图灵：楼兰
你的神秘技术宝藏

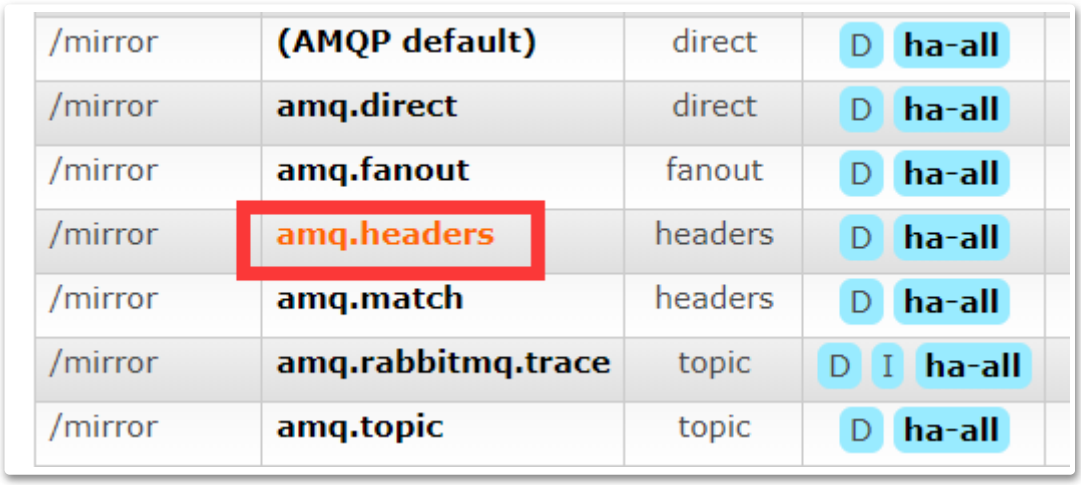
RabbitMQ是一个功能非常全面的MQ产品，本身是基于AMQP这样一个非常严格的开放式协议构建的，又历经了非常多企业的业务场景验证，所以，RabbitMQ的强大，代表的是一个生态，而不仅仅是一个MQ产品。永远不要觉得之前在公司用过或者学会了一些常用的编程框架，就能够彻底掌握好RabbitMQ。这一章节，主要是结合一些应用场景，对上一章节一大堆的编程模型进行查漏补缺。当然，这里的一些使用经验，也不可能完全囊括RabbitMQ的所有问题，还有很多问题，需要自行理解和总结。

# 一、Headers路由

在官网的体验示例中，还有一种路由策略并没有提及，那就是Headers路由。其实官网之所以没有过多介绍，就是因为这种策略在实际中用得比较少，但是在某些比较特殊的业务场景，还是挺好用的。

官网示例中的集中路由策略， direct,fanout,topic等这些Exchange，都是以routingkey为关键字来进行消息路由的，但是这些Exchange有一个普遍的局限就是都是只支持一个字符串的形式，而不支持其他形式。Headers类型的Exchange就是一种忽略routingKey的路由方式。他通过Headers来进行消息路由。这个headers是一个键值对，发送者可以在发送的时候定义一些键值对，接受者也可以在绑定时定义自己的键值对。当键值对匹配时，对应的消费者就能接收到消息。匹配的方式有两种，一种是all，表示需要所有的键值对都满足才行。另一种是any，表示只要满足其中一个键值就可以了。而这个值，可以是List、Boolean等多个类型。

关于Headers路由的示例，首先在Web管理页面上，可以看到默认创建了一个amqp.headers这样的Exchange交换机，这个就是Headers类型的路由交换机。然后关于Headers路由的示例，可以查看示例代码。BasicDemo和SpringBootDemo中均有详细的使用示例。



| | | | |
|---|---|---|---|
| /mirror | (AMQP default) | direct | D ha-all |
| /mirror | amq.direct | direct | D ha-all |
| /mirror | amq.fanout | fanout | D ha-all |
| /mirror | amq.headers | headers | D ha-all |
| /mirror | amq.match | headers | D ha-all |
| /mirror | amq.rabbitmq.trace | topic | D I ha-all |
| /mirror | amq.topic | topic | D ha-all |

例如我们收集应用日志时，如果需要实现按Log4j那种向上收集的方式，就可以使用这种Headers路由策略。

> 日志等级分为 debug - info - warning - error四个级别。而针对四个日志级别，按四个队列进行分开收集，收集时，每个队列对应一个日志级别，然后收集该日志级别以上级别的所有日志(包含当前日志级别)。像这种场景，就比较适合使用Headers路由机制。

# 二、分组消费模式

## 1、什么是分组消费模式

我们回顾下RabbitMQ的消费模式，Exchange与Queue之间的消息路由都是通过RoutingKey关键字来进行的，不同类型的Exchange对RoutingKey进行不同的处理。那有没有不通过RoutingKey来进行路由的策略呢?

> 这个问题其实是很常见的，一个产品的业务模型设计得再完美，也会有照顾不到的场景。例如ShardingSphere分库分表时，默认都是基于SQL语句来进行分库分表，但是为了保证产品灵活性，也提供了hint强制路由策略，脱离了SQL的限制。

在RabbitMQ产品当中，确实没有这样的路由策略，但是在SpringCloudStream框架对RabbitMQ进行封装时，提供了一个这种策略，即分区消费策略。

这种策略很类似于kafka的分组消费策略。 我们回忆一下，在kafka中的分组策略，是不同的group，都会消费到同样的一份message副本，而在同一个group中，只会有一个消费者消费到一个message。这种分组消费策略，严格来说，在Rabbit中是不存在的。RabbitMQ是通过不同类型的exchange来实现不同的消费策略的。这虽然与kafka的这一套完全不同，但是在SpringCloudStream针对RabbitMQ的实现中，可以很容易的看到kafka这种分组策略的影子。

当有多个消费者实例消费同一个bingding时，Spring Cloud Stream同样是希望将这种分组策略，移植到RabbitMQ中来的。就是在不同的group中，会同样消费同一个Message，而在同一个group中，只会有一个消费者消息到一个Message。

## 2、实例测试

要使用分组消费策略，需要在生产者和消费者两端都进行分组配置。

1、生产者端 核心配置

```
#指定参与消息分区的消费端节点数量
spring.cloud.stream.bindings.output.producer.partition-count=2
#只有消费端分区ID为1的消费端能接收到消息
spring.cloud.stream.bindings.output.producer.partition-key-expression=1
```

2、消费者端启动两个实例，组成一个消费者组

消费者1 核心配置

```
#启动消费分区
spring.cloud.stream.bindings.input.consumer.partitioned=true
#参与分区的消费端节点个数
spring.cloud.stream.bindings.input.consumer.instance-count=2
#设置该实例的消费端分区ID
spring.cloud.stream.bindings.input.consumer.instance-index=1
```

消费者2 核心配置

```
#启动消费分区
spring.cloud.stream.bindings.input.consumer.partitioned=true
#参与分区的消费端节点个数
spring.cloud.stream.bindings.input.consumer.instance-count=2
#设置该实例的消费端分区ID
spring.cloud.stream.bindings.input.consumer.instance-index=0
```

这样就完成了一个分组消费的配置。两个消费者实例会组成一个消费者组。而生产者发送的消息，只会被消费者1 消费到(生产者的partition-key-expression 和 消费者的 instance-index 匹配)。

# 3、实现原理

实际上，在跟踪查看RabbitMQ的实现时，就会发现，Spring Cloud Stream在增加了消费者端的分区设置后，会对每个有效的分区创建一个单独的queue，这个队列的队列名是在原有队列名后面加上一个索引值。而发送者端的消息，会最终发送到这个带索引值的队列上，而不是原队列上。这样就完成了分区消费。


我们的示例中，分组表达式是直接指定的，这样其实是丧失了灵活性的。实际开发中，可以将这个分组表达式放到消息的header当中，在发送消息时指定，这样就更有灵活性了。

例如：将生产者端的分组表达式配置为header['partitonkey']

```
#生产者端设置
spring.cloud.stream.bindings.output.producer.partition-key-expression=header['partitionkey']
```

这样，就可以在发送消息时，给消息指定一个header属性，来控制控制分组消费的结果。

```
Message message = MessageBuilder.withPayload(str).setHeader("partitionKey", 0).build();
source.output().send(message);
```

> 这个实验请自行验证。

分组消费策略是在原有路由策略上的一个补充，在实际生产中也是经常会用到的一种策略。并且，MQ的使用场景是非常多的，这也意味着，不管MQ产品设计得如何完善，在复杂场景下，往往都不可能满足所有的使用要求。这时，如果想要自行设计一些更灵活的使用方式，那么这种分组消费的模式就是一个很好的示例。

# 三、死信队列

死信队列是RabbitMQ中非常重要的一个特性。简单理解，他是RabbitMQ对于未能正常消费的消息进行的一种补救机制。死信队列也是一个普通的队列，同样可以在队列上声明消费者，继续对消息进行消费处理。

对于死信队列，在RabbitMQ中主要涉及到几个参数。

```
x-dead-letter-exchange: mirror.dlExchange    对应的死信交换机
x-dead-letter-routing-key:  mirror.messageExchange1.messageQueue1   死信交换机routing-key
x-message-ttl:  3000   消息过期时间
durable:    true   持久化，这个是必须的。
```

在这里，x-dead-letter-exchange指定一个交换机作为死信交换机，然后x-dead-letter-routing-key指定交换机的RoutingKey。而接下来，死信交换机就可以像普通交换机一样，通过RoutingKey将消息转发到对应的死信队列中。

## 1、何时会产生死信

有以下三种情况，RabbitMQ会将一个正常消息转成死信

- 消息被消费者确认拒绝。消费者把requeue参数设置为true(false)，并且在消费后，向RabbitMQ返回拒绝。channel.basicReject或者channel.basicNack。
- 消息达到预设的TTL时限还一直没有被消费。
- 消息由于队列已经达到最长长度限制而被丢掉

> TTL即最长存活时间 Time-To-Live 。消息在队列中保存时间超过这个TTL，即会被认为死亡。死亡的消息会被丢入死信队列，如果没有配置死信队列的话，RabbitMQ会保证死了的消息不会再次被投递，并且在未来版本中，会主动删除掉这些死掉的消息。
>
> 设置TTL有两种方式，一是通过配置策略指定，另一种是给队列单独声明TTL
>
> 策略配置方式 - Web管理平台配置 或者 使用指令配置  60000为毫秒单位
>
> ```
> rabbitmqctl set_policy TTL ".*" '{"message-ttl":60000}' --apply-to queues
> ```
>
> 在声明队列时指定 - 同样可以在Web管理平台配置，也可以在代码中配置：
>
> ```
> Map<String, Object> args = new HashMap<String, Object>();
> args.put("x-message-ttl", 60000);
> channel.queueDeclare("myqueue", false, false, false, args);
> ```

# 2、死信队列的配置方式

RabbitMQ中有两种方式可以声明死信队列，一种是针对某个单独队列指定对应的死信队列。另一种就是以策略的方式进行批量死信队列的配置。

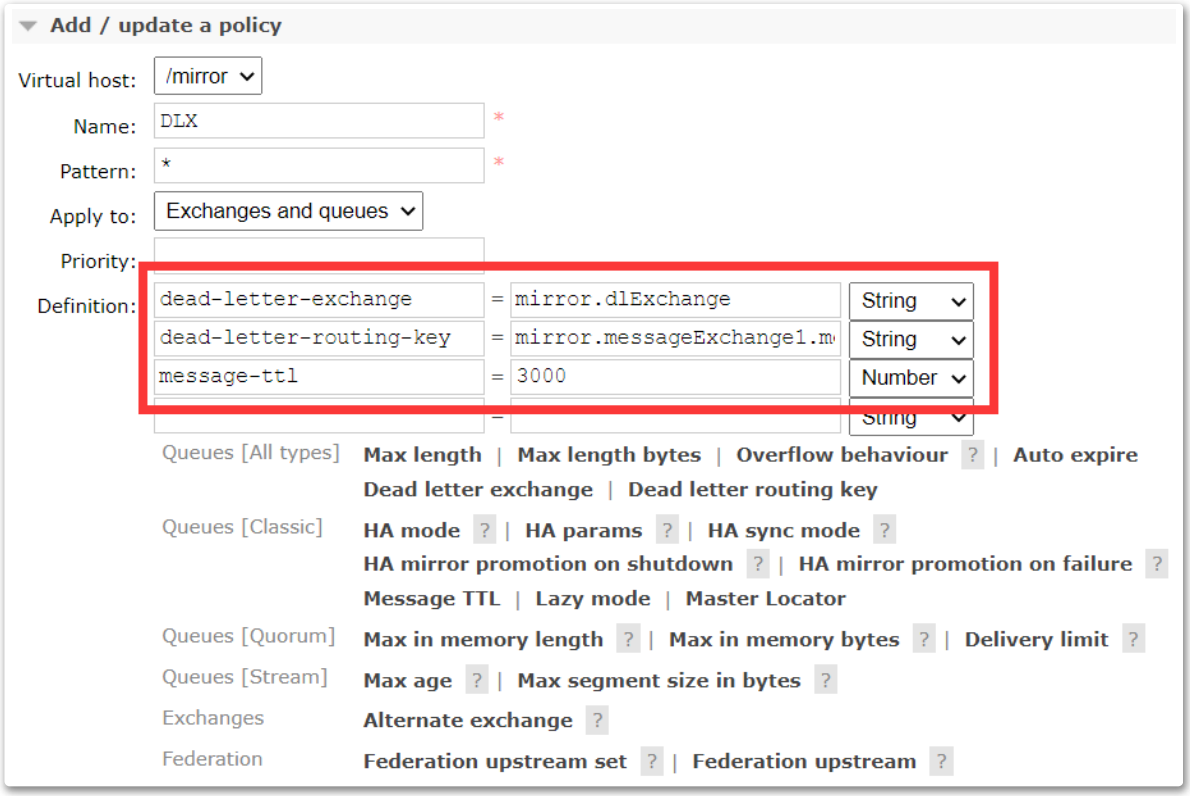
针对多个队列，可以使用策略方式，配置统一的死信队列。、

```
rabbitmqctl set_policy DLX ".*" '{"dead-letter-exchange":"my-dlx"}' --apply-to queues
```

针对队列单独指定死信队列的方式主要是之前提到的三个属性。

```
channel.exchangeDeclare("some.exchange.name", "direct");

Map<String, Object> args = new HashMap<String, Object>();
args.put("x-dead-letter-exchange", "some.exchange.name");
channel.queueDeclare("myqueue", false, false, false, args);
```

这些参数，也可以在RabbitMQ的管理页面进行配置。例如配置策略时：




另外，你会注意到，在对队列进行配置时，只有Classic经典队列和Quorum仲裁队列才能配置死信队列，而目前Stream流式队列，并不支持配置死信队列。

# 3、关于参数x-dead-letter-routing-key

死信在转移到死信队列时，他的Routing key 也会保存下来。但是如果配置了x-dead-letter-routing-key这个参数的话，routingkey就会被替换为配置的这个值。

另外，死信在转移到死信队列的过程中，是没有经过消息发送者确认的，**所以并不能保证消息的安全性。**

# 4、如何确定一个消息是不是死信

消息被作为死信转移到死信队列后，会在Header当中增加一些消息。在官网的详细介绍中，可以看到很多内容，比如时间、原因(rejected,expired,maxlen)、队列等。然后header中还会加上第一次成为死信的三个属性，并且这三个属性在以后的传递过程中都不会更改。

- x-first-death-reason
- x-first-death-queue
- x-first-death-exchange

## 5、死信队列如何消费

其实从前面的配置过程能够看到，所谓死信交换机或者死信队列，不过是在交换机或者队列之间建立一种死信对应关系，而死信队列可以像正常队列一样被消费。他与普通队列一样具有FIFO的特性。对死信队列的消费逻辑通常是对这些失效消息进行一些业务上的补偿。

> RabbitMQ中，是不存在延迟队列的功能的，而通常如果要用到延迟队列，就会采用TTL+死信队列的方式来处理。
>
> RabbitMQ提供了一个rabbitmq_delayed_message_exchange插件，可以实现延迟队列的功能，但是并没有集成到官方的发布包当中，需要单独去下载。这里就不去讨论了。

# 四、消费优先级与流量控制

关于消费队列的优先级，关键是**x-priority** 这个参数，可以指定队列的优先级。默认情况下，RabbitMQ会根据round-robin策略，把消息均匀的给不同的消费者进行处理。但是有了优先级之后，RabbitMQ会保证优先级高的队列先进行消费，而同一优先级的队列，还是会使用round-robin轮询策略来进行分配。

与之对应的是RabbitMQ的流量控制配置，关键是channel.basicQos(int prefetch_size, int prefetch_count, boolean global)。 这个方法中，prefetch_count设置了当前消费者节点最多保持的未答复的消息个数，prefetch_size设置了当前消费节点最多保持的未答复的消息大小，然后global参数为true则表示该配置针对当前channel的所有队列，而默认情况下是false，表示该配置只针对当前消费者队列。最常用的方式就是只设定一个prefetch_count参数。

这两个参数实际上都是为了配置当前消费节点的消息吞吐量。当消费者集群中的业务处理能力或者消息配置不一样时，可以通过给不同的消费节点配置不同的prefetch_count，再结合消费优先级的配置来实现流量控制策略。

# 五、远程数据分发插件-Federation Plugin

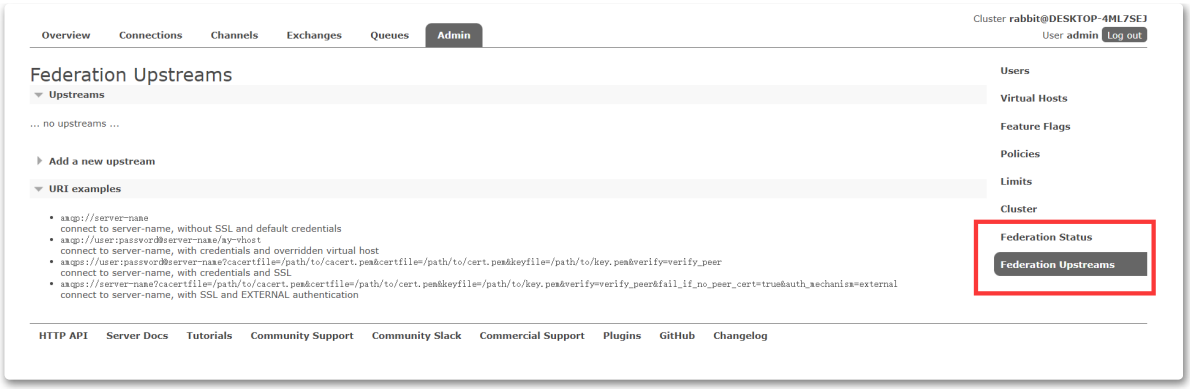
如果我们需要在多个RabbitMQ服务之间进行消息同步，那么，首选的方案自然是通过RabbitMQ集群来进行。但是，在某些网络状况比较差的场景下，搭建集群会不太方便。例如，某大型企业，可能在北京机房和长沙机房分别搭建RabbitMQ服务，然后希望长沙机房需要同步北京机房的消息，这样可以让长沙的消费者服务可以直接连接长沙本地的RabbitMQ，而不用费尽周折去连接北京机房的RabbitMQ服务。这时要如何进行数据同步呢？搭建一个网络跨度这么大的集群显然就不太划算。这时就可以考虑使用RabbitMQ的Federation插件，搭建联邦队列Federation。通过Federation可以搭建一个单向的数据同步通道(当然，要搭建双相同步也是可以的)。

## 1、启动插件

RabbitMQ的官方运行包中已经包含了Federation插件。只需要启动后就可以直接使用。

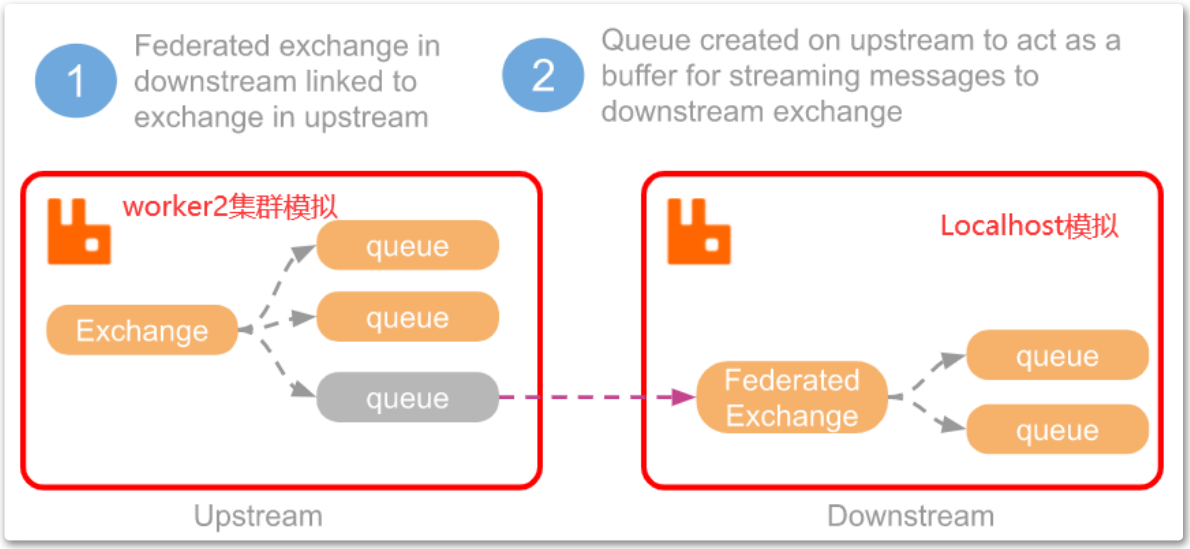
```
# 确认联邦插件
rabbitmq-plugins list|grep federation

[  ] rabbitmq_federation               3.9.15
[  ] rabbitmq_federation_management    3.9.15

# 启用联邦插件
rabbitmq-plugins.bat enable rabbitmq_federation
# 启用联邦插件的管理平台支持
rabbitmq-plugins.bat enable rabbitmq_federation_management
```

插件启用完成后，可以在管理控制台的Admin菜单看到两个新增选项 Federation Status和Federation Upstreams。




# 2、配置Upstream

Upstream表示是一个外部的服务节点，在RabbitMQ中，可以是一个交换机，也可以是一个队列。他的配置方式是由下游服务主动配置一个与上游服务的链接，然后数据就会从上游服务主动同步到下游服务中。



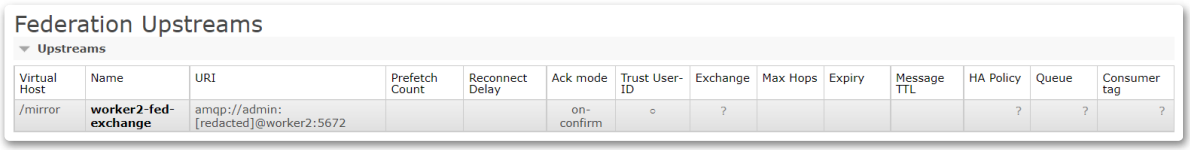

接下来我们用本地localhost的RabbitMQ服务来模拟DownStream下游服务，去指向一个worker2机器上搭建的RabbitMQ服务，搭建一个联邦交换机Federation Exchange。

先在本地用程序的方式，声明一个交换机和交换队列。

```
1  public class DownStreamConsumer {
2      public static void main(String[] args) throws IOException,
   TimeoutException {
3          ConnectionFactory factory = new ConnectionFactory();
4          factory.setHost("localhost");
5          factory.setPort(5672);
6          factory.setUsername("admin");
7          factory.setPassword("admin");
8          factory.setVirtualHost("/mirror");
9          Connection connection = factory.newConnection();
```

```
        Channel channel = connection.createChannel();

        channel.exchangeDeclare("fed_exchange","direct");
        channel.queueDeclare("fed_queue",true,false,false,null);
        channel.queueBind("fed_queue","fed_exchange","routKey");

        Consumer myconsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties, byte[] body)
                    throws IOException {
                System.out.println("========================");
                String routingKey = envelope.getRoutingKey();
                System.out.println("routingKey >" + routingKey);
                String contentType = properties.getContentType();
                System.out.println("contentType >" + contentType);
                long deliveryTag = envelope.getDeliveryTag();
                System.out.println("deliveryTag >" + deliveryTag);
                System.out.println("content:" + new String(body, "UTF-8"));
                // (process the message components here ...)
                //消息处理完后，进行答复。答复过的消息，服务器就不会再次转发。
                //没有答复过的消息，服务器会一直不停转发。
//                channel.basicAck(deliveryTag, false);
            }
        };
        channel.basicConsume("fed_queue", true, myconsumer);
    }
}
```

然后在本地RabbitMQ中配置一个上游服务，服务的名字Name属性随意，URI指向Worker2上的/mirror虚拟机(配置方式参看页面上的示例)：amqp://admin:admin@worker2:5672/ (注意，在添加时，如果指定了Upstream的Virtual Host是mirror，那么在URI中就不能再添加Virtual host配置了，默认会在上游服务中使用相同的VirtualHost。)

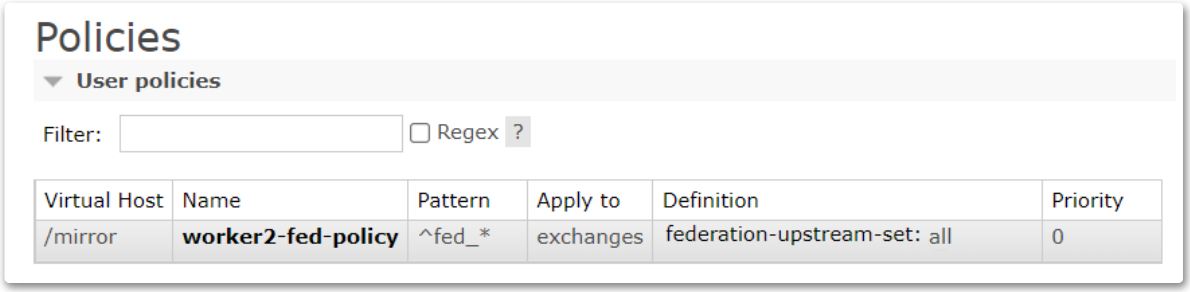



> 注意： 1、其他的相关参数，可以在页面上查看帮助。例如，默认情况下，Upstream会使用和Downstream同名的Exchange，但是也可以通过Upstream配置中的Exchange参数指定不同的。

> 2、关于Virtual Host虚拟机配置，如果在配置Upstream时指定了Virtual Host属性，那么在URI中就不能再添加Virtaul Host配置了，默认会在Upstream上使用相同的Virtual Host。

# 3、配置Federation策略

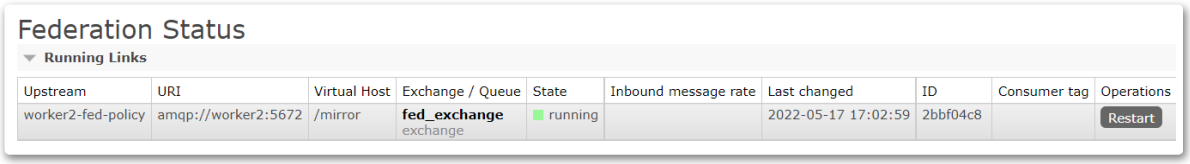
接下来需要配置一个指向上游服务的Federation策略。在配置策略时可以选择是针对Exchange交换机还是针对Queue队列。配置策略时，同样有很多参数可以选择配置。最简化的一个配置如下：



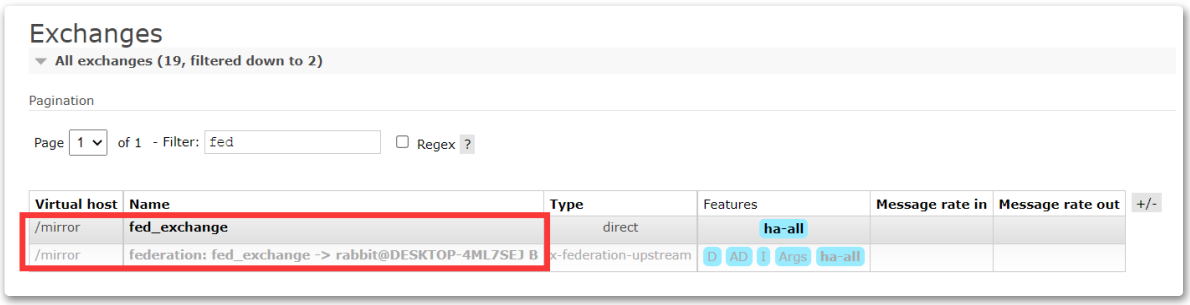

> 注意：每个策略的Definition部分，至少需要指定一个Federation目标。federation-upstream-set参数表示是以set集合的方式针对多个Upstream生效，all表示是全部Upstream。而federation-upstream参数表示只对某一个Upstream生效。

# 4、测试

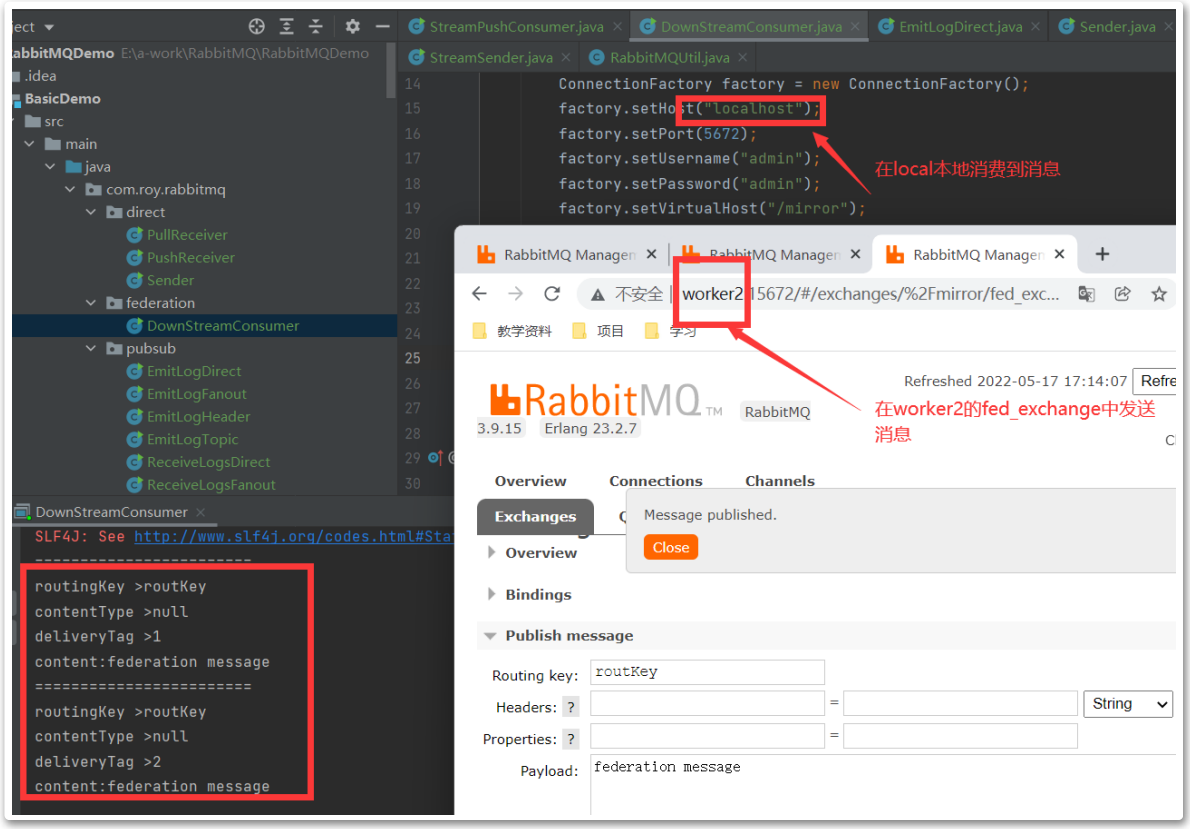
配置完Upstream和对应的策略后，进入Federation Status菜单就能看到Federation插件的执行情况。状态为running表示启动成功，如果配置出错，则会提示失败原因这个提示非常非常简单




然后，在远程服务Worker2的RabbitMQ服务中，可以看到对应生成的Federation交换机。



接下来就可以尝试在上游服务Worker2的fed_exchange中发送消息，消息会同步到Local本地的联邦交换机中，从而被对应的消费者消费到。




在很多网络情况复杂的大型项目中，Federation插件甚至会比镜像集群使用更多，这也是为什么RabbitMQ有了集群机制后，依然一直保留着Federation插件。

> 在我们的实验过程中，会在上游服务重新生成一个新的Exchange交换机，这显然是不太符合实际情况的。我们通常的使用方式是希望将上游MQ中的某一个已有的Exchange交换机或者Queue队列的数据同步到下游一个新的Exchange或者Queue中。这要如何配置呢？大家自行理解实验把。

# 六、懒队列 Lazy Queue

RabbitMQ从3.6.0版本开始，就引入了懒队列的概念。懒队列会尽可能早的将消息内容保存到硬盘当中，并且只有在用户请求到时，才临时从硬盘加载到RAM内存当中。

懒队列的设计目标是为了支持非常长的队列(数百万级别)。队列可能会因为一些原因变得非常长-也就是数据堆积。

- 消费者服务宕机了

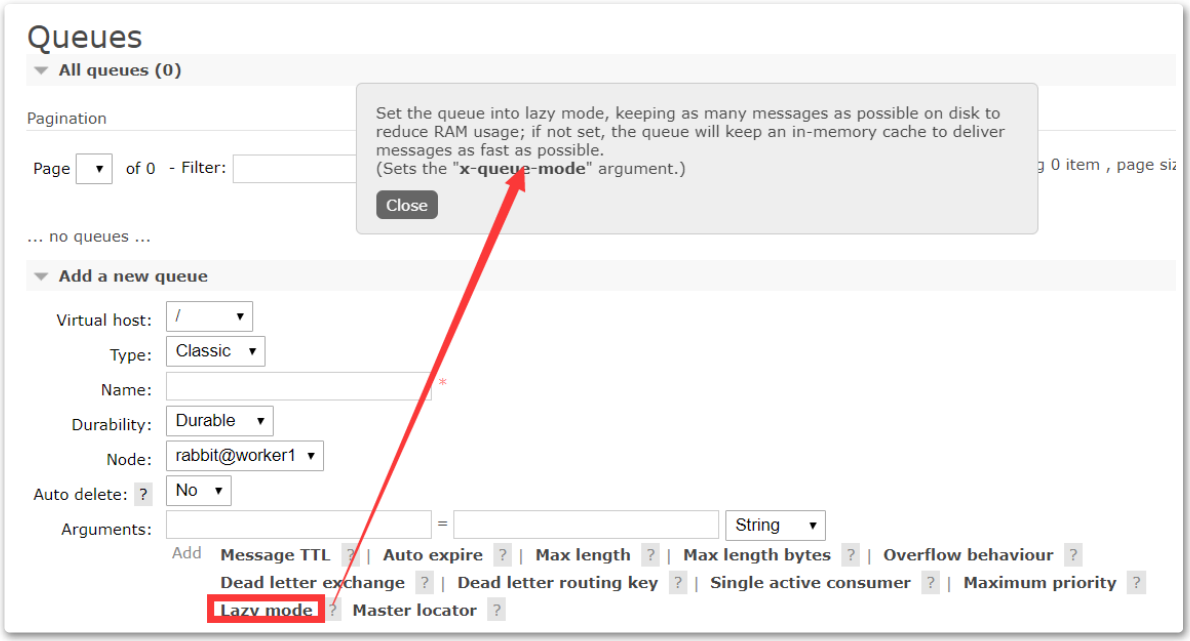
- 有一个突然的消息高峰，生产者生产消息超过消费者
- 消费者消费太慢了

默认情况下，RabbitMQ接收到消息时，会保存到内存以便使用，同时把消息写到硬盘。但是，消息写入硬盘的过程中，是会阻塞队列的。RabbitMQ虽然针对写入硬盘速度做了很多算法优化，但是在长队列中，依然表现不是很理想，所以就有了懒队列的出现。

懒队列会尝试尽可能早的把消息写到硬盘中。这意味着在正常操作的大多数情况下，RAM中要保存的消息要少得多。当然，这是以增加磁盘IO为代价的。

声明懒队列有两种方式：

1、给队列指定参数




在代码中可以通过x-queue-mode参数指定

```
Map<String, Object> args = new HashMap<String, Object>();
  args.put("x-queue-mode", "lazy");
  channel.queueDeclare("myqueue", false, false, false, args);
```

2、设定一个策略，在策略中指定queue-mode 为 lazy。

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

要注意的是，当一个队列被声明为懒队列，那即使队列被设定为不持久化，消息依然会写入到硬盘中。并且，在镜像集群中，大量的消息也会被同步到当前节点的镜像节点当中，并写入硬盘。这会给集群资源造成很大的负担。

最后一句话总结：**懒队列适合消息量大且长期有堆积的队列，可以减少内存使用，加快消费速度。但是这是以大量消耗集群的网络及磁盘IO为代价的。**

# 七、消息分片存储插件-Sharding Plugin

谈到Sharding，你是不是就想到了分库分表？对于数据库的分库分表，分库可以减少数据库的IO性能压力，而真正要解决单表数据太大的问题，就需要分表。

对于RabbitMQ同样，通过集群模式，能够增大他的吞吐量，但是，针对单个队列，如何增加吞吐量呢？普通集群保证不了数据的高可用，而镜像队列虽然可以保证消息高可用，但是消费者并不能对消息增加消费并发度，所以，RabbitMQ的集群机制并不能增加单个队列的吞吐量。

上面的懒队列其实就是针对这个问题的一种解决方案。但是很显然，懒队列的方式属于治标不治本。真正要提升RabbitMQ单队列的吞吐量，还是要从数据也就是消息入手，只有将数据真正的分开存储才行。RabbitMQ提供的Sharding插件，就是一个可选的方案。他会真正将一个队列中的消息分散存储到不同的节点上，并提供多个节点的负载均衡策略实现对等的读与写功能。

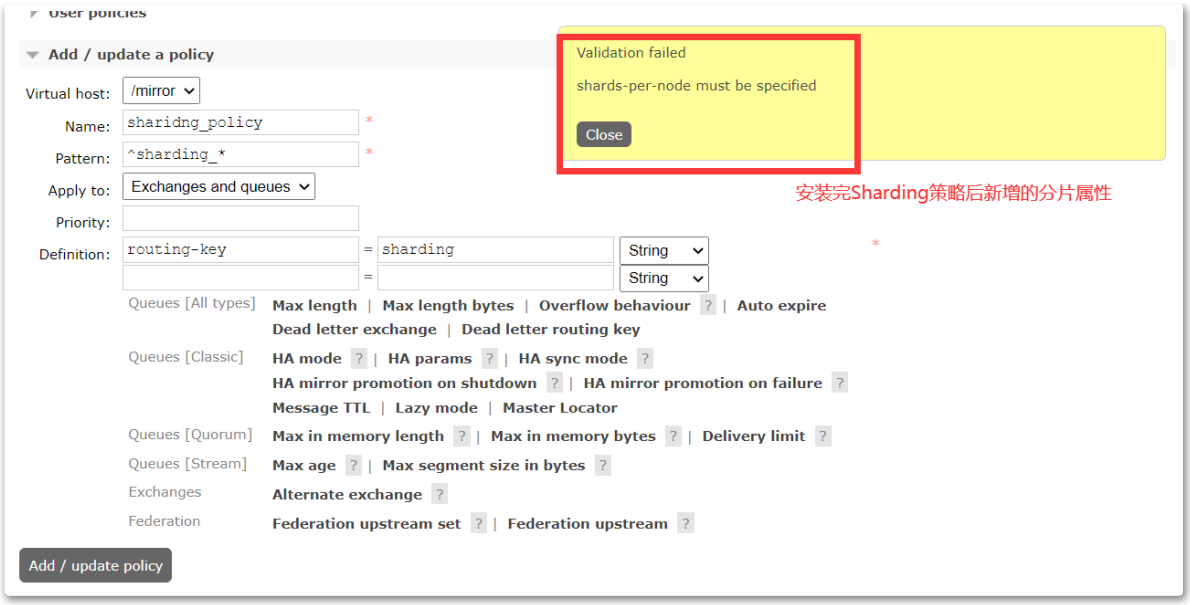
## 1、安装Sharding插件

在当前RabbitMQ的运行版本中，已经包含了Sharding插件，需要使用插件时，只需要安装启用即可。

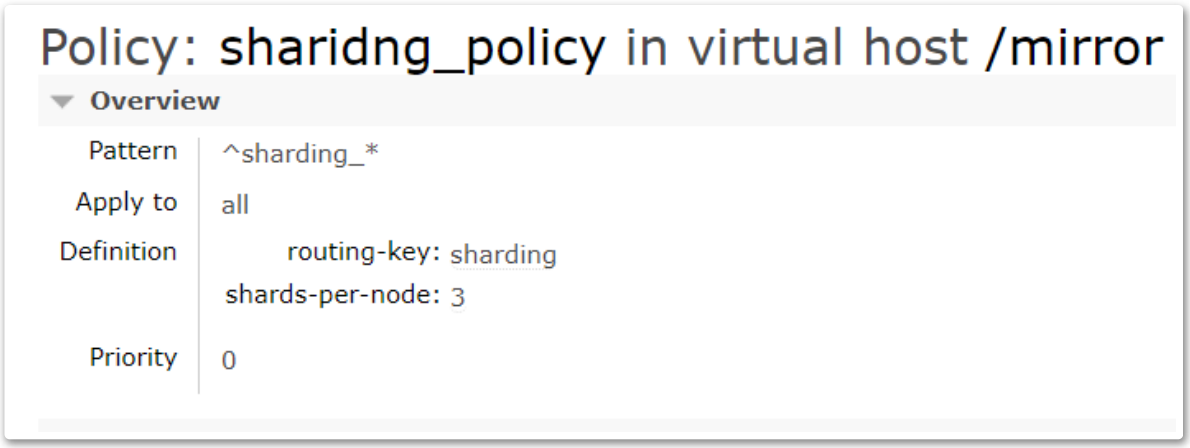
```
rabbitmq-plugins enable rabbitmq_sharding
```

## 2、配置Sharding策略

启用完成后，需要配置Sharding的策略。

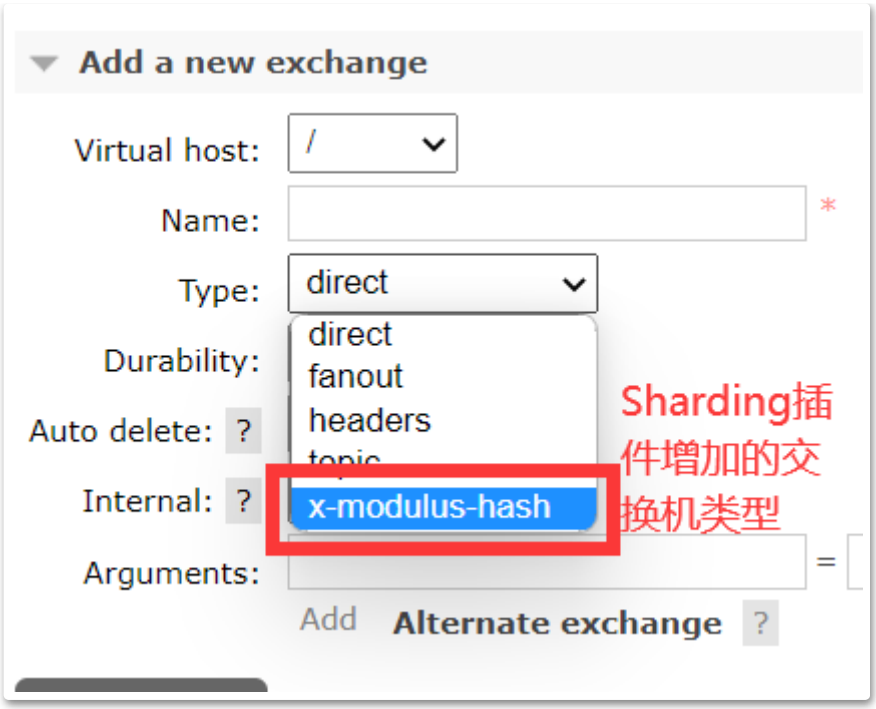



按照要求，就可以配置一个针对sharding_开头的交换机和队列的策略。




# 3、新增带Sharding的Exchange交换机

在创建队列时，可以看到，安装了Sharding插件后，多出了一种队列类型，x-modulus-hash

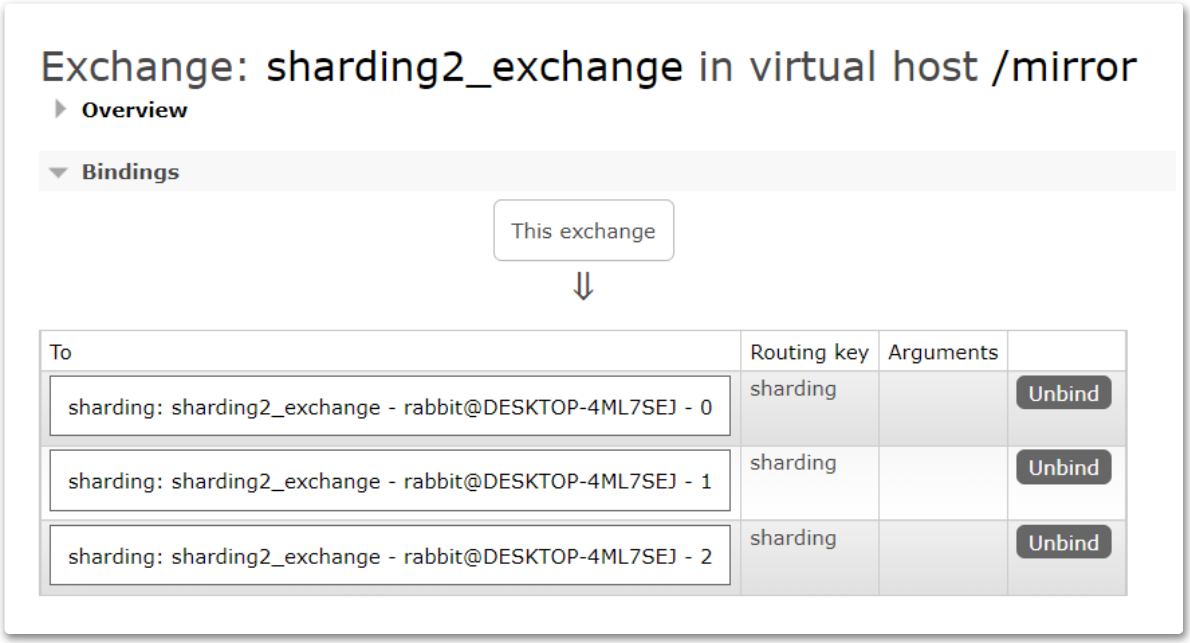



# 4、往分片交换机上发送消息

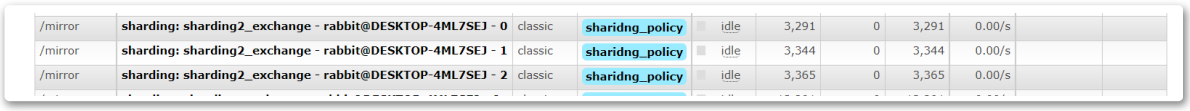
接下来，就可以用下面的生产者代码，在RabbitMQ上声明一个x-modulus-hash类型的交换机，并往里面发送一万条消息。

```
public class ShardingProducer {
    private static final String EXCHANGE_NAME = "sharding2_exchange";

    public static void main(String[] args) throws Exception{
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        factory.setPort(5672);
        factory.setUsername("admin");
        factory.setPassword("admin");
        factory.setVirtualHost("/mirror");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        //发送者只管往exchange里发消息，而不用关心具体发到哪些queue里。
        channel.exchangeDeclare(EXCHANGE_NAME, "x-modulus-hash");
        String message = "LOG INFO 44444";
        for(int i = 0 ; i < 10000 ; i ++){
            channel.basicPublish(EXCHANGE_NAME, String.valueOf(i), null,
message.getBytes());
        }
        channel.close();
        connection.close();
    }
}
```

启动后，就会在RabbitMQ上声明一个sharding_exchange。查看这个交换机的详情，可以看到他的分片情况：




并且，一万条消息被平均分配到了三个队列当中。




Sharding插件带来的x-modulus-hash类型Exchange，会忽略之前的routingkey配置，而将消息以轮询的方式平均分配到Exchange绑定的所有队列上。

# 5、消费分片交换机上的消息

现在sharding2_exchange交换机上的消息已经平均分配到了三个碎片队列上。这时如何去消费这些消息呢？你会发现这些碎片队列的名字并不是毫无规律的，他是有一个固定的格式的。都是固定的这种格式：**sharding: {exchangename}-{node}-{shardingindex}** 。你当然可以针对每个队列去单独声明消费者，这样当然是能够消费到消息的，但是这样，你消费到的消息就是一些零散的消息了，这不符合分片的业务场景要求。

数据分片后，还是希望能够像一个普通队列一样消费到完整的数据副本。这时，Sharding插件提供了一种伪队列的消费方式。你可以声明一个名字为 **exchangename** 的伪队列，然后像消费一个普通队列一样去消费这一系列的碎片队列。

> 为什么说是伪队列？exchange、queue傻傻分不清楚？因为名为exchangename的队列实际是不存在的。

```
public class ShardingConsumer {
    public static final String QUEUENAME="sharding2_exchange";
    public static void main(String[] args) throws IOException,
TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        factory.setPort(5672);
        factory.setUsername("admin");
        factory.setPassword("admin");
        factory.setVirtualHost("/mirror");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUENAME,false,false,false,null);

        Consumer myconsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope,
                                       AMQP.BasicProperties properties,
byte[] body)
                    throws IOException {
                System.out.println("========================");
                String routingKey = envelope.getRoutingKey();
                System.out.println("routingKey >" + routingKey);
                String contentType = properties.getContentType();
                System.out.println("contentType >" + contentType);
                long deliveryTag = envelope.getDeliveryTag();
                System.out.println("deliveryTag >" + deliveryTag);
                System.out.println("content:" + new String(body, "UTF-8"));
                // (process the message components here ...)
                //消息处理完后，进行答复。答复过的消息，服务器就不会再次转发。
                //没有答复过的消息，服务器会一直不停转发。
//                channel.basicAck(deliveryTag, false);
            }
        };
        channel.basicConsume(QUEUENAME, true, myconsumer);
    }
}
```

# 6、注意事项

Sharding插件将消息分散存储时，是尽量按照轮询的方式进行。

首先，这些消息在分片的过程中，是没有考虑消息顺序的，这会让RabbitMQ中原本就不是很严谨的消息顺序变得更加雪上加霜。所以，**Sharding插件适合于那些对于消息延迟要求不严格，以及对消费顺序没有任何要求的的场景。**

然后，Sharding插件消费伪队列的消息时，会从消费者最少的碎片中选择队列。这时，如果你的这些碎片队列中已经有了很多其他的消息，那么再去消费伪队列消息时，就会受到这些不均匀数据的影响。所以，**如果使用Sharding插件，这些碎片队列就尽量不要单独使用了。**

RabbitMQ针对各种复杂场景，还有非常多精妙的设计以及扩展。我们多学习一些扩展的场景，也还只是冰山一角，但是，这些场景已经超出了国内80%企业的应用范围。多学习了解这些扩展场景，对我们的好处不光是能把RabbitMQ用得更好，更重要在于能帮你更深入的理解MQ的应用场景以及各种细节问题。而对于业务问题挖掘深度，才是决定一个程序员实力的前提。

如果你回头再看下这一章节的各种复杂场景，你会发现，这一章节中的队列，还全都只是回到了最初的经典队列。如果结合这些场景，再依次来分析一下新出的Quorum队列，Stream队列，你会有什么感觉?

**你以为你学懂了RabbitMQ，其实只是坐井观天。学得越多，不懂的也越多!**

> 有道云笔记链接地址
>
> 文档：RabbitMQ3高级使用场景.md
> 链接：http://note.youdao.com/noteshare?id=dc1eae561453d5d925889fd0dda401e7&sub=C1163A0A917A424F9A576C8F7E2BCCC4