Java多线程中提到的原子性和可见性、有序性

1、原子性（Atomicity）

原子性是指在一个操作中就是cpu不可以在中途暂停然后再调度，既不被中断操作，要不执行完成，要不就不执行。

如果一个操作时原子性的，那么多线程并发的情况下，就不会出现变量被修改的情况

比如 a=0；（a非long和double类型） 这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++； 这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。

非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

（由Java内存模型来直接保证的原子性变量操作包括read、load、use、assign、store和write六个，大致可以认为基础数据类型的访问和读写是具备原子性的。如果应用场景需要一个更大范围的原子性保证，Java内存模型还提供了lock和unlock操作来满足这种需求，尽管虚拟机未把lock与unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐匿地使用这两个操作，这两个字节码指令反映到Java代码中就是同步块—synchronized关键字，因此在synchronized块之间的操作也具备原子性。）

2、可见性(Visibility)

可见性就是指当一个线程修改了线程共享变量的值，其它线程能够立即得知这个修改。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方法来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的

区别是volatile的特殊规则保证了新值能立即同步到主内存，以及每使用前立即从内存刷新。因为我们可以说volatile保证了线程操作时变量的可见性，而普通变量则不能保证这一点。

除了volatile之外，Java还有两个关键字能实现可见性，它们是synchronized。同步块的可见性是由“对一个变量执行unlock操作之前，必须先把此变量同步回主内存中(执行store和write操作)”这条规则获得的，而final关键字的可见性是指：被final修饰的字段是构造器一旦初始化完成，并且构造器没有把“this”引用传递出去，那么在其它线程中就能看见final字段的值。

（可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果。另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就这这个操作同样存在线程安全问题。）

3、有序性(Ordering)

Java内存模型中的程序天然有序性可以总结为一句话：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行语义”，后半句是指“指令重排序”现象和“工作内存中主内存同步延迟”现象。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则来获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

先行发生原则：

如果Java内存模型中所有的有序性都只靠volatile和synchronized来完成，那么有一些操作将会变得很啰嗦，但是我们在编写Java并发代码的时候并没有感觉到这一点，这是因为Java语言中有一个“先行发生”(Happen-Before)的原则。这个原则非常重要，它是判断数据是否存在竞争，线程是否安全的主要

依赖。

先行发生原则是指Jαvα内存模型中定义的两项操作之间的依序关系，如果说操作A先行发生于操作B，其实就是说发生操作B之前，操作A产生的影响能被操作B观察到，“影响”包含了修改了内存中共享变量的值、发送了消息、调用了方法等。它意味着什么呢？如下例：

```
//线程A中执行
i = 1;

//线程B中执行
j = i;

//线程C中执行
i = 2;
```

假设线程A中的操作”i=1“先行发生于线程B的操作”j=i“，那么我们就可以确定在线程B的操作执行后，变量j的值一定是等于1，结出这个结论的依据有两个，一是根据先行发生原则，”i=1“的结果可以被观察到；二是线程C登场之前，线程A操作结束之后没有其它线程会修改变量i的值。现在再来考虑线程C，我们依然保持线程A和B之间的先行发生关系，而线程C出现在线程A和B操作之间，但是C与B没有先行发生关系，那么j的值可能是1，也可能是2，因为线程C对应变量i的影响可能会被线程B观察到，也可能观察不到，这时线程B就存在读取到过期数据的风险，不具备多线程的安全性。

下面是Jαvα内存模型下一些”天然的“先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

α.程序次序规则(Pragram Order Rule)：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环结构。

b.管程锁定规则(Monitor Lock Rule)：一个unlock操作先行发生于后面对同一个锁的lock操作。这里必须强调的是同一个锁，而”后面“是指时间上的先后顺序。

c.volatile变量规则(Volatile Variable Rule)：对一个volatile变量的写操作先行发生于后面对这个变量的读取操作，这里的”后面“同样指时间上的先后顺序。

d.线程启动规则(Thread Start Rule)：Thread对象的start()方法先行发生于此线程的每一个动作。

e.线程终于规则(Thread Termination Rule)：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过Thread.join()方法结束，Thread.isAlive()的返回值等作段检测到线程已经终止执行。

f.线程中断规则(Thread Interruption Rule)：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.interrupted()方法检测是否有中断发生。

g.对象终结规则(Finalizer Rule)：一个对象初始化完成(构造方法执行完成)先行发生于它的finalize()方法的开始。

g.传递性(Transitivity)：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论。

一个操作”时间上的先发生“不代表这个操作会是”先行发生“，那如果一个操作”先行发生“是否就能推导出这个操作必定是”时间上的先发生“呢？也是不成立的，一个典型的例子就是指令重排序。所以时间上的先后顺序与先生发生原则之间基本没有什么关系，所以衡量并发安全问题一切必须以先行发生原则为准。