# Java内存模型

## Java内存模型的基础

### 并发模型的两个关键问题

线程之间如何通信，线程之间如何同步；

**通信**是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种：**共享内存**和**消息传递**

**共享内存并发模型：**线程之间共享程序的公共状态，通过写-读内存中的公共状态进行隐式通信。

**消息传递并发模型**：线程之间没有公共状态，线程之间必须通过发送消息来显式通信

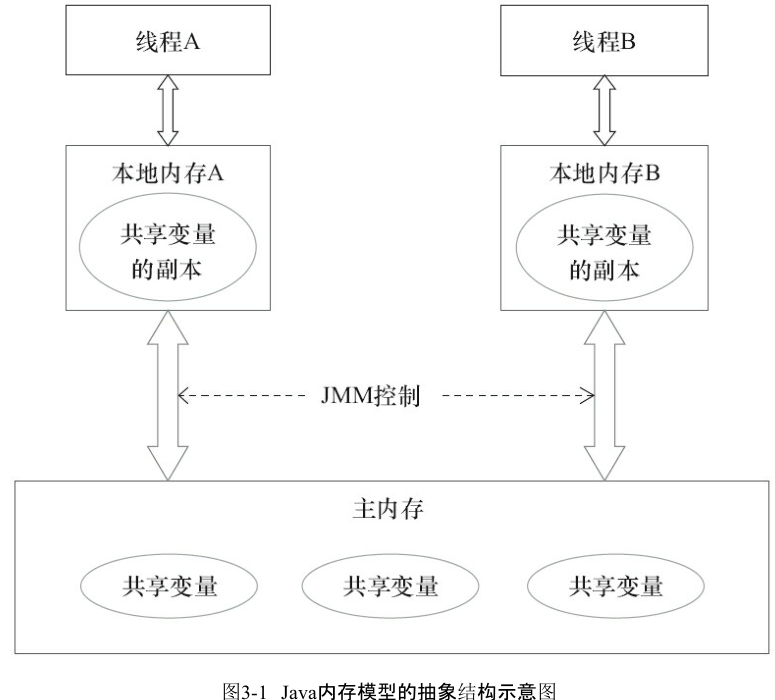
**同步**：程序中用于控制不同线程间，操作执行顺序的机制。

Java并发采用的是共享内存模型，java线程之间通信总是隐式进行，通信过程对程序员完全透明

### Java内存模型的抽象结构

堆内存在线程之间共享，实例，静态域，数组元素所有对象都在堆内存中。局部变量，方法参数和异常处理器参数不会在线程之间共享，不会有内存可见性问题。

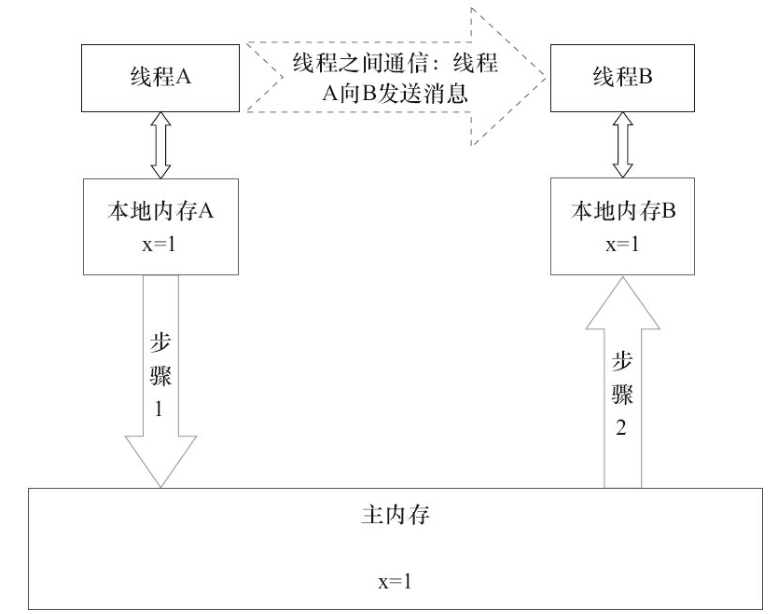
JMM（java memory model）java内存模型，**jmm决定一个线程对共享变量的写入何时对另一个线程可见**。抽象角度：jmm定义了线程和主存之间的抽象关系，线程之间的共享变量存储在主存中，每个线程都有一个**私有的本地内存**，本地内存中存储着该线程以读写**共享变量的副本**。本地内存是jmm的一个概念，并不真实存在，包括缓存、缓冲区、寄存器和其他硬件和编译器优化。



线程AB之间要通信的话，必须要经历2个步骤：

①线程a把本地内存a中更新过的内存变量刷新到主存中

②线程b到主内存中去读线程A之前已经更新过的共享变量



JMM通过控制**主内存和每个线程的本地内存之间的交互**，来为java程序员提供内存**可见性**保证。

### 从源代码到指令序列的重排序

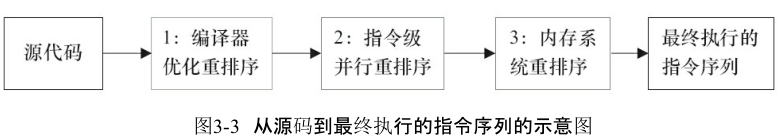
三种重排序：

①编译器优化重排序。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指令级并行重排序，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。

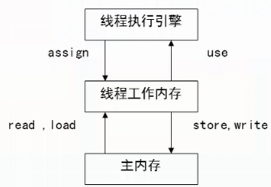
③内存系统重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是乱序执行。

Java源代码到最终实际执行的指令序列，会分别经历下面三种重排序，





Storeload barrier是一个全能型屏障，同时具有其他3个屏障的效果。



### Happens—before

在JMM中，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要存在happens-before关系。既可以是在一个线程内，也可以是在不同线程之间

与程序员密切相关happens-before规则如下：

①程序顺序规则：一个线程中的操作，先行发生于该线程的任意后面的操作

②监视器锁规则：对一个锁的解锁，先行发生与随后对这个锁的加锁操作

③volatile变量规则：对一个volatile域的写，先行发生于任意后面对这个volatile域的读

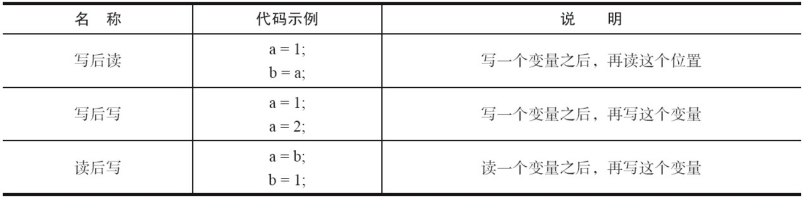
④传递性：如果a先行发生于b，且b先行发生于c，那么a先行发生于c

注：两个操作之间具有happens-before关系，**并不意味着前一个操作必须要在后一个操作之前执行**！！happens-before仅仅要求前一个操作的执行结果对后一个**操作可见**，且前一个操作**按顺序排在**后一个操作之前。

## 重排序

编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段。

### 数据依赖性

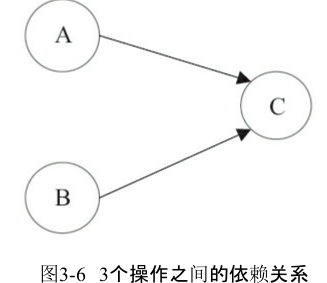
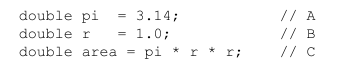


如果两个操作访问同一个变量，且这两个操作有一个为写操作，此时这两个操作之间存在数据依赖性，如上表所示。

### As -if-serial

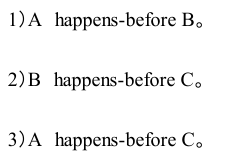
不管怎么重排序，单线程程序的执行结果不能被改变。编译器、runtime和处理器都必须遵守as-if-serial语义

如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。



编译器和处理一下可以变换AB的位置，不影响结果

### 程序顺序规则

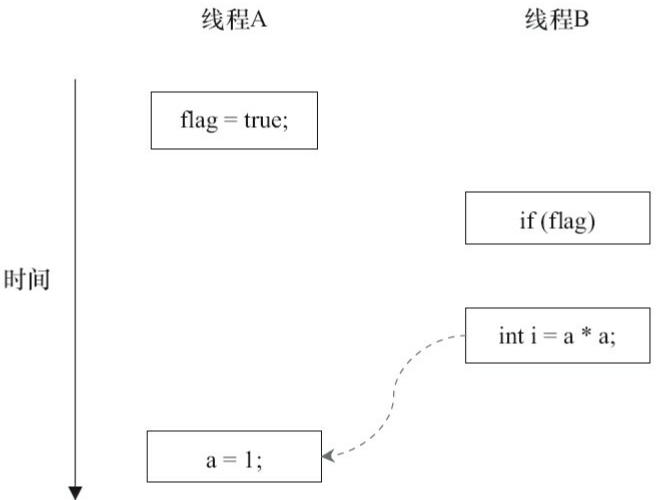


这里A happens-before B,但实际上执行B却可以在A之前执行。JMM仅要求前一个操作的执行结果对后一个操作可见，而且前一个操作顺序排在第二个操作之前。这里操作a的执行结果不需要对b可见；且重排序a与b之后的执行结果，与操作ab按顺序执行的结果一致。这种情况下，JMM认为重排序不非法，允许重排序

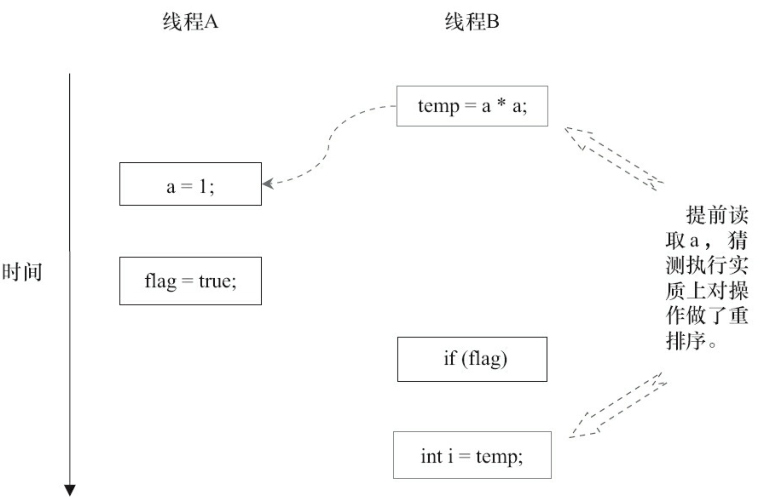
### 重排序对多线程的影响

|  |
| --- |
| **package** cn.huangwei.third;  **public** **class** ReorderTest {  **int** a = 0;  **boolean** flag = **false**;    **public** **void** writer(){  a = 1;//1  flag = **true**;//2  }    **public** **void** reader(){  **if**(flag){//3  **int** i = a \* a;//4  }  }  } |

由于操作1和2之间没有数据依赖，编译器和处理器可以对两个操作重排，同样操作3和4没有数据依赖关系，也可以对两个操作数重排序。



操作1和2进行重排，程序执行时，线程a首先标记flag，随后b读入这个变量。由于条件判断为真，线程读取变量a，此时变量a没有写入，因此语义被重排破坏了



操作3和操作4存在控制依赖关系，当代码存在控制依赖时，会影响指令序列的并行度。为此处理器会猜测执行克服控制相关性对并行度的影响。

线程b提前读取a，计算a\*a，然后把计算结果临时存在一个名为重排序的缓冲中，当操作3判断为真时，将计算结果写入i中；猜测执行破坏了多线程程序的语义。

**单线程程序**中，对存在**控制依赖的操作重排序，不会改变执行结果**；但在**多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果**。

## 顺序一致性

顺序一致性内存模型是一个理论参考模型，处理器的内存模型和编程语言的内存模型都会以顺序一致性内存模型作为参照

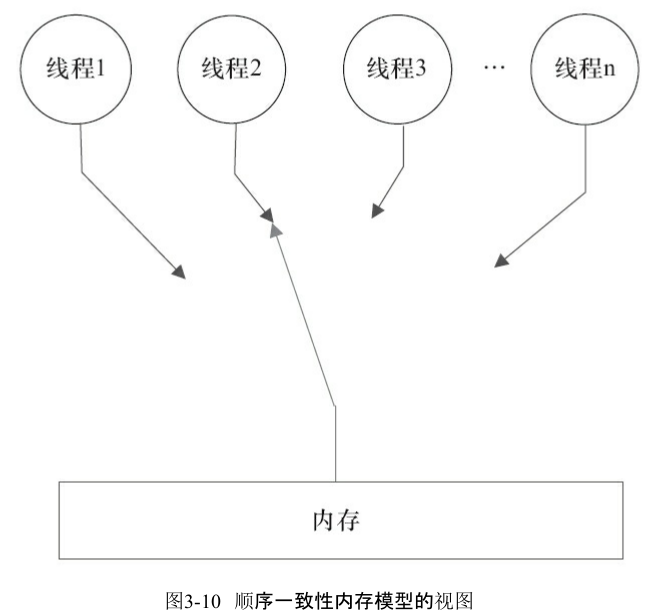
### 数据竞争与顺序一致性

数据竞争：在一个线程中**写一个变量**，在另一个线程**读一个变量**，而且**写和读没有通过同步**来排序。

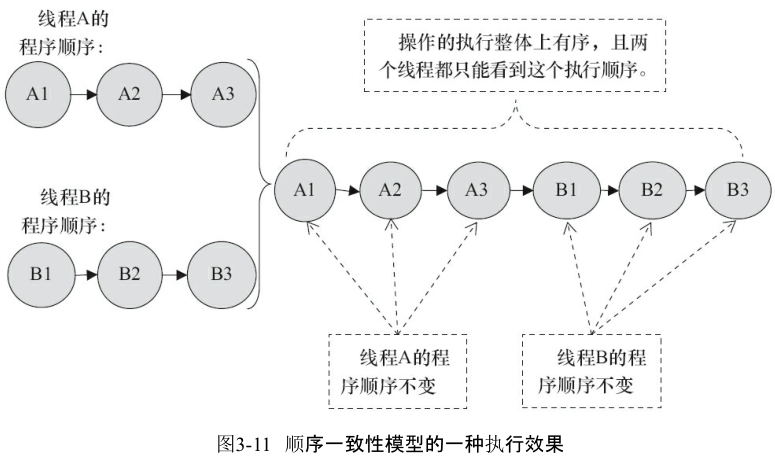
JMM对正确同步的多线程程序的内存一致性做了如下包含保证：

如果程序是**正确同步**的，程序的执行将具有**顺序一致性**，即程序的**执行结果**与程序在**顺序一致性内存模型**中的**执行结果**相同。

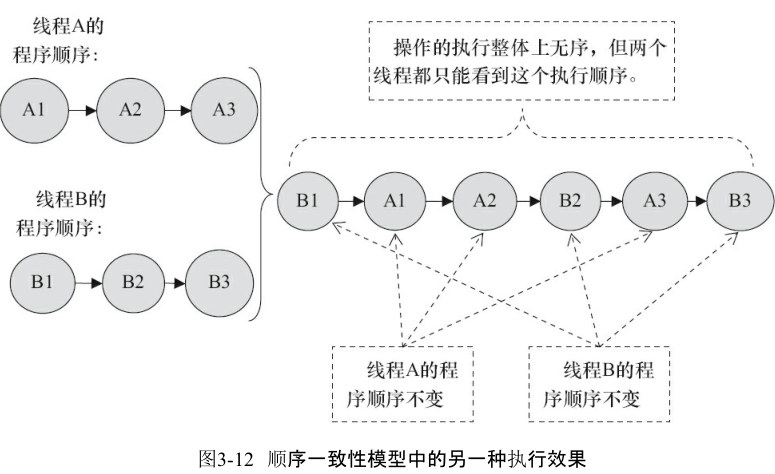
### 顺序一致性内存模型



顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摇摆的开关可以连接到任意一个线程，同时，每个线程必须按照程序的顺序来执行内存读写操作。



使用监视器进行同步的顺序一致性模型的执行结果。

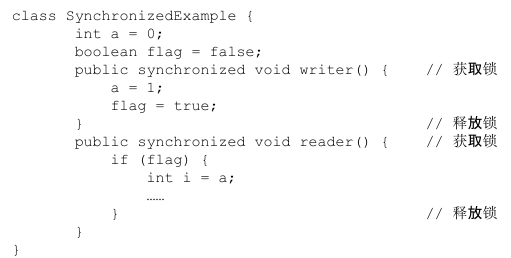


未进行同步程序在顺序一致性模型中的执行结果

未同步程序在顺序一致性模型中虽然整**体执行顺序是无序**的，但所有线程都只能看到一个一致的整体执行顺序，线程AB看到的执行顺序为B1→A1→A2→B2→A3→B3；之所以能看到这个保证是因为**顺序一致性内存模型中的每个操作必须立即对任意线程可见**。

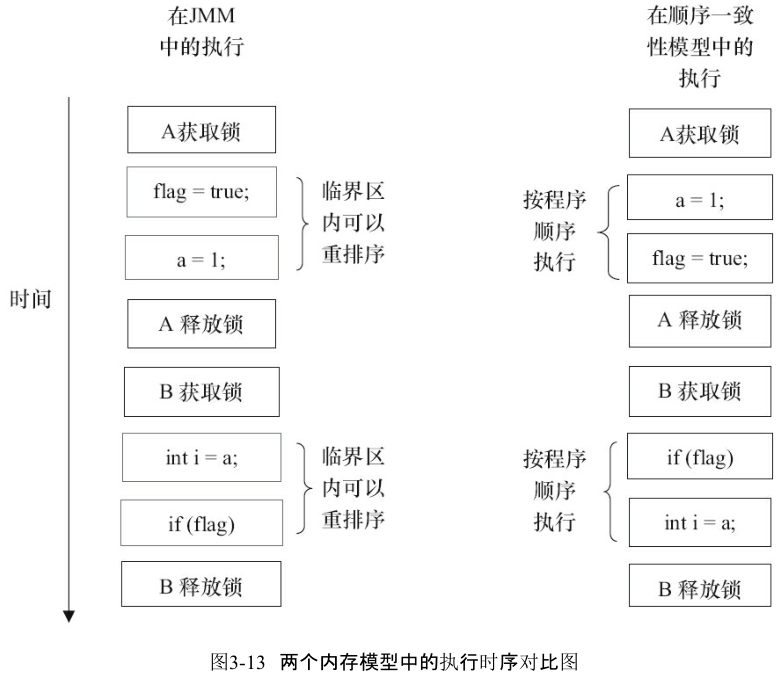
JMM中没有这个保证，**未同步程序在JMM中不仅整体执行无序，而且所有线程看到的操作顺序也可能不一致**；例如线程a对数据修改，然后存在本地内存中，在没有刷新到主存之前，其他操作会认为这个修改操作根本没有执行。

### 同步程序的顺序一致性效果



顺序一致性模型中，所有操作完全按程序的顺序串行执行。

JMM中，临界区的代码可以重排序，虽然线程A在临界区内进行了重排，但由于监视器互斥执行，这里的线程B根本无法“观察”到线程A的重排，这种重排既提高的效率，又不影响执行结果。



JMM的基本方针为：在不改变（正确同步）程序执行结果的前提下，尽可能地为编译器和处理器的优化打开方便之门

### 未同步程序的执行特性

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。因为这样做会使的JMM禁止大量的处理器和编译器的优化，这对程序的性能会有很大影响；而且未同步程序在顺序一致性模型中执行时，整体是无序的，其执行结果往往无法预知。保持结果一致没有什么意义。

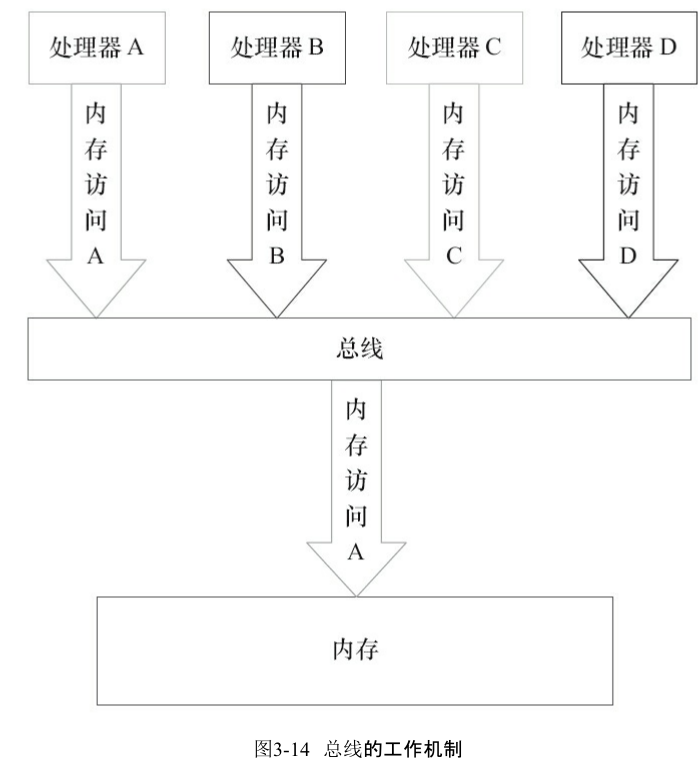
未同步程序在顺序一致性模型和JMM模型中的执行特性差异：

①顺序一致性模型保证单线程内的操作会按程序顺序执行，JMM不保证单线程内操作会按程序的顺序执行。

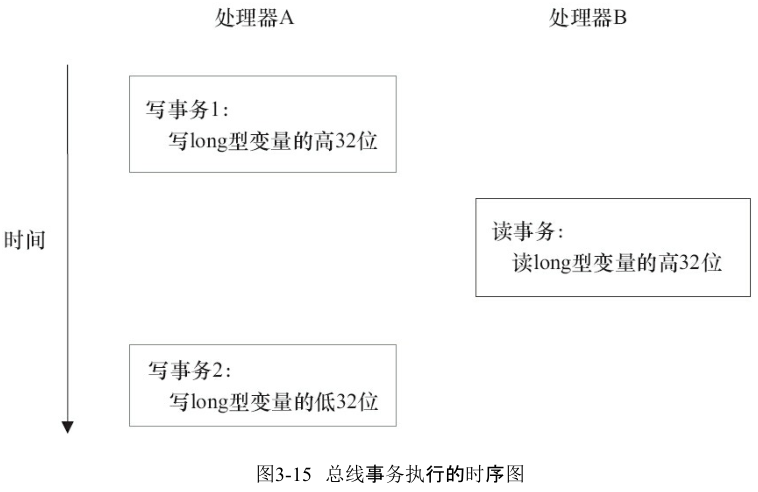
②前者保证所有线程只能看到一致的操作，后者不保证所有线程能看到一致操作

③JMM不保证64位的long和double的写操作具有原子性，顺序一致性模型保证所有内存读写操作具有原子性

第三点与处理器总线工作密切相关，处理器和内存数据传递通过总线事务完成，包括读/写事务。



如果A处理器执行总线事务期间（不管是读还是写），即使其他处理器发起了总线事务，此时其他处理器请求会被总线禁止



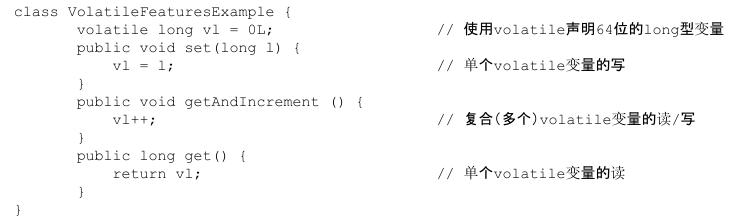
JSR-133以前的就模型中，64位的long/double变量读写可以被拆成两个32位的读写操作，如果出现上图的状况，没有写完就开始读，就会有意想不到的结果。

JSR-133内存模型开始（即从JDK5开始），仅仅只允许写操作能够拆分，读操作不可拆分，读操作具有原子性。

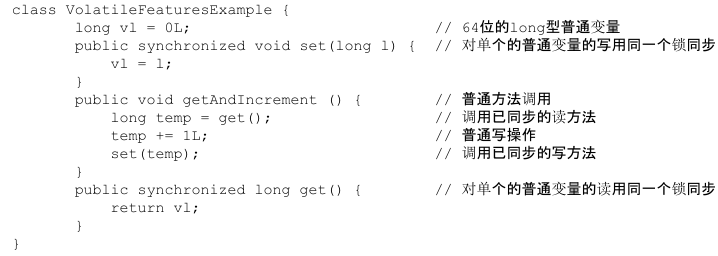
## Volatile内存语义

### Volatile特性

理解volatile的一个好方法是把对volatile变量的单个读写，看成是使用同一个锁对这些单个读写进行了同步。



多线程中，程序与下面等价



一个volatile变量的单个读/写操作，与(一个普通变量都是使用同一个锁来同步的读/写操作)，它们之间的执行效果相同。

锁的happens-before规则保证释放锁和获取锁的两个线程之间的内存可见性，这意味着对一个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

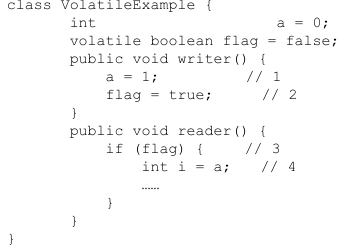
特性：

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

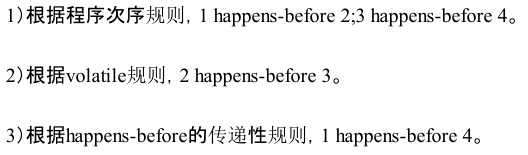
单个读写操作的原子性：但是不保证i++这种复合操作的原子性

### Volatile写-读建立的happens-before关系

从内存语义上将，volatile的写-读与锁的释放-获取有相同的效果：volatile写和锁的释放有相同语义，volatile的读与锁的获取有相同语义



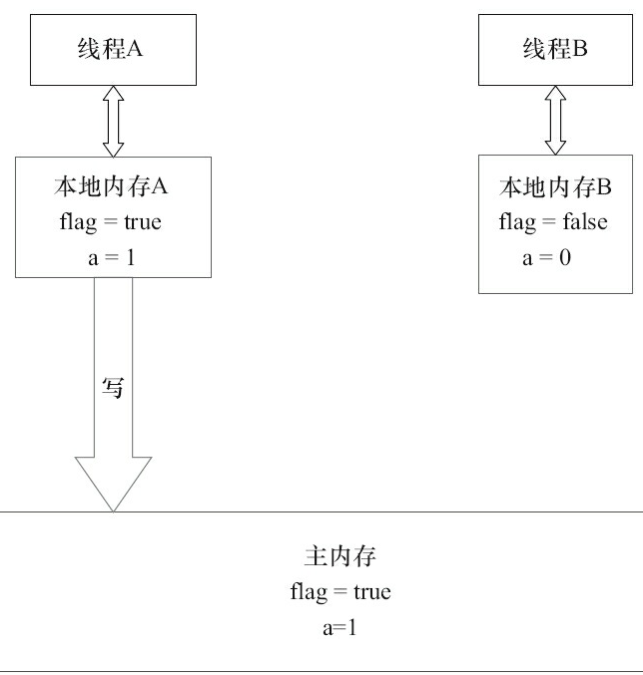
线程A执行writer之后，线程B执行reader，根据happens-before规则



这里线程A写一个volatile变量flag后，线程B读同一个volatile变量。**线程A写volatile之前所有可见的共享变量，比如a = 1；，在线程B读同一个volatile变量后，立即能看到共享变量a**；因为上述happens-before可知，a = 1,happens-before于int I = a；

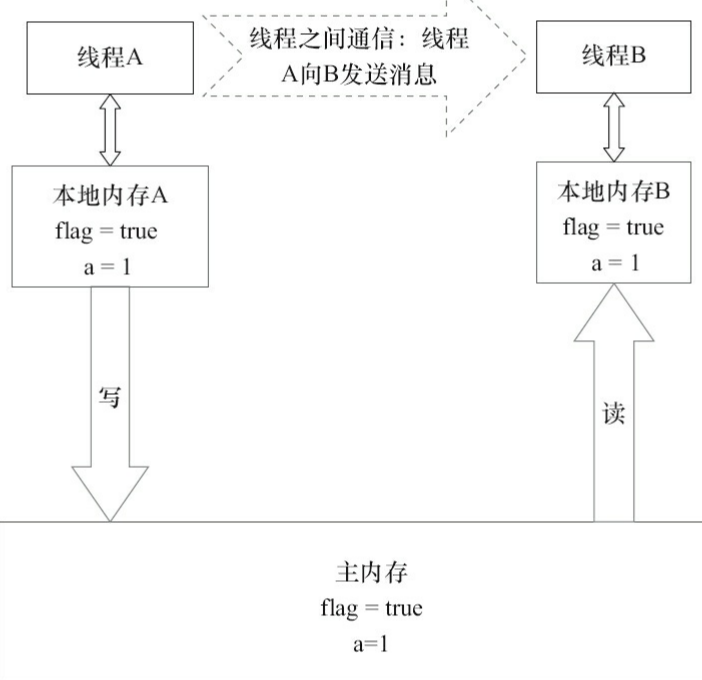
### Volatile写-读内存语义

Volatile写内存语义：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。



线程A在**写flag变量后**，本地内存A中被线程A更新过的两个共享变量的值flag=true，a = 1被刷新到主内存中。此时，本地内存A和主内存的共享变量是一致的

Volatile读的内存语义：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读取共享变量。



当线程B执行if（flag）时，需要读取flag这个volatile变量，此时本地内存B包含的值已经置为无效，线程B必须从主存中读取共享变量。

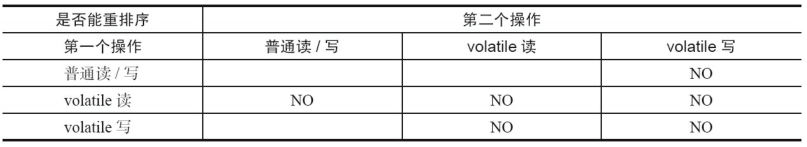
总结：

①线程A**写一个volatile变量**，实际上是线程A接下来要**读这个volatile变量的某个线程**发出了（我要对共享变量**进行了修改**）的消息

②线程B读一个volatile变量，实际上是线程B接受了之前某线程发出的（在我第二次写之前，你先把这次修改的内容拿去）的消息

③线程A写一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息

### Volatile内存语义的实现



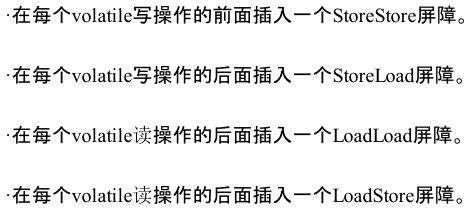
①当第一个操作时普通变量的读/写时，如果第二个操作为volatile写，则不能重排序

②当第二个操作是volatile写时，不管第一个操作是什么都不能重排序

③当第一个操作是volatile读时，无论第二个操作是啥，都不能重排序。

④当第一个操作是volatile写时，第二个操作是volatile读或写时，不能重排序

为了实现volatile语义，JMM采用保守策略，插入内存屏障

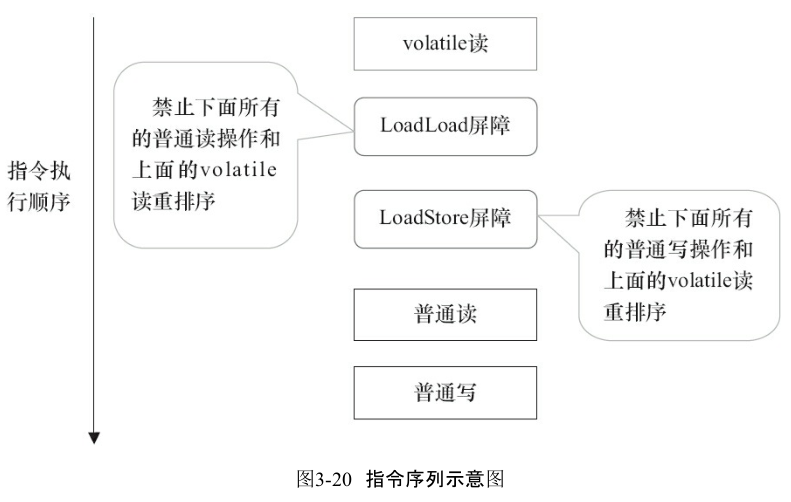


StoreStore屏障可以保证在volatile写之前，其前面的所有普通写操作已经对任意处理器可见了。这是因为StoreStore屏障将保障上面所有的普通写在volatile写之前刷新到主内存。

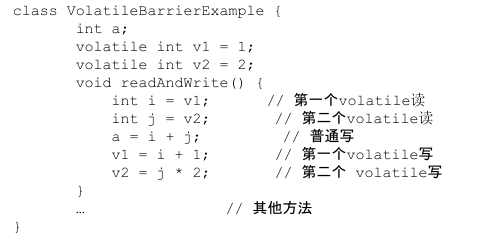
volatile写后面的StoreLoad屏障，JMM使用保守策略，要么每个volatile写后面，或者每个volatile读前面插入storeload屏障。最终采取了volatile写的后面插入一个StoreLoad屏障

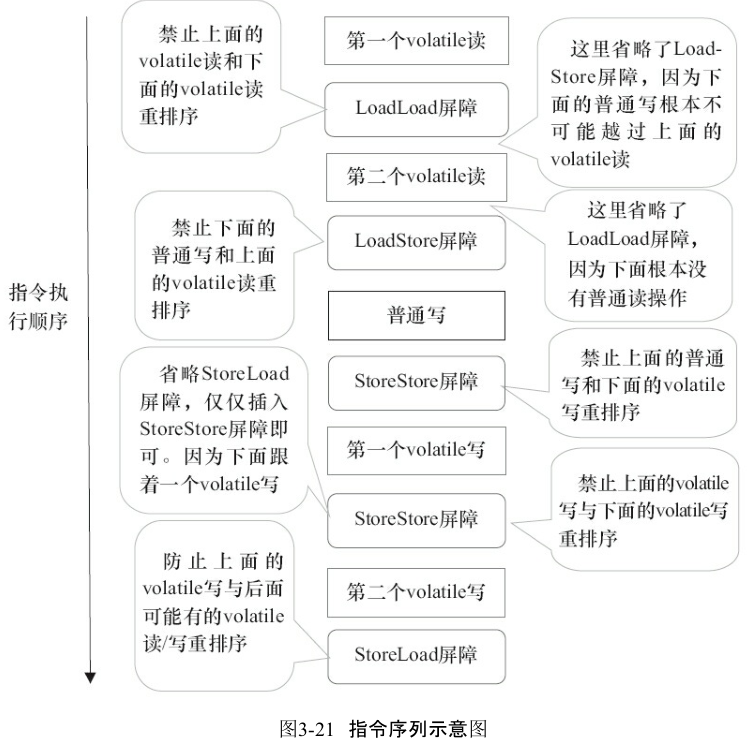


因为volatile写-读内存语义的常见使用模式是：**一个写线程写volatile变量**，**多个读线程**读同一个**volatile变量**。当读线程的数量**大大超过**写线程时，选择在volatile写之后插入StoreLoad屏障，能够让写线程对**volatile变量的修改刷新到内存**，对其他线程的读可见，这样使得效率和正确性得到提升



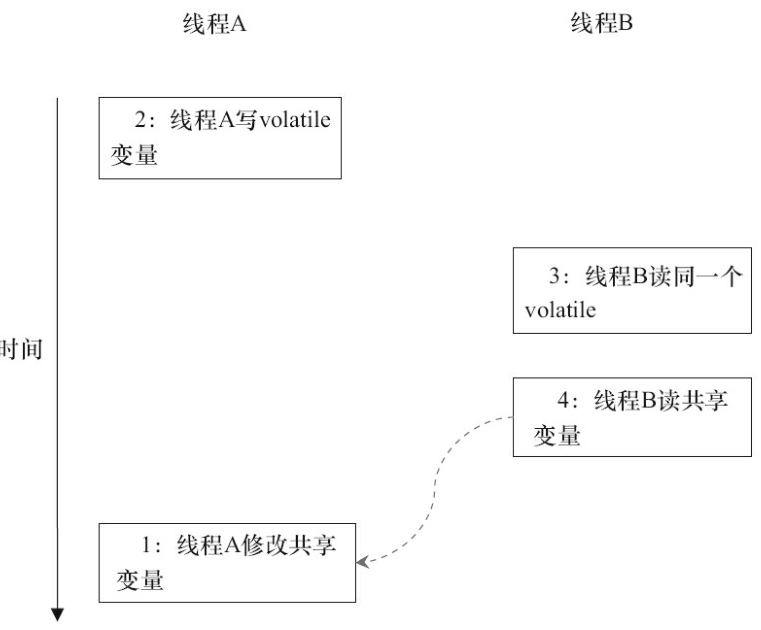
LoadLoad屏障用来禁止处理器把上面的volatile读与下面的普通读重排序。LoadStore屏障用来禁止处理器把上面的volatile读与下面的普通写重排序。





注意，最后的StoreLoad屏障不能省略。因为第二个volatile写之后，方法立即return。此时编译器可能无法准确断定后面是否会有volatile读或写，为了安全起见，编译器通常会在这里插入一个StoreLoad屏障。

### JSR-133为什么要增强volatile内存语义



在旧的内存模型中，当1和2之间没有数据依赖关系时，1和2之间就可能被重排序（3和4类似）。其结果就是：读线程B执行4时，不一定能看到写线程A在执行1时对共享变量的修改。旧的内存模型中，volatile的写-读并没有像锁的释放-获得那样的内存语义。为了提供一种比锁更轻量级的线程之间通信的机制。因此对其进行增强，确保volatile的写-读和锁的释放-获取具有相同的内存语义。编译器和处理器的重排序只要能破坏volatile的内存语义，就会被编译器和处理器的内存屏障插入策略禁止

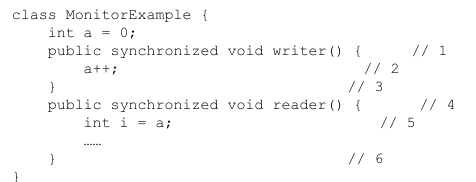
**由于volatile仅仅保证对单个volatile变量的读/写具有原子性**，而锁的互斥执行的特性可以确保对**整个临界区代码的执行**具有原子性。在功能上，锁比volatile更强大；在可伸缩性和执行性能上，volatile更有优势。

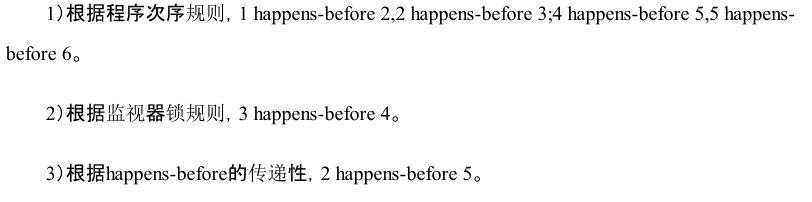
## 锁的内存语义

众所周知，锁可以让临界区互斥执行。这里将介绍锁的另一个同样重要，但常常被忽略的功能：锁的内存语义。

### 锁的释放-获取建立的happens-before关系

锁是Java并发编程中最重要的同步机制，，锁除了让临界区互斥执行外，还可以让**释放锁的线程**向**获取同一个锁的线程发送消息**



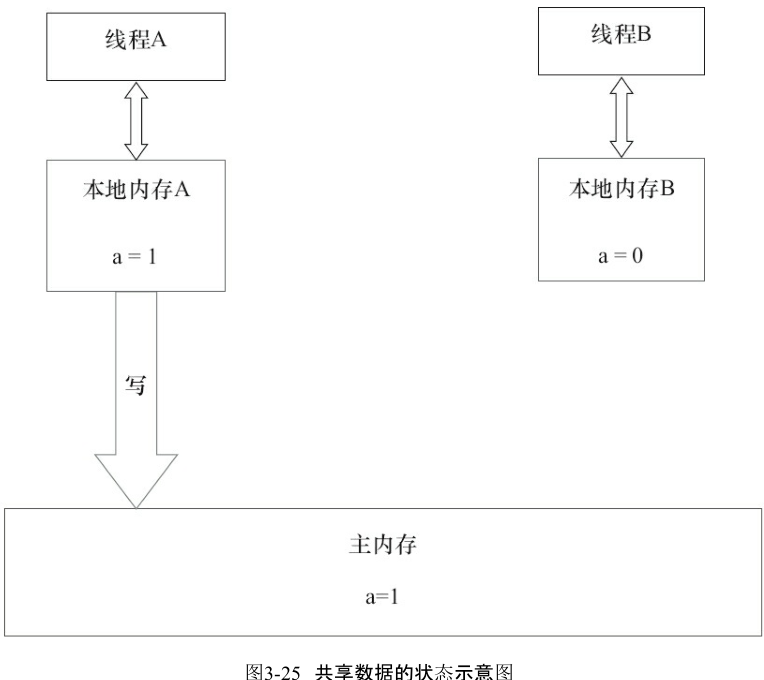




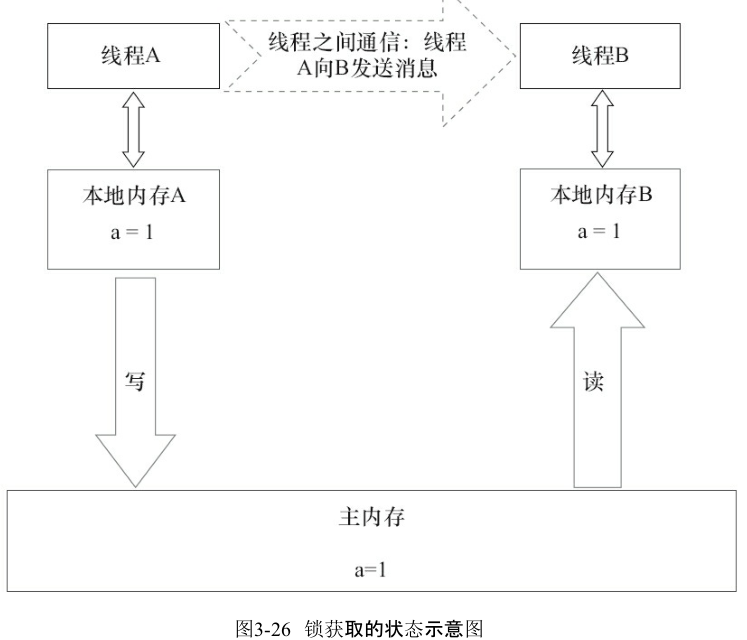
上图表示，线程A释放锁之后，线程B获取同一个锁，又因为**2 happens-before 5**，因此，线程A在**释放锁之前**所有**可见的共享变量**，在线程B**获取同一个锁之后**，**(**一定要获取，未获取的话，共享变量还在临界区，仍被锁住，无法方法访问，**)**立即变得对**B线程可见**。

### 锁的释放和获取的内存语义

当线程A**释放锁**时，JMM会把该线程对应的**本地内存中的共享变量刷新到主内存**中。以上面的MonitorExample程序为例，A线程释放锁后，共享数据的状态示意图如图3-25所示。



当**线程B获取锁**时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的**临界区代码必须从主内存中读取共享变量**。



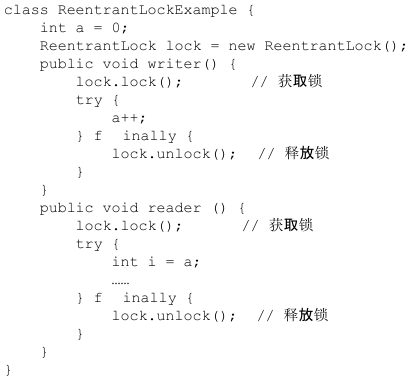
对比**锁释放-获取**的内存语义与**volatile写-读**的内存语义可以看出：**锁释放与volatile写**有相同的内存语义；**锁获取与volatile读**有相同的内存语义。

线程A释放一个锁，实质上是线程A对将要获取这个锁的某个线程发出了（我对共享变量进行修改了的）消息。

线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在我再次修改前，你去内存读取我修改的东西的）消息。

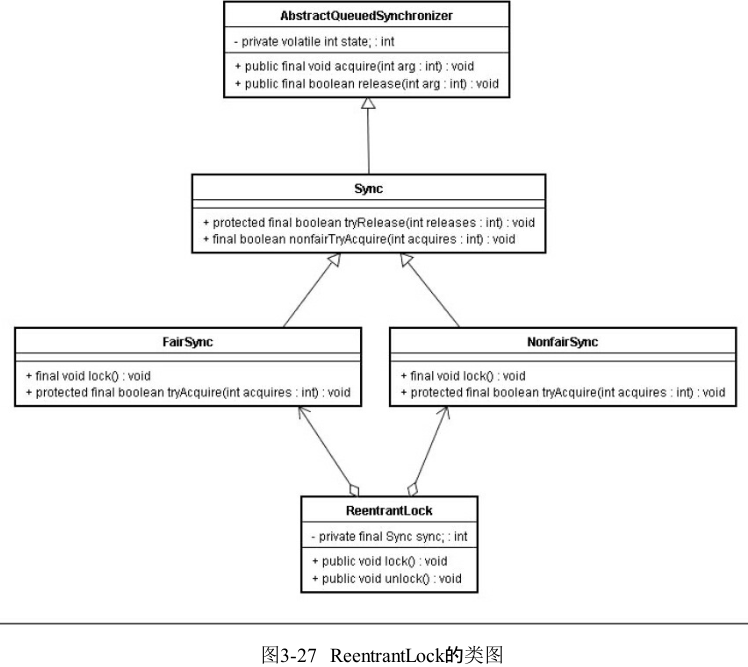
### 锁内存语义的实现

借助ReentrantLock的源代码，来分析锁内存语义的具体实现机制



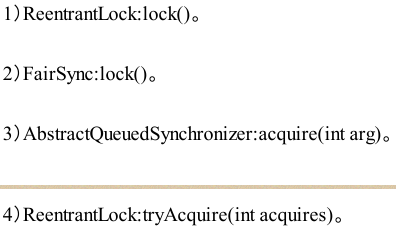
在ReentrantLock中，调用lock()方法获取锁；调用unlock()方法释放锁。

ReentrantLock的实现依赖于J**ava同步器框架**AbstractQueuedSynchronizer（抽象队列同步器）（本文简称之为AQS）。**AQS使用一个整型的volatile变量（命名为state）来维护同步状态**，马上我们会看到，这个volatile变量是ReentrantLock内存语义实现的关键。



ReentrantLock分为**公平锁**和**非公平锁**

使用**公平锁**时，加锁方法lock()调用轨迹如下。

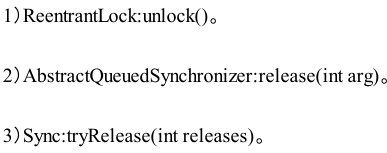


在第四步真正开始加锁，使用的fairSync内部类里面的tryAcquire方法，下面是该方法的源代码。

|  |
| --- |
| **protected** **final** **boolean** tryAcquire(**int** acquires) {  **final** Thread current = Thread.*currentThread*();  **int** c = getState();//**读取volatile变量state**  **如果c==0，说明锁没有被任何线程占用，然后继续进行判断：**  **调用**hasQueuedPredecessors判断是否有其他线程比当前线程等待时间长，，  如果有，不进入；如果没有则获取该锁，使用CAS设置锁的状态，并且锁的拥有者为当前线程  **if** (c == 0) {  **if** (!hasQueuedPredecessors() &&  compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  **return** **true**;  }  }  **如果c！=0，表明该锁已经线程占有，则判断该锁是不是被当前线程占有；如果是就设置state，返回true；否则返回false**  **else** **if** (current == getExclusiveOwnerThread()) {  **int** nextc = c + acquires;  **if** (nextc < 0)  **throw** **new** Error("Maximum lock count exceeded");  setState(nextc);  **return** **true**;  }  **return** **false**;  }  **查询是否有任何线程等待获取的时间长于当前线程**。如果h==t，则说明只有一个线程就是当前线程，如果h！=t说明至少两个以上线程在队列中，如果h.next为空，说明只有一个线程，同上，如果不为空，则判断h.next是否是当前线程，  **public** **final** **boolean** hasQueuedPredecessors() {  Node t = tail;  Node h = head;  Node s;  **return** h != t &&  ((s = h.next) == **null** || s.thread != Thread.*currentThread*());  } |

从上面源代码中我们可以看出，加锁方法首先读volatile变量state

在使用公平锁时，解锁方法unlock()调用轨迹如下。

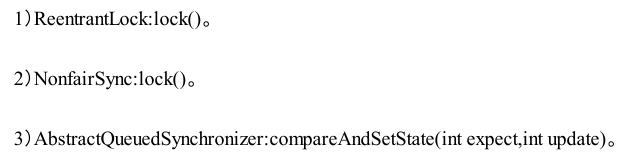


第三步真正开始释放锁，下面是sync中tryRelease方法的源代码

|  |
| --- |
| **protected** **final** **boolean** tryRelease(**int** releases) {  因为有锁重入的存在，所以在这里要递减锁计数  **int** c = getState() - releases;  必须由当前拥有锁的线程进行解锁操作，如果当前线程不是拥有锁线程就抛异常  **if** (Thread.*currentThread*() != getExclusiveOwnerThread())  **throw** **new** IllegalMonitorStateException();  标识锁是否已经空闲  **boolean** free = **false**;  如果c为0，表示锁已经完全被释放，则将锁拥有者的线程置为null  **if** (c == 0) {  free = **true**;  setExclusiveOwnerThread(**null**);  }  设定当前的state  setState(c);  **return** free;  } |

从上面的源代码可以看出，在释放锁的最后写volatile变量state；公平锁在释放锁的最后写volatile变量state，在获取锁时首先读这个volatile变量。线程释放锁之前，以及在写volatile变量之前（因为一旦写完volatile，就会引起刷新内存的操作）的共享变量，在另一个线程获取锁之后，以及读取同一个volatile变量后（因为读取volatile变量后，必然得到的是主存的最新数据）那些共享变量立即对获取锁的线程可见

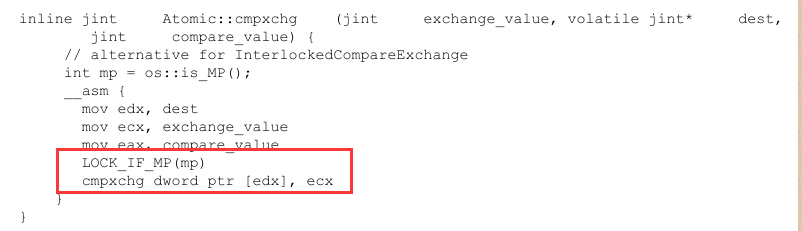
使用非公平锁时，加锁方法lock()调用轨迹如下。



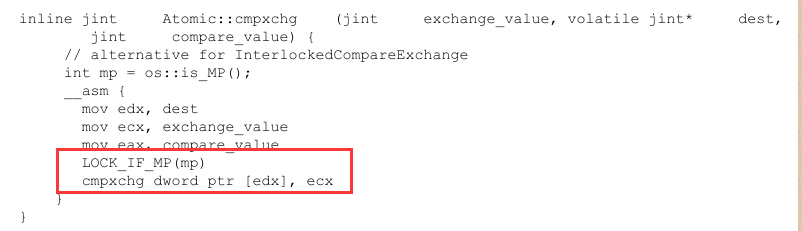
在第3步真正开始加锁

|  |
| --- |
| **final** **void** lock() {  **if** (compareAndSetState(0, 1))  setExclusiveOwnerThread(Thread.*currentThread*());  **else**  **如果CAS失败说明，锁已经被占用，则调用AQS的acquire方法进行锁重入的判断，再次**  **再次尝试获取锁，入队等操作**  acquire(1);  }  以原子操作更新state变量，从0变为1，CAS操作，如果当前状态值等于预期值，则以原子方式将同步状态设置为给定更新值  **protected** **final** **boolean** compareAndSetState(**int** expect, **int** update) {  // See below for intrinsics setup to support this  **return** ***unsafe***.compareAndSwapInt(**this**, ***stateOffset***, expect, update);  }  释放锁与公平锁相同 |

CAS如何**同时具有volatile读和volatile写**的内存语义：



IntelX86处理器中的源码



程序会根据当前处理器是多处理器还是单处理决定是否为cmpxchg加上lock前缀。

Intel手册对lock前缀的说明如下：

①确保对内存的读-改-写操作的原子性，以前的处理器会锁住总线，现在一般采用缓存锁定

②禁止该指令，与之前或之后的读写指令重排

③把写缓冲去的所有数据刷新到内存

第二和第三具有内存屏障效果，可以同时实现volatile的读写内存语义。

总结：

公平锁和非公平锁释放时，最后都要写一个volatile变量state

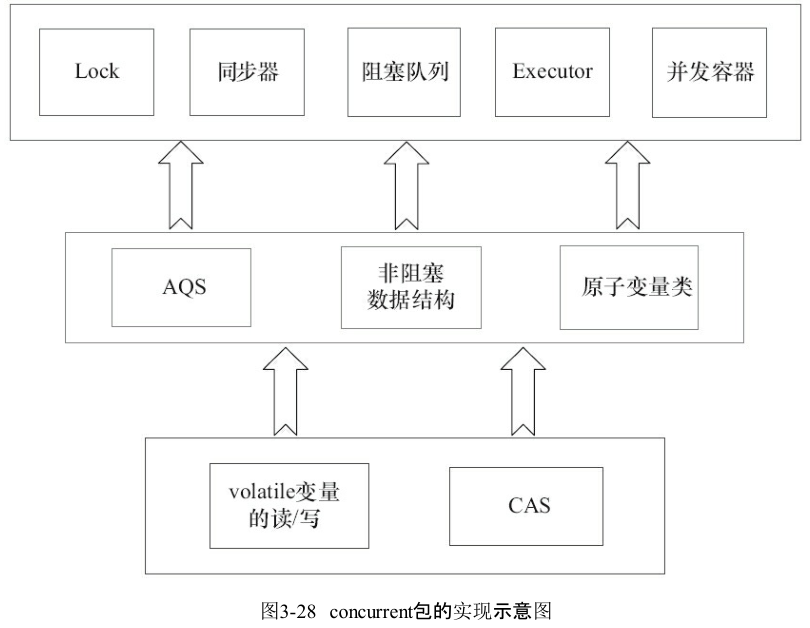
公平锁获取时，首先会读volatile变量

非公平锁获取时，首先会用CAS更新volatile变量，这个操作同时具有volatile读和volatile写的内存语义。

锁的释放和获取的实现，至少有下面两种方式：

①利用volatile变量的写-读具有的内存语义

②利用CAS所附带的volatile读和volatile写的内存语义



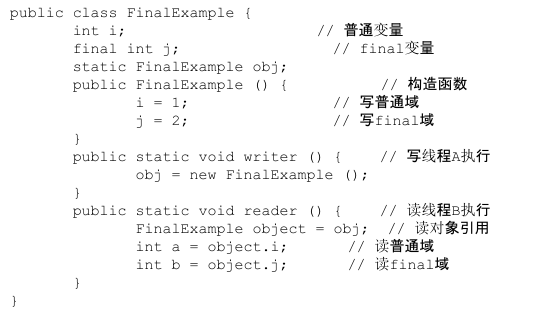
## final域的内存语义

### final域的重排序规则

编译器和处理器要遵守两个重排序规则：

1）在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

2）初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

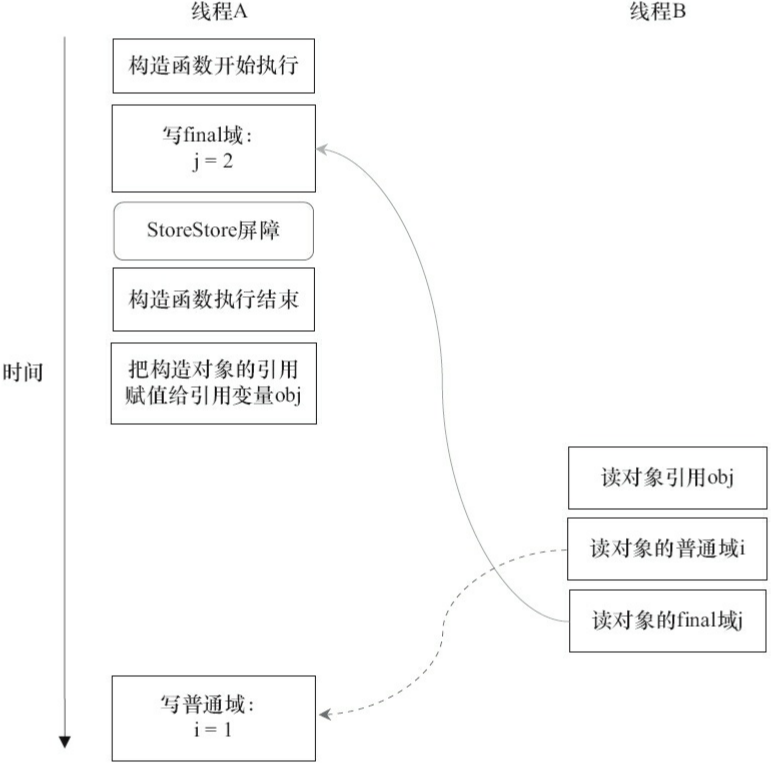


### 写final域的重排序规则

写final域的重排序规则禁止把final域的写重排序到构造函数之外，包含两个方面的内容：

1）JMM禁止编译器把final域的**写重排序到构造函数之外**。

2）编译器会在final域的写之后，构造函数return之前，**插入一个StoreStore屏障**。这个屏障禁止**处理器把final域的写重排序到构造函数之外**。

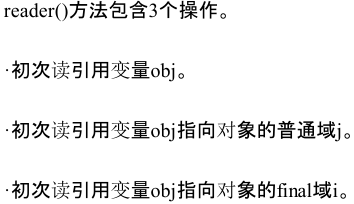


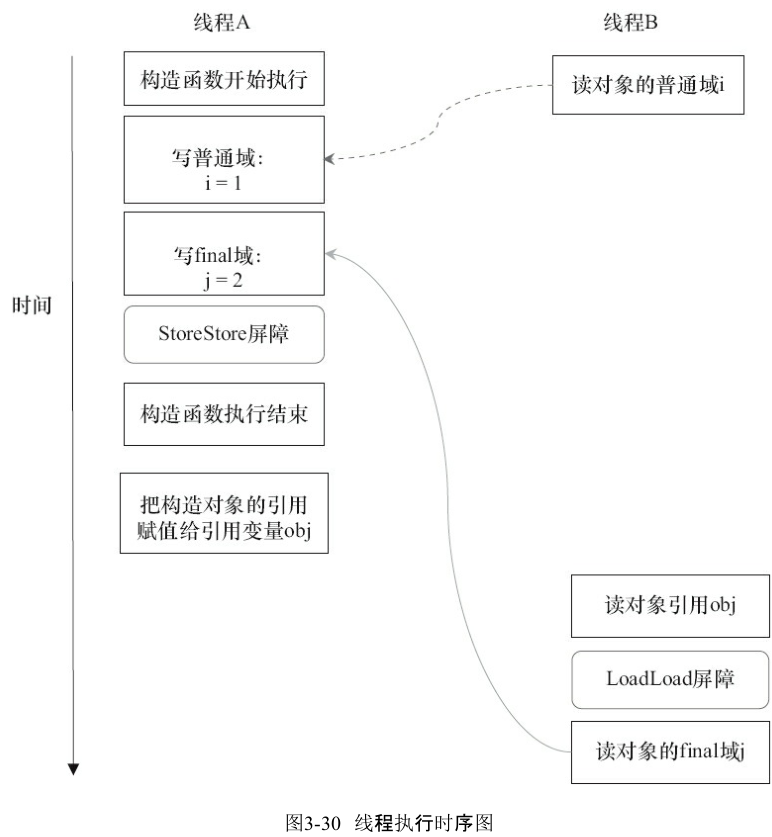
写final重排序的规则可以确保：在对象引用为任意线程可见之前，对象的final域已经被

正确初始化过了，**而普通域不具有这个保障**。

### 读final域的重排序规则

读final域的重排序规则是，在一个线程中，**初次读包含final对象的引用与初次读该对象包含的final域**，JMM禁止处理器重排序这两个操作（注意，这个规则仅仅针对处理器）。编译器会在读final域操作的前面插入一个**LoadLoad屏障**。

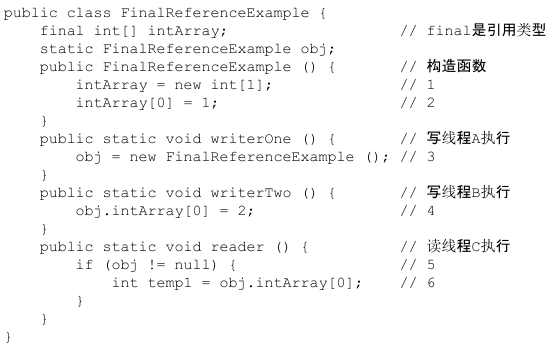




读final域的重排序规则可以确保：在读一个**对象的final域**之前，一定会先读包含**这个final**

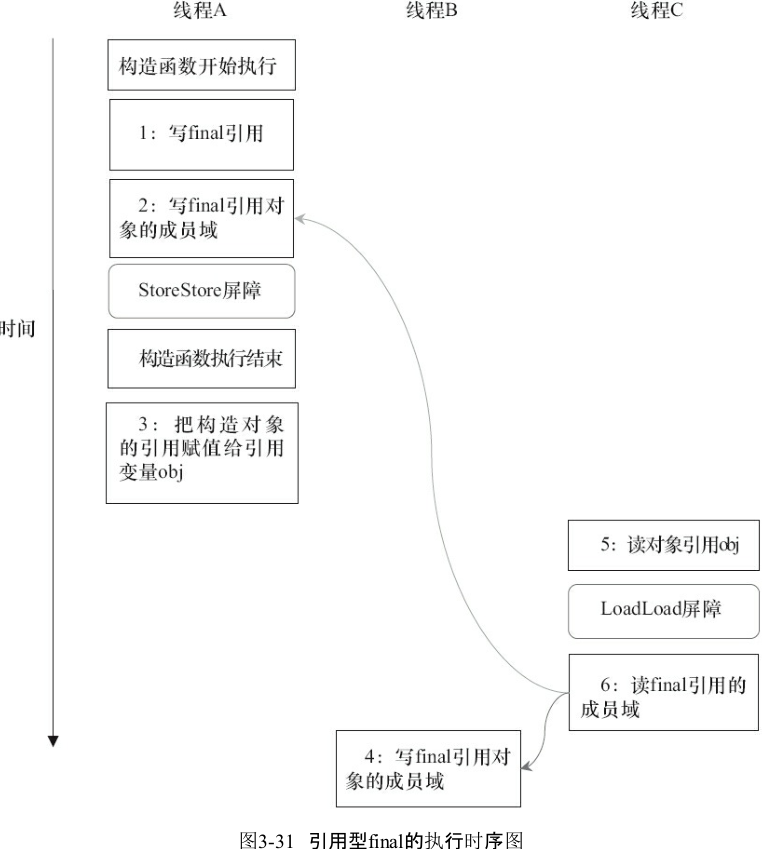
**域的对象的引用。如果该引用不为null，那么该对象的final域一定已经被初始化了。**

### Final域为引用类型



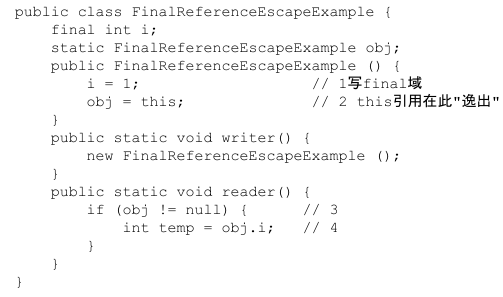
引用类型，**写final域的重排序**规则对编译器和处理器增加了如下约束：在构造函数内**对一个final引用的对象的成员域的写入**，与随后在**构造函数外把这个被构造对象的引用赋值给一个引用变量**，这两个操作之间不能重排序。

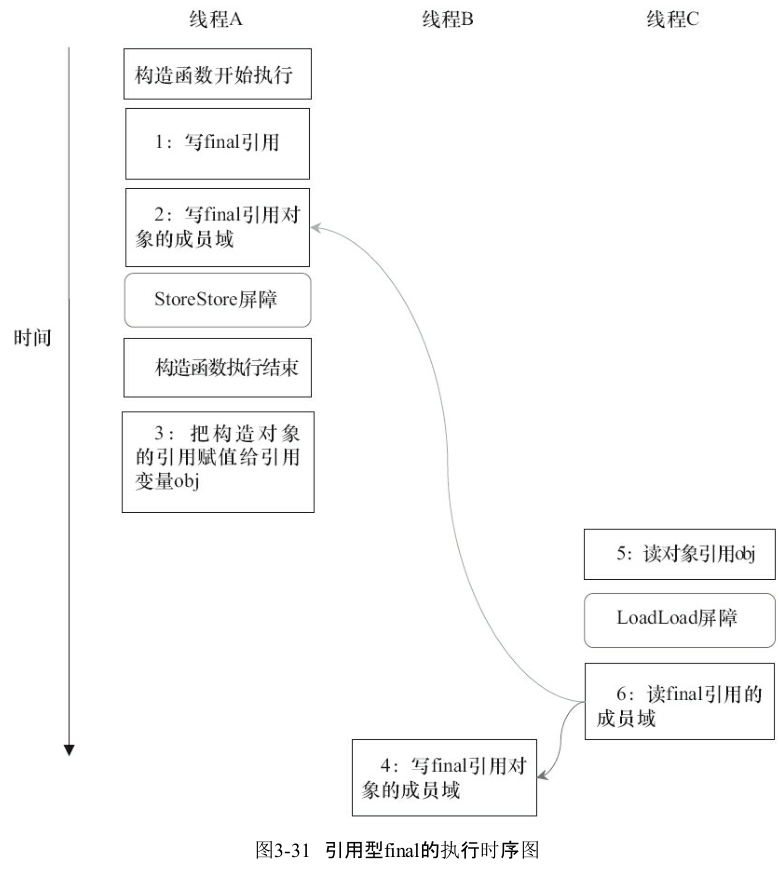
1是对final域的写入，2是对这个final域引用的对象的成员域的写入，3是把被构造的对象的引用赋值给某个引用变量。这里除了前面提到的1不能和3重排序外，2和3也不能重排序。

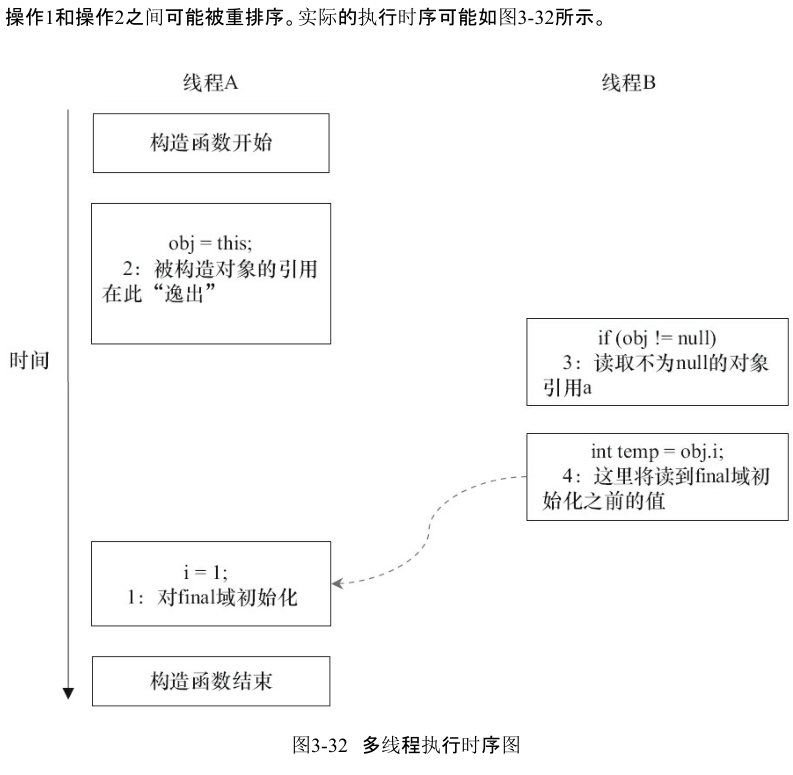


**JMM可以确保读线程C至少能看到写线程A在构造函数中对final引用对象的成员域的写入**。即C至少能看到数组下标0的值为1。JMM不保证线程B的写入对读线程C可见，因为写线程B和读线程C之间存在数据竞争，此时的执行结果不可预知。

### 为什么final引用不能从构造函数内“溢出”







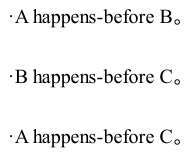
在构造函数返回前，被构造对象的引用不能为其他线程所见，因为此时的final域可能还没有被初始化。在构造函数返回后，任意线程都将保证能看到final域正确初始化之后的值。

## Happens-before

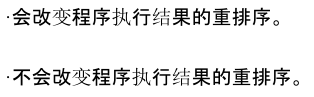
### JMM的设计

JSR-133专家组在设计JMM时的核心目标就是找到一个好的平衡点：一方面，要为程序员提供足够强的内存可见性保证；另一方面，对编译器和处理器的限制要尽可能地放松。





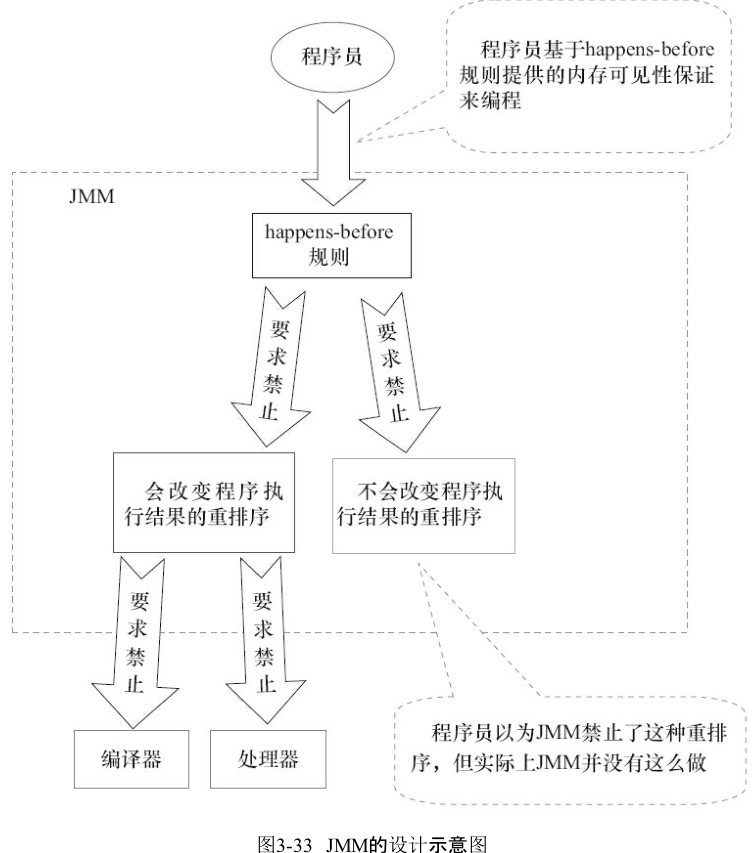
2和3是必需的，但1是不必要的。因此，JMM把happens-before要求禁止的重排序分为了下面两类。



·对于会改变程序执行结果的重排序，JMM要求**编译器和处理器必须禁止这种重排序**。

·对于不会改变程序执行结果的重排序，JMM对**编译器和处理器不做要求**（JMM允许这种

重排序）。



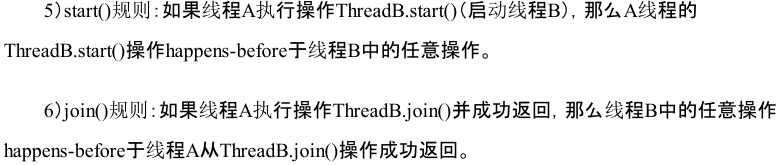
**只要不改变程序的执行结果**（指的是单线程程序和正确同步的多线程程序），**编译器和处理器怎么优化都行**。例如：一个锁只会被单个线程访问，那么这个锁可以被消除。一个volatile变量只会被单个线程访问，那么编译器可以把这个volatile变量当作一个普通变量来对待

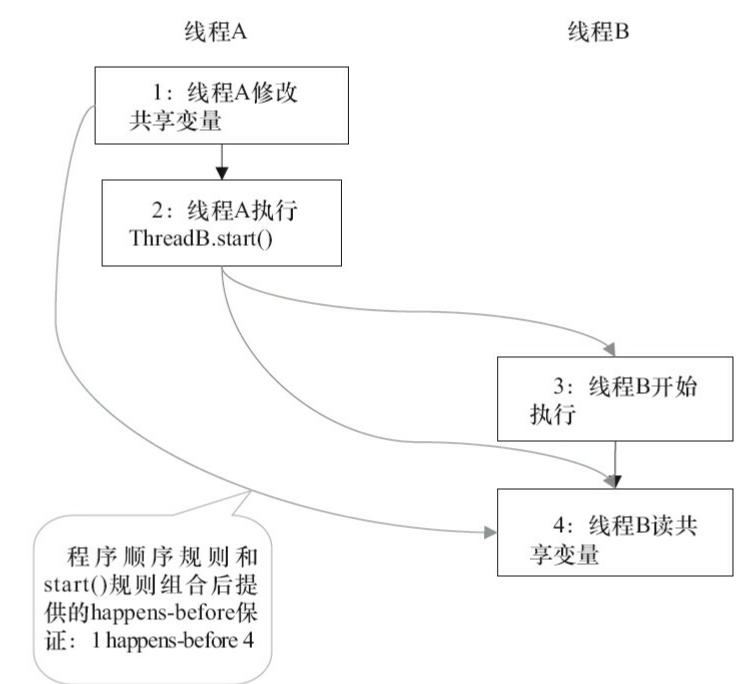
1）如果**一个操作happens-before另一个操作**，那么第一个操作的**执行结果**将对第二个操作**可见**，而且第一个操作的**执行顺序**排在第二个操作**之前**。

2）两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的**执行结果**，与按happens-before关系来执行的**结果一致**，那么这种重排序并不非法（也就是说，JMM允许这种重排序）。

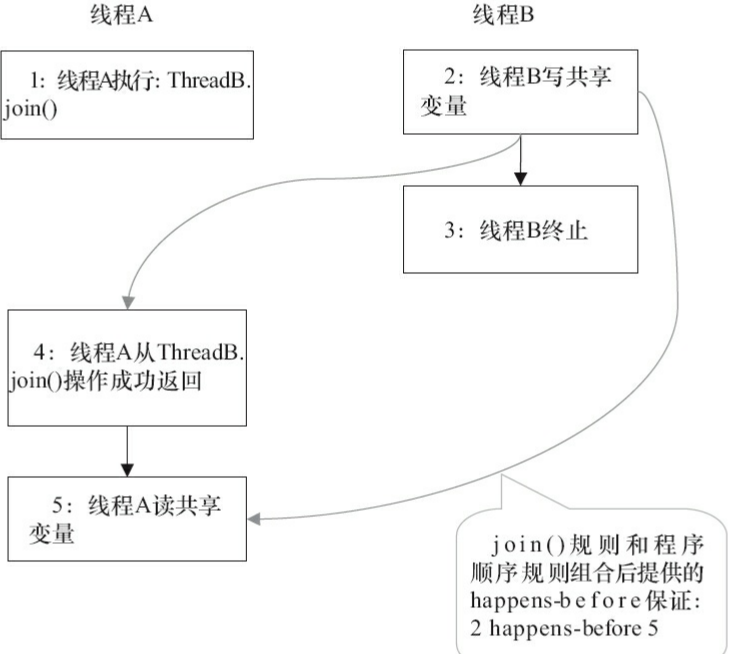
### Happens-before的规则

除了之前提到的四种；还有如下两种





1 happens-before 2由程序顺序规则产生。2 happens-before 4由start()规则产生。根据传递性，将有1 happens-before 4。这实意味着，线程A在执行**ThreadB.start()之前对共享变量所做的修改**，接下来在**线程B开始执行后都将确保对线程B可见**。



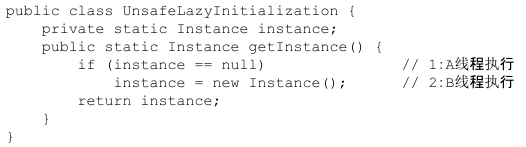
2 happens-before 4由join()规则产生；4 happens-before 5由程序顺序规则产生。根据传递性规则，将有2 happens-before 5。这意味着，**线程A执行操作ThreadB.join()并成功返回后**，**线程B中的任意操作**都将对线程A可见。

## 双重检查锁定与延迟初始化

在Java多线程程序中，有时候需要采用延迟初始化来降低初始化类和创建对象的开销。双重检查锁定是常见的延迟初始化技术，但它是一个错误的用法。后面进行分析

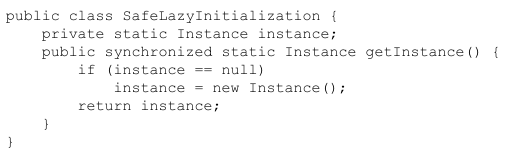
### 双重检查锁定的由来

非线程安全的延迟初始化对象的实例代码。



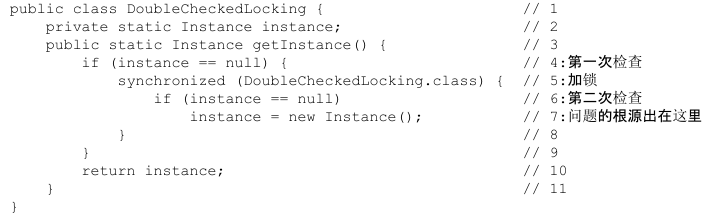
A线程执行代码1的同时，B线程执行代码2，此时，线程A可能看到instance引用的对象还没有初始化。

对getInstance方法做同步处理实现线程安全的延迟初始化。



对getInstance做了同步处理，synchronized将导致性能开销，如果getInstance方法被多个线程频繁调用，将会导致程序执行性能下降。反之，如果getInstance方法不会被频繁调用，那么这个延迟初始化方案将能提供令人满意的性能。

**双重检查锁定**：



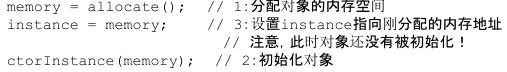
如果第一次检查不为null，那么就不需要执行下面的加锁和初始化过程，表面看起来完美实际上存在问题

### 问题的根源

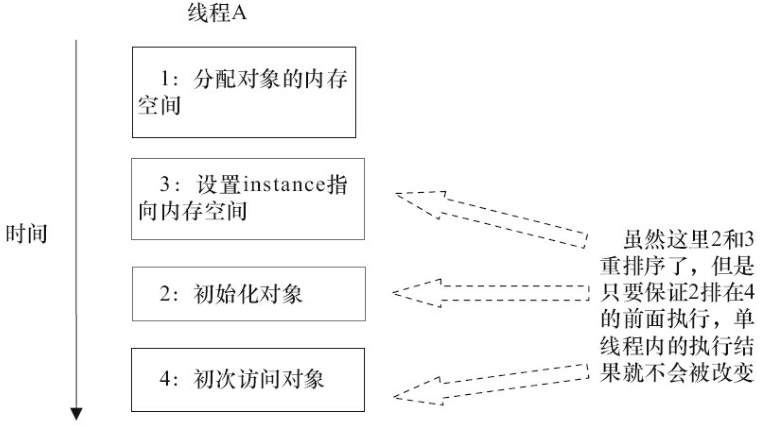
instance=new Singleton();可以分为如下代码

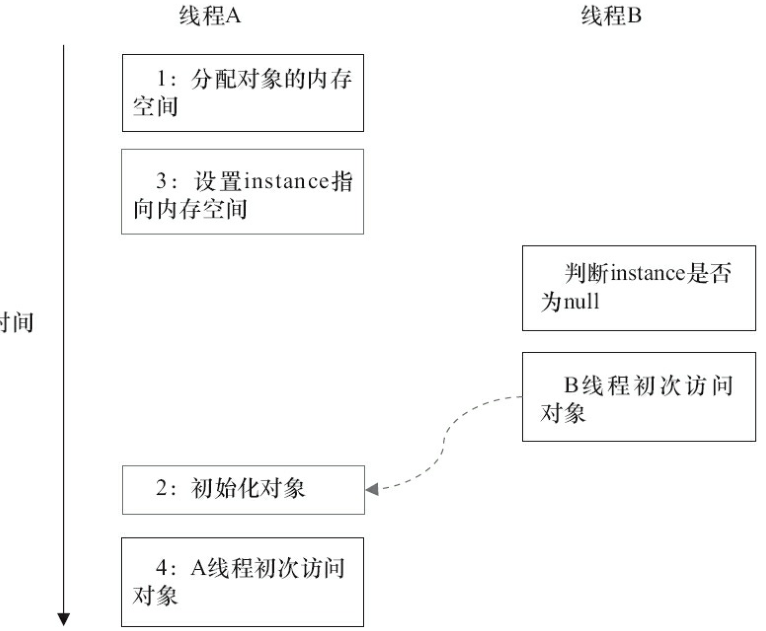


**2,3之间可能发生重排；**



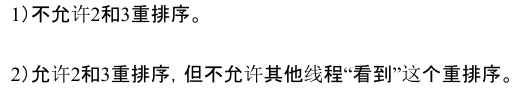
intra-thread semantics保证重排序不会改变**单线程内的程序执行结果**。上面3行伪代码的2和3之间虽然被重排序了，但这个重排序并不会违反intra-thread semantics。



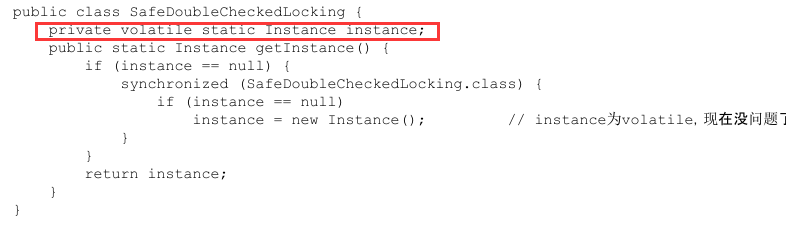


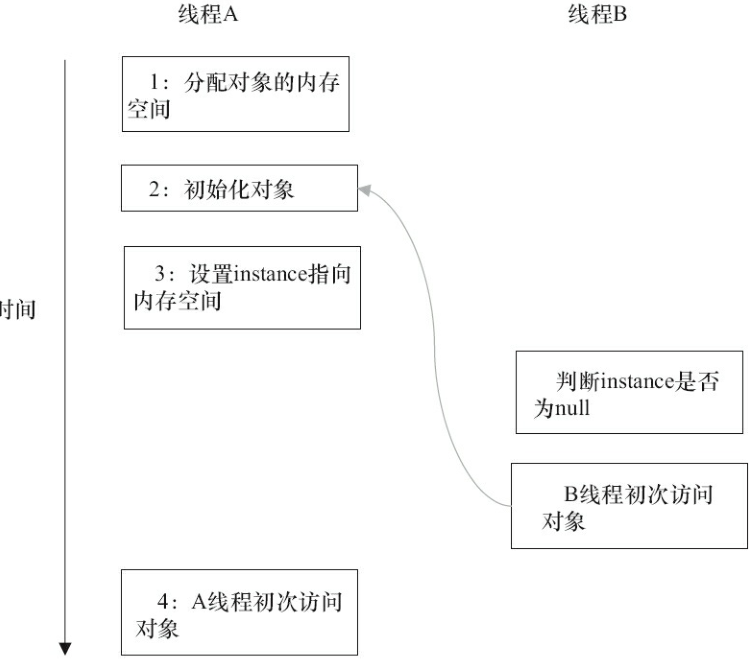
如果在双重检查锁定示例代码中第7行instance = new Singleton（）；如果按照上面的时序执行，b线程将会看到一个没有被初始化的对象。

解决这个问题：



### 基于volatile的解决方案





这个方案本质上是通过禁止图3-39中的2和3之间的重排序，来保证线程安全的延迟初始

化。

### 基于类初始化的解决方案

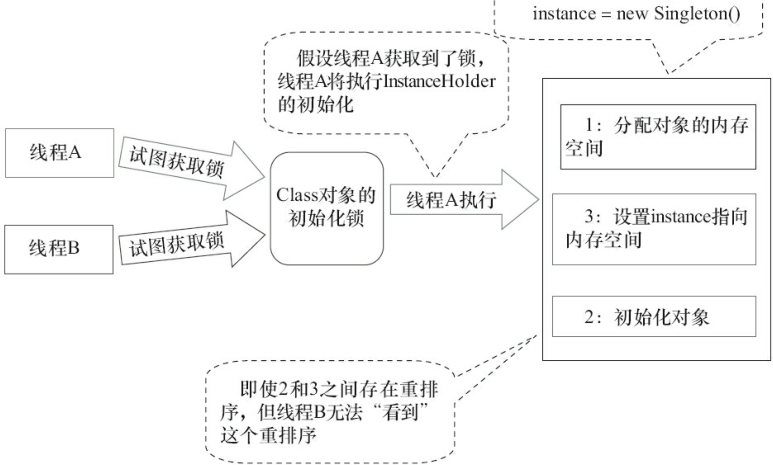
JVM在类的初始化阶段（即在Class被加载后，且被线程使用之前），会执行类的初始化。在执行类的初始化期间，JVM会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

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（这个方案被称之为

Initialization On Demand Holder idiom）。

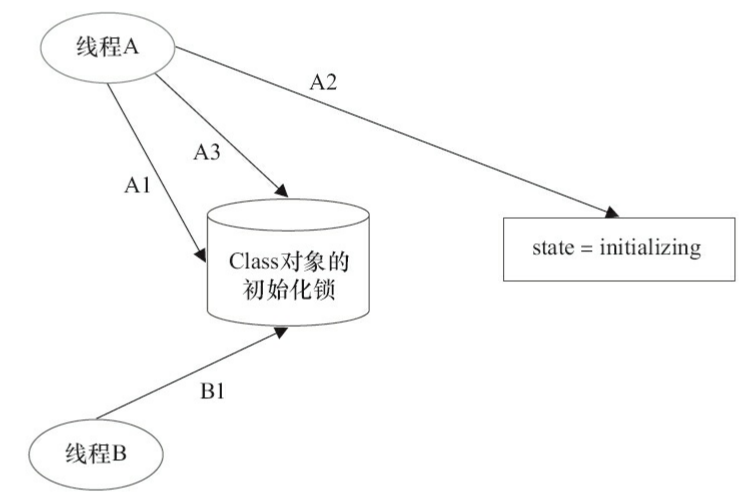


由于主动使用而引起的类的初始化。



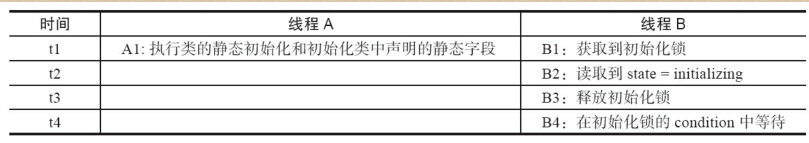
java初始化一个类或接口的处理过程如下（这里对类初始化处理过程的说明，省略了与本文无关的部分）

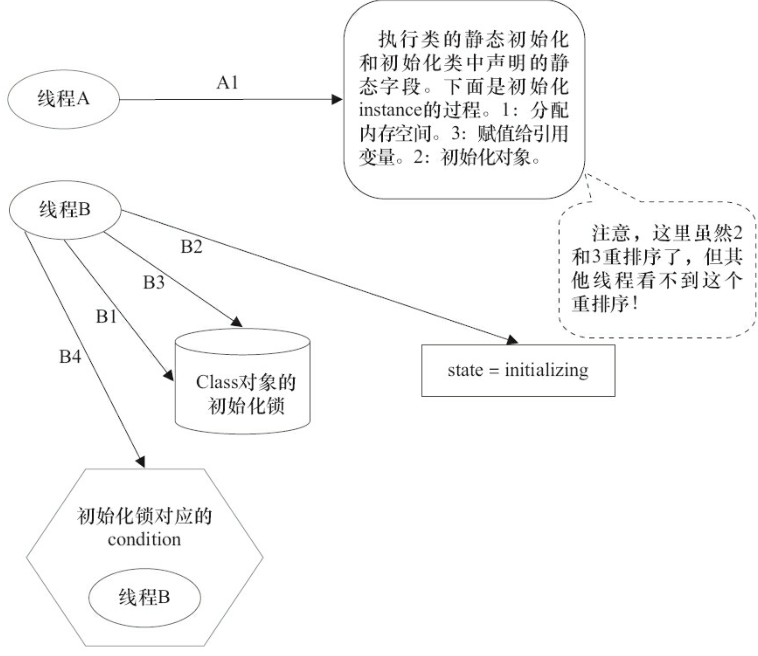
**第1阶段**：通过在**Class对象上同步**（即获取Class对象的初始化锁），来**控制类或接口的初始化**。这个**获取锁的线程会一直等待，直到当前线程能够获取到这个初始化锁**



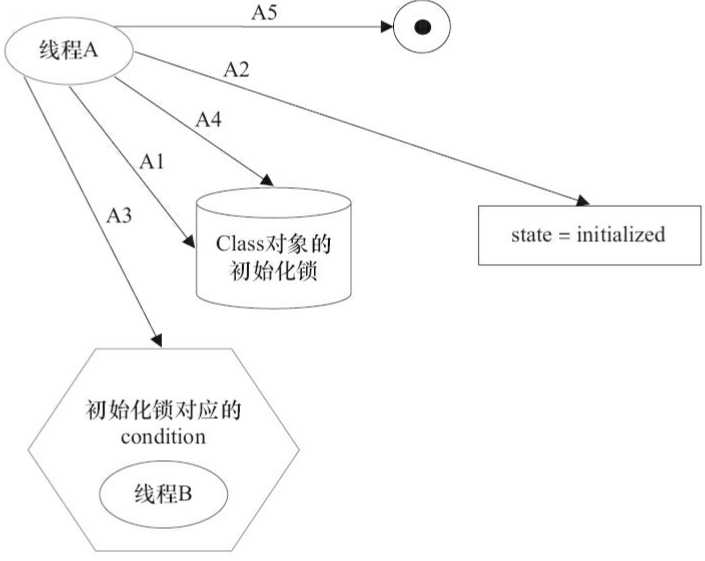


**第2阶段**：线程A执行**类的初始化**，同时线程B在**初始化锁对应的condition上等待**。





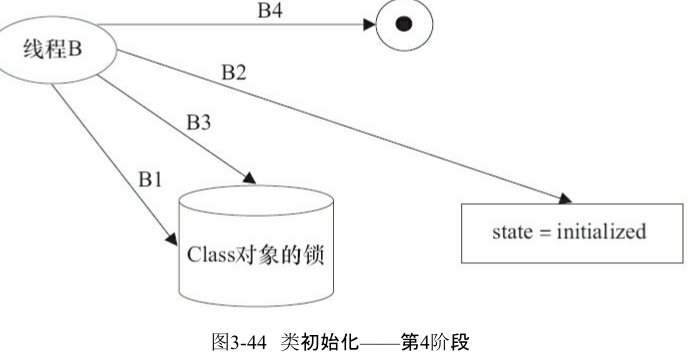
第3阶段：线程A设置**state=initialized**，然后**唤醒在condition中等待的所有线程**。



A1：获取初始化锁；A2：设置state=initialized，A3：唤醒在condition中等待的所有线程；

A4：释放初始化锁；A5：线程A的初始化剁成处理完成。

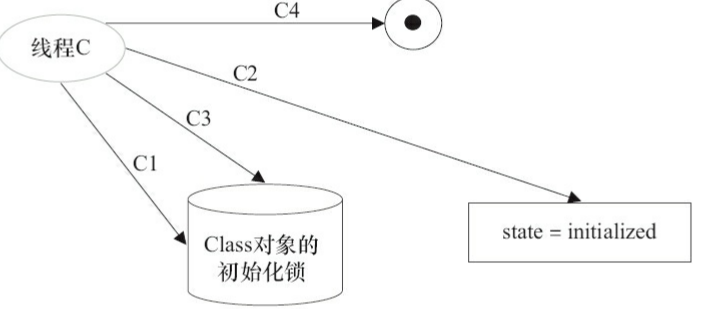
**第4阶段**：**线程B结束类的初始化处理**。





线程A**执行类的初始化时的写入操作**（执行类的静态初始化和初始化类中声明的静态字段,即上述程序中的new Instance()），线程B一定能看到。

**第5阶段：线程C执行类的初始化的处理。**





线程A执行类的初始化时的写入操作，线程C一定能看

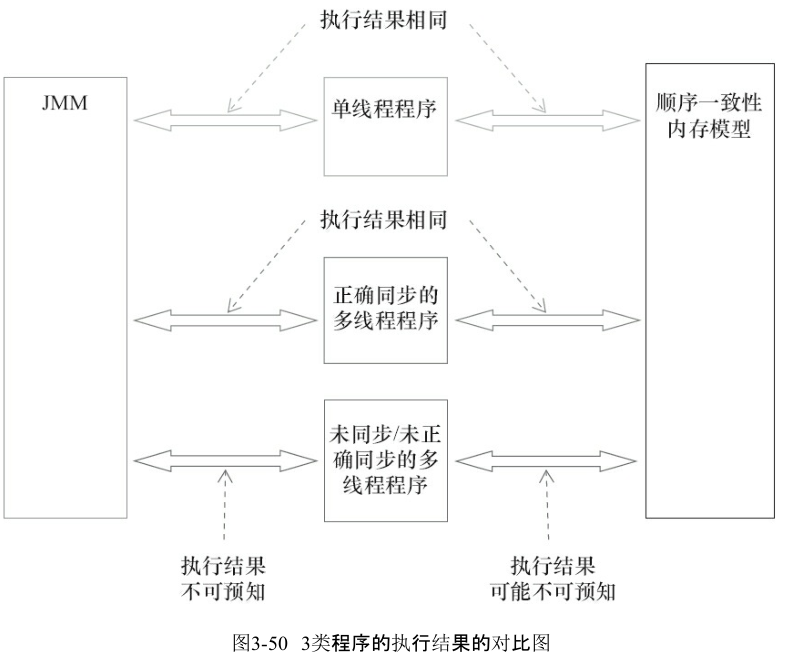
## Java内存模型综述

### JMM的内存可见性保证

·**单线程程序**。单线程程序不会出现内存可见性问题。

·**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）。

·**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（0、null、false）。



### JSR-133对旧内存模型的修补

·增强volatile的内存语义。**旧内存模型**允许**volatile变量与普通变量重排序**。JSR-133严格**限制volatile变量与普通变量的重排序**，**使volatile的写-读和锁的释放-获取具有相同的内存语义。**

·增强final的内存语义。在旧内存模型中，**多次读取同一个final变量的值可能会不相同**。为此，JSR-133为final增加了两个重排序规则。在保证**final引用不会从构造函数内逸出的情况下，final具有了初始化安全性。**