并发编程需要处理的两个问题：线程之间如何通信以及如何同步。通信一般有两种方式：**消息传递**和**共享内存**。

在共享内存的并发模型中，线程之间共享程序的公共状态，线程之间**通过写-读内存中的公共状态**来隐式通信。

**同步是指程序用于控制不同线程之间操作发生先后顺序的机制**。在共享内存模型中，同步是显式进行的。程序员必须显式指明某个方法或某段代码需要在线程之间互斥进行。

Java使用的是共享内存模型，因此这里不介绍消息传递。

内存模型(JMM)的抽象：

在Java中，所有**实例域、静态域和数组元素**(这里把他们统称为共享变量)存放在堆内存中，堆内存在线程之间共享。局部变量、方法定义参数、异常处理器参数不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型影响。

线程之间的共享变量存放在**主内存**(Main Memory)中，每个线程都有一个私有的共享变量本地副本。**本地内存**是一个抽象概念，并不实际存在。

示例：线程A和B都拥有共享变量x的副本，x的初始值为0(即主内存和两个本地内存的值都为0)。现A更新x的值为1(A的本地副本值变为1)。当A和B需要通信时，A把x的新值刷新到主内存，然后B再从主内存中读取x的新值。**疑问**：A更新了x值后，并不马上刷新到主内存？**解答**：刷新的时机不确定，如果x被synchronized、volatile、final、lock等同步原语修饰，则可确定。

**重排序**：

为了提高程序的执行性能，编译器和处理器常常会对指令做重排序。分为三种，如下：

1、编译器优化的重排序。在不改变单线程程序语义的前提下，可以重新安排语句执行顺序。

2、指令级并行的重排序。现代处理器使用了指令级并行技术(Instruction-Level Parallelism， ILP)来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3、内存系统的重排序。

从Java源代码到最终的机器指令，会分别经历以上三种重排序。在编译器级，JMM通过编译规则禁止某些重排序，在处理器级，JMM通过在指令序列中插入特定的内存屏障来禁止特定类型的重排序，从而保证不同平台下的内存可见性一致性。

如果两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。编译器和处理器不会改变存在数据依赖性的操作之间的执行顺序。(单个线程、单个处理器)

as-if-serial语义：不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。

**happens-before**：

在JDK5后，Java使用了JSR-133内存模型。JSR-133通过happens-before这个概念来阐述操作之间的内存可见性。如果一个操作的结果需要对另一个操作可见，那么这两个操作之间必须存在happens-before关系。这两个操作可以在用一个线程内，也可以在不同线程内。

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

监视器锁规则：对一个监视器锁的解锁，happens-before于随后对该监视器的加锁。

volatile变量规则：对一个volatile域的写，happens-before于任意后续对该变量的读。

传递性：如果A happens-before于B，B happens-before于C，则A happens-before于C。

**happens-before仅仅说明可见性，并没有规定执行的顺序性**。即，如果A happens-before于B，并不能说明A一定比B先执行。