作者	部门	日期
付韬	基础业务开发部	2021/4/13

1. 什么是消息中间件

- **消息（Message）**是指在应用间传送的数据。
- **消息队列中间件（Message QueueMiddleware，简称为MQ）**是指利用高效可靠的消息传递机制进行**与平台无关**的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展**进程间的通信**。消息中间件提供了有保证的消息发送，应用程序开发人员无须了解远程过程调用（RPC）和网络通信协议的细节。

2. 消息中间件的作用

- **解耦**：假设现在，日志不光要插入到数据库里，还要在硬盘中增加文件类型的日志，同时，一些关键日志还要通过邮件的方式发送给指定的人。那么，如果不使用MQ，就需要在原来的代码上做扩展，除了B服务，还要加上日志文件的存储和日志邮件的发送。但是，如果你使用了MQ，那么，是不需要做更改的，它还是将消息放到MQ中即可，其它的服务，无论是原来的B服务还是新增的日志文件存储服务或日志邮件发送服务，都直接从MQ中获取消息并处理即可。这就是解耦，它的好处是提高系统灵活性，扩展性。
    - *数据的生产与数据的使用是解耦的。*
- **冗余（存储）**：有些情况下，处理数据的过程会失败。消息中间件可以把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。在把一个消息从消息中间件中删除之前，需要你的处理系统明确地指出该消息已经被处理完成，从而确保你的数据被安全地保存直到你使用完毕。
- **扩展性**: 对于需要与原系统对接的应用，只需要接入MQ即可，原系统不需要做任何改动。
- **削峰（缓冲）**：在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果以能处理这类峰值为标准而投入资源，无疑是巨大的浪费。使用消息中间件能够使关键组件支撑突发访问压力，不会因为突发的超负荷请求而完全崩溃。
    - 如订单先会显示处理中，最终下单结果需要过一段时间才知道。
- **顺序保证**：在大多数使用场景下，数据处理的顺序很重要，大部分消息中间件支持一定程度上的顺序性。
- **异步通信**：在很多时候应用不想也不需要立即处理消息。消息中间件提供了异步处理机制，允许应用把一些消息放入消息中间件中，但并不立即处理它，在之后需要的时候再慢慢处理。
    - 如发短信，邮件。

异步、解耦、消峰是MQ的三大主要应用场景。

- 缺点：
  - 系统复杂性增加。毕竟是增加了一个中间件MQ，那么系统变得更复杂就是不可避免的。
  - 系统可用性降低：MQ若是挂了，容易引起整个服务挂掉。

3. RabbitMQ的起源

RabbitMQ是采用Erlang语言（所以在安装RabbitMQ之前需要安装Erlang环境）实现AMQP（Advanced Message Queuing Protocol，高级消息队列协议）的消息中间件，它最初起源于金融系统，用于在分布式系统中存储转发消息。

在AMQP出现之前，JMS（Java Message Service），JMS通过提供公共JavaAPI的方式实现Java应用程序只需针对JMS API编程，选择合适的MQ驱动即可，JMS会打理好其他部分。ActiveMQ就是JMS的一种实现。但是JMS只适用于Java应用程序之间，并不能扩平台。

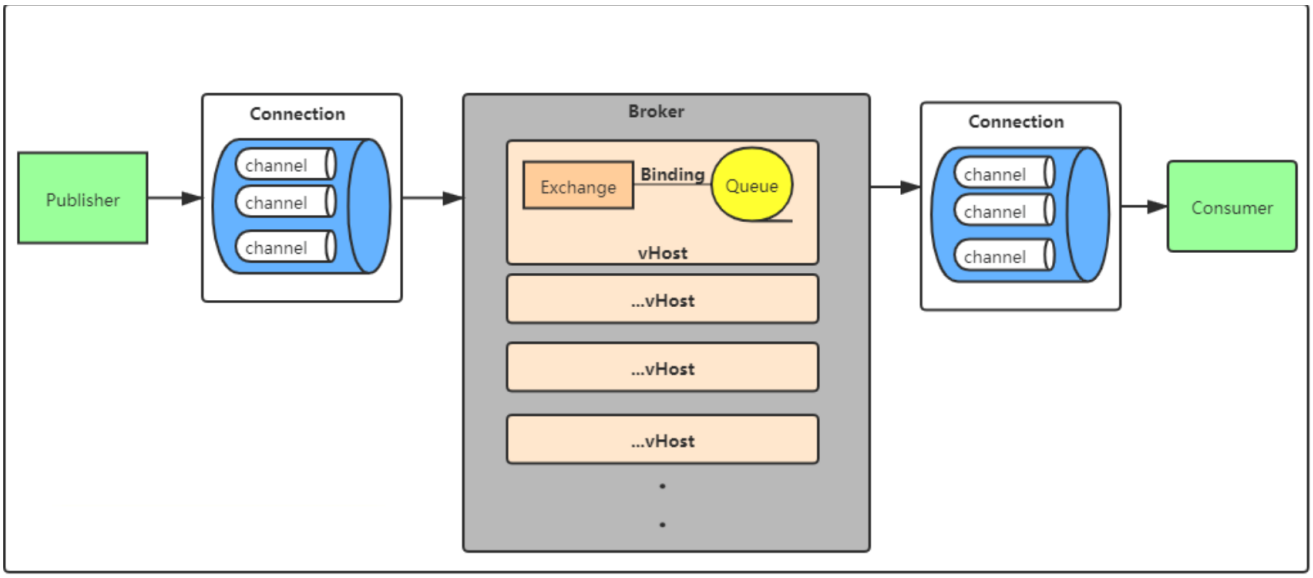
在2006年6月，由Cisco、Redhat、iMatix等联合制定了AMQP的公开标准，由此AMQP登上了历史的舞台。它为面向消息的中间件设计，基于此**协议**的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。从而实现了跨平台。

- RabbitMQ名字来源：取Rabbit这样一个名字，是因为兔子行动非常迅速且繁殖起来非常疯狂，RabbitMQ的开创者认为以此命名这个分布式软件再合适不过了。

4. RabbitMQ核心组件与概念

- **Server/Broker**：RabbitMQ服务器，接收客户端连接，实现AMQP的服务器实体。
- **Connection**：连接，应用程序与Broker的TCP网络连接。
- **Channel**：信道
- 应用程序基于信道Channel与RabbitMQ通信，信道是建立在真实存在的TCP连接之上的，一旦TCP连接打开，应用程序就可以创建Channel信道。
- 为什么需要信道:
  - 因为对于系统来说，TCP连接的创建和销毁需要昂贵的开销
    - 且OS能提供的TCP连接本身就是有限的
- **Message**：消息。服务器和应用程序之间传递的数据，本质上就是一段数据，由Properties和Body组成。Body内为实际要传递的消息。
- **Exchange**：交换机。接收消息，根据路由键转发消息到绑定的队列。
- 交换机类型：
  - **fanout**: 忽略路由键，将消息分发到与Exchange绑定的所有Queue
  - **direct**: 完全匹配路由键
  - **topic**: 路由键通配符(我们将被英文句点号“`.`”分隔开的每一段独立的字符串称为一个单词)
  - `*` 匹配一个单词
  - `#` 匹配零个或多个单词
- **Binding**：Exchange和Queue之间的绑定关系，binding中可以指定routing key。
- **Routing key**：路由键，Exchange可根据这个值将消息路由到不同的Queue。

- **Queue**：也称为Message Queue，消息队列，保存消息并将它们发送给消费者。（Q: 拉，推?）
- **Virtual Host**：相当于一个独立的RabbitMQ，一个Virtual Host可以有若干个Exchange和Queue，可以用来隔离Exchange和Queue。每个Virtual Host中的数据完全隔离。同一个Virtual Host里面不能有相同名称的Exchange和Queue。权限控制的最小粒度是Virtual Host。默认是 `/` 。
- **Producer**：生产者：生产消息，将消息推送到Exchange。生产者永远只与Exchange交互。
- **Consumer**：消费者：消费消息，从Queue中get消息或者监听Queue推动过来的消息。永远只与Queue交互。



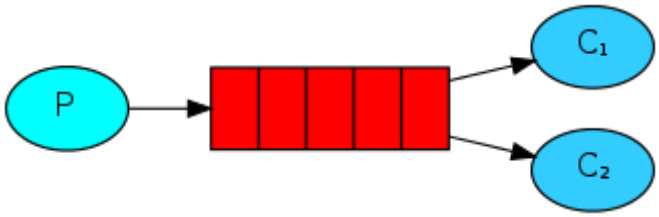
- 运转流程:
  - 生产者:
    1. 生产者连接到RabbitMQ Broker，并指定VH，建立一个连接（Connection），开启一个信道（Channel）。
    2. 生产者声明一个交换器，并设置相关属性，比如交换机类型、是否持久化等。
    3. 生产者声明一个队列并设置相关属性，比如是否排他、是否持久化、是否自动删除等。
    4. 生产者通过路由键将交换器和队列绑定起来。
    5. 生产者发送消息至RabbitMQ Broker，其中包含路由键、交换器等信息。
    6. 相应的交换器根据接收到的路由键查找相匹配的队列。
    7. 如果找到，则将从生产者发送过来的消息存入相应的队列中。
    8. 如果没有找到，则根据生产者配置的属性选择丢弃还是回退给生产者。
    9. 关闭信道。
    10. 关闭连接。
  - 消费者:
    1. 消费者连接到RabbitMQ Broker，并指定VH，建立一个连接（Connection），开启一个信道（Channel）。
    2. 消费者向RabbitMQ Broker请求消费相应队列中的消息，可能会设置相应的回调函数，以及做一些准备工作。
    3. 等待RabbitMQ Broker回应并投递相应队列中的消息，消费者接收消息。
    4. 消费者确认（ack）接收到的消息，通知RabbitMQ Broker。

5. RabbitMQ从队列中删除相应已经被确认的消息。
6. 关闭信道。
7. 关闭连接。

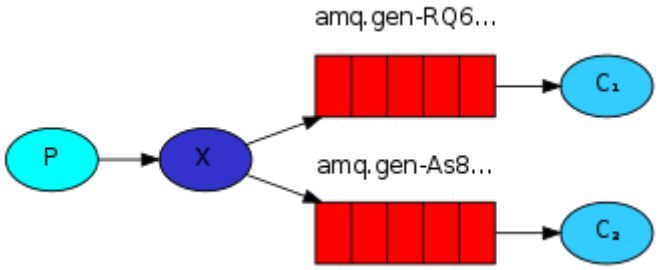
5. 官网使用示例

1. **Hello World P2P**
   - 这种情况下看似是生产者直接连接的Queue，但实际情况并非如此。

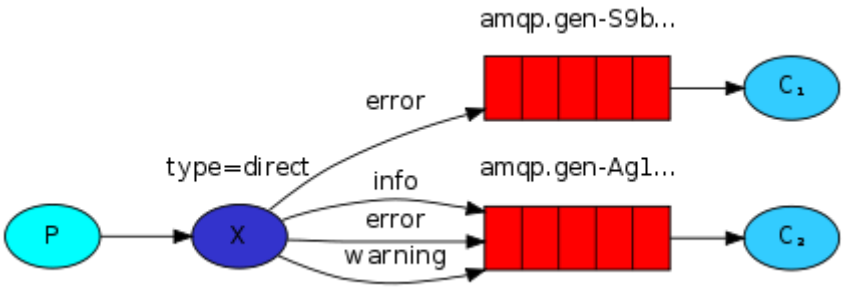
2. **Work Queues**



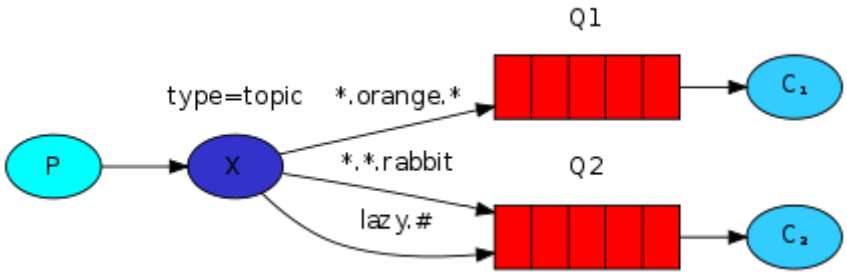
3. **Publish/Subscribe**
   - exchange type = fanout



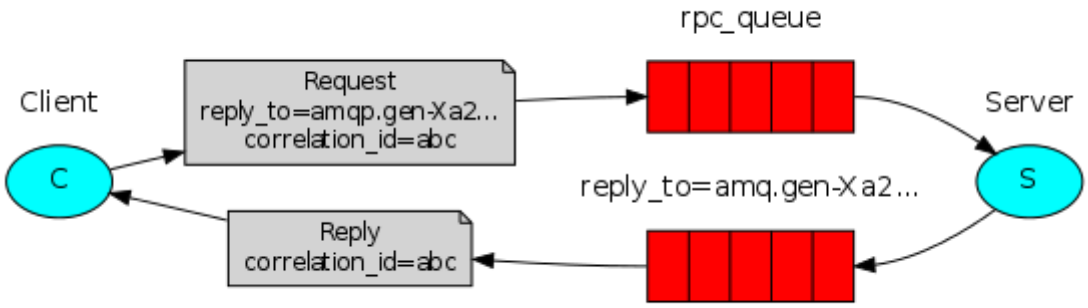
4. **Routing**
   - exchange type = direct



5. **Topics**
   - exchange type = topic

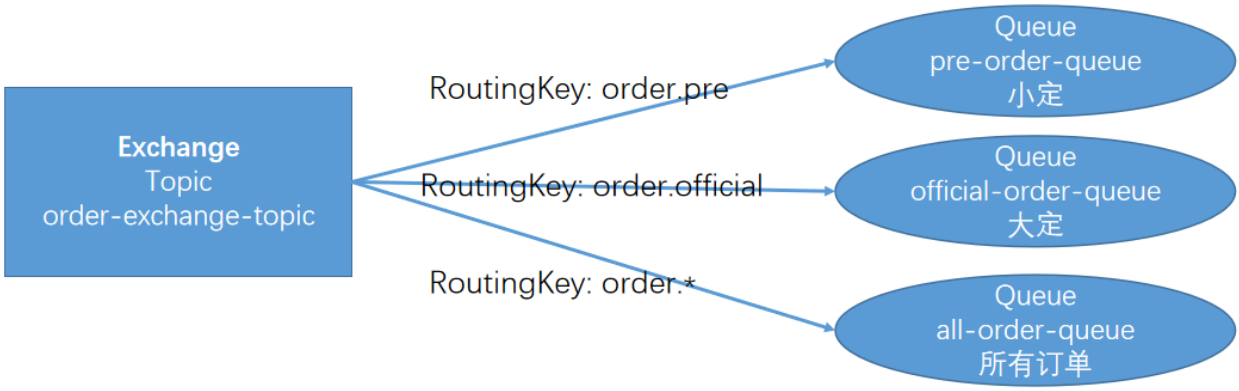


6. **RPC**
   - reply_to
   - correlation_id



6. 使用示例-不在SpringBoot环境下

> 目标：订单（大定，小定）。存在三个消费者分别监听小定、大定、和所有订单。



- 定义 `topic` 类型的订单交换机，名称为 `order-exchange-topic`
- 定义三个队列
  - 预订单(小定) `pre-order-queue` ，路由键为 `order.pre`
  - 正式订单(大定) `official-order-queue` 路由键为 `order.official`
  - 所有订单 `all-order-queue` 路由键为 `order.*`

Code

- 定义

```java
@Slf4j
public class OrderDemo {
    public static void main(String[] args) throws NoSuchAlgorithmException, KeyManagementException, URISyntaxException, IOException, TimeoutException {
        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setUri("amqp://futao:123456789@localhost:5672");
        connectionFactory.setVirtualHost("/tech-sharing");
        // 创建TCP连接
        Connection connection = connectionFactory.newConnection();
        // 创建通道
        Channel channel = connection.createChannel();
        // 定义交换机
        channel.exchangeDeclare( exchange: "order-exchange-topic", BuiltinExchangeType.TOPIC, durable: true, autoDelete: false, internal: false, arguments: null);

        // 定义队列-小定
        channel.queueDeclare( queue: "pre-order-queue", durable: true, exclusive: false, autoDelete: false, arguments: null);
        // 定义队列-大定
        channel.queueDeclare( queue: "official-order-queue", durable: true, exclusive: false, autoDelete: false, arguments: null);
        // 定义队列-所有定单
        channel.queueDeclare( queue: "all-order-queue", durable: true, exclusive: false, autoDelete: false, arguments: null);

        // 绑定
        channel.queueBind( queue: "pre-order-queue", exchange: "order-exchange-topic", routingKey: "order.pre");
        channel.queueBind( queue: "official-order-queue", exchange: "order-exchange-topic", routingKey: "order.official");
        channel.queueBind( queue: "all-order-queue", exchange: "order-exchange-topic", routingKey: "order.*");
```

- 生产

```java
// 生产者发送消息
new Thread(() -> {
    // 小定
    for (int i = 0; i < 10; i++) {
        try {
            String msg = "preOrder-" + i;
            channel.basicPublish( exchange: "order-exchange-topic", routingKey: "order.pre", new AMQP.BasicProperties(), msg.getBytes(StandardCharsets.UTF_8));
            log.info("消息[{}]投递成功", msg);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    // 大定
    for (int i = 0; i < 10; i++) {
        try {
            String msg = "officialOrder-" + i;
            channel.basicPublish( exchange: "order-exchange-topic", routingKey: "order.official", new AMQP.BasicProperties(), msg.getBytes(StandardCharsets.UTF_8));
            log.info("消息[{}]投递成功", msg);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}, name: "producer").start();
```

- 消费
    - 拉模式

```java
/**
 * 消费者通过主动get的方式消费
 *
 * @param connection
 */
private static void byGet(Connection connection) {
    // 消费消息-小定
    new Thread(() -> {
        try {
            // 一条消息
            GetResponse response = connection.createChannel().basicGet( queue: "pre-order-queue",  autoAck: true);
            log.info("get到小定:[{}]", new String(response.getBody(), StandardCharsets.UTF_8));
        } catch (IOException e) {
            e.printStackTrace();
        }
    },  name: "pre-consumer").start();

    // 消费消息-大定
    new Thread(() -> {
        // 一条消息
        try {
            GetResponse response = connection.createChannel().basicGet( queue: "official-order-queue",  autoAck: true);
            log.info("get到大定:[{}]", new String(response.getBody(), StandardCharsets.UTF_8));
        } catch (IOException e) {
            e.printStackTrace();
        }
    },  name: "official-consumer").start();

    // 消费消息-所有订单
    new Thread(() -> {
        // 一条消息
        try {
            GetResponse response = connection.createChannel().basicGet( queue: "all-order-queue",  autoAck: true);
            log.info("get到订单:[{}]", new String(response.getBody(), StandardCharsets.UTF_8));
        } catch (IOException e) {
            e.printStackTrace();
        }
    },  name: "all-consumer").start();
}
```

- 推模式

```java
/**
 * 推模式消费消息
 *
 * @param connection
 */
private static void byConsumer(Connection connection) {
    // 小定
    new Thread(() -> {
        try {
            // 创建消费者通道
            Channel channel = connection.createChannel();
            // 检查队列是否存在
            channel.queueDeclarePassive("pre-order-queue");
            // 每次拉取的消息数量
            channel.basicQos( prefetchCount: 1);
            channel.basicConsume( queue: "pre-order-queue", autoAck: true, new DefaultConsumer(channel) {
                @Override
                public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                    log.info("接收到小定:[{}]", new String(body, StandardCharsets.UTF_8));
                }
            });
        } catch (IOException e) {
            e.printStackTrace();
        }
    }, name: "pre-consumer").start();
    // 小定
    new Thread(() -> {
        try {
            // 创建消费者通道
            Channel channel = connection.createChannel();
            // 每次拉取的消息数量
            channel.basicQos( prefetchCount: 1);
            channel.basicConsume( queue: "official-order-queue", autoAck: true, new DefaultConsumer(channel) {
                @Override
                public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                    log.info("接收到大定:[{}]", new String(body, StandardCharsets.UTF_8));
                }
            });
        } catch (IOException e) {
            e.printStackTrace();
        }
    }, name: "pre-consumer").start();

    // 所有订单
```

7. 使用示例-SpringBootStart环境下

1. 导入RabbitMQ的starter依赖
    - 导入的是amqp的依赖，但是SpringBoot默认使用的实现就是RabbitMQ

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2. 添加RabbitMQ配置

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=futao
spring.rabbitmq.password=123456789
spring.rabbitmq.virtual-host=/tech-sharing
```

3. 定义交换机

```
    /**
     * 定义持久化topic交换机
     *
     * @return
     */
    @Bean
    public Exchange orderExchangeTopic() {
        return ExchangeBuilder
                .topicExchange("order-exchange-topic")
                .durable(true)
                .build();
    }
```

4. 定义队列

```
    /**
     * 定义预订订单持久化队列
     *
     * @return
     */
    @Bean
    public Queue preOrderQueue() {
        return QueueBuilder
                .durable("pre-order-queue")
                .build();
    }

    /**
     * 定义正式订单持久化队列
     *
     * @return
     */
    @Bean
    public Queue officialOrderQueue() {
        return QueueBuilder
                .durable("official-order-queue")
                .build();
    }

    /**
     * 定义所有订单持久化队列
     *
     * @return
     */
    @Bean
    public Queue allOrderQueue() {
        return QueueBuilder
                .durable("all-order-queue")
                .build();
```

```
    }
```

5. 定义绑定关系

```
    /**
     * preOrder队列与交换机进行绑定
     *
     * @param preOrderQueue
     * @param orderExchangeFanout
     * @return
     */
    @Bean
    public Binding preOrderBinding(Queue preOrderQueue, Exchange
orderExchangeFanout) {
        return BindingBuilder
                .bind(preOrderQueue)
                .to(orderExchangeFanout)
                // 路由键
                .with("order.pre")
                .noargs();
    }

    /**
     * official队列与交换机进行绑定
     *
     * @param officialOrderQueue
     * @param orderExchangeFanout
     * @return
     */
    @Bean
    public Binding officialOrderBinding(Queue officialOrderQueue, Exchange
orderExchangeFanout) {
        return BindingBuilder
                .bind(officialOrderQueue)
                .to(orderExchangeFanout)
                // 路由键
                .with("order.official")
                .noargs();
    }

    /**
     * 所有订单队列与交换机进行绑定
     *
     * @param allOrderQueue
     * @param orderExchangeFanout
     * @return
     */
    @Bean
```

```
    public Binding allOrderBinding(Queue allOrderQueue, Exchange
orderExchangeFanout) {
        return BindingBuilder
                .bind(allOrderQueue)
                .to(orderExchangeFanout)
                // 路由键
                .with("order.*")
                .noargs();
    }
```

6. 生产消息

```
/**
 * 订单生产者
 *
 * @author futao <1185172056@qq.com> <https://github.com/FutaoSmile>
 * @date 2021/2/9
 */
@Slf4j
@Component
public class OrderProducer {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private Exchange orderExchangeTopic;

    @PostConstruct
    public void send() {
        log.info("开始推送订单消息");
        int msgCount = 1_00;
        // 小定
        IntStream.rangeClosed(1, msgCount)
                .forEach(i -> {
                    Order order = new Order(i, new BigDecimal(i),
OrderStatusEnum.UN_PAY.getStatus());

 rabbitTemplate.convertAndSend(orderExchangeTopic.getName(), "order.pre",
order);
                });

        // 大定
        IntStream.rangeClosed(1, msgCount)
                .forEach(i -> {
                    Order order = new Order(i, new BigDecimal(i),
OrderStatusEnum.UN_PAY.getStatus());
```

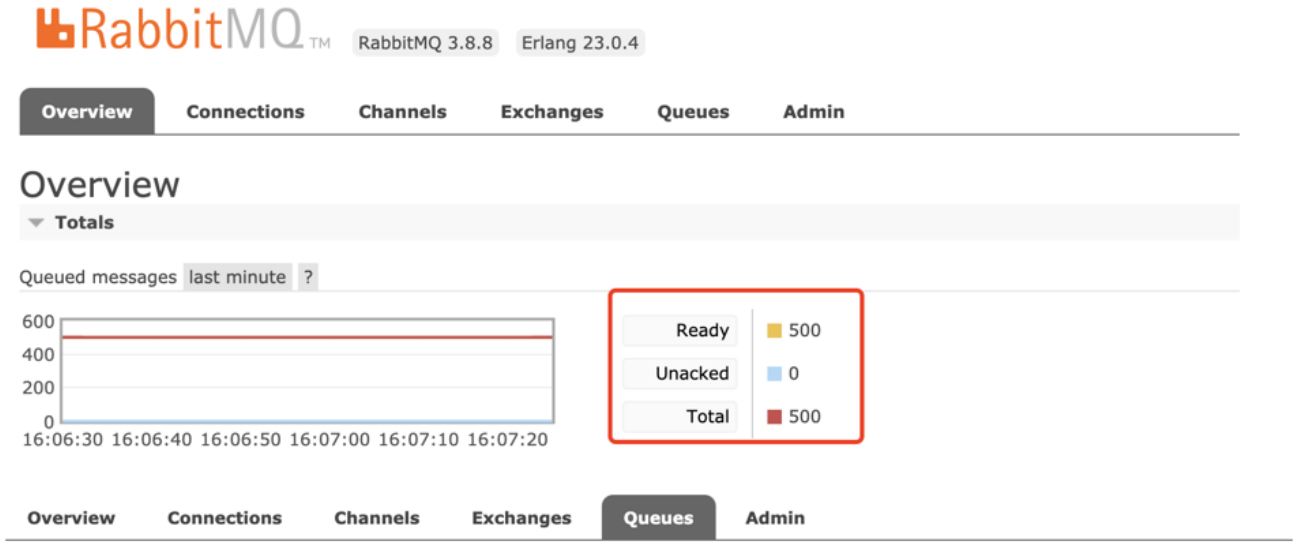
```
 rabbitTemplate.convertAndSend(orderExchangeTopic.getName(),
"order.official", order);
                });

        // 路由键是xxx
        IntStream.rangeClosed(1, msgCount)
                .forEach(i -> {
                    Order order = new Order(i, new BigDecimal(i),
OrderStatusEnum.UN_PAY.getStatus());

 rabbitTemplate.convertAndSend(orderExchangeTopic.getName(), "order.xxx",
order);
                });
        log.info("订单消息推送完成");
    }
}
```

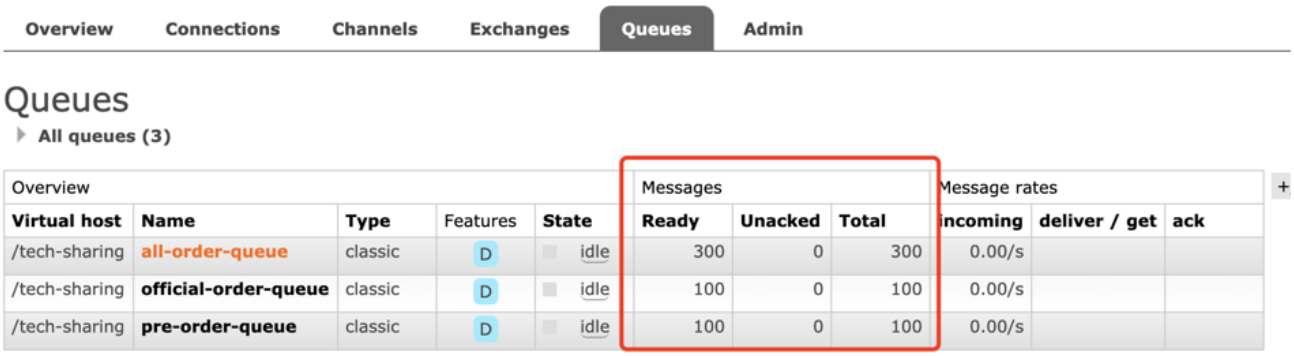
7. 消费消息

```
/**
 * 订单消费着
 *
 * @author futao <1185172056@qq.com> <https://github.com/FutaoSmile>
 * @date 2021/2/9
 */
@Slf4j
@Component
public class OrderConsumer {

    /**
     * 小定消费者
     * 手动签收，并发为1
     *
     * @param message
     */
    @RabbitHandler
    @RabbitListener(queues = "pre-order-queue", ackMode = "AUTO")
    public void preOrderConsumer(Message message, Order order) {
        log.info("接收到小定:{}", JSON.toJSONString(order, true));
    }

    /**
     * 大定消费者
     * 手动签收，并发为4
     *
     * @param message
     */
```

```
    @RabbitHandler
    @RabbitListener(queues = "official-order-queue", ackMode = "MANUAL",
concurrency = "4")
    public void officialOrderConsumer(Message message, Order order, Channel
channel) throws IOException {
        log.info("接收到大定:{}", JSON.toJSONString(order, true));
        // 进行手动签收
        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        log.info("deliveryTag:{}", deliveryTag);
        // ack
        channel.basicAck(deliveryTag, false);
        // nack
        //channel.basicNack(deliveryTag, false, true);
    }

    /**
     * 所有订单消费者
     * 自动，并发为4
     *
     * @param message
     */
    @RabbitHandler
    @RabbitListener(queues = "all-order-queue", ackMode = "AUTO",
concurrency = "4")
    public void allOrderConsumer(Message message, Order order) {
        log.info("接收到订单:{}", JSON.toJSONString(order, true));
    }
}
```

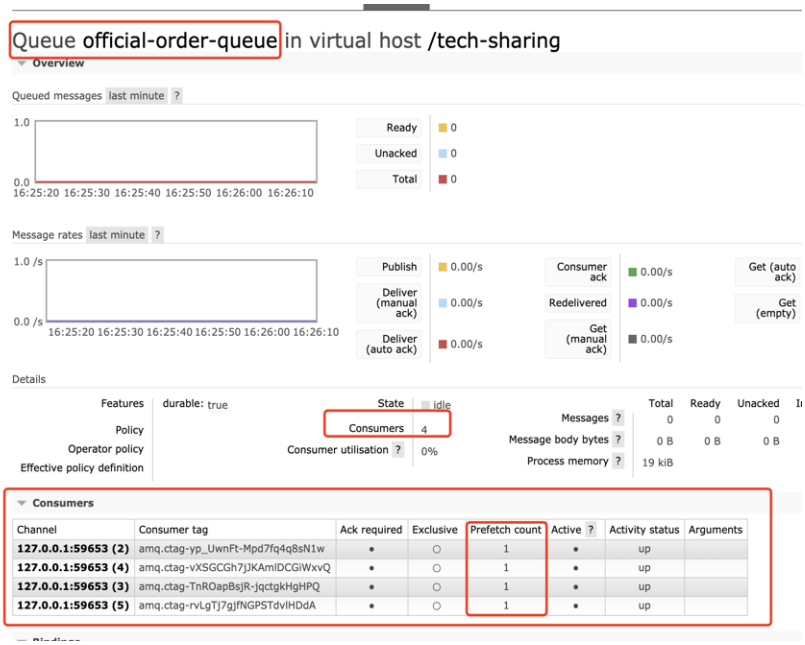
8. 插件-Web端管理后台





开始消费

- 签收模式(手动/自动)
  - 手动拒签是否再次入队requeue
  - 是否批量签收
- 消费者并发量
- 消费者保护
  - 限流



9. 其他

- 代码仓库:
  - https://gitee.com/FutaoSmile/tech-sharing-mq
  - https://github.com/FutaoSmile/tech-sharing-mq