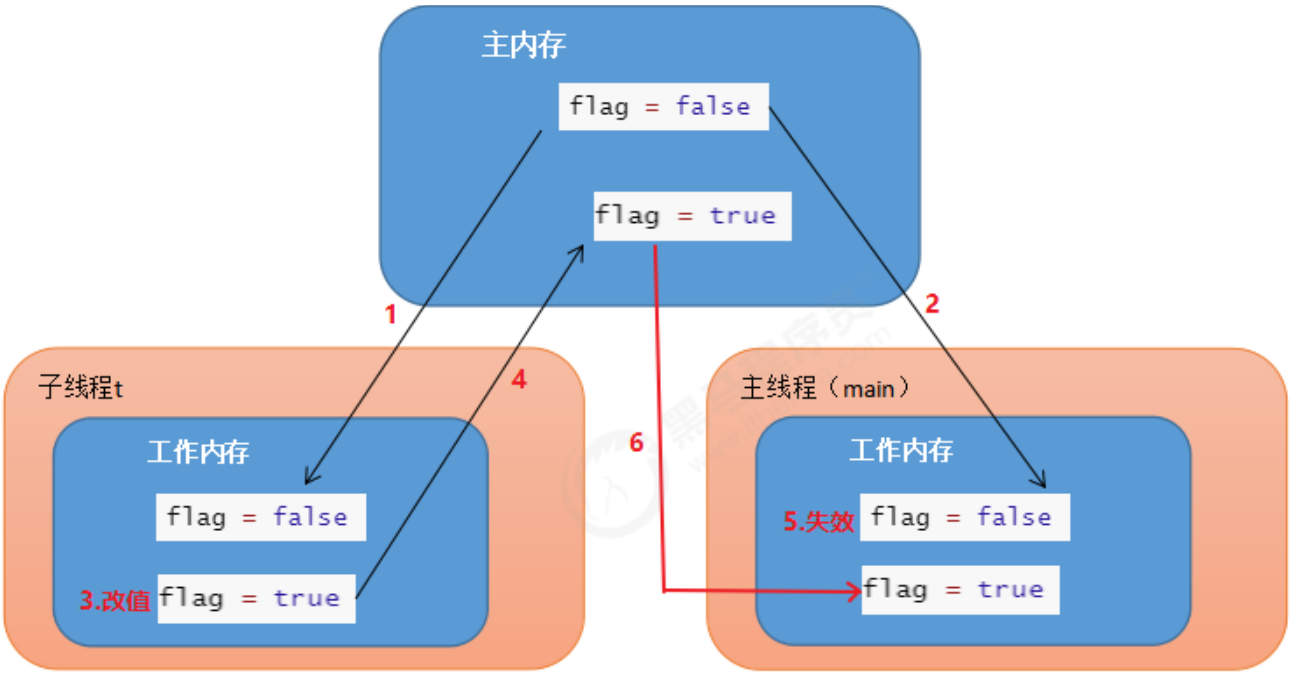
f.线程释放锁

volatile关键字修饰

使用volatile关键字修改该变量

```
private volatile boolean flag ;
```

工作原理：



1. 子线程t从主内存读取到数据放入其对应的工作内存
2. 将flag的值更改为true，但是这个时候flag的值还没有写会主内存
3. 此时main方法main方法读取到了flag的值为false
4. 当子线程t将flag的值写回去后，失效其他线程对此变量副本
5. 再次对flag进行操作的时候线程会从主内存读取最新的值，放入到工作内存中

   总结： volatile保证不同线程对共享变量操作的可见性，也就是说一个线程修改了volatile修饰的变量，当修改写回主内存时，另外一个线程立即看到最新的值。

1.3.3 小结

volatile修饰的变量可以在多线程并发修改下，实现线程间变量的可见性。

第二章 volatile的其他特性

2.1 volatile特性概述

volatile总体概览

在上节中，我们已经研究完了**volatile可以实现并发下共享变量的可见性**，除了volatile可以保证可见性外，volatile还具备如下一些突出的特性：

volatile的原子性问题：volatile不能保证原子性操作。

禁止指令重排序：volatile可以防止指令重排序操作。

2.2 volatile不保证原子性

所谓的原子性是指在一次操作或者多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行。volatile不保证原子性。

2.1.1 问题案例演示

```
public class VolatileAtomicThread implements Runnable {

    // 定义一个int类型的遍历
    private int count = 0 ;

    @Override
    public void run() {
        // 对该变量进行++操作，100次
        for(int x = 0 ; x < 100 ; x++) {
            count++ ;
            System.out.println("count ==========>>>> " + count);
        }
    }

}

public class VolatileAtomicThreadDemo {

    public static void main(String[] args) {

        // 创建VolatileAtomicThread对象
        VolatileAtomicThread volatileAtomicThread = new VolatileAtomicThread() ;

        // 开启100个线程对count进行++操作
        for(int x = 0 ; x < 100 ; x++) {
            new Thread(volatileAtomicThread).start();
        }

    }

}
```

执行结果：不保证一定是10000

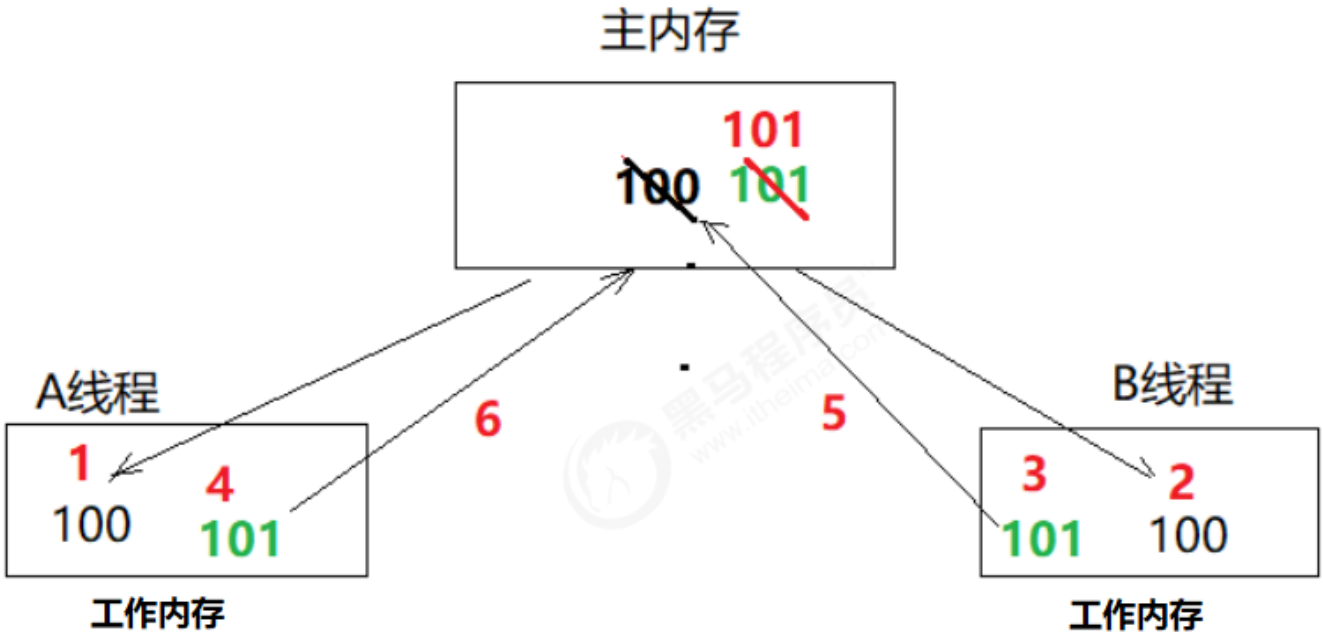
2.1.2 问题原理说明

以上问题主要是发生在count++操作上：

count++操作包含3个步骤：

- 从主内存中读取数据到工作内存
- 对工作内存中的数据进行++操作
- 将工作内存中的数据写回到主内存

count++操作不是一个原子性操作，也就是说在某一个时刻对某一个操作的执行，有可能被其他的线程打断。



1）假设此时x的值是100，线程A需要对改变量进行自增1的操作，首先它需要从主内存中读取变量x的值。由于CPU的切换关系，此时CPU的执行权被切换到了

B线程。A线程就处于就绪状态，B线程处于运行状态

2）线程B也需要从主内存中读取x变量的值,由于线程A没有对x值做任何修改因此此时B读取到的数据还是100

3）线程B工作内存中x执行了+1操作，但是未刷新之主内存中

4）此时CPU的执行权切换到了A线程上，由于此时线程B没有将工作内存中的数据刷新到主内存，因此A线程工作内存中的变量值还是100，没有失效。

A线程对工作内存中的数据进行了+1操作

5）线程B将101写入到主内存

6）线程A将101写入到主内存

虽然计算了2次，但是只对A进行了1次修改。

2.1.3 volatile原子性测试

代码测试

```
// 定义一个int类型的变量
private volatile int count = 0 ;
```

2.1.4 小结

在多线程环境下，volatile关键字可以保证共享数据的可见性，但是并不能保证对数据操作的原子性（在多线程环境下volatile修饰的变量也是线程不安全的）。

在多线程环境下，要保证数据的安全性，我们还需要使用锁机制。

2.1.5 问题解决

使用锁机制

我们可以给count++操作添加锁，那么count++操作就是临界区的代码，临界区只能有一个线程去执行，所以count++就变成了原子操作。

```
public class VolatileAtomicThread implements Runnable {

    // 定义一个int类型的变量
    private volatile int count = 0 ;
    private static final Object obj = new Object();

    @Override
    public void run() {

        // 对该变量进行++操作，100次
        for(int x = 0 ; x < 100 ; x++) {
            synchronized (obj) {
                count++ ;
                System.out.println("count ==========>>>> " + count);
            }
        }

    }

}
```

原子类

概述：java从JDK1.5开始提供了java.util.concurrent.atomic包(简称Atomic包)，这个包中的原子操作类提供了一种用法简单，性能高效，线程安全地更新一个变量的方式。

AtomicInteger

原子型Integer，可以实现原子更新操作

```
public AtomicInteger():                    初始化一个默认值为0的原子型Integer
public AtomicInteger(int initialValue):    初始化一个指定值的原子型Integer

int get():                                 获取值
int getAndIncrement():                     以原子方式将当前值加1，注意，这里返回的是自增前的值。
int incrementAndGet():                     以原子方式将当前值加1，注意，这里返回的是自增后的值。
int addAndGet(int data):                   以原子方式将输入的数值与实例中的值（AtomicInteger里的
value）相加，并返回结果。
int getAndSet(int value):                  以原子方式设置为newValue的值，并返回旧值。
```

演示基本使用。

案例改造

使用AtomicInteger对案例进行改造.

```
public class VolatileAtomicThread implements Runnable {

    // 定义一个int类型的变量
    private AtomicInteger atomicInteger = new AtomicInteger() ;

    @Override
    public void run() {

        // 对该变量进行++操作，100次
        for(int x = 0 ; x < 100 ; x++) {
            int i = atomicInteger.getAndIncrement();
            System.out.println("count ==========>>>> " + i);
        }

    }

}
```

2.3 禁止指令重排序

2.3.1 概述

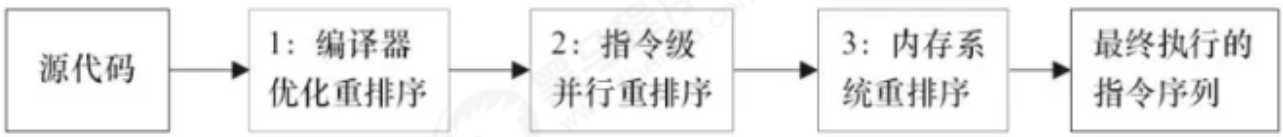
什么是重排序：为了提高性能，编译器和处理器常常会对既定的代码执行顺序进行指令重排序。

原因：一个好的内存模型实际上会放松对处理器和编译器规则的束缚，也就是说软件技术和硬件技术都为同一个目标而进行奋斗：在不改变程序执行结果的前提下，尽可能提高执行效率。JMM对底层尽量减少约束，使其能够发挥自身优势。因此，在执行程序时，**为了提高性能，编译器和处理器常常会对指令进行重排序。**一般重排序可以分为如下三种：

1.编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序；

2.指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果**不存在数据依赖性**，处理器可以改变语句对应机器指令的执行顺序；

3.内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行的。



2.3.2 重排序的好处

重排序可以提高处理的速度。

2.3.3 重排序问题案例演示

引入：重排序虽然可以提高执行的效率，但是在并发执行下，JVM虚拟机底层并不能保证重排序下带来的安全性等问题，请看如下案例：

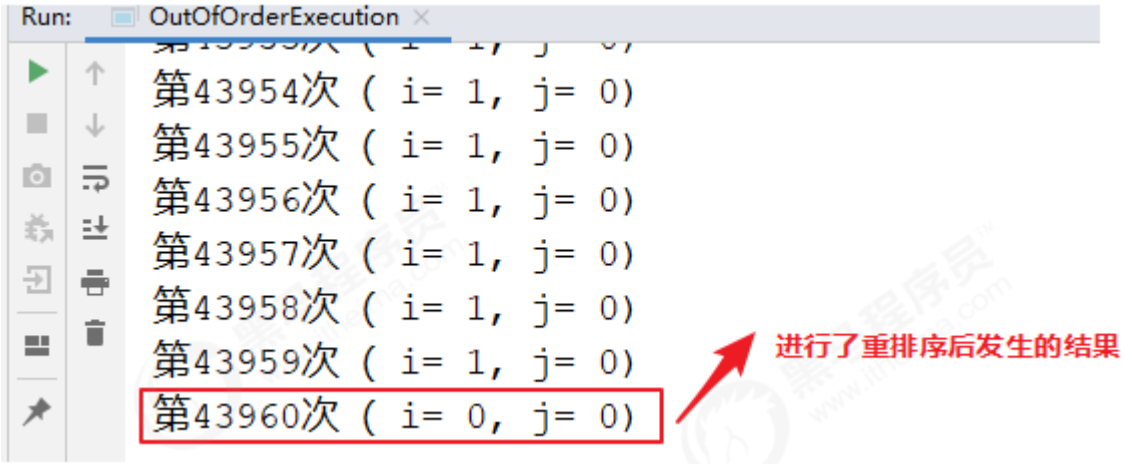
```
public class OutOfOrderExecution {

    private static int i = 0, j = 0;
    private static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
        int count = 0; // 计数
        while(true) {
            count++;
            i = 0;
            j = 0;
            a = 0;
            b = 0;
            Thread one = new Thread(new Runnable() {
                @Override
                public void run() {
                    a = 1;
                    i = b;
                }
            });
            Thread two = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    j = a;
                }
            });
            two.start();
            one.start();
            one.join();
            two.join();

            String result = "第" + count + "次 ( i= " + i + ", j= " + j + ")";
            if (i == 0 && j == 0) {
```

```
                System.out.println(result);
                break;
            } else {
                System.out.println(result);
            }
        }
    }
}
```

以上程序中的4行代码的执行顺序决定了**最终 i 和 j 的值**，在执行的过程中可能会**出现三种**情况如下：

1. **a = 1 ; i=b(0) ; b = 1 ; y = a(1)** ，最终 (i= 0, j= 1)
2. **b = 1 ; j=a(0) ; a = 1 ; i = b(1)** ，最终 (i= 1, j= 0)
3. **b = 1 ; a=1 ; i = b(1) ; j = a(1)** ，最终 (i= 1, j= 1)

但是有一种情况大家可能是没有发现的，经过测试如下：



现象分析

```
a = 1;
i = b;
```

和

```
b = 1;
j = a;
```

按照以前的观点：代码执行的顺序是不会改变的，也就第一个线程是a=1是在i=b之前执行的，第二个线程b=1是在j=a之前执行的。

发生了重排序：在线程1和线程2内部的两行代码的**实际执行顺序**和代码在Java文件中的顺序是不一致的，代码指令并不是严格按照代码顺序执行的，他们的顺序改变了，这样就是发生了重排序，这里颠倒的 是 a = 1 ,i = b 以及j=a , b =1 的顺序，从而发生了指令重排序。直接获取了i = b(0) , j = a(0)的值！显然这个值是不对的。

2.3.4 volatile禁止重排序

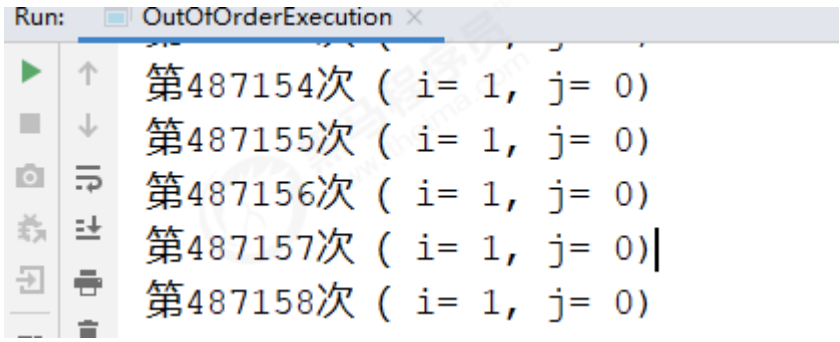
volatile修饰变量后可以实现禁止指令重排序!

volatile禁止重排序案例演示：

```
public class OutOfOrderExecution {
    // 使用volatile修改变量。
    private volatile static int i = 0, j = 0;
    private volatile static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
```

```
        int count = 0; // 计数
        while(true) {
            count++;
            i = 0;
            j = 0;
            a = 0;
            b = 0;
            Thread one = new Thread(new Runnable() {
                @Override
                public void run() {
                    a = 1;
                    i = b;
                }
            });
            Thread two = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    j = a;
                }
            });
            two.start();
            one.start();
            one.join();
            two.join();

            String result = "第" + count + "次 ( i= " + i + ", j= " + j + ")";
            if (i == 0 && j == 0) {
                System.out.println(result);
                break;
            } else {
                System.out.println(result);
            }
        }
    }
}
```

经过很长的测试都没有出现过重排序，从而实现了业务的安全性：



2.3.5 小结

- 使用volatile可以禁止指令重排序，从而修正重排序可能带来的并发安全问题。

第三章 volatile内存语义

3.1 volatile写读建立的happens-before关系

3.1.1 概述

上面的内容讲述了重排序原则，为了提高处理速度，JVM会对代码进行编译优化，也就是指令重排序优化，并发编程下指令重排序会带来一些安全隐患：**如指令重排序导致的多个线程操作之间的不可见性。**如果让程序员再去了解这些底层的实现以及具体规则，那么程序员的负担就太重了，严重影响了并发编程的效率。

从JDK 5开始，提出了happens-before的概念，通过这个概念来阐述操作之间的内存可见性。如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在happens-before关系。这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

所以为了解决多线程的可见性问题，就搞出了happens-before原则，让线程之间遵守这些原则。编译器还会优化我们的语句，所以等于是给了编译器优化的约束。不能让它优化的不知道东南西北了！

简单来说：happens-before 应该翻译成：前一个操作的结果可以被后续的操作获取。讲白点就是前面一个操作变量a赋值为1，那后面一个操作肯定能知道a已经变成了1。

3.1.2 happens-before规则

具体的一共有六项规则：

1. **程序顺序规则（单线程规则）**

- 解释：一个线程中的每个操作，happens-before于该线程中的任意后续操作
  - 同一个线程中前面的所有写操作对后面的操作可见

2. **锁规则（Synchronized,Lock等）**

- 解释：对一个锁的解锁，happens-before于随后对这个锁的加锁。
  - 如果线程1解锁了monitor a，接着线程2锁定了a，那么，线程1解锁a之前的写操作都对线程2可见（线程1和线程2可以是同一个线程）

3. **volatile变量规则**：

- 解释：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。
  - **如果线程1写入了volatile变量v（临界资源），接着线程2读取了v，那么，线程1写入v及之前的写操作都对线程2可见（线程1和线程2可以是同一个线程）**

4. **传递性**：
   - 解释：如果A happens-before B，且B happens-before C，那么A happens-before C。
   - A h-b B， B h-b C 那么可以得到 A h-b C
5. **start()规则：**
   - 解释：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。
   - 假定线程A在执行过程中，通过执行ThreadB.start()来启动线程B，那么线程A对共享变量的修改在接下来线程B开始执行前对线程B可见。注意：线程B启动之后，线程A在对变量修改线程B未必可见
6. **join()规则**

- 解释：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。
    - 线程t1写入的所有变量，在任意其它线程t2调用t1.join()，或者t1.isAlive() 成功返回后，都对t2可见

3.1.3 volatile写读建立的happens-before规则

happens-before有一个原则是：如果A是对volatile变量的写操作，B是对同一个变量的读操作，那么hb(A,B)

案例演示

```
public class VisibilityHP {
    int a = 1;
    int b = 2;

    private void write() {
        a = 3;
        b = a;
    }

    private void read() {
        System.out.println("b=" + b + ";a=" + a);
    }

    public static void main(String[] args) {
        while (true) {
            VisibilityHP test = new VisibilityHP();
            new Thread(new Runnable() {
                @Override
                public void run() {
                    test.write();
                }
            }).start();

            new Thread(new Runnable() {
                @Override
                public void run() {
                    test.read();
                }
            }).start();
        }
    }
}
```

分析以上案例存在四种执行情况：

b=3;a=3

b=2;a=1

b=2;a=3

第四种情况（低概率）是：没给b加volatile,那么有可能出现a=1 , b = 3 。因为a虽然被修改了，但是其他线程不可见，而b恰好其他线程可见，造成了b=3 , a=1。

如何解决第四种情况题呢?

按照happens-before规则，我们只需要给b加上volatile，那么b之前的写入（ a = 3;）将对读取b之后的代码可见，也就是说即使a不加volatile,只要b读取到3，那么b之前的操作（a=3）就一定是可见的，此时就绝对不会出现b=3的时候而读取到a=1了。

3.1.4 volatile重排序规则小结

volatile重排序规则:

第一个操作	第二个操作		
	普通读/写	volatile读	volatile写
普通读/写			不允许
volatile读	不允许	不允许	不允许
volatile写		不允许	不允许

- 写volatile变量时，无论前一个操作是什么，都不能重排序
- 读volatile变量时，无论后一个操作是什么，都不能重排序
- 当先写volatile变量，后读volatile变量时，不能重排序

第四章 volatile高频面试与总结

4.1 long和double的原子性

概述

在java中，long和double都是8个字节共64位(一个字节=8bit)，那么如果是一个32位的系统，读写long或double的变量时会涉及到原子性问题，因为32位的系统要读完一个64位的变量，需要分两步执行，每次读取32位，这样就对double和long变量的赋值就会出现问题： 如果有两个线程同时写一个变量内存，一个进程写低32位，而另一个写高32位，这样将导致获取的64位数据是失效的数据。

案例演示

```
public class LongTest09 implements Runnable{
    private volatile static long aLong = 0;
    private volatile long value;

    public LongTest09(long value) {
        this.setValue(value);
    }
    public long getValue() {
        return value;
    }
    public void setValue(long value) {
        this.value = value;
    }
```

```
    @Override
    public void run() {
        int i = 0;
        while (i < 100000) {
            LongTest09.aLong = this.getValue();
            i++;
            // 赋值操作！！
            long temp = LongTest09.aLong;
            // 取出值操作！！！
            if (temp != 1L && temp != -1L) {
                System.out.println("出现错误结果" + temp);
                System.exit(0);
            }
        }
        System.out.println("运行正确");
    }


    public static void main(String[] args) throws InterruptedException {
        // 获取并打印当前JVM是32位还是64位的
        String sysNum = System.getProperty("sun.arch.data.model");
        System.out.println("系统的位数："+sysNum);
        LongTest09 t1 = new LongTest09(1);
        LongTest09 t2 = new LongTest09(-1);
        Thread T1 = new Thread(t1);
        Thread T2 = new Thread(t2);
        T1.start();
        T2.start();
        T1.join();
        T2.join();
    }
}
```

测试结果

上面的代码在32位环境和64位环境执行的结果是不一样的： **32位环境：出现错误结果** 原因：32位环境无法一次读取long类型数据，多线程环境下对Long变量的读写是不完整的，导致temp变量既不等于1也不等于-1。出现了long和double读写的原子性问题了。 **64位环境：运行正确**

小结

结论：如果是在64位的系统中，那么对64位的long和double的读写都是原子操作的。即可以以一次性读写long或double的整个64bit。如果在32位的JVM上，long和double就不是原子性操作了。

解决方法：需要使用volatile关键字来防止此类现象

- 对于64位的long和double，如果没有被volatile修饰，那么对其操作可以不是原子的。在操作的时候，可以分成两步，每次对32位操作。
- 如果使用volatile修饰long和double，那么其读写都是原子操作
- 在实现JVM时，可以自由选择是否把读写long和double作为原子操作；

- java中对于long和double类型的写操作不是原子操作，而是分成了两个32位的写操作。读操作是否也分成了两个32位的读呢？在JSR-133之前的规范中，读也是分成了两个32位的读，但是从JSR-133规范开始，即JDK5开始，读操作也都具有原子性；
- java中对于其他类型的读写操作都是原子操作(除long和double类型以外)；
- 对于引用类型的读写操作都是原子操作，无论引用类型的实际类型是32位的值还是64位的值；
- Java商业虚拟机已经解决了long和double的读写操作的原子性。

4.2 volatile在双重检查加锁的单例中的应用

单例概述

单例是需要在内存中永远只能创建一个类的实例，

单例的作用：节约内存和保证共享计算的结果正确，以及方便管理。

单例模式的适用场景：

- 全局信息类：例如任务管理器对象，或者需要一个对象记录整个网站的在线流量等信息。
- 无状态工具类：类似于整个系统的日志对象等，我们只需要一个单例日志对象负责记录，管理系统日志信息。

单例模式有8种

单例模式我们可以提供出8种写法，有很多时候我们存在**饿汉式单例的概念**，以及**懒汉式单例的概念。**

饿汉式单例的含义是：在获取单例对象之前对象已经创建完成了。

懒汉式单例是指：在真正需要单例的时候才创建出该对象。

饿汉单例的2种写法

特点：在获取单例对象之前对象已经创建完成了。

饿汉式（静态常量）

```
/**
 * 饿汉式（静态常量）
 */
public class Singleton1 {
    private  static final Singleton1 INSTANCE = new Singleton1();
    private Singleton1() {

    }
    public static Singleton1 getInstance() {
        return INSTANCE;
    }
}
```

饿汉式（静态代码块）

```
/**
 *  饿汉式（静态代码块）（可用）
 */
```

```
public class Singleton2 {

    private final static Singleton2 INSTANCE;

    static {
        INSTANCE = new Singleton2();
    }

    private Singleton2() {
    }

    public static Singleton2 getInstance() {
        return INSTANCE;
    }
}
```

懒汉式单例4种写法

特点：在真正需要单例的时候才创建出该对象。在Java程序中，有时候可能需要推迟一些高开销对象的初始化操作，并且只有在使用这些对象的时候才初始化，此时，程序员可能会采用延迟初始化。

值得注意的是：要正确的实现线程安全的延迟初始化还是需要一些技巧的，否则很容易出现问题。

懒汉式（线程不安全）

```
/**
 * 描述：  懒汉式（线程不安全，不推荐的方案）
 */
public class Singleton3 {

    private static Singleton3 instance;

    private Singleton3() {

    }

    public static Singleton3 getInstance() {
        if (instance == null) {
            instance = new Singleton3();
        }
        return instance;
    }
}
```

懒汉式（线程安全，性能差）

分析

使用synchronized关键字修饰方法包装线程安全，但性能差多，并发下只能有一个线程正在进入获取单例对象。

案例

```
/**
```

```
 * 描述:  懒汉式（线程安全 ,性能差，不推荐）
 */
public class Singleton4 {

    private static Singleton4 instance;

    private Singleton4() {

    }

    public synchronized static Singleton4 getInstance() {
        if (instance == null) {
            instance = new Singleton4();
        }
        return instance;
    }
}
```

懒汉式（线程不安全）

特点：是一种优化后的似乎线程安全的机制。

```
/**
 * 描述:  懒汉式（线程不安全 ,不推荐）
 */
public class Singleton5 {
    private static Singleton5 instance;
    private Singleton5() {

    }
    public static Singleton5 getInstance() {
        // 性能得到了优化，但是依然不能保证第一次获取对象的线程安全!
        if (instance == null) {
            // A , B
            synchronized (Singleton5.class) {
                instance = new Singleton5();
            }
        }
        return instance;
    }
}
```

懒汉式（volatile双重检查模式,推荐）

案例代码

```
/**
 * 描述:  双重检查，推荐面试中进行使用。
 */
public class Singleton6 {
    // 静态属性，volatile保证可见性和禁止指令重排序
    private volatile static Singleton6 instance = null;
```

```
    // 私有化构造器
    private Singleton6(){}

    public static  Singleton6 getInstance(){
        // 第一重检查锁定
        if(instance==null){
            // 同步锁定代码块
            synchronized(Singleton6.class){
                // 第二重检查锁定
                if(instance==null){
                    // 注意：非原子操作
                    instance=new Singleton6();
                }
            }
        }
        return instance;
    }
}
```

分析

双重检查的优点：线程安全，延迟加载，效率较高！

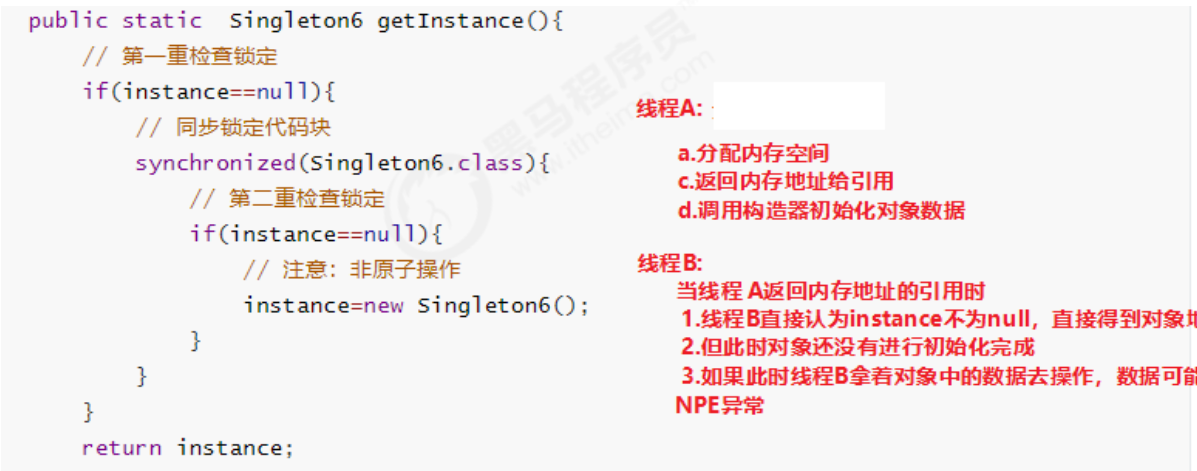
以上是否就可以了呢，答案是否定的，实际上我们还需要加上volatile修饰，为何要使用volatile保证安全?

1、禁止指令重排序

- 对象实际上创建对象要进过如下几个步骤

```
a．分配内存空间。
b．调用构造器，  初始化实例。
c．返回地址给引用
```

- 所以，new Singleton()是一个非原子操作，编译器可能会重排序【构造函数可能在整个对象初始化完成前执行完毕，即赋值操作（只是在内存中开辟一片存储区域后直接返回内存的引用）在初始化对象前完成】。而线程C在线程A赋值完时判断instance就不为null了，此时C拿到的将是一个没有初始化完成的半成品。这样是很危险的。因为极有可能线程C会继续拿着个没有初始化的对象中的数据进行操作，此时容易触发**“NPE异常”**
- **图解如下**



2、保证可见性。

- 由于可见性问题，线程A在自己的工作线程内创建了实例，但此时还未同步到主存中；此时线程C在主存中判断instance还是null，那么线程C又将在自己的工作线程中创建一个实例，这样就创建了多个实例。
- 如果加上了volatile修饰instance之后，保证了可见性，一旦线程A返回了实例，线程C可以立即发现Instance不为null。

静态内部类单例方式

引入：JVM在类初始化阶段（即在Class被加载后，且线程使用之前），会执行类的初始化。在执行类的初始化期间，JVM会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

基于这个特性，可以实现另一种线程安全的延迟初始化方案

```
/**
 * 描述：静态内部类方式，可用
 */
public class Singleton7 {

    private Singleton7() {
    }

    private static class SingletonInstance {
        private static final Singleton7 INSTANCE = new Singleton7();
    }

    // 线程 A   线程B
    public static Singleton7 getInstance() {
        return SingletonInstance.INSTANCE;
    }
```

小结

1. 静态内部类是在被调用时才会被加载，这种方案实现了懒汉单例的一种思想，需要用到的时候才去创建单例，加上JVM的特性，这种方式又实现了线程安全的创建单例对象。

2.通过对比基于volatile的双重检查锁定方案和基于类初始化方案的对比，我们会发现基于类初始化的方案的实现代码更简洁。但是基于volatile的双重检查锁定方案有一个额外的优势：**除了可以对静态字段实现延迟加载初始化外，还可以对实例字段实现延迟初始化。**

枚举实现单例

```
/**
 * 描述:       枚举单例
 */
public enum Singleton8 {
    INSTANCE;

    public void whatever() {

    }
}
```

4.3 volatile的使用场景

4.3.1 纯赋值操作

概述

volatile不适合做a++等操作。

适合做纯复制操作:如 boolean flag = false/true;

案例代码

```
public class UseVolatile1 implements Runnable {

    volatile boolean flag = false;
    AtomicInteger realA = new AtomicInteger();

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            setDone();
            realA.incrementAndGet();
        }
    }

    private void setDone() {
        flag = true;   // 纯赋值操作符合预期
        // flag = !flag ; // 这样做不符合预期
    }
}

class Test{
     public static void main(String[] args) throws InterruptedException {
        UseVolatile1 r =  new UseVolatile1();
        Thread thread1 = new Thread(r);
        Thread thread2 = new Thread(r);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(r.flag);
        System.out.println(r.realA.get());
    }
}
```

小结

volatile可以适合做多线程中的纯赋值操作：如果一个共享变量自始至终只被各个线程赋值,而没有其他的操作,那么就可以用volatile来代替synchronized或者代替原子变量,因为赋值自身是有原子性的,而volatile又保证了可见性,所以就足以保证线程安全。

4.3.2 触发器

概念

按照volatile的可见性和禁止重排序以及happens-before规则，volatile可以作为刷新之前变量的触发器。我们可以将某个变量设置为volatile修饰，其他线程一旦发现该变量修改的值后，触发获取到的该变量之前的操作都将是最新的且可见。

案例演示

```
public class VisibilityHP {
    int a = 1;
    int b = 2;
    int c = 3;
    volatile boolean flag = false;

    private void write() {
        a = 3;
        b = 4 ;
        c = a;
        flag = true;
    }

    private void read() {
        // flag被volatile修饰，充当了触发器，一旦值为true,此处立即对变量之前的操作可见。
        while(flag){
              System.out.println("a=" + a + ";b=" + b +",c="+c );
        }
    }

    public static void main(String[] args) {
        while (true) {
            VisibilityHP test = new VisibilityHP();
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    test.write();
                }
            }).start();

            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    test.read();
                }
```

```
            }).start();
        }
    }
}
```

小结

volatile可以作为刷新之前变量的触发器。我们可以将某个变量设置为volatile修饰，其他线程一旦发现该变量修改的值后，触发获取到的该变量之前的操作都将是最新的且可见。

4.4 volatile与synchronized

4.4.1 区别

- volatile只能修饰实例变量和类变量，而synchronized可以修饰方法，以及代码块。
- volatile保证数据的可见性，但是不保证原子性(多线程进行写操作，不保证线程安全);而synchronized是一种排他（互斥）的机制。
- volatile用于禁止指令重排序：可以解决单例双重检查对象初始化代码执行乱序问题。
- volatile可以看做是轻量版的synchronized,volatile不保证原子性，但是如果是对一个共享变量进行多个线程的赋值，而没有其他的操作，那么就可以用volatile来代替synchronized,因为赋值本身是有原子性的，而volatile又保证了可见性，所以就可以保证线程安全了。

4.5 volatile的总结

4.5.1 总体总结

1. volatile 修饰符适用于以下场景:某个属性被多个线程共享,其中有一个线程修改了此属性,其他线程可以立即得到修改后的值,比如boolean flag ;或者作为触发器,实现轻量级同步。
2. volatile属性的读写操作都是无锁的,它不能替代synchronized ,因为它没有提供原子性和互斥性。因为无锁,不需要花费时间在获取锁和释放锁_上,所以说它是低成本的。
3. volatile只能作用于属性,我们用volatile修饰属性,这样compilers就不会对这个属性做指令重排序。
4. volatile 提供了可见性,任何一个线程对其的修改将立马对其他线程可见。volatile 属性不会被线程缓存,始终从主存中读取。
5. volatile提供了happens-before保证,对volatile变量v的写入happens- before所有其他线程后续对v的读操作。
6. volatile可以使得long和double的赋值是原子的。
7. volatile可以在单例双重检查中实现可见性和禁止指令重排序，从而保证安全性。