RabbitMQ

xbZhong

2025-10-06

本页PDF

RabbitMQ

是一种消息队列，负责在不同的服务之间传递消息

核心作用：解耦、异步、削峰填谷、可靠消息传递

同步调用

整个流程是**顺序执行**的，一步接一步，必须等前一步完成才能开始下一步

优势：

- 时效性强，等待到结果才返回

问题：

- 扩展性差
- 性能下降
- 级联失败

异步调用

基于消息通知的方式，包含三个角色：

- **消息发送者：**投递消息的人，就是原来的**调用者**
- **消息接收者：**接收和处理消息的人，就是原来的**服务提供**者
- **消息代理：**管理、暂存、转发消息

优势：

- 耦合度低，扩展性强
- 异步调用，无需等待
- 故障隔离，下游服务故障不影响上游服务
- 缓存消息，流量削峰填谷

问题：

- 不能立刻得到结果，时效性差
- 不确定下游业务是否执行成功

基本介绍

RabbitMQ整体架构

- `virtual-host`：虚拟主机，起到数据隔离的作用

- `publisher`：消息发送者
- `consumer`：消息的消费者
- `queue`：队列，存储信息
- `exchange`：交换机，负责路有消息

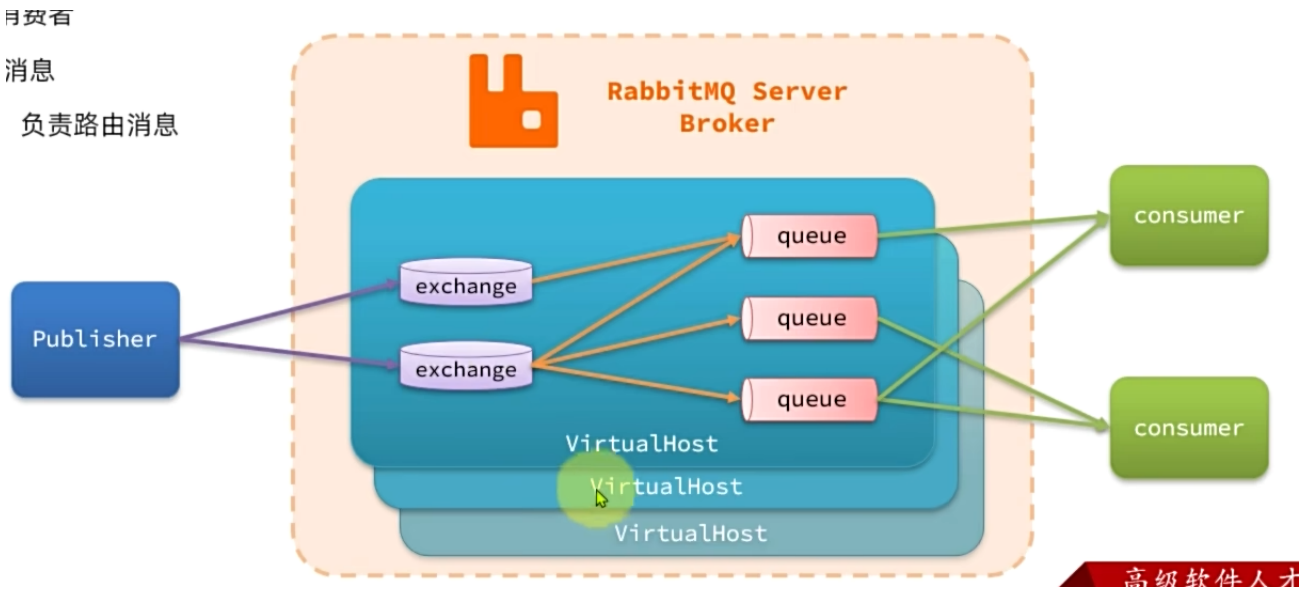

image-20251004143308203

Java客户端

使用`Spring AMQP`进行开发，是基于`AMQP`协议定义的一套API规范

操作步骤

- 导入依赖

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

- 进行配置

```
spring:
  rabbitmq:
    host: 192.168.150.101 # 你的虚拟机IP
    port: 5672 # 端口
    virtual-host: /hmall # 虚拟主机
    username: hmall # 用户名
    password: 123 # 密码
```

- `SpringAMQP` 提供了 `RabbitTemplate` 工具类，方便我们发送消息
  - 使用的时候引入 `RabbitTemplate` 工具类，使用**两个参数**的 `convertAndSend` 方法**指定队列名和信息**进行消息发送
- `SpringAMQP` 提供声明式的消息监听，我们需要通过**注解**在方法上**声明要监听的队列名称**
  - 自定义类，并使用 `@Component` 将其注册为Bean对象
  - 使用 `@RabbitListener` 声明要**监听的队列名**

Work Queues

任务模型，让**多个消费者绑定到一个队列**，共同消费队列中的消息

- 同一个消息**只能被一个消费者处理**
- 默认情况下，多条消息会采用**轮询**的方式被**均匀分配**给多个消费者，**不会**根据消费者的处理速度或负载来调整分配，而是严格按顺序分发
  - 修改配置文件，设置 `preFetch` 值为1，确保**同一时刻最多投递给消费者1条消息**，实现**能者多劳**

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        prefetch: 1 # 每次只能获取一条消息，处理完成才能获取下一个消息
```

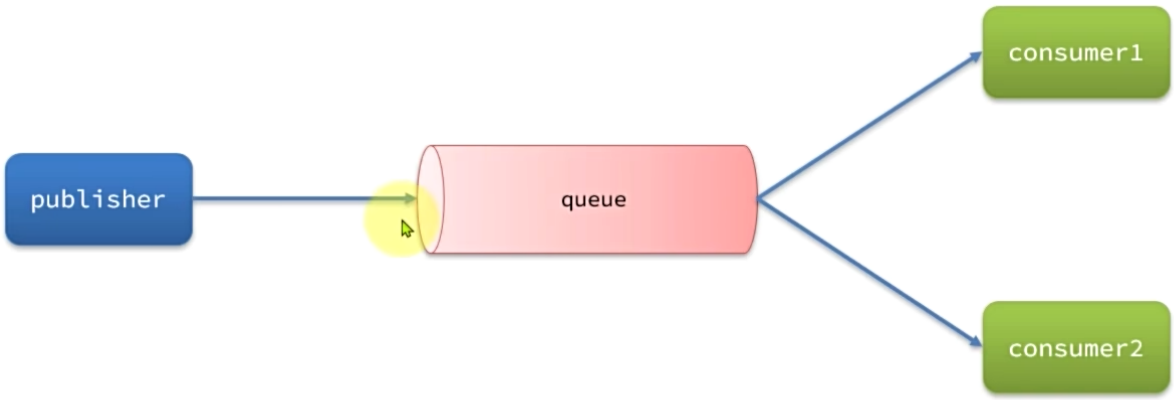

image-20251005125952532

交换机

交换机的作用主要是**接收**发送者发送的消息，并将消息**路由**到与其绑定的队列

常用交换机类型：

- `Fanout`：广播
- `Direct`：定向
- `Topic`：话题

Fanout交换机

`Fanout Exchange` 会将接收到的消息路由到**每一个跟其绑定的** `queue`，也叫广播模式

使用步骤：

- 同上，但是使用**三个参数**的 `convertAndSend` 方法**指定交换机名称、队列名和信息**进行消息发送

```
rabbitTemplate.convertAndSend(exchangeName,null,message);
```

Direct交换机

`Direct Exchange` 会将接收到的消息根据规则路由到指定的Queue，称为**定向路由**

- 每一个Queue都与Exchange设置一个 **`BindingKey`**
- 发布者发送消息时，指定消息的 **`RoutingKey`**
- Exchange将消息路由到 `BindingKey` 与消息 `RoutingKey` **一致**的队列

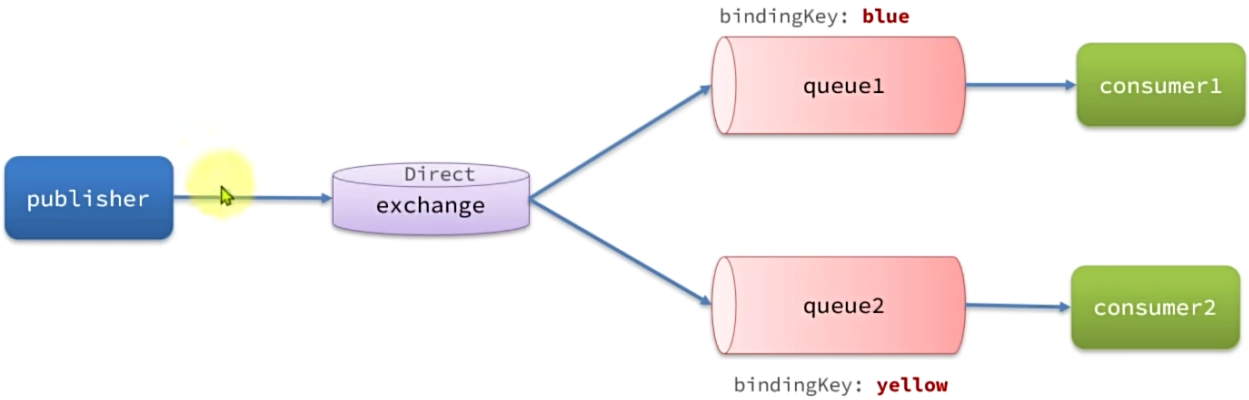

image-20251005141740256

使用方式：

- 使用三个参数的 `convertAndSend` 方法指定 `RoutingKey`

```
rabbitTemplate.convertAndSend(exchangeName,"red",message);
```

Topic交换机

`Topic Exchange` 也是基于 `RoutingKey` 做消息路由，但是 `RoutingKey` 通常是多个单词的组合，并且以 `.` 分割

`Queue` 与 `Exchange` 指定 `BingdingKey` 时可以使用通配符：

- `#`：代指0或多个单词
- `*`：代指一个单词

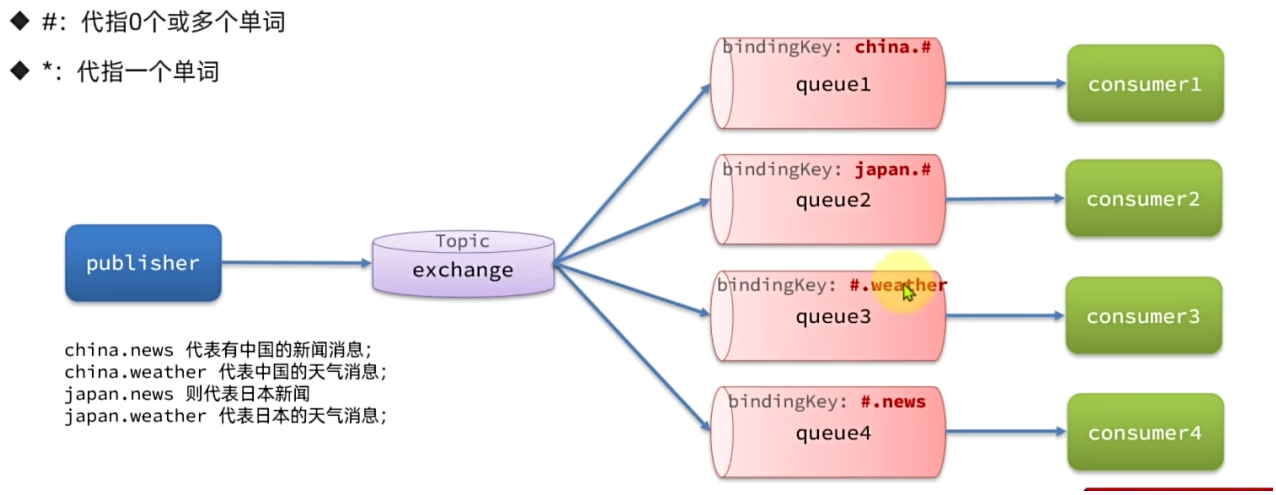

image-20251005143159018

使用方式：

- 使用三个参数的 `convertAndSend` 方法指定 `RoutingKey`

```
rabbitTemplate.convertAndSend(exchangeName,"usa.news",message);
```

声明队列交换机

基于Bean声明交换机

`SpringAMQP` 提供了几个类，用来声明**队列、交换机及其绑定关系**

- `Queue`：用于声明**队列**，可以用工厂类 `QueueBuilder` 构建
- `Exchange`：用于声明**交换机**，可以用工厂类 `ExchangeBuilder` 构建

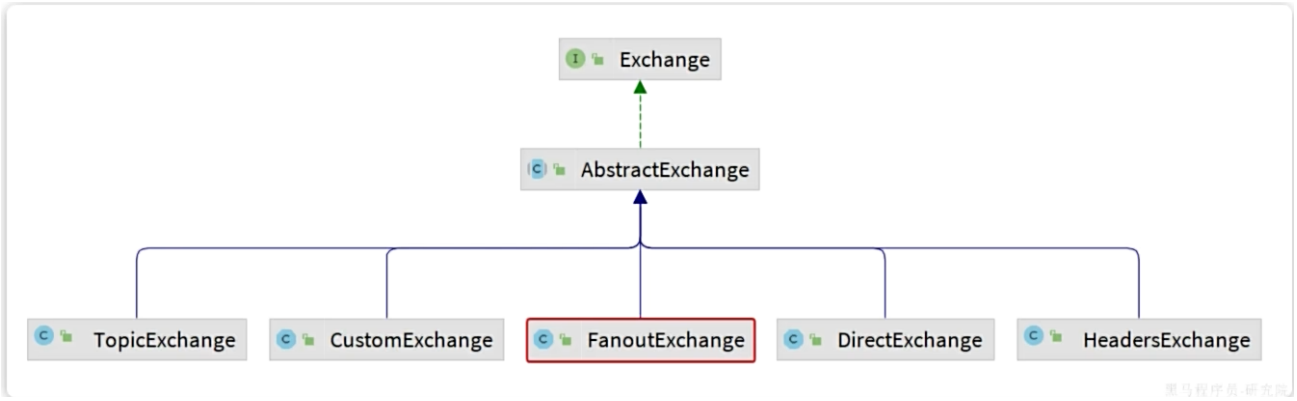

image-20251005143848191

- `Bingding`：用于声明**队列和交换机的绑定关系**，可以用工厂类 `BindingBuilder` 构建

例如：需要使用**自定义配置类**进行声明

- `Binding bindingQueue1(Queue fanoutQueue1, FanoutExchange fanoutExchange)`：参数是Bean对象，Spring默认传入

```java
@Configuration
public class FanoutConfig {
    /**
     * 声明交换机
     * @return Fanout类型交换机
     */
    @Bean
    public FanoutExchange fanoutExchange(){
        return new FanoutExchange("hmall.fanout");
    }

    /**
     * 第1个队列
     */
    @Bean
    public Queue fanoutQueue1(){
        return new Queue("fanout.queue1");
    }

    /**
     * 绑定队列和交换机
     */
    @Bean
    public Binding bindingQueue1(Queue fanoutQueue1, FanoutExchange fanoutExchange){
        return BindingBuilder.bind(fanoutQueue1).to(fanoutExchange);
    }

    /**
     * 第2个队列
     */
    @Bean
    public Queue fanoutQueue2(){
        return new Queue("fanout.queue2");
    }

    /**
     * 绑定队列和交换机
     */
    @Bean
    public Binding bindingQueue2(Queue fanoutQueue2, FanoutExchange fanoutExchange){
        return BindingBuilder.bind(fanoutQueue2).to(fanoutExchange);
    }
}
```

基于注解声明交换机

`SpringAMQP` 还提供了基于 `@RabbitListener` 注解声明队列和交换机的方式：

- 使用 `bindings` 参数指定绑定关系
  - 使用 `values` 参数指定消息队列
  - 使用 `exchange` 参数指定交换机
    - 使用 `name` 参数指定交换机名称，使用 `type` 指定交换机类型
  - 使用 `key` 指定 `bindingKeys` ，接收参数为 `List` 类型

```
@RabbitListener(bindings = @QueueBinding(
    value = @Queue(name = "direct.queue1"),
    exchange = @Exchange(name = "hmall.direct", type = ExchangeTypes.DIRECT),
    key = {"red", "blue"}
))
public void listenDirectQueue1(String msg){
    System.out.println("消费者1接收到direct.queue1的消息：【" + msg + "】");
}
```

消息转换器

底层会基于**JDK自带序列化方式**对对象进行序列化

建议采用**JSON序列化**代替默认的JDK序列化：

- 引入 `jackson` 依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.9.10</version>
</dependency>
```

- **在启动类**配置 `MessageConvertor`

```
@Bean
public MessageConverter messageConverter(){
    // 1.定义消息转换器
    Jackson2JsonMessageConverter jackson2JsonMessageConverter = new
Jackson2JsonMessageConverter();
    // 2.配置自动创建消息id，用于识别不同消息，也可以在业务中基于ID判断是否是重复消息
    jackson2JsonMessageConverter.setCreateMessageIds(true);
    return jackson2JsonMessageConverter;
}
```

发送者可靠性

发送者重连

可能会出现发送者连接MQ失败的情况，通过配置可以**开启连接失败后的重连机制**：

- 但是重连机制是阻塞的，会**影响业务性能**

```yaml
spring:
  rabbitmq:
    connection-timeout: 1s # 设置MQ的连接超时时间
    template:
      retry:
        enabled: true # 开启超时重试机制
        initial-interval: 1000ms # 失败后的初始等待时间
        multiplier: 1 # 失败后下次的等待时长倍数，下次等待时长 = initial-interval * multiplier
        max-attempts: 3 # 最大重试次数
```

发送者确认

- `ConfirmCallback` 关注 **消息是否到达 Exchange**
  - 消息到达Exchange返回ACK，否则返回NACK
- `ReturnsCallback` 关注 **消息是否从 Exchange 路由到 Queue**
  - 触发条件是：**消息成功到达 Exchange，但无法路由到任何队列**

`SpringAMQP` 提供了 **`Publisher Confirm`** **和** **`Publisher Return`** 两种确认机制，开启确认机制后，发送者发送信息给MQ，MQ会返回确认结果给发送者，有以下几种情况：

- 消息投递到了MQ，但是路由失败，此时会通过 `PublisherReturn` **返回路由异常原因**，然后返回 `ACK` ，告知投递成功
- 临时消息投递到了 `Exchange` ，并且入队成功，通过 `PublisherConfirm` 返回 `ACK` ，告知投递成功
- 持久消息投递到了 `Exchange` ，并且入队成功并完成持久化，通过 `PublisherConfirm` 返回 `ACK` ，告知投递成功
- 其它情况都会返回 `NACK` ，告知投递失败

开启确认机制

- 在 `yaml` 文件添加配置
  - `publisher-confirm-type` 有三种模式可选：
    - `none` ：关闭 `confirm` 机制
    - `simple` ：同步阻塞等待MQ的回执信息
    - `correlated` ：MQ异步回调方式返回回执信息

```yaml
spring:
  rabbitmq:
    publisher-confirm-type: correlated # 开启publisher confirm机制，并设置confirm类型
    publisher-returns: true # 开启publisher return机制
```

- 每个 `RabbitTemplate` 在项目启动的时候配置一个 `ReturnCallback`

```java
@Slf4j
@AllArgsConstructor
@Configuration
public class MqConfig {
    private final RabbitTemplate rabbitTemplate;

    @PostConstruct
    public void init(){
        rabbitTemplate.setReturnsCallback(new RabbitTemplate.ReturnsCallback() {
            @Override
            public void returnedMessage(ReturnedMessage returned) {
                log.error("触发return callback,");
                log.debug("exchange: {}", returned.getExchange());
                log.debug("routingKey: {}", returned.getRoutingKey());
                log.debug("message: {}", returned.getMessage());
                log.debug("replyCode: {}", returned.getReplyCode());
                log.debug("replyText: {}", returned.getReplyText());
            }
        });
    }
```

- 发送消息的时候**指定消息ID、消息** `ConfirmCallback`
  - 需要获取 `Future` 对象并且重写两个方法，定义**回调函数**
  - 发送消息的时候需要把 `CorrelationData` 对象发送过去，**后续可以通过其获取执行结果**

```java
@Test
void testPublisherConfirm() {
    // 1.创建CorrelationData
    CorrelationData cd = new CorrelationData();
    // 2.给Future添加ConfirmCallback
    cd.getFuture().addCallback(new ListenableFutureCallback<CorrelationData.Confirm>() {
        @Override
        public void onFailure(Throwable ex) {
            // 2.1.Future发生异常时的处理逻辑，基本不会触发
            log.error("send message fail", ex);
        }
        @Override
        public void onSuccess(CorrelationData.Confirm result) {
            // 2.2.Future接收到回执的处理逻辑，参数中的result就是回执内容
            if(result.isAck()){ // result.isAck()，boolean类型，true代表ack回执，false 代表 nack回执
                log.debug("发送消息成功，收到 ack!");
            }else{ // result.getReason()，String类型，返回nack时的异常描述
                log.error("发送消息失败，收到 nack, reason : {}", result.getReason());
            }
        }
    });
    // 3.发送消息
    rabbitTemplate.convertAndSend("hmall.direct", "q", "hello", cd);
}
```

数据持久化

RabbitMQ实现**数据持久化**包括3个方面：

- 交换机持久化
- 队列持久化
- 消息持久化
  - 会把消息**同时保存在内存和磁盘**

消息非持久化的问题：

- 内存达到上限，需要把数据临时写入磁盘缓解内存压力，此时会**造成MQ阻塞**
- 消息可能会丢失

Lazy Queue

惰性队列：

- 接收到消息后**直接存入磁盘，不再存储到内存**
  - 所有消息（包括非持久化消息）都会写入到磁盘，但**重启后非持久化消息不会保留**

- 消费者要消费消息时才会从**磁盘中读取并加载到内存**（可以**提前缓存部分消息到内存**，最多2048条）

在3.12版本后，所有队列都是Lazy Queue模式，无法更改

声明方式：声明队列时，指定`x-queue-mode`属性为lazy即可

- 基于Bean的方式

```
@Bean
public Queue queue(){
    return QueueBuilder
        .durable("Lazy-queue")
        .lazy()
        .build();
}
```

- 基于注解的方式

```
@RabbitListener(queuesToDeclare = @Queue(
        name = "lazy.queue",
        durable = "true",
        arguments = @Argument(name = "x-queue-mode", value = "lazy")
))
public void listenLazyQueue(String msg){
    log.info("接收到 lazy.queue的消息: {}", msg);
}
```

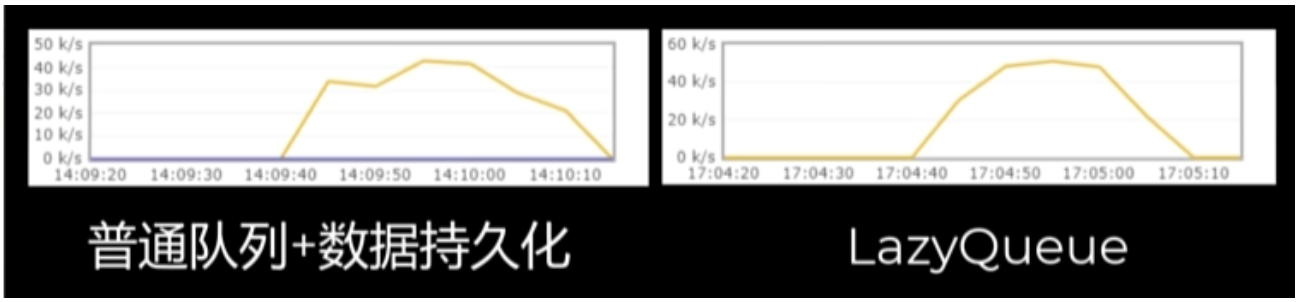

image-20251006144125410

消费者的可靠性

消费者确认机制

是为了确认消费者**是否成功处理消息**，当消费者处理消息结束后，应该向RabbitMQ发送一个回执，告知RabbitMQ自己的**消息处理状态**：

- **ack**：成功处理消息，RabbitMQ从队列中**删除该消息**
- **nack**：消息处理失败，RabbitMQ需要**再次投递消息**
- **reject**：消息处理失败并拒绝该消息，RabbitMQ从队列中**删除该消息**

- 消息格式有问题，无论重试几次都会失败

`SpringAMQP` 允许我们**通过配置文件选择ACK处理方式**：

- **`none`**：不处理。消息投递给消费者后立刻ack，消息会立刻从MQ删除，非常不安全
- **`manual`**：手动模式。需要自己在业务代码中调用api，发送ack或者reject，灵活
- **`auto`**：自动模式。`SpringAMQP` 利用AOP对我们的消息处理逻辑做了**环绕增强**，当业务正常执行时自动返回ack，出现异常时根据异常判断返回不同结果：
  - 业务异常，返回nack
  - 消息处理或校验异常，返回reject

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: none # 不做处理
```

消费者失败重试机制

`SpringAMQP` 提供了消费者**失败重试机制**，在消费者出现异常时**利用本地重试**，而不是无限的requeue到MQ

可以通过 `yaml` 文件进行配置：

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        retry:
          enabled: true # 开启消费者失败重试
          initial-interval: 1000ms # 初识的失败等待时长为1秒
          multiplier: 1 # 失败的等待时长倍数，下次等待时长 = multiplier * last-interval
          max-attempts: 3 # 最大重试次数
          stateless: true # true无状态；false有状态。如果业务中包含事务，这里改为false
```

开启重试模式之后，重试次数耗尽，如果消息依然失败，则需要用 **`MessageRecoverer`** 接口来处理，它包含三种不同的实现：

- `RejectAndDontRequeueRecoverer`：重试耗尽后，直接reject，丢弃消息，默认的实现方式
- `ImmediateRequeueMessageRecoverer`：重试耗尽后，返回nack，消息重新入队
- `RepublishMessageRecoverer`：重试耗尽后，将失败消息投递到指定的交换机

实现方式：

- 定义接受失败消息的交换机、队列及其绑定关系
- 定义 `RepublishMessageRecoverer`，声明为Bean对象

```java
@Bean
    public MessageRecoverer errorMessageRecover(RabbitTemplate rabbitTemplate) {
        return new
RepublishMessageRecoverer(rabbitTemplate,"error.direct","error");
    }
```

业务幂等性

幂等是一个数学概念，即 `f(x)=f(f(x))` ，指的是用一个业务，执行**一次或多次对业务状态的影响是一致的**

image-20251006152239253

唯一消息id

给每个消息都设置一个唯一id，利用id区分是否是重复消息：

- 每一条消息都生成一个唯一的id，与消息一起投递给消费者
- 消费者接收到消息后处理自己的业务，业务处理成功后将消息ID保存到数据库
- 下次又收到相同消息，去数据库查询是否存在，存在则视为重复消息放弃处理

在**消息转换器中进行设置**

```java
@Bean
public MessageConverter messageConverter(){
    // 1.定义消息转换器
    Jackson2JsonMessageConverter jjmc = new Jackson2JsonMessageConverter();
    // 2.配置自动创建消息id，用于识别不同消息，也可以在业务中基于ID判断是否是重复消息
    jjmc.setCreateMessageIds(true);
    return jjmc;
}
```

监听的时候使用 `Message` 类型进行接收

- 使用 `getBody()` 获得消息信息
- 使用 `getMessageProperties().getMessageId()` 获得消息ID

延迟消息

延迟消息：发送者发送消息时指定一个时间，**消费者不会立刻收到信息，而是指定时间之后才收到消息**

死信交换机

当一个队列中的消息满足下列情况之一时，可以成为**死信**：

- 消费者使用 `basic.reject` 或 `basic.nack` 声明消费失败，并且消息的 `requeue` 参数设置为false
- 消息是一个**过期消息**，超时无人消费
- 要投递的**队列消息满了**，无法投递

如果一个队列中的消息已经成为死信，并且这个队列通过 `dead-letter-exchange` 属性指定了一个交换机，那么队列中的死信就会投递到这个交换机中，而这个交换机就称为**死信交换机**

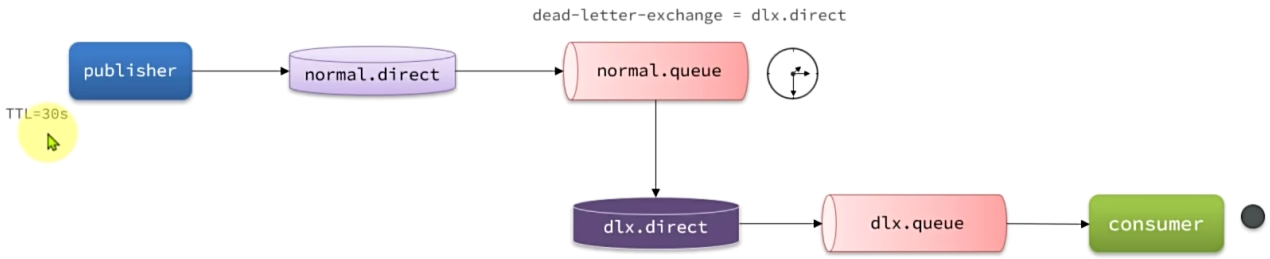

image-20251006154840679

- 给消息设置过期时间
- 普通队列不设置消费者
- 消息过期后投递给死信交换机，交给其消费者处理

延迟消息插件

可以将普通交换机**改造为支持延迟消息功能的交换机**，当消息投递到交换机后可以暂存一定时间，到期后再**投递给队列**

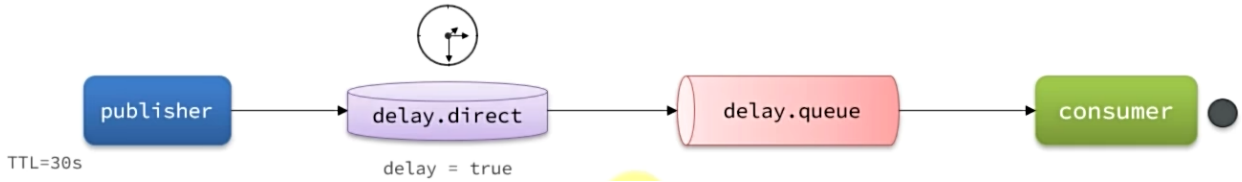

image-20251006155739041

需要**下载插件**，并将其集成到RabbitMQ中

基于注解的方式：在 `@Exchange` 注解增加一个 `delayed` 的参数，使其为true

```
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(name = "delay.queue", durable = "true"),
        exchange = @Exchange(name = "delay.direct", delayed = "true"),
        key = "delay"
))
public void listenDelayMessage(String msg){
    log.info("接收到delay.queue的延迟消息：{}", msg);
}
```

基于Bean的方式：使用 `.delayed()` 声明

```java
@Bean
    public DirectExchange delayExchange(){
        return ExchangeBuilder
                .directExchange("delay.direct") // 指定交换机类型和名称
                .delayed() // 设置delay的属性为true
                .durable(true) // 持久化
                .build();
    }
```

发送消息的时候需要通过消息头 `x-delay` 来**设置过期时间**

- 使用一个**消息后置处理器**的匿名内部类，使用 `.getMessageProperties()` 方法获取消息头，用 `.setDelay()` 方法对消息设置过期时间

```java
// 2.发送消息，利用消息后置处理器添加消息头
    rabbitTemplate.convertAndSend("delay.direct", "delay", message, new
MessagePostProcessor() {
        @Override
        public Message postProcessMessage(Message message) throws AmqpException {
            // 添加延迟消息属性
            message.getMessageProperties().setDelay(5000);
            return message;
        }
    });
```