

# 一、预约下单

## 1、生成订单前端整合

### (1) 封装api方法

创建api/orderinfo.js

```
1 import request from '@/utils/request'
2
3 const api_name = `/api/order/orderInfo`
4
5 export default {
6   submitOrder(scheduleId, patientId) {
7     return request({
8       url: `${api_name}/auth/submitOrder/${scheduleId}/${patientId}`,
9       method: 'post'
10    })
11  }
12 }
```

### (2) 在booking.vue组件完善下单方法

```
1 submitOrder() {
2   if(this.patient.id == null) {
3     this.$message.error('请选择就诊人')
4     return
5   }
6   // 防止重复提交
7   if(this.submitBnt == '正在提交...') {
8     this.$message.error('重复提交')
9     return
10  }
11
12  this.submitBnt = '正在提交...'
13  orderInfoApi.submitOrder(this.scheduleId, this.patient.id).then(response => {
14    let orderId = response.data
```

```

15     window.location.href = '/order/show?orderId=' + orderId
16   }).catch(e => {
17     this.submitBnt = '确认挂号'
18   })
19 },

```

## 2、生成订单后处理逻辑-更新订单信息

预约成功后我们要 **更新订单信息**，**更新预约数** 和 **短信提醒预约成功**，为了提高下单的并发性，这部分逻辑我们就交给mq为我们完成，预约成功发送消息即可

### (1) 修改OrderServiceImpl方法

```

1  //生成预约挂号订单
2  @Override
3  public Long saveOrders(String scheduleId, Long patientId) {
4
5      .....
6
7      //发送请求,调用医院模拟系统接口，完成下单
8      //使用工具类方法
9      JSONObject result=
10         HttpRequestHelper.sendRequest(paramMap,"http://localhost:9998/order/su
11      //根据医院模拟系统返回数据，进行处理，如果返回状态码200，下单成功，否则下单失败
12      if(result.getInteger("code")==200) { //下单成功
13         System.out.println("下单成功");
14         JSONObject jsonObject = result.getJSONObject("data");
15         //预约记录唯一标识（医院预约记录主键）
16         String hosRecordId = jsonObject.getString("hosRecordId");
17         //预约序号
18         Integer number = jsonObject.getInteger("number");
19         //取号时间
20         String fetchTime = jsonObject.getString("fetchTime");
21         //取号地址
22         String fetchAddress = jsonObject.getString("fetchAddress");
23         //更新订单
24         orderInfo.setHosRecordId(hosRecordId);
25         orderInfo.setNumber(number);
26         orderInfo.setFetchTime(fetchTime);
27         orderInfo.setFetchAddress(fetchAddress);

```

```
28     baseMapper.updateById(orderInfo);
29     //排班可预约数
30     Integer reservedNumber = jsonObject.getInteger("reservedNumber");
31     //排班剩余预约数
32     Integer availableNumber = jsonObject.getInteger("availableNumber");
33     //发送mq信息更新号源和短信通知
34
35     } else { //下单失败
36         System.out.println("下单失败");
37         throw new YyghException(20001, "下单失败");
38     }
39     //返回订单号
40     return orderInfo.getId();
41 }
```

### 3、生成订单后处理逻辑-rabbit-util模块封装

#### (1) rabbitMQ简介

以商品订单场景为例，

如果商品服务和订单服务是两个不同的微服务，在下单的过程中订单服务需要调用商品服务进行扣库存操作。按照传统的方式，下单过程要等到调用完毕之后才能返回下单成功，如果网络产生波动等原因使得商品服务扣库存延迟或者失败，会带来较差的用户体验，如果在高并发的场景下，这样的处理显然是不合适的，那怎么进行优化呢？这就需要消息队列登场了。

消息队列提供一个异步通信机制，消息的发送者不必一直等待到消息被成功处理才返回，而是立即返回。消息中间件负责处理网络通信，如果网络连接不可用，消息被暂存于队列当中，当网络畅通的时候在将消息转发给相应的应用程序或者服务，当然前提是这些服务订阅了该队列。如果在商品服务和订单服务之间使用消息中间件，既可以提高并发量，又降低服务之间的耦合度。

RabbitMQ就是这样一款消息队列。RabbitMQ是一个开源的消息代理的队列服务器，用来通过普通协议在完全不同的应用之间共享数据。

#### 典型应用场景：

异步处理。把消息放入消息中间件中，等到需要的时候再去处理。


流量削峰。例如秒杀活动，在短时间内访问量急剧增加，使用消息队列，当消息队列满了就拒绝响应，跳转到错误页面，这样就可以使得系统不会因为超负载而崩溃

## (2) 安装rabbitMQ

```
1 #拉取镜像
2 docker pull rabbitmq:3.8-management
3
4 #创建容器启动
5 docker run -d --restart=always -p 5672:5672 -p 15672:15672 --name rabbitmq ra
```

## (3) 服务rabbitMQ后台


管理后台: <http://IP:15672>

 RabbitMQ Management

×

+

http://192.168.44.165:15672/

  
Username:  \*  
Password:  \*

# Overview

Totals

Queued messages

last minute

?

Currently idle

Message rates

last minute

?

Currently idle

Global counts

?

Connections: 0

Channels: 0

Exchanges: 7

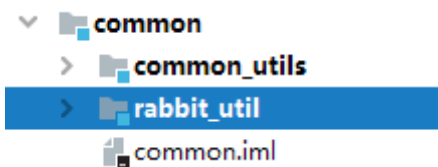
Queues: 0

Consumers: 0

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?
rabbit@256140967bc6	106 65536 available	0 58893 available	557 1048576 available	86 MiB 693 MiB high water

## (4) 在common搭建rabbit\_util模块



New Module

Add as module to

com.atguigu:common:0.0.1-SNAPSHOT

Parent

com.atguigu:common:0.0.1-SNAPSHOT

GroupId

com.atguigu

ArtifactId

rabbit\_util

Version

0.0.1-SNAPSHOT

## (4) 在rabbit\_util引入依赖

```

1 <dependencies>
2   <!--rabbitmq消息队列-->
3   <dependency>

```

```

4         <groupId>org.springframework.boot</groupId>
5         <artifactId>spring-boot-starter-actuator</artifactId>
6     </dependency>
7     <dependency>
8         <groupId>org.springframework.cloud</groupId>
9         <artifactId>spring-cloud-starter-bus-amqp</artifactId>
10    </dependency>
11    <dependency>
12        <groupId>com.alibaba</groupId>
13        <artifactId>fastjson</artifactId>
14    </dependency>
15 </dependencies>

```

## (5) 添加service方法

```

1 @Service
2 public class RabbitService {
3     @Autowired
4     private RabbitTemplate rabbitTemplate;
5     /**
6      * 发送消息
7      * @param exchange 交换机
8      * @param routingKey 路由键
9      * @param message 消息
10    */
11    public boolean sendMessage(String exchange, String routingKey, Object message) {
12        rabbitTemplate.convertAndSend(exchange, routingKey, message);
13        return true;
14    }
15 }

```

## (6) 配置mq消息转换器

```

1 @Configuration
2 public class MQConfig {
3     @Bean
4     public MessageConverter messageConverter(){
5         return new Jackson2JsonMessageConverter();
6     }
7 }

```

```
6     }  
7 }
```

说明：默认是字符串转换器

## (7) 添加常量类

```
1 public class MqConst {  
2     /**  
3      * 预约下单  
4      */  
5     public static final String EXCHANGE_DIRECT_ORDER = "exchange.direct.order";  
6     public static final String ROUTING_ORDER = "order";  
7     //队列  
8     public static final String QUEUE_ORDER = "queue.order";  
9  
10    /**  
11     * 短信  
12     */  
13    public static final String EXCHANGE_DIRECT_MSM = "exchange.direct.msm";  
14    public static final String ROUTING_MSM_ITEM = "msm.item";  
15    //队列  
16    public static final String QUEUE_MSM_ITEM = "queue.msm.item";  
17 }
```