服务异步通讯

高级篇-rabbitmq的高级特性

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

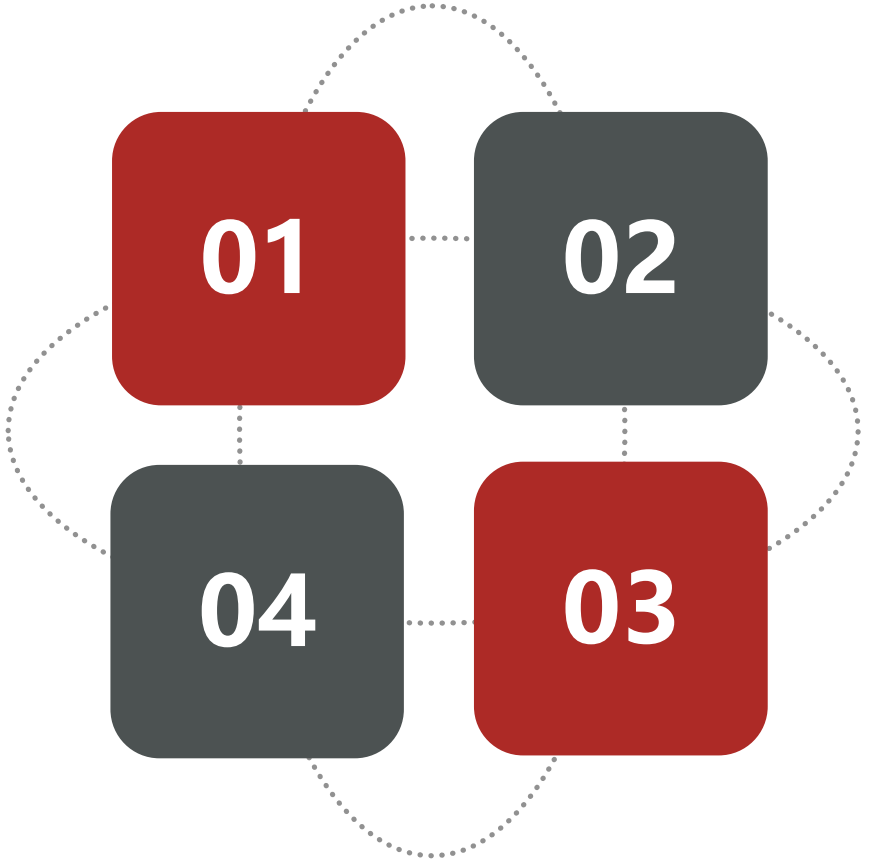
MQ的一些常见问题

消息可靠性问题

如何确保发送的消息至少被消费一次

延迟消息问题

如何实现消息的延迟投递



高可用问题

如何避免单点的MQ故障而导致的不可用问题

消息堆积问题

如何解决数百万消息堆积，无法及时消费的问题

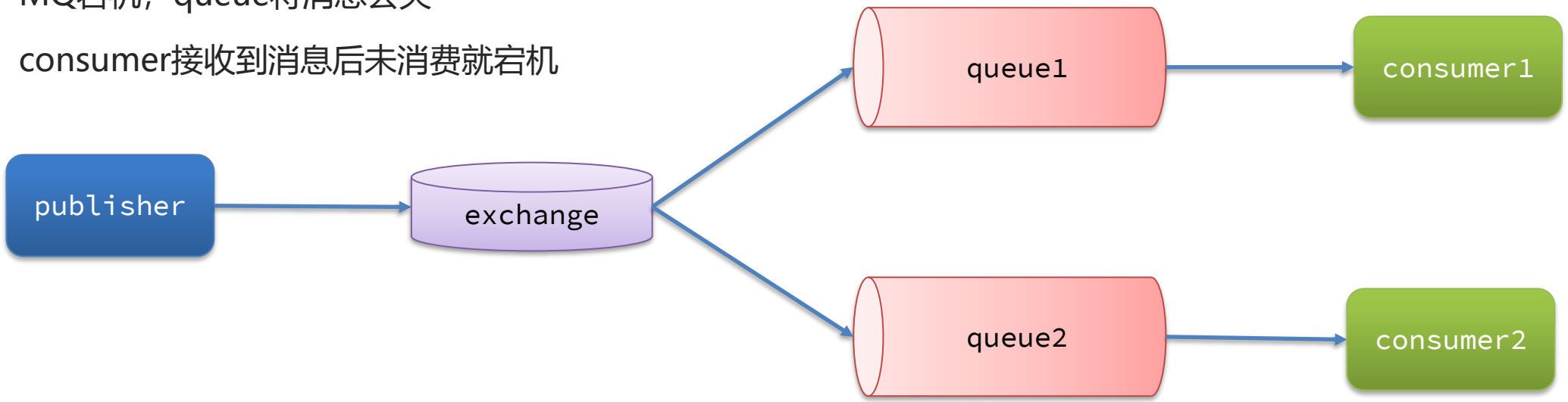
目录

Contents

消息可靠性问题

消息从生产者发送到exchange，再到queue，再到消费者，有哪些导致消息丢失的可能性?

- 发送时丢失:
  - 生产者发送的消息未送达exchange
  - 消息到达exchange后未到达queue
- MQ宕机，queue将消息丢失
- consumer接收到消息后未消费就宕机



消息可靠性

- 生产者消息确认
- 消息持久化
- 消费者消息确认
- 消费失败重试机制

目录

Contents

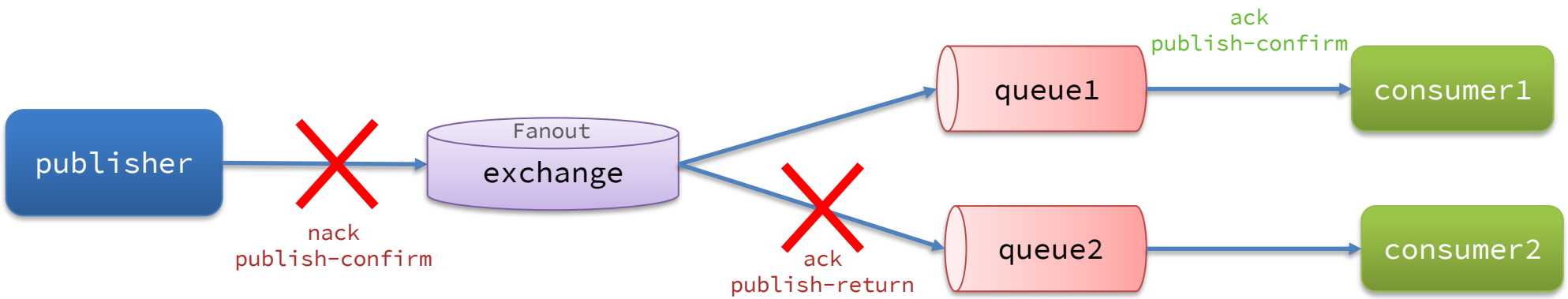
生产者确认机制

RabbitMQ提供了publisher confirm机制来避免消息发送到MQ过程中丢失。消息发送到MQ以后，会返回一个结果给发送者，表示消息是否处理成功。结果有两种请求：

- publisher-confirm，发送者确认
  - 消息成功投递到交换机，返回ack
  - 消息未投递到交换机，返回nack
- publisher-return，发送者回执
  - 消息投递到交换机了，但是没有路由到队列。返回ACK，及路由失败原因。

注意

确认机制发送消息时，需要给每个消息设置一个全局唯一id，以区分不同消息，避免ack冲突



引入Demo工程

首先，我们需要引入课前资料提供的mq-advanced-demo工程：

mq-advanced-demo

SpringAMQP实现生产者确认

1. 在publisher这个微服务的application.yml中添加配置：

```yaml
spring:
  rabbitmq:
    publisher-confirm-type: correlated
    publisher-returns: true
    template:
      mandatory: true
```

配置说明：

- publish-confirm-type：开启publisher-confirm，这里支持两种类型：
  - simple：同步等待confirm结果，直到超时
  - correlated：异步回调，定义ConfirmCallback，MQ返回结果时会回调这个ConfirmCallback
- publish-returns：开启publish-return功能，同样是基于callback机制，不过是定义ReturnCallback
- template.mandatory：定义消息路由失败时的策略。true，则调用ReturnCallback；false：则直接丢弃消息

SpringAMQP实现生产者确认

2. 每个RabbitTemplate只能配置一个ReturnCallback，因此需要在项目启动过程中配置：

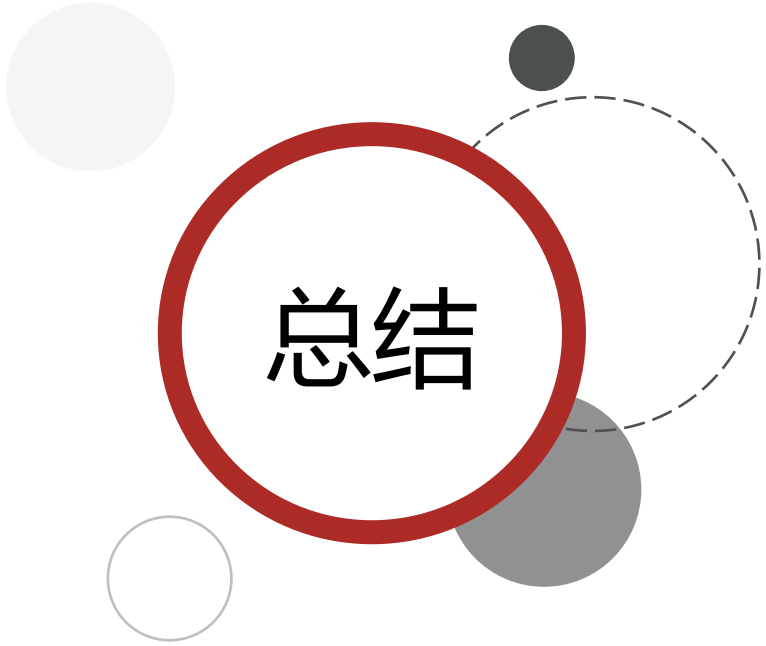
```
@Slf4j
@Configuration
public class CommonConfig implements ApplicationContextAware {

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        // 获取RabbitTemplate
        RabbitTemplate rabbitTemplate = applicationContext.getBean(RabbitTemplate.class);
        // 设置ReturnCallback
        rabbitTemplate.setReturnCallback((message, replyCode, replyText, exchange, routingKey) -> {
            log.info("消息发送失败，应答码{}，原因{}，交换机{}，路由键{},消息{}",
                    replyCode, replyText, exchange, routingKey, message.toString());
        });
    }
}
```

SpringAMQP实现生产者确认

3. 发送消息，指定消息ID、消息ConfirmCallback

```
@Test
public void testSendMessage2SimpleQueue() throws InterruptedException {
    // 消息体
    String message = "hello, spring amqp!";
    // 消息ID，需要封装到CorrelationData中
    CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
    // 添加callback
    correlationData.getFuture().addCallback(
            result -> {
                if(result.isAck()){
                    // ack，消息成功
                    log.debug("消息发送成功，ID:{}", correlationData.getId());
                }else{
                    // nack，消息失败
                    log.error("消息发送失败，ID:{}，原因{}",correlationData.getId(), result.getReason());
                }
            },
            ex -> log.error("消息发送异常，ID:{}，原因{}",correlationData.getId(),ex.getMessage())
    );
    // 发送消息
    rabbitTemplate.convertAndSend("amq.direct", "simple", message, correlationData);
}
```



SpringAMQP中处理消息确认的几种情况：

- publisher-comfirm：
  - 消息成功发送到exchange，返回ack
  - 消息发送失败，没有到达交换机，返回nack
  - 消息发送过程中出现异常，没有收到回执
- 消息成功发送到exchange，但没有路由到queue，

  调用ReturnCallback

目录

Contents

消息持久化

MQ默认是内存存储消息，开启持久化功能可以确保缓存在MQ中的消息不丢失。

1. 交换机持久化：

```
@Bean
public DirectExchange simpleExchange(){
    // 三个参数：交换机名称、是否持久化、当没有queue与其绑定时是否自动删除
    return new DirectExchange("simple.direct", true, false);
}
```

2. 队列持久化：

```
@Bean
public Queue simpleQueue(){
    // 使用QueueBuilder构建队列，durable就是持久化的
    return QueueBuilder.durable("simple.queue").build();
}
```

3. 消息持久化，SpringAMQP中的的消息默认是持久的，可以通过MessageProperties中的DeliveryMode来指定

```
Message msg = MessageBuilder
        .withBody(message.getBytes(StandardCharsets.UTF_8)) // 消息体
        .setDeliveryMode(MessageDeliveryMode.PERSISTENT) // 持久化
        .build();
```

目录

Contents

消费者确认

RabbitMQ支持消费者确认机制，即：消费者处理消息后可以向MQ发送ack回执，MQ收到ack回执后才会删除该消息。而SpringAMQP则允许配置三种确认模式：

- manual：手动ack，需要在业务代码结束后，调用api发送ack。
- auto：自动ack，由spring监测listener代码是否出现异常，没有异常则返回ack；抛出异常则返回nack
- none：关闭ack，MQ假定消费者获取消息后会成功处理，因此消息投递后立即被删除

配置方式是修改application.yml文件，添加下面配置：

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        prefetch: 1
        acknowledge-mode: none # none，关闭ack；manual，手动ack；auto：自动ack
```

目录

Contents

消费者失败重试

当消费者出现异常后，消息会不断requeue（重新入队）到队列，再重新发送给消费者，然后再次异常，再次requeue，无限循环，导致mq的消息处理飙升，带来不必要的压力：

Overview				Messages			Message rates			+/-
Name	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
simple.queue	classic	D	running	0	1	1	0.00/s	3,370/s	0.00/s	

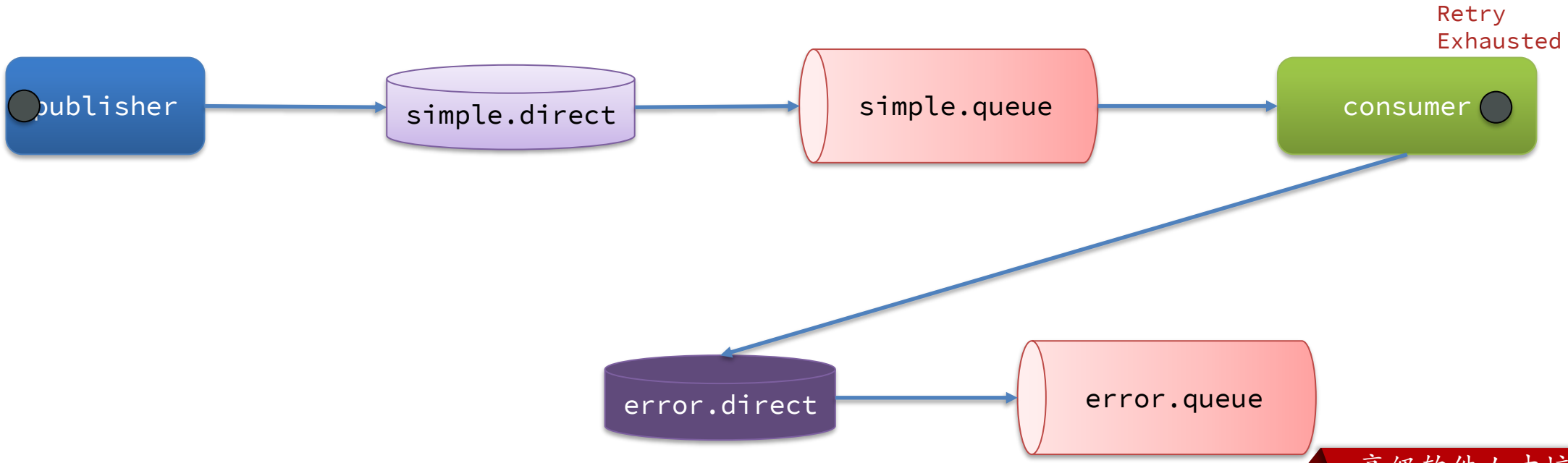
我们可以利用Spring的retry机制，在消费者出现异常时利用本地重试，而不是无限制的requeue到mq队列。

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        prefetch: 1
        retry:
          enabled: true # 开启消费者失败重试
          initial-interval: 1000 # 初始的失败等待时长为1秒
          multiplier: 1 # 下次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
```

消费者失败消息处理策略

在开启重试模式后，重试次数耗尽，如果消息依然失败，则需要有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。默认就是这种方式
- 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



消费者失败消息处理策略

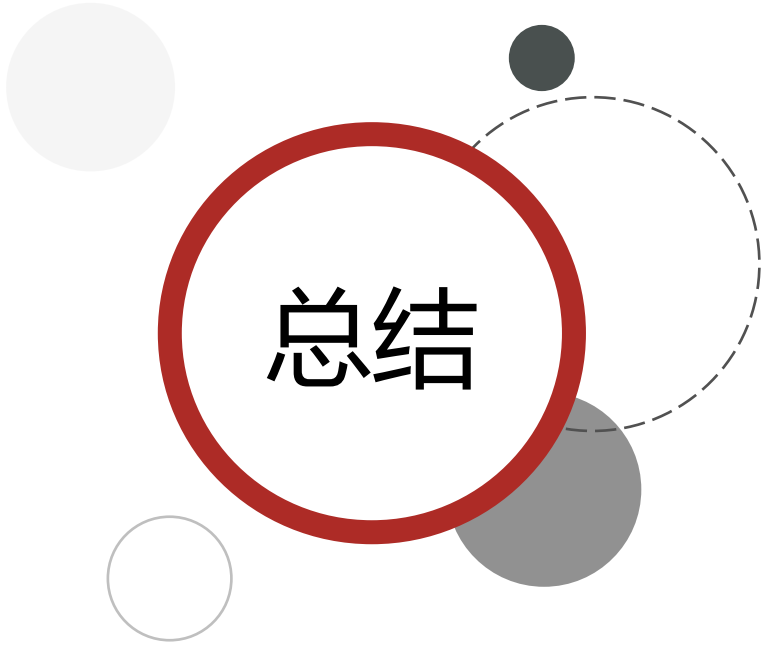
测试下RepublishMessageRecoverer处理模式：

- 首先，定义接收失败消息的交换机、队列及其绑定关系：

```
@Bean
public DirectExchange errorMessageExchange(){
    return new DirectExchange("error.direct");
}
@Bean
public Queue errorQueue(){
    return new Queue("error.queue", true);
}
@Bean
public Binding errorBinding(){
    return BindingBuilder.bind(errorQueue()).to(errorMessageExchange()).with("error");
}
```

- 然后，定义RepublishMessageRecoverer：

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



如何确保RabbitMQ消息的可靠性?

- 开启生产者确认机制，确保生产者的消息能到达队列
- 开启持久化功能，确保消息未消费前在队列中不会丢失
- 开启消费者确认机制为auto，由spring确认消息处理成功后完成ack
- 开启消费者失败重试机制，并设置MessageRecoverer，多次重试失败后将消息投递到异常交换机，交由人工处理

死信交换机

- 初识死信交换机
- TTL
- 延迟队列

目录

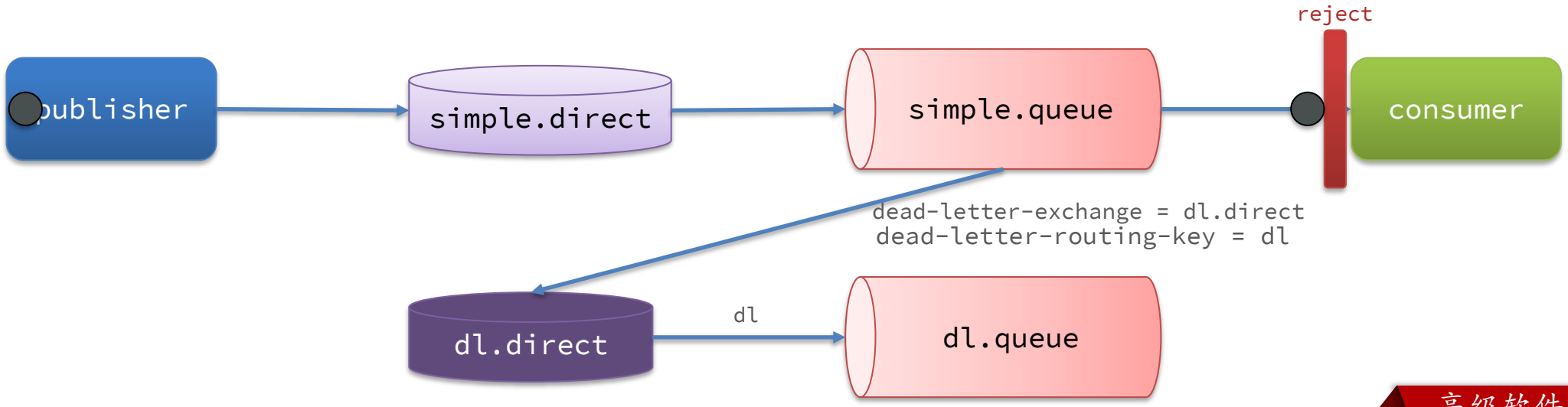
Contents

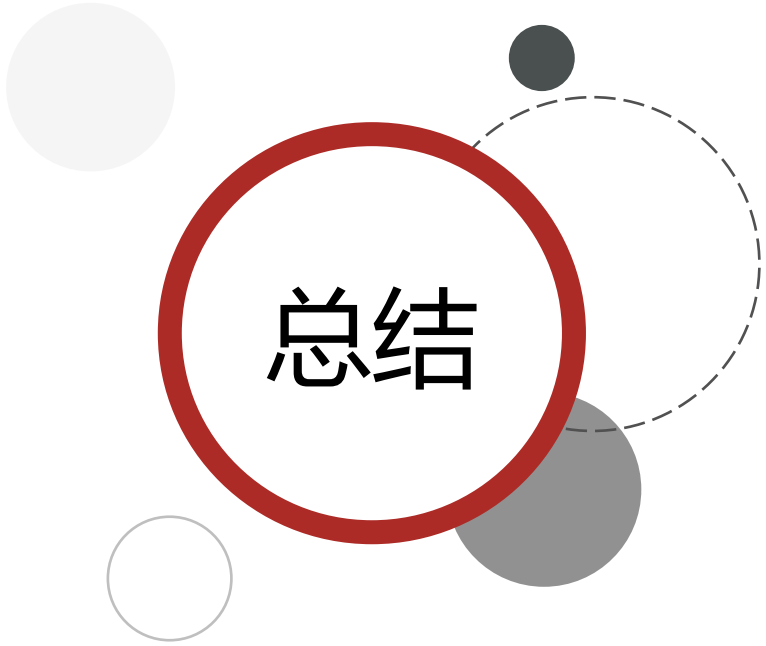
初识死信交换机

当一个队列中的消息满足下列情况之一时，可以成为死信（dead letter）：

- 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
- 要投递的队列消息堆积满了，最早的消息可能成为死信

如果该队列配置了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，而这个交换机称为死信交换机（Dead Letter Exchange，简称DLX）。





什么样的消息会成为死信?

- 消息被消费者reject或者返回nack
- 消息超时未消费
- 队列满了

如何给队列绑定死信交换机?

- 给队列设置dead-letter-exchange属性，指定一个交换机
- 给队列设置dead-letter-routing-key属性，设置死信交换机与死信队列的RoutingKey

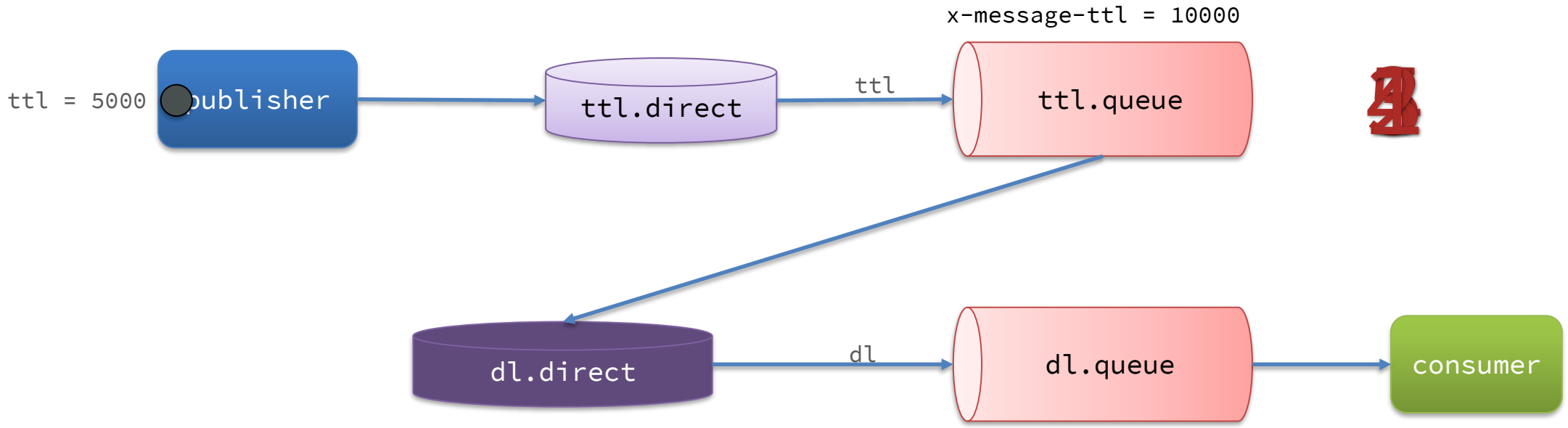
目录

Contents

TTL

TTL，也就是Time-To-Live。如果一个队列中的消息TTL结束仍未消费，则会变为死信，ttl超时分为两种情况：

- 消息所在的队列设置了存活时间
- 消息本身设置了存活时间



TTL

我们声明一组死信交换机和队列，基于注解方式：

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

TTL

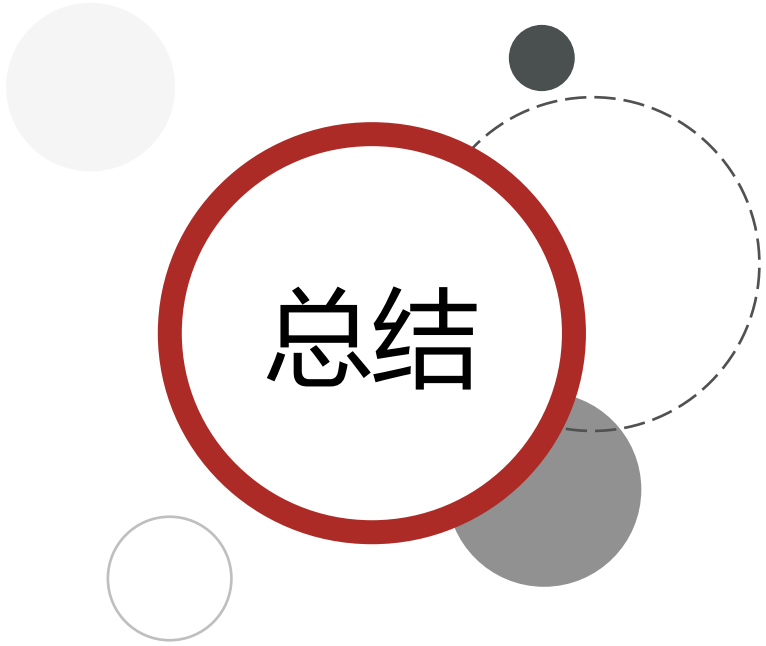
要给队列设置超时时间，需要在声明队列时配置x-message-ttl属性：

```
@Bean
public DirectExchange ttlExchange(){
    return new DirectExchange("ttl.direct");
}
@Bean
public Queue ttlQueue(){
    return QueueBuilder.durable("ttl.queue") // 指定队列名称，并持久化
            .ttl(10000) // 设置队列的超时时间，10秒
            .deadLetterExchange("dl.direct") // 指定死信交换机
            .deadLetterRoutingKey("dl") // 指定死信RoutingKey
            .build();
}
@Bean
public Binding simpleBinding(){
    return BindingBuilder.bind(ttlQueue()).to(ttlExchange()).with("ttl");
}
```

TTL

发送消息时，给消息本身设置超时时间

```
@Test
public void testTTLMsg() {
    // 创建消息
    Message message = MessageBuilder
            .withBody("hello, ttl message".getBytes(StandardCharsets.UTF_8))
            .setExpiration("5000")
            .build();
    // 消息ID，需要封装到CorrelationData中
    CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
    // 发送消息
    rabbitTemplate.convertAndSend("ttl.direct", "ttl", message, correlationData);
}
```



消息超时的两种方式是？

- 给队列设置ttl属性，进入队列后超过ttl时间的消息变为死信
- 给消息设置ttl属性，队列接收到消息超过ttl时间后变为死信
- 两者共存时，以时间短的ttl为准

如何实现发送一个消息20秒后消费者才收到消息？

- 给消息的目标队列指定死信交换机
- 消费者监听与死信交换机绑定的队列
- 发送消息时给消息设置ttl为20秒

目录

Contents

延迟队列

利用TTL结合死信交换机，我们实现了消息发出后，消费者延迟收到消息的效果。这种消息模式就称为延迟队列（Delay Queue）模式。

延迟队列的使用场景包括：

- 延迟发送短信
- 用户下单，如果用户在15 分钟内未支付，则自动取消
- 预约工作会议，20分钟后自动通知所有参会人员

延迟队列插件

因为延迟队列的需求非常多，所以RabbitMQ的官方也推出了一个插件，原生支持延迟队列效果。

详细安装过程参考课前资料文档《RabbitMQ部署指南.md》中的第2节《安装DelayExchange插件》：

SpringAMQP使用延迟队列插件

DelayExchange的本质还是官方的三种交换机，只是添加了延迟功能。因此使用时只需要声明一个交换机，交换机的类型可以是任意类型，然后设定delayed属性为true即可。

基于注解方式：

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

SpringAMQP使用延迟队列插件

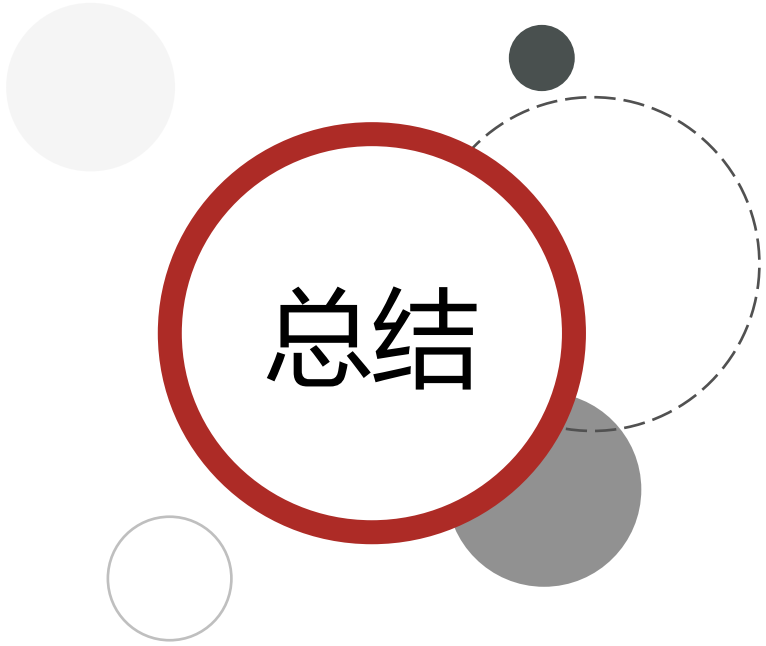
基于java代码的方式：

```
@Bean
public DirectExchange delayedExchange(){
    return ExchangeBuilder
            .directExchange("delay.direct") // 指定交换机类型和名称
            .delayed() // 设置delay属性为 true
            .durable(true) // 持久化
            .build();
}
@Bean
public Queue delayedQueue(){
    return new Queue( name: "delay.queue");
}
@Bean
public Binding delayedBinding(){
    return BindingBuilder.bind(delayedQueue()).to(delayedExchange()).with( routingKey: "delay");
}
```

SpringAMQP使用延迟队列插件

然后我们向这个delay为true的交换机中发送消息，一定要给消息添加一个header：x-delay，值为延迟的时间，单位为毫秒：

```
@Test
public void testDelayedMsg() {
    // 创建消息
    Message message = MessageBuilder
            .withBody("hello, delayed message".getBytes(StandardCharsets.UTF_8))
            .setHeader("x-delay", 10000)
            .build();
    // 消息ID，需要封装到CorrelationData中
    CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
    // 发送消息
    rabbitTemplate.convertAndSend( exchange: "delay.direct",  routingKey: "delay", message, correlationData);
    log.debug("发送消息成功");
}
```



延迟队列插件的使用步骤包括哪些?

- 声明一个交换机，添加delayed属性为true
- 发送消息时，添加x-delay头，值为超时时间

惰性队列

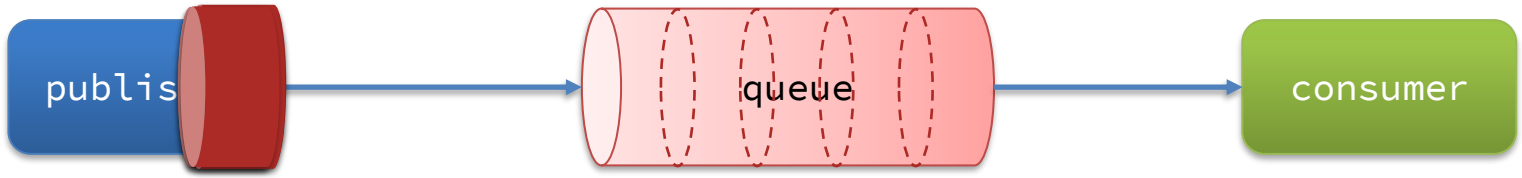
- 消息堆积问题
- 惰性队列

目录

Contents

消息堆积问题

当生产者发送消息的速度超过了消费者处理消息的速度，就会导致队列中的消息堆积，直到队列存储消息达到上限。最早接收到的消息，可能就会成为死信，会被丢弃，这就是消息堆积问题。



解决消息堆积有三种种思路：

- 增加更多消费者，提高消费速度
- 在消费者内开启线程池加快消息处理速度
- 扩大队列容积，提高堆积上限

目录

Contents

惰性队列

从RabbitMQ的3.6.0版本开始，就增加了Lazy Queues的概念，也就是惰性队列。

惰性队列的特征如下：

- 接收到消息后直接存入磁盘而非内存
- 消费者要消费消息时才会从磁盘中读取并加载到内存
- 支持数百万条的消息存储

而要设置一个队列为惰性队列，只需要在声明队列时，指定x-queue-mode属性为lazy即可。可以通过命令行将一个运行中的队列修改为惰性队列：

```
rabbitmqctl set_policy Lazy "^lazy-queue$" '{"queue-mode":"lazy"}' --apply-to queues
```

惰性队列

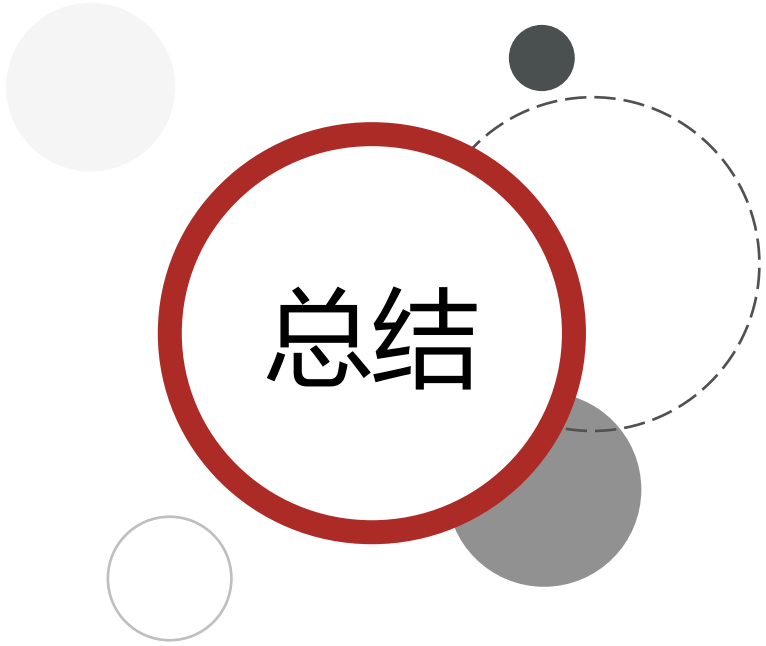
用SpringAMQP声明惰性队列分两种方式：

- @Bean的方式：

```
@Bean
public Queue lazyQueue(){
    return QueueBuilder
            .durable("lazy.queue")
            .lazy() // 开启x-queue-mode为lazy
            .build();
}
```

- 注解方式：

```
@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);
}
```



消息堆积问题的解决方案?

- 队列上绑定多个消费者，提高消费速度
- 给消费者开启线程池，提高消费速度
- 使用惰性队列，可以再mq中保存更多消息

惰性队列的优点有哪些?

- 基于磁盘存储，消息上限高
- 没有间歇性的page-out，性能比较稳定

惰性队列的缺点有哪些?

- 基于磁盘存储，消息时效性会降低
- 性能受限于磁盘的IO

MQ集群

- 集群分类
- 普通集群
- 镜像集群
- 仲裁队列

目录

Contents

集群分类

RabbitMQ的是基于Erlang语言编写，而Erlang又是一个面向并发的语言，天然支持集群模式。RabbitMQ的集群有两种模式：

- **普通集群**：是一种分布式集群，将队列分散到集群的各个节点，从而提高整个集群的并发能力。
- **镜像集群**：是一种主从集群，普通集群的基础上，添加了主从备份功能，提高集群的数据可用性。

镜像集群虽然支持主从，但主从同步并不是强一致的，某些情况下可能有数据丢失的风险。因此在RabbitMQ的3.8版本以后，推出了新的功能：**仲裁队列**来代替镜像集群，底层采用Raft协议确保主从的数据一致性。

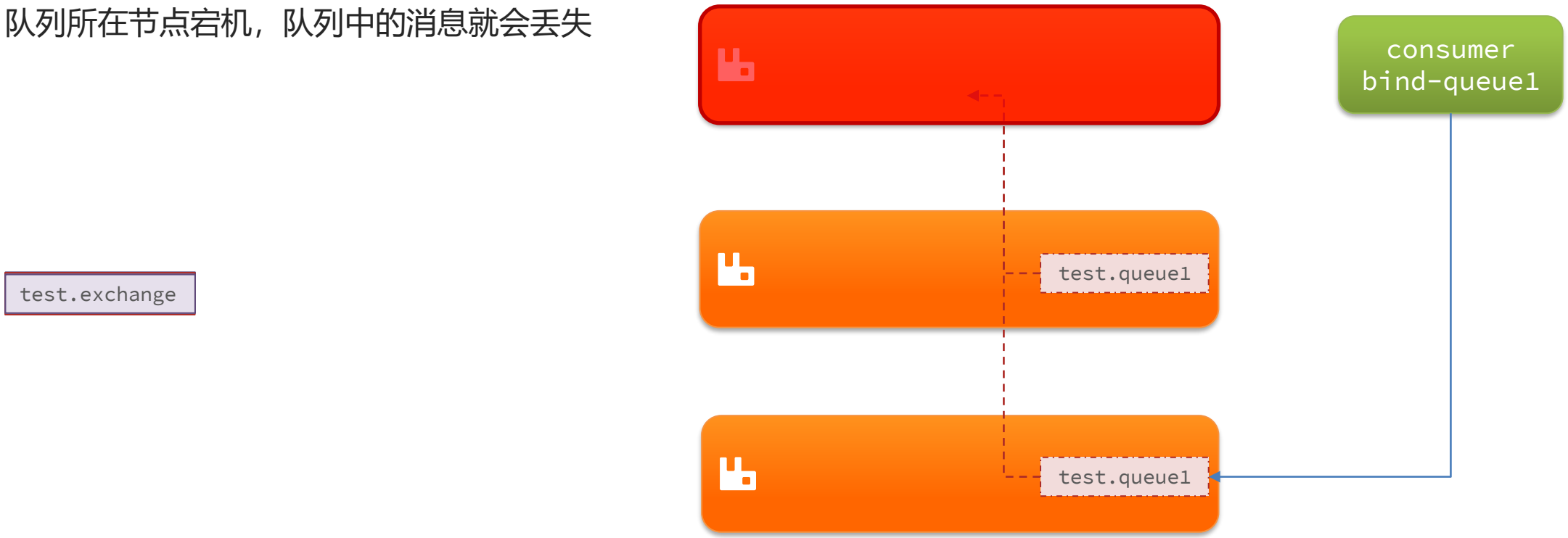
目录

Contents

普通集群

普通集群，或者叫标准集群（classic cluster），具备下列特征：

- 会在集群的各个节点间共享部分数据，包括：交换机、队列元信息。不包含队列中的消息。
- 当访问集群某节点时，如果队列不在该节点，会从数据所在节点传递到当前节点并返回
- 队列所在节点宕机，队列中的消息就会丢失



普通集群

详细的搭建步骤可以参考课前资料：

RabbitMQ部署指南.md

目录

Contents

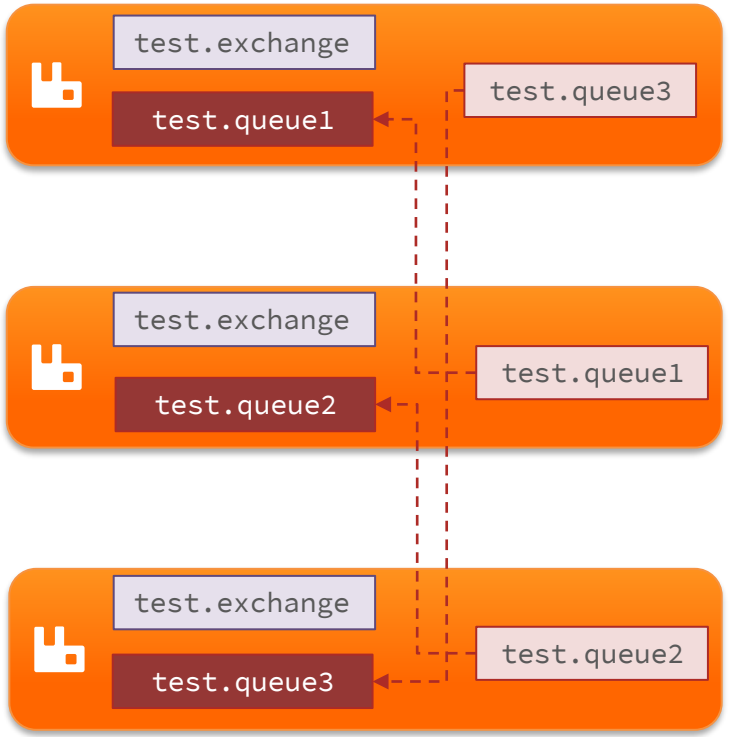
镜像集群

镜像集群：本质是主从模式，具备下面的特征：

- 交换机、队列、队列中的消息会在各个mq的镜像节点之间同步备份。
- 创建队列的节点被称为该队列的**主节点**，备份到的其它节点叫做该队列的**镜像**节点。
- 一个队列的主节点可能是另一个队列的镜像节点
- 所有操作都是主节点完成，然后同步给镜像节点
- 主宕机后，镜像节点会替代成新的主

详细的搭建步骤可以参考课前资料：

RabbitMQ部署指南.md



目录

Contents

仲裁队列

仲裁队列：仲裁队列是3.8版本以后才有的新功能，用来替代镜像队列，具备下列特征：

- 与镜像队列一样，都是主从模式，支持主从数据同步
- 使用非常简单，没有复杂的配置
- 主从同步基于Raft协议，强一致

详细的搭建步骤可以参考课前资料：

RabbitMQ部署指南.md

仲裁队列

SpringAMQP创建仲裁队列：

```
@Bean
public Queue quorumQueue() {
    return QueueBuilder
            .durable("quorum.queue") // 持久化
            .quorum() // 仲裁队列
            .build();
}
```

仲裁队列

SpringAMQP连接集群，只需要在yaml中配置即可：

```yaml
spring:
  rabbitmq:
    addresses: 192.168.150.105:8071, 192.168.150.105:8072, 192.168.150.105:8073
    username: itcast
    password: 123321
    virtual-host: /
```

传智教育旗下高端IT教育品牌