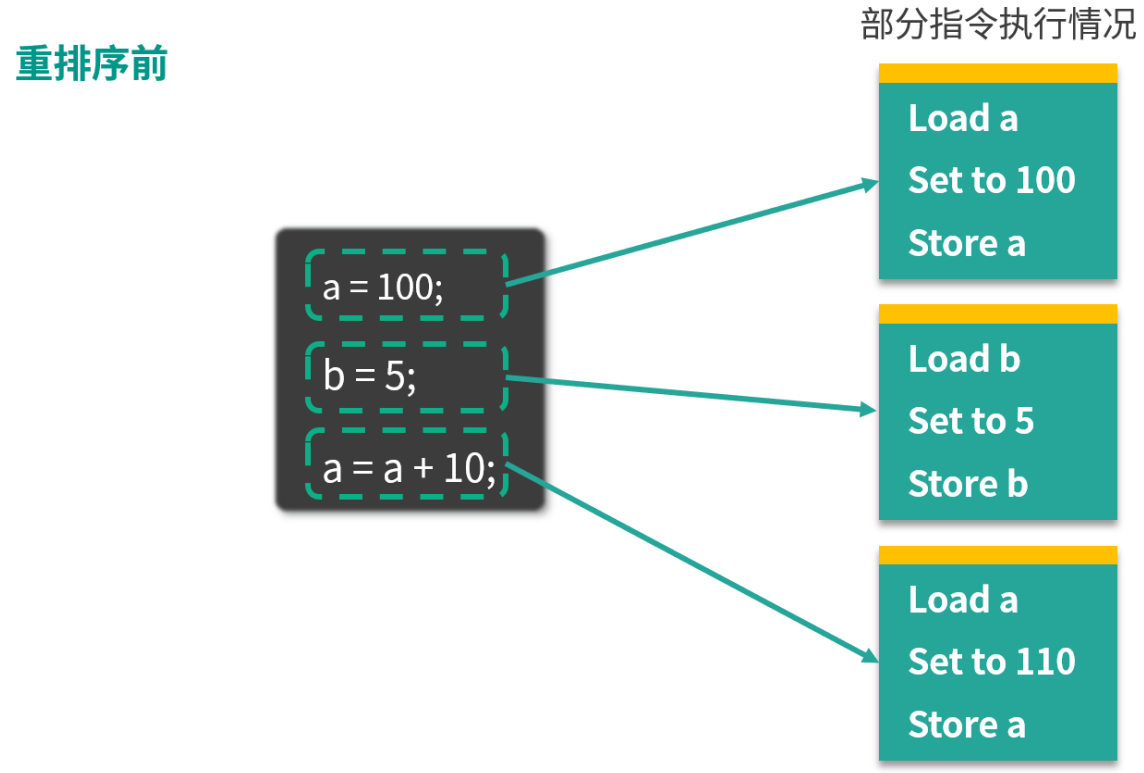
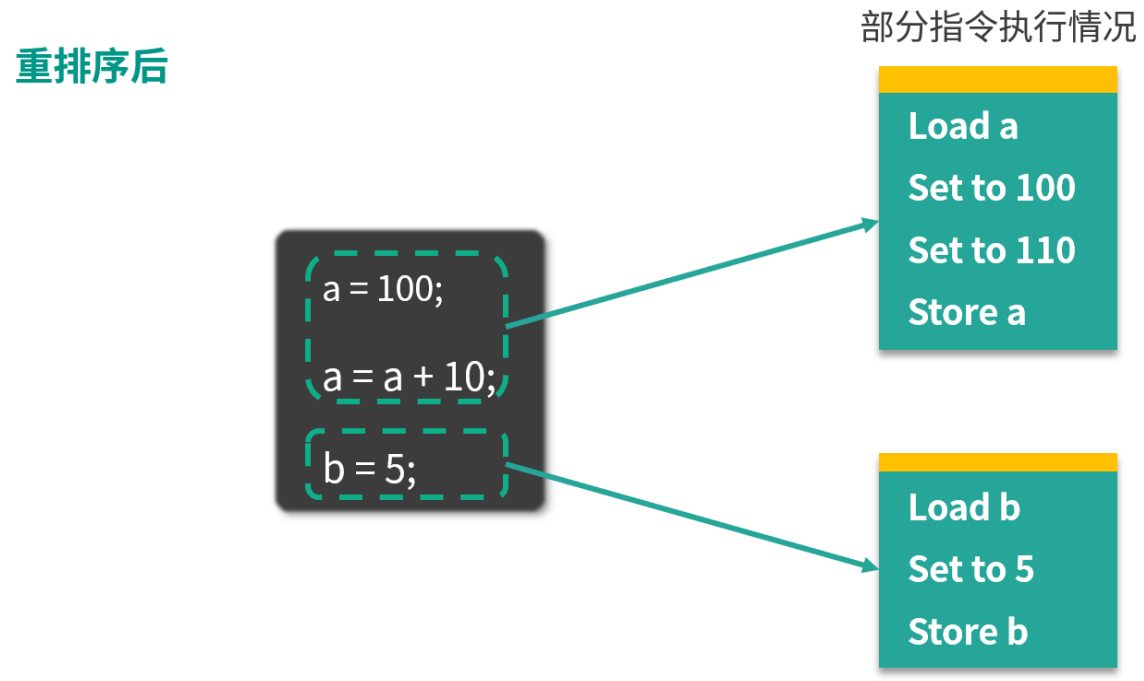
什么是指令重排序？为什么要重排序？
重排序的 3 种情况：
什么是原子性和原子操作：
Java 中的原子操作有哪些：
为什么 long 和 double没有原子性：
synchronized 不仅保证了原子性，还保证了可见性：
主内存和工作内存的关系？
CPU简单结构图：
为什么要缓存？
为什么要设置多级缓存？
JMM的抽象：主内存和工作内存
主内存和工作内存的关系
happens-before介绍：
Happens-before 关系的规则有哪些？
哪些符合happens-before规则：
volatile 的作用是什么？与 synchronized 有什么异同？
volatile 和 synchronized的异同：
volatile 的适合场景：
单例模式的双重检查锁模式为什么必须加 volatile？
单例模式双重检查锁模式的写法（懒汉式的线程安全写法）：
“为什么要 double-check[就是代码的 if (singleton == null)]？去掉任何一次的 check 行不行？”
在双重检查锁模式中为什么需要使用 volatile 关键字：
volatile的底层原理？

什么是指令重排序？为什么要重排序？

重排序讲解：





如上图对比，重排序前如果按顺序执行会有两次执行“Load a”和“Store a“，而重排序后把两次合并成一次。

重排序的 3 种情况：

（1）编译器优化：

编译器（包括 JVM、JIT 编译器等）出于优化的目的，例如当前有了数据 a，把对 a 的操作放到一起效率会更高，避免读取 b 后又返回来重新读取 a 的时间开销，此时在编译的过程中会进行一定程度的重排。不过重排序并不意味着可以任意排序，它需要需要保证重排序后，不改变单线程内的语义，否则如果能任意排序的话，程序早就逻辑混乱了。

（2）CPU 重排序：

CPU 同样会有优化行为，这里的优化和编译器优化类似，都是通过乱序执行的技术来提高整体的执行效率。所以即使之前编译器不发生重排，CPU 也可能进行重排，我们在开发中，一定要考虑到重排序带来的后果。

（3） 内存的“重排序”：

内存系统内不存在真正的重排序，但是内存会带来看上去和重排序一样的效果，所以这里的“重排序”打了双引号。由于内存有缓存的存在，在 JMM 里表现为主存和本地内存，而主存和本地内存的内容可能不一致，所以这也会导致程序表现出乱序的行为。

举个例子，线程 1 修改了 a 的值，但是修改后没有来得及把新结果写回主存或者线程 2 没来得及读到最新的值，所以线程 2 看不到刚才线程 1 对 a 的修改，此时线程 2 看到的 a 还是等于初始值。但是线程 2 却可能看到线程 1 修改 a 之后的代码执行效果，表面上看起来像是发生了重顺序。

什么是原子性和原子操作：

即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素**打断**，要么就都不执行。原子性就像数据库里面的事务一样，他们是一个团队，**同生共死**。

Java 中的原子操作有哪些：

- 除了 long 和 double 之外的基本类型（int、byte、boolean、short、char、float）的读/写操作，都天然的具备原子性；
- 所有引用 reference 的读/写操作；
- 加了 volatile 后，所有变量的读/写操作（包含 long 和 double）。这也就意味着 long 和 double 加了 volatile 关键字之后，对它们的读写操作同样具备原子性；
- 在 java.concurrent.Atomic 包中的一部分类的一部分方法是具备原子性的，比如 AtomicInteger 的 incrementAndGet 方法。

为什么 long 和 double没有原子性：

只是32位的操作系统没有原子性，因为对于32位操作系统来说，单次次操作能处理的最长长度为32bit，而long和double类型8字节64bit，所以对long的读写都要两条指令才能完成。

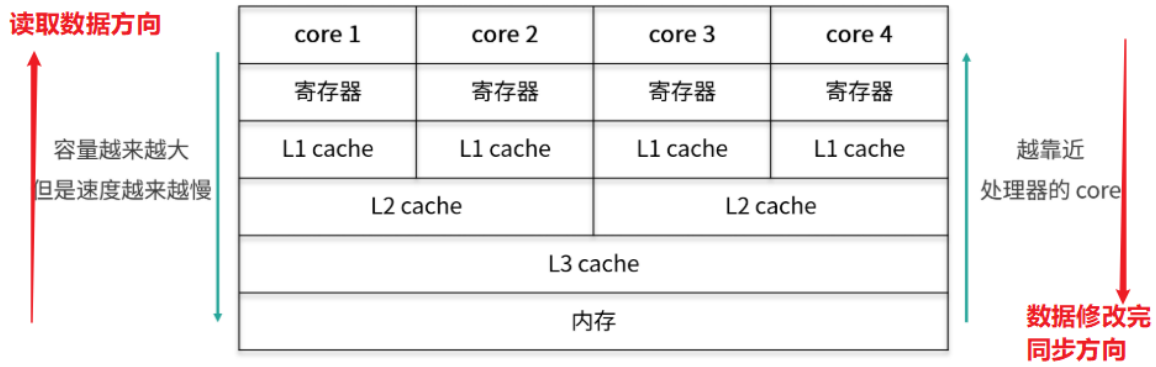
要想保证原子性，需加volatile 字段。（现在主流的Java 虚拟机几乎都会把 64 位数据的读写操作作为原子操作来对待，所以我们不用额外的把 long 和 double 声明为 volatile）

synchronized 不仅保证了原子性，还保证了可见性：

synchronized 不仅保证了临界区内最多同时只有一个线程执行操作，同时还保证了在前一个线程释放锁之后，之前所做的所有修改，都能被获得同一个锁的下一个线程所**看到**，也就是能读取到最新的值。因为如果其他线程看不到之前所做的修改，依然也会发生线程安全问题。

主内存和工作内存的关系？

CPU简单结构图：



如上图所示，越靠近核心速度也越快，每个核心在获取数据时，都会将数据从内存一层层往上读取，同样，后续对于数据的修改也是先写入到自己的 L1 缓存中，然后等待时机再逐层往下同步，直到最终刷回内存。。

> 一、寄存器的定义
>
> 寄存器是中央处理器内的组成部分。寄存器是有限存贮容量的高速存bai贮部件，它们可用来暂存指令、数据和地址。在中央处理器的控制部件中，包含的寄存器有指令寄存器(IR)和程序计数器(PC)。在中央处理器的算术及逻辑部件中，寄存器有累加器(ACC)。
>
> 二、寄存器的作用
>
> 1、可将寄存器内的数据执行**算术及逻辑运算**
>
> 2、存于寄存器内的地址可用来指向内存的某个位置，即寻址
>
> 3、可以用来读写数据到电脑的周边设备。

为什么要缓存?

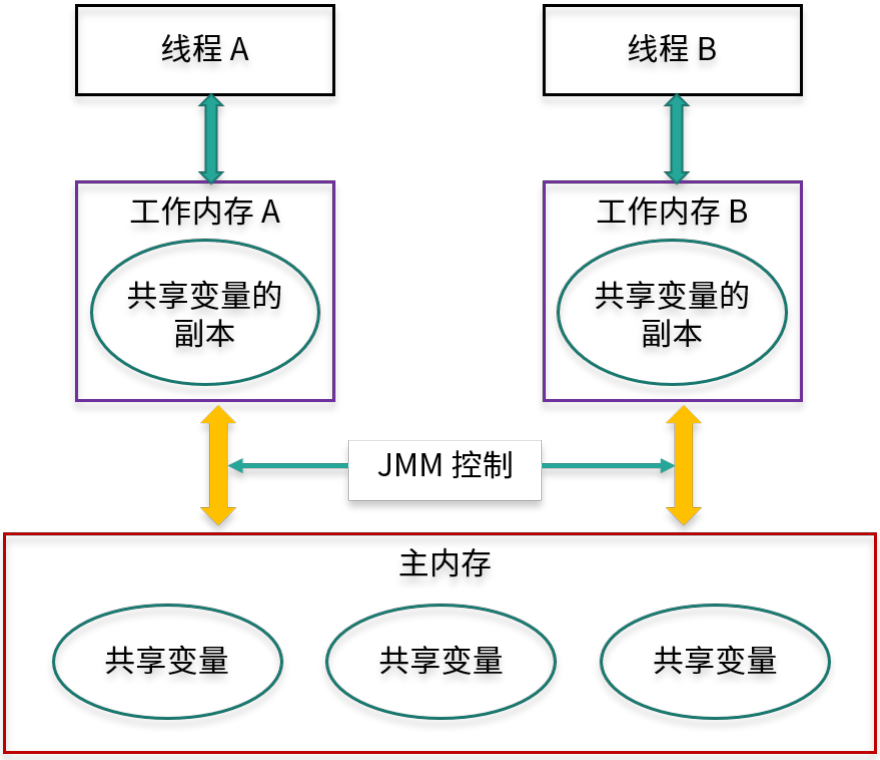
答：CPU 的处理速度很快，相比之下，内存的速度就显得很慢，所以为了提高 CPU 的整体运行效率，**减少空闲时间**，在 CPU 和内存之间会有 cache 层，也就是缓存层的存在。（缓存的速度比内存要快得多）

为什么要设置多级缓存?

答：因为一级缓存成本太高，导致无法生产太大的一级缓存（Intel的CPU的一级缓存更小），只能生产二级缓存来弥补，继而后面又产生了三级缓存。

JMM的抽象：主内存和工作内存

JMM屏蔽了 L1 缓存、L2 缓存、L3 缓存底层细节，**JMM 定义了一套读写数据的规范**，抽象出来的主内存和工作内存的概念。



主内存和工作内存的关系

JMM 有以下规定：

（1）所有的变量都存储在主内存中，同时每个线程拥有自己独立的工作内存，而工作内存中的变量的内容是主内存中该变量的拷贝；

（2）线程**不能直接读 / 写主内存中的变量**，但可以操作自己工作内存中的变量，然后再同步到主内存中，这样，其他线程就可以看到本次修改；

（3） 主内存是由多个线程所共享的，但线程间不共享各自的工作内存，如果**线程间需要通信，则必须借助主内存中转来完成。**

happens-before介绍：

happens-before是一套规则：如果两个操作满足 happens-before 关系，那么第二个操作在执行时就一定能保证看见第一个操作执行的结果。

Happens-before 关系的规则有哪些?

- **程序次序规则**：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。
- **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**：对一个变量的写操作先行发生于后面对这个变量的读操作。
- **传递规则**：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C。
- **线程启动规则**：Thread对象的start()方法先行发生于此线程的每个一个动作。
- **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。
- **线程终结规则**：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行。
- **对象终结规则**：一个对象的初始化完成先行发生于他的finalize()方法的开始。

哪些符合happens-before规则：

答：比如单线程内的两个操作；锁操作：解锁的操作对于加锁的操作都是可见的；volatile 修饰的变量；中断规则，如果一个线程被其他线程 interrupt，那么在检测中断时，一定能看到此次中断的发生等等。

volatile 的作用是什么？与 synchronized 有什么异同?

答：volatile 第一个作用是保证可见性，第二个作用是禁止重排序（CPU和编译器优化语句会将语句重排序）

volatile 和 synchronized的异同：

相同点：

1.都可以保证线程安全。

不同点：

1.volatile 没有原子性和互斥性。

2.volatile 是无锁的，synchronized是有锁的

volatile 的适合场景：

适用场合1：布尔标记位

因为布尔值没有复合操作，姿势改变flag 的值，于是可以保证可见性。

适用场合2：触发器

省略

单例模式的双重检查锁模式为什么必须加 volatile？

单例模式双重检查锁模式的写法（懒汉式的线程安全写法）：

```
public class Singleton {
    private static volatile Singleton singleton;
    private Singleton() {
    }
    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

“为什么要 double-check[就是代码的 if (singleton == null)]？去掉任何一次的 check 行不行？”

答：第一个check是为了所有线程都可以进入（如果第一个线程获取到对象那么后面的线程就不能再创建对象了，**减少锁的获取**），保证性能第二个check为了保证只有一个Singleton实例化对象被创建。

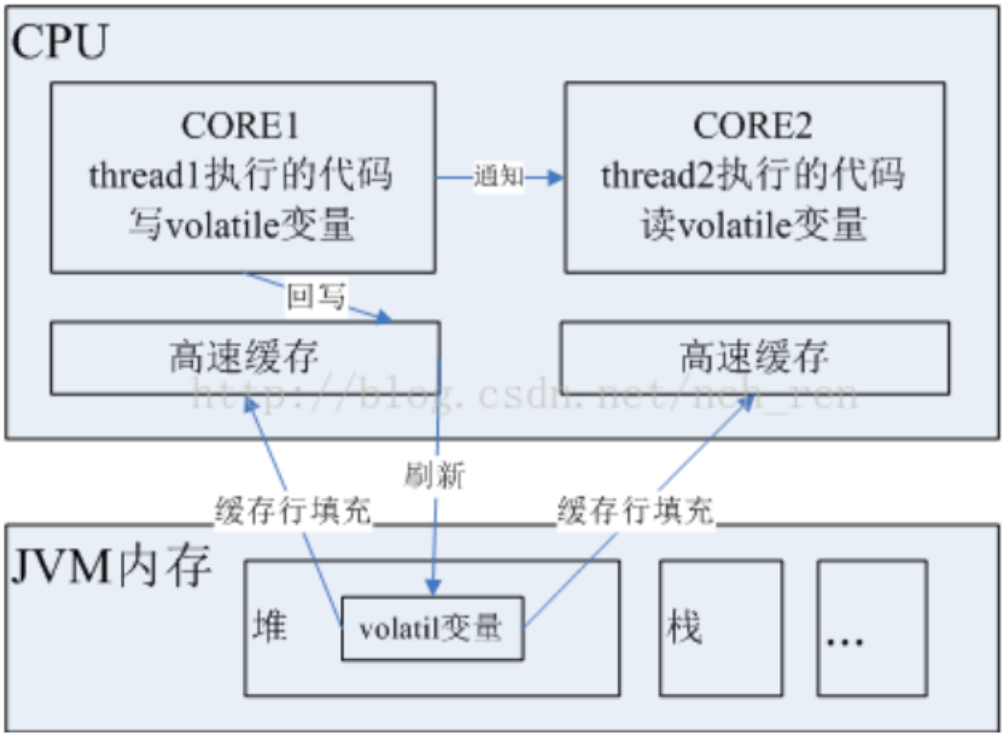
在双重检查锁模式中为什么需要使用 volatile 关键字：

答：因为singleton = new Singleton()有三步，如下图，为了防止CPU指令重排序的优化，将顺序打乱，所以加volatile 关键字：

volatile的底层原理?

在说这个问题之前，我们先看看CPU是如何执行java代码的。

首先编译之后Java代码会被编译成字节码.class文件，在运行时会被加载到JVM中，JVM会将.class转换为具体的CPU执行指令，CPU加载这些指令逐条执行。



以多核CPU为例（两核），我们知道CPU的速度比内存要快得多，为了弥补这个性能差异，CPU内核都会有自己的高速缓存区，当内核运行的线程执行一段代码时，首先将这段代码的指令集进行缓存行填充到高速缓存，如果非volatil变量当CPU执行修改了此变量之后，会将修改后的值回写到高速缓存，然后再刷新到内存中。如果在刷新会内存之前，由于是共享变量，那么CORE2中的线程执行的代码也用到了这个变量，这是变量的值依然是旧的。volatile关键字就会解决这个问题的，如何解决呢，首先被volatile关键字修饰的共享变量在转换成汇编语言时，会加上一个以lock为前缀的指令，当CPU发现这个指令时，立即做两件事：

1. 将当前内核高速缓存行的数据立刻回写到内存；
2. 使在其他内核里缓存了该内存地址的数据无效。

第一步很好理解，第二步如何做到呢？

MESI协议：在早期的CPU中，是通过在总线加LOCK#锁的方式实现的，但这种方式开销太大，所以Intel开发了缓存一致性协议，也就是MESI协议，该解决缓存一致性的思路是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，那么他会发出信号通知其他CPU将该变量的缓存行设置为无效状态。当其他CPU使用这个变量时，首先会去嗅探是否有对该变量更改的信号，当发现这个变量的缓存行已经无效时，会从新从内存中读取这个变量。

使用volatile的好处：从底层实现原理我们可以发现，volatile是一种非锁机制，这种机制可以避免锁机制引起的线程上下文切换和调度问题。因此，volatile的执行成本比synchronized更低。

```
1    volatile的不足：使用volatile关键字，可以保证可见性，但是却不能保证原子操作
```