大纲

用户态与内核态

JDK早期，synchronized 叫做重量级锁， 因为申请锁资源必须通过kernel, 系统调用

```
;hello.asm
 ;write(int fd, const void *buffer, size_t nbytes)

 section data
     msg db "Hello", 0xA
     len equ $ - msg

 section .text
 global _start
 _start:

     mov edx, len
     mov ecx, msg
     mov ebx, 1 ;文件描述符1 std_out
     mov eax, 4 ;write函数系统调用号 4
     int 0x80

     mov ebx, 0
     mov eax, 1 ;exit函数系统调用号
     int 0x80
```

CAS

Compare And Swap (Compare And Exchange) / 自旋 / 自旋锁 / 无锁 （无重量锁）

因为经常配合循环操作，直到完成为止，所以泛指一类操作

cas(v, a, b)，变量v，期待值a, 修改值b

ABA问题，你的女朋友在离开你的这段儿时间经历了别的人，自旋就是你空转等待，一直等到她接纳你为止

解决办法（版本号 AtomicStampedReference），基础类型简单值不需要版本号

Unsafe

AtomicInteger:

```
public final int incrementAndGet() {
         for (;;) {
             int current = get();
             int next = current + 1;
             if (compareAndSet(current, next))
                 return next;
         }
     }

 public final boolean compareAndSet(int expect, int update) {
         return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
     }
```

Unsafe:

```
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
```

运用:

```
package com.mashibing.jol;

 import sun.misc.Unsafe;

 import java.lang.reflect.Field;

 public class T02_TestUnsafe {

     int i = 0;
     private static T02_TestUnsafe t = new T02_TestUnsafe();

     public static void main(String[] args) throws Exception {
         //Unsafe unsafe = Unsafe.getUnsafe();

         Field unsafeField = Unsafe.class.getDeclaredFields()[0];
         unsafeField.setAccessible(true);
         Unsafe unsafe = (Unsafe) unsafeField.get(null);

         Field f = T02_TestUnsafe.class.getDeclaredField("i");
         long offset = unsafe.objectFieldOffset(f);
         System.out.println(offset);

         boolean success = unsafe.compareAndSwapInt(t, offset, 0, 1);
         System.out.println(success);
         System.out.println(t.i);
         //unsafe.compareAndSwapInt()
     }
 }
```

jdk8u: unsafe.cpp:

cmpxchg = compare and exchange

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
   UnsafeWrapper("Unsafe_CompareAndSwapInt");
   oop p = JNIHandles::resolve(obj);
   jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
   return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
 UNSAFE_END
```

jdk8u: atomic_linux_x86.inline.hpp **93行**

is_MP = Multi Processor

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     compare_value) {
   int mp = os::is_MP();
   __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                     : "=a" (exchange_value)
                     : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
                     : "cc", "memory");
   return exchange_value;
 }
```

jdk8u: os.hpp is_MP()

```
static inline bool is_MP() {
     // During bootstrap if _processor_count is not yet initialized
     // we claim to be MP as that is safest. If any platform has a
     // stub generator that might be triggered in this phase and for
     // which being declared MP when in fact not, is a problem - then
     // the bootstrap routine for the stub generator needs to check
     // the processor count directly and leave the bootstrap routine
     // in place until called after initialization has ocurred.
     return (_processor_count != 1) || AssumeMP;
   }
```

jdk8u: atomic_linux_x86.inline.hpp

```
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

最终实现：

cmpxchg = cas修改变量值

```
lock cmpxchg 指令
```

硬件：

lock最终实现比较复杂，可以锁缓存，锁总线（拉高北桥电平信号），都可以

markword

工具：JOL = Java Object Layout

```
<dependencies>
        <!-- https://mvnrepository.com/artifact/org.openjdk.jol/jol-core -->
        <dependency>
            <groupId>org.openjdk.jol</groupId>
            <artifactId>jol-core</artifactId>
            <version>0.9</version>
        </dependency>
    </dependencies>
```

jdk8u: markOop.hpp

```
// Bit-format of an object header (most significant first, big endian layout below):
 //
 //  32 bits:
 //  --------
 //             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
 //             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
 //             size:32 ------------------------------------------>| (CMS free block)
 //             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
 //
 //  64 bits:
 //  --------
 //  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
 //  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
 //  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
 //  size:64 ----------------------------------------------------->| (CMS free block)
 //
 //  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
 //  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
 //  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object)
 //  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
```

synchronized的横切面详解

1. synchronized原理
2. 升级过程
3. 汇编实现
4. vs reentrantLock的区别

java源码层级

synchronized(o)

字节码层级

monitorenter moniterexit

JVM层级（Hotspot）

```
package com.mashibing.insidesync;

 import org.openjdk.jol.info.ClassLayout;

 public class T01_Sync1 {
```

```
    public static void main(String[] args) {
        Object o = new Object();

        System.out.println(ClassLayout.parseInstance(o).toPrintable());
    }
}

com.mashibing.insidesync.T01_Sync1$Lock object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4   (object header)  05 00 00 00 (00000101 00000000 00000000 00000000) (5)
      4     4   (object header)  00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4   (object header)  49 ce 00 20 (01001001 11001110 00000000 00100000) (536923721)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

com.mashibing.insidesync.T02_Sync2$Lock object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4   (object header)  05 90 2e 1e (00000101 10010000 00101110 00011110) (506368005)
      4     4   (object header)  1b 02 00 00 (00011011 00000010 00000000 00000000) (539)
      8     4   (object header)  49 ce 00 20 (01001001 11001110 00000000 00100000) (536923721)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes tota
```

InterpreterRuntime:: monitorenter方法

```
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
  if (PrintBiasedLockingStatistics) {
    Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
  }
  Handle h_obj(thread, elem->obj());
  assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
         "must be NULL or an object");
  if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
  } else {
    ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
  }
  assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
         "must be NULL or an object");
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

synchronizer.cpp

revoke_and_rebias

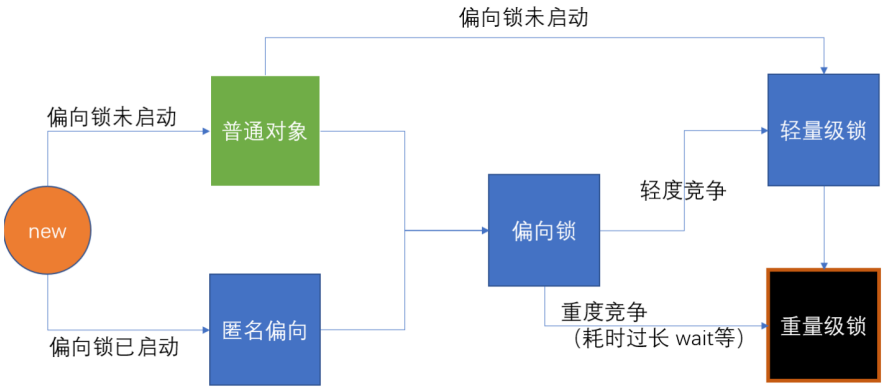
```
void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock, bool attempt_rebias, TRAPS) {
  if (UseBiasedLocking) {
    if (!SafepointSynchronize::is_at_safepoint()) {
      BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREAD);
      if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
        return;
      }
    } else {
      assert(!attempt_rebias, "can not rebias toward VM thread");
      BiasedLocking::revoke_at_safepoint(obj);
    }
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
  }

  slow_enter (obj, lock, THREAD) ;
}

void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
  markOop mark = obj->mark();
  assert(!mark->has_bias_pattern(), "should not see bias pattern here");

  if (mark->is_neutral()) {
    // Anticipate successful CAS -- the ST of the displaced mark must
    // be visible <= the ST performed by the CAS.
    lock->set_displaced_header(mark);
    if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
      TEVENT (slow_enter: release stacklock) ;
      return ;
    }
    // Fall through to inflate() ...
  } else
  if (mark->has_locker() && THREAD->is_lock_owned((address)mark->locker())) {
    assert(lock != mark->locker(), "must not re-lock the same lock");
    assert(lock != (BasicLock*)obj->mark(), "don't relock with same BasicLock");
    lock->set_displaced_header(NULL);
    return;
  }

#if 0
  // The following optimization isn't particularly useful.
  if (mark->has_monitor() && mark->monitor()->is_entered(THREAD)) {
    lock->set_displaced_header (NULL) ;
    return ;
  }
#endif

  // The object header will never be displaced to this lock,
  // so it does not matter what the value is, except that it
  // must be non-zero to avoid looking like a re-entrant lock,
  // and must not look locked either.
  lock->set_displaced_header(markOopDesc::unused_mark());
```

```
    ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}
```

inflate方法：膨胀为重量级锁

锁升级过程

JDK8 markword实现表：



Hotspot的实现

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
无锁态（new）	unused	hashCode（如果有调用）	unused	分代年龄	0	0	1

锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0	1

锁状态	62位	2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针	0	0
重量级锁	指向互斥量（重量级锁）的指针	1	0
GC标记信息	CMS过程用到的标记信息	1	1

自旋锁什么时候升级为重量级锁?

为什么有自旋锁还需要重量级锁?

> 自旋是消耗CPU资源的，如果锁的时间长，或者自旋线程多，CPU会被大量消耗
>
> 重量级锁有等待队列，所有拿不到锁的进入等待队列，不需要消耗CPU资源

偏向锁是否一定比自旋锁效率高?

> 不一定，在明确知道会有多线程竞争的情况下，偏向锁肯定会涉及锁撤销，这时候直接使用自旋锁
>
> JVM启动过程，会有很多线程竞争（明确），所以默认情况启动时不打开偏向锁，过一段儿时间再打开

new - 偏向锁 - 轻量级锁 （无锁, 自旋锁，自适应自旋）- 重量级锁

synchronized优化的过程和markword息息相关

用markword中最低的三位代表锁状态 其中1位是偏向锁位 两位是普通锁位

1. Object o = new Object()
   锁 = 0 01 无锁态
   注意：如果偏向锁打开，默认是匿名偏向状态
2. o.hashCode()
   001 + hashcode

```
00000001 10101101 00110100 00110110
 01011001 00000000 00000000 00000000
```

   little endian big endian

   00000000 00000000 00000000 01011001 00110110 00110100 10101101 00000000
3. 默认synchronized(o)
   00 -> 轻量级锁
   默认情况 偏向锁有个时延，默认是4秒
   why? 因为JVM虚拟机自己有一些默认启动的线程，里面有好多sync代码，这些sync代码启动时就知道肯定会有竞争，如果使用偏向锁，就会造成偏向锁不断的进行锁撤销和锁升级的操作，效率较低。

```
-XX:BiasedLockingStartupDelay=0
```

4. 如果设定上述参数
   new Object () - > 101 偏向锁 ->线程ID为0 -> Anonymous BiasedLock
   打开偏向锁，new出来的对象，默认就是一个可偏向匿名对象101
5. 如果有线程上锁
   上偏向锁，指的就是，把markword的线程ID改为自己线程ID的过程
   偏向锁不可重偏向 批量偏向 批量撤销

6. 如果有线程竞争
   撤销偏向锁，升级轻量级锁
   线程在自己的线程栈生成LockRecord ，用CAS操作将markword设置为指向自己这个线程的LR的指针，设置成功者得到锁
7. 如果竞争加剧
   竞争加剧：有线程超过10次自旋， -XX:PreBlockSpin， 或者自旋线程数超过CPU核数的一半， 1.6之后，加入自适应自旋 Adaptive Self Spinning ， JVM自己控制
   升级重量级锁：-> 向操作系统申请资源，linux mutex , CPU从3级-0级系统调用，线程挂起，进入等待队列，等待操作系统的调度，然后再映射回用户空间

(以上实验环境是JDK11，打开就是偏向锁，而JDK8默认对象头是无锁)

偏向锁默认是打开的，但是有一个时延，如果要观察到偏向锁，应该设定参数

如果计算过对象的hashCode，则对象无法进入偏向状态！

> 轻量级锁重量级锁的hashCode存在与什么地方？
>
> 答案：线程栈中，轻量级锁的LR中，或是代表重量级锁的ObjectMonitor的成员中

关于epoch: (不重要)

> **批量重偏向与批量撤销**渊源：从偏向锁的加锁解锁过程中可看出，当只有一个线程反复进入同步块时，偏向锁带来的性能开销基本可以忽略，但是当有其他线程尝试获得锁时，就需要等到safe point时，再将偏向锁撤销为无锁状态或升级为轻量级，会消耗一定的性能，所以在多线程竞争频繁的情况下，偏向锁不仅不能提高性能，还会导致性能下降。于是，就有了批量重偏向与批量撤销的机制。
>
> **原理**以class为单位，为每个class维护**解决场景**批量重偏向（bulk rebias）机制是为了解决：一个线程创建了大量对象并执行了初始的同步操作，后来另一个线程也来将这些对象作为锁对象进行操作，这样会导致大量的偏向锁撤销操作。批量撤销（bulk revoke）机制是为了解决：在明显多线程竞争剧烈的场景下使用偏向锁是不合适的。
>
> 一个偏向锁撤销计数器，每一次该class的对象发生偏向撤销操作时，该计数器+1，当这个值达到重偏向阈值（默认20）时，JVM就认为该class的偏向锁有问题，因此会进行批量重偏向。每个class对象会有一个对应的epoch字段，每个处于偏向锁状态对象的Mark Word中也有该字段，其初始值为创建该对象时class中的epoch的值。每次发生批量重偏向时，就将该值+1，同时遍历JVM中所有线程的栈，找到该class所有正处于加锁状态的偏向锁，将其epoch字段改为新值。下次获得锁时，发现当前对象的epoch值和class的epoch不相等，那就算当前已经偏向了其他线程，也不会执行撤销操作，而是直接通过CAS操作将其Mark Word的Thread Id 改成当前线程Id。当达到重偏向阈值后，假设该class计数器继续增长，当其达到批量撤销的阈值后（默认40），JVM就认为该class的使用场景存在多线程竞争，会标记该class为不可偏向，之后，对于该class的锁，直接走轻量级锁的逻辑。

没错，我就是厕所所长

加锁，指的是锁定对象

锁升级的过程

JDK较早的版本 OS的资源 互斥量 用户态 -> 内核态的转换 重量级 效率比较低

现代版本进行了优化

无锁 - 偏向锁 -轻量级锁（自旋锁）-重量级锁

偏向锁 - markword 上记录当前线程指针，下次同一个线程加锁的时候，不需要争用，只需要判断线程指针是否同一个，所以，偏向锁，偏向加锁的第一个线程 。hashCode备份在线程栈上 线程销毁，锁降级为无锁

有争用 - 锁升级为轻量级锁 - 每个线程有自己的LockRecord在自己的线程栈上，用CAS去争用markword的LR的指针，指针指向哪个线程的LR，哪个线程就拥有锁

自旋超过10次，升级为重量级锁 - 如果太多线程自旋 CPU消耗过大，不如升级为重量级锁，进入等待队列（不消耗CPU）-XX:PreBlockSpin

自旋锁在 JDK1.4.2 中引入，使用 -XX:+UseSpinning 来开启。JDK 6 中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）。

自适应自旋锁意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

偏向锁由于有锁撤销的过程revoke，会消耗系统资源，所以，在锁争用特别激烈的时候，用偏向锁未必效率高。还不如直接使用轻量级锁。

锁重入

sychronized是可重入锁

重入次数必须记录，因为要解锁几次必须得对应

偏向锁 自旋锁 -> 线程栈 -> LR + 1

重量级锁 -> ? ObjectMonitor字段上

synchronized最底层实现

```
public class T {
     static volatile int i = 0;

     public static void n() { i++; }

     public static synchronized void m() {}

     publics static void main(String[] args) {
         for(int j=0; j<1000_000; j++) {
             m();
             n();
         }
     }
 }
```

java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly T

C1 Compile Level 1 (一级优化)

C2 Compile Level 2 (二级优化)

找到m() n()方法的汇编码，会看到 lock comxchg .....指令

synchronized vs Lock (CAS)

```
在高争用 高耗时的环境下synchronized效率更高
  在低争用 低耗时的环境下CAS效率更高
  synchronized到重量级之后是等待队列（不消耗CPU）
  CAS（等待期间消耗CPU）

  一切以实测为准
```

锁消除 lock eliminate

```
public void add(String str1,String str2){
         StringBuffer sb = new StringBuffer();
         sb.append(str1).append(str2);
 }
```

我们都知道 StringBuffer 是线程安全的，因为它的关键方法都是被 synchronized 修饰过的，但我们看上面这段代码，我们会发现，sb 这个引用只会在 add 方法中使用，不可能被其它线程引用（因为是局部变量，栈私有），因此 sb 是不可能共享的资源，JVM 会自动消除 StringBuffer 对象内部的锁。

锁粗化 lock coarsening

```
public String test(String str){

        int i = 0;
        StringBuffer sb = new StringBuffer():
        while(i < 100){
            sb.append(str);
            i++;
        }
        return sb.toString():
 }
```

JVM 会检测到这样一连串的操作都对同一个对象加锁（while 循环内 100 次执行 append，没有锁粗化的就要进行 100 次加锁/解锁），此时 JVM 就会将加锁的范围粗化到这一连串的操作的外部（比如 while 虚幻体外），使得这一连串操作只需要加一次锁即可。

锁降级（不重要）

https://www.zhihu.com/question/63859501

其实，只被VMThread访问，降级也就没啥意义了。所以可以简单认为锁降级不存在!

超线程

一个ALU + 两组Registers + PC

参考资料

http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

volatile的用途

1.线程可见性

```
package com.mashibing.testvolatile;

 public class T01_ThreadVisibility {
     private static volatile boolean flag = true;

     public static void main(String[] args) throws InterruptedException {
         new Thread(()-> {
             while (flag) {
                 //do sth
             }
             System.out.println("end");
         }, "server").start();


         Thread.sleep(1000);

         flag = false;
     }
 }
```

2.防止指令重排序

问题：DCL单例需不需要加volatile?

CPU的基础知识

- 缓存行对齐
  缓存行64个字节是CPU同步的基本单位，缓存行隔离会比伪共享效率要高
  Disruptor

```
package com.mashibing.juc.c_028_FalseSharing;

 public class T02_CacheLinePadding {
     private static class Padding {
         public volatile long p1, p2, p3, p4, p5, p6, p7; //
```

```
    }

    private static class T extends Padding {
        public volatile long x = 0L;
    }

    public static T[] arr = new T[2];

    static {
        arr[0] = new T();
        arr[1] = new T();
    }

    public static void main(String[] args) throws Exception {
        Thread t1 = new Thread(()->{
            for (long i = 0; i < 1000_0000L; i++) {
                arr[0].x = i;
            }
        });

        Thread t2 = new Thread(()->{
            for (long i = 0; i < 1000_0000L; i++) {
                arr[1].x = i;
            }
        });

        final long start = System.nanoTime();
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println((System.nanoTime() - start)/100_0000);
    }
}
```

MESI

- 伪共享
- 合并写
  CPU内部的4个字节的Buffer

```
package com.mashibing.juc.c_029_WriteCombining;

public final class WriteCombining {

    private static final int ITERATIONS = Integer.MAX_VALUE;
    private static final int ITEMS = 1 << 24;
    private static final int MASK = ITEMS - 1;

    private static final byte[] arrayA = new byte[ITEMS];
    private static final byte[] arrayB = new byte[ITEMS];
    private static final byte[] arrayC = new byte[ITEMS];
    private static final byte[] arrayD = new byte[ITEMS];
    private static final byte[] arrayE = new byte[ITEMS];
    private static final byte[] arrayF = new byte[ITEMS];

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

        for (int i = 1; i <= 3; i++) {
            System.out.println(i + " SingleLoop duration (ns) = " + runCaseOne());
            System.out.println(i + " SplitLoop  duration (ns) = " + runCaseTwo());
        }
    }

    public static long runCaseOne() {
        long start = System.nanoTime();
        int i = ITERATIONS;

        while (--i != 0) {
            int slot = i & MASK;
            byte b = (byte) i;
            arrayA[slot] = b;
            arrayB[slot] = b;
            arrayC[slot] = b;
            arrayD[slot] = b;
            arrayE[slot] = b;
            arrayF[slot] = b;
        }
        return System.nanoTime() - start;
    }

    public static long runCaseTwo() {
        long start = System.nanoTime();
        int i = ITERATIONS;
        while (--i != 0) {
            int slot = i & MASK;
            byte b = (byte) i;
            arrayA[slot] = b;
            arrayB[slot] = b;
            arrayC[slot] = b;
```

- 指令重排序

```
package com.mashibing.jvm.c3_jmm;

public class T04_Disorder {
    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;
        for(;;) {
            i++;
            x = 0; y = 0;
            a = 0; b = 0;
            Thread one = new Thread(new Runnable() {
                public void run() {
                    //由于线程one先启动，下面这句话让它等一等线程two. 读着可根据自己电脑的实际性能适当调整等待时间.
```

```
                    //shortWait(100000);
                    a = 1;
                    x = b;
                }
            });

            Thread other = new Thread(new Runnable() {
                public void run() {
                    b = 1;
                    y = a;
                }
            });
            one.start();other.start();
            one.join();other.join();
            String result = "第" + i + "次 (" + x + "," + y + "）";
            if(x == 0 && y == 0) {
                System.err.println(result);
                break;
            } else {
                //System.out.println(result);
            }
        }
    }


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

系统底层如何实现数据一致性

1. MESI如果能解决，就使用MESI
2. 如果不能，就锁总线

系统底层如何保证有序性

1. 内存屏障sfence mfence lfence等系统原语
2. 锁总线

volatile如何解决指令重排序

1: volatile i

2: ACC_VOLATILE

3: JVM的内存屏障

```
屏障两边的指令不可以重排！保障有序！
```

4：hotspot实现

bytecodeinterpreter.cpp

```
int field_offset = cache->f2_as_index();
          if (cache->is_volatile()) {
            if (support_IRIW_for_not_multiple_copy_atomic_cpu) {
              OrderAccess::fence();
            }
```

orderaccess_linux_x86.inline.hpp

```
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
   }
 }
```

用hsdis观察synchronized和volatile

1. 安装hsdis (自行百度)
2. 代码

```
public class T {

   public static volatile int i = 0;

   public static void main(String[] args) {
     for(int i=0; i<1000000; i++) {
        m();
        n();
     }
   }

   public static synchronized void m() {

   }

   public static void n() {
     i = 1;
   }
 }
```

3. `java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly T > 1.txt`

输出结果

由于JIT会为所有代码生成汇编，请搜索T::m T::n，来找到m() 和 n()方法的汇编码

```
============================= C1-compiled nmethod ==============================
 ----------------------------------- Assembly -----------------------------------

Compiled method (c1)      67    1       3       java.lang.Object::<init> (1 bytes)
 total in heap  [0x00007f81d4d33010,0x00007f81d4d33360] = 848
 relocation     [0x00007f81d4d33170,0x00007f81d4d33198] = 40
 main code      [0x00007f81d4d331a0,0x00007f81d4d33260] = 192
 stub code      [0x00007f81d4d33260,0x00007f81d4d332f0] = 144
 metadata       [0x00007f81d4d332f0,0x00007f81d4d33300] = 16
 scopes data    [0x00007f81d4d33300,0x00007f81d4d33318] = 24
 scopes pcs     [0x00007f81d4d33318,0x00007f81d4d33358] = 64
 dependencies   [0x00007f81d4d33358,0x00007f81d4d33360] = 8

 --------------------------------------------------------------------------------
 [Constant Pool (empty)]

 --------------------------------------------------------------------------------

 [Entry Point]
   # {method} {0x00007f81d3cfe650} '<init>' '()V' in 'java/lang/Object'
   #           [sp+0x40]  (sp of caller)
   0x00007f81d4d331a0:   mov    0x8(%rsi),%r10d
   0x00007f81d4d331a4:   shl    $0x3,%r10
   0x00007f81d4d331a8:   cmp    %rax,%r10
   0x00007f81d4d331ab:   jne    0x00007f81d47eed00           ;   {runtime_call ic_miss_stub}
   0x00007f81d4d331b1:   data16 data16 nopw 0x0(%rax,%rax,1)
   0x00007f81d4d331bc:   data16 data16 xchg %ax,%ax
 [Verified Entry Point]
   0x00007f81d4d331c0:   mov    %eax,-0x14000(%rsp)
   0x00007f81d4d331c7:   push   %rbp
   0x00007f81d4d331c8:   sub    $0x30,%rsp
   0x00007f81d4d331cc:   movabs $0x7f81d3f33388,%rdi         ;   {metadata(method data for {method} {0x00007f81d3cfe650}
'<init>' '()V' in 'java/lang/Object')}
   0x00007f81d4d331d6:   mov    0x13c(%rdi),%ebx
   0x00007f81d4d331dc:   add    $0x8,%ebx
   0x00007f81d4d331df:   mov    %ebx,0x13c(%rdi)
   0x00007f81d4d331e5:   and    $0x1ff8,%ebx
   0x00007f81d4d331eb:   cmp    $0x0,%ebx
   0x00007f81d4d331ee:   je     0x00007f81d4d33204           ;*return {reexecute=0 rethrow=0 return_oop=0}
                                                            ; - java.lang.Object::<init>@0 (line 50)
   0x00007f81d4d331f4:   add    $0x30,%rsp
   0x00007f81d4d331f8:   pop    %rbp
   0x00007f81d4d331f9:   mov    0x108(%r15),%r10
   0x00007f81d4d33200:   test   %eax,(%r10)                 ;   {poll_return}
   0x00007f81d4d33203:   retq
   0x00007f81d4d33204:   movabs $0x7f81d3cfe650,%r10         ;   {metadata({method} {0x00007f81d3cfe650} '<init>' '()V' in
'java/lang/Object')}
   0x00007f81d4d3320e:   mov    %r10,0x8(%rsp)
   0x00007f81d4d33213:   movq   $0xffffffffffffffff,(%rsp)
```