# JAVA并发编程

## 第二章 Java并发机制底层实现原理

### Volatile的应用：

#### volatile的定义：

《JAVA语言规范3rd》：Java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。

#### volatile的实现：

java代码：

instance = new Singleton(); //instance是volatile变量。

转化为汇编码：

0x……：movb $0x0,0x……(%esi)；0x……：**lock** addl $0x0,(%esp)；

IA-32手册分析此汇编码，Lock前缀在多核处理器引发2件事情：

1. 将当前处理器缓存行的数据写回到内存。
2. 这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

### Sychronized的实现原理与应用

#### 实现同步的基础

JAVA中每一个对象都可以作为锁，具体表现为3种：

* 对于普通同步方法，锁是当前实例对象
* 对于静态同步方法，锁是当前类的Class对象
* 对于同步方法块，锁是sychronized括号里配置的对象。

JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，但两者的实现细节不一样。

#### JAVA对象头

Sychronized用的锁是存在Java对象头里的，若对象是数组，则虚拟机用3words存储，否则用2字宽。

表2-1 JAVA对象头的长度

|  |  |  |
| --- | --- | --- |
| 长度 | 内容 | 说明 |
| 32/64bit | Mark Word | 存储对象的hashCode、分代年龄、锁标记位 |
| 32/64bit | Class Metadata Address | 存储到对象类型数据的指针 |
| 32/64bit | Array length | 数组长度（如果对象是数组） |

轻量级锁00

重量级10

GC标记11

偏向锁01

#### 偏向锁

大多情况锁不仅不存在多线程竞争，而且总由同一线程多次获得，引入偏向锁。

一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储偏向的线程ID，以后该线程进出同步块不需要进行CAS操作来加锁或解锁，只需测试Mark Word是否存储着指向当前线程的偏向锁，测试失败则再测试偏向锁标识是否置1，没有设置则用CAS竞争锁，设置了则尝试用CAS将对象头的偏向锁指向当前线程。

偏向锁的撤销：当其他线程尝试竞争时，等到全局安全点，首先暂停拥有偏向锁的线程，然后检查它是否活着，若不活动，则将对象头设置成无锁状态，若活着，拥有偏向锁的栈会执行，遍历偏向对象的锁记录，栈中的锁记录和对象头Mark Word要么重新偏向于其他线程，要么恢复到无锁或标记对象不适合作为偏向锁，最后唤醒暂停的线程。

#### 轻量级锁

加锁：线程执行同步块之前，JVM在当前线程的栈帧中创建用于存储锁记录的空间，将对象头中的Mark Word复制到锁记录中(Displaced Mark Word)，然后线程尝试使用CAS将Mark Word替换为指向锁记录的指针，若成功，则获得锁，若失败，则通过自旋来获取。

解锁：使用原子的CAS操作将Displaced Mark Word替换回对象头，若成功，表示没有竞争，若失败，表示存在竞争（有其他线程想同时CAS替换Mark Word），锁升级成重量级锁。

锁升级后不会恢复成轻量级锁。

其他线程试图获取重量级锁时会被直接阻塞。持有重量级锁的线程释放锁时会唤醒这些线程。

#### 锁的优缺点对比

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 场景 |
| 偏向锁 | 加锁和解锁不需要额外的开销，和执行非同步方法相比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗 | 适用于只有一个线程访问同步块的场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到竞争的线程，使用自旋会消耗CPU | 追求响应时间  同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不消耗CPU | 线程阻塞，响应时间缓慢。 | 追求吞吐量，同步块执行速度较长。 |

### 原子操作的实现原理、CAS的三大问题

#### 使用总线锁保证原子性

使用处理器提供的一个LOCK #信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，该处理器可独占共享内存。开销较大。

#### 使用缓存锁保证原子性

内存区域如果被缓存在处理器的缓存行中，并且Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言LOCK#信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效。

两种不使用该方法的情况：1、当操作的数据不能被缓存在处理器内部，或者跨多个缓存行。会调用总线锁定。2、不支持缓存锁定的处理器。

#### JAVA中实现原子操作

使用**锁**和**循环CAS**的方式。

CAS的三大问题：ABA问题（使用版本号解决），循环时间开销大（pause指令----可以延迟流水线执行指令、可以避免在退出循环时因内存顺序冲突引起CPU流水线被清空），只能保证一个共享变量的原子操作（可以用锁；或者把多个共享变量合成一个如i=2,j=a =>ij=2a再用CAS操作。JDK1.5提供了AtomicReference类）。

=========================第二章DONE=========================

## 第三章 JAVA内存模型（JMM-JSR-133）

核心是happens-before。只要不改变程序的执行结果（单线程和正确同步的多线程程序），编译器和处理器怎么优化都行。

### Java内存模型基础

#### 并发模型的两大关键问题

**线程之间如何通信（交换信息）、线程之间如何同步。**

线程之间的通信机制有2种：共享内存（通过写-读内存中的公共状态进行隐式通信，JAVA并发采用此方法）和消息传递（发送消息来显式通信）

#### Java内存模型的抽象结构

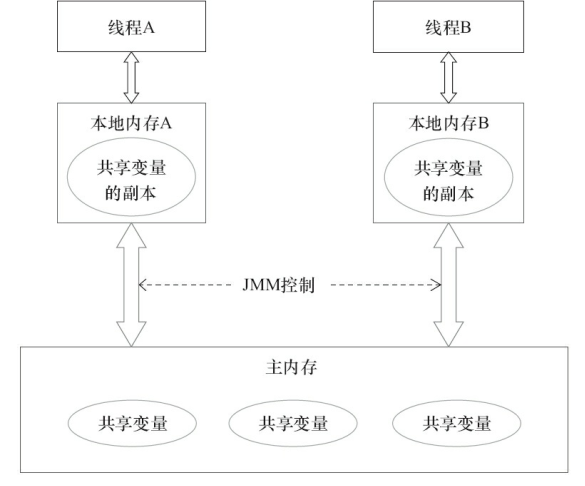


图3-1java内存模型抽象结构示意图

如图3-1，如果线程A要与线程B通信，必须经历2个步骤：

Step1:线程A把本地内存A中更新过的共享变量刷新到主内存中去。

Step2:线程B到主内存中去读取线程A之前已更新过的共享变量。

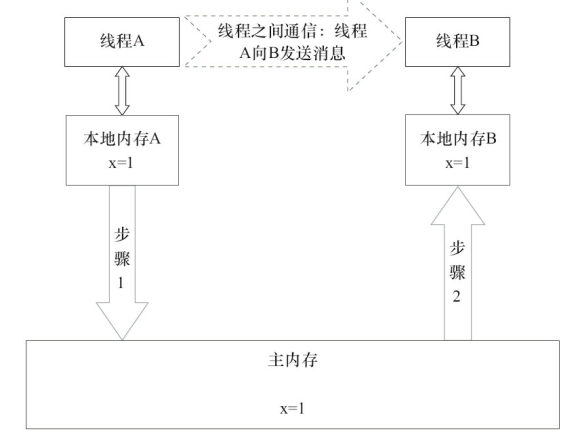


图3-2 线程之间的通信图

JMM通过控制主内存与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。

#### 指令序列的重排序

执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。分3种类型：

* 编译器优化的重排序：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。
* 指令集并行（ILP）的重排序：现代处理器采用ILP来将多条指令重叠执行，若不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
* 内存系统的重排序：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在乱序执行。

从JAVA代码到最终实际执行的指令序列，会经历这3种重排序。

JMM重排序规则：对于编译器，可禁止特定类型的编译器重排序。对于处理器，要求Java编译器在生成指令序列时插入特定类型的内存屏障指令来禁止特定类型的处理器重排序。

#### 内存屏障类型

|  |  |  |
| --- | --- | --- |
| 屏障类型 | 指令示例 | 说明 |
| LoadLoad Barries | Load1;LoadLoad;Load2 | 确保Load1数据的装载先于Load2及后续装载指令的装载 |
| StoreStroe Barries | Store1;StoreStore;Store2 | 确保Store1数据对其他处理器可见(刷新到内存)先于Store2及所有后续存储指令的存储 |
| LoadStore Barries | Load1;LoadStore;Store2 | 确保Load1数据装载先于Store2及所有后续的存储指令刷新到内存 |
| StoreLoad Barries | Store1;StoreLoad;Load2 | 确保Store1数据对其他处理器变得可见先于Load2及所有后续装载指令的装载。StoreLoad Barries会使该屏障之前的所有内存指访问指令(存储和装载指令)完成之后，才执行该屏障之后的内存访问指令。是一个全能型屏障，开销昂贵。 |

#### happens-before规则

* 程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意后续操作
* 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
* volatile变量规则：对一个volatile域的写，happen-before于任意后续对这个volatile的读。
* 传递性：A happens-before B，B happens before C => A happens-before C。

happens-before仅要求前一个操作的执行结果对后一个操作可见，并不意味着前一个操作必须要在后一个操作之前执行！

### 重排序

重排序是指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2个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。

|  |  |  |
| --- | --- | --- |
| 名称 | 代码示例 | 说明 |
| 写后读 | a=1;b=a; | 写一个变量之后，再读这个位置 |
| 写后写 | a=1;a=2; | 写一个变量之后，再写这个变量 |
| 读后写 | a=b;b=1; | 读一个变量之后，再写这个变量 |

表3-1 数据依赖类型表

编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。

这里的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作。不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

#### as-if-serial语义

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。

为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作进行重排序，因为这种重排序会改变执行结果。

#### 程序顺序规则

happens-before

### 顺序一致性

#### 数据竞争

如果程序未正确同步时，就可能存在数据竞争，JAVA内存模型规范对数据竞争的定义如下：

在一个线程中写一个变量，在另一个线程中读同一个变量，而且写和读没有通过同步来排序。

当代码中存在数据竞争时，程序的执行结果往往产生违反直觉的结果。如果一个多线程程序能正确同步，这个程序将是一个没有数据竞争的程序。

JMM对正确同步的多线程程序的内存一致性做了如下保证：

如果程序正确同步，程序的执行将具有顺序一致性。这里的同步包括常用同步原语(synchronized、volatile、final)的正确使用

#### 顺序一致性内存模型

两大特性：

1）一个线程中的所有操作必须按照程序的顺序来执行

2）（不管程序是否同步）所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，**每个操作都必须原子执行且立刻对所有线程可见**。

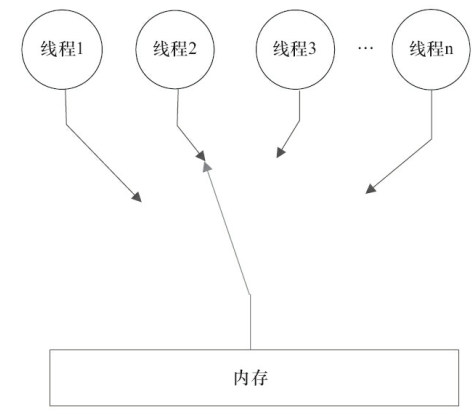


图3-3 顺序一致性内存模型视图

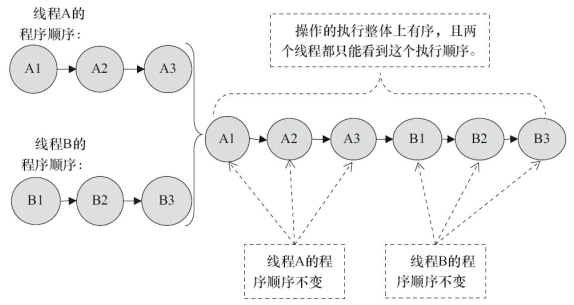


图3-4 使用监视器锁来正确同步

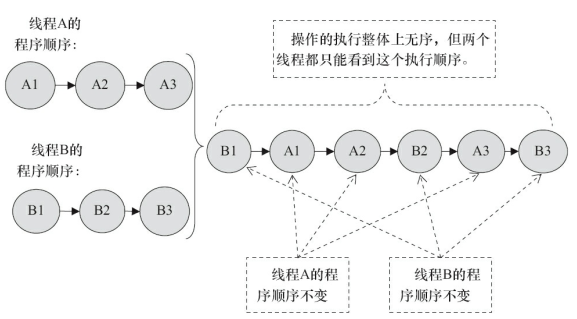
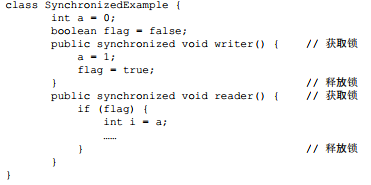


图3-5 未同步

#### 同步程序的顺序一致性效果



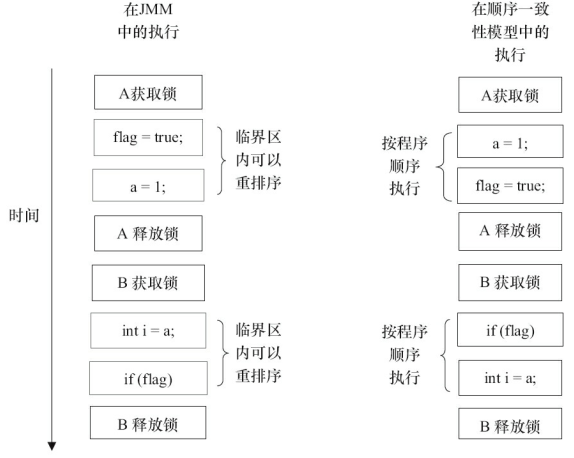


图3-6 两个内存模型中的执行时序对比图

**JMM在具体实现上的基本方针为：**在不改变（正确同步的）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提供**最小安全性**：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），JMM保证读操作读取到的值不会无中生有的冒出来。

因此，JMM在堆上分配对象时，首先会对内存空间进行清零，然后才会在上面分配对象。

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。

未同步程序在两个模型中的执行特性的差异：

* 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。
* 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。
* JMM不保证对64的long和double型变量的写操作具有原子性，顺序一致性保证对所有的内存读/写操作都具有原子性。

### volatile的内存语义

示例代码：

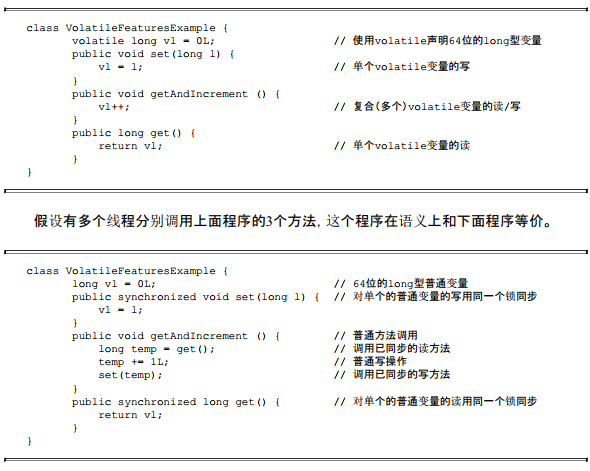


图3-7 volatile的语义

Volatile特性：

* 可见性：对一个volatile变量的读，总数能看到（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。

#### volatile写-读的内存语义

volatile写：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。

volatile读：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。线程接下来从主内存中读取共享变量。

在读线程B读一个volatile变量后，写线程A在写这个volatile变量之前所有可见的共享变量的值都将立即变得对读线程B可见。

总结：

* 线程A写一个volatile变量，实质上是线程A向接下来将要读这个volatile变量的某个线程发出了（其对共享变量所做修改的）消息。
* 线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。
* 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

#### volatile重排序规则



* 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。此规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。
* 第一个操作是volatile读时，都不重排序。确保volatile读之后的操作不会被编译器重排序到volatile读之前。
* 第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

#### JMM内存屏障插入策略

* 在每个volatile写操作前面插入一个SS屏障，后面插入一个SL屏障。
* 在每个volatile读操作后面插入一个LL屏障和一个LS屏障。

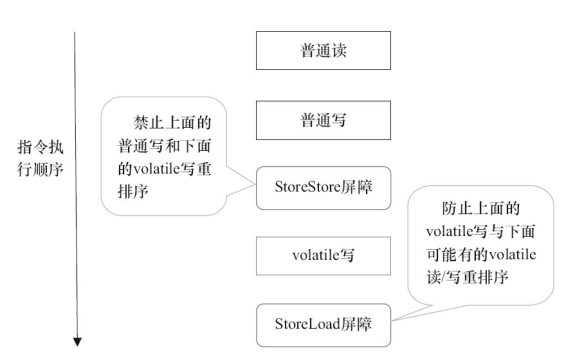


图3-8 指令序列示意图

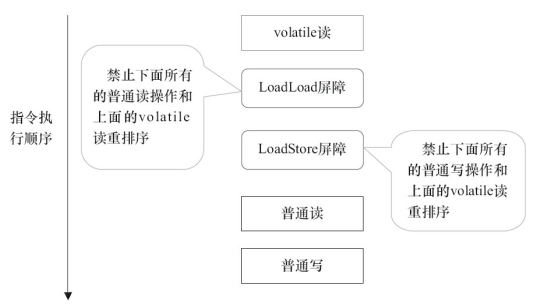
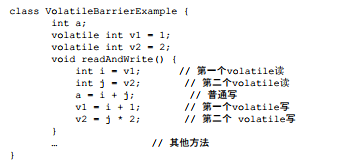


图3-9 指令序列示意图

【示例】：



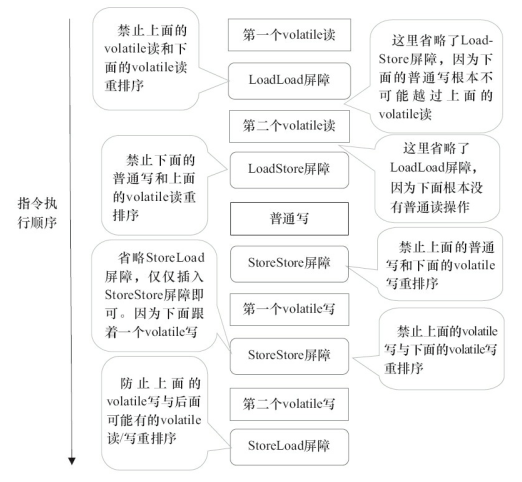
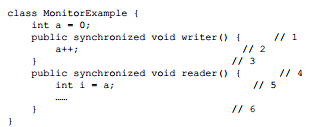


图3-10 指令序列示意图

* X86不会对读-读，读-写，写-写操作重排序，因此X86忽略这3种内存屏障，JMM仅需在volatile写后面插入一个SL即可实现volatile写-读的内存语义。Volatile写比volatile读开销大很多。

### 锁的内存语义

【示例】：



其中包含的happens-before关系：

1. 根据程序次序规则，1 happens-before 2, 2 h-b 3, 4 h-b 5, 5 h-b 6
2. 根据监视器锁规则，3 h-b 4
3. 根据happens-before传递性： 2 h-b 5。

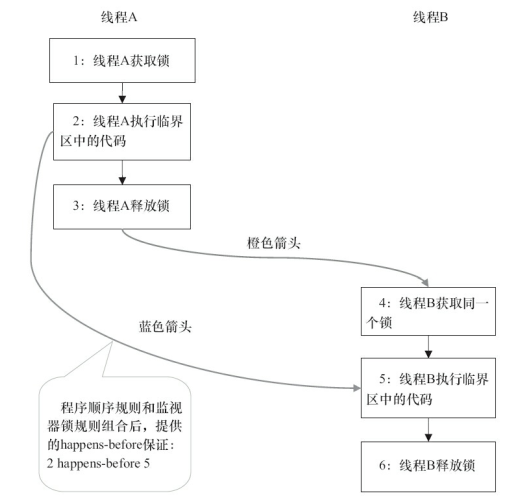


图3-11 happens-before关系图

橙色箭头代表监视器锁规则，蓝色箭头为h-b保证，黑色箭头为程序顺序规则。

A释放锁后，B获取同一个锁，2 h-b 5，所以A释放锁之前的所有可见的共享变量，在B获取同一个锁后会立刻对B可见。

#### 锁的释放和获取的内存语义

当线程释放锁时，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。

当线程获取锁时，JMM会把该线程对应的本地内存置为无效，从而使被监视器保护的临界代码必须从主内存中读取共享变量。

锁释放与volatile写具有相同的内存语义，锁获取与volatile读具有相同的内存语义。

总结：

* 线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（其对共享变量所做修改的）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（在释放这个锁之前对共享变量所做修改的）消息。
* 线程A释放锁（写volatile），随后线程B获取这个锁（读volatile），这个过程实质上是线程A通过主内存向线程B发送消息。

#### 锁的内存语义实现

**从对ReentrantLock的分析可以看出，锁释放-获取的内存语义实现由2种方式：**

* 利用volatile变量的写-读所具有的内存语义
* 利用CAS所附带的volatile读和volatile写的内存语义。

编译器不会对volatile读与volatile读后面的任意内存操作重排序，且不会对volatile写和volatile写前面的任意内存操作重排序 ===> 为了同时实现volatile读和写的内存语义，编译器不能对CAS与CAS前面和后面的任意内存操作重排序。

CAS底层用到**lock**前缀（多处理器）和**cmpxchg**。

**Lock前缀的说明：**

1. 确保对内存的读-改-写操作原子执行：前期CPU使用锁总线，后期使用缓存锁定来保证原子性。
2. 禁止该指令与之前和之后的读和写指令重排序
3. 把写缓冲区中的所有数据刷新到内存中。

**公平锁和非公平锁的内存语义总结：**

* 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，首先会去读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读和写的内存语义。

#### 总结-线程通信的4种方式

* A线程写volatile变量，随后B线程读这个volatile变量。
* A线程写volatile变量，随后B线程用CAS更新这个volatile变量。
* A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。
* A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

#### Concurrent包

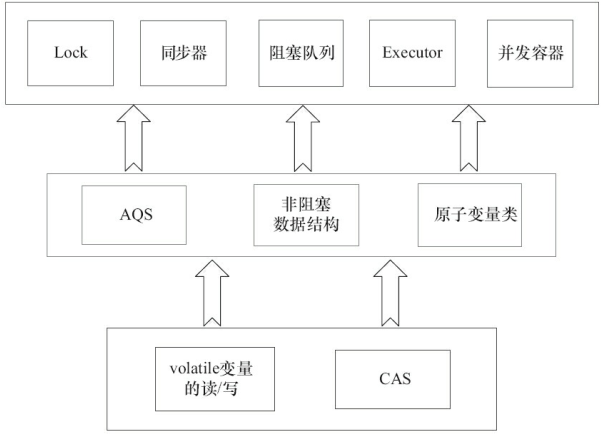


图3-12 concurrent包的实现示意图

### final域的内存语义

#### final域的重排序规则

* 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。
* 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

JMM禁止编译器把final域的写重排序到构造函数外。编译器会在final域的写之后，构造函数return之前，插入一个SS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。

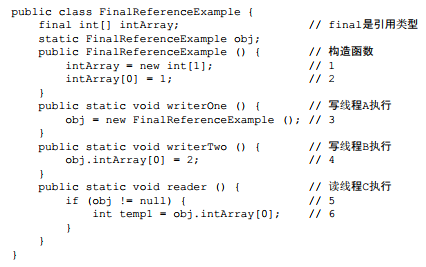
#### 读final域的重排序规则

初次读对象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。编译器会在读final域操作的前面插入一个LL屏障。

#### Final域为引用类型

写final域增加了约束：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，这两个操作不能重排序。

#### 【例子】



JMM可以确保C至少能看到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。

如果要确保读线程C看到写线程B对数组元素的写入，写线程B和读线程C之间需要使用同步原语（lock和volatile）来确保内存可见性。

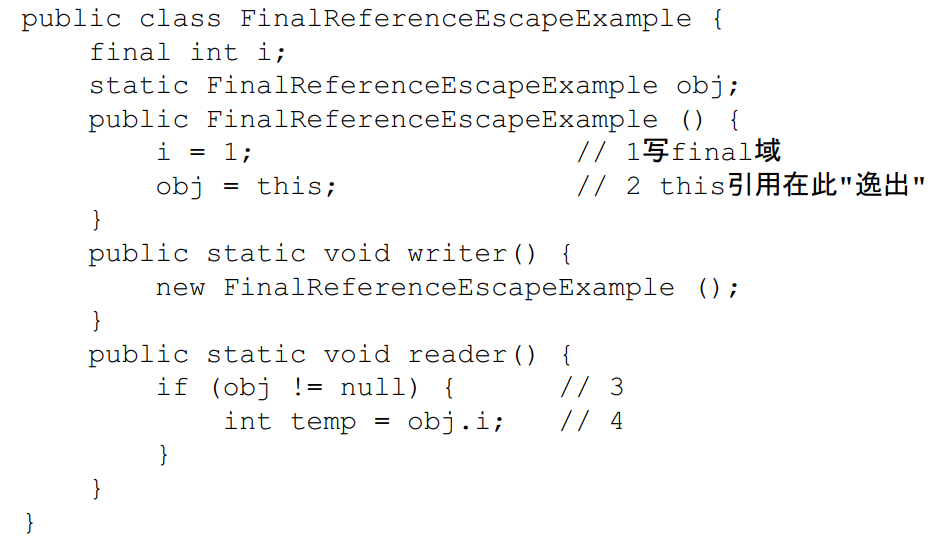
（问题：volatile读volatile写，在2个线程中会乱序吗？）

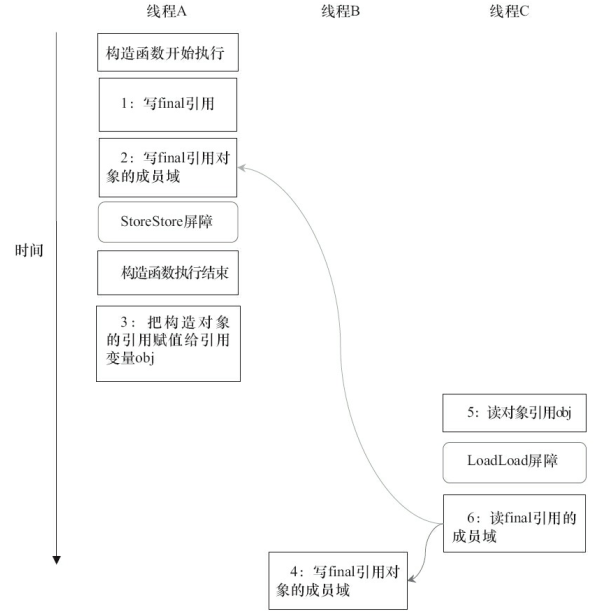
#### 为什么final引用不能从构造函数“逸出”

回顾：在引用变量为任意线程可见之前，该引用变量指向的对象的final域已经在构造函数中被正确初始化过了。

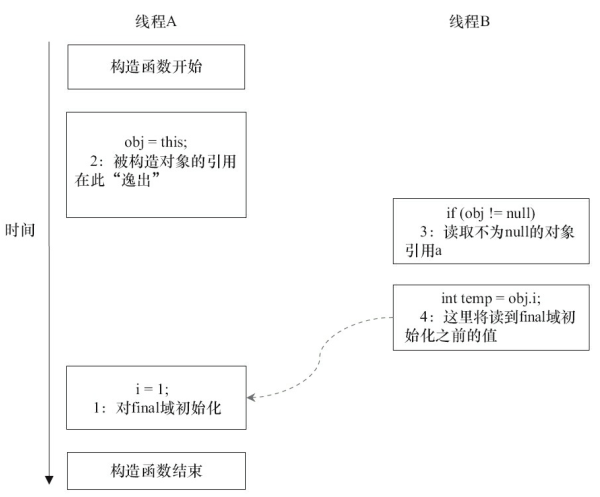
**因此必须保证：在构造函数的内部，不能让这个被构造对象的引用为其他线程所见。也就是对象引用不能在构造函数中”** **逸出”**

否则问题如下：





线程A执行writer(),B执行reader()，操作2使得对象还未完成构造前就对线程B可见。但操作1和操作2可能重排序，导致在构造函数返回前，被构造对象的引用不能为其他线程可见，因为此时final域可能还没有被初始化。



#### Final域的处理器实现

* 写final域的重排序规则会要求编译器在final域的写之后，构造函数return之前插入一个SS屏障，
* 读final域的重排序规则要求编译器在读final域的操作前插入一个LL屏障
* X86可以省略SS屏障和LL屏障，因此不需要做任何操作。

### **happens-before**

**JMM的核心概念**

#### JMM的设计

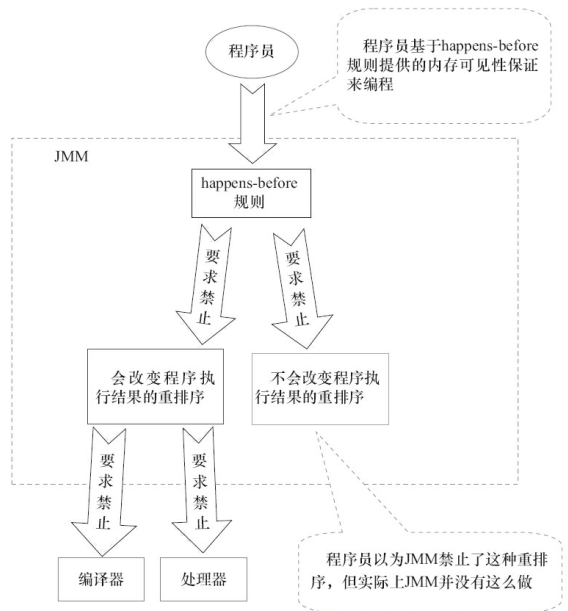
**设计JMM时，要考虑两个关键因素：**

* 程序员对内存模型的使用。 程序员希望内存模型易于理解、易于编程。程序员希望基于一个强内存模型来编写代码。
* 编译器和处理器对内存模型的实现。 编译器和处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。编译器和处理器希望实现一个弱内存模型。

因此设计JMM的**核心目标**就是找到一个好的平衡点：一方面，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放松。

**JMM把Happens-before要求禁止的重排序分为如下两类：**

* 会改变程序执行结果的重排序（JMM策略：JMM要求编译器和处理器必须禁止这种重排序）
* 不会改变程序执行结果的重排序（JMM策略：JMM对编译器和处理器不做要求，即允许这种重排序。）



如图可看出，happens-before规则满足了程序员的需求。**JMM遵循一个基本原则**：只要不改变程序的执行结果（单线程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：**编译器细致分析后发现一个锁只会被单线程访问，那么这个锁可以消除，如果一个volatile变量只会被单线程访问，那这个volatile变量会被当做普通变量。

#### Happens-before定义

JSR-133:

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）两个操作之前存在happens-before关系，并不意味着java平台的具体实现必须按照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。（JMM允许这种重排序）

**上面的1）是JMM对程序员的承诺，2）是JMM对编译器和处理器重排序的约束规则。**

Happens-before关系本质上和as-if-serial语义是一回事：

* as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before关系保证正确同步的多线程程序的执行结果不被改变。
* as-if-serial语义让人以为单线程是按程序顺序执行的，happens-before让人以为正确同步的多线程是按happens-before指定的顺序来执行的。

as-if-serial和happens-before的目的都是在不改变程序执行结果的前提下尽可能提高程序的并行度。

#### happens-before规则

JSR-133:

1）程序顺序规则：一个线程中的每个操作happens-before于该线程中任意后续操作

2）监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁

3）volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

4）传递性：A happens-before B, B happens-before C, 🡺 A happens-before C

5）start()规则：线程A执行ThreadB.start()（启动线程B），那么A的ThreadB.start()操作happens-before B的任意操作。

6）join()规则：线程A执行ThreadB.join()并成功返回，则B的任意操作happens-before于线程A从ThreadB.join()操作的成功返回。

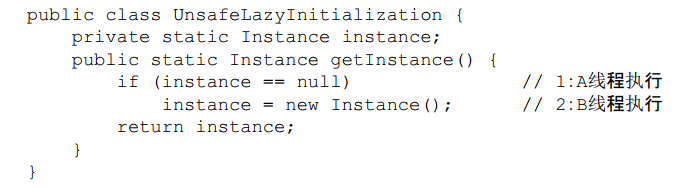
### 双重检查锁与延迟初始化（懒加载-单例模式）

在JAVA多线程中，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。双重检查锁是常见的延迟初始化技术，但它是一个错误用法。

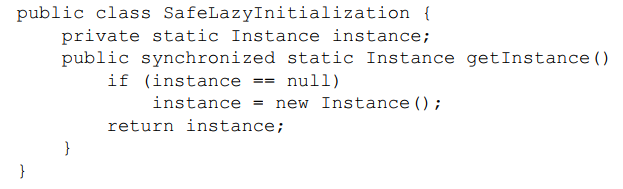
#### 双重检查锁的由来-懒加载

**延迟初始化总结：类中声明一个static Instance的instance，但不new，线程执行getInstance时再new**

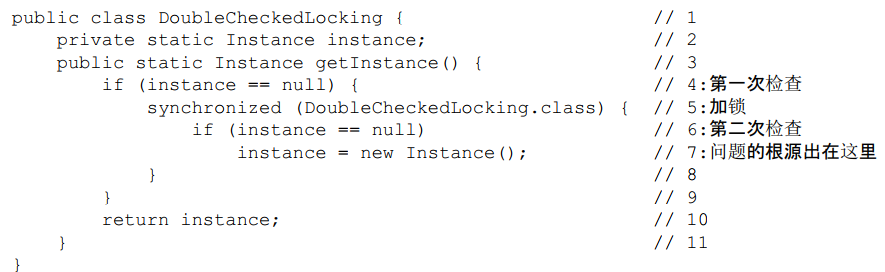
如下是非线程安全的延迟初始化：



线程安全的延迟初始化：在方法套sychronized



频繁调用sychronized造成极大开销，由此引出双重检查锁：在代码块层面sychronized。



如上代码，如果第一次检查instance不为null，那么就不需要执行下面的加锁和初始化操作。因此，可大幅降低sychronized带来的性能开销。

* 多个线程视图在同一时间创建对象时，会通过加锁来保证只有一个线程能创建对象。
* 在对象创建好之后，执行getinstance()方法将不需要获取锁，直接返回自己创建好的对象。

但是，**在线程执行到第四行时，代码读取到的instance不为null时，instance引用的对象有可能还没有完成初始化。**

#### 问题的根源

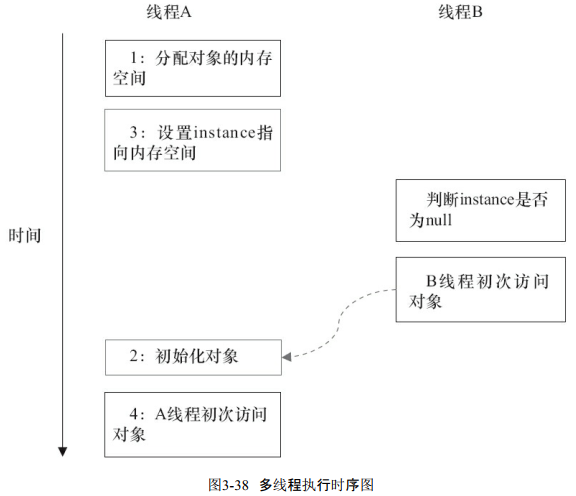
instance = new Singleton();创建对象时可以分解成如下3行伪代码：

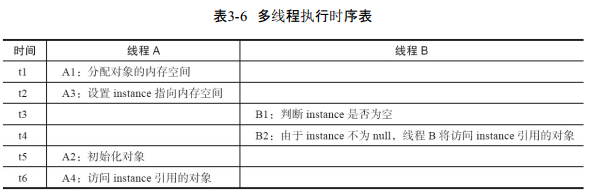
memory=allocate(); //1:分配对象的内存空间

ctorInstance(memory); //2:初始化对象

instance=memory; //3:设置instance指向刚分配的内存地址

这3行代码的2和3可能被重排序，此时instance指向了内存地址却还没被初始化。但不违法intra-thread semantics，却导致：



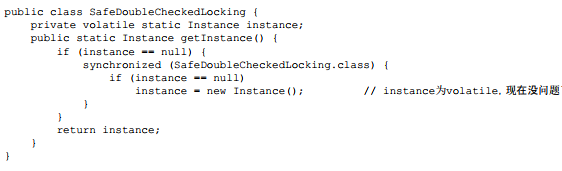


两个解决方法：

1. 不允许2和3重排序
2. 允许2和3重排序，但不允许其他线程“看到”这个重排序。

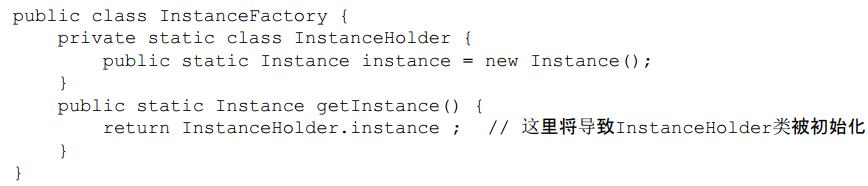
#### 基于volatile的解决方案-基于方法1

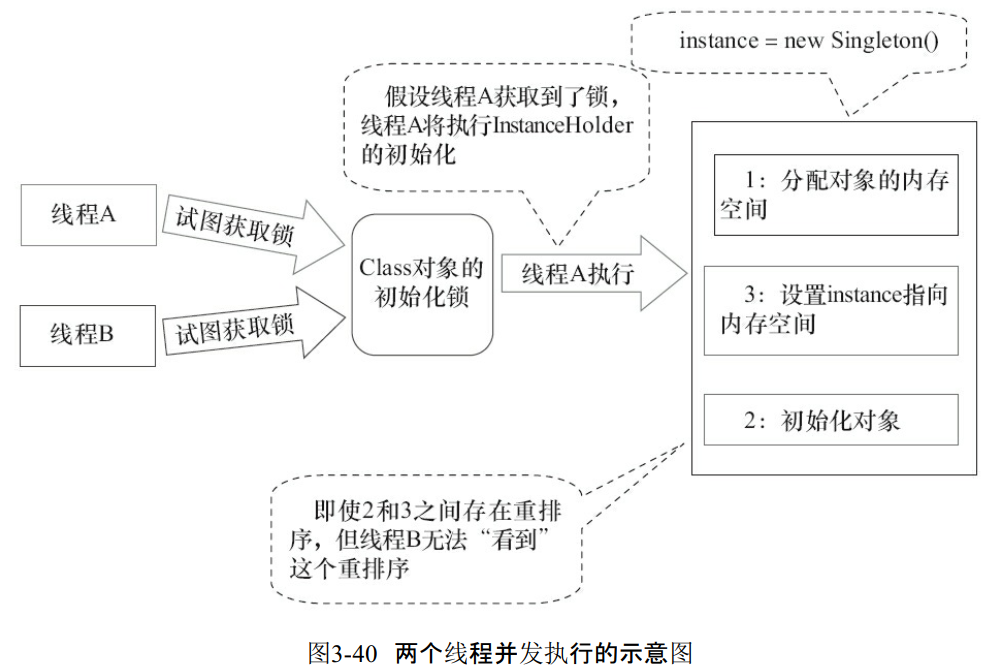
把instance声明为volatile型。但需要jdk1.5以上。本质上是通过方法1.



#### 基于类初始化的解决方案-基于方法2

JVM在类的初始化阶段，（在Class被加载后，被线程使用前），会执行类的初始化，在此期间，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。因此可如下实现：





* 初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。首次发生如下任意一种情况时，一个类或接口类型T被立即初始化。

1. T是一个类，而且一个T类型的实例被创建
2. T是一个类，且T中声明的一个静态方法被调用。
3. T中声明的一个静态字段被复制。
4. T中声明的一个静态字段被使用，而且这个字段不是一个常量字段。
5. T是一个顶级类，而且一个断言语句嵌套在T内部被执行。

在instanceFactory中，首次执行getInstance()方法的线程将导致InstanceHolder类被初始化**（符合情况4）**.

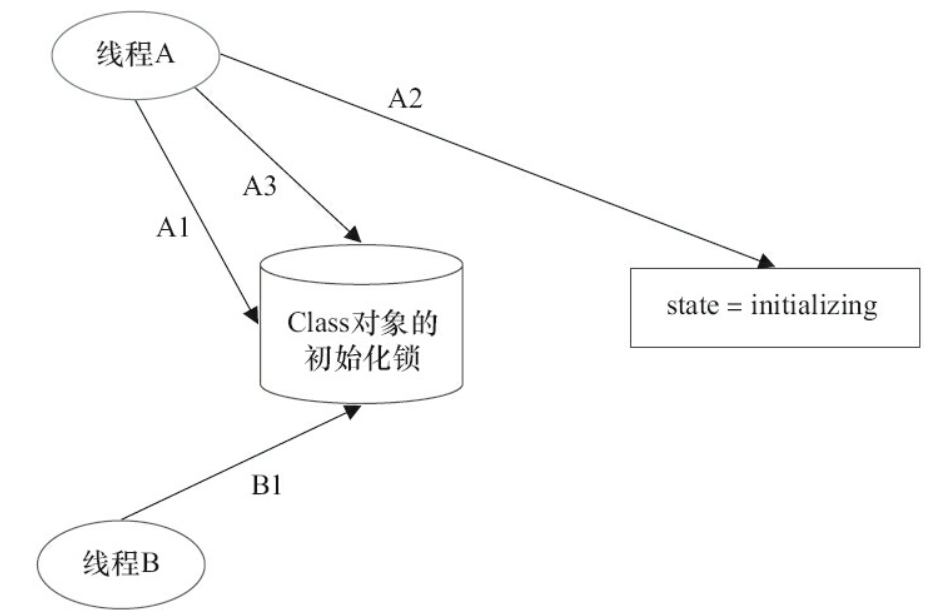
多线程可能同时尝试初始化一个类或接口，所以Java初始化一个类或接口时需要做细致的同步处理：

对于每一个类或接口C，都有唯一的初始化锁LC与之对应。从C到LC的映射，由JVM实现。JVM在类初始化期间会获得这个初始化锁，并且**每个线程至少获取一次锁来确保这个类已经被初始化过了。**

* 类的初始化的处理如下5个阶段：

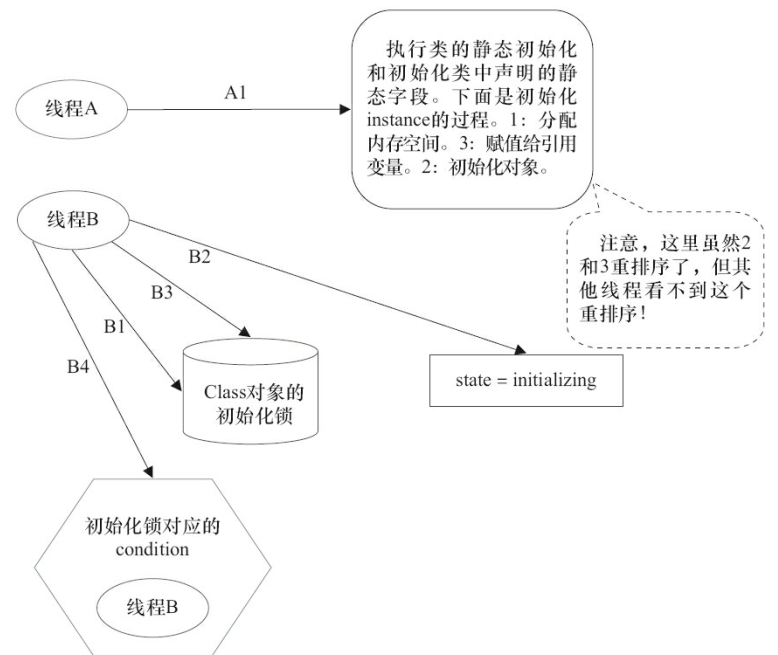
**第1阶段：通过在Class对象上同步（即获取Class对象的初始化锁），来控制类或接口的初始化。这个获取锁的线程会一致等待，直到当前线程能够获取到这个初始化锁。**

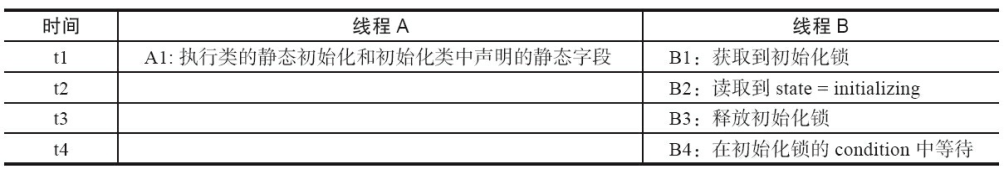
情景：Class对象未初始化，初始state=noInitialization，线程A和B试图同时初始化这个Class对象。



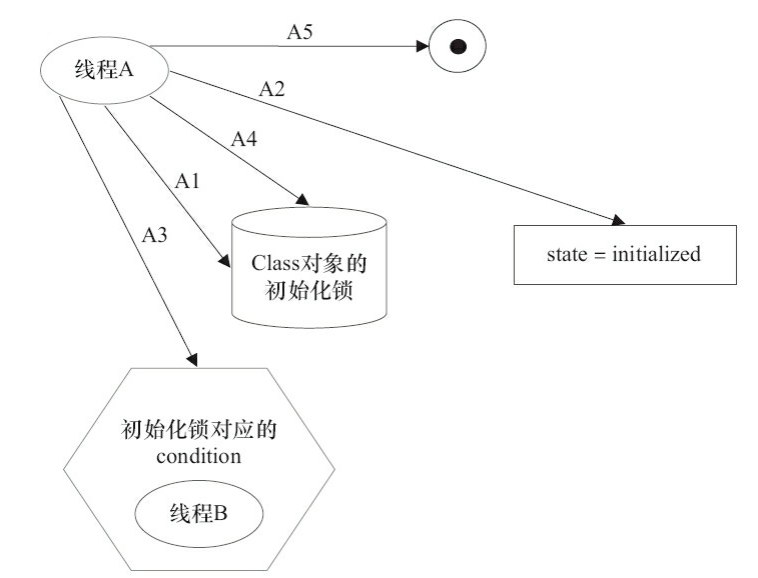


**第2阶段：线程A执行类的初始化，B在初始化锁对应的condition上等待。**



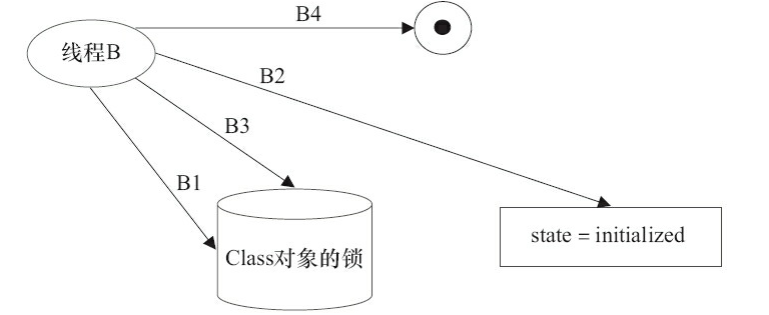


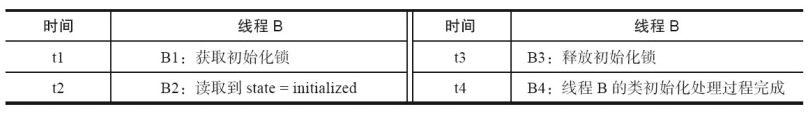
**第3阶段：线程A设置state=initialized，然后唤醒在condition中等待的所有线程。**

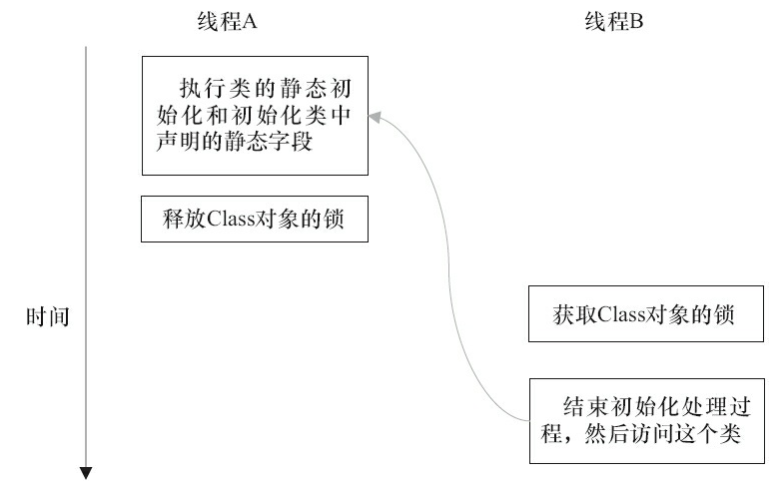


|  |  |
| --- | --- |
| **时间** | **线程A** |
| t1 | A1:获取初始化锁 |
| t2 | A2:设置state=initialized |
| t3 | A3:唤醒在condition中等待的所有线程 |
| t4 | A4:释放初始化锁 |
| t5 | A5:线程A的初始化处理过程完成 |

**第4阶段：线程B结束类的初始化处理**





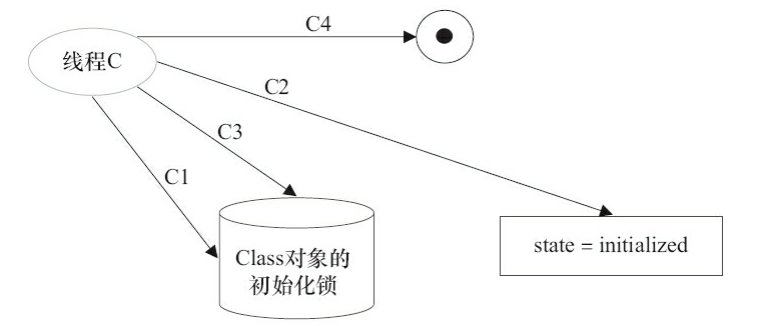


线程A在第2阶段的A1执行类初始化，在第3阶段的A4释放初始化锁；

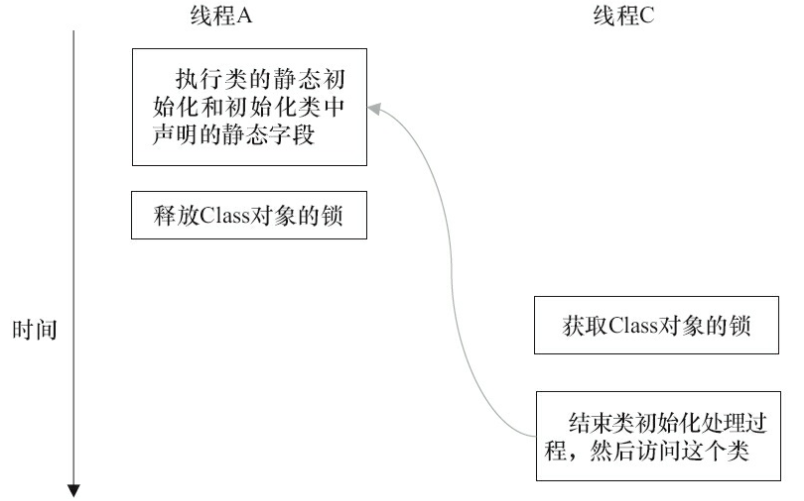
线程B在第4阶段的B1获取同一初始化锁，并在第4阶段的B4之后才开始访问这个类。

Happens-before保证：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），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。**







A和B都经历了2次获取锁-释放锁，C只需要经历1次。

Happens-before保证：线程A执行类的初始化时的写入操作，线程C一定能看到。

\*condition和state为虚构的。JVM具体实现只需要实现类似功能即可。

#### 2种方案比较

基于volatile可以实现对静态和实例字段的延迟初始化，而基于类初始化只能实现静态字段的延迟初始化。但基于类初始化代码更简洁。

因此volatile更适合对实例字段的延迟初始化，基于类初始化更适合实现静态字段的延迟初始化。

#### 延迟加载优缺

字段延迟初始化降低了初始化类或创建实例的开销，但增加了访问被延迟初始化的字段的开销。

大多数时候，正常初始化要优于延迟初始化。

## 第四章 JAVA并发编程基础

## 名词总结

|  |  |  |
| --- | --- | --- |
| 术语 | 英文单词 | 术语描述 |
| 内存屏障 | Memory barriers | 是一组处理器指令，用于实现对内存操作的顺序限制。 |
| 缓冲行 | Cache line | CPU高速缓存中可以分配的最小存储单位，处理器填写缓存行时会加载整个内存行，现代CPU要执行几百次CPU指令 |
| 原子操作 | Stomic operations | 不可中断的一个或一系列操作 |
| 缓存行填充 | Cache line fill | 当处理器识别到内存中读取操作数是可缓存的，处理器读取整个高速缓存行到适当的缓存(L1,L2,L3的或所有) |
| 缓存命中 | Cache hit | 高速缓存行填充操作的内存位置任然是下次处理器访问的地址时，处理器从缓存中读操作数而不是内存中。 |
| 写命中 | Write hit | 处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存地址是否在缓存行中，存在则写回到缓存而不是内存中。 |
| 写缺失 | Write misses the cache | 一个有效的缓存行被写入到不存在的内存区域。 |
| =================================================================== | | |
| 缓存行 | Cache line | 缓存的最小操作单位 |
| 比较并交换 | Compare and swap | CAS操作需要输入两个数值，一个旧值（期望操作前的值）和一个新值，比较旧值是否变化（与期望值相等），如果没有就用新值替换。 |
| CPU流水线 | CPU pipeline | CPU流水线的工作方式就像工业生产上的装配流水线，在CPU中由5-6个不同功能的电路单元组成一条指令处理流水线，然后将一条X86指令分成5-6步后由这些电路单元分别执行，这样就能实现在一个CPU时钟周期完成一条指令，因此提高CPU的运算速度。 |
| 内存顺序冲突 | Memory order violation | 内存顺序冲突一般是由假共享引起的，假共享是指多个CPU同时修改一个缓存行的不同部分而引起其中一个CPU的操作无效，这种情况CPU必须清空流水线。 |