张子谦

作业二 Kafka 消息中间件处理订单

> 张子谦：52011190121
>
> 本次实现功能：Kafka消息中间件和Websocket实现对前端的消息通知。

一、下订单的消息通讯逻辑

a) 控制器类

接受前端传递来的POST参数，检查参数的数量是否合乎规定。如果合规定，产生一个UUID，作为这个订单请求的唯一标识符，然后通过卡夫卡消息中间件，写入到订单队伍中。

为什么需要UUID呢，因为这个UUID是返回给前端的，前端通过这个UUID作为一个客户端标识符然后开启一个WebSocket会话，通过这个UUID，服务器后端可以主动的把订单处理的消息推给前端。（提前预习了一下第五课的内容，如果有说的不对的地方助教老师请谅解 ^_^

那么还有一个问题，是在前端生成UUID还是在后端生成，我这里仔细考虑了一下，如果前端生成UUID，作为参数传递过来，那么如果有人用POST发一连串相同的请求（也就是UUID全都是一样的），那么会导致卡夫卡消息中间件信箱里面的key全都是一样了，这一点不是我想要的，为了避免伪造UUID，所以把UUID我决定放在后端生成，下发给前端。那么后端生成有什么缺点呢，后端接到请求马上就生成uuid，然后返给前端，我的前端处理的方法就是一个重定向到一个下单完成的页面，同时附上请求参数uuid，在这个页面组件开始渲染的同时WebSocket连接就开启了，经过我实际测试，如果在订单不是特别臃肿的情况下，理论来说是这么个顺序：

1. 用户下单
2. 后端完成订单处理，开始向前端发送信息
3. 客户端开启 Websocket 的回话，客户端上线
4. 前端页面跳转到订单已经收到页面
5. 前端收到后端发来的通知

当然在订单非常臃肿的情况下

1. 用户下单
2. 前端页面跳转到订单已经收到页面
3. 客户端开启 Websocket 的回话，客户端上线
4. 后端完成订单处理，开始向前端发送信息
5. 前端用户收到后端发来的通知，处理完成订单

所以这里就要求后端可能需要发多次消息，直到发送消息成功，才停止。因此我对于后端的代码sendToUser这一个函数设置了多次发送消息的机制，中间间隔时间为1s，一共发送十次，如果没有结果就不发送。

服务器端控制器类完成订单的处理：

- 首先校验用户的传递过来的参数的数量（数量不对直接不处理）
- 创建UUID，然后通过卡夫卡消息中间件传递到信箱Topic里面，标注key为后端生成的UUID
- 返回给前端用户订单的信息已经收到。

```
    @RequestMapping("/order/makeorder")
    public Msg orderMake(@RequestBody Map<String, String> params) throws Exception
{
        // 校验一下参数的数量
        int itemNum = (params.size() - 6) / 2 ;
        if(itemNum <= 0)
            return null;
        // 创建一个UUID
        String Order_UUID = UUID.randomUUID().toString().toUpperCase();
        JSONObject data = new JSONObject();
        data.put("uuid",Order_UUID);
        kafkaTemplate.send("orderQueue", Order_UUID, params.toString());
        return MsgUtil.makeMsg(MsgCode.SUCCESS, MsgUtil.SUCCESS_MSG,data);
    }
```

b）监听器类

首先，我设置了两个卡夫卡信箱Topic

- 第一个信箱用来存放**控制器接收到前端发来的订单的数据**（前端POST请求的数据），我这里的处理是前端的POST请求的Body参数**作为一个字符串整体**全部放入这个信箱，然后把这个字符串再在处理的时候解析转化为Map对象，然后开始处理。这样的优点是：节约了对字符串解析的时间，控制器只需要校验参数数量对不对，至于后面的参数问题在监听器中处理。
- 第二个信箱是**处理完成的消息队列**。

那么，至于监听器，我们设置两个监听器：

- 第一个监听器监听上述的第一个订单信箱：处理前端传来的订单参数，然后通过服务层与数据库交互。
- 第二个监听器监听上述的第二个结果信箱，把信箱中的订单的处理结果返回给用户。

```
package com.zzq.ebook.listener;

import com.zzq.ebook.constant.constant;
import com.zzq.ebook.service.OrderService;
import com.zzq.ebook.utils.tool.ToolFunction;
import com.zzq.ebook.utils.websocket.WebSocketServer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;
import java.util.Map;
import java.util.Objects;

@Component
public class OrderListener {
    @Autowired
    private OrderService orderService;
    @Autowired
```

```
    private KafkaTemplate<String, String> kafkaTemplate;
    @Autowired
    private WebSocketServer webSocketServer;

    @KafkaListener(topics = "orderQueue", groupId = "group_topic_order")
    public void orderQueueListener(ConsumerRecord<String, String> record) throws Exception {
        System.out.println(record.value());
        Map<String,String> params = ToolFunction.mapStringToMap(record.value());
        int itemNum = (params.size() - 6) / 2 ;
        String orderFrom = params.get("orderFrom");
        String username = params.get(constant.USERNAME);
        String receivename = params.get("receivename");
        String postcode = params.get("postcode");
        String phonenumber = params.get("phonenumber");
        String receiveaddress = params.get("receiveaddress");
        int[] bookIDGroup = new int[itemNum];
        int[] bookNumGroup = new int[itemNum];

        for(int i=1; i<=itemNum; i++){
            bookIDGroup[i-1] = Integer.parseInt(params.get("bookIDGroup" + i));
            bookNumGroup[i-1] = Integer.parseInt(params.get("bookNumGroup" + i));
        }

        // 根据购买的来源，把数组交给服务层业务函数
        int result = -1;
        if(Objects.equals(orderFrom, "ShopCart")) {
            result = orderService.orderMakeFromShopCart(bookIDGroup,bookNumGroup,username,receivename,
                    postcode, phonenumber, receiveaddress,itemNum);
        }
        else if(Objects.equals(orderFrom, "DirectBuy")){
            result = orderService.orderMakeFromDirectBuy(bookIDGroup,bookNumGroup,username,receivename,
                    postcode, phonenumber, receiveaddress,itemNum);
        }

        kafkaTemplate.send("orderFinished",  record.key(), "Done Order");
    }

    @KafkaListener(topics = "orderFinished", groupId = "group_topic_order")
    public void orderFinishedListener(ConsumerRecord<String, String> record) throws InterruptedException {
        String value = record.key();
        System.out.println("orderFinishedListener 输出" + value);
        webSocketServer.sendMessageToUser(value, record.value());
    }

}
```

c）前端部分

- 前端我单独创建了一个组件，作为一个WebSocket连接的组件
- 该组件可以创建连接、关闭连接、不断的发送心跳包，以及发送消息。
- 创建WebSocket连接的时候，需要将后端对应的URL，已经收到消息的处理函数传递到这个`createWebSocket()`函数进来。

```
let websocket, lockReconnect = false;
let createWebSocket = (url, handleEvent) => {
    websocket = new WebSocket(url);
    websocket.onopen = function () {
        heartCheck.reset().start();
    }
    websocket.onerror = function () {
        reconnect(url);
    };
    websocket.onclose = function (e) {
        console.log('websocket 断开: ' + e.code + ' ' + e.reason + ' ' +
e.wasClean)
    }
    websocket.onmessage = function (event) {
        lockReconnect=true;
        handleEvent(event);
        //event 为服务端传输的消息，在这里可以处理
    }
}
let reconnect = (url) => {
    if (lockReconnect) return;
    // 没连接上会一直重连，设置延迟避免请求过多
    setTimeout(function () {
        createWebSocket(url);
        lockReconnect = false;
    }, 4000);
}
//60秒间歇性检查
let heartCheck = {
    timeout: 60000,
    timeoutObj: null,
    reset: function () {
        clearInterval(this.timeoutObj);
        return this;
    },
    start: function () {
        this.timeoutObj = setInterval(function () {
            // 这里发送一个心跳，后端收到后，返回一个心跳消息，
            // onmessage拿到返回的心跳就说明连接正常
            websocket.send("HeartBeat");
        }, this.timeout)
    }
```

```
}
//关闭连接
let closeWebSocket=()=> {
    websocket && websocket.close();
}
export {
    websocket,
    createWebSocket,
    closeWebSocket
};

```

d) WebSocket的后端部分

- 这一部分代码的实现参考了下一个章节给的样例代码
- 代码中：`"/websocket/transfer/{userId}"`的`userID`使用的是前面说的UUID

```
package com.zzq.ebook.utils.websocket;
import com.sun.org.apache.bcel.internal.generic.RETURN;
import org.springframework.stereotype.Component;

import javax.websocket.*;
import javax.websocket.server.PathParam;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;


@ServerEndpoint("/websocket/transfer/{userId}")
@Component
public class WebSocketServer {
    public WebSocketServer() {
        //每当有一个连接，都会执行一次构造方法
        System.out.println("新的连接已经开启");
    }
    private static final AtomicInteger COUNT = new AtomicInteger();
    private static final ConcurrentHashMap<String, Session> SESSIONS = new
ConcurrentHashMap<>();
    public int sendMessage(Session toSession, String message) {
        if (toSession != null) {
            try {
                toSession.getBasicRemote().sendText(message);
                return 0;
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("发送的时候对方不在线");
```

```
            return 1;
        }
        return 1;
    }

    public void sendMessageToUser(String user, String message) throws
InterruptedException {
        for (int i = 0; i < 10; i++){
            Session toSession = SESSIONS.get(user);
            if(sendMessage(toSession, message) == 0)
                return;
            Thread.sleep(1000);
        }
    }

    @OnMessage
    public void onMessage(String message) {
        System.out.println("服务器收到消息：" + message);
    }

    @OnOpen
    public void onOpen(Session session, @PathParam("userId") String userId) {
        if (SESSIONS.get(userId) != null) {
            return;
        }
        SESSIONS.put(userId, session);
        COUNT.incrementAndGet();
        System.out.println(userId + "加入，当前在线人数：" + COUNT);
    }
    @OnClose
    public void onClose(@PathParam("userId") String userId) {
        SESSIONS.remove(userId);
        COUNT.decrementAndGet();
        System.out.println(userId + "退出，当前在线人数：" + COUNT);
    }

    @OnError
    public void onError(Session session, Throwable throwable) {
        System.out.println("发生错误");
        throwable.printStackTrace();
    }
}
```

二、下订单的效果截图

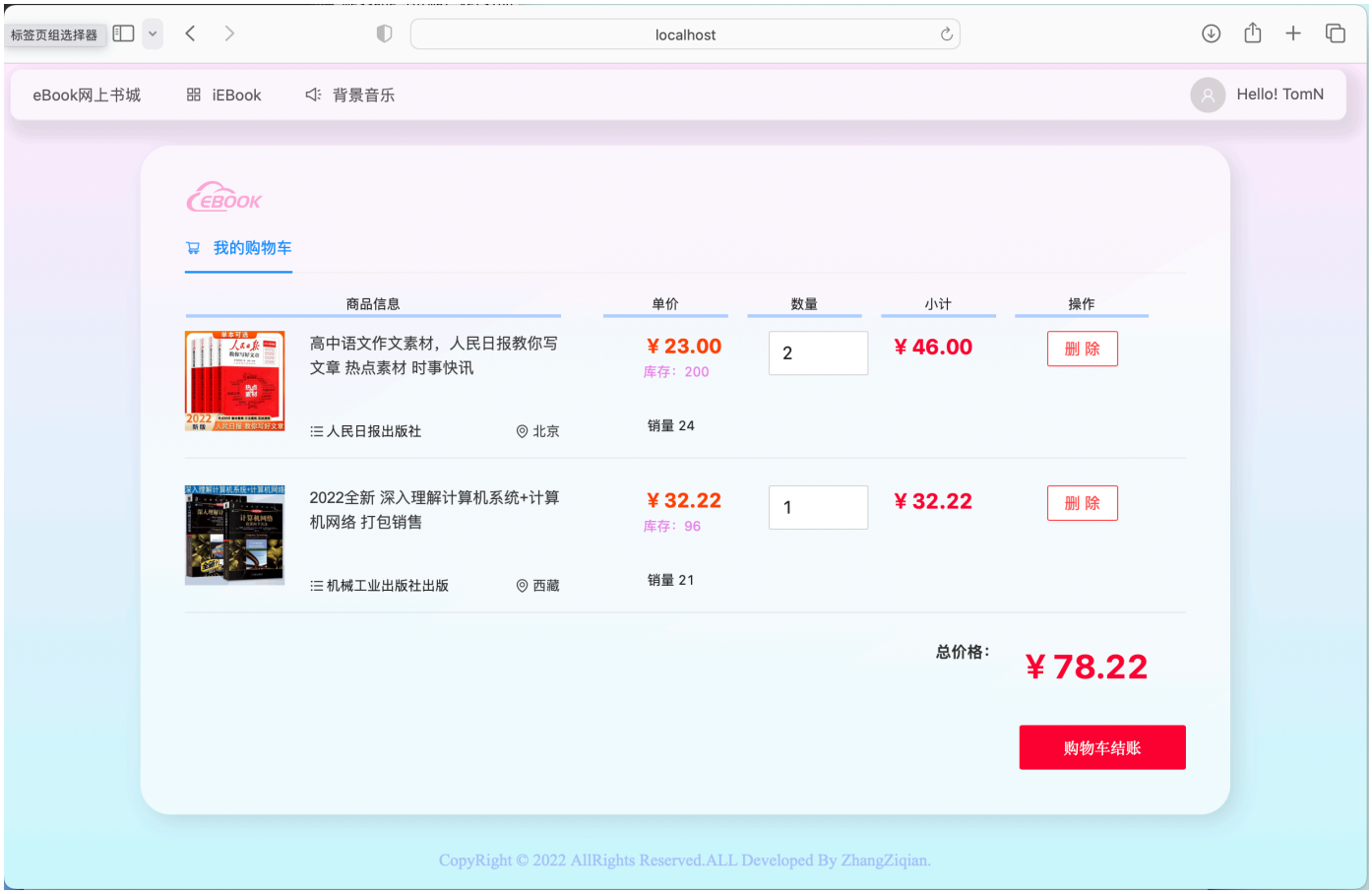
在正式启动服务器之前，务必需要执行下面的两个代码开启卡夫卡消息中间件。然后才能启动后端：

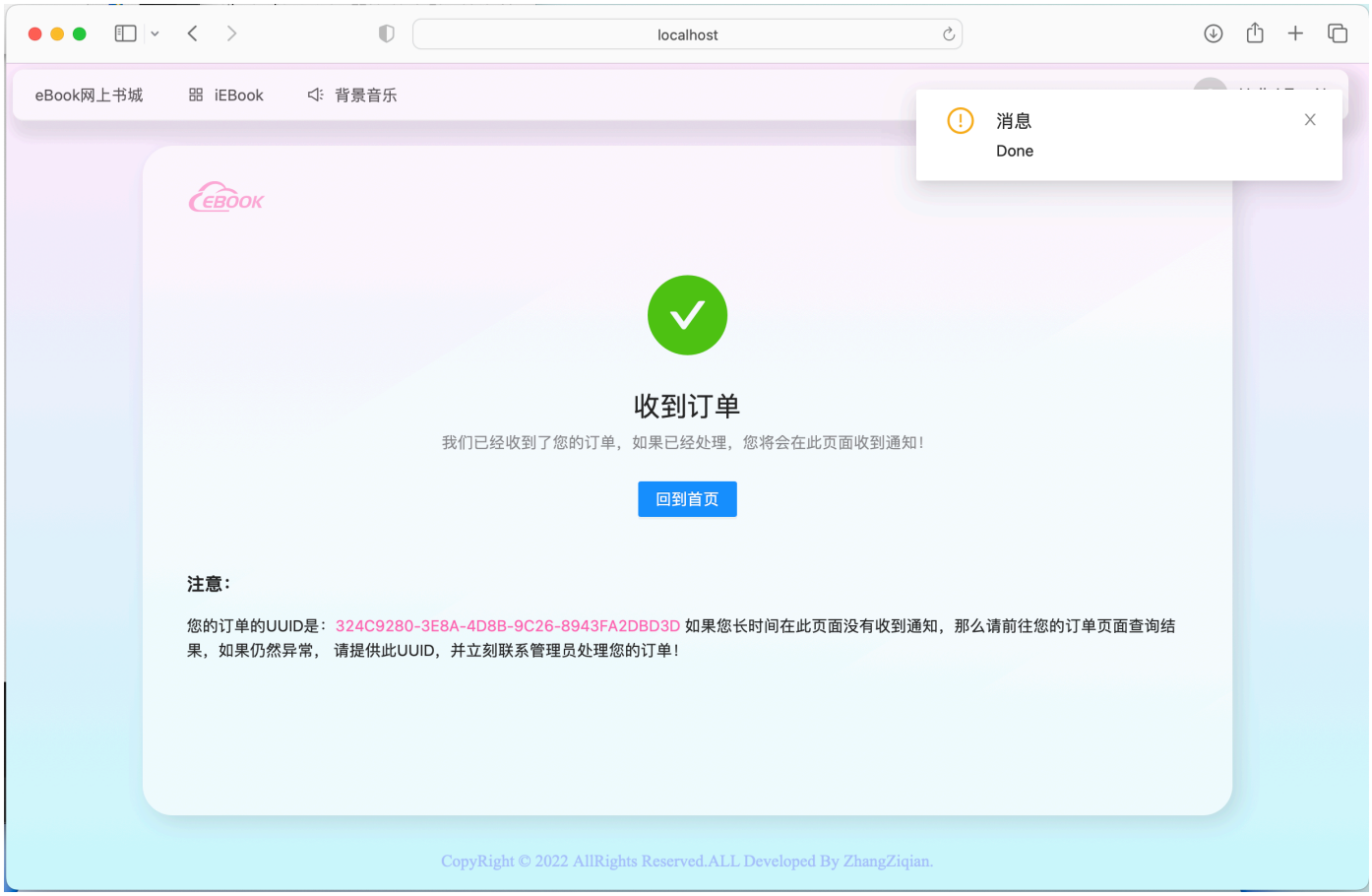
```
# Start the ZooKeeper service
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

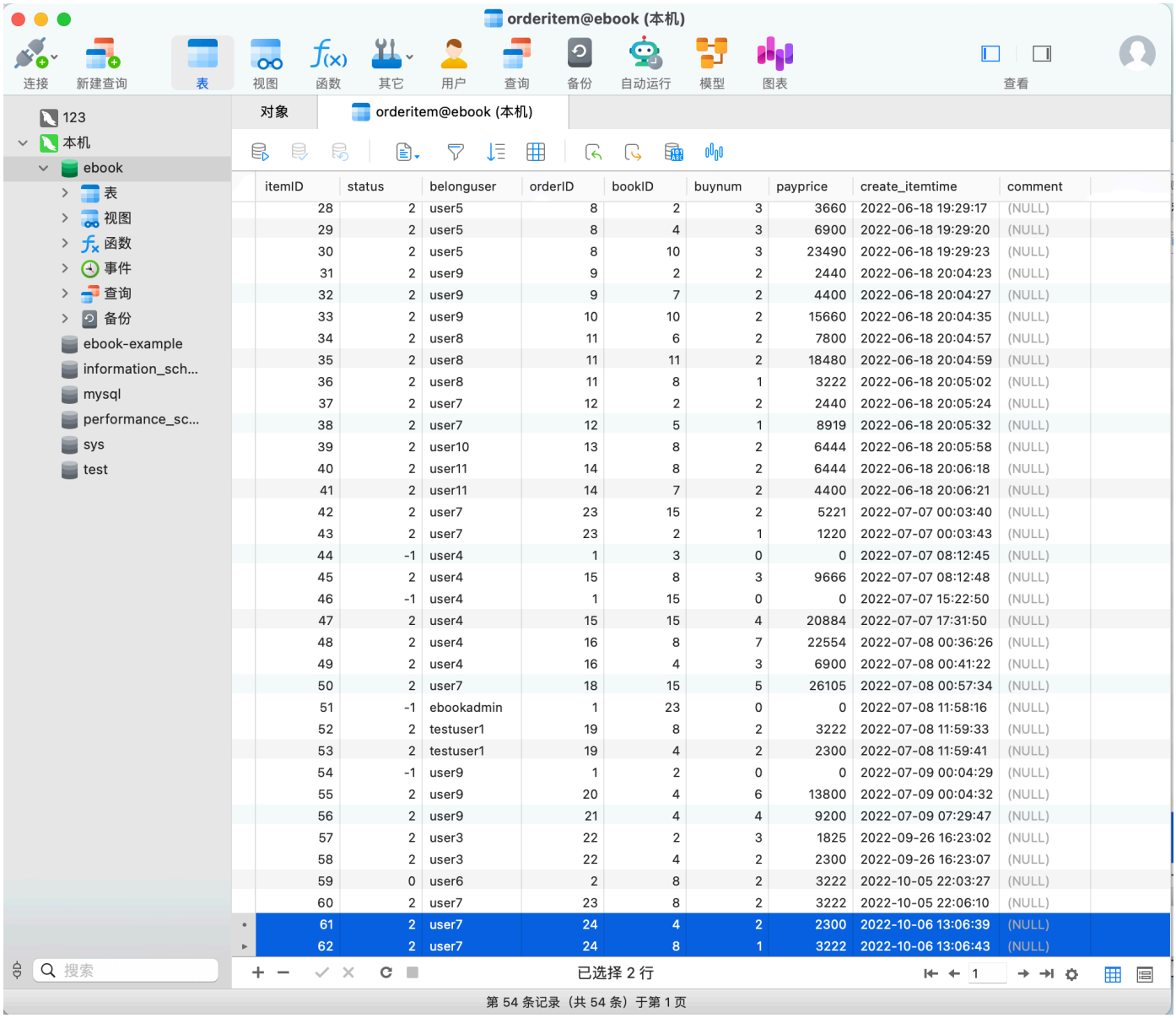
```
# Start the Kafka broker service
$ bin/kafka-server-start.sh config/server.properties
```

- 用户单机下订单完成后，会被跳转到一个结果页面，这个页面会展示订单的UUID参数
- 此时，WebSocket连接已经建立，一旦后端处理好了订单，那么就会通知前端，以消息框的形式呈现

为了说明下定的结果是符合逻辑的，请依次看下面的购物车截图、下单结果截图、数据库截图。




localhost
eBook网上书城
iEBook
背景音乐
消息
Done
收到订单
我们已经收到了您的订单，如果已经处理，您将会在此页面收到通知！
回到首页
注意：
您的订单的UUID是：324C9280-3E8A-4D8B-9C26-8943FA2DBD3D 如果您长时间在此页面没有收到通知，那么请前往您的订单页面查询结果，如果仍然异常， 请提供此UUID，并立刻联系管理员处理您的订单！
CopyRight © 2022 AllRights Reserved.ALL Developed By ZhangZiqian.



itemID	status	belonguser	orderID	bookID	buynum	payprice	create_itemtime	comment
28	2	user5	8	2	3	3660	2022-06-18 19:29:17	(NULL)
29	2	user5	8	4	3	6900	2022-06-18 19:29:20	(NULL)
30	2	user5	8	10	3	23490	2022-06-18 19:29:23	(NULL)
31	2	user9	9	2	2	2440	2022-06-18 20:04:23	(NULL)
32	2	user9	9	7	2	4400	2022-06-18 20:04:27	(NULL)
33	2	user9	10	10	2	15660	2022-06-18 20:04:35	(NULL)
34	2	user8	11	6	2	7800	2022-06-18 20:04:57	(NULL)
35	2	user8	11	11	2	18480	2022-06-18 20:04:59	(NULL)
36	2	user8	11	8	1	3222	2022-06-18 20:05:02	(NULL)
37	2	user7	12	2	2	2440	2022-06-18 20:05:24	(NULL)
38	2	user7	12	5	1	8919	2022-06-18 20:05:32	(NULL)
39	2	user10	13	8	2	6444	2022-06-18 20:05:58	(NULL)
40	2	user11	14	8	2	6444	2022-06-18 20:06:18	(NULL)
41	2	user11	14	7	2	4400	2022-06-18 20:06:21	(NULL)
42	2	user7	23	15	2	5221	2022-07-07 00:03:40	(NULL)
43	2	user7	23	2	1	1220	2022-07-07 00:03:43	(NULL)
44	-1	user4	1	3	0	0	2022-07-07 08:12:45	(NULL)
45	2	user4	15	8	3	9666	2022-07-07 08:12:48	(NULL)
46	-1	user4	1	15	0	0	2022-07-07 15:22:50	(NULL)
47	2	user4	15	15	4	20884	2022-07-07 17:31:50	(NULL)
48	2	user4	16	8	7	22554	2022-07-08 00:36:26	(NULL)
49	2	user4	16	4	3	6900	2022-07-08 00:41:22	(NULL)
50	2	user7	18	15	5	26105	2022-07-08 00:57:34	(NULL)
51	-1	ebookadmin	1	23	0	0	2022-07-08 11:58:16	(NULL)
52	2	testuser1	19	8	2	3222	2022-07-08 11:59:33	(NULL)
53	2	testuser1	19	4	2	2300	2022-07-08 11:59:41	(NULL)
54	-1	user9	1	2	0	0	2022-07-09 00:04:29	(NULL)
55	2	user9	20	4	6	13800	2022-07-09 00:04:32	(NULL)
56	2	user9	21	4	4	9200	2022-07-09 07:29:47	(NULL)
57	2	user3	22	2	3	1825	2022-09-26 16:23:02	(NULL)
58	2	user3	22	4	2	2300	2022-09-26 16:23:07	(NULL)
59	0	user6	2	8	2	3222	2022-10-05 22:03:27	(NULL)
60	2	user7	23	8	2	3222	2022-10-05 22:06:10	(NULL)
61	2	user7	24	4	2	2300	2022-10-06 13:06:39	(NULL)
62	2	user7	24	8	1	3222	2022-10-06 13:06:43	(NULL)

三、结果与思考分析

根据本次作业的要求，有三种方法展示前端页面：

1. 在前端工程中，使用JavaScript监听订单处理结果消息发送到的Topic，然后刷新页面；
2. 在前端发送Ajax请求获取订单的最新状态，后端接收到请求后将订单状态返回给前端去显示；
3. 采用WebSocket 方式，后端的消息监听器类监听到消息处理结果Topic 中的消息后，通过WebSocket发送给前端；

我选择的方法是**第三种**，理由如下：

- WebSocket是一种主动的方式，后端可以主动的把结果推送给前端的客户端，而不需要前段通过定时设定，然后来不断的查询我们的订单结果。这就类比我们课上讲过的改作业，下订单就好比学生把作业交给老师（服务器），服务器改完作业主动通知学生来拿，如果采用的是1或者2的方法，需要学生每隔一段时间就来询问老师作业是否改完了，这浪费并且消耗了连接和资源，是一种不合理的方式，所以我这里果断弃用了前面的两种。归结来说三个优点
    - **推送功能**：支持服务器端向客户端推送功能。服务器可以直接发送数据而不用等待客户端的请求。
    - **减少通信量**：只要建立起websocket连接，就一直保持连接，在此期间可以源源不断的传送消息，直到关闭请求。也就避免了HTTP的非状态性。和http相比，不但每次连接时的总开销减少了，而且

websocket的首部信息量也小，通信量也减少了。
    - **减少资源消耗**：如果用AJax轮询的话，我们需要专门设置一个接口，运行相关的查询代码，而且由于前端是定时的不断的发请求来查询，相关的查询结果的代码要运行很多次，这浪费了资源，反而如果只用WebSocket，推送代码只用运行一次。
    - 但是也有缺点：比如需要浏览器支持WebSocket，例如我的Safari浏览器在WebSocket的测试中出现了一些问题（由于一些安全性的原因，但是Edge浏览器就可以正常保证WebSocket的连接），同时如果只是单页面涉及到WebSocket还好，涉及到多页面，定时推送，复杂的推送，就非常容易出问题了，不管是前端，还是后端都会遇到一些问题。
- **使用JavaScript监听的优缺点分析：**
    - 优点就是直接交互，简洁明了，降低了后端Spring的压力(不需要单独写一个接口，单独运行相关的Kafuka查询组件的信息)。
    - 缺点也是直接交互，使用JS监听，相当于用户客户端直接和信箱交互，这就会导致我的卡夫卡Topic信箱直接暴露，我认为这是一个不好、不安全的方式。卡夫卡信箱里面涉及到用户的订单数据，所以使用后端直接交互，然后后端Spring暴露访问接口，更加安全。
- **使用Ajax轮询的方式优缺点分析：**
    - 相比较于上个方法JS直接跟Topic信箱交互，这个不安全，所以Ajax轮询的话，单独有一个查询接口（还可以增加鉴权），所以比较安全
    - 缺点相比较于上个JS直接交互的方法，这个是间接交互，所以消耗资源，更耗时间。