1. Volatile关键字
2. 定义

Java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它锁单独获得这个变量。即只要变量被volatile修饰了，则Java可以确保所有线程看到这个变量的值都是一致的，某个线程对该变量进行更新时，则其他线程也能看到最新的，这就是内存可见性。

1. volatile实现内存可见性的过程

线程写volatile变量的过程：

1. 改变线程工作内存中该变量副本；
2. 将改变后的副本从工作内存刷新到主内存；

线程读volatile变量的过程：

1. 从主内存中读取volatile变量的最新值到线程的工作内存；
2. 从工作内存读取volatile变量的副本；
3. volatile实现内存可见性原理

写操作时，通过在写操作指令后添加一条store屏障指令，让工作内存的副本刷新到主内存；

读操作时，通过在读操作前添加load屏障指令，及时读取到主内存的值；

内存屏障是一种CPU指令，用于控制指定条件下的重排序和内存可见性问题，Java编译器也会根据内存屏障的规则禁止重排序。

1. volatile不具备原子性

volatile能够实现可见性，有序性，但是没有实现原子性，可以通过以下的方案解决：

1. 使用synchronized修饰修改共享变量的代码段，使该段代码具有原子性

public synchronized void addCount(){

cout++;

}

1. 使用ReentrantLock（可重入锁），手动加锁与解锁，性能比synchronized更好

private Lock lock = new ReentrantLock();

public void addCount(){

lock.lock();

count++;

lock.unlock();

}

1. 使用AtomicInteger（原子操作），该类型使用CAS操作，代替原有的Integer类，能够实现原子性，且底层没有使用锁，性能最高，推荐使用。当我们操作的类型有线程安全问题时，可以使用这种原子类型，以Atomic开头的基本数据类型。

public static AtomicInteger count = new AtomicInteger(0);

public void addCount(){

count.incrementAndGet();

}

1. volatile的使用场景

首先得是一个多线程的环境，其次多线程中不同线程对于该变量有读写操作，这样内存一致性才有发挥的空间，最后是能够使用其他的方法来解决volatile的原子性问题。总的来说是该变量要独立于其他的变量和该变量以前的值。

1. 与synchronized的比较

synchronized是一个重量级的锁，volatile是轻量级的，成本更低，因为volatile不会引起线程上下文的切换和调度。且synchronized底层实现中使用操作系统的互斥锁实现的，volatile的底层实现中没有使用锁，因此效率更高。

synchronized能够实现原子性，内存可见性，有序性；volatile只能实现有序性，内存可见性。