链滴

重新认识 volatile

作者：zouchanglin
原文链接：https://ld246.com/article/1576070925464
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

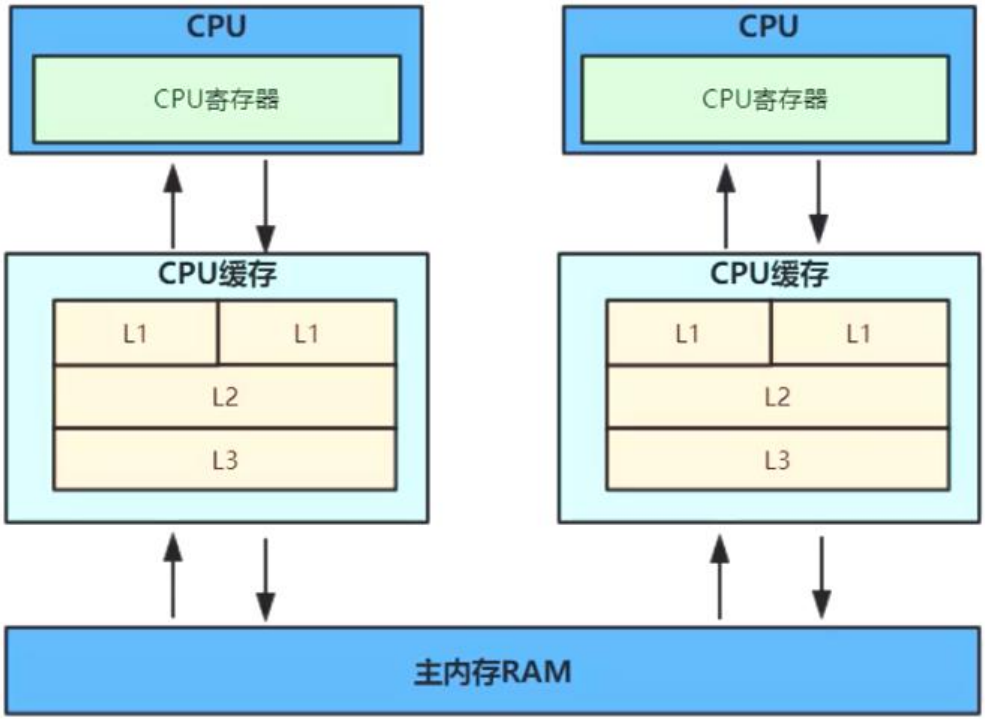
1、CPU多核心缓存架构分析

并发编程：肯定是为了更合理充分的利用多核CPU架构的性能

CPU 主频率远远高于主存，所以引入CPU缓存，CPU加载数据无法绕过缓存，而且对于多核CPU，级缓存不是共享的，二级和三级缓存是线程共享的！CPU计算的数据并不是直接从主存中去拿，而且过层层在缓存中寻找。

下图是一个单CPU多核心的架构图：

CPU多核缓存架构图

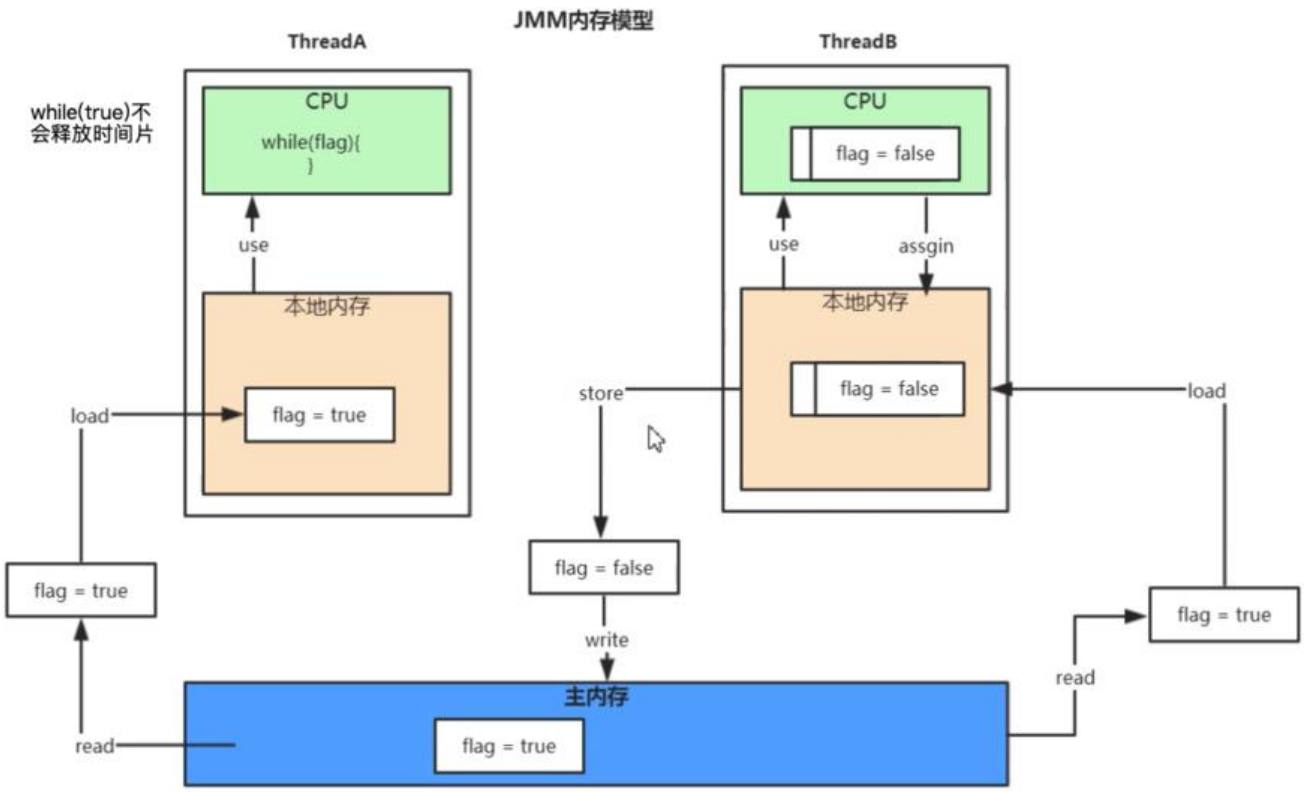


2、CPU缓存一致性协议

先看这样一段代码：

```java
package xpu.demo;

public class VisibilityTest {
    private boolean flag = true;

    private void refresh(){
        flag = false;
        System.out.println(Thread.currentThread().getName()+"修改flag");
    }

    private void load() {
        System.out.println(Thread.currentThread().getName()+"开始执行...");
        while (flag){

        }
        System.out.println(Thread.currentThread().getName()+"跳出循环: flag=" + flag);
    }

    public static void main(String[] args) {
        VisibilityTest test = new VisibilityTest();
        new Thread(()->test.load(), "threadA").start();
        try {
            Thread.sleep(2000);
            new Thread(()->test.refresh(), "threadB").start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

加载过程就是ThreadA通过read指令读取到flag，在通过load指令加载到缓存，也就是JMM中的本内存，CPU再通过寄存器去使用flag，ThreadB也是同样的流程，通过assign指令为本地内存中的fla赋值，再写入主存：



由于while(true)并没有释放时间片，所以在这里可以让它去进行上下文切换，则就会有时间清除缓存

Title

```
private void load() {
    System.out.println(Thread.currentThread().getName()+"开始执行...");
    while (flag){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println(Thread.currentThread().getName()+"跳出循环: flag=" + flag);
}
```

两次的执行结果肯定很简单

原文链接：重新认识 volatile

```java
    private void load() {
        System.out.println(Thread.currentThread().getName()+"开始执行...");
        while (flag){
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName()+"跳出循环: flag=" + flag);
    }

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

VisibilityTest › load()

Run: VisibilityTest

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_231.jdk/Contents/Home/bin/java ...
threadA开始执行...
threadB修改flag
threadA跳出循环: flag=false
```

```java
    private void load() {
        System.out.println(Thread.currentThread().getName()+"开始执行...");
        while (flag){
//          try {
//              Thread.sleep(1000);
//          } catch (InterruptedException e) {
//              e.printStackTrace();
//          }
        }
        System.out.println(Thread.currentThread().getName()+"跳出循环: flag=" + flag);
    }

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

VisibilityTest › load()

Run: VisibilityTest

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_231.jdk/Contents/Home/bin/java ...
threadA开始执行...
threadB修改flag
```

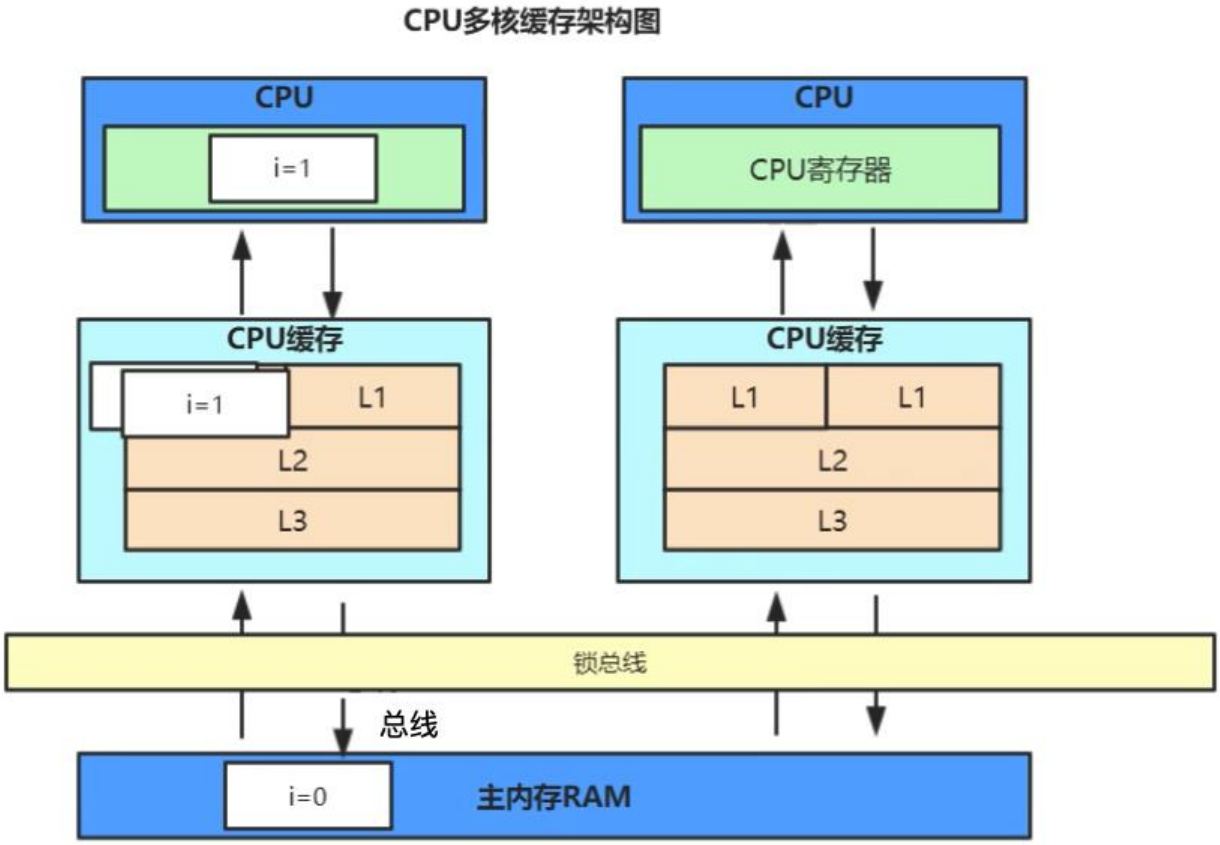
什么情况下的上下文切换不会清除缓存呢？可以设置一个非常小的休眠时间

原文链接：重新认识 volatile

```java
private void load() {
    System.out.println(Thread.currentThread().getName()+"开始执行...");
    while (flag){
        shortWait(500000); // 500000纳秒(500微秒)足够清除缓存了
        //shortWait(500); // 500纳秒还没有清除缓存
    }
    System.out.println(Thread.currentThread().getName()+"跳出循环: flag=" + flag);
}

// 纳秒级别的等待
private static void shortWait(long interval){
    long start = System.nanoTime();
    long end;
    do{
        end = System.nanoTime();
    }while (start + interval >= end);
}
```

这个时候，设置500纳秒的Sleep和500000纳秒的等待会导致结果完全不一样！！

这与CPU多核心架构有关系，CPU修改之后的值并不会立即刷新到主存，这便导致了缓存不一致的问，这是属于CPU架构的问题，不仅仅存在于JVM层面，只要是这种的CPU架构都会出现这种问题，所首先得靠加锁来解决这种问题，在早期有一种东西叫做总线锁：



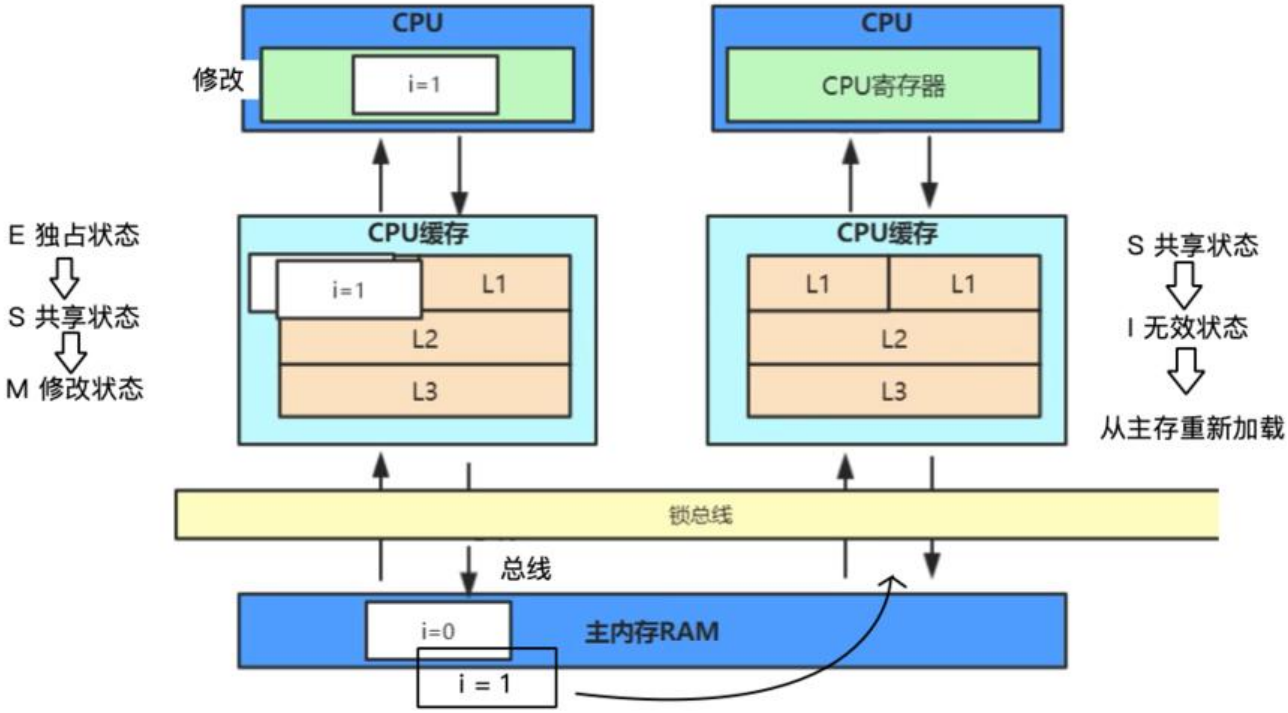
一旦发生数据修改的回写操作，直接把总线加锁，这样别的线程在使用数据的时候就不得不重新从主得到新的数据，而且存在严重的性能问题，如果存在大量IO操作时，还使用这种总线锁，那么肯定IO能肯定会下降！

原文链接：重新认识 volatile

于是出现了缓存一致性协议：

主要用到的缓存一致性协议时MESI协议（M修改、E独占、S共享、I无效四种状态，这四种状态记录是缓存行的转态）。

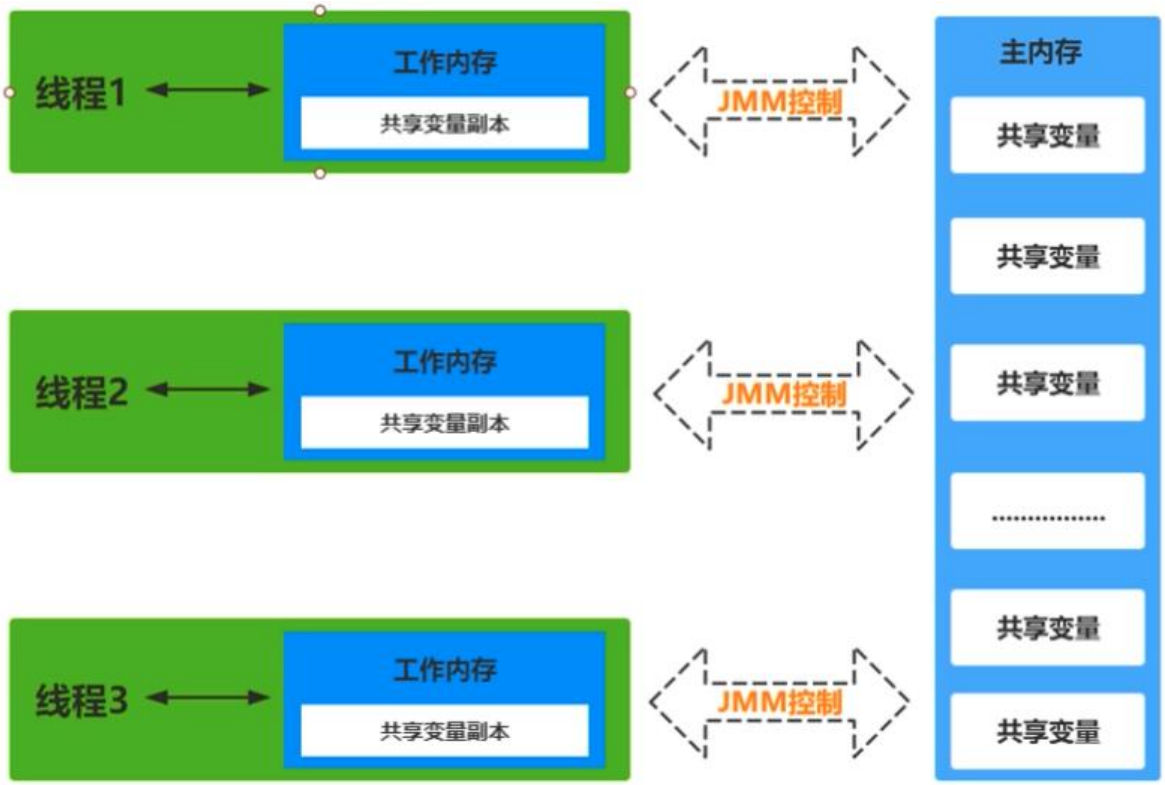
这里还有一个机制叫做总线嗅探机制，嗅探通过总线的数据，如果只有一个核心用到，那么就会给这缓存行一个状态叫做E状态（独占状态），当另一个线程也用到了这个变量的时候，CPU就会通过广机制通知其他的核心把这个缓存行的状态修改为S状态（共享状态），接下来其中一个线程对这个变进行了修改，那么对于这个线程来说，这个变量的缓存行变成了M状态（修改状态），回写的时候会过总线，总线嗅探机制嗅探到这个变量的缓存行已经是M状态，它就会通过其他的核心，说缓存无效则其他核心上的缓存行就会变成I状态（无效状态），变成无效状态后如果还需要使用这个变量，那么定只能重新从主存中去加载这个变量到缓存，而且必须等待修改核心修改（回写主存）完成！



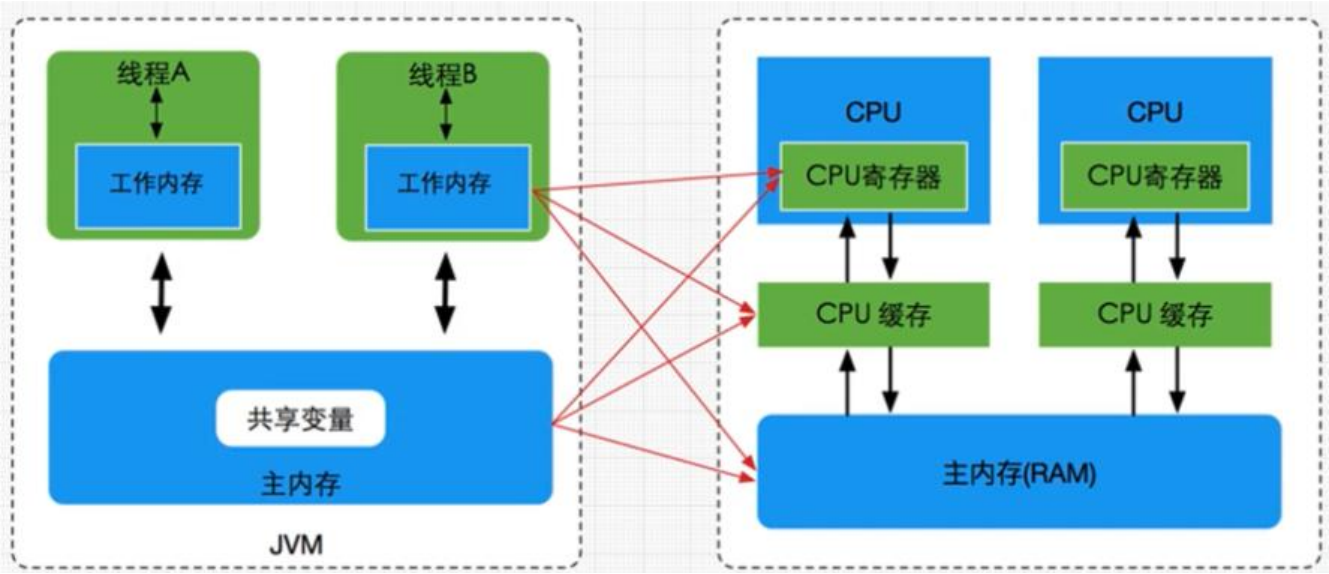
这里锁的缓存行最大值为 64 byte ，总线锁主要是用到了lock原语

3、内存模型JMM实现原理

JMM描述的是一种抽象的概念，一组规则，通过这组规则控制程序中各个变量在共享数据区域和私有据区域的访问方式，JMM是围绕原子性、有序性、可见性展开的



这个JMM模型只是一个抽象的概念，图上画出的也只是逻辑空间，通过对应到硬件内存架构是这样的：



4、Volatile关键字原理剖析

在上面的例子中，我们明显可以通过volatile关键字来解决这个问题，那么volatile又是如何实现的呢？

通过汇编指令来看看就知道了，运行时加虚拟机运行参数：

-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -Xcomp

但是Mac环境可能会由于缺少部分包：hsdis-amd64.dylib，在这里下载就好了 https://github.com/volvedmicrobe/benchmarks/blob/master/hsdis-amd64.dylib，下载完毕后放在：/Library/Java/JavaVirtualMachines/jdk1.8.0_231.jdk/Contents/Home/jre 就好了

先看看如何保证可见性的：

底层实现了 lock addl $0x0,(%rsp)，触发了缓存一致性协议

```
0x0000000109f78a7e: mov    $0x0,%edx
0x0000000109f78a83: mov    %dl,0xc(%rsi)
0x0000000109f78a86: lock addl $0x0,(%rsp)     ;*putfield flag
                                              ; - xpu.demo.VisibilityTest::refresh@2 (line 7)
```

什么是重排序呢？是指编译器生成了指令序列，处理乱序执行！

接下来看看指令重排序的一个例子：

Title

```java
package xpu.demo;

// volatile保证有序性
public class ReOrderTest {
    private static int x = 0, y = 0;
    private static int a = 0, b = 0;

    public static void main(String[] args)
            throws InterruptedException {
        int i = 0;
        while (true){
            i++;

            x = 0;
            y = 0;
            a = 0;
            b = 0;

            Thread thread1 = new Thread(()->{
                shortWait(20000);
                a = 1;
                x = b;
            });

            Thread thread2 = new Thread(()->{
                b = 1;
                y = a;
            });

            thread1.start();
            thread2.start();
            thread1.join();
            thread2.join();

            System.out.println("第"+ i + "次(" + x + "," + y +")");
            if(x==0 && y==0){
                break;
            }
        }
    }

    // 纳秒级别的等待
    private static void shortWait(long interval){
        long start = System.nanoTime();
        long end;
        do{
            end = System.nanoTime();
        }while (start + interval >= end);
    }
}

```

原文链接：重新认识 volatile

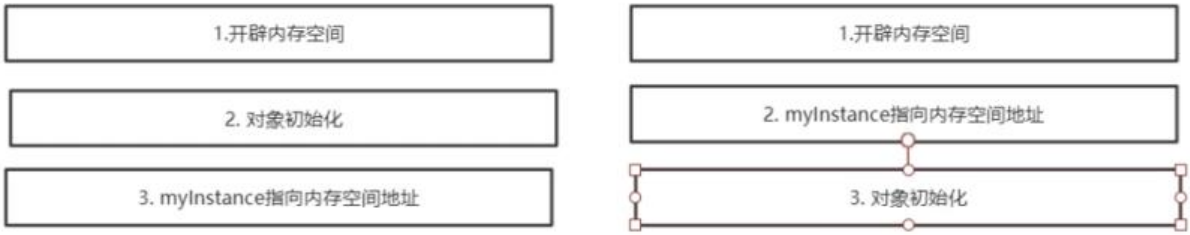
打印出来（0，0）（0，1）（1，0）（1，1）都有

这就好比单例模式，请看下面这个例子：

Title

```java
public class SingletonFactory {
    private volatile static SingletonFactory myInstance;

    private static SingletonFactory getMyInstance(){
        if(myInstance == null){
            synchronized (SingletonFactory.class){
                if(myInstance == null){
                    myInstance = new SingletonFactory();
                }
            }
        }
        return myInstance;
    }
}
```

myInstance = new SingletonFactory();



myInstance = new SingletonFactory()，这句话主要是三个指令构成的，如果不能保证指令有序性话，那么拿到的对象就是未被初始化的对象，是无效的！

对于x86架构的CPU，加上volatile写后面的storeLoad内存屏障，我们也可以手动加上内存屏障：

Title

```java
package xpu.demo;


import sun.misc.Unsafe;

import java.lang.reflect.Field;

// volatile保证有序性
public class ReOrderTest {
    private static int x = 0, y = 0;
    private static int a = 0, b = 0;

    public static void main(String[] args)
            throws InterruptedException {
        int i = 0;
        while (true){
            i++;

            x = 0;
            y = 0;
            a = 0;
            b = 0;

            Thread thread1 = new Thread(()->{
                shortWait(20000);
                a = 1; //volatile 写，内存屏障StoreLoad
                //手动加内存屏障
                UnsafeFactory.getUnsafe().storeFence();
                x = b; //volatile 读，再普通写
            });

            Thread thread2 = new Thread(()->{
                b = 1;
                //手动加内存屏障
                UnsafeFactory.getUnsafe().storeFence();
                y = a;
            });

            thread1.start();
            thread2.start();
            thread1.join();
            thread2.join();

            System.out.println("第"+ i + "次(" + x + "," + y +")");
            if(x==0 && y==0){
                break;
            }
        }
    }

    // 纳秒级别的等待
    private static void shortWait(long interval){
        long start = System.nanoTime();
        long end;
        do{
            end = System.nanoTime();
        }while (start + interval >= end);
```

原文链接：重新认识 volatile

这个加上内存屏障的方法定义是 public native void storeFence(); 因此是原生实现，可以看看OpenJK的虚拟机实现，可以看到对于64位机器使用的是rsp寄存器，对于非64位用的是esp寄存器

```
#include "vm_version_x86.hpp"

// Implementation of class OrderAccess.

inline void OrderAccess::loadload()   { acquire(); }
inline void OrderAccess::storestore() { release(); }
inline void OrderAccess::loadstore()  { acquire(); }
inline void OrderAccess::storeload()  { fence(); }
```

```
inline void OrderAccess::fence() {
  if (os::is_MP()) {
    // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
    __asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
#else
    __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
  }
}
```

5、可见性、有序性、原子性详解

并发编程的三大特性：

可见性、有序性、原子性

volatile保证可见性与有序性，但是不能保证原子性，要保证原子性需要借助 synchronized、Lock锁制，同理也能保证有序性与可见性，因为 synchronized和Lock能够保证任一时刻只有个线程访问该码块。关于volatile不保证原子性这个其实很好证明：

```
package xpu.demo;
public class AtomTest {
    private static volatile int sum = 0;

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                for (int j = 0; j < 10000; j++) {
                    sum++;
                }
            }).start();
        }
        System.out.println(sum);
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_231.jdk/Contents/Home/bin/java ...
55558

Process finished with exit code 0
```

原文链接：重新认识 volatile

因为线程进行了很多次无效计算，所以结果并不是100000，他们都是从主存中拿的值，而且值都是的，但是计算过程却不是原子的，准确的说应该是资源并没有被锁定，导致自己修改的时候别人也在改，所以正确理解volatile的作用是很重要的！

最后看看CAS操作对应的汇编指令：

原文链接：重新认识 volatile