分布式消息服务 RabbitMQ 版

最佳实践

文档版本 01

发布日期 2024-04-01

华为云计算技术有限公司

商标声明


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 RabbitMQ 业务迁移

应用场景

RabbitMQ业务迁移主要涉及到以下两个场景：

- 将线下单机或集群实例迁移到线上RabbitMQ实例。
- 将低版本的RabbitMQ实例迁移到高版本的RabbitMQ实例，例如将3.7.17版本的RabbitMQ实例迁移到3.8.35版本的RabbitMQ实例。

迁移原理

在实际业务场景中，RabbitMQ实例存在多个生产者和消费者，在业务迁移时，不会迁移数据，只是通过逐个增加/关闭消费者、生产者的方式迁移实例，这种迁移方式可以实现业务无感迁移。

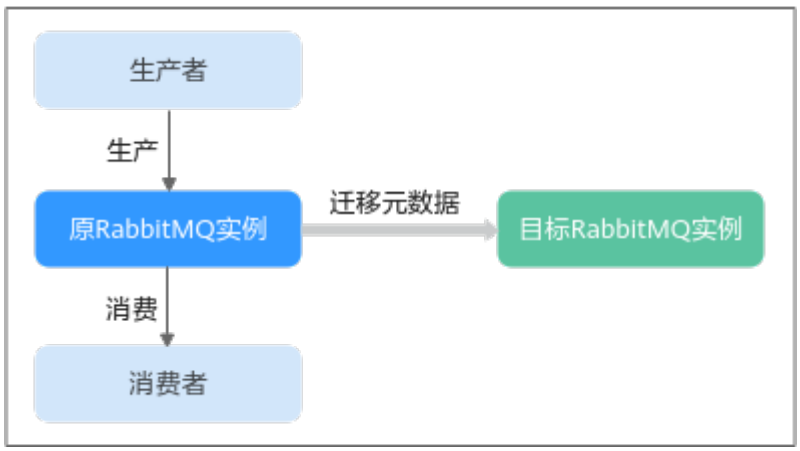
迁移准备

在云上创建目标RabbitMQ实例，具体请参考**购买实例**。

实施步骤（双读模式迁移实例）

步骤1 将原RabbitMQ实例的元数据迁移到目标RabbitMQ实例。

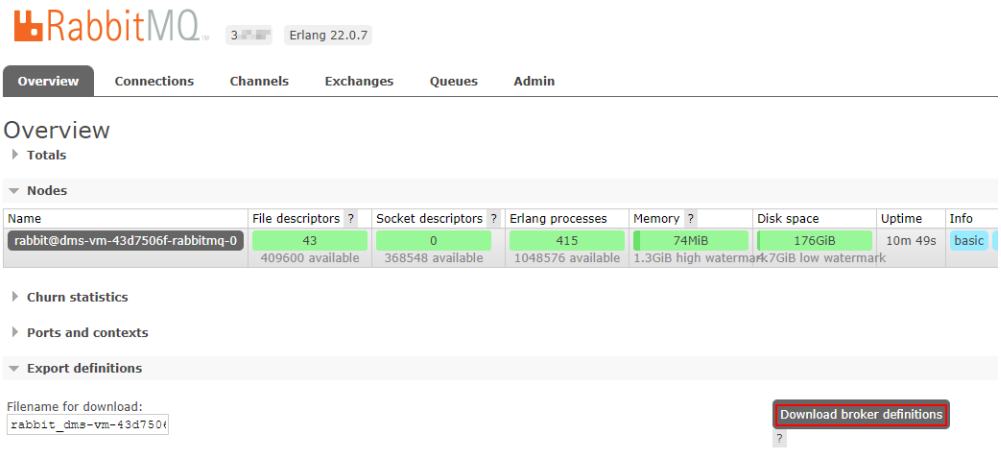
图 1-1 迁移元数据



迁移步骤如下：

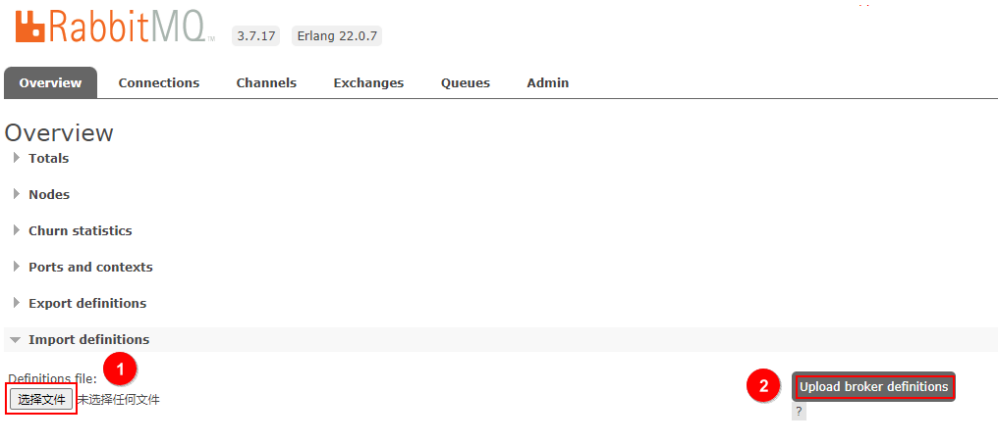
1. 登录原RabbitMQ的WebUI页面，在“Overview”页签中，单击“Download broker definitions”，导出元数据。

图 1-2 导出元数据



2. 登录目标RabbitMQ的WebUI页面，在“Overview”页签中，单击“选择文件”，选择步骤1.1中导出的元数据，单击“Upload broker definitions”，上传元数据。

图 1-3 导入元数据

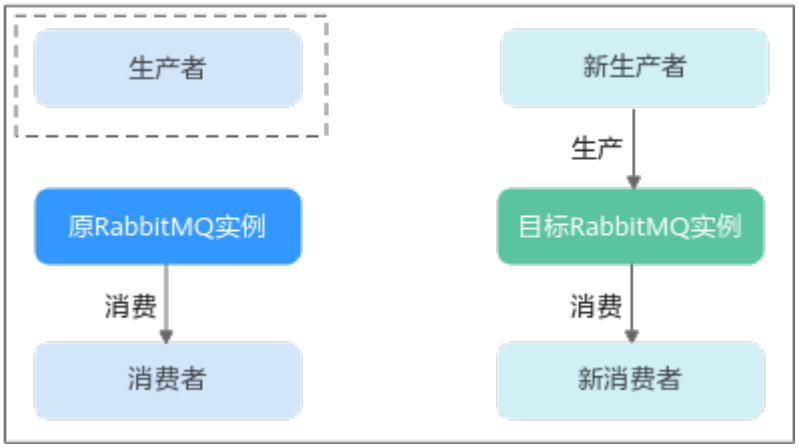


步骤2 为目标RabbitMQ实例添加新的消费者，准备消费目标实例的消息。

图 1-4 添加新消费者

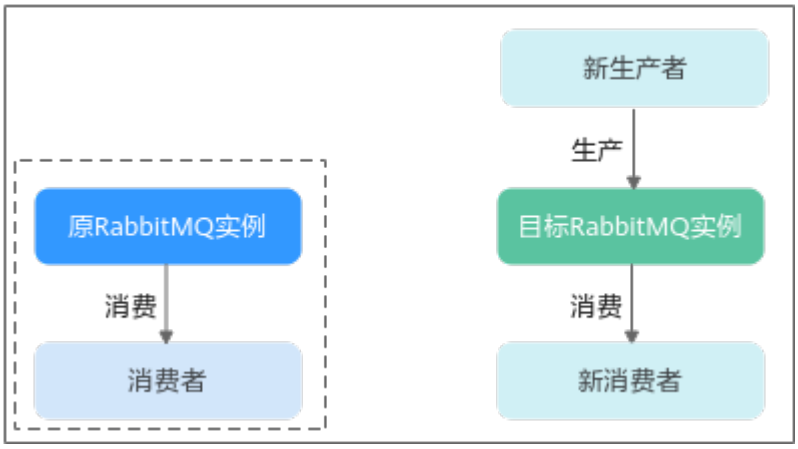
步骤3 为目标RabbitMQ实例添加新的生产者，下线原RabbitMQ实例的生产者，旧的消费者继续消费原RabbitMQ实例中的消息。

图 1-5 迁移生产者



步骤4 旧的消费者消费完原RabbitMQ实例的全部消息后，下线旧的消费者和原RabbitMQ实例。

图 1-6 下线旧的消费者和原 RabbitMQ 实例



----结束

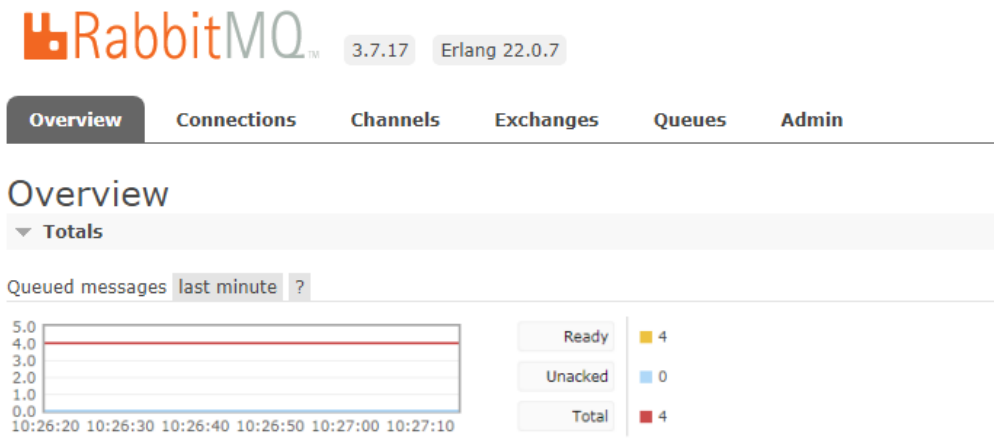
迁移后检查

通过以下方法，确认原实例是否消费完成：

- 在RabbitMQ WebUI页面查看，如图1-7所示。

  Overview视图中，可消费消息数（Ready）以及未确认的消息数（Unacked）都为0时，说明消费完成。

  图 1-7 RabbitMQ WebUI

- 调用API查看。

  curl -s -u *username*:*password* -XGET http://*ip*:*port*/api/overview

  参数说明：

  - username：原实例登录RabbitMQ WebUI的账号
  - password：原实例登录RabbitMQ WebUI的密码
  - ip：原实例登录RabbitMQ WebUI的IP地址
  - port：原实例登录RabbitMQ WebUI的端口号

  回显信息中“messages_ready”和“messages_unacknowledged”都为0时，说明消费完成。

  图 1-8 回显信息


```
"queue_totals":{
        "messages":4,
        "messages_details":{
                "rate":0
        },
        "messages_ready":4,
        "messages_ready_details":{
                "rate":0
        },
        "messages_unacknowledged":0,
        "messages_unacknowledged_details":{
                "rate":0
        }
},
```

2 队列迁移

在RabbitMQ集群上，队列在各个节点分布不均衡会导致部分节点压力过大，无法更有效的利用集群。这可能是扩容节点、删除队列等原因导致的。

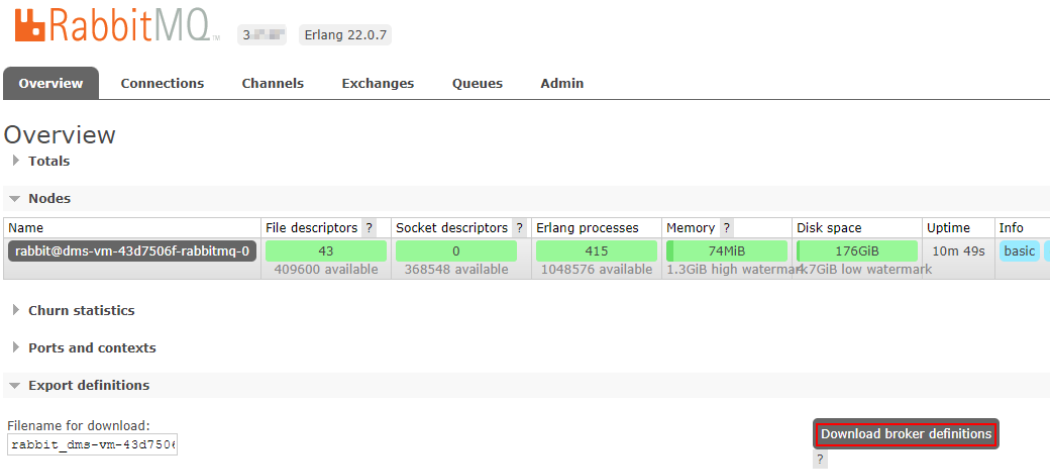
设置队列负载均衡的方法如下：

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

删除队列重建

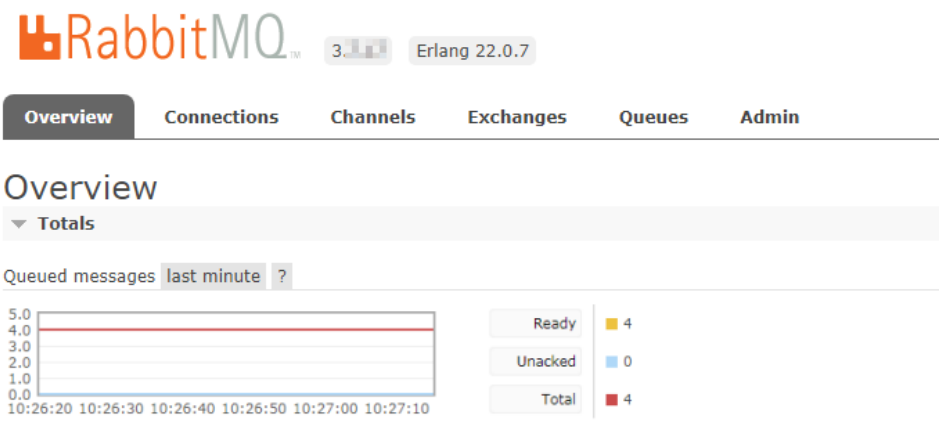
步骤1 **登录RabbitMQ WebUI页面**。

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。

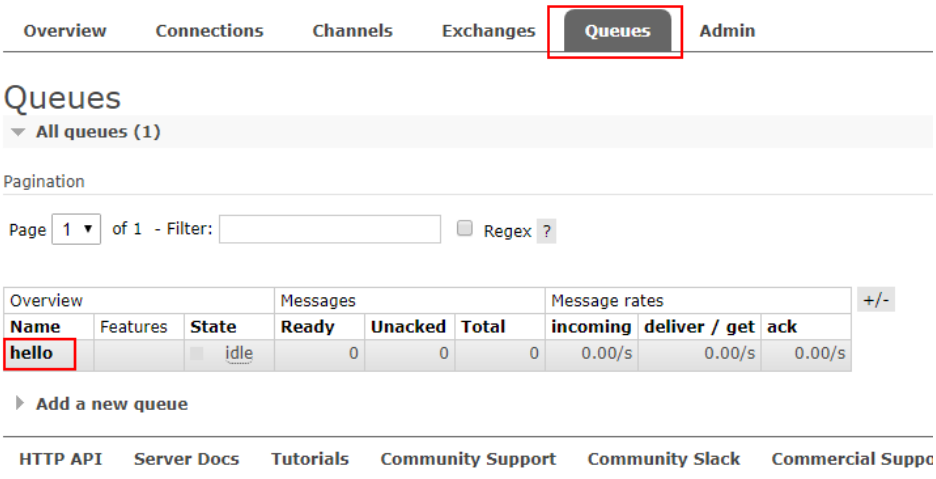
1. 在“Overview”页签中，确认数据是否已消费完。



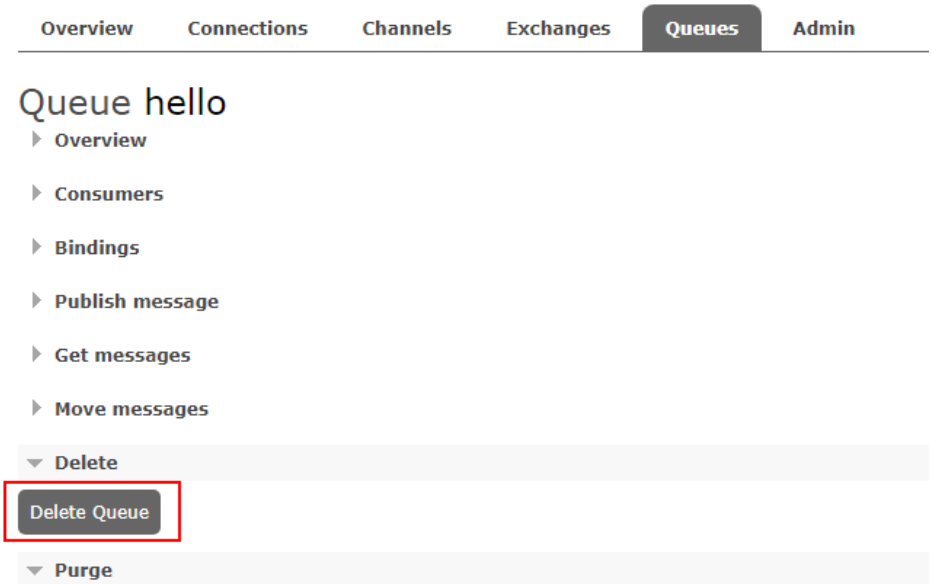
可消费消息数（Ready）和未确认的消息数（Unacked）都为0时，说明消费完成。

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。

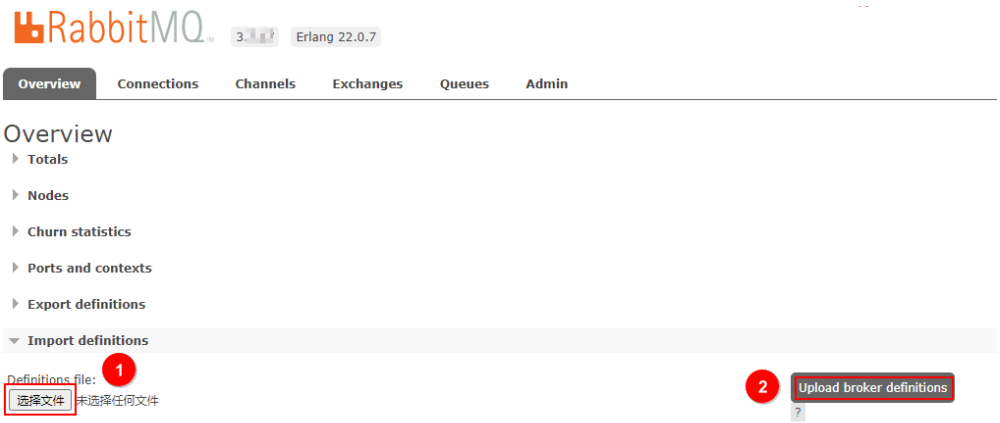


   b. 单击“Delete Queue”，删除队列。

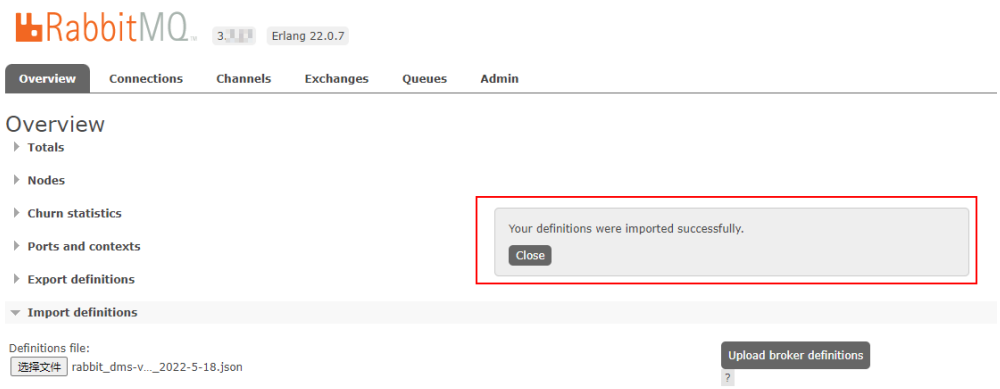


步骤4 在“Overview”页签中，上传已导出的元数据。

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。



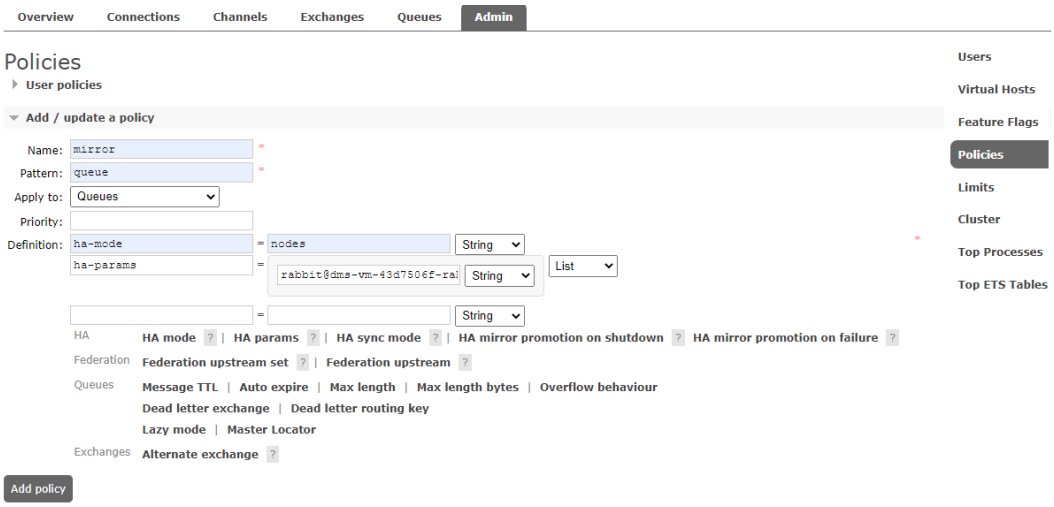
实例会自动将队列均衡创建在各个节点上，在“Queues”页签中查看队列分布详情。

----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：可选参数，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

3 网络异常自动恢复

本章节介绍客户端与服务端由于服务端重启、网络抖动等原因造成网络连接断开时，如何在客户端设置网络自动恢复。**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。**

须知

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。

触发自动恢复的场景

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

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，建议您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();
// enable automatic recovery if using RabbitMQ Java client library prior to version 4.0.0.
factory.setAutomaticRecoveryEnabled(true);
// configure various connection settings

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

4 节点重启后消费者如何重连

本章节以Java中使用的RabbitMQ客户端amqp-client为例介绍节点重启后消费者如何重连。

amqp-client自带重连机制，但是自带的重连机制只会重试一次，一次连不上后就不会再执行了，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丧失的消费能力。

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。如果连上了，这个通道就会继续连接消费。如果连不上，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创建消费者的新通道继续消费。

以下提供一个简单的代码示例，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();
        factory.setHost("100.00.000.000");
        factory.setPort(5672);

        factory.setUsername("name");
        factory.setPassword("password");
        Connection connection = factory.newConnection();

        createNewConnection(connection);
    }
```



```
    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);
        }

    }
}
```

5 如何实现 RabbitMQ 的高性能

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

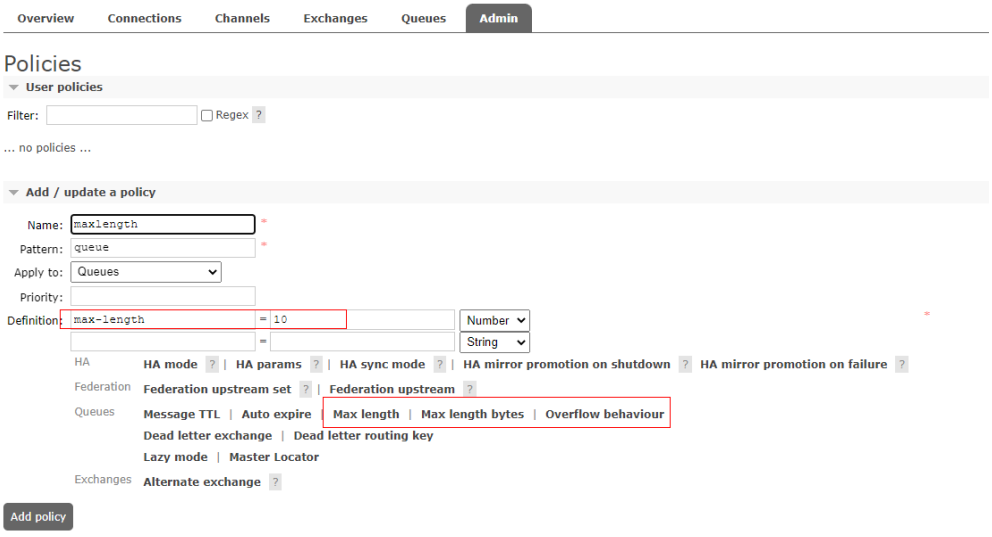
队列中存在大量消息时，会给内存使用带来沉重的负担，为了释放内存，RabbitMQ会将消息刷新到磁盘。这个过程通常需要时间，由于需要重建索引，重启包含大量消息的集群非常耗时。当刷盘的消息过多时，会阻塞队列处理消息，从而降低队列速度，对RabbitMQ节点的性能产生负面影响。

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，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，您还可以使用以下两个插件优化负载均衡：

Consistent hash exchange

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";
```

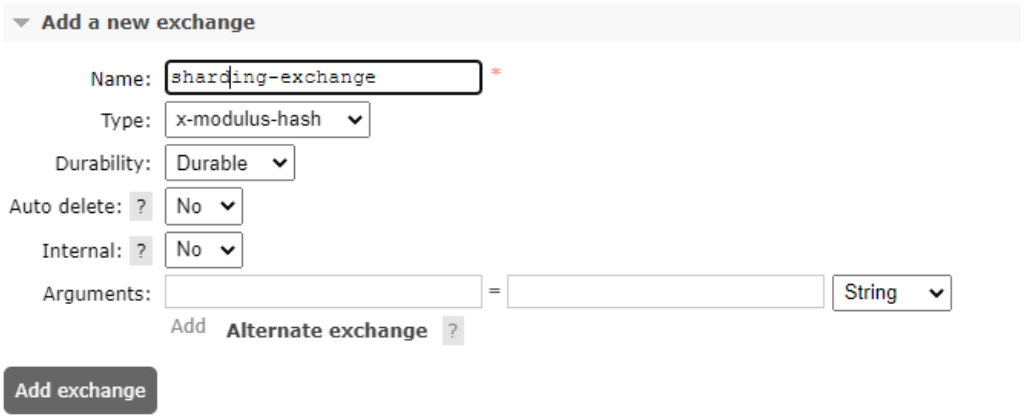
```
  private static String EXCHANGE_TYPE = "x-consistent-hash";

  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();
  }
}
```

RabbitMQ sharding

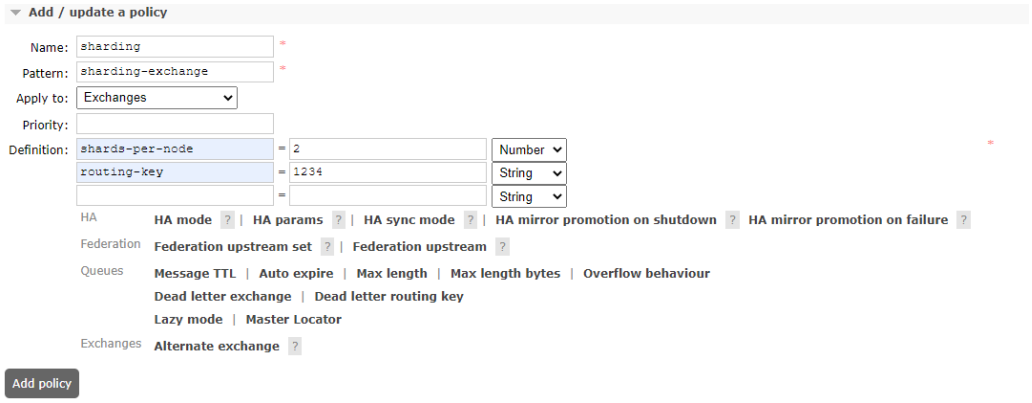
该插件自动对队列进行分区，也就是说，一旦您将一个交换器定义为sharded，那么在每个集群节点上自动创建支持队列，并在它们之间共享消息。该插件提供了一个集中发送消息的位置，并通过向集群中的其他节点添加队列，实现负载均衡。使用此插件时，需要确保消费者从所有队列中消费。

配置RabbitMQ sharding插件的步骤如下：

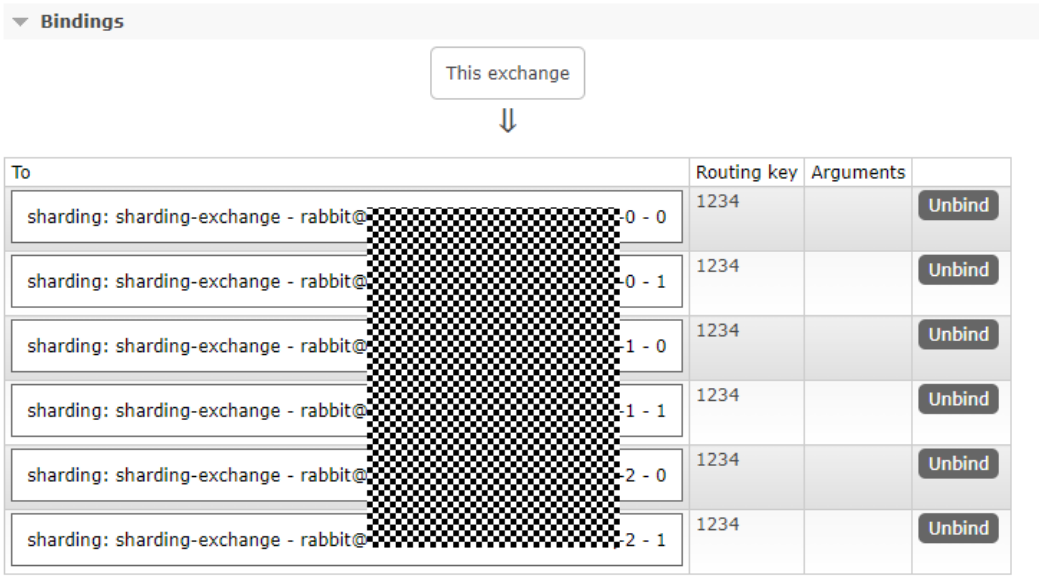
步骤1 创建x-modulus-hash属性交换器。



步骤2 为该交换器添加策略。



步骤3 单击该交换器详情，查看是否配置成功。



----结束

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：exclusive queue只能在创建它的连接中使用，当此连接关闭或消失时，exclusive queue会被删除。

设置方法如下：

```
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。