

25 | 异步处理好用，但非常容易用错

2020-05-12 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



讲述：王少泽


时长 20:19 大小 18.62M



你好，我是朱晔。今天，我来和你聊聊好用但容易出错的异步处理。

异步处理是互联网应用不可或缺的一种架构模式，大多数业务项目都是由同步处理、异步处理和定时任务处理三种模式相辅相成实现的。

区别于同步处理，异步处理无需同步等待流程处理完毕，因此适用场景主要包括：

服务于主流程的分支流程。比如，在注册流程中，把数据写入数据库的操作是主流，但注册后给用户发优惠券或欢迎短信的操作是分支流程，时效性不强，可以进行异步处理。


用户不需要实时看到结果的流程。比如，下单后的配货、送货流程完全可以进行异步处理，每个阶段处理完成后，再给用户发推送或短信让用户知晓即可。

同时，异步处理因为可以有 MQ 中间件的介入用于任务的缓冲的分发，所以相比于同步处理，在应对流量洪峰、实现模块解耦和消息广播方面有功能优势。

不过，异步处理虽然好用，但在实现的时候却有三个最容易犯的错，分别是异步处理流程的可靠性问题、消息发送模式的区分问题，以及大量死信消息堵塞队列的问题。今天，我就用三个代码案例结合目前常用的 MQ 系统 RabbitMQ，来和你具体聊聊。

今天这一讲的演示，我都会使用 Spring AMQP 来操作 RabbitMQ，所以你需要先引入 amqp 依赖：

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-amqp</artifactId>
4 </dependency>
```

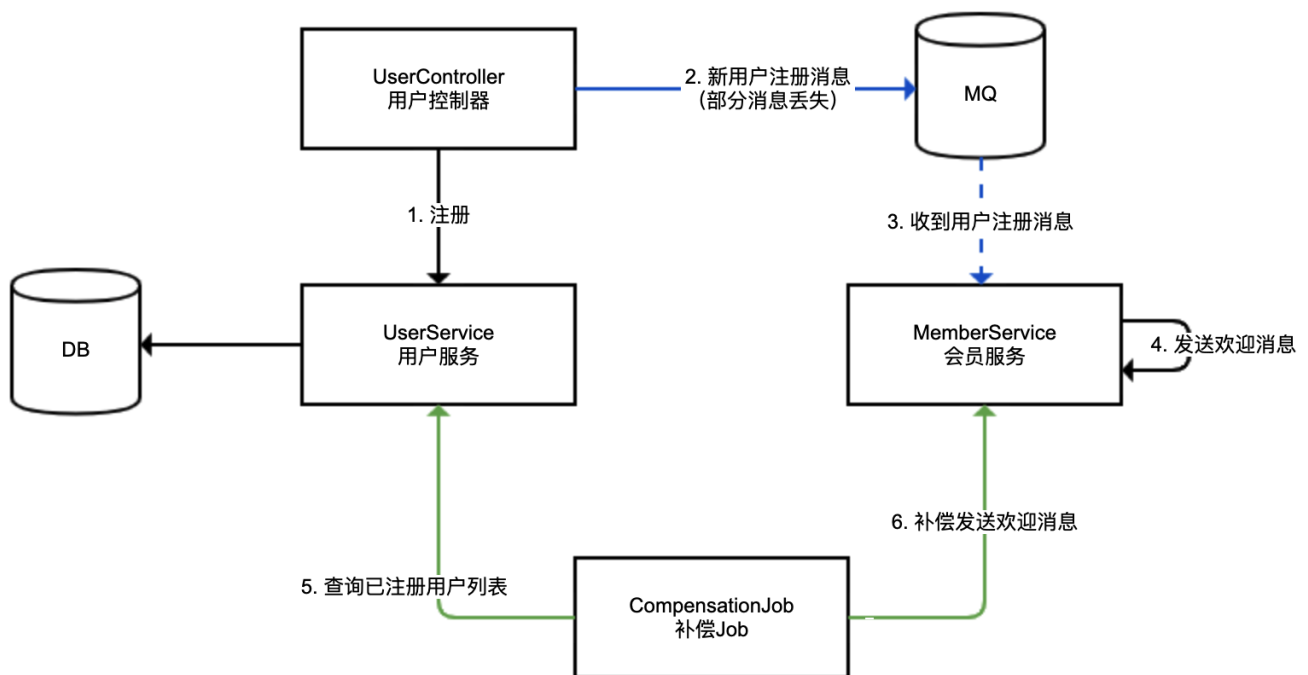
 复制代码

异步处理需要消息补偿闭环

使用类似 RabbitMQ、RocketMQ 等 MQ 系统来做消息队列实现异步处理，虽然说消息可以落地到磁盘保存，即使 MQ 出现问题消息数据也不会丢失，但是异步流程在消息发送、传输、处理等环节，都可能发生消息丢失。此外，任何 MQ 中间件都无法确保 100% 可用，需要考虑不可用时异步流程如何进行。

因此，**对于异步处理流程，必须考虑补偿或者说建立主备双活流程。**

我们来看一个用户注册后异步发送欢迎消息的场景。用户注册落数据库的流程为同步流程，会员服务收到消息后发送欢迎消息的流程为异步流程。



我们分析一下：

蓝色的线，使用 MQ 进行的异步处理，我们称作主线，可能存在消息丢失的情况（虚线代表异步调用）；

绿色的线，使用补偿 Job 定期进行消息补偿，我们称作备线，用来补偿主线丢失的消息；

考虑到极端的 MQ 中间件失效的情况，我们要求备线的处理吞吐能力达到主线的能力水平。

我们来看一下相关的实现代码。

首先，定义 UserController 用于注册 + 发送异步消息。对于注册方法，我们一次性注册 10 个用户，用户注册消息不能发送出去的概率为 50%。

复制代码

```
1 @RestController
2 @Slf4j
3 @RequestMapping("user")
4 public class UserController {
5     @Autowired
6     private UserService userService;
7     @Autowired
8     private RabbitTemplate rabbitTemplate;
9 }
```

```

10     @GetMapping("register")
11     public void register() {
12         //模拟10个用户注册
13         IntStream.rangeClosed(1, 10).forEach(i -> {
14             //落库
15             User user = userService.register();
16             //模拟50%的消息可能发送失败
17             if (ThreadLocalRandom.current().nextInt(10) % 2 == 0) {
18                 //通过RabbitMQ发送消息
19                 rabbitTemplate.convertAndSend(RabbitConfiguration.EXCHANGE, Rabi
20                     log.info("sent mq user {}", user.getId());
21             }
22         });
23     }
24 }

```

然后，定义 MemberService 类用于模拟会员服务。会员服务监听用户注册成功的消息，并发送欢迎短信。我们使用 ConcurrentHashMap 来存放那些发过短信的用户 ID 实现幂等，避免相同的用户进行补偿时重复发送短信：

 复制代码

```

1  @Component
2  @Slf4j
3  public class MemberService {
4      //发送欢迎消息的状态
5      private Map<Long, Boolean> welcomeStatus = new ConcurrentHashMap<>();
6      //监听用户注册成功的消息，发送欢迎消息
7      @RabbitListener(queues = RabbitConfiguration.QUEUE)
8      public void listen(User user) {
9          log.info("receive mq user {}", user.getId());
10         welcome(user);
11     }
12     //发送欢迎消息
13     public void welcome(User user) {
14         //去重操作
15         if (welcomeStatus.putIfAbsent(user.getId(), true) == null) {
16             try {
17                 TimeUnit.SECONDS.sleep(2);
18             } catch (InterruptedException e) {
19             }
20             log.info("memberService: welcome new user {}", user.getId());
21         }
22     }
23 }

```

对于 MQ 消费程序，处理逻辑务必考虑去重（支持幂等），原因有几个：


MQ 消息可能会因为中间件本身配置错误、稳定性等原因出现重复。

自动补偿重复，比如本例，同一条消息可能既走 MQ 也走补偿，肯定会出现重复，而且考虑到高内聚，补偿 Job 本身不会做去重处理。

人工补偿重复。出现消息堆积时，异步处理流程必然会延迟。如果我们提供了通过后台进行补偿的功能，那么在处理遇到延迟的时候，很可能会先进行人工补偿，过了一段时间后处理程序又收到消息了，重复处理。我之前就遇到过一次由 MQ 故障引发的事故，MQ 中堆积了几十万条发放资金的消息，导致业务无法及时处理，运营以为程序出错了就先通过后台进行了人工处理，结果 MQ 系统恢复后消息又被重复处理了一次，造成大量资金重复发放。

接下来，定义补偿 Job 也就是备线操作。

我们在 CompensationJob 中定义一个 @Scheduled 定时任务，5 秒做一次补偿操作，因为 Job 并不知道哪些用户注册的消息可能丢失，所以是全量补偿，补偿逻辑是：每 5 秒补偿一次，按顺序一次补偿 5 个用户，下一次补偿操作从上一次补偿的最后一个用户 ID 开始；对于补偿任务我们提交到线程池进行“异步”处理，提高处理能力。

 复制代码

```
1 @Component
2 @Slf4j
3 public class CompensationJob {
4     //补偿Job异步处理线程池
5     private static ThreadPoolExecutor compensationThreadPool = new ThreadPoolExecutor
6         (10, 10,
7         1, TimeUnit.HOURS,
8         new ArrayBlockingQueue<>(1000),
9         new ThreadFactoryBuilder().setNameFormat("compensation-threadpool-"),
10     @Autowired
11     private UserService userService;
12     @Autowired
13     private MemberService memberService;
14     //目前补偿到哪个用户ID
15     private long offset = 0;
16
17     //10秒后开始补偿，5秒补偿一次
18     @Scheduled(initialDelay = 10_000, fixedRate = 5_000)
19     public void compensationJob() {
20         log.info("开始从用户ID {} 补偿", offset);
21         //获取从offset开始的用户
22         userService getUsersAfterIdWithLimit(offset, 5).forEach(user -> {
23             compensationThreadPool.execute(() -> memberService.welcome(user));
24             offset = user.getId();
25         });
26     }
27 }
```

```
25         });
26     }
27 }
```

为了实现高内聚，主线和备线处理消息，最好使用同一个方法。比如，本例中 MemberService 监听到 MQ 消息和 CompensationJob 补偿，调用的都是 welcome 方法。

此外值得一说的是，Demo 中的补偿逻辑比较简单，生产级的代码应该在以下几个方面进行加强：

考虑配置补偿的频次、每次处理数量，以及补偿线程池大小等参数为合适的值，以满足补偿的吞吐量。

考虑备线补偿数据进行适当延迟。比如，对注册时间在 30 秒之前的用户再进行补偿，以方便和主线 MQ 实时流程错开，避免冲突。

诸如当前补偿到哪个用户的 offset 数据，需要落地数据库。

补偿 Job 本身需要高可用，可以使用类似 XXLJob 或 ElasticJob 等任务系统。

运行程序，执行注册方法注册 10 个用户，输出如下：

 复制代码

```
1 [17:01:16.570] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.a.compensation.UserCon
2 [17:01:16.571] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.a.compensation.UserCon
3 [17:01:16.572] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.a.compensation.UserCon
4 [17:01:16.573] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.a.compensation.UserCon
5 [17:01:16.594] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
6 [17:01:18.597] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
7 [17:01:18.601] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
8 [17:01:20.603] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
9 [17:01:20.604] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
10 [17:01:22.605] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
11 [17:01:22.606] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
12 [17:01:24.611] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
13 [17:01:25.498] [scheduling-1] [INFO ] [o.g.t.c.a.compensation.CompensationJob::
14 [17:01:27.510] [compensation-threadpool-1] [INFO ] [o.g.t.c.a.compensation.Meml
15 [17:01:27.510] [compensation-threadpool-3] [INFO ] [o.g.t.c.a.compensation.Meml
16 [17:01:27.511] [compensation-threadpool-2] [INFO ] [o.g.t.c.a.compensation.Meml
17 [17:01:30.496] [scheduling-1] [INFO ] [o.g.t.c.a.compensation.CompensationJob::
18 [17:01:32.500] [compensation-threadpool-6] [INFO ] [o.g.t.c.a.compensation.Meml
19 [17:01:32.500] [compensation-threadpool-9] [INFO ] [o.g.t.c.a.compensation.Meml
```

```
20 [17:01:35.496] [scheduling-1] [INFO ] [o.g.t.c.a.compensation.CompensationJob:]
21 [17:01:37.501] [compensation-threadpool-0] [INFO ] [o.g.t.c.a.compensation.Meml
22 - - - - -
```

可以看到：

总共 10 个用户，MQ 发送成功的用户有四个，分别是用户 1、5、7、8。

补偿任务第一次运行，补偿了用户 2、3、4，第二次运行补偿了用户 6、9，第三次运行补充了用户 10。

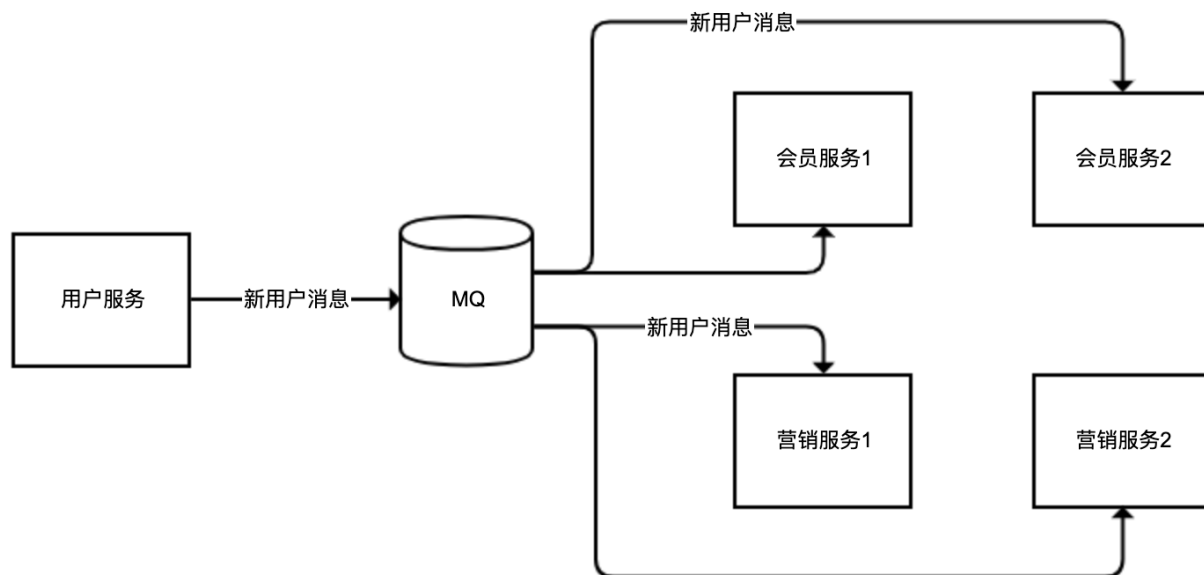
最后提一下，针对消息的补偿闭环处理的最高标准是，能够达到补偿全量数据的吞吐量。也就是说，如果补偿备线足够完善，即使直接把 MQ 停机，虽然会略微影响处理的及时性，但至少确保流程都能正常执行。

注意消息模式是广播还是工作队列

在今天这一讲的一开始，我们提到异步处理的一个重要优势，是实现消息广播。

消息广播，和我们平时说的“广播”意思差不多，就是希望同一条消息，不同消费者都能分别消费；而队列模式，就是不同消费者共享消费同一个队列的数据，相同消息只能被某一个消费者消费一次。

比如，同一个用户的注册消息，会员服务需要监听以发送欢迎短信，营销服务同样需要监听以发送新用户小礼物。但是，会员服务、营销服务都可能多个实例，我们期望的是同一个用户的消息，可以同时广播给不同的服务（广播模式），但对于同一个服务的不同实例（比如会员服务 1 和会员服务 2），不管哪个实例来处理，处理一次即可（工作队列模式）：



在实现代码的时候，我们务必确认 MQ 系统的机制，确保消息的路由按照我们的期望。

对于类似 RocketMQ 这样的 MQ 来说，实现类似功能比较简单直白：如果消费者属于一个组，那么消息只会由同一个组的一个消费者来消费；如果消费者属于不同组，那么每个组都能消费一遍消息。

而对于 RabbitMQ 来说，消息路由的模式采用的是队列 + 交换器，队列是消息的载体，交换器决定了消息路由到队列的方式，配置比较复杂，容易出错。所以，接下来我重点和你讲讲 RabbitMQ 的相关代码实现。

我们还是以上面的架构图为例，来演示使用 RabbitMQ 实现广播模式和工作队列模式的坑。

第一步，实现会员服务监听用户服务发出的新用户注册消息的那部分逻辑。

如果我们启动两个会员服务，那么同一个用户的注册消息应该只能被其中一个实例消费。

我们分别实现 RabbitMQ 队列、交换器、绑定三件套。其中，队列用的是匿名队列，交换器用的是直接交换器 `DirectExchange`，交换器绑定到匿名队列的路由 Key 是空字符串。在收到消息之后，我们会打印所在实例使用的端口：


```

1 //为了代码简洁直观，我们把消息发布者、消费者、以及MQ的配置代码都放在了一起
2 @Slf4j
3 @Configuration
4 @RestController
5 @RequestMapping("workqueuewrong")
6 public class WorkQueueWrong {
7
8     private static final String EXCHANGE = "newuserExchange";
9     @Autowired
10    private RabbitTemplate rabbitTemplate;
11
12    @GetMapping
13    public void sendMessage() {
14        rabbitTemplate.convertAndSend(EXCHANGE, "", UUID.randomUUID().toString
15    }
16
17    //使用匿名队列作为消息队列
18    @Bean
19    public Queue queue() {
20        return new AnonymousQueue();
21    }
22
23    //声明DirectExchange交换器，绑定队列到交换器
24    @Bean
25    public Declarables declarables() {
26        DirectExchange exchange = new DirectExchange(EXCHANGE);
27        return new Declarables(queue(), exchange,
28            BindingBuilder.bind(queue()).to(exchange).with(""));
29    }
30
31    //监听队列，队列名称直接通过SpEL表达式引用Bean
32    @RabbitListener(queues = "#{queue.name}")
33    public void memberService(String userName) {
34        log.info("memberService: welcome message sent to new user {} from {}",
35    }
36
37 }

```

使用 12345 和 45678 两个端口启动两个程序实例后，调用 sendMessage 接口发送一条消息，输出的日志，显示**同一个会员服务两个实例都收到了消息**：

```

[18:05:09.600] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#0-1] [INFO ] [o.g.t.c.a.fanoutvswork.WorkQueueWrong:45 ] -
memberService: welcome message sent to new user 925d7d88-3b53-4524-9309-274f137eae5 from 12345

```

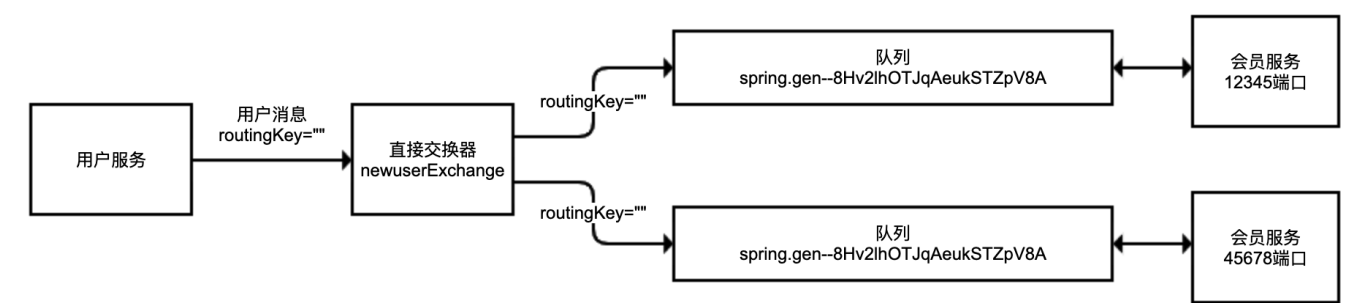
```

[18:05:09.585] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#0-1] [INFO ] [o.g.t.c.a.fanoutvswork.WorkQueueWrong:45 ] -
memberService: welcome message sent to new user 925d7d88-3b53-4524-9309-274f137eae5 from 45678

```

出现这个问题的原因是，我们没有理清楚 RabbitMQ 直接交换器和队列的绑定关系。

如下图所示，RabbitMQ 的直接交换器根据 routingKey 对消息进行路由。由于我们的程序每次启动都会创建匿名（随机命名）的队列，所以相当于每一个会员服务实例都对应独立的队列，以空 routingKey 绑定到直接交换器。用户服务发出消息的时候也设置了 routingKey 为空，所以直接交换器收到消息之后，发现有多条队列匹配，于是都转发了消息：

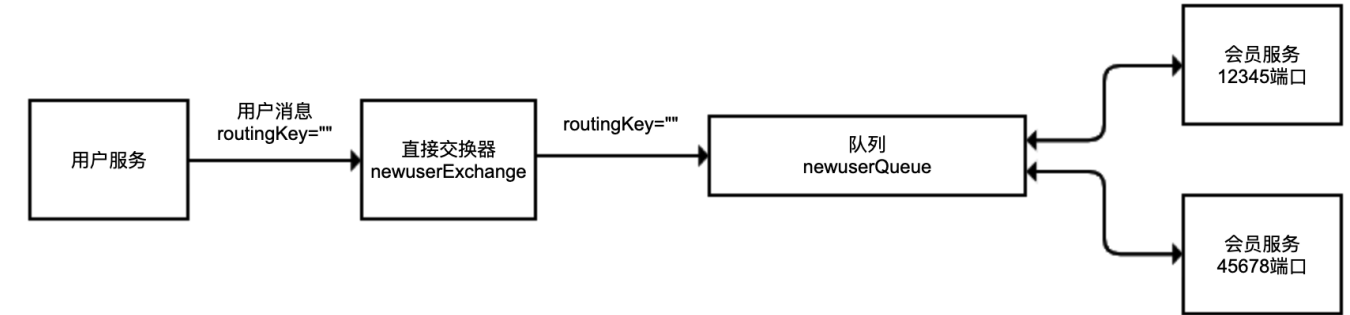


要修复这个问题其实很简单，对于会员服务不要使用匿名队列，而是使用同一个队列即可。把上面代码中的匿名队列替换为一个普通队列：

复制代码

```
1 private static final String QUEUE = "newuserQueue";
2 @Bean
3 public Queue queue() {
4     return new Queue(QUEUE);
5 }
```


测试发现，对于同一条消息来说，两个实例中只有一个实例可以收到，不同的消息按照轮询分发给不同的实例。现在，交换器和队列的关系是这样的：



第二步，进一步完整实现用户服务需要广播消息给会员服务和营销服务的逻辑。

我们希望会员服务和营销服务都可以收到广播消息，但会员服务或营销服务中的每个实例只需要收到一次消息。

代码如下，我们声明了一个队列和一个广播交换器 FanoutExchange，然后模拟两个用户服务和两个营销服务：

 复制代码

```
1  @Slf4j
2  @Configuration
3  @RestController
4  @RequestMapping("fanoutwrong")
5  public class FanoutQueueWrong {
6      private static final String QUEUE = "newuser";
7      private static final String EXCHANGE = "newuser";
8      @Autowired
9      private RabbitTemplate rabbitTemplate;
10
11      @GetMapping
12      public void sendMessage() {
13          rabbitTemplate.convertAndSend(EXCHANGE, "", UUID.randomUUID().toString)
14      }
15      //声明FanoutExchange，然后绑定到队列，FanoutExchange绑定队列的时候不需要routingKey
16      @Bean
17      public Declarables declarables() {
18          Queue queue = new Queue(QUEUE);
19          FanoutExchange exchange = new FanoutExchange(EXCHANGE);
20          return new Declarables(queue, exchange,
21              BindingBuilder.bind(queue).to(exchange));
22      }
23      //会员服务实例1
24      @RabbitListener(queues = QUEUE)
25      public void memberService1(String userName) {
26          log.info("memberService1: welcome message sent to new user {}", userName);
27      }
28      //会员服务实例2
29      @RabbitListener(queues = QUEUE)
30      public void memberService2(String userName) {
31          log.info("memberService2: welcome message sent to new user {}", userName);
32      }
33      //营销服务实例1
34      @RabbitListener(queues = QUEUE)
35      public void promotionService1(String userName) {
36          log.info("promotionService1: gift sent to new user {}", userName);
37      }
38      //营销服务实例2
39      @RabbitListener(queues = QUEUE)
```

```

42     public void promotionService2(String userName) {
43         log.info("promotionService2: gift sent to new user {}", userName);
44     }
45

```

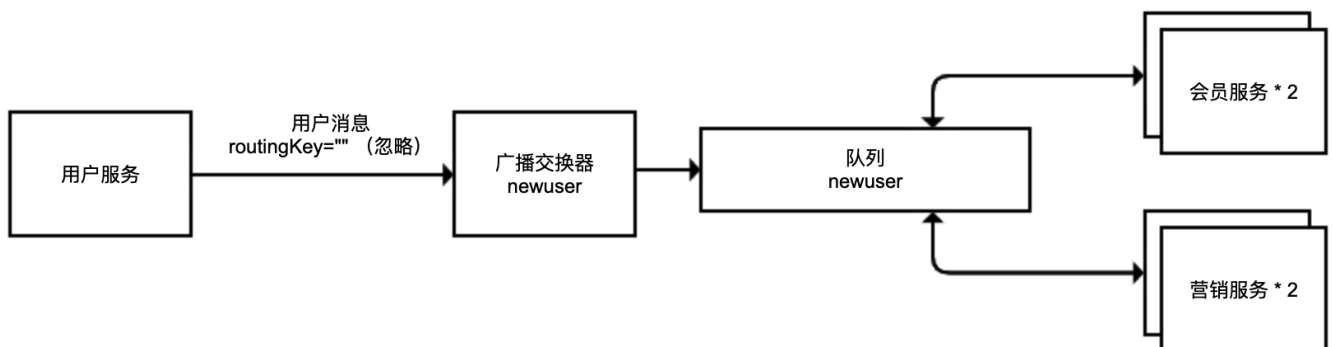
我们请求四次 sendMessage 接口，注册四个用户。通过日志可以发现，**一条用户注册的消息，要么被会员服务收到，要么被营销服务收到，显然这不是广播**。那，我们使用的 FanoutExchange，看名字就应该是实现广播的交换器，为什么根本没有起作用呢？

```

[18:53:21.802] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#0-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueWrong:44 ] -
memberService1: welcome message sent to new user 1080e392-2cac-4eaf-9a3e-1f18e86feb1f
[18:53:23.770] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#1-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueWrong:50 ] -
memberService2: welcome message sent to new user d00baca9-575d-436b-ba64-c5f124e81864
[18:53:25.716] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#2-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueWrong:56 ] -
promotionService1: gift sent to new user eeb08f9b-326a-463f-9c76-85bacl457ee
[18:53:26.733] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#3-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueWrong:61 ] -
promotionService2: gift sent to new user 9fc32a8a-4c44-4ffe-b429-ac4adddee32d

```

其实，广播交换器非常简单，它会忽略 routingKey，广播消息到所有绑定的队列。在这个案例中，两个会员服务 and 两个营销服务都绑定了同一个队列，所以这四个服务只能收到一次消息：



修改方式很简单，我们把队列进行拆分，会员和营销两组服务分别使用一条独立队列绑定到广播交换器即可：

复制代码

```

1  @Slf4j
2  @Configuration
3  @RestController
4  @RequestMapping("fanoutright")
5  public class FanoutQueueRight {
6      private static final String MEMBER_QUEUE = "newusermember";
7      private static final String PROMOTION_QUEUE = "newuserpromotion";
8      private static final String EXCHANGE = "newuser";
9      @Autowired
10     private RabbitTemplate rabbitTemplate;
11     @GetMapping

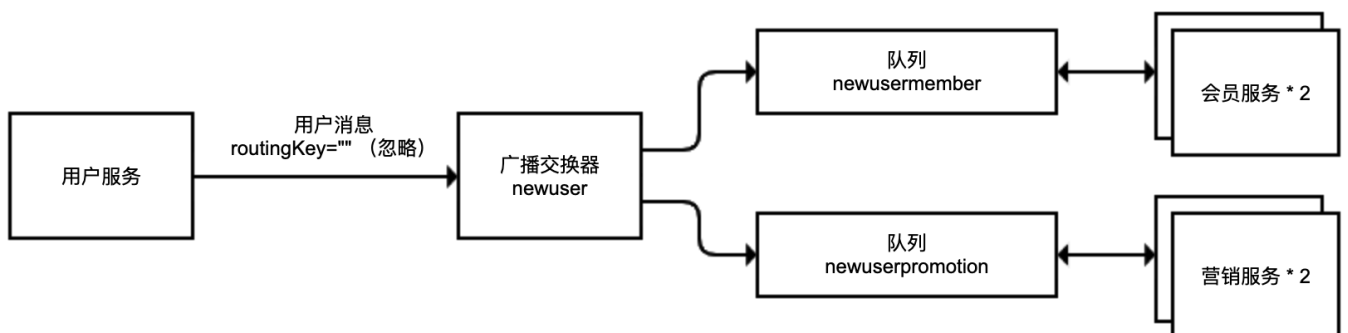
```

```

12     public void sendMessage() {
13         rabbitTemplate.convertAndSend(EXCHANGE, "", UUID.randomUUID().toString
14     }
15     @Bean
16     public Declarables declarables() {
17         //会员服务队列
18         Queue memberQueue = new Queue(MEMBER_QUEUE);
19         //营销服务队列
20         Queue promotionQueue = new Queue(PROMOTION_QUEUE);
21         //广播交换器
22         FanoutExchange exchange = new FanoutExchange(EXCHANGE);
23         //两个队列绑定到同一个交换器
24         return new Declarables(memberQueue, promotionQueue, exchange,
25                                 BindingBuilder.bind(memberQueue).to(exchange),
26                                 BindingBuilder.bind(promotionQueue).to(exchange));
27     }
28     @RabbitListener(queues = MEMBER_QUEUE)
29     public void memberService1(String userName) {
30         log.info("memberService1: welcome message sent to new user {}", userNa
31     }
32     @RabbitListener(queues = MEMBER_QUEUE)
33     public void memberService2(String userName) {
34         log.info("memberService2: welcome message sent to new user {}", userNa
35     }
36     @RabbitListener(queues = PROMOTION_QUEUE)
37     public void promotionService1(String userName) {
38         log.info("promotionService1: gift sent to new user {}", userName);
39     }
40     @RabbitListener(queues = PROMOTION_QUEUE)
41     public void promotionService2(String userName) {
42         log.info("promotionService2: gift sent to new user {}", userName);
43     }
44 }

```

现在，交换器和队列的结构是这样的：



从日志输出可以验证，对于每一条 MQ 消息，会员服务和营销服务分别都会收到一次，一条消息广播到两个服务的同时，在每一个服务的两个实例中通过轮询接收：

```
[19:02:19.774] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#2-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueRight:60 ] - promotionService1: gift sent to new user 109a080c-5497-440f-9ccc-009d6101bee7
[19:02:19.774] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#0-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueRight:48 ] - memberService1: welcome message sent to new user 109a080c-5497-440f-9ccc-009d6101bee7
[19:02:28.827] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#3-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueRight:65 ] - promotionService2: gift sent to new user 7e5ee753-967d-4274-b70c-60d01ec44120
[19:02:28.827] [org.springframework.amqp.rabbit.RabbitListenerEndpointContainer#1-1] [INFO ] [o.g.t.c.a.fanoutvswork.FanoutQueueRight:54 ] - memberService2: welcome message sent to new user 7e5ee753-967d-4274-b70c-60d01ec44120
```

所以说，理解了 RabbitMQ 直接交换器、广播交换器的工作方式之后，我们对消息的路由方式了解得很清晰了，实现代码就不会出错。

对于异步流程来说，消息路由模式一旦配置出错，轻则可能导致消息的重复处理，重则可能导致重要的服务无法接收到消息，最终造成业务逻辑错误。

每个 MQ 中间件对消息的路由处理的配置各不相同，我们一定要先了解原理再着手编码。


别让死信堵塞了消息队列

我们在介绍 [线程池](#) 的时候提到，如果线程池的任务队列没有上限，那么最终可能会导致 OOM。使用消息队列处理异步流程的时候，我们也同样要注意消息队列的任务堆积问题。对于突发流量引起的消息队列堆积，问题并不大，适当调整消费者的消费能力应该就可以解决。但在很多时候，消息队列的堆积堵塞，是因为有大量始终无法处理的消息。

比如，用户服务在用户注册后发出一条消息，会员服务监听到消息后给用户派发优惠券，但因为用户并没有保存成功，会员服务处理消息始终失败，消息重新进入队列，然后还是处理失败。这种在 MQ 中像幽灵一样回荡的同一条消息，就是死信。

随着 MQ 被越来越多的死信填满，消费者需要花费大量时间反复处理死信，导致正常消息的消费受阻，最终 MQ 可能因为数据量过大而崩溃。


我们来测试一下这个场景。首先，定义一个队列、一个直接交换器，然后把队列绑定到交换器：

 复制代码

```
1 @Bean
2 public Declarables declarables() {
3     //队列
4     Queue queue = new Queue(Consts.QUEUE);
5     //交换器
6     DirectExchange directExchange = new DirectExchange(Consts.EXCHANGE);
7     //快速声明一组对象，包含队列、交换器，以及队列到交换器的绑定
```

```
8     return new Declarables(queue, directExchange,
9                             BindingBuilder.bind(queue).to(directExchange).with(Consts.ROUTING_I
10
```

然后，实现一个 `sendMessage` 方法来发送消息到 MQ，访问一次提交一条消息，使用自增标识作为消息内容：

 复制代码

```
1 //自增消息标识
2 AtomicLong atomicLong = new AtomicLong();
3 @Autowired
4 private RabbitTemplate rabbitTemplate;
5
6 @GetMapping("sendMessage")
7 public void sendMessage() {
8     String msg = "msg" + atomicLong.incrementAndGet();
9     log.info("send message {}", msg);
10    //发送消息
11    rabbitTemplate.convertAndSend(Consts.EXCHANGE, msg);
12 }
```

收到消息后，直接抛出空指针异常，模拟处理出错的情况：

 复制代码

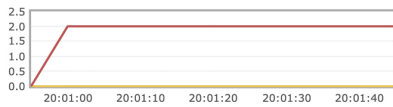
```
1 @RabbitListener(queues = Consts.QUEUE)
2 public void handler(String data) {
3     log.info("got message {}", data);
4     throw new NullPointerException("error");
5 }
```

调用 `sendMessage` 接口发送两条消息，然后来到 RabbitMQ 管理台，可以看到这两条消息始终在队列中，不断被重新投递，导致重新投递 QPS 达到了 1063。

Queue test

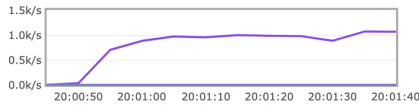
Overview

Queued messages last minute ?



Ready	0
Unacked	2
Total	2

Message rates last minute ?



Publish	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s
Deliver (manual ack)	1,063/s	Redelivered	1,063/s	Get (empty)	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s		

Details

Features	Policy	Operator policy	Effective policy definition	State	running
	Policy			Consumers	1
				Consumer utilisation	100%
				Messages	2
				Message body bytes	8iB
				Process memory	173kiB
				Total	2
				Ready	0
				Unacked	2
				In memory	2
				Persistent	2
				Transient, Paged Out	0

同时，在日志中可以看到大量异常信息：

[复制代码](#)

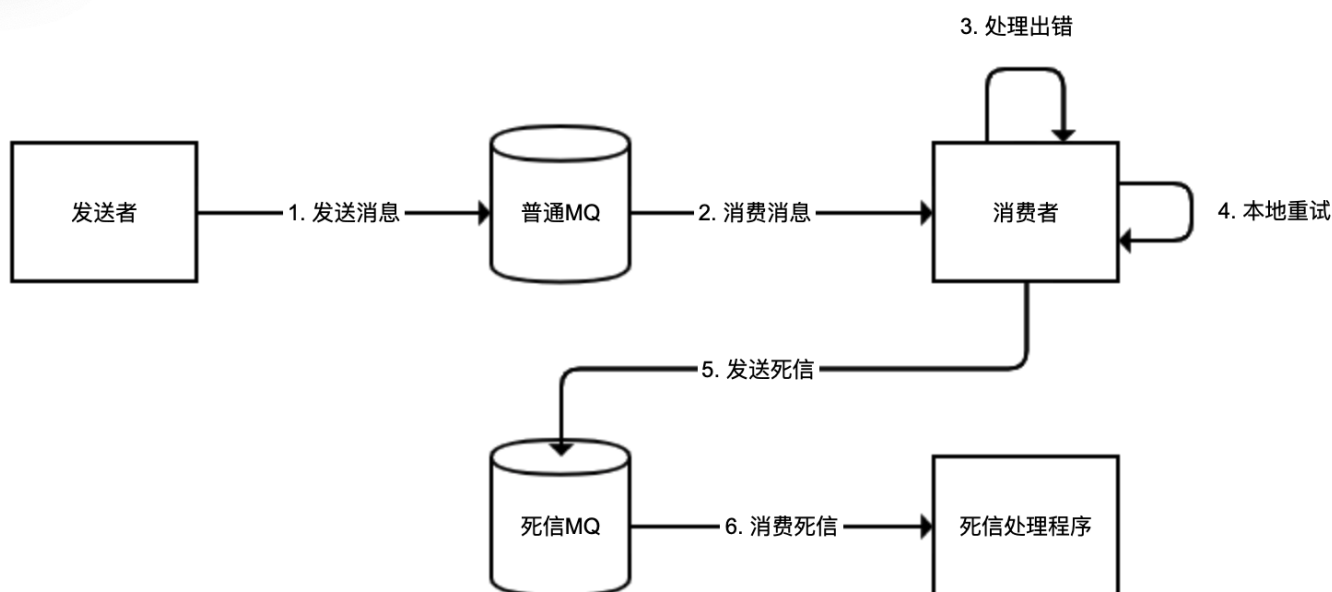
```
1 [20:02:31.533] [org.springframework.amqp.rabbit.RabbitListenerEndpointContain
2 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException: List
3 at org.springframework.amqp.rabbit.listener.adapter.MessagingMessageListener,
4 at org.springframework.amqp.rabbit.listener.adapter.MessagingMessageListener,
5 at org.springframework.amqp.rabbit.listener.adapter.MessagingMessageListener,
6 at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer
7 at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer
8 at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer
9 at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer
10 at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer
11 at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.d
12 at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.r
13 at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.a
14 at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer$A:
15 at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer$A:
16 at java.lang.Thread.run(Thread.java:748)
17 Caused by: java.lang.NullPointerException: error
18 at org.geekbang.time.commonmistakes.asyncprocess.deadletter.MQListener.handl
19 at sun.reflect.GeneratedMethodAccessor46.invoke(Unknown Source)
20 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
21 at java.lang.reflect.Method.invoke(Method.java:498)
22 at org.springframework.messaging.handler.invocation.InvocableHandlerMethod.d
23 at org.springframework.messaging.handler.invocation.InvocableHandlerMethod.i
24 at org.springframework.amqp.rabbit.listener.adapter.HandlerAdapter.invoke(Ha
25 at org.springframework.amqp.rabbit.listener.adapter.MessagingMessageListener,
26 ... 13 common frames omitted
```

解决死信无限重复进入队列最简单的方式是，在程序处理出错的时候，直接抛出 `AmqpRejectAndDontRequeueException` 异常，避免消息重新进入队列：

复制代码

```
1 throw new AmqpRejectAndDontRequeueException("error");
```

但，我们更希望的逻辑是，对于同一条消息，能够先进行几次重试，解决因为网络问题导致的偶发消息处理失败，如果还是不行的话，再把消息投递到专门的一个死信队列。对于来自死信队列的数据，我们可能只是记录日志发送报警，即使出现异常也不会再重复投递。整个逻辑如下图所示：




针对这个问题，Spring AMQP 提供了非常方便的解决方案：

首先，定义死信交换器和死信队列。其实，这些都是普通的交换器和队列，只不过被我们专门用于处理死信消息。

然后，通过 `RetryInterceptorBuilder` 构建一个 `RetryOperationsInterceptor`，用于处理失败时候的重试。这里的策略是，最多尝试 5 次（重试 4 次）；并且采取指数退避重试，首次重试延迟 1 秒，第二次 2 秒，以此类推，最大延迟是 10 秒；如果第 4 次重试还是失败，则使用 `RepublishMessageRecoverer` 把消息重新投入一个“死信交换器”中。


最后，定义死信队列的处理程序。这个案例中，我们只是简单记录日志。

对应的实现代码如下：

 复制代码

```
1 //定义死信交换器和队列，并且进行绑定
2 @Bean
3 public Declarables declarablesForDead() {
4     Queue queue = new Queue(Consts.DEAD_QUEUE);
5     DirectExchange directExchange = new DirectExchange(Consts.DEAD_EXCHANGE);
6     return new Declarables(queue, directExchange,
7         BindingBuilder.bind(queue).to(directExchange).with(Consts.DEAD_ROU
8     }
9 //定义重试操作拦截器
10 @Bean
11 public RetryOperationsInterceptor interceptor() {
12     return RetryInterceptorBuilder.stateless()
13         .maxAttempts(5) //最多尝试（不是重试）5次
14         .backOffOptions(1000, 2.0, 10000) //指数退避重试
15         .recoverer(new RepublishMessageRecoverer(rabbitTemplate, Consts.DE
16         .build();
17 }
18 //通过定义SimpleRabbitListenerContainerFactory，设置其adviceChain属性为之前定义的Ret
19 @Bean
20 public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(Con
21     SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerCon
22     factory.setConnectionFactory(connectionFactory);
23     factory.setAdviceChain(interceptor());
24     return factory;
25 }
26 //死信队列处理程序
27 @RabbitListener(queues = Consts.DEAD_QUEUE)
28 public void deadHandler(String data) {
29     log.error("got dead message {}", data);
30 }
```

执行程序，发送两条消息，日志如下：

 复制代码

```
1 [11:22:02.193] [http-nio-45688-exec-1] [INFO ] [o.g.t.c.a.d.DeadLetterControll
2 [11:22:02.219] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
3 [11:22:02.614] [http-nio-45688-exec-2] [INFO ] [o.g.t.c.a.d.DeadLetterControll
4 [11:22:03.220] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
5 [11:22:05.221] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
6 [11:22:09.223] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
7 [11:22:17.224] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
8 [11:22:17.226] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
9 [11:22:17.227] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
10 [11:22:17.229] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
```

```
11 [11:22:18.232] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
12 [11:22:20.237] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
13 [11:22:24.241] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
14 [11:22:32.245] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
15 [11:22:32.246] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
16 [11:22:32.250] [org.springframework.amqp.rabbit.RabbitListenerEndpointContaine
```

可以看到：

msg1 的 4 次重试间隔分别是 1 秒、2 秒、4 秒、8 秒，再加上首次的失败，所以最大尝试次数是 5。

4 次重试后，RepublishMessageRecoverer 把消息发往了死信交换器。

死信处理程序输出了 got dead message 日志。

这里需要尤其注意的一点是，虽然我们几乎同时发送了两条消息，但是 msg2 是在 msg1 的四次重试全部结束后才开始处理。原因是，**默认情况下**

SimpleMessageListenerContainer 只有一个消费线程。可以通过增加消费线程来避免性能问题，如下我们直接设置 concurrentConsumers 参数为 10，来增加到 10 个工作线程：

```
1 @Bean
2 public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(Coni
3     SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerCon
4     factory.setConnectionFactory(connectionFactory);
5     factory.setAdviceChain(interceptor());
6     factory.setConcurrentConsumers(10);
7     return factory;
8 }
```

 复制代码

当然，我们也可以设置 maxConcurrentConsumers 参数，来让 SimpleMessageListenerContainer 自己动态地调整消费者线程数。不过，我们需要特别注意它的动态开启新线程的策略。你可以通过 [🔗 官方文档](#)，来了解这个策略。

重点回顾

在使用异步处理这种架构模式的时候，我们一般都会使用 MQ 中间件配合实现异步流程，需要重点考虑四个方面的问题。

第一，要考虑异步流程丢消息或处理中断的情况，异步流程需要有备线进行补偿。比如，我们今天介绍的全量补偿方式，即便异步流程彻底失效，通过补偿也能让业务继续进行。

第二，异步处理的时候需要考虑消息重复的可能性，处理逻辑需要实现幂等，防止重复处理。

第三，微服务场景下不同服务多个实例监听消息的情况，一般不同服务需要同时收到相同的消息，而相同服务的多个实例只需要轮询接收消息。我们需要确认 MQ 的消息路由配置是否满足需求，以避免消息重复或漏发问题。

第四，要注意始终无法处理的死信消息，可能会引发堵塞 MQ 的问题。一般在遇到消息处理失败的时候，我们可以设置一定的重试策略。如果重试还是不行，那可以把这个消息扔到专有的死信队列特别处理，不要让死信影响到正常消息的处理。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

思考与讨论

1. 在用户注册后发送消息到 MQ，然后会员服务监听消息进行异步处理的场景下，有些时候我们会发现，虽然用户服务先保存数据再发送 MQ，但会员服务收到消息后去查询数据库，却发现数据库中还没有新用户的信息。你觉得，这可能是什么问题呢，又该如何解决呢？
2. 除了使用 Spring AMQP 实现死信消息的重投递外，RabbitMQ 2.8.0 后支持的死信交换器 DLX 也可以实现类似功能。你能尝试用 DLX 实现吗，并比较下这两种处理机制？

关于使用 MQ 进行异步处理流程，你还遇到过其他问题吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 24 | 业务代码写完, 就意味着生产就绪了?

下一篇 26 | 数据存储: NoSQL与RDBMS如何取长补短、相辅相成?

精选留言 (11)

写留言



vivi

2020-05-12

我之前做过一个demo 是基于canal做mysql数据同步, 需要将解析好的数据发到kafka里面, 再进行处理。在使用的时候发现这么一个问题, 就是kafka多partition消费时不能保证消息的顺序消费, 进而导致mysql数据同步异常。

由于kafka可以保证在同一个partition内消息有序, 于是我自定义了一个分区器, 将数据的id取hashcode然后根据partition的数量取余作为分区号, 保证同一条数据的binlog能投...
展开

作者回复: 这个实现很赞

3

10



Darren

2020-05-12

第一个问题：

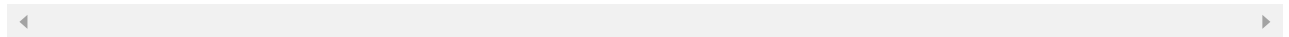
每天晒白牙大佬的回答和老师的回复已经很棒了，我就不班门弄斧了。

第二个问题：

自定义的私信队列，其实是发送失败，主要是生产者发送到mq的时候，发送失败，进了自定义的私信队列；...

展开 ▾

作者回复: 感谢分享 📬



💬 1

👍 5



每天晒白牙

2020-05-12

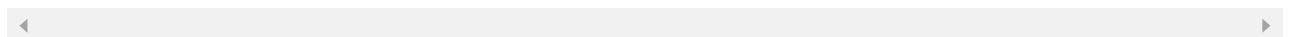
老师，我理解的异步处理不仅仅是通过 MQ 来实现，还有其他方式
比如开新线程执行，返回 Future
还有各种异步框架，比如 Vertx，它是通过 callback 的方式实现

思考题...

展开 ▾

作者回复: 是的，或许本文标题可以改为消息队列：XXX 😊，不过文中的一些点是可以泛化到你提到的两种异步处理的

思考题一是我真实遇到的问题，当时倒不是因为主从的问题，而是因为业务代码把保存数据和发 MQ 消息放在了一个事务中，有概率收到消息的时候事务还没有提交完成，当时开发同学的处理方式是收 MQ 消息的时候 sleep 1 秒，或许应该是先提交事务，完成后再发 MQ 消息，但是这又出来一个问题 MQ 消息发送失败怎么办？所以后来演化为建立本地消息表来确保 MQ 消息可补偿，把业务处理和保存 MQ 消息到本地消息表操作在相同事务内处理，然后异步发送和补偿发送消息表中的消息到 MQ



💬 3

👍 4



203.

2020-05-12

老师 我这里有个问题 关于 Stream 的，业务需求里需要按某几个字段去重(acctId,billingCycleId,prodInstId,offerId)
我这里想到了遍历集合 areaDatas 后用 contains 方法判断 重写 AcctItemYzfBean 实体类的 equals 方法实现，

请问有没有更好的方法？代码如下...

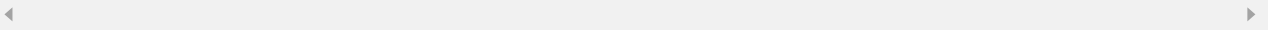
展开 ▾

作者回复: 比如下面的类，id1和id2重复认为是重复的，id3不需要考虑

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
static class Test {
    private String id1;
    private String id2;
    @EqualsAndHashCode.Exclude
    private String id3;
}
```

通过Set去重或者通过distinct去重即可：

```
List<Test> list = new ArrayList<>();
list.add(new Test("a","b","c"));
list.add(new Test("a","b","d"));
System.out.println(list.stream().collect(Collectors.toSet()));
System.out.println(list.stream().distinct().collect(Collectors.toList()));
```



💬 1

👍 2

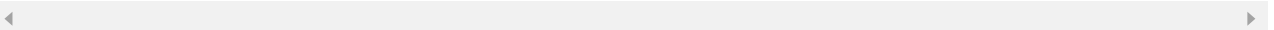


郭石龙

2020-05-12

老师，你好，如果有多个补偿实例，会不会造成消息重复？

作者回复: 补偿需要配合幂等



💬

👍 1



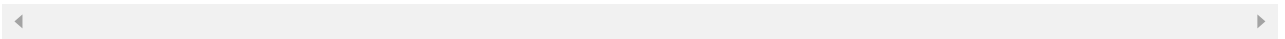
飞翔

2020-05-30

老师 生产者发送给mq消息 即使异步发送也会有listener 来监听投递消息是否成功 如果失败 重试不就行了？ 不是类似kafka 有100%投递 100%保证消费的配置嘛

展开 ▾

作者回复: 你是指补偿吗？我遇到过mq瘫痪的情况，没有补偿这个时候除了干着急我们还能做啥



似曾相识

2020-05-17

“ConcurrentHashMap 来存放那些发过短信的用户 ID 实现幂等”

老师 3.能将ConcurrentHashMap换成 LinkedHashMap吗？通过LUR 还可以定时 删除一些数据，避免集合过大，这样做对吗？

展开 ∨



似曾相识

2020-05-17

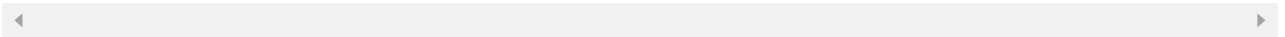
老师

1.如果实际生产中用使用 ConcurrentHashMap 来存放那些发过短信的用户 ID 实现幂等，如何一直往map中增加，会不会oom呢？

2.如果数据量巨大 使用ConcurrentSkipListMap 跳表会不会更好一些呢？

展开 ∨

作者回复: 这只是demo生产应用肯定用数据库做幂等的



G小调

2020-05-15

第一个问题，是否可以这样解决

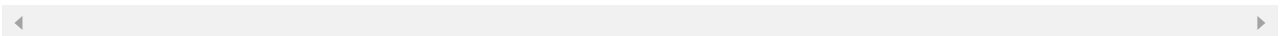
1.先保存用户注册的数据，同时记录下要发送mq的消息，入库在一个事务里

2.通过异步任务定时拉取mq的消息表，发送到mq，进行处理

但这个有个问题，异步任务就能执行mq的的业务，那mq的价值是不是减少了

展开 ∨

作者回复: 其实这就是本地事务消息的实现 第二步不一定需要定时任务拉取 第一步完成后直接发mq即可 定时任务拉取只用来补偿



汝林外史

2020-05-13

哈哈，写一个几率比较小的情况：注册register的代码中把异常都吃掉了，没抛出来，注册又报错了，但还是继续执行并且发了消息。

展开 ▾



王鹏

2020-05-12

mq发信息写到了事务中，导致了mq的消费时，事务还没有提交

作者回复: 是，我遇到的就是这个情况

