A.

前端发送下订单请求后，会连接WebSocket一个包含用户userId的域名，后端成功下订单后，kafka监听到消息处理结果Topic中的消息，调用WebSocketServer的sendMessageToUser函数，通过session将处理的结果发到对应用户的端口。

使用线程安全的集合类型来维护 Session 的原因：

1.数据一致性：在多线程环境中，多个线程可能会同时访问和修改同一个 Session。如果不使用线程安全的集合类型，可能会导致数据不一致的问题，例如一个线程修改了 Session 中的数据，而另一个线程在读取时看到的是不完整或错误的数据。

2.避免竞争条件：在并发场景下，线程之间的竞争可能导致错误或不可预测的行为。线程安全的集合类型通过内置的同步机制，确保在任何时刻只有一个线程能够访问或修改数据，从而避免了竞争条件的发生。

3.简化代码：使用线程安全的集合类型可以简化并发编程的复杂性。开发者不必手动实现锁或其他同步机制，从而减少了出错的机会，并提高了代码的可读性和可维护性。

ConcurrentHashMap 是线程安全的原因：

ConcurrentHashMap 是 Java 中提供的一种线程安全的哈希表实现，其线程安全性主要体现在以下几个方面：

1. 分段锁（Segment Locking）：ConcurrentHashMap 将整个哈希表划分为多个段，每个段可以独立地进行操作。这样，在一个段上的操作不会阻塞其他段的操作，从而提高了并发性能。

2. 无锁读取：对于读取操作，ConcurrentHashMap 采用了无锁的方式，这意味着在进行读取时不需要获取锁，从而提高了读取的速度和性能。

3. 细粒度锁：在写入操作时，ConcurrentHashMap 采用了细粒度锁策略，只在必要的情况下锁定特定的段，而不是整个哈希表。这种策略使得多个线程可以同时对不同的段进行写操作，从而减少了线程之间的阻塞。

4. 使用 volatile 关键字：ConcurrentHashMap 内部使用 volatile 关键字来确保内存的可见性，确保一个线程对集合的修改对其他线程是可见的，从而避免了由于线程缓存导致的数据不一致问题。

B

i.不断地将事务操作的结果执行落盘操作

潜在问题：

1.性能开销大：每次操作都直接写入磁盘，会带来频繁的磁盘I/O操作。由于磁盘的I/O速度远低于内存的速度，频繁的写入操作会严重影响系统性能，导致事务处理效率大大降低。

2.无效或冗余操作：在事务尚未完成之前，可能会有许多不必要的磁盘写入操作。比如，某个事务在执行到中途发生了回滚，则前面已经写入磁盘的操作会变得无效，浪费资源。

解决方案：

1.使用缓冲区和日志：数据库可以先将操作的结果写入内存缓冲区（Buffer Pool），并在适当的时机（例如事务提交时）将结果写入磁盘。这减少了频繁的磁盘I/O操作。

2.采用延迟写策略：通过延迟写入机制（例如批量写入或定时写入），减少每次操作都立即写入磁盘的频率。这可以大幅提升性能。

3.使用undo日志：在进行落盘操作之前，首先将操作日志写入到日志文件中，这样即便系统发生故障，也可以通过日志进行恢复，保证持久性。

ii.事务执行提交后再将事务操作的结果执行落盘操作

潜在问题：

1.数据丢失风险：如果在事务提交后还未落盘的时间窗口内发生系统故障，事务的结果可能尚未写入磁盘，导致事务的结果丢失。这违背了持久性的要求。

2.恢复成本高：如果事务提交后立即发生系统故障，而结果尚未持久化，系统恢复时需要重新执行较多的操作，增加恢复的复杂度和成本。

解决方案：

1.使用redo日志：在事务提交之前，首先将事务的操作记录到日志文件中。在事务提交后，如果发生故障，可以根据日志进行恢复，即使数据尚未完全写入磁盘，仍然可以通过redo日志（log）确保数据的持久性。

2.事务提交时同步写入日志和数据：在事务提交时，确保在日志文件和数据库数据文件中同时写入相关数据，从而减少数据丢失的风险。

3.启用自动检查点（Checkpoint）机制：定期创建数据库的检查点，确保事务提交的结果已经写入磁盘，从而减少恢复期间需要处理的日志量，提升故障后的恢复效率。