分布式消息服务 RabbitMQ 版

最佳实践

文档版本	01
发布日期	2024-09-03

华为云计算技术有限公司

商标声明

HUAWEI和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 实现网络异常时 RabbitMQ 客户端自动恢复

方案概述

由于服务端重启、网络抖动等原因造成客户端网络连接断开时，将导致客户端无法正常生产和消费消息。

通过在客户端侧设置重连机制，使客户端在网络连接断开时自动恢复连接，降低网络故障对业务的影响。以下场景会触发网络自动恢复：

- 在连接的I/O循环中抛出未处理的异常
- 检测到Socket读取超时
- 检测到服务端心跳丢失

说明

- 4.0.0及以上版本的Java客户端默认支持网络自动恢复，无需设置。
- 如果应用程序使用**Connection.Close**方法关闭连接，则不会启用或触发网络自动恢复。

网络异常时 RabbitMQ 客户端重试连接示例代码

客户端和服务端的初始连接失败，不会触发自动恢复，可在客户端编写对应的应用程序代码，通过重试连接来解决初始连接失败的问题。

以下示例演示了使用Java客户端通过重试连接解决初始连接失败的问题。

```
ConnectionFactory factory = new ConnectionFactory();
// 对于4.0.0版本之前的RabbitMQ Java客户端，开启自动恢复功能
factory.setAutomaticRecoveryEnabled(true);

// 配置连接设置
try {
  Connection conn = factory.newConnection();
} catch (java.net.ConnectException e) {
  Thread.sleep(5000);
  // apply retry logic
}
```

2 实现 RabbitMQ 节点重启后消费者自动重连

方案概述

RabbitMQ的amqp-client虽然自带重连机制，但是自带的重连机制只会重试一次，重连失败后就不再执行。这时如果消费者没有做额外的重试机制，那么这个消费者就彻底断开与服务端的连接，无法消费消息。

amqp-client在节点断连后，根据与通道建立的节点不同，产生不同的错误。

- 如果通道连接的是队列所在的节点，消费者就会收到一个shutdown信号。这时amqp-client的重连机制就会生效，尝试重新连接服务端。如果连接成功，这个通道就会继续连接消费。如果连接失败，就会执行**channel.close**方法，关闭这个通道。
- 如果通道连接的不是队列所在的节点，消费者不会触发关闭动作，而是由服务端发送的一个取消动作。这个动作对amqp-client来说并不是异常行为，所以日志上不会有明显的报错，但是连接最终还是会关闭。

amqp-client出现上面两种错误时，会分别回调**handleShutdownSignal**以及**handleCancel**方法。您可以通过重写这两种方法，在回调时执行重写的重连逻辑，就能在通道关闭后重新为消费者创建新的通道继续消费。

RabbitMQ 节点重启后消费者自动重连示例代码

以下提供一个简单的Java代码示例，此示例能够解决上面的两种错误，实现消费者的持续消费。

```
package rabbitmq;

import com.rabbitmq.client.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

public class RabbitConsumer {

    public static void main(String... args) throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        // 配置实例的连接地址和端口
        factory.setHost("192.168.0.2");
        factory.setPort(5672);

        // 配置实例连接的用户名和密码
        factory.setUsername("name");
        factory.setPassword("password");
```

```
        Connection connection = factory.newConnection();

        createNewConnection(connection);
    }

    // 重连处理
    public static void createNewConnection(Connection connection) {
        try {
            Thread.sleep(1000);
            Channel channel = connection.createChannel();
            channel.basicQos(64);
            channel.basicConsume("queue-01", false, new CustomConsumer(channel, connection));
        } catch (Exception e) {
//              e.printStackTrace();
            createNewConnection(connection);
        }
    }

    static class CustomConsumer implements Consumer {

        private final Channel _channel;
        private final Connection _connection;

        public CustomConsumer(Channel channel, Connection connection) {
            _channel = channel;
            _connection = connection;
        }

        @Override
        public void handleConsumeOk(String consumerTag) {
        }

        @Override
        public void handleCancelOk(String consumerTag) {

        }

        @Override
        public void handleCancel(String consumerTag) throws IOException {
            System.out.println("handleCancel");
            System.out.println(consumerTag);
            createNewConnection(_connection);
        }

        @Override
        public void handleShutdownSignal(String consumerTag, ShutdownSignalException sig) {
            System.out.println("handleShutdownSignal");
            System.out.println(consumerTag);
            System.out.println(sig.getReason());
            createNewConnection(_connection);
        }

        @Override
        public void handleRecoverOk(String consumerTag) {

        }

        @Override
        public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,
byte[] body) throws IOException {
            String message = new String(body, StandardCharsets.UTF_8);
            System.out.println(" [x] Received '" + message + "'");
            _channel.basicAck(envelope.getDeliveryTag(), false);
        }

    }
}
```

3 提高 RabbitMQ 性能

本章节基于吞吐量和可靠性两个指标，指导您通过设置队列长度、集群负载均衡、优先队列数量等参数，实现RabbitMQ的高性能。

使用较小的队列长度

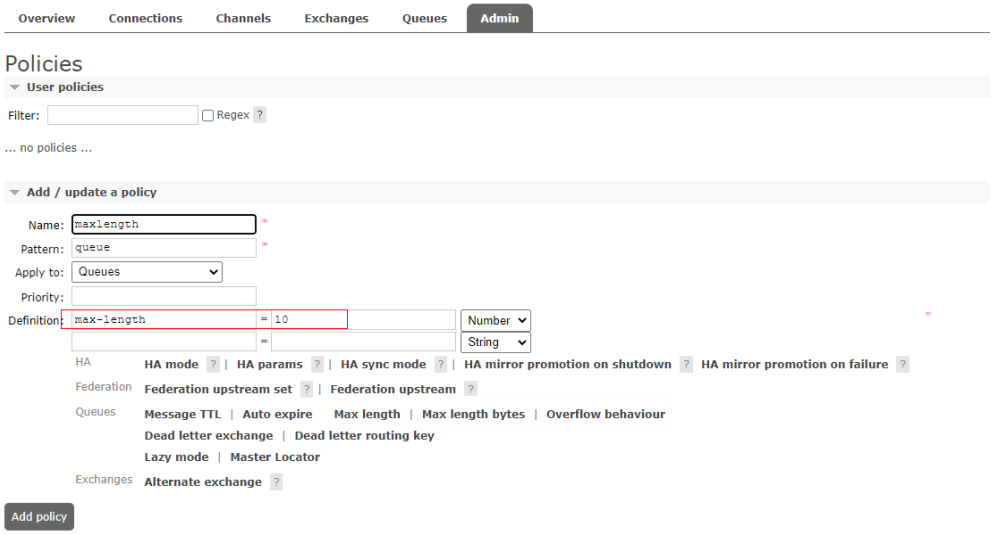
队列中存在大量消息时，会给内存使用带来沉重的负担。为了释放内存，RabbitMQ会将消息刷新到磁盘。刷盘需要重建索引，重启包含大量消息的集群，导致这个过程非常耗时。当刷盘的消息过多时，会阻塞队列处理消息，从而降低队列速度，影响RabbitMQ节点的性能。

要获得最佳性能，应尽可能缩短队列。建议**始终保持队列消息堆积的数量在0左右**。

对于经常受到消息峰值影响的应用程序，和对吞吐量要求较高的应用程序，建议在队列上设置**最大长度**。这样可以通过丢弃队列头部的消息来保持队列长度，队列长度永远不会大于最大长度设置。

最大长度可以通过Policy设置，也可以通过在队列声明时使用对应参数设置。

- 在RabbitMQ WebUI的Policy中设置。



- 在队列声明时使用对应参数设置。

```
//创建队列
HashMap<String, Object> map = new HashMap<>();
```

```
//设置队列最大长度
map.put("x-max-length",10 );
//设置队列溢出方式保留前10
map.put("x-overflow","reject-publish" );
channel.queueDeclare(queueName,false,false,false,map);
```

当队列长度超过设置的最大长度时，RabbitMQ的默认做法是将队列头部的信息（队列中最老的消息）丢弃或变成死信。可以通过设置不同的**overflow**值来改变这种方式，具体如下：

- 如果**overflow**值设置为**drop-head**，表示从队列前面丢弃或dead-letter消息，保存后n条消息。
- 如果**overflow**值设置为**reject-publish**，表示最近发布的消息将被丢弃，即保存前n条消息。

说明

- 如果同时使用以上两种方式设置队列的最大长度，两者中较小的值将被使用。
- 超过队列最大长度的消息会被丢弃，请谨慎使用。

使用集群的负载均衡

队列的性能受单个CPU内核控制，当一个RabbitMQ节点处理消息的能力达到瓶颈时，可以通过集群进行扩展，从而达到提升吞吐量的目的。

使用多个节点，集群会自动将队列均衡的创建在各个节点上。除了使用集群模式，您还可以使用**Consistent hash exchange**插件优化负载均衡。该插件使用交换器来平衡队列之间的消息。根据消息的路由键，发送到交换器的消息一致且均匀地分布在多个队列中。该插件创建路由键的散列，并将消息传播到与该交换器具有绑定关系的队列中。使用此插件时，需要确保消费者从所有队列中消费。使用示例如下：

- 使用不同的路由键来路由消息。

```
public class ConsistentHashExchangeExample1 {
  private static String CONSISTENT_HASH_EXCHANGE_TYPE = "x-consistent-hash";

  public static void main(String[] argv) throws IOException, TimeoutException, InterruptedException {
    ConnectionFactory cf = new ConnectionFactory();
    Connection conn = cf.newConnection();
    Channel ch = conn.createChannel();

    for (String q : Arrays.asList("q1", "q2", "q3", "q4")) {
      ch.queueDeclare(q, true, false, false, null);
      ch.queuePurge(q);
    }

    ch.exchangeDeclare("e1", CONSISTENT_HASH_EXCHANGE_TYPE, true, false, null);

    for (String q : Arrays.asList("q1", "q2")) {
      ch.queueBind(q, "e1", "1");
    }

    for (String q : Arrays.asList("q3", "q4")) {
      ch.queueBind(q, "e1", "2");
    }

    ch.confirmSelect();

    AMQP.BasicProperties.Builder bldr = new AMQP.BasicProperties.Builder();
    for (int i = 0; i < 100000; i++) {
      ch.basicPublish("e1", String.valueOf(i), bldr.build(), "".getBytes("UTF-8"));
    }

    ch.waitForConfirmsOrDie(10000);
```

```
    System.out.println("Done publishing!");
    System.out.println("Evaluating results...");
    // wait for one stats emission interval so that queue counters
    // are up-to-date in the management UI
    Thread.sleep(5);

    System.out.println("Done.");
    conn.close();
  }
}
```

- 通过不同的header来路由消息。

  该方式需要为交换器提供“hash-header”参数设置，且消息必须带有header，否则会被路由到相同的队列。

```
public class ConsistentHashExchangeExample2 {
  public static final String EXCHANGE = "e2";
  private static String EXCHANGE_TYPE = "x-consistent-hash";

  public static void main(String[] argv) throws IOException, TimeoutException, InterruptedException {
    ConnectionFactory cf = new ConnectionFactory();
    Connection conn = cf.newConnection();
    Channel ch = conn.createChannel();

    for (String q : Arrays.asList("q1", "q2", "q3", "q4")) {
      ch.queueDeclare(q, true, false, false, null);
      ch.queuePurge(q);
    }

    Map<String, Object> args = new HashMap<>();
    args.put("hash-header", "hash-on");
    ch.exchangeDeclare(EXCHANGE, EXCHANGE_TYPE, true, false, args);

    for (String q : Arrays.asList("q1", "q2")) {
      ch.queueBind(q, EXCHANGE, "1");
    }

    for (String q : Arrays.asList("q3", "q4")) {
      ch.queueBind(q, EXCHANGE, "2");
    }

    ch.confirmSelect();


    for (int i = 0; i < 100000; i++) {
      AMQP.BasicProperties.Builder bldr = new AMQP.BasicProperties.Builder();
      Map<String, Object> hdrs = new HashMap<>();
      hdrs.put("hash-on", String.valueOf(i));
      ch.basicPublish(EXCHANGE, "", bldr.headers(hdrs).build(), "".getBytes("UTF-8"));
    }

    ch.waitForConfirmsOrDie(10000);

    System.out.println("Done publishing!");
    System.out.println("Evaluating results...");
    // wait for one stats emission interval so that queue counters
    // are up-to-date in the management UI
    Thread.sleep(5);

    System.out.println("Done.");
    conn.close();
  }
}
```

- 使用消息属性来路由消息，例如message_id、correlation_id或timestamp属性。

  该方式需要使用“hash-property”参数来声明交换器，且消息必须带有所选择的消息属性，否则会被路由到相同的队列。

```
public class ConsistentHashExchangeExample3 {
  public static final String EXCHANGE = "e3";
  private static String EXCHANGE_TYPE = "x-consistent-hash";

  public static void main(String[] argv) throws IOException, TimeoutException, InterruptedException {
    ConnectionFactory cf = new ConnectionFactory();
    Connection conn = cf.newConnection();
    Channel ch = conn.createChannel();

    for (String q : Arrays.asList("q1", "q2", "q3", "q4")) {
      ch.queueDeclare(q, true, false, false, null);
      ch.queuePurge(q);
    }

    Map<String, Object> args = new HashMap<>();
    args.put("hash-property", "message_id");
    ch.exchangeDeclare(EXCHANGE, EXCHANGE_TYPE, true, false, args);

    for (String q : Arrays.asList("q1", "q2")) {
      ch.queueBind(q, EXCHANGE, "1");
    }

    for (String q : Arrays.asList("q3", "q4")) {
      ch.queueBind(q, EXCHANGE, "2");
    }

    ch.confirmSelect();


    for (int i = 0; i < 100000; i++) {
      AMQP.BasicProperties.Builder bldr = new AMQP.BasicProperties.Builder();
      ch.basicPublish(EXCHANGE, "", bldr.messageId(String.valueOf(i)).build(), "".getBytes("UTF-8"));
    }

    ch.waitForConfirmsOrDie(10000);

    System.out.println("Done publishing!");
    System.out.println("Evaluating results...");
    // wait for one stats emission interval so that queue counters
    // are up-to-date in the management UI
    Thread.sleep(5);

    System.out.println("Done.");
    conn.close();
  }
}
```

自动删除不再使用的队列

客户端可能连接失败导致队列被残留，大量的残留队列会影响实例的性能。RabbitMQ提供三种自动删除队列的方法：

- 在队列中设置TTL策略：例如TTL策略设置为28天，当持续28天队列未被使用时，此队列将被删除。
- 使用auto-delete队列：当最后一个消费者退出或通道/连接关闭（或与服务器的TCP连接丢失）时，auto-delete队列会被删除。
- 使用exclusive queue：只能在创建exclusive queue的连接中使用，当此连接关闭或消失时，exclusive queue会被删除。

设置auto-delete队列和exclusive queue的方法如下：

```
boolean exclusive = true;
boolean autoDelete = true;
channel.queueDeclare(QUEUENAME, durable, exclusive, autoDelete, arguments);
```

限制使用优先队列的数量

每个优先队列会启动一个Erlang进程，过多的优先队列会影响性能。在大多数情况下，建议使用不超过5个优先队列。

连接和通道

每个连接使用大约100 KB的内存（如果使用TLS会更多），成千上万的连接会导致RabbitMQ负载很高，极端情况下，会导致内存溢出。AMQP协议引入了通道的概念，一个连接中可以有多个通道。连接是长期存在的，AMQP连接的握手过程比较复杂，至少需要7个TCP数据包（如果使用TLS会更多）。相对连接来说，打开和关闭通道会更简单，但是建议通道也设置为长期存在的。例如，应该为每个生产者线程重用相同的通道，不要在每次生产时都打开通道。最佳实践是重用连接并将线程之间的连接与通道多路复用。

推荐使用Spring AMQP线程池：ConnectionFactory是Spring AMQP定义的连接工厂，负责创建连接。

不要在线程之间共享通道

大多数客户端并未实现通道的线程安全，所以不要在线程之间共享通道。

不要频繁打开和关闭连接或通道

频繁打开和关闭连接或通道会发送和接收大量的TCP包，从而导致更高的延迟，确保不要频繁打开和关闭连接或通道。

生产者和消费者使用不同的连接

生产者和消费者使用不同的连接以实现高吞吐量。当生产者发送太多消息给服务端处理时，RabbitMQ会将压力传递到TCP连接上。如果在同一个TCP连接上消费，服务端可能不会收到来自客户端的消息确认，从而影响消费性能。若消费速度过低，服务端将不堪重负。

大量的连接和通道可能会影响 RabbitMQ 管理接口的性能

RabbitMQ会收集每个连接和通道的数据进行分析和显示，大量连接和通道会影响RabbitMQ管理接口的性能。

禁用未使用的插件

插件可能会消耗大量CPU或占用大量内存，建议禁用未使用的插件。

4 设置队列负载均衡

方案概述

在RabbitMQ集群上，由于扩容节点、删除队列等原因，会导致队列在各个节点分布不均衡，从而造成部分节点压力过大，无法更有效地利用集群。

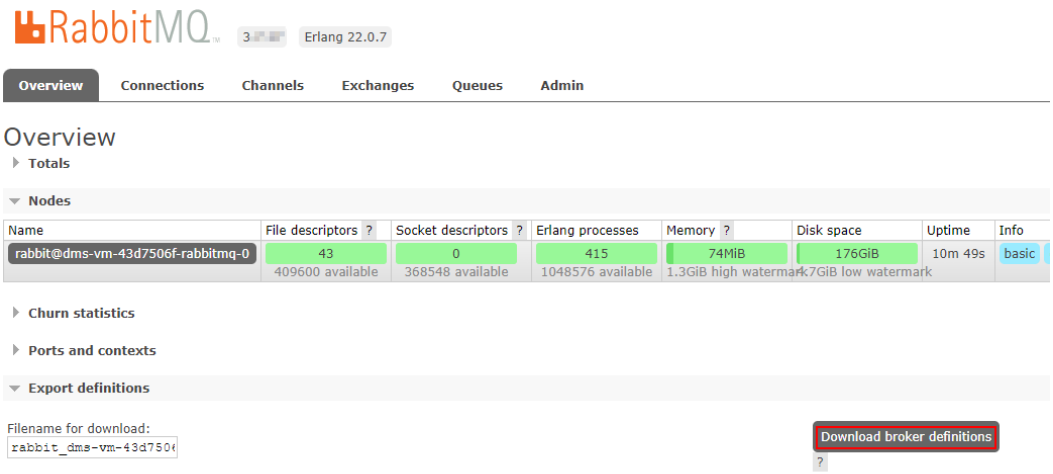
这时候需要手动设置节点间队列的负载均衡，提升集群的利用率。设置队列负载均衡的方法如下：

- **删除队列重建**
- **通过Policy修改master节点方式**

删除队列重建

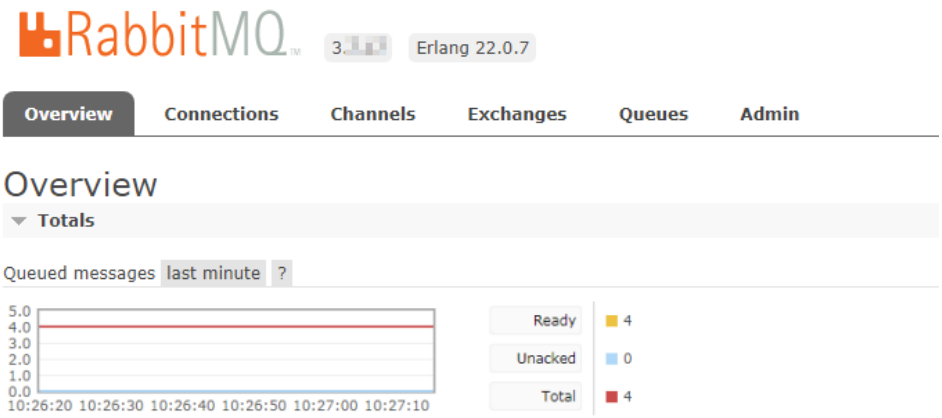
步骤1 **登录RabbitMQ WebUI页面**。

步骤2 在“Overview”页签中，单击“Download broker definitions”，导出元数据。



步骤3 停止生产，等待数据消费完，然后删除原有队列。

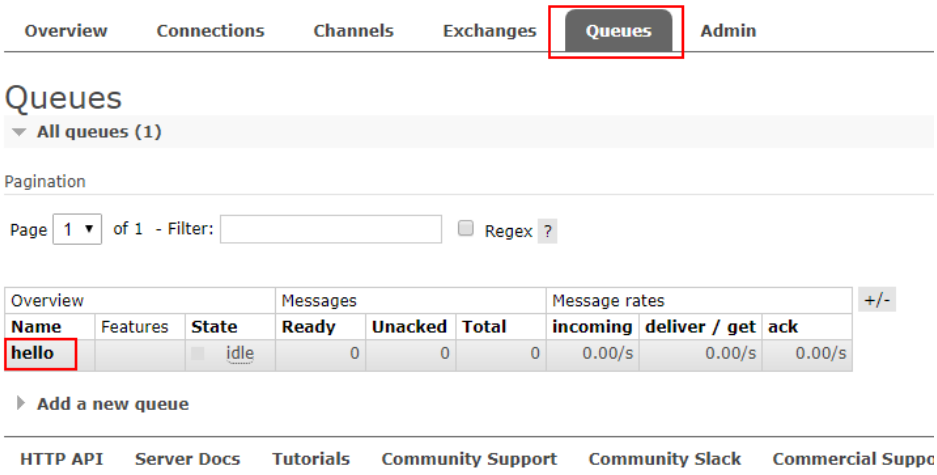
1. 在“Overview”页签中，确认数据是否已消费完。



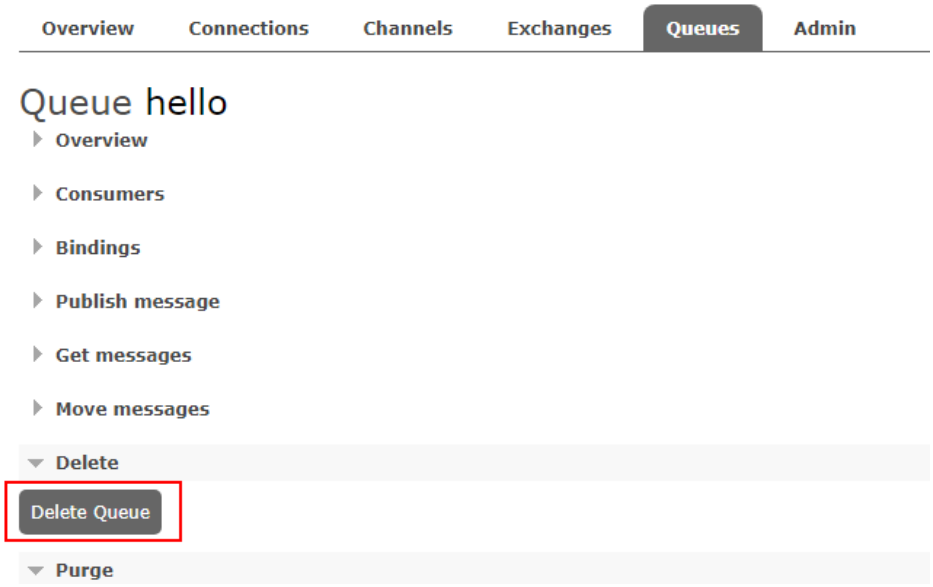
可消费消息数（Ready）和未确认的消息数（Unacked）都为0时，说明消费完成。

2. 等数据消费完后，删除原有队列。

   a. 在“Queues”页签，单击需要删除的队列名称，进入队列详情页面。

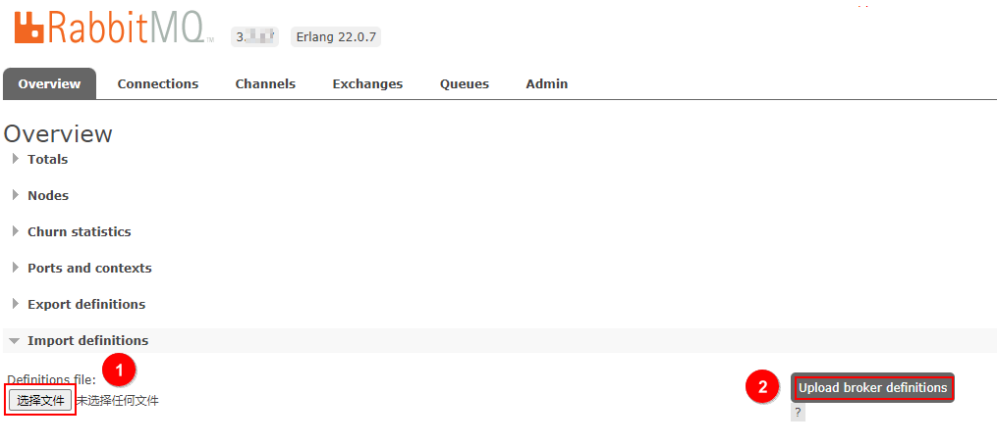


   b. 单击“Delete Queue”，删除队列。

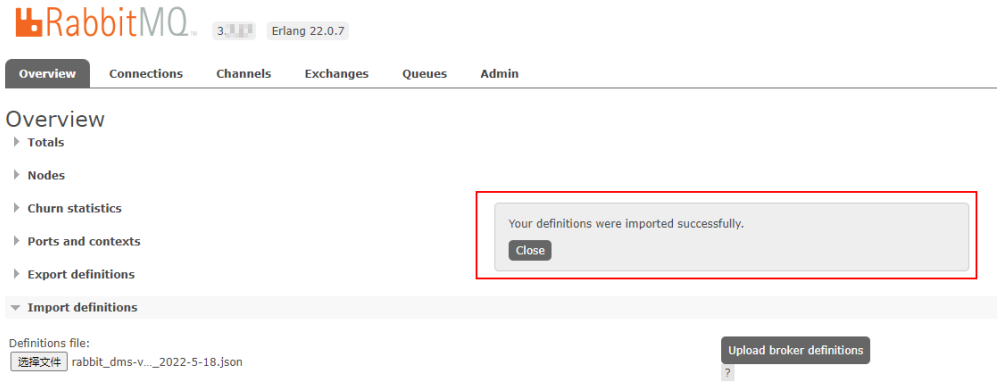


步骤4 在“Overview”页签中，上传已导出的元数据。

1. 在“Overview”页签中，单击“选择文件”，选择已导出的元数据。
2. 单击“Upload broker definitions”，上传元数据。



上传成功后，显示如下信息。



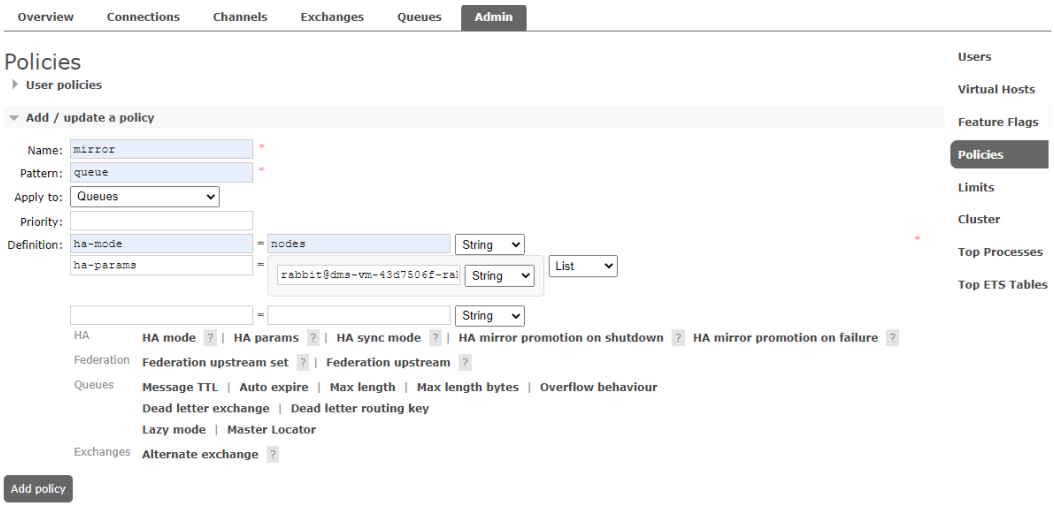
实例会自动将队列均衡创建在各个节点上，在“Queues”页签中查看队列分布详情。

----结束

通过 Policy 修改 master 节点方式

步骤1 **登录RabbitMQ WebUI页面**。

步骤2 在“Admin > Policies”页面中，新增一条策略。



- Name：填写策略名称。
- Pattern：队列匹配模式，填写队列名称，会匹配前缀同名的队列。
- Apply to：选择“Queues”。
- Priority：策略优先级，数字越大，优先级越高。
- Definition：镜像定义。填写“ha-mode”和“ha-params”参数。其中“ha-mode”设置为“nodes”，“ha-params”设置为队列希望迁移到的目的节点名称。

步骤3 单击“Add policy”。

说明

- 队列数据同步需要很长时间，为防止消息丢失，在队列数据完成同步前，原先的master节点依然生效。
- 待队列切换完成后，可删除**步骤2**中新增的策略。

----结束

5 通过消息幂等实现消息去重

方案概述

在RabbitMQ的业务处理过程中，如果消息重发了多次，消费者端对该重复消息消费多次与消费一次的结果是相同的，多次消费并没有对业务产生负面影响，那么这个消息处理过程是幂等的。消息幂等保证了无论消息被重复投递多少次，最终的处理结果都是一致的，避免了因消息重复而对业务产生影响。

例如在支付场景下，用户购买商品后进行支付，由于网络不稳定导致用户收到多次扣款请求，导致重复扣款。但实际上扣款业务只应进行一次，商家也只应产生一条订单流水。这时候使用消息幂等就可以避免这个问题。

在实际应用中，导致消息重复的原因有网络闪断、客户端故障等，且可能发生在消息生产阶段，也可能发生在消息消费阶段。因此，可以将消息重复的场景分为以下两类：

- 生产者发送消息时发生消息重复：

  生产者发送消息时，消息成功发送至服务端。如果此时发生网络闪断，导致生产者未收到服务端的响应，此时生产者会认为消息发送失败，因此尝试重新发送消息至服务端。当消息重新发送成功后，在服务端中就会存在两条内容相同的消息，最终消费者会消费到两条内容一样的重复消息。
- 消费者消费消息时发生消息重复：

  消费者消费消息时，服务端将消息投递至消费者并完成业务处理。如果此时发生网络闪断，导致服务端未收到消费者的响应，此时服务端会认为消息投递失败。为了保证消息至少被消费一次，服务端会尝试投递之前已被处理过的消息，最终消费者会消费到两条内容一样的重复消息。

实施方法

对于消息重复的场景，一般可以使用全局唯一ID来判断该消息是否已消费过。如果已经消费过，则直接返回处理结果，否则进行消息处理，并将全局ID记录下来。

- 生产者为每一条消息设置唯一的messageID，示例代码如下：

```
//持久化消息，并且生成随机的全局唯一messageID
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties().builder();
builder.deliveryMode(2);
builder.messageId(UUID.randomUUID().toString());

//自定义发送的消息
String message = "message content";
```

```
//生产消息，exchangeName和routingKey根据实际填写Queue所属的Exchange名称和Routing Key
channel.basicPublish("exchangeName", "routingKey", false, builder.build(),
message.getBytes(StandardCharsets.UTF_8));
String messageId = builder.build().getMessageId();
System.out.println("messageID: " + messageId);
System.out.println("Send message success!");
//关闭信道
channel.close();
//关闭连接
connection.close();
```

- 消费者根据messageID对消息进行幂等处理，示例代码如下：

```
//创建一个以messageID为主键的数据库表，利用数据库主键去重的方式来处理RabbitMQ幂等。
//在消费者消费前先去数据库查询这条消息是否存在，如果存在表示消息已被消费，无需处理；如果不存在表示消息未被消费，执行消费操作
//queueName根据实际填写要消费的Queue名称
channel.basicConsume("queueName", false, new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
            //获取messageID，并判断是否为空
            String messageId = properties.getMessageId();
            if (StringUtils.isBlank(messageId)){
                logger.info("messageId is null");
                return;
            }
            //查询数据库中是否存在主键为messageID的记录，如果存在，说明这条消息已经被消费，无需处理，否则消费消息，并且在消费完成后将消息记录入库
            //数据库查询逻辑省略
            //todo

            //如果数据库中没有messageID的记录，则执行消费，否则提示消息已消费
            if (null == "{数据库查出来的结果记录}"){
                //获取消息
                String message = new String(body,StandardCharsets.UTF_8);
                //手动响应
                channel.basicAck(envelope.getDeliveryTag(),false);
                logger.info("[x] received message: "  + message + "," + "messageId:" + messageId);

                //存入数据库表中，标识该消息已消费
                //数据库插入操作省略
                //todo
            } else {
                //如果根据messageID查询到消息已消费，则不进行消费
                logger.error("该消息已消费，无需重复消费");
            }
    }
});
```

6 DMS for RabbitMQ 安全使用建议

安全性是华为云与您的共同责任。华为云负责云服务自身的安全，提供安全的云；作为租户，您需要合理使用云服务提供的安全能力对数据进行保护，安全地使用云。详情请参见责任共担。

本文提供了使用DMS for RabbitMQ过程中的安全最佳实践，旨在为提高整体安全能力提供可操作的规范性指导。根据该指导文档您可以持续评估DMS for RabbitMQ资源的安全状态，更好的组合使用DMS for RabbitMQ提供的多种安全能力，提高对DMS for RabbitMQ资源的整体安全防御能力，保护存储在DMS for RabbitMQ内的数据不泄露、不被篡改，以及数据在传输过程中不泄露、不被篡改。

本文从以下几个维度给出建议，您可以评估DMS for RabbitMQ的使用情况，并根据业务需要在本指导的基础上进行安全配置。

- 通过访问控制，保护数据安全性
- SSL链路传输加密方式访问DMS for RabbitMQ
- 不存储敏感数据
- 构建数据的恢复和容灾能力
- 审计是否存在异常数据访问
- 使用最新版本SDK获得更好的操作体验和更强的安全能力

通过访问控制，保护数据安全性

1. **建议对不同角色的IAM用户仅设置最小权限，避免权限过大导致数据泄露或被误操作。**

   为了更好的进行权限隔离和管理，建议您配置独立的IAM管理员，授予IAM管理员IAM策略的管理权限。IAM管理员可以根据您业务的实际诉求创建不同的用户组，用户组对应不同的数据访问场景，通过将用户添加到用户组并将IAM策略绑定到对应用户组，IAM管理员可以为不同职能部门的员工按照最小权限原则授予不同的数据访问权限，详情请参见权限管理。

2. **建议配置安全组访问控制，保护您的数据不被异常读取和操作。**

   租户配置安全组的入方向、出方向规则限制，可以控制连接实例的网络范围，避免DMS for RabbitMQ暴露给不可信第三方，详情请参见配置安全组。安全组入方向规则的“源地址”应避免设置为0.0.0.0/0。

3. **建议将访问RabbitMQ实例方式设置为密码访问，防止未经认证的客户端误操作实例。**

RabbitMQ 3.8.35版本默认使用密码访问，RabbitMQ AMQP-0-9-1版本需要开启ACL访问控制功能，开启ACL权限控制后，生产消息和消费消息时，需要鉴权。

4. **开启敏感操作多因子认证保护您的数据不被误删。**

   DMS for RabbitMQ支持敏感操作保护，开启后执行删除实例等敏感操作时，系统会进行身份验证，进一步对数据的高危操作进行控制，保证DMS for RabbitMQ数据的安全性。详情请参见敏感操作。

SSL 链路传输加密方式访问 DMS for RabbitMQ

为了确保数据传输过程中不被窃取和破坏，建议使用SSL链路传输加密方式访问DMS for RabbitMQ。RabbitMQ 3.8.35版本需要开启SSL功能，RabbitMQ AMQP-0-9-1版本不支持SSL功能。

不存储敏感数据

DMS for RabbitMQ暂不支持数据加密，建议不要将敏感数据存入消息队列。

构建数据的恢复和容灾能力

预先构建数据的容灾和恢复能力，可以有效避免异常数据处理场景下数据误删、破坏的问题。

1. **建议使用RabbitMQ集群实例，获得异常场景数据快速恢复能力。**

   在生产环境中建议使用RabbitMQ集群实例，在实例某个broker故障的情况下，不影响RabbitMQ实例持续提供服务。

2. **建议使用多个可用区构建数据容灾能力。**

   RabbitMQ集群实例支持跨可用区部署，支持跨可用区容灾。如果创建实例时选择了多个可用区，当一个可用区异常时，不影响RabbitMQ实例持续提供服务。

审计是否存在异常数据访问

1. **开启云审计服务，记录RabbitMQ的所有访问操作，便于事后审查。**

   云审计服务（Cloud Trace Service，CTS），是华为云安全解决方案中专业的日志审计服务，提供对各种云资源操作记录的收集、存储和查询功能，可用于支撑安全分析、合规审计、资源跟踪和问题定位等常见应用场景。

   您开通云审计服务并创建和配置追踪器后，CTS可记录RabbitMQ的管理事件和数据事件用于审计。详情请参见**查看RabbitMQ审计日志**。

2. **使用云监控服务对RabbitMQ进行实时监控和告警。**

   为使您更好地掌握RabbitMQ实例状态，华为云提供了云监控服务（Cloud Eye）。您可使用该服务监控自己的RabbitMQ实例，执行自动实时监控、告警和通知操作，帮助您实时掌握RabbitMQ实例中所产生的请求、流量等信息。

   云监控服务不需要开通，会在您创建RabbitMQ实例后自动启动。相关文档请参见**RabbitMQ支持的监控指标**。

使用最新版本 SDK 获得更好的操作体验和更强的安全能力

建议您升级SDK并使用最新版本，从客户侧对您的数据和RabbitMQ使用过程提供更好的保护。最新版本SDK在各语言对应界面下载，请参见**SDK概述**。