主讲老师：Fox

课前说明：
JMM属于整个Java并发编程中最难的部分也是最重要的部分（JAVA多线程通信模型——共享内存模型），涉及的理论知识比较多，我会从三个维度去分析：

- JAVA层面
- JVM层面
- 硬件层面

这块如何学？
这部分理解并发的三大特性，JMM工作内存和主内存关系，知道多线程之间如何通信的，掌握volatile能保证可见性和有序性，CAS就可以了，后续JVM层面和硬件层面的分析，基础比较薄弱的同学听不懂可以先跳过，从后面的Java锁机制课程听起，掌握常用的并发工具类，并发容器之后再来看JMM这块。

并发专题不可避免的会涉及到计算机组成原理和操作系统知识，对这块基础比较薄弱，感兴趣，想系统性学习的同学可以关注影子老师录制的计算机组成原理和操作系统课程（持续录制中）。
计算机基础系列-计算机组成原理
计算机基础系列-操作系统

文档：1. 并发编程之深入理解JMM&volatile详?..
链接：http://note.youdao.com/noteshare?id=b59b76f2e515429833e56e1b5c4fdb62&sub=7DBA1EF567514D178F3ED5FD99484FE1

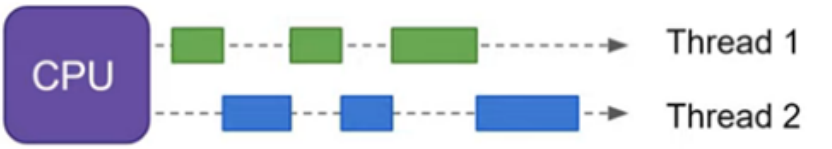
并发知识体系：https://www.processon.com/view/link/615d4a610e3e74663e97fa0e#map

并发和并行

目标都是最大化CPU的使用率

并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。所以无论从微观还是从宏观来看，二者都是一起执行的。

并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



并行在多处理器系统中存在，而并发可以在单处理器和多处理器系统中都存在，并发能够在单处理器系统中存在是因为并发是并行的假象，并行要求程序能够同时执行多个操作，而并发只是要求程序假装同时执行多个操作（每个小时间片执行一个操作，多个操作快速切换执行）

多线程同步，互斥，分工

并发三大特性

并发编程Bug的源头：可见性、原子性和有序性问题

可见性

当一个线程修改了共享变量的值，其他线程能够看到修改的值。Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方法来实现可见性的。

如何保证可见性

- 通过 volatile 关键字保证可见性。
- 通过 内存屏障保证可见性。
- 通过 synchronized 关键字保证可见性。
- 通过 Lock保证可见性。
- 通过 final 关键字保证可见性

有序性

即程序执行的顺序按照代码的先后顺序执行。JVM 存在指令重排，所以存在有序性问题。

如何保证有序性

- 通过 volatile 关键字保证可见性。
- 通过 内存屏障保证可见性。
- 通过 synchronized关键字保证有序性。
- 通过 Lock保证有序性。

原子性

一个或多个操作，要么全部执行且在执行过程中不被任何因素打断，要么全部不执行。在 Java 中，对基本数据类型的变量的读取和赋值操作是原子性操作（64位处理器）。不采取任何的原子性保障措施的自增操作并不是原子性的。

如何保证原子性

- 通过 synchronized 关键字保证原子性。
- 通过 Lock保证原子性。
- 通过 CAS保证原子性。

思考：在 32 位的机器上对 long 型变量进行加减操作是否存在并发隐患？

https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7

可见性问题深入分析

我们通过下面的Java小程序来分析Java的多线程可见性的问题

```java
/**
 * @author Fox
 *
 * -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -Xcomp
 */
public class VisibilityTest {

 private boolean flag = true;

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

  public void load() {
  System.out.println(Thread.currentThread().getName() + "开始执行.....");
  int i = 0;
  while (flag) {
  i++;
  //TODO 业务逻辑

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

  public static void main(String[] args) throws InterruptedException {
  VisibilityTest test = new VisibilityTest();

  // 线程threadA模拟数据加载场景
  Thread threadA = new Thread(() -> test.load(), "threadA");
  threadA.start();

  // 让threadA执行一会儿
  Thread.sleep(1000);
  // 线程threadB通过flag控制threadA的执行时间
  Thread threadB = new Thread(() -> test.refresh(), "threadB");
  threadB.start();

  }


  public static void shortWait(long interval) {
  long start = System.nanoTime();
  long end;
```

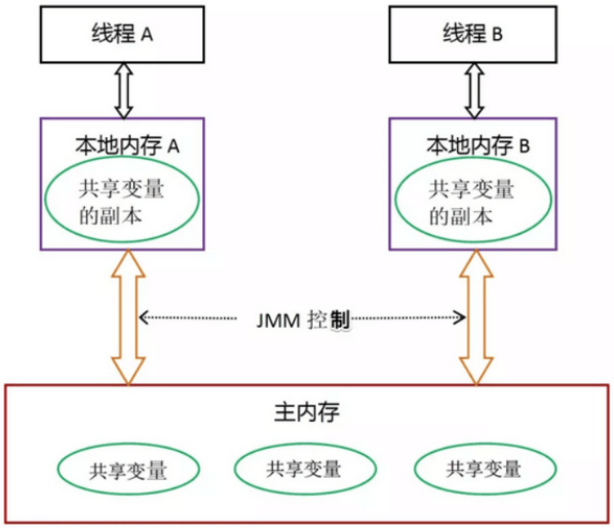
```
45  do {
46  end = System.nanoTime();
47  } while (start + interval >= end);
48  }
49 }
```

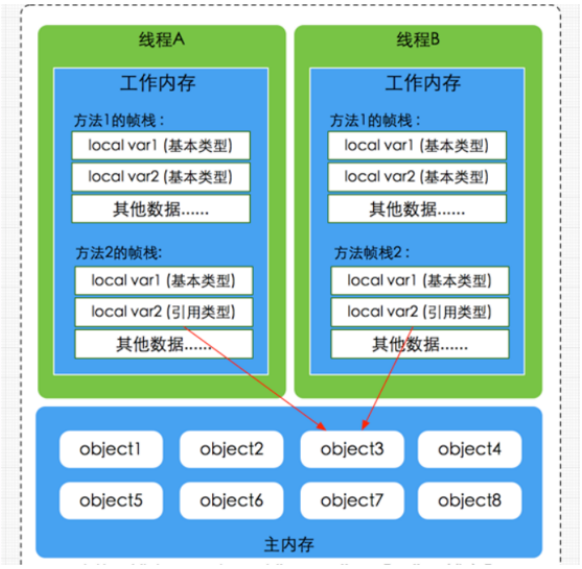
思考：上面例子中为什么多线程对共享变量的操作存在可见性问题?

Java内存模型（JMM）

JMM定义

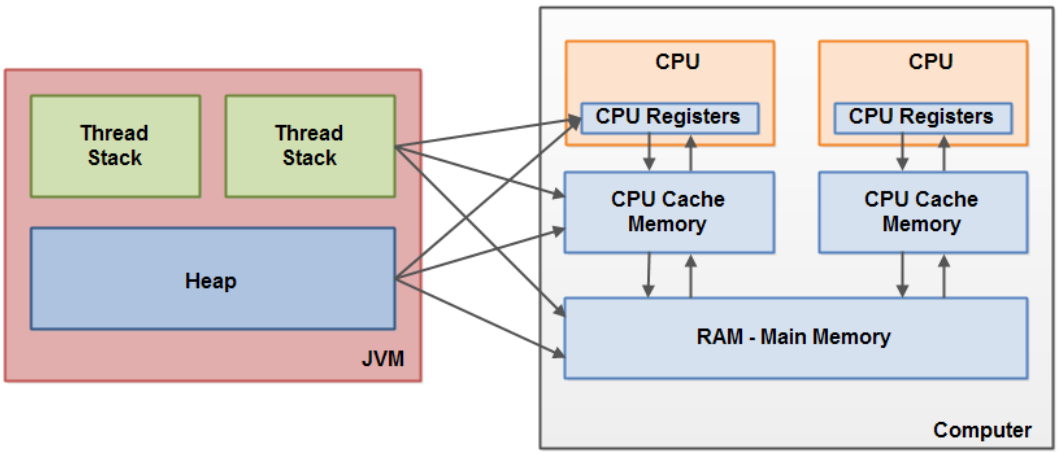
Java虚拟机规范中定义了Java内存模型（Java Memory Model，JMM），用于屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的并发效果，JMM规范了Java虚拟机与计算机内存是如何协同工作的：规定了一个线程如何和何时可以看到由其他线程修改过后的共享变量的值，以及在必须时如何同步的访问共享变量。JMM描述的是一种抽象的概念，一组规则，通过这组规则控制程序中各个变量在共享数据区域和私有数据区域的访问方式，JMM是围绕原子性、有序性、可见性展开的。





JMM与硬件内存架构的关系

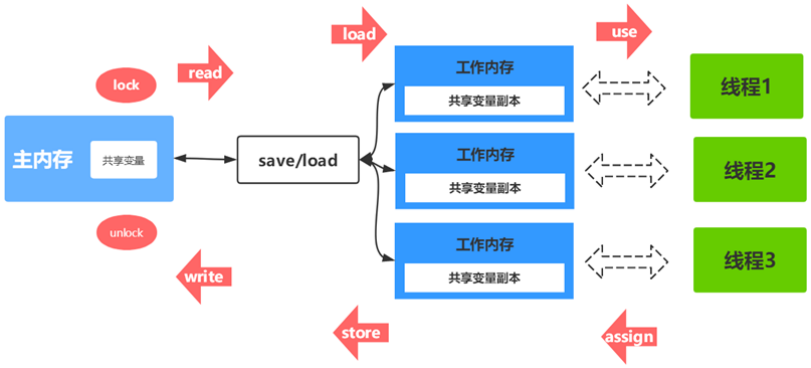
Java内存模型与硬件内存架构之间存在差异。硬件内存架构没有区分线程栈和堆。对于硬件，所有的线程栈和堆都分布在主内存中。部分线程栈和堆可能有时候会出现在CPU缓存中和CPU内部的寄存器中。如下图所示，Java内存模型和计算机硬件内存架构是一个交叉关系：



内存交互操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下八种操作来完成：

- lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
- unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
- load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
- use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
- assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
- write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。



Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

- 如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行read和load操作， 如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。但Java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。

- 不允许read和load、store和write操作之一单独出现
- 不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。
- 不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。
- 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。
- 一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。lock和unlock必须成对出现
- 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行load或assign操作初始化变量的值
- 如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。
- 对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

问题：
什么时候刷主内存，线程执行结束刷回主内存吗，还是一旦更新完之后就立马刷主内存
本地内存什么时候会没有

JMM的内存可见性保证

按程序类型，Java程序的内存可见性保证可以分为下列3类：

- 单线程程序。单线程程序不会出现内存可见性问题。编译器、runtime和处理器会共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同。
- 正确同步的多线程程序。正确同步的多线程程序的执行将具有顺序一致性（程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同）。这是JMM关注的重点，JMM通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。
- 未同步/未正确同步的多线程程序。JMM为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值。未同步程序在JMM中执行时，整体上是无序的，其执行结果无法预知。 JMM不保证未同步程序的执行结果与该程序在顺序一致性模型中的执行结果一致。

未同步程序在JMM中的执行时，整体上是无序的，其执行结果无法预知。未同步程序在两个模型中的执行特性有如下几个差异。

1）顺序一致性模型保证单线程内的操作会按程序的顺序执行，而JMM不保证单线程内的操作会按程序的顺序执行，比如正确同步的多线程程序在临界区内的重排序。

2）顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而JMM不保证所有线程能看到一致的操作执行顺序。

3）顺序一致性模型保证对所有的内存读/写操作都具有原子性，而JMM不保证对64位的long型和double型变量的写操作具有原子性（32位处理器）。

> JVM在32位处理器上运行时，可能会把一个64位long/double型变量的写操作拆分为两个32位的写操作来执行。这两个32位的写操作可能会被分配到不同的总线事务中执行，此时对这个64位变量的写操作将不具有原子性。从JSR-

> 133内存模型开始（即从JDK5开始），仅仅只允许把一个64位long/double型变量的写操作拆分为两个32位的写操作来执行，任意的读操作在JSR-133中都必须具有原子性

volatile的内存语义

volatile的特性

- 可见性：对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。
- 原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性（基于这点，我们通过会认为volatile不具备原子性）。volatile仅仅保证对单个volatile变量的读/写具有原子性，而锁的互斥执行的特性可以确保对整个临界区代码的执行具有原子性。

> 64位的long型和double型变量，只要它是volatile变量，对该变量的读/写就具有原子性。

- 有序性：对volatile修饰的变量的读写操作前后加上各种特定的内存屏障来禁止指令重排序来保障有序性。

> 在JSR-133之前的旧Java内存模型中，虽然不允许volatile变量之间重排序，但旧的Java内存模型允许volatile变量与普通变量重排序。为了提供一种比锁更轻量级的线程之间通信的机制，JSR-133专家组决定增强volatile的内存语义：严格限制编译器和处理器对volatile变量与普通变量的重排序，确保volatile的写-读和锁的释放-获取具有相同的内存语义。

volatile写-读的内存语义

- 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。
- 当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量。

volatile可见性实现原理

JMM内存交互层面实现

volatile修饰的变量的read、load、use操作和assign、store、write必须是连续的，即修改后必须立即同步回主内存，使用时必须从主内存刷新，由此保证volatile变量操作对多线程的可见性。

硬件层面实现

通过lock前缀指令，会锁定变量缓存行区域并写回主内存，这个操作称为“缓存锁定”，缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据。一个处理器的缓存回写到内存会导致其他处理器的缓存无效。

volatile在hotspot的实现

字节码解释器实现

JVM中的字节码解释器(bytecodeInterpreter)，用C++实现了JVM指令，其优点是实现相对简单且容易理解，缺点是执行慢。

bytecodeInterpreter.cpp

```
//
// Now store the result
//
int field_offset = cache->f2_as_index();
if (cache->is_volatile()) {
  if (tos_type == itos) {
    obj->release_int_field_put(field_offset,
  } else if (tos_type == atos) {
    VERIFY_OOP(STACK_OBJECT(-1));
    obj->release_obj_field_put(field_offset,
    OrderAccess::release_store(&BYTE_MAP_BASE
  } else if (tos_type == btos) {
    obj->release_byte_field_put(field_offset
  } else if (tos_type == ltos) {
    obj->release_long_field_put(field_offset
  } else if (tos_type == ctos) {
    obj->release_char_field_put(field_offset
  } else if (tos_type == stos) {
    obj->release_short_field_put(field_offset
  } else if (tos_type == ftos) {
    obj->release_float_field_put(field_offset
  } else {
    obj->release_double_field_put(field_offse
  }
  OrderAccess::storeload();
```

模板解释器实现

模板解释器(templateInterpreter)，其对每个指令都写了一段对应的汇编代码，启动时将每个指令与对应汇编代码入口绑定，可以说是效率做到了极致。

templateTable_x86_64.cpp

```
1  void TemplateTable::volatile_barrier(Assembler::Membar_mask_bits
2   order_constraint) {
3   // Helper function to insert a is-volatile test and memory barrier
4   if (os::is_MP()) { // Not needed on single CPU
5   __ membar(order_constraint);
6   }
7  }
8
9  // 负责执行putfield或putstatic指令
10  void TemplateTable::putfield_or_static(int byte_no, bool is_static, RewriteControl rc)
{
11   // ...
12   // Check for volatile store
13   __ testl(rdx, rdx);
14   __ jcc(Assembler::zero, notVolatile);
15
16   putfield_or_static_helper(byte_no, is_static, rc, obj, off, flags);
17   volatile_barrier(Assembler::Membar_mask_bits(Assembler::StoreLoad |
18   Assembler::StoreStore));
19   __ jmp(Done);
20   __ bind(notVolatile);
```

```
21
22  putfield_or_static_helper(byte_no, is_static, rc, obj, off, flags);
23
24  __ bind(Done);
25  }
26
```

assembler_x86.hpp

```
1  // Serializes memory and blows flags
2  void membar(Membar_mask_bits order_constraint) {
3  // We only have to handle StoreLoad
4  // x86平台只需要处理StoreLoad
5  if (order_constraint & StoreLoad) {
6
7  int offset = -VM_Version::L1_line_size();
8  if (offset < -128) {
9  offset = -128;
10  }
11
12  // 下面这两句插入了一条lock前缀指令: lock addl $0, $0(%rsp)
13  lock(); // lock前缀指令
14  addl(Address(rsp, offset), 0); // addl $0, $0(%rsp)
15  }
16  }
17
```

在linux系统x86中的实现

orderAccess_linux_x86.inline.hpp

```
1  inline void OrderAccess::storeload() { fence(); }
2  inline void OrderAccess::fence() {
3   if (os::is_MP()) {
4   // always use locked addl since mfence is sometimes expensive
5  #ifdef AMD64
6   __asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
7  #else
8   __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
9  #endif
10   }
11  }
```

x86处理器中利用lock实现类似内存屏障的效果。

lock前缀指令的作用

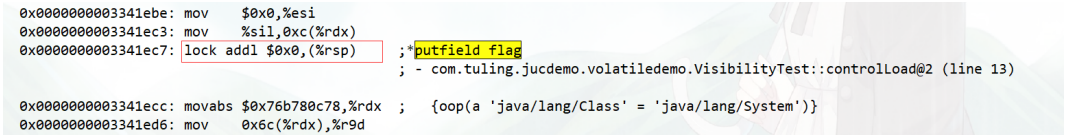
1. 确保后续指令执行的原子性。在Pentium及之前的处理器中，带有lock前缀的指令在执行期间会锁住总线，使得其它处理器暂时无法通过总线访问内存，很显然，这个开销很大。在新的处理器中，Intel使用缓存锁定来保证指令执行的原子性，缓存锁定将大大降低lock前缀指令的执行开销。
2. LOCK前缀指令具有类似于内存屏障的功能，禁止该指令与前面和后面的读写指令重排序。

3. LOCK前缀指令会等待它之前所有的指令完成、并且所有缓冲的写操作写回内存(也就是将store buffer中的内容写入内存)之后才开始执行，并且根据缓存一致性协议，刷新store buffer的操作会导致其他cache中的副本失效。

汇编层面volatile的实现

添加下面的jvm参数查看之前可见性Demo的汇编指令

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



验证了可见性使用了**lock前缀指令**

从硬件层面分析Lock前缀指令

《64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf》中有如下描述：

> The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:
> • Guaranteed atomic operations
> • Bus locking, using the LOCK# signal and the LOCK instruction prefix
> • Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

32位的IA-32处理器支持对系统内存中的位置进行锁定的原子操作。这些操作通常用于管理共享的数据结构(如信号量、段描述符、系统段或页表)，在这些结构中，两个或多个处理器可能同时试图修改相同的字段或标志。处理器使用三种相互依赖的机制来执行锁定的原子操作:

- 有保证的原子操作
- 总线锁定，使用LOCK#信号和LOCK指令前缀
- 缓存一致性协议，确保原子操作可以在缓存的数据结构上执行(缓存锁);这种机制出现在Pentium 4、Intel Xeon和P6系列处理器中

CPU缓存架构剖析

笔记

有序性问题深入分析

思考：下面的Java程序中x和y的最终结果是什么?

```
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;

 /**
 * x,y:
 */
 Thread thread1 = new Thread(new Runnable() {
 @Override
 public void run() {
 shortWait(20000);
 a = 1;
 x = b;

 }
 });
 Thread thread2 = new Thread(new Runnable() {
 @Override
 public void run() {
 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;
 }

 }

 }

 public static void shortWait(long interval){
 long start = System.nanoTime();
 long end;
 do{
```

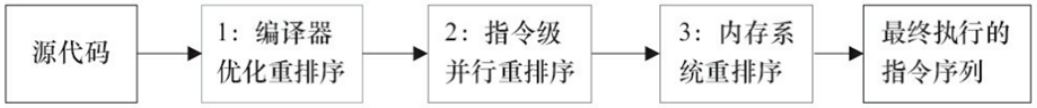
```
55    end = System.nanoTime();
56    }while(start + interval >= end);
57    }
58  }
```

指令重排序

Java语言规范规定JVM线程内部维持顺序化语义。即只要程序的最终结果与它顺序化情况的结果相等，那么指令的执行顺序可以与代码顺序不一致，此过程叫指令的重排序。

指令重排序的意义：JVM能根据处理器特性（CPU多级缓存系统、多核处理器等）适当的对机器指令进行重排序，使机器指令能更符合CPU的执行特性，最大限度的发挥机器性能。

在编译器与CPU处理器中都能执行指令重排优化操作



volatile重排序规则

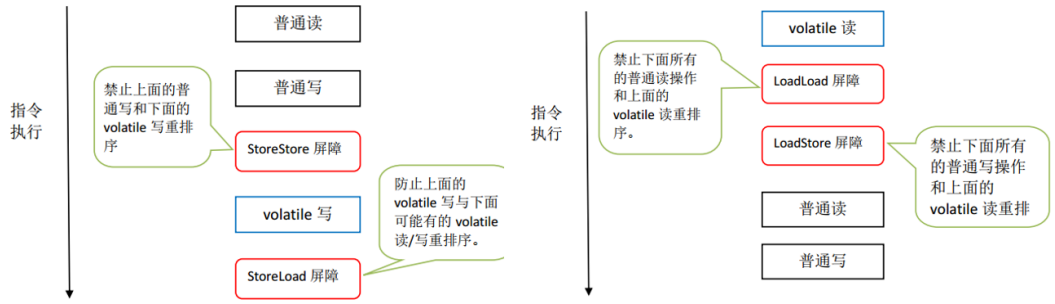
是否能重排序	第二个操作		
第一个操作	普通读/写	volatile 读	volatile 写
普通读/写			NO
volatile 读	NO	NO	NO
volatile 写		NO	NO

volatile禁止重排序场景：

1. 第二个操作是volatile写，不管第一个操作是什么都不会重排序
2. 第一个操作是volatile读，不管第二个操作是什么都不会重排序
3. 第一个操作是volatile写，第二个操作是volatile读，也不会发生重排序

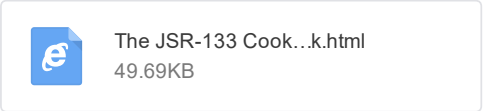
JMM内存屏障插入策略

1. 在每个volatile写操作的前面插入一个StoreStore屏障
2. 在每个volatile写操作的后面插入一个StoreLoad屏障
3. 在每个volatile读操作的后面插入一个LoadLoad屏障
4. 在每个volatile读操作的后面插入一个LoadStore屏障



如何充分压榨硬件性能，压榨CPU计算能力，减少CPU等待时间（机械同感）

JSR133规范



x86处理器不会对读-读、读-写和写-写操作做重排序, 会省略掉这3种操作类型对应的内存屏障。仅会对写-读操作做重排序，所以volatile写-读操作只需要在volatile写后插入StoreLoad屏障

Processor	LoadStore	LoadLoad	StoreStore	StoreLoad	Data dependency orders loads?	Atomic Conditional	Other Atomics	Atomics provide barrier?
sparc-TSO	no-op	no-op	no-op	membar (StoreLoad)	yes	CAS: casa	swap, ldstub	full
x86	no-op	no-op	no-op	mfence or cpuid or locked insn	yes	CAS: cmpxchg	xchg, locked insn	full
ia64	*combine with* st.rel or ld.acq	ld.acq	st.rel	mf	yes	CAS: cmpxchg	xchg, fetchadd	target + acq/rel
arm	dmb (see below)	dmb (see below)	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc	lwsync (see below)	hwsync (see below)	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only
pa-risc	no-op	no-op	no-op	no-op	yes	*build from* ldcw	ldcw	*(NA)*

JVM层面的内存屏障

在JSR规范中定义了4种内存屏障：

LoadLoad屏障：（指令Load1; LoadLoad; Load2），在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。

LoadStore屏障：（指令Load1; LoadStore; Store2），在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。

StoreStore屏障：（指令Store1; StoreStore; Store2），在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。

StoreLoad屏障：（指令Store1; StoreLoad; Load2），在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能

由于x86只有store load可能会重排序，所以只有JSR的StoreLoad屏障对应它的mfence或lock前缀指令，其他屏障对应空操作

硬件层内存屏障

硬件层提供了一系列的内存屏障 memory barrier / memory fence(Intel的提法)来提供一致性的能力。拿X86平台来说，有几种主要的内存屏障：

1. lfence，是一种Load Barrier 读屏障

2. sfence, 是一种Store Barrier 写屏障

3. mfence, 是一种全能型的屏障，具备lfence和sfence的能力

4. Lock前缀，Lock不是一种内存屏障，但是它能完成类似内存屏障的功能。Lock会对CPU总线和高速缓存加锁，可以理解为CPU指令级的一种锁。它后面可以跟ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG等指令。

内存屏障有两个能力：

1. 阻止屏障两边的指令重排序

2. 刷新处理器缓存/冲刷处理器缓存

对Load Barrier来说，在读指令前插入读屏障，可以让高速缓存中的数据失效，重新从主内存加载数据；对Store Barrier来说，在写指令之后插入写屏障，能让写入缓存的最新数据写回到主内存。

Lock前缀实现了类似的能力，它先对总线和缓存加锁，然后执行后面的指令，最后释放锁后会把高速缓存中的数据刷新回主内存。在Lock锁住总线的时候，其他CPU的读写请求都会被阻塞，直到锁释放。

不同硬件实现内存屏障的方式不同，Java内存模型屏蔽了这种底层硬件平台的差异，由JVM来为不同的平台生成相应的机器码。