龙果学院
roncoo.com
2016年 版教程

Java + 您
全栈修养

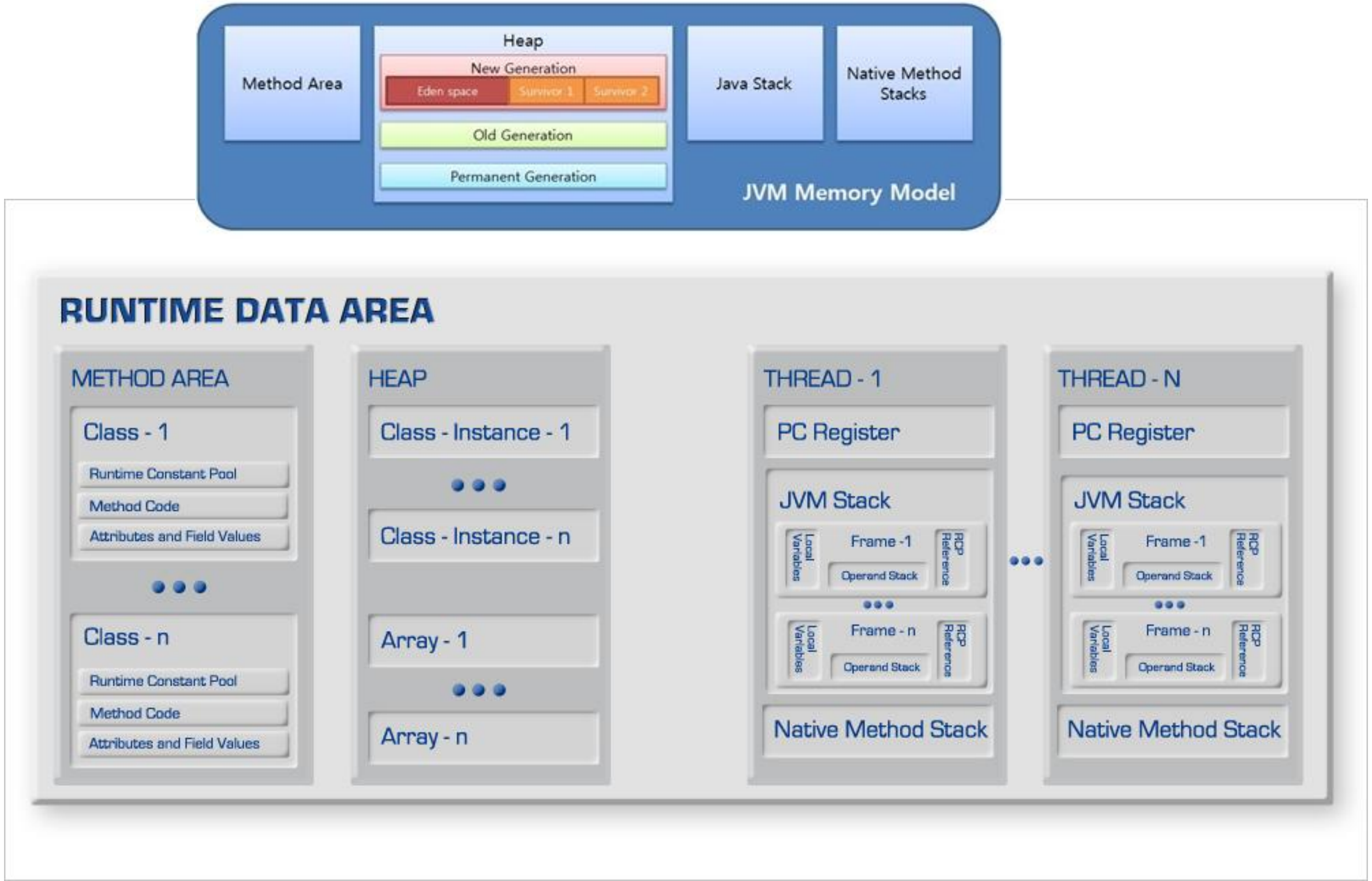
技术与自我营销两手抓，做一个有追求的程序猿

Leader.us@Mycat

本节目录

Java内存模型

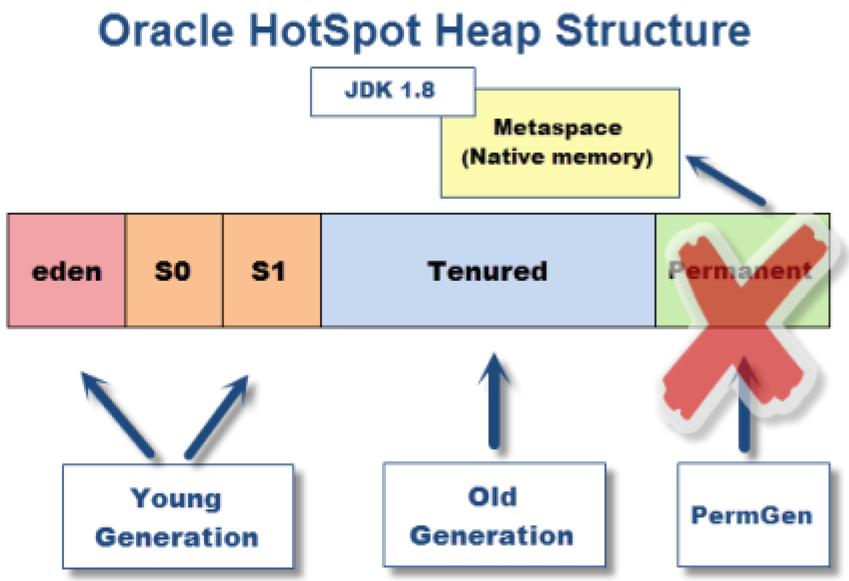
Java内存模型


Method Area
Heap
New Generation
Eden space
Survivor 1
Survivor 2
Old Generation
Permanent Generation
Java Stack
Native Method Stacks
JVM Memory Model
RUNTIME DATA AREA
METHOD AREA
Class - 1
Runtime Constant Pool
Method Code
Attributes and Field Values
Class - n
Runtime Constant Pool
Method Code
Attributes and Field Values
HEAP
Class - Instance - 1
Class - Instance - n
Array - 1
Array - n
THREAD - 1
PC Register
JVM Stack
Local Variables
Frame -1
Operand Stack
RCP Reference
Local Variables
Frame - n
Operand Stack
RCP Reference
Native Method Stack
THREAD - N
PC Register
JVM Stack
Local Variables
Frame -1
Operand Stack
RCP Reference
Local Variables
Frame - n
Operand Stack
RCP Reference
Native Method Stack

Java内存模型

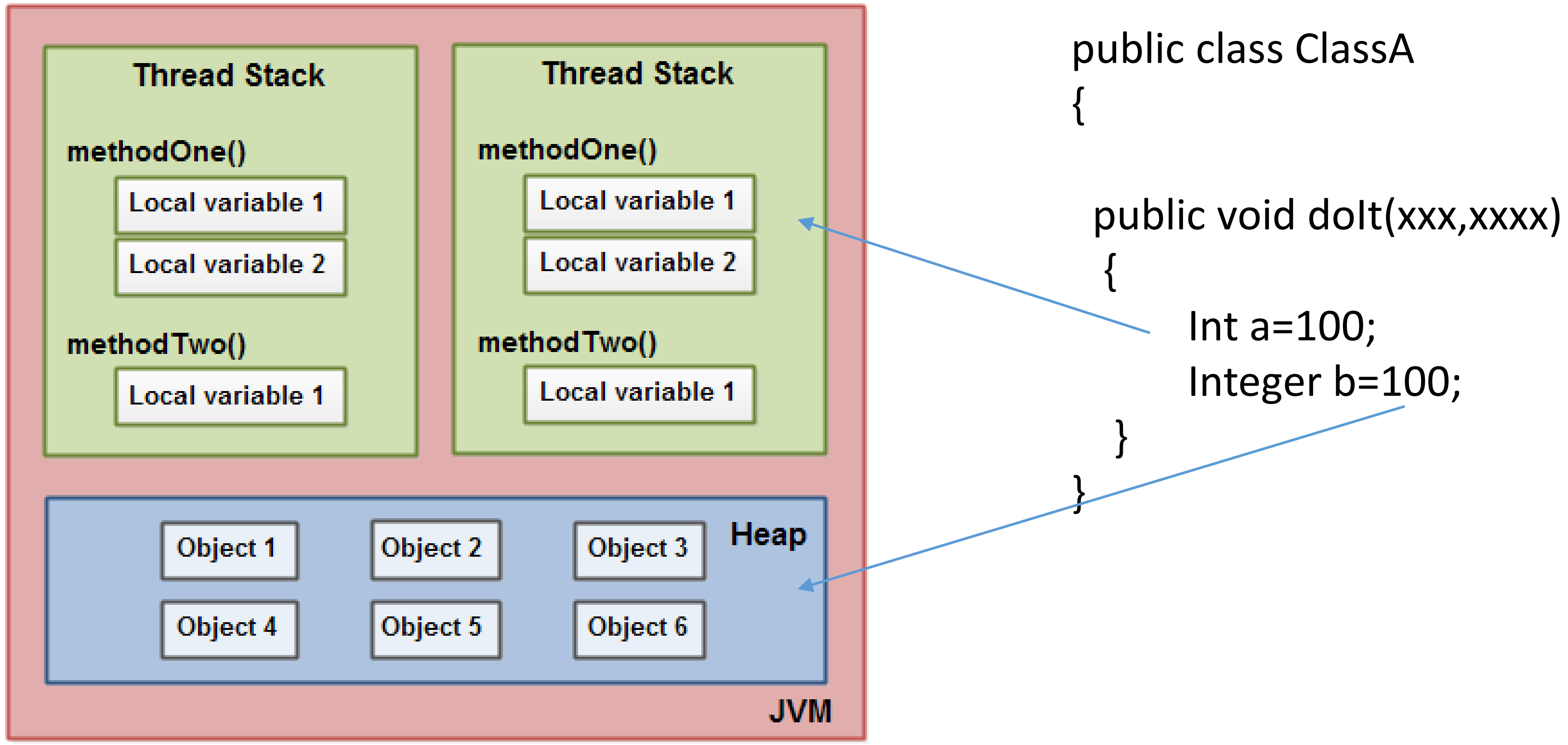
The JDK 1.8 HotSpot JVM is now using native memory for the representation of class metadata and is called **Metaspace**. It is important to realize that this implementation approach is not that new. Both JRockit and IBM JVM's have been using the native memory to store the class metadata since several years now. The HotSpot JRE 1.8 is essentially an union of the best features of JRockit and HotSpot following Oracle's decision to converge these two implementations to a single project.

-XX:MetaspaceSize-XX:MaxMetaspaceSize



Java内存模型

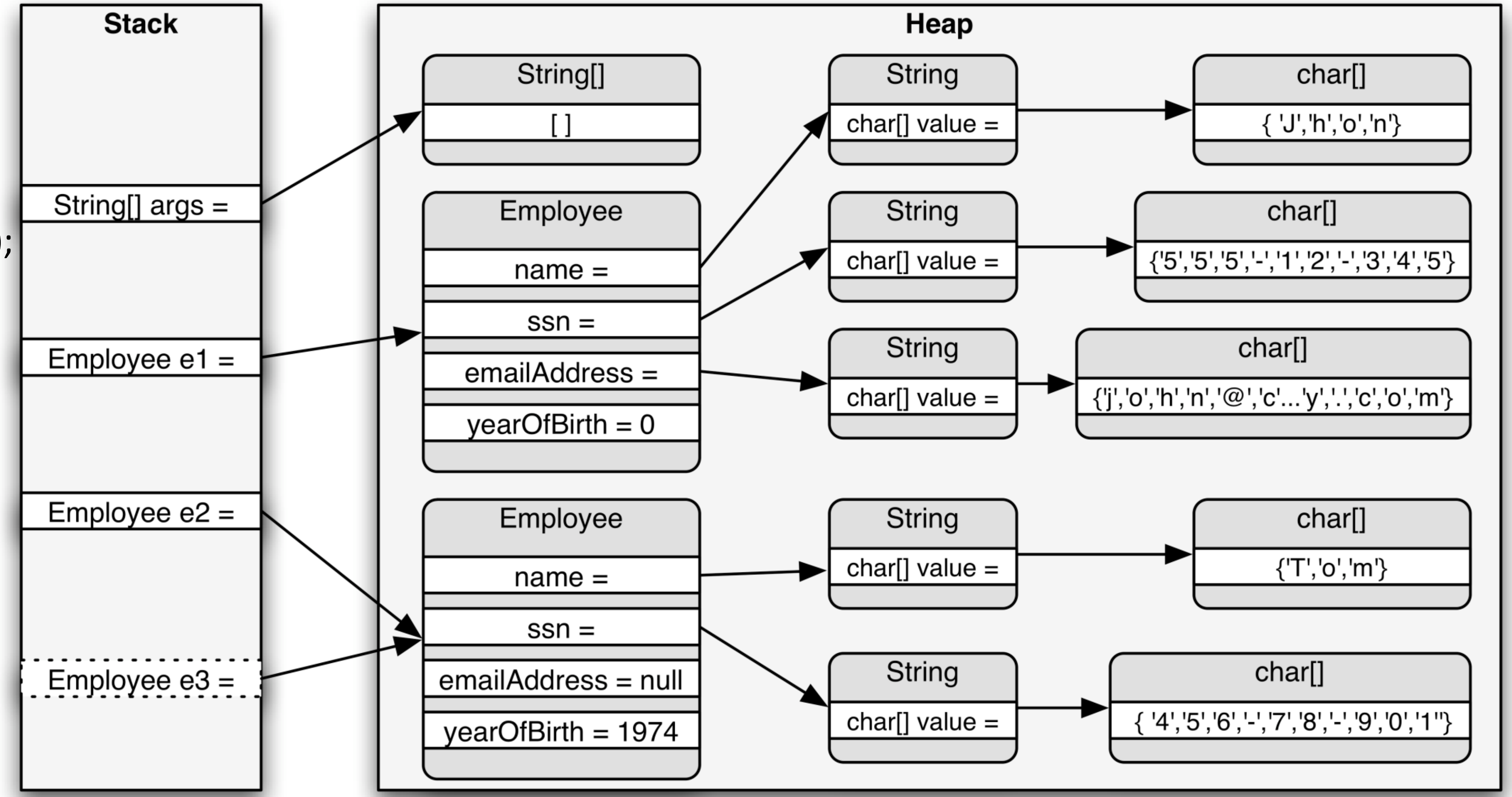
龙果学院 roncoo.com



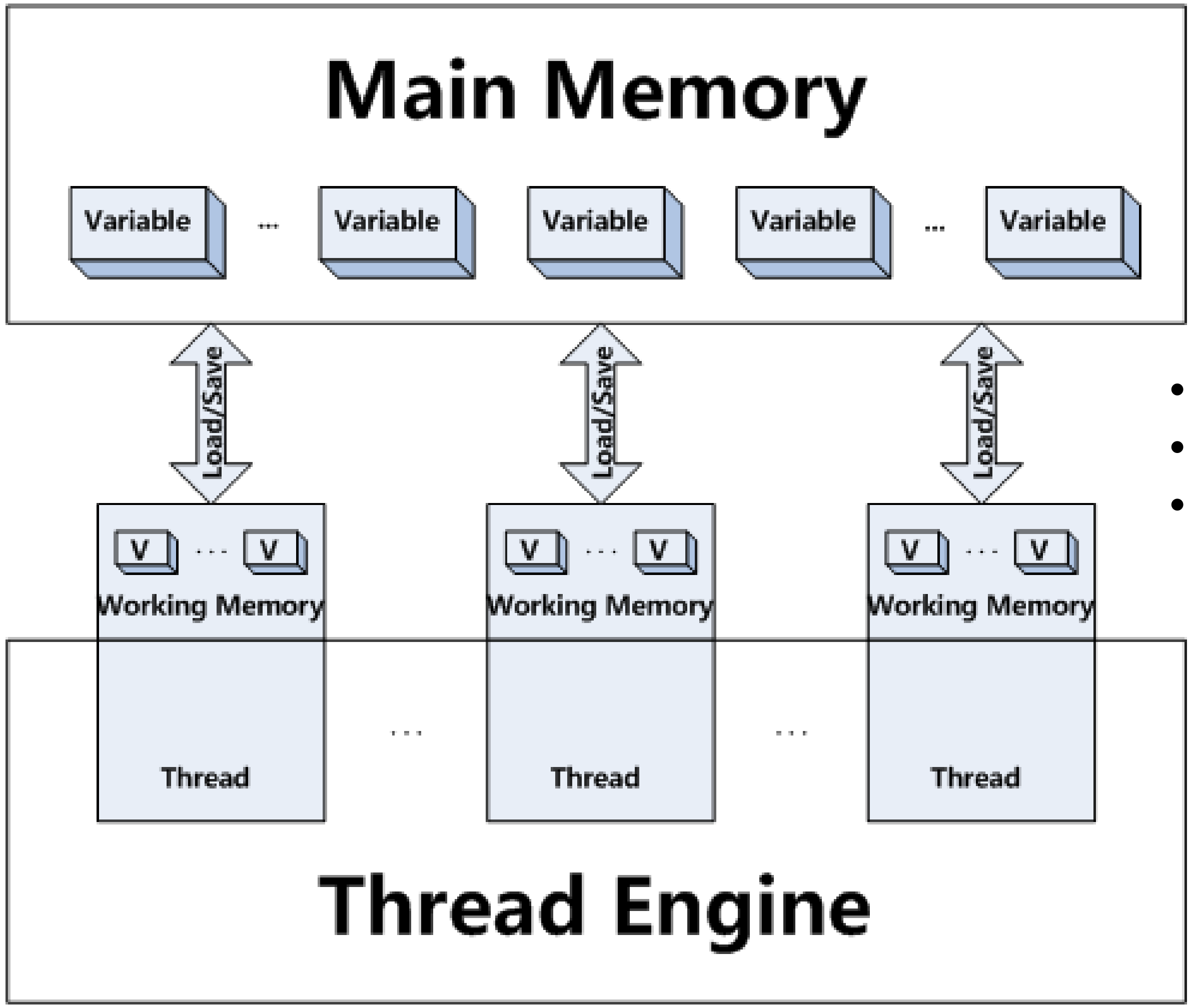
```
public class ClassA
{

  public void doIt(xxx,xxxx)
   {
         Int a=100;
         Integer b=100;
    }
}
```

Java内存模型

```
String[] args={};
Employee e1=new Employee();
e1.name="jhon";
e1.ssn="555-12-345";
el.emailAddress=xxxx;
....
Employee  e2=new ....
Employee  e3=e2;
```



Java内存模型

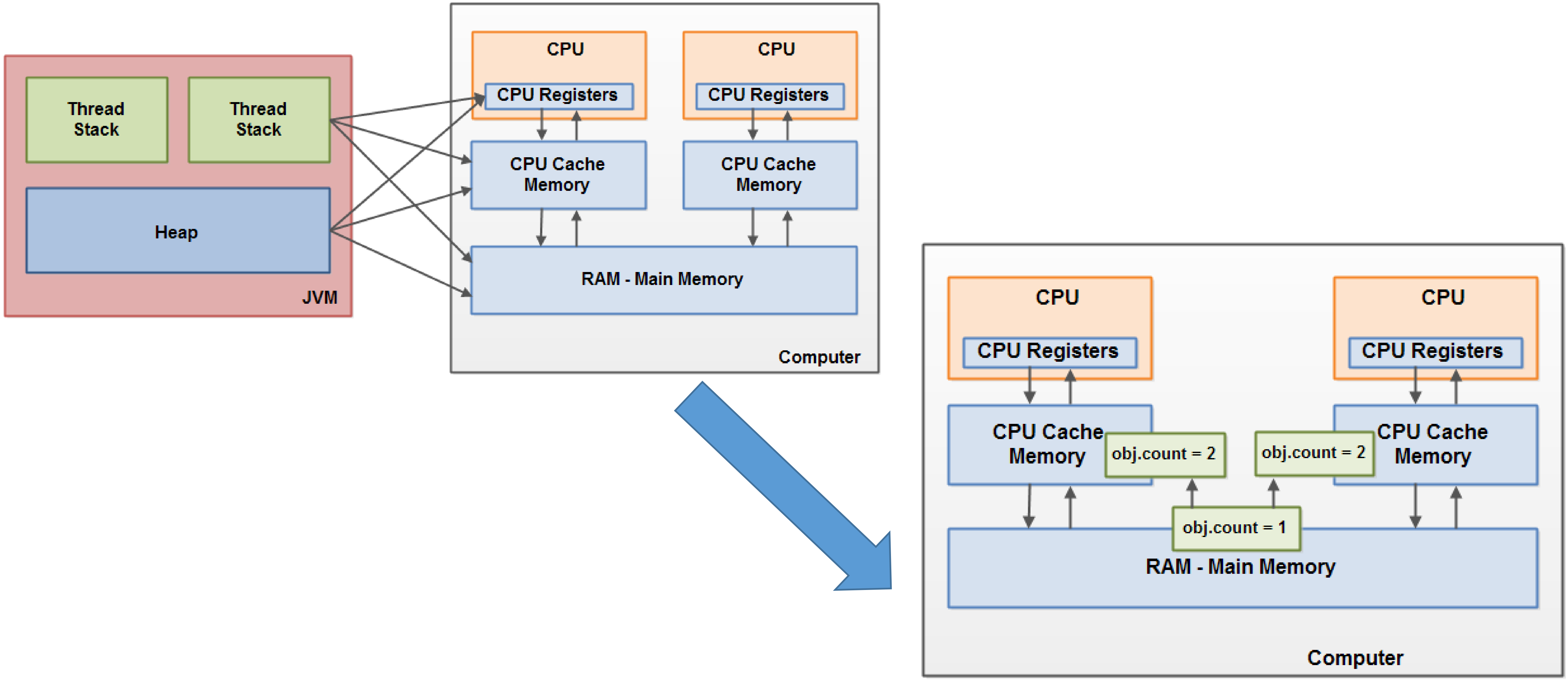


- 线程运行时有自己的“独立”工作内存区
- 线程工作内存区与主存之间存在同步问题
- 默认情况下，JVM并不要求每个变量在任意时刻都保持同步

多线程内存可见性问题

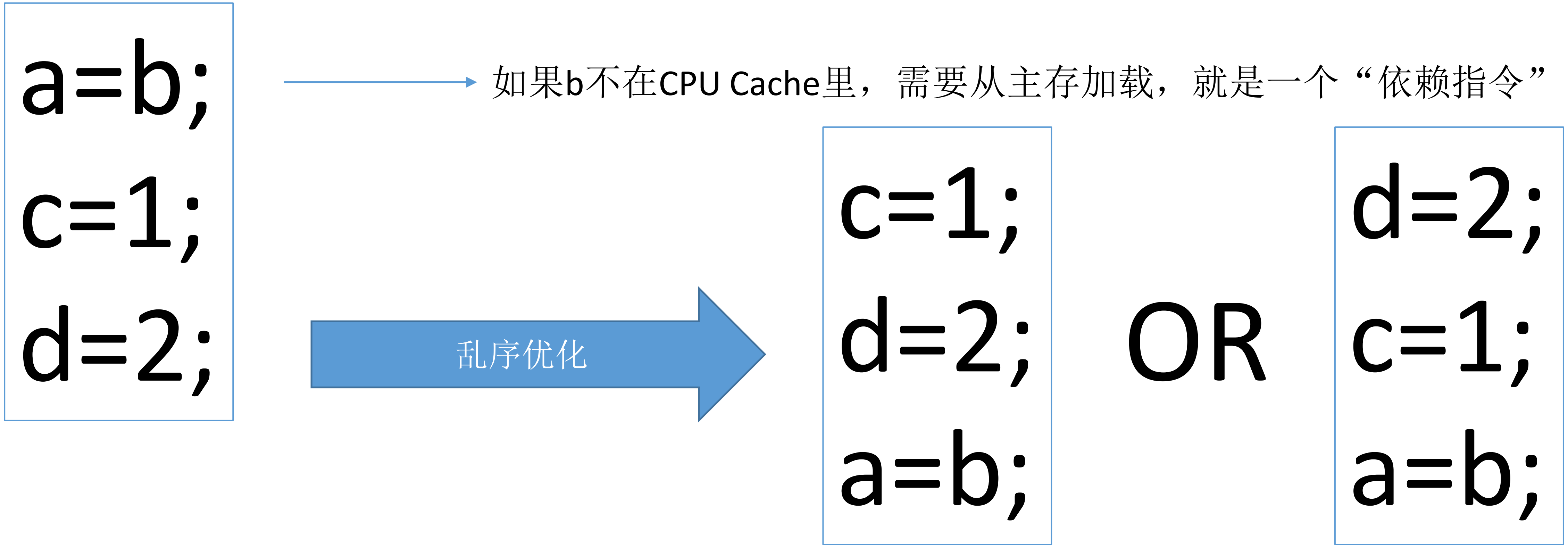
Java内存模型

龙果学院 roncoo.com



CPU乱序执行问题

在按序执行中，一旦遇到**指令依赖**的情况，流水线就会停滞，如果采用乱序执行，就可以跳到下一个**非依赖指令**并发布它。这样，执行单元就可以总是处于工作状态，把时间浪费减到最少。



```
a=b;
c=1;
d=2;
```

如果b不在CPU Cache里，需要从主存加载，就是一个“依赖指令”

```
c=1;
d=2;
a=b;
```

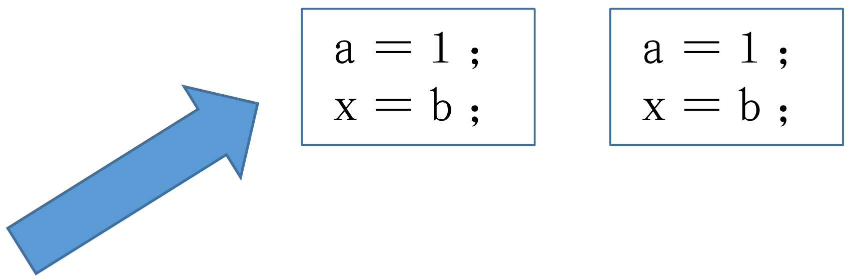
```
d=2;
c=1;
a=b;
```

Java内存模型

龙果学院 roncoo.com

CPU乱序功验证测试

```
import java.util.stream.IntStream;
//PossibleReordering recorder
public class PossibleReordering {
 int x = 0, y = 0;
 int a = 0, b = 0;
public static void doTest(int i)
{
PossibleReordering ordering=new PossibleReordering();
  Thread one = new Thread(()->{ordering.a = 1;ordering.x =
ordering.b;});
  Thread other = new Thread(()->{ordering.b = 1;ordering.y =
ordering.a;});
    one.start();other.start();
   try {
   one.join();
other.join();
} catch (InterruptedException e) {
}
   System.out.println(" run case :"+i+"(" + ordering.x + "," +
ordering.y + ")");
}
public static void main(String[] args) throws InterruptedException {
    IntStream.range(0, 100).forEach(PossibleReordering::doTest);
}
}
```

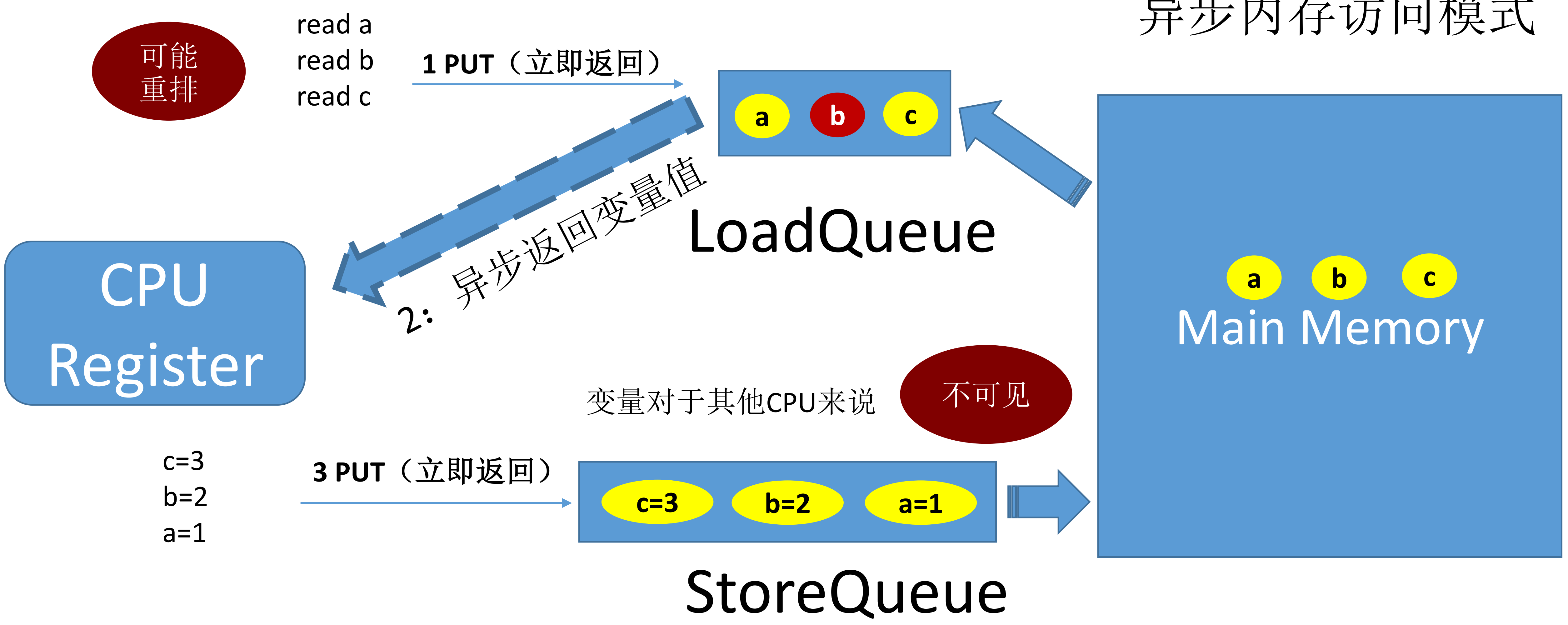


```
run case :0(1,0)
run case :1(0,1)
run case :2(0,1)
run case :3(0,1)
run case :4(0,1)
run case :5(1,0)
run case :6(0,1)
run case :7(0,1)
```

Java内存模型

CPU的内存机制

异步内存访问模式



可见：乱序+内存可见性问题不是Java的问题

Java内存模型

CPU的内存机制

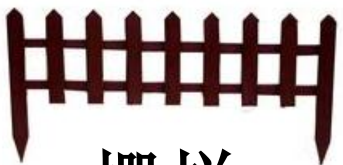
So there are two things at play here:

- "**immediacy**" of the store being visible to other CPUs
- "**order**" " in which stores (loads too, but let's focus on stores) appear to other CPUs.

Immediacy means: can the CPU proceed to next instruction (this is simplified view because out of order/speculation makes this more complex) before the store is globally visible.

Ordering means: when the stores *are* made visible (not necessarily before next instruction executes) they need to appear in the specified order.

The machinery that makes "delayed" stores manifest itself is the store buffer in the CPU. Writes can go there first, and not be drained to the coherent caches**; if a pending store is sitting in the store buffer, no other CPU can see it, but the CPU that issued the write *can* see it. This comes into play with store-forwarding: a CPU can read out its own store from the buffer even though it's not globally visible.** Why can this be a problem? Suppose you're writing a basic mutex. Some CPU releases it by doing a simple store with no full fence. **The store sits in the buffer; all other CPUs still think mutex is locked**. The same CPU tries to acquire the lock again before the store makes it out of the buffer. Since it observes it as free, it acquires the lock again. This is an unfair advantage.



解决乱序+内存可见性 fences =栅栏

之终极武器

期待神作：《哈利波特之扫帚遇上栅栏》

Java内存模型

Store/Load指令

LoadLoad Barriers

The sequence: Load1; LoadLoad; Load2

ensures that Load1's data are loaded before data accessed by Load2 and all subsequent load instructions are loaded. In general, explicit LoadLoad barriers are needed on processors that perform speculative loads and/or out-of-order processing in which waiting load instructions can bypass waiting stores. On processors that guarantee to always preserve load ordering, the barriers amount to no-ops.

StoreStore Barriers

The sequence: Store1; StoreStore; Store2

ensures that Store1's data are visible to other processors (i.e., flushed to memory) before the data associated with Store2 and all subsequent store instructions. In general, StoreStore barriers are needed on processors that do not otherwise guarantee strict ordering of flushes from write buffers and/or caches to other processors or main memory.

LoadStore Barriers

The sequence: Load1; LoadStore; Store2

ensures that Load1's data are loaded before all data associated with Store2 and subsequent store instructions are flushed. LoadStore barriers are needed only on those out-of-order procesors in which waiting store instructions can bypass loads.

StoreLoad Barriers

The sequence: Store1; StoreLoad; Load2

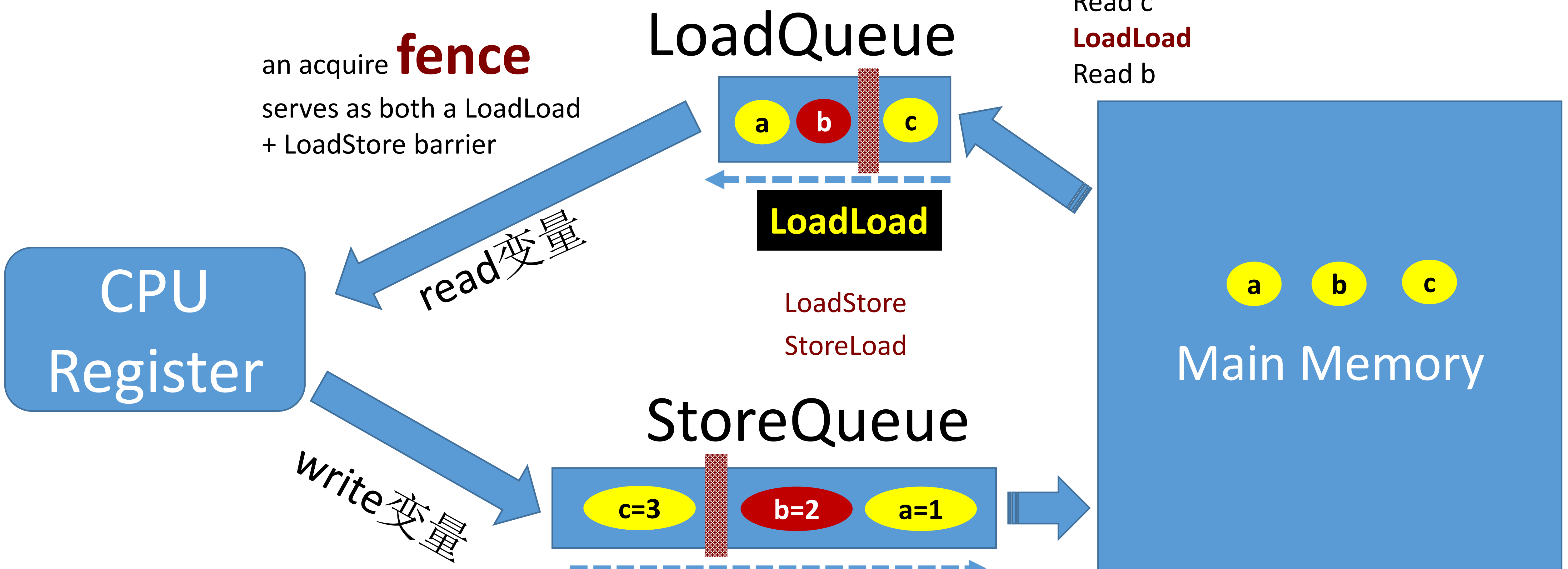
ensures that Store1's data are made visible to other processors (i.e., flushed to main memory) before data accessed by Load2 and all subsequent load instructions are loaded. StoreLoad barriers protect against a subsequent load incorrectly using Store1's data value rather than that from a more recent store to the same location performed by a different processor. Because of this, on the processors discussed below, a StoreLoad is strictly necessary only for separating stores from subsequent loads of the same location(s) as were stored before the barrier. StoreLoad barriers are needed on nearly all recent multiprocessors, and are usually the most expensive kind. Part of the reason they are expensive is that they must disable mechanisms that ordinarily bypass cache to satisfy loads from write-buffers. This might be implemented by letting the buffer fully flush, among other possible stalls.

Java内存模型

Store/Load指令

an acquire **fence** serves as both a LoadLoad + LoadStore barrier

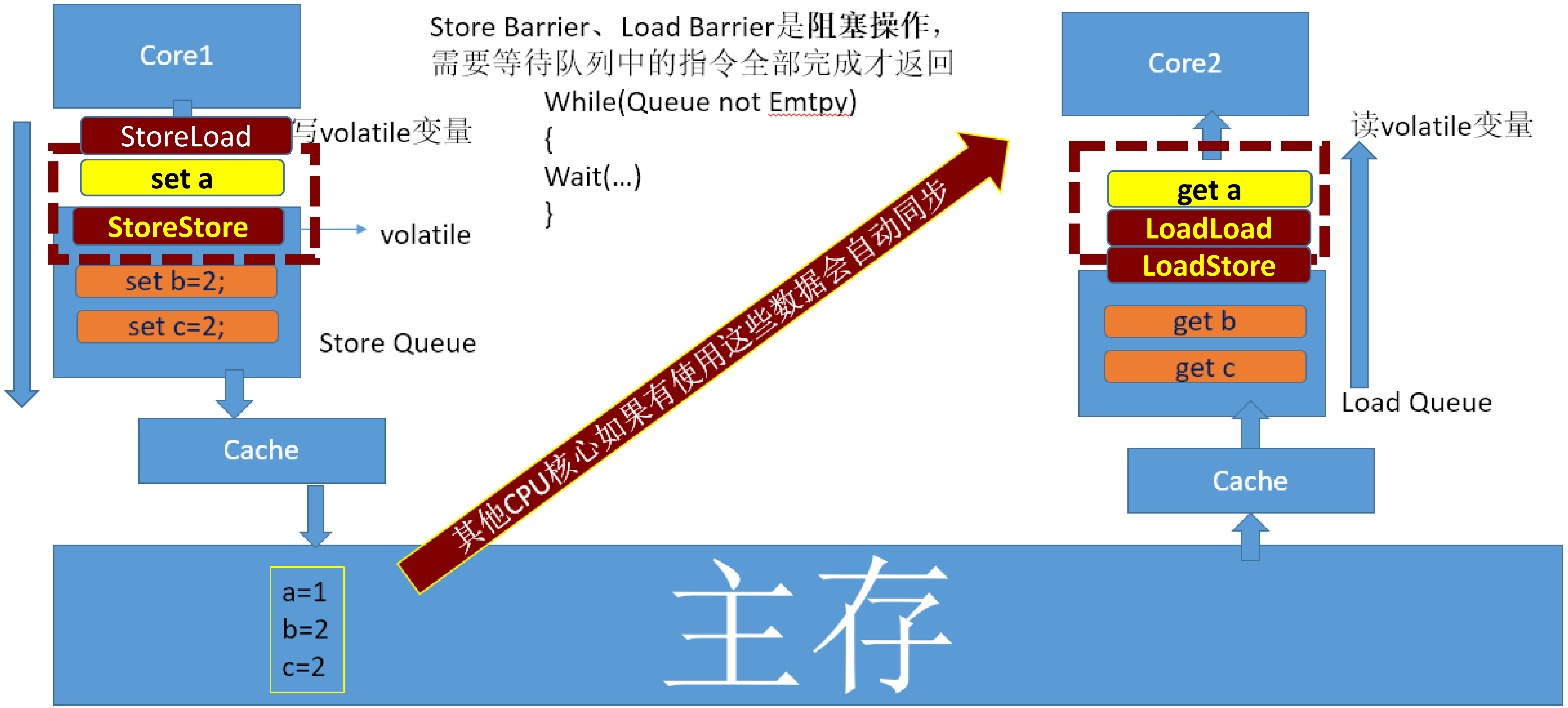
a release **fence** functions as both a LoadStore + StoreStore barrier.



b=2
StoreStore
c=3

volatile大法！！

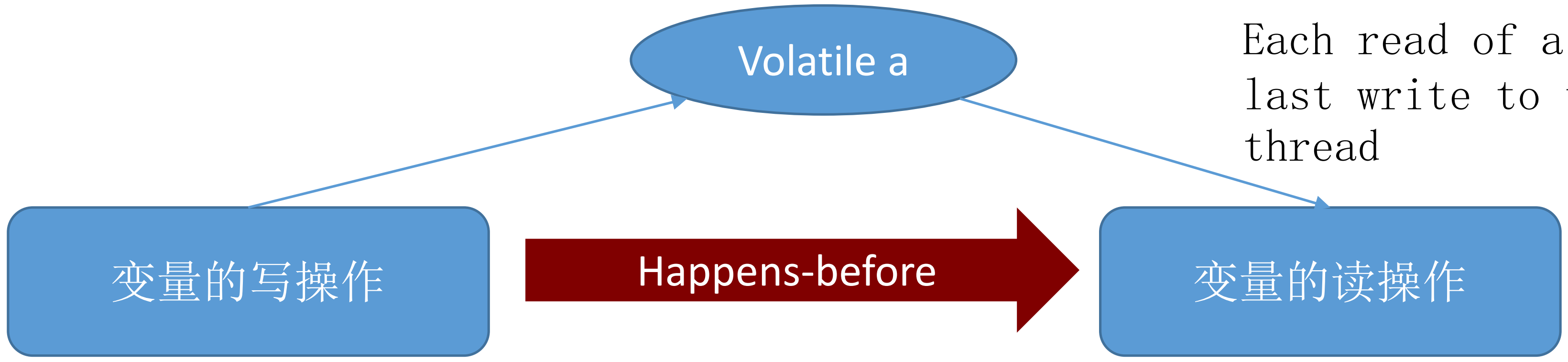
Java内存模型 Volatile变量



总结：**volatile** 字段在写入前后插入**Store**屏障，在读取前插入**Load**屏障。

Java内存模型

Volatile变量



Each read of a volatile will see the last write to that volatile by any thread

JDK 5：

If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.

```
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }
```

Thread1

Warning

Each read or write of a volatile field acts like "**half**" a synchronization, for purposes of visibility.

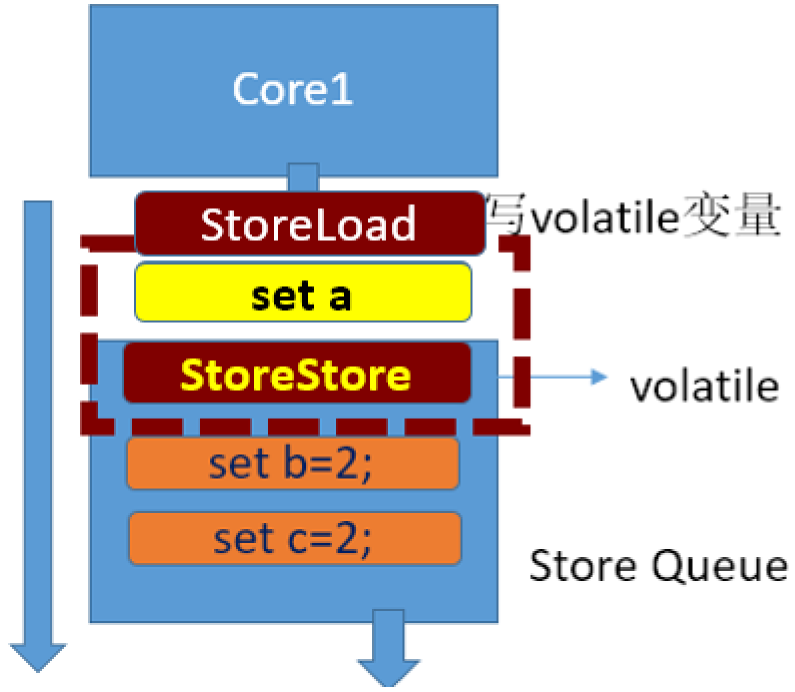
Note：Atomic类增加了lasySet方法，提升volatile变量的写性能

```
public void reader() {
  if (v == true) {
    //uses x - guaranteed to see 42.
  }
}
```

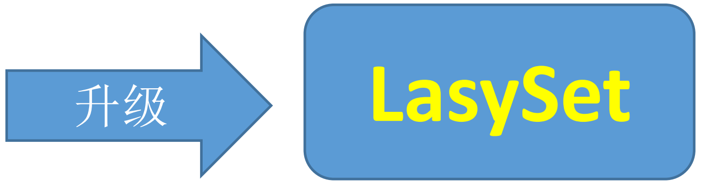
Thread2

Java内存模型 LasySet指令

龙果学院 roncoo.com



StoreLoad是一个耗费CPU时间的“栅栏”操作，所以去掉这个操作！



```
AtomicInteger i=new AtomicInteger(5);
i.lazySet(5);
```

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in section 17.4 of The Java™ Language Specification.

get has the memory effects of reading a volatile variable.

set has the memory effects of writing (assigning) a volatile variable.

lazySet has the memory effects of writing (assigning) a volatile variable except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-volatile writes. Among other usage contexts, lazySet may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.

weakCompareAndSet atomically reads and conditionally writes a variable but does not create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet. compareAndSet and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables.

lazySet offers a single writer a consistent volatile write mechanism, i.e. it is perfectly legitimate for a single writer to use lazySet to increment a counter, multiple threads incrementing the same counter will have to resolve the competing writes using CAS, which is exactly what happens under the covers of Atomic* for incAndGet.

Java内存模型

JDK8的内存屏障指令

龙果学院
roncoo.com

sun.misc.Unsafe

JDK 9进一步作出了修正

The three methods are:

```
/**
 * Ensures lack of reordering of loads before the fence
 * with loads or stores after the fence.
 */
void loadFence();
```

["volatile"-load] [LoadFence] [loads/stores]

Corresponds to C11 atomic_thread_fence(memory_order_acquire)

```
/**
 * Ensures lack of reordering of stores before the fence
 * with loads or stores after the fence.
 */
void storeFence();
```

[stores/loads] [StoreFence] ["volatile"-store]

Corresponds to C11 atomic_thread_fence(memory_order_release)

```
/**
 * Ensures lack of reordering of loads or stores before the fence
 * with loads or stores after the fence.
 */
void fullFence();
```

Corresponds to C11 atomic_thread_fence(memory_order_seq_cst).

C++ & Java

Java内存模型

JDK8的内存屏障指令

龙果学院 roncoo.com

JVMs don't have advertised mechanisms providing memory orderings that were not envisioned originally or in the JSR 133 memory model specs. (But are present for example in the recent C11/C++11 specs.) These include several constructions **used in java.util.concurrent and other low-level libraries that currently rely on undocumented** (and possibly transient) properties of existing intrinsics.**Adding these methods at the VM level permits use by JDK libraries in support of JDK 8 features, while also opening up the possibility of later exporting the base functionality via new java.util.concurrent APIs**. This may become essential to **allow people developing non-JDK low-level libraries** if upcoming modularity support makes these methods impossible for others to access.

The three methods provide the three different kinds of memory fences that some compilers and processors need to ensure that particular accesses (loads and stores) do not become reordered. In practice, they are identical in effect to existing getXXXVolatile, putXXXVolatile, and putOrderedXXX methods, except that they do not actually perform an access; they just ensure the ordering. However, they conceptually differ in one way: According to the current JMM, some language-level uses of volatile may be reordered with some uses of non-volatile variables. But this would not be allowed here. (It is not allowed in the current intrinsics either, but this is an undocumented difference between intrinsics-based vs language-based volatile access.)

JEP 171: Fence Intrinsics

Owner	Doug Lea
Created	2012/11/27 20:00
Updated	2014/08/07 09:45
Type	Feature
Status	Completed
Component	hotspot / runtime
Scope	JDK
Discussion	hostspot dash dev at openjdk dot java dot net
Priority	4
Endorsed by	Mark Reinhold
Release	8
Issue	8046161

Java内存模型 C++11 fences

龙果学院 roncoo.com

C++11 Standards Committee did a great job specifying the meaning of these memory fences. They enable robust algorithms which scale well across multiple cores, and map nicely onto today’s most common CPU architectures.

Acquire and release **fences** are considered low-level lock-free operations. If you stick with higher-level, sequentially consistent atomic types, such as volatile variables in Java 5+, or default atomics in C++11, you don’t need acquire and release fences. **The tradeoff is that sequentially consistent types are slightly less scalable or performant for some algorithms.**

你不就是只会volatile么？我还知道比你更快的“栅栏”！！

Who 怕 Who！！

Java内存模型

Powerful fences

Atomic operations	Java int var	Java volatile int var	C11 atomic_int var
Relaxed load	x = var;	not available	x = atomic_load_explicit(var, memory_order_relaxed);
Relaxed store	var = x;	not available	atomic_store_explicit(var, x, memory_order_relaxed);
Barrier load	Unsafe.fullFence(); x = var;	x = var;	x = atomic_load(var);
Barrier store	var = x; Unsafe.fullFence();	var = x;	atomic_store(var, x);
Compare-And-Swap	Unsafe.compareAndSwapInt(var, 0, 1);	Unsafe.compareAndSwapInt(var, 0, 1);	atomic_compare_exchange_strong(var, 0, 1);
Fetch-And-Add	Unsafe.getAndAddInt(var, 1);	Unsafe.getAndAddInt(var, 1);	atomic_fetch_add(var, 1);

Unsafe.fullFence()
Unsafe.loadFence()
Unsafe.storeFence()

Unsafe.compareAndSwapInt()
Unsafe.compareAndSwapObject()
Unsafe.compareAndSwapLong()
Unsafe.getAndAddInt()

Happens-before规则

全局共享变量x和y，两个线程分别执行下面两条指令：

初始条件： x = y = 0;

线程 1	线程 2
x = 1;	y=1;
r1 = y;	r2 = x;

因为多线程程序是交错执行的，所以程序可能有如下几种执行顺序：

Execution 1	Execution 2	Execution 3

x = 1;
r1 = y;
y = 1;
r2 = x;
结果:r1==0 and r2 == 1
	y = 1;
	r2 = x;
	x = 1;
	r1 = y;
	结果: r1 == 1 and r2 == 0
		x = 1;
		y = 1;
		r1 = y;
		r2 = x;
		结果: r1 == 1 and r2 == 1

解决思路

Happens-before规则

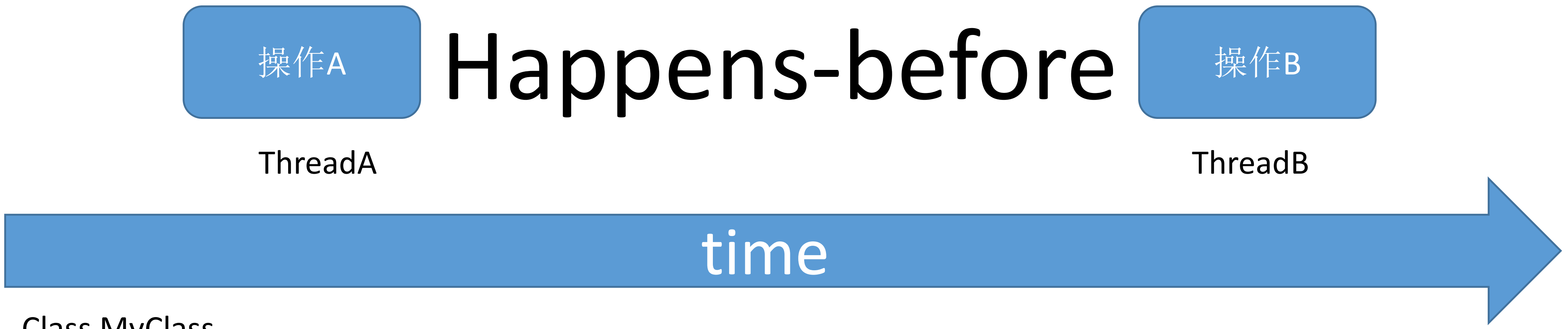
1： 锁/同步

Volatile变量

2： volatile

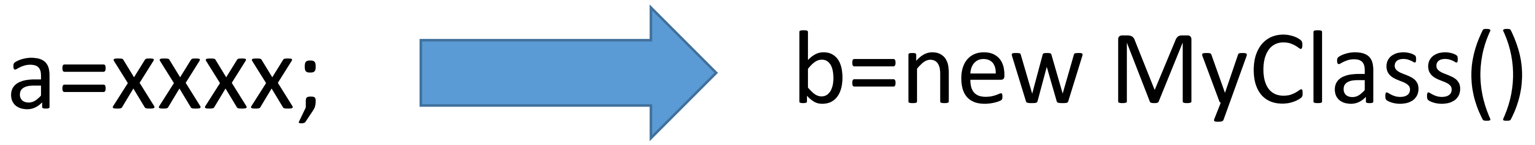
Happens-before规则

保证不同线程并发执行时候的指令先后顺序

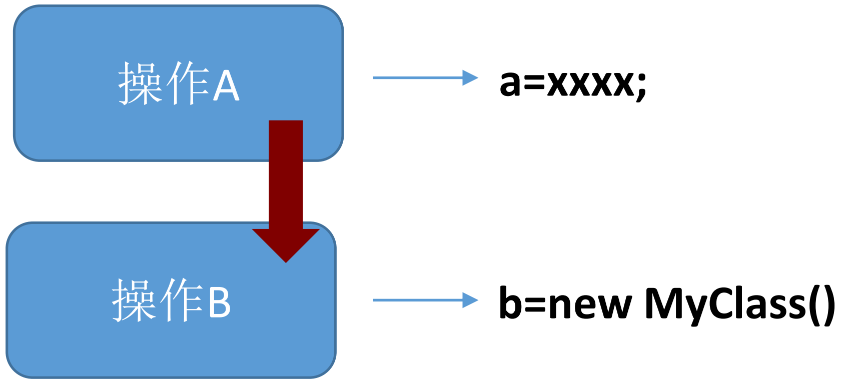


Class MyClass
Private final Object a;
Public MyClass()
{
 a=xxxx;
}

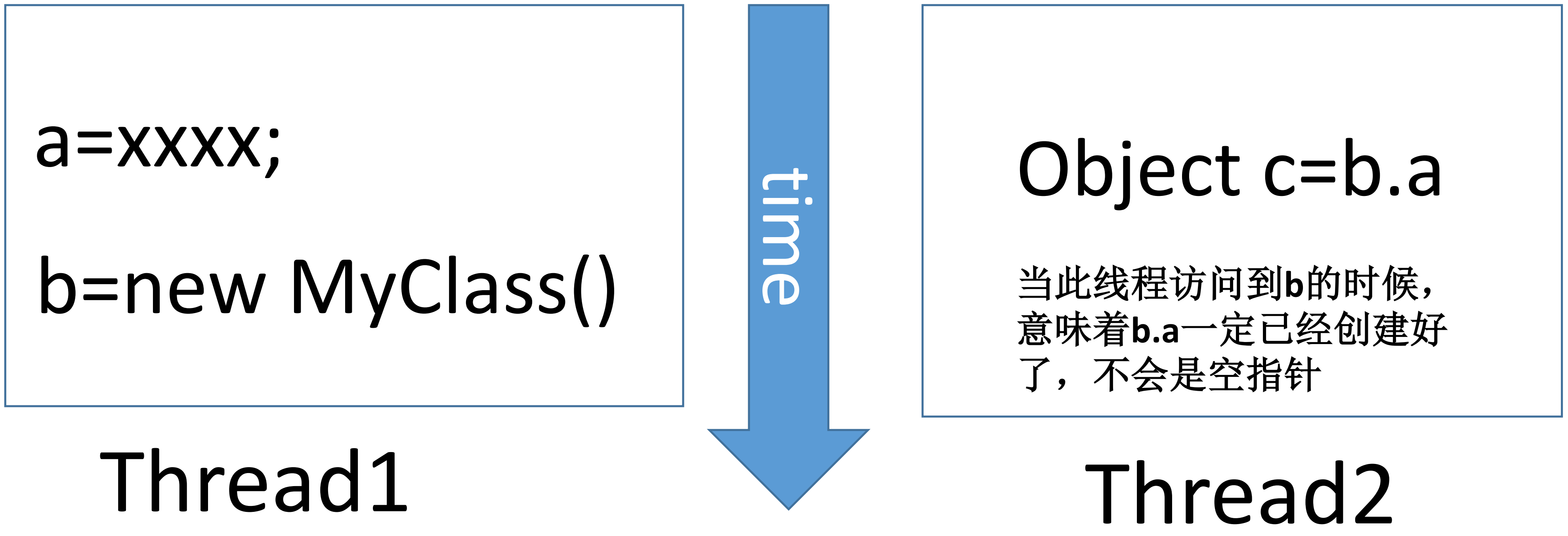
b=new MyClass()



Happens-before规则



Happens-before

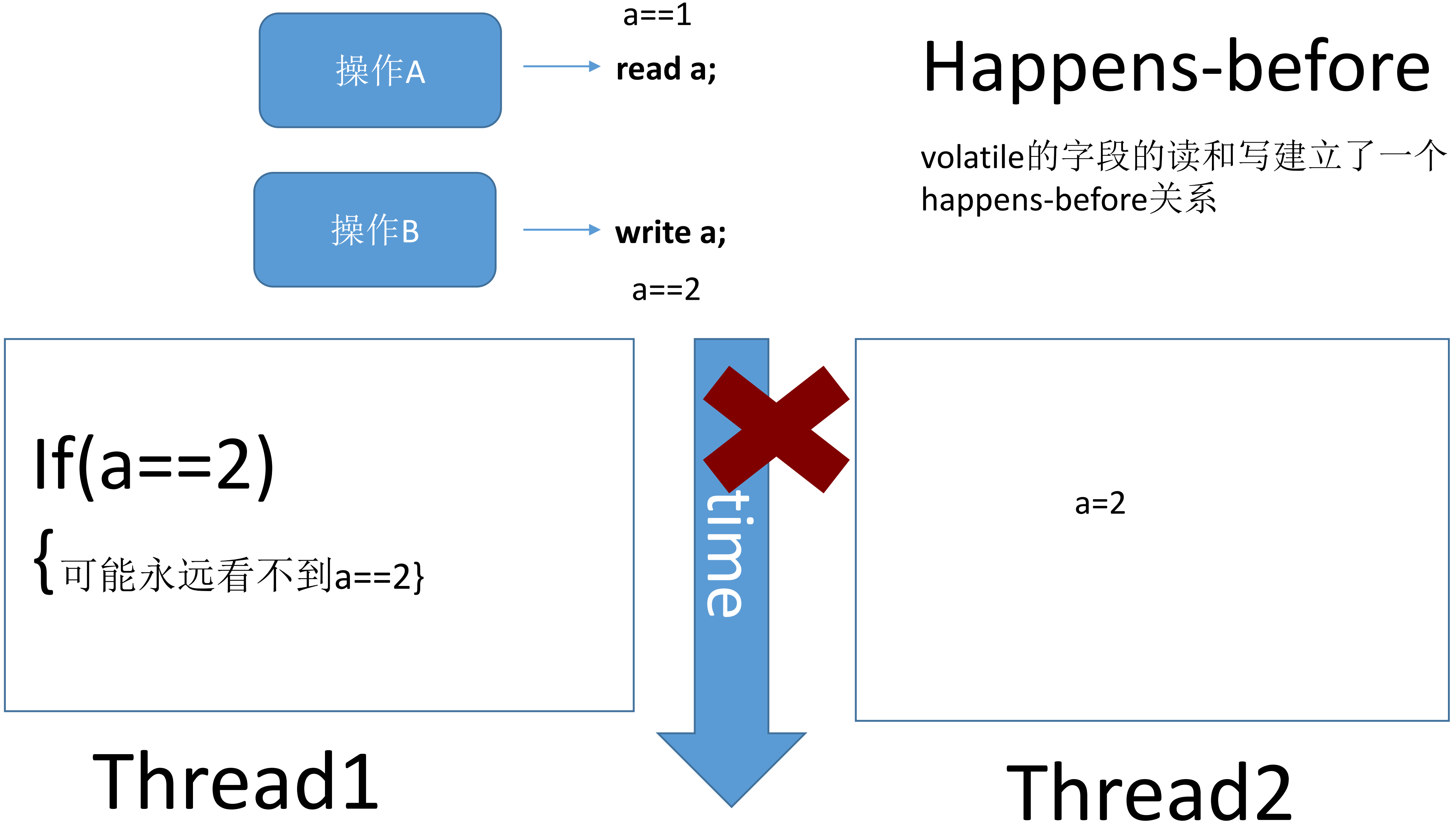


```
a=xxxx;

b=new MyClass()
```

```
Object c=b.a
```

当此线程访问到b的时候，意味着b.a一定已经创建好了，不会是空指针

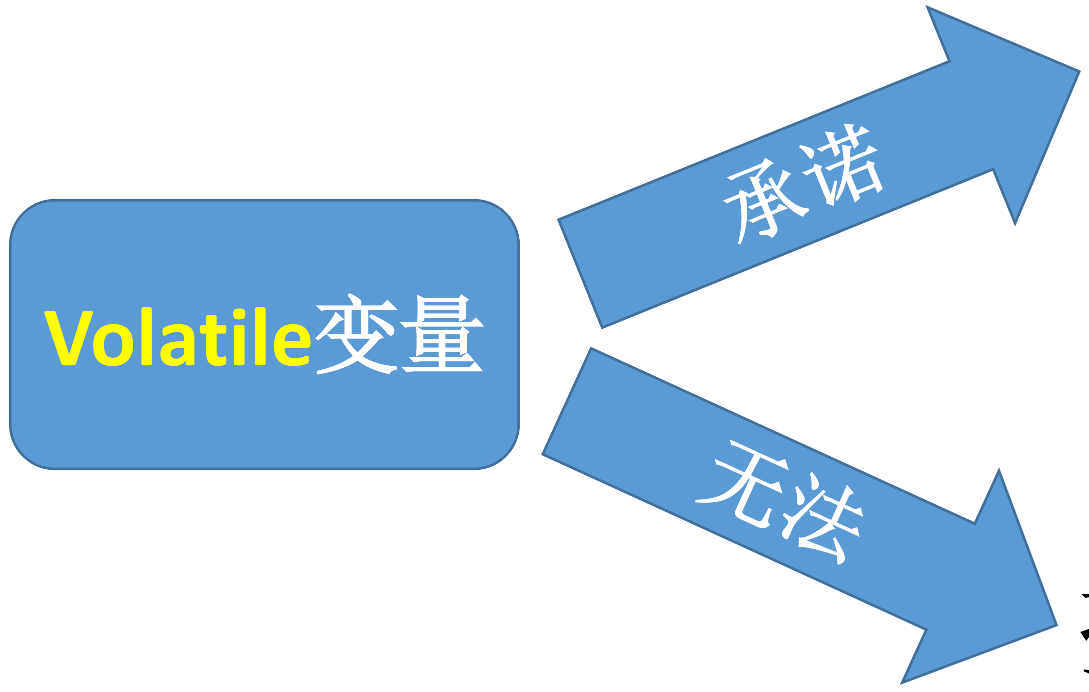
Happens-before规则



Happens-before

volatile的字段的读和写建立了一个happens-before关系

CAS指令



内存可见性

Happens-Before

提供变量修改的原子操作

int volatile I;

i=i+1;

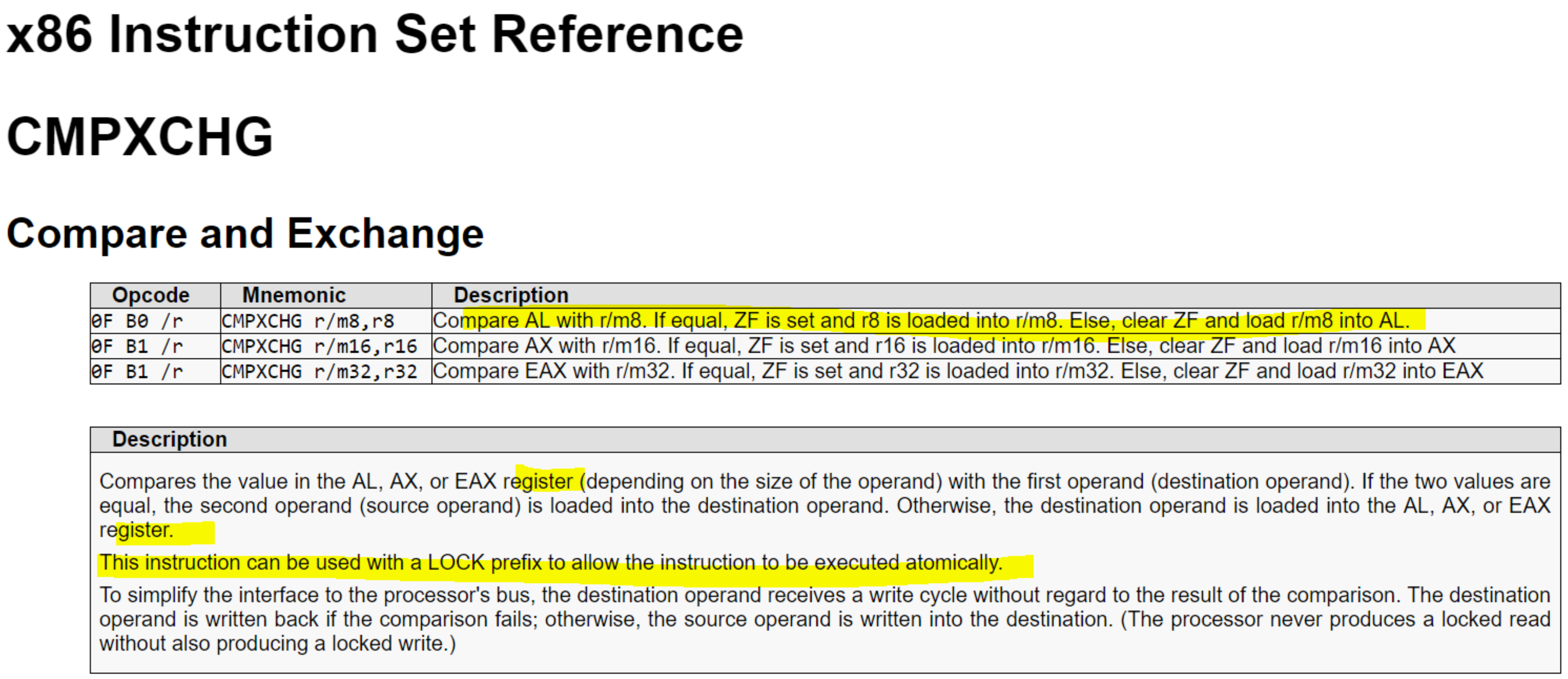
多线程同时执行会有并发问题

```
read i
i=i+1
write i
```

✗

CAS指令

龙果学院 roncoo.com



x86 Instruction Set Reference

CMPXCHG

Compare and Exchange

Opcode	Mnemonic	Description
`0F B0 /r`	`CMPXCHG r/m8,r8`	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
`0F B1 /r`	`CMPXCHG r/m16,r16`	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX
`0F B1 /r`	`CMPXCHG r/m32,r32`	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX

Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

CAS指令

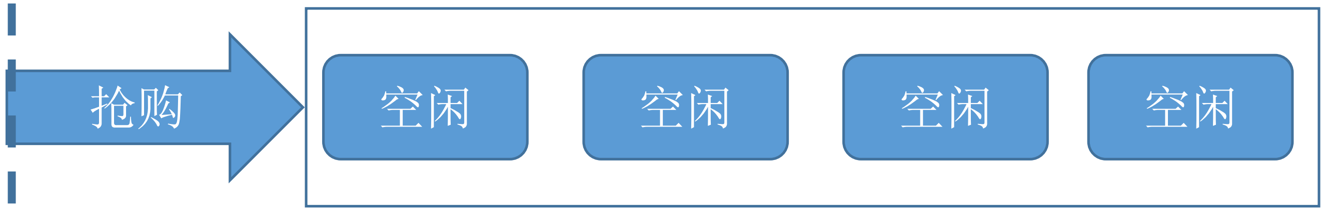
```
 * @since 1.5
 * @author Doug Lea
*/
public class AtomicInteger extends Number implements
java.io.Serializable {
 public final int getAndIncrement() ;
 public final boolean compareAndSet(int expect, int update);
```

java.util.concurrent.atomic

原子++操作

```
While(true)
{
  theTicket=ticks.getNextAvailable();
 if(theTicket.sellFlag. compareAndSet(0,1))
   {
        //终于抢到了，
       do 发微信告诉妹子晚上看电影
       do 告诉Leader课程班助理，晚上加班不上课
       break;
    }
}
```

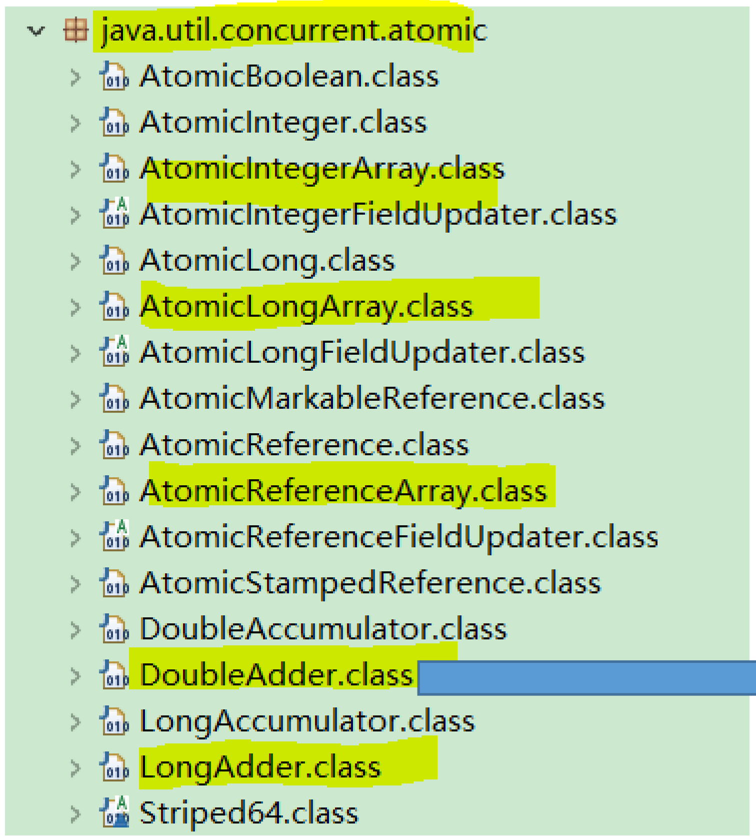
多线程高级并发控制



电影票(ticks【数组或链表】）

Atomic类型数据

java.util.concurrent.atomic 原子操作



- java.util.concurrent.atomic
  - AtomicBoolean.class
  - AtomicInteger.class
  - AtomicIntegerArray.class
  - AtomicIntegerFieldUpdater.class
  - AtomicLong.class
  - AtomicLongArray.class
  - AtomicLongFieldUpdater.class
  - AtomicMarkableReference.class
  - AtomicReference.class
  - AtomicReferenceArray.class
  - AtomicReferenceFieldUpdater.class
  - AtomicStampedReference.class
  - DoubleAccumulator.class
  - DoubleAdder.class
  - LongAccumulator.class
  - LongAdder.class
  - Striped64.class

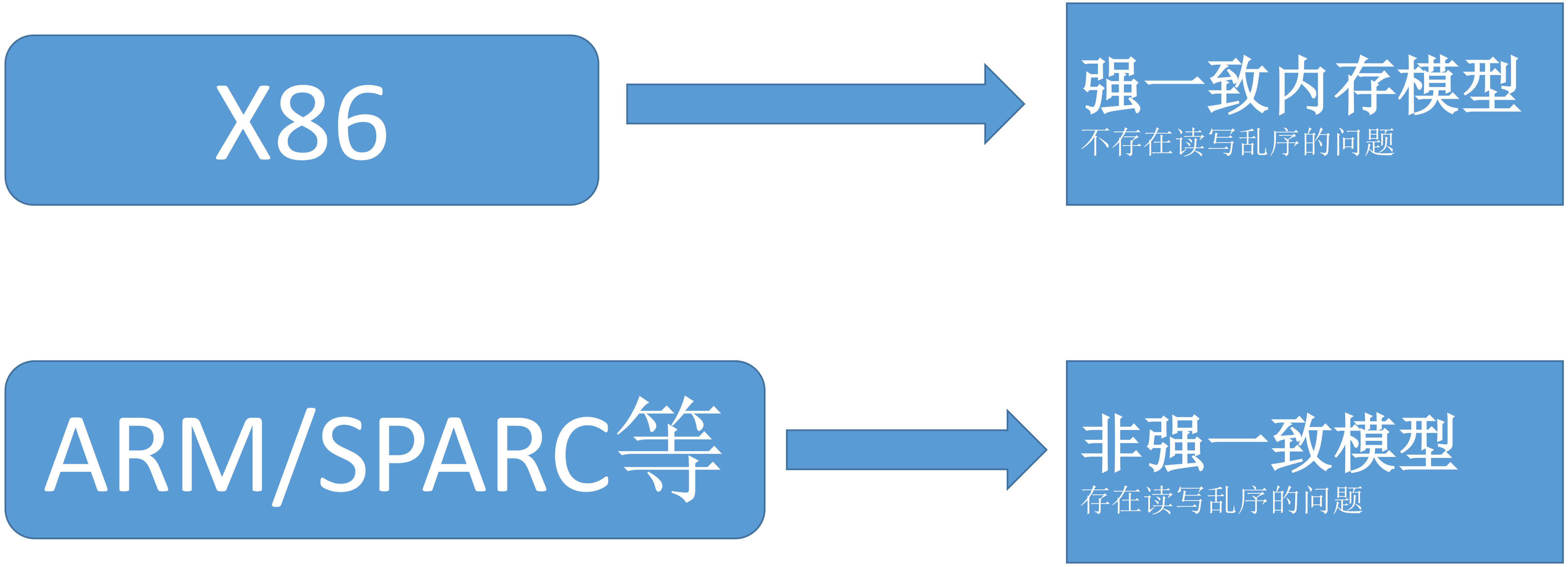
数组、对象引用等都提供了原子操作能力

JDK8 Newbility!!!

under high contention, expected throughput of this class is significantly higher, at the expense of higher space consumption.

Java 8 加法器 – 所做的事情是当一个直接CAS由于竞争失败时，它将delta保存在为该线程分配的一个内部单元对象中，然后当intValue()被调用时，它会将这些临时单元的值再相加到结果和中。这就减少了返回重新CAS或者阻塞其他线程的必要。多么聪明的做法！

一些参考资料



一些参考资料

The x86/x64 instruction set actually does contains three fence instructions: LFENCE, SFENCE, and **MFENCE**. LFENCE and SFENCE are apparently not needed on the current architecture, but MFENCE is useful to go around one particular issue: if a core reads a memory location it previously wrote, the read may be served from the store buffer, even though the write has not yet been written to memory

The mainstream x86 and x64 processors implement a strong memory model where memory access is effectively volatile. So, **a volatile field forces the compiler to avoid some high-level optimizations** like hoisting a read out of a loop, but otherwise results in the same assembly code as a non-volatile read.

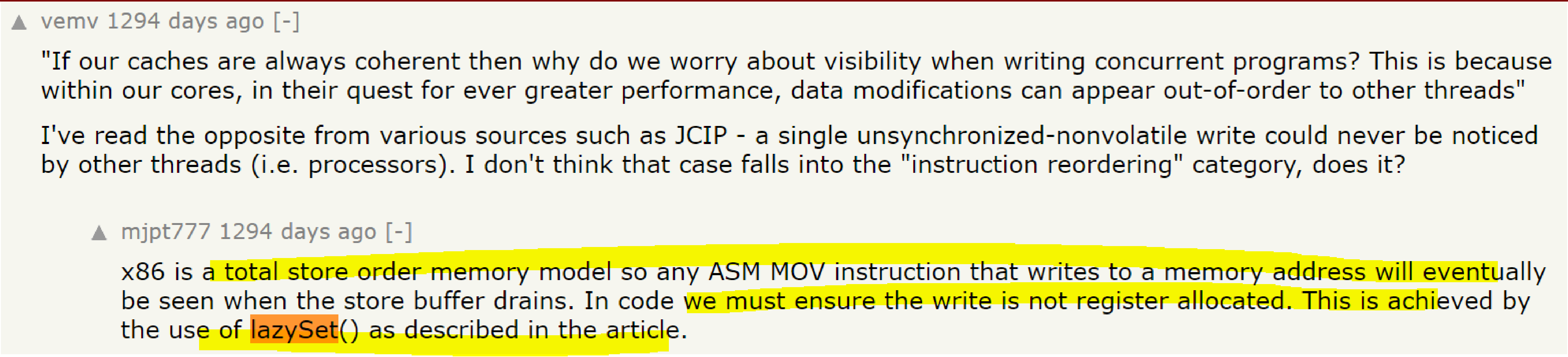
Hoisting

Loop-invariant expressions can be hoisted out of loops, thus improving run-time performance by executing the expression only once rather than at each iteration.

```
void f (int x, int y)
{
  int i;
  for (i = 0; i < 100; i++)
   {
     a[i] = x + y;
   }
}
```

```
void f (int x, int y)
{
  int i;
  int t= x + y;
  for (i = 0; i < 100; i++)
   {
     a[i] = t;
   }
}
```

一些参考资料



▲ vemv 1294 days ago [-]

"If our caches are always coherent then why do we worry about visibility when writing concurrent programs? This is because within our cores, in their quest for ever greater performance, data modifications can appear out-of-order to other threads"

I've read the opposite from various sources such as JCIP - a single unsynchronized-nonvolatile write could never be noticed by other threads (i.e. processors). I don't think that case falls into the "instruction reordering" category, does it?

▲ mjpt777 1294 days ago [-]

x86 is a total store order memory model so any ASM MOV instruction that writes to a memory address will eventually be seen when the store buffer drains. In code we must ensure the write is not register allocated. This is achieved by the use of lazySet() as described in the article.

龙果学院
roncoo.com

谢谢观看

Leader全栈Java报名群QQ 332702697