# Part 1

## 1.volatile是java虚拟机提供的轻量级的同步机制

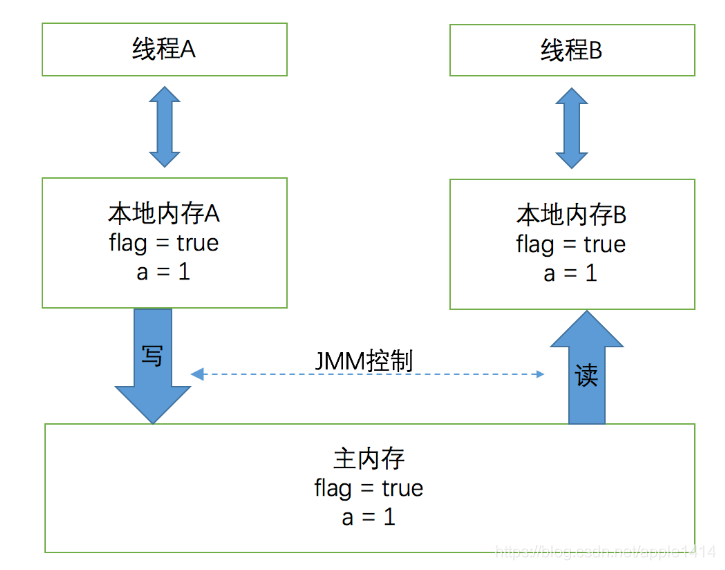
(1)保证可见性

(2)不保证原子性

(2禁止指令重排

## 2.volatile 写读的内存语义

JMM的过程如下



volatile内存语义实现

JMM采取保守策略对volatile读写插入内存屏障：

1）每个 volatile 写前插入 StoreStore屏障

2）每个 volatile 写后插入 StoreLoad屏障

3）每个 volatile 读后插入 LoadLoad屏障

4）每个 volatile 读后插入 LoadStore屏障

由于volatile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。功能上，锁更强大；在可伸缩性和执行性能上，volatile更有优势。

## 3.volatile和synchronized的区别？

（1）volatile保证变量的可见性，即只保证了对volatile变量读和写操作的原子性，并不能保证对i++这种复合操作的原子性。 而synchronized则是锁定当前临界资源，只有当前线程可以访问该资源，其他线程被阻塞住。保证了可见性和原子性。

（2）volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和代码块和类。

（3）volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

## AtomicInteger使用总结

AtomicInteger是在使用非阻塞算法实现并发控制，在一些高并发程序中非常适合，但并不能每一种场景都适合，不同场景要使用使用不同的数值类。

## [happens before原则](https://www.cnblogs.com/hujinshui/p/10421496.html)

　　JSR-133使用happens-before的概念来指定两个操作之间的执行顺序。由于这两个操作可以在一个线程内，也可以在不同线程之间。因此，JMM可以通过happens-before关系向程序员提供跨线程的内存可见性保证（如果A线程的写操作a与B线程的读操作b之间存在happens-before关系，尽管a操作与b操作在不同的线程中执行，但JMM向程序员保证a操作将对操作b可见）。

　　happens-before关系的定义：

　　　　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，而且第一个操作执行顺序排在第二个操作之前。

这是JMM对程序员的承诺。从程序员的角度来说，可以这样理解happens-before关系：如果A happens-before B，那么Java内存模型将向程序员保证——A操作的结果将对B可见，且A的执行顺序排在B之前。注意！这是Java内存模型向程序员做出的保证。

　　　　2、两个操作之间存在happens-before关系，并不意味着Java平台的具体辉县必须要按照happens-before关系指定的顺序孩子来执行。如果重排序之后的执行结果，与按happens-before关系来执行的顺序结果一致，那么这种重排序是合法的（也就是说JMM允许这种重排序）。

这是JMM对编译器和处理器重排序的约束原则。JMM其实遵循一个基本原则：只要不改变程序的执行结果，编译器和处理器怎么优化都行。

## [如何理解Java中的happens-before？](https://blog.csdn.net/tianruirui/article/details/103019758)

happens-before是Java内存模型中定义的两个操作之间的偏序关系，即如果操作A在操作B之前先发生，那么操作A产生的操作结果，操作B可以观察到，或者说操作A的结果影响到操作B。可以认为Java内存模型中的这种与生俱来的原则实现了可见性和顺序性。

happens-before无需任何同步器的协助，只要两个操作之间的关系符合以下列出的这些规则，或者可以由以下这些规则推导出，那么就可以保证它们的顺序性，否则Java虚拟机可以进行任意重排序。

程序次序规则

在一个线程内，按照控制流顺序，书写在前面的操作先行发生于书写在后面的操作。

管程锁定规则

一个unlock操作先行发生于后面对同一个锁的lock操作。

volatile变量规则

对一个volatile变量的写入操作先行发生于后面对这个变量的读操作。

线程启动规则

Thread对象的start()方法先行发生于此线程的每一个动作。

线程终止规则

线程中的所有操作都先行发生于对此线程的终止检测。

线程中断规则

对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupt()方法检测到是否有中断发生。

对象终结规则

一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize()方法的开始。

传递性

如果操作A先行发生于操作B，操作B先行发生于操作C，那么就可以得出操作A先行发生于操作C。

## [volatile的原理和实现机制](https://blog.csdn.net/yinbucheng/article/details/71305951?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-4.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-4.nonecase)

　　前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何保证可见性和禁止指令重排序的。

　　下面这段话摘自《深入理解Java虚拟机》：

　　“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

　　lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

　　1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

　　2）它会强制将对缓存的修改操作立即写入主存；

　　3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

## [基本概念](https://blog.csdn.net/kai402458953/article/details/89516604?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase)

### 可见性

当一个线程修改了共享变量时，另一个线程可以读取到这个修改后的值。

内存屏障（Memory Barriers）

处理器的一组指令，用于实现对内存操作的顺序限制。

### 缓冲行

CPU告诉缓存中可以分配的最小存储单位，处理器填写缓存行时，会加载整个缓存行。

### Lock前缀的指令

Lock前缀的指令在多核处理器下会发生两件事情：

1）将当前处理器的缓存行的数据写回到系统内存。

2）这个写回内存的操作会使其他CPU缓存了该内存的地址的数据无效。

### 缓存一致性协议

在多处理器下，为零保证各个处理器的缓存是一致的，每个处理器都会通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了。当处理器发现自己缓存行对应的地址被修改，就会将当前处理器的缓存行设置为无效状态。当处理器对这个数据进行读写的时候，会重新把数据从内存中读取到处理器缓存中。

### CAS

CompareAndSwap 比较并交换  
CAS操作需要输入两个值，一个旧值（执行CAS操作前的值，期望值）和一个新值，只有当当前值等于旧值时，才可以将当前值设置为新值，否则不设置。这是一个原子操作，由硬件保证。

### 重排序规则

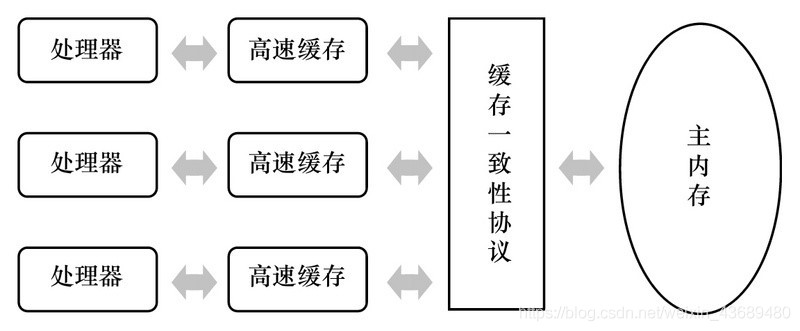
从根本上来所，JMM 对编译器和处理器的重排序限制只有一条，只要不改变程序执行的结果（指的是单线程或者正确同步的多线程环境下），那么编译器和处理器怎么优化都可以。

### Volatile

从上面的Lock前缀指令和缓存一致性协议可以看出来，这就是volatile的实现原理了。  
实际上，valatile变量被写入时，确实加了一个Lock前缀的指定，以此来达到可见性的目的。

# [内存屏障](https://blog.csdn.net/weixin_43689480/article/details/97103325)

## 为什么要有内存屏障

这个是为了解决因为cpu，高速缓存，主内存出现的时候，导致的可见性和重序性问题，什么问题呢，看下面的代码  
我们都知道计算机运算任务需要CPU和内存相互配合共同完成，其中CPU负责逻辑计算，内存负责数据存储。CPU要与内存进行交互，如读取运算数据、存储运算结果等。由于内存和CPU的计算速度有几个数量级的差距，为了提高CPU的利用率，现代处理器结构都加入了一层读写速度尽可能接近CPU运算速度的高速缓存来作为内存与CPU之间的缓冲：将运算需要使用的数据复制到缓存中，让CPU运算可以快速进行，计算结束后再将计算结果从缓存同步到主内存中，这样处理器就无须等待缓慢的内存读写了。就像下面这样  


高速缓存的引入解决了CPU和内存之间速度的矛盾，但是在多CPU系统中也带来了新的问题：可见性问题和重排序问题。

首先是可见性问题：假设有两个线程A、B分别在两个不同的CPU上运行，它们共享同一个变量X。如果线程A对X进行修改后，并没有将X更新后的结果同步到主内存，则变量X的修改对B线程是不可见的。这样就会造成可见性问题

然后是重排序问题：假设A、B两个线程共享两个变量X、Y，A和B分别在不同的CPU上运行。在A中先更改变量X的值放到高速缓存区，然后再更改变量Y的值放到高速缓存区。这时有可能发生Y的值被同步回主内存，而X的值没有同步回主内存的情况，此时对于B线程来说是无法感知到X变量被修改的，或者可以认为对于B线程来说，Y变量的修改被重排序到了X变量修改的前面。

就是为了解决上面的多线程里面的可见性和重序性问题，所以有了下面的内存屏障技术

## 内存屏障的主要组成

### 首先是硬件上面的内存屏障

Load屏障，是x86上的”ifence“指令，在其他指令前插入ifence指令，可以让高速缓存中的数据失效，强制当前线程从主内存里面加载数据

Store屏障，是x86的”sfence“指令，在其他指令后插入sfence指令，能让当前线程写入高速缓存中的最新数据更新写入主内存，让其他线程可见。

### Java里面的内存屏障

在java里面有4种，就是 LoadLoad,StoreStore,LoadStore,StoreLoad，实际上也能看出来，这四种都是上面的两种的组合产生的

LoadLoad屏障：举例语句是Load1; LoadLoad; Load2(这句里面的LoadLoad里面的第一个Load对应Load1加载代码，然后LoadLoad里面的第二个Load对应Load2加载代码)，此时的意思就是在Load2加载代码在要读取的数据之前，保证Load1加载代码要从主内存里面读取的数据读取完毕。

StoreStore屏障：举例语句是 Store1; StoreStore; Store2(这句里面的StoreStore里面的第一个Store对应Store1存储代码，然后StoreStore里面的第二个Store对应Store2存储代码)。此时的意思就是在Store2存储代码进行写入操作执行前，保证Store1的写入操作已经把数据写入到主内存里面，确认Store1的写入操作对其它处理器可见。

LoadStore屏障：举例语句是 Load1; LoadStore; Store2(这句里面的LoadStore里面的Load对应Load1加载代码，然后LoadStore里面的Store对应Store2存储代码)，此时的意思就是在Store2存储代码进行写入操作执行前，保证Load1加载代码要从主内存里面读取的数据读取完毕。

StoreLoad屏障：举例语句是Store1; StoreLoad; Load2(这句里面的StoreLoad里面的Store对应Store1存储代码，然后StoreLoad里面的Load对应Load2加载代码)，在Load2加载代码在从主内存里面读取的数据之前，保证Store1的写入操作已经把数据写入到主内存里面，确认Store1的写入操作对其它处理器可见。

## Volatile关键字里面的内存屏障是起作用的

在每个volatile写操作前插入StoreStore屏障，这样就能让其他线程修改A变量后，把修改的值对当前线程可见，

在写操作后插入StoreLoad屏障，这样就能让其他线程获取A变量的时候，能够获取到已经被当前线程修改的值

在每个volatile读操作前插入LoadLoad屏障，这样就能让当前线程获取A变量的时候，保证其他线程也都能获取到相同的值，这样所有的线程读取的数据就一样了，

在读操作后插入LoadStore屏障；这样就能让当前线程在其他线程修改A变量的值之前，获取到主内存里面A变量的的值。

# Reference

[【volatile相关面试题】](https://blog.csdn.net/apple1414/article/details/101177643)

[【AtomicInteger类的理解与使用】](https://blog.csdn.net/u012734441/article/details/51619751?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase)