HW 3

WebSocket 消息格式

```Java
 CreateOrderResponse  {  // 非代码，仅体现消息内容和格式
    String status;
    Integer orderId;
    Integer userId;
}
```

效果如图（实际上，可以只返回 status）：

筛选客户端的方式设计

通过 `Kafka` 的 `topicListener` 获得创建订单的消息后，进行订单创建，并获得订单创建的返回值 `CreateOrderResponse`，直接在当前 topic 内使用 `webSocketServer` 类实现的 `sendMessageToUser` 方法根据用户 ID 将结果返回给创建订单的用户。

```Java
public void sendMessageToUser(String userId, String message) {
    System.out.println(userId);
    Session toSession = SESSIONS.get(userId);
    log.info("send message "+ message);
    sendMessage(toSession, message);
}
```

附：

原计划在一个新的 `topic` 中返回 `WebSocket` 的消息，但是由于在使用 `Kafka` 的过程中，默认的 Serializer 不支持直接传递对象信息，在尝试许久自定义 `Serializer` 失败之后选择了目前的处理方式，失败的原因是 kafkaTemplate 传递的自定义消息被 `topic` 以 `CustomRecord` 接收之后默认识别为 `<String, String>` 格式（即使更改也没用）。

由于时间问题未进行更深入探究，之后的改进方案（2 选 1）：

- 传递 String 消息，采用 Encoder 和 Decoder 进行编/解码来传递对象信息
- 放弃 producer、consumer 结合的 kafkaTemplate，自定义 kafka producer 和 consumer

为什么选择线程安全的集合类

在并发情况下，使用线程不安全的数据结构可能会产生并发错误（**线程体当中连续的多行操作未必能够连续执行很可能操作只完成了一部分，时间片突然耗尽，此时，另一个线程抢到时间片，直接拿走并访问了操作不完整的数据**）。

比较直观的理解是，多个用户同时建立连接，某个用户访问 session 链表/hash 表时，在他之前建立连接的用户还并未将自己的 session 信息写入，导致该用户拿到的数据不完整，当这个后来的用户再将自己的 session 信息写入后，这两人的 session 信息可能只有一个会被保留（视数据结构而定），这就导致了并发错误。

为什么 ConcurrentHashMap 是线程安全的

在 JDK1.8 版本中 `ConcurrentHashMap` 采用了 `CAS+synchronized` 的方法来保证并发，线程安全。

> CAS：Compare and Swap，即比较再交换。**CAS 是一种操作系统级别的支持，是用来更新数据的原始操作，是一个原子操作。**CAS 有 3 个参数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。

以 `ConcurrentHashMap` 的 `put` 操作为例，它先判断 hash 后的位置是否为 null，若为 null，则用 `CAS` 写入，若 `CAS` 写入失败，则用自旋保证写入成功，当前 hash 值等于 MOVED(-1)时，需要进行扩容，若以上都不满足，则采用 `synchronized` 阻塞锁，来将数据进行写入，整个过程是线程安全且支持并发的。