

# RabbitMQ

消息队列中间件

消息队列中间件

消息

消息队列中间件

消息

- 消息队列中间件

消息

- 消息
- 消息
- 消息

消息

消息队列中间件

- 消息队列中间件
- 消息队列中间件
- 消息队列中间件

消息

- 消息队列中间件
- 消息队列中间件
- 消息队列中间件
- 消息队列中间件


消息

- 消息队列中间件
- 消息队列中间件

消息

RabbitMQ消息

- `virtual-host` 消息队列中间件
- `publisher` 消息队列
- `consumer` 消息队列
- `queue` 消息队列
- `exchange` 消息队列

image-20251004143308203

Java消息

Spring AMQP 依赖\*\* AMQP \*\*API

依赖

- 依赖

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

- 依赖

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

- SpringAMQP 使用 RabbitTemplate 发送消息
  - 使用 RabbitTemplate 的 convertAndSend 方法发送消息
- SpringAMQP 接收消息
  - 使用 @Component 标注 Bean
  - 使用 @RabbitListener 标注方法

## Work Queues

工作队列

- 工作队列
- 工作队列
  - 使用 prefetch 设置预取消息数量1

```
spring:
  rabbitmq:
    listener:
      simple:
        prefetch: 1 # 预取消息数量
```

image-20251005125952532

图

工作队列

工作队列

- Fanout 消息
- Direct 消息
- Topic 消息

### Fanout消息

Fanout Exchange 消息通过\*\*消息队列 queue \*\*消息队列

消息

- 消息通过 convertAndSend 消息队列

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

### Direct消息

Direct Exchange 消息通过Queue消息队列

- 消息Queue通过Exchange\*\* BindingKey \*\*
- 消息通过\*\* RoutingKey \*\*
- Exchange消息 BindingKey 消息 RoutingKey 消息



image-20251005141740256

消息

- 消息通过 convertAndSend 消息 RoutingKey

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

### Topic消息

Topic Exchange 消息 RoutingKey 消息 RoutingKey 消息 . 消息

Queue 消息 Exchange 消息 BindingKey 消息

- # 消息
- \* 消息



image-20251005143159018

消息

- 消息通过 convertAndSend 消息 RoutingKey


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

消息

消息Bean消息

SpringAMQP 消息

- Queue 的构建器是 QueueBuilder
- Exchange 的构建器是 ExchangeBuilder

 image-20251005143848191

- Binding 的构建器是 BindingBuilder

下面是一个完整的例子

- Binding bindingQueue1(Queue fanoutQueue1, FanoutExchange fanoutExchange) 的 Bean 在 Spring 中

```
@Configuration
public class FanoutConfig {

    /**
     * 配置
     * @return FanoutExchange
     */
    @Bean
    public FanoutExchange fanoutExchange(){
        return new FanoutExchange("hmall.fanout");
    }

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

    /**
     * 配置2
     */
    @Bean
    public Binding bindingQueue1(Queue fanoutQueue1, FanoutExchange fanoutExchange){
        return BindingBuilder.bind(fanoutQueue1).to(fanoutExchange);
    }

    /**
     * 配置3
     */
    @Bean
    public Queue fanoutQueue2(){
        return new Queue("fanout.queue2");
    }

    /**
     * 配置4
     */
    @Bean
    public Binding bindingQueue2(Queue fanoutQueue2, FanoutExchange fanoutExchange){
```

```

        return BindingBuilder.bind(fanoutQueue2).to(fanoutExchange);
    }
}

```

队列绑定

SpringAMQP 队列绑定 @RabbitListener 队列绑定

- 队列 bindings 队列绑定
  - 队列 values 队列绑定
  - 队列 exchange 队列绑定
    - 队列 name 队列绑定 type 队列绑定
  - 队列 key 队列 bindingKeys 队列 List 队列

```

@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("队列1队列direct.queue1队列" + msg + "队列");
}

```

队列

队列JDK队列绑定

队列JSON队列绑定JDK队列

- 队列 jackson 队列

```

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

```

- 队列 MessageConvertor

```

@Bean
public MessageConverter messageConverter(){
    // 1.队列绑定
    Jackson2JsonMessageConverter jackson2JsonMessageConverter = new
    Jackson2JsonMessageConverter();
    // 2.队列绑定id队列绑定ID队列绑定
    jackson2JsonMessageConverter.setCreateMessageIds(true);
    return jackson2JsonMessageConverter;
}

```

队列

配置

配置MQ连接和重试策略

- 配置MQ连接和重试策略

```
spring:
  rabbitmq:
    connection-timeout: 1s # MQ连接超时
    template:
      retry:
        enabled: true # 是否启用重试
        initial-interval: 1000ms # 初始重试间隔
        multiplier: 1 # 重试间隔倍数 = initial-interval * multiplier
        max-attempts: 3 # 最大重试次数
```

配置

- ConfirmCallback 回调方法 **Exchange**
  - 回调Exchange的ACK/NACK
- ReturnsCallback 回调方法 **Exchange** 和 **Queue**
  - 回调Exchange和Queue的ACK/NACK

SpringAMQP 支持\*\* Publisher Confirm 和 Publisher Return \*\*回调方法，用于处理MQ消息的ACK/NACK。

- 配置MQ连接和重试策略 PublisherReturn 回调方法 ACK 回调
- 配置Exchange 回调方法 PublisherConfirm 回调 ACK 回调
- 配置Exchange 回调方法 PublisherConfirm 回调 ACK 回调
- 配置 NACK 回调

配置

- 配置 yml 配置
  - publisher-confirm-type 配置
    - none 不确认
    - simple 简单MQ
    - correlated MQ消息相关

```
spring:
  rabbitmq:
    publisher-confirm-type: correlated # 配置publisher confirm回调
    publisher-returns: true # 配置publisher return
```

- 配置 RabbitTemplate 回调 ReturnCallback

```
@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("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());
        }
    });
}

```

- 消息ID ConfirmCallback
  - Future 消息ID
  - CorrelationData 消息ID

```

@Test
void testPublisherConfirm() {
    // 1. CorrelationData
    CorrelationData cd = new CorrelationData();
    // 2. Future ConfirmCallback
    cd.getFuture().addCallback(new ListenableFutureCallback<CorrelationData.Confirm>
() {
        @Override
        public void onFailure(Throwable ex) {
            // 2.1. Future 消息ID
            log.error("send message fail", ex);
        }
        @Override
        public void onSuccess(CorrelationData.Confirm result) {
            // 2.2. Future 消息ID result
            if(result.isAck()){ // result.isAck() boolean true ack false
                log.debug("ack!");
            }else{ // result.getReason() String nack
                log.error("nack, reason : {}", result.getReason());
            }
        }
    });
    // 3.
    rabbitTemplate.convertAndSend("hmall.direct", "q", "hello", cd);
}

```

消息ID

RabbitMQ 消息ID 3 消息ID

- 消息体
- 消息头
- 消息属性
  - 消息属性名称

消息体

- 消息体是消息的载体，消息体是MQ的
- 消息体

## Lazy Queue

消息体

- 消息体是消息的载体，消息体是MQ的
  - 消息体是消息的载体，消息体是MQ的
- 消息体是消息的载体，消息体是MQ的2048

3.12消息体Lazy Queue消息体

消息体 x-queue-mode 消息lazy

- 消息Bean

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

- 消息体

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

image-20251006144125410

消息体

消息体

消息体 RabbitMQ消息体 RabbitMQ消息体

- ack消息体 RabbitMQ消息体

- **nack** 通知 RabbitMQ 消息失败
- **reject** 通知 RabbitMQ 消息失败
  - 消息会被重新入队

SpringAMQP 配置 ACK 模式

- **none** 需要手动调用 ack 方法从 MQ 中移除消息
- **\*\* manual \*\*** 需要手动调用 api 中的 ack 或 reject 方法
- **auto** 默认 SpringAMQP 的 AOP 会自动调用 ack 方法从 MQ 中移除消息
  - 消息失败会 nack
  - 消息失败会 reject

```
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: none # 手动
```

配置重试

SpringAMQP 配置消息失败后重新入队 MQ

配置 yaml 配置

```
spring:
  rabbitmq:
    listener:
      simple:
        retry:
          enabled: true # 是否启用
          initial-interval: 1000ms # 初始间隔1s
          multiplier: 1 # 每次重试间隔 = multiplier * last-interval
          max-attempts: 3 # 最大重试次数
          stateless: true # true 每次重试都会重新入队 false 只重试一次
```

配置消息失败后重新入队 MQ 需要配置 MessageRecoverer

- **RejectAndDontRequeueRecoverer** 消息失败后 reject 消息
- **ImmediateRequeueMessageRecoverer** 消息失败后 nack 消息
- **RepublishMessageRecoverer** 消息失败后重新入队

配置 Bean

- 配置 MessageRecoverer Bean
- 配置 RepublishMessageRecoverer Bean

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

$f(x)=f(f(x))$




- 0000000000000000id000000000000
- 0000000000000000000000000000ID00000000
- 00000000000000000000000000000000000000

```
@Bean
public MessageConverter messageConverter(){
    // 1.初始化
    Jackson2JsonMessageConverter jjmc = new Jackson2JsonMessageConverter();
    // 2.设置id属性
    jjmc.setCreateMessageIds(true);
    return jjmc;
}
```

- `getBody()` 返回消息内容
- `getMessageProperties().getMessageId()` 返回消息ID

[illegible]

- `basic.reject` `basic.nack` `requeue` `false`
- `basic.reject` `basic.nack` `requeue` `false`
- `basic.reject` `basic.nack` `requeue` `false`

- 
- 
- 

[illegible]

image-20251006155739041

使用RabbitMQ

使用 @Exchange 设置 delayed 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("delay.queue接收消息{}", msg);
}
```

Bean 设置 .delayed() 为 true

```
@Bean
public DirectExchange delayExchange(){
    return ExchangeBuilder
        .directExchange("delay.direct") // 设置名称
        .delayed() // 设置 delayed 为 true
        .durable(true) // 持久化
        .build();
}
```

使用 x-delay 设置延迟

- 使用 getMessageProperties() 获取消息属性，使用 setDelay() 设置延迟

```
// 2. 使用 x-delay 设置延迟
rabbitTemplate.convertAndSend("delay.direct", "delay", message, new
MessagePostProcessor() {
    @Override
    public Message postProcessMessage(Message message) throws AmqpException {
        // 设置延迟
        message.getMessageProperties().setDelay(5000);
        return message;
    }
});
```