RabbitMQ高级功能详解以及常用插件实战

图灵：楼兰
你的神秘技术宝藏

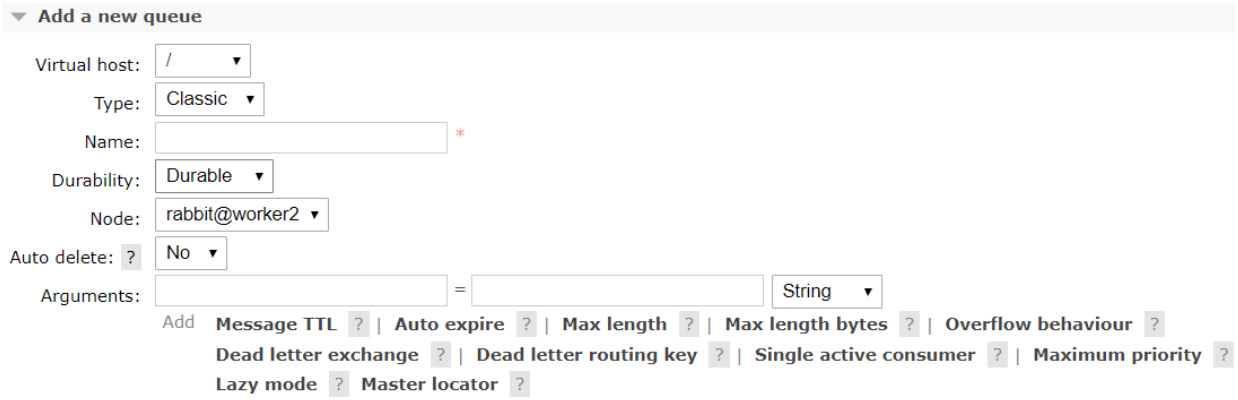
RabbitMQ是一个功能非常全面的MQ产品，本身是基于AMQP这样一个非常严格的开放式协议构建的，又历经了非常多企业的业务场景验证，所以，RabbitMQ的强大，代表的是一个生态，而不仅仅是一个MQ产品。这一章节，主要是结合一些应用场景，对上一章节一大堆的编程模型进行查漏补缺。带大家走出从Demo到实际应用的第一步。

一、选择合适的队列

之前我们一直在使用Classic经典队列。其实在创建队列时可以看到，我们实际上是可以选择三种队列类型的，classic经典队列，Quorum仲裁队列，Stream流式队列。后面这两种队列也是RabbitMQ在最近的几个大的版本中推出的新的队列类型。3.8.x推出了Quorum仲裁队列，3.9.x推出了Stream流式队列。这些新的队列类型都是RabbitMQ针对现代新的业务场景做出的大的改善。最明显的，以往的RabbitMQ版本，如果消息产生大量积累就会严重影响消息收发的性能。而这两种新的队列可以极大的提升RabbitMQ的消息堆积性能。

1、Classic经典队列

这是RabbitMQ最为经典的队列类型。在单机环境中，拥有比较高的消息可靠性。



在这个图中可以看到，经典队列可以选择是否持久化(**Durability**)以及是否自动删除(**Auto delete**)两个属性。

其中，Durability有两个选项，Durable和Transient。 Durable表示队列会将消息保存到硬盘，这样消息的安全性更高。但是同时，由于需要有更多的IO操作，所以生产和消费消息的性能，相比Transient会比较低。

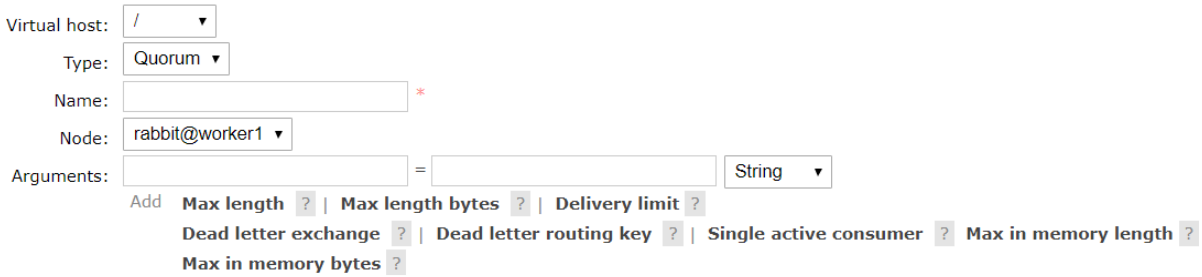
Auto delete属性如果选择为是，那队列将在至少一个消费者已经连接，然后所有的消费者都断开连接后删除自己。

后面的Arguments部分，还有非常多的参数，可以点击后面的问号逐步了解。

在RabbitMQ中，经典队列是一种非常传统的队列结构。消息以FIFO先进先出的方式存入队列。消息被Consumer从队列中取出后就会从队列中删除。如果消息需要重新投递，就需要再次入队。这种队列都依靠各个Broker自己进行管理，在分布式场景下，管理效率是不太高的。并且这种经典队列不适合积累太多的消息。如果队列中积累的消息太多了，会严重影响客户端生产消息以及消费消息的性能。因此，**经典队列主要用在数据量比较小，并且生产消息和消费消息的速度比较稳定的业务场景。**比如内部系统之间的服务调用。

2、Quorum仲裁队列

仲裁队列，是RabbitMQ从3.8.0版本，引入的一个新的队列类型，整个3.8.X版本，也都是在围绕仲裁队列进行完善和优化。仲裁队列相比Classic经典队列，在分布式环境下对消息的可靠性保障更高。官方文档中表示，未来会使用Quorum仲裁队列代替传统Classic队列。



> 关于Quorum的详细介绍见 https://www.rabbitmq.com/quorum-queues.html，这里只是对其中的重点进行下解读

Quorum是基于Raft一致性协议实现的一种新型的分布式消息队列，他实现了持久化，多备份的FIFO队列，主要就是针对RabbitMQ的镜像模式设计的。简单理解就是quorum队列中的消息需要有集群中多半节点同意确认后，才会写入到队列中。这种队列类似于RocketMQ当中的DLedger集群。这种方式可以保证消息在集群内部不会丢失。同时，Quorum是以牺牲很多高级队列特性为代价，来进一步保证消息在分布式环境下的高可靠。

从整体功能上来说，Quorum队列是在Classic经典队列的基础上做减法，因此对于RabbitMQ的长期使用者而言，其实是会影响使用体验的。他与普通队列的区别：

Feature	Classic Mirrored	Quorum
Non-durable queues	yes	no
Exclusivity	yes	no
Per message persistence	per message	always
Membership changes	automatic	manual
Message TTL (Time-To-Live)	yes	yes (since 3.10)
Queue TTL	yes	partially (lease is not renewed on queue re-declaration)
Queue length limits	yes	yes (except `x-overflow`: `reject-publish-dlx`)
Lazy behaviour	yes	always (since 3.10)
Message priority	yes	no
Consumer priority	yes	yes
Dead letter exchanges	yes	yes
Adheres to policies	yes	yes (see Policy support)
Poison message handling	no	yes
Global QoS Prefetch	yes	no

从官方这个比较图就能看到，Quorum队列大部分功能都是在Classic队列基础上做减法，比如Non-durable queues表示是非持久化的内存队列。Exclusivity表示独占队列，即表示队列只能由声明该队列的Connection连接来进行使用，包括队列创建、删除、收发消息等，并且独占队列会在声明该队列的Connection断开后自动删除。

其中有个特例就是Poison Message handling(处理有毒的消息)。所谓毒消息是指消息一直不能被消费者正常消费(可能是由于消费者失败或者消费逻辑有问题等)，就会导致消息不断的重新入队，这样这些消息就成为了毒消息。这些读消息应该有保障机制进行标记并及时删除。Quorum队列会持续跟踪消息的失败投递尝试次数，并记录在"x-delivery-count"这样一个头部参数中。然后，就可以通过设置 Delivery limit参数来定制一个毒消息的删除策略。当消息的重复投递次数超过了Delivery limit参数阈值时，RabbitMQ就会删除这些毒消息。当然，如果配置了死信队列的话，就会进入对应的死信队列。

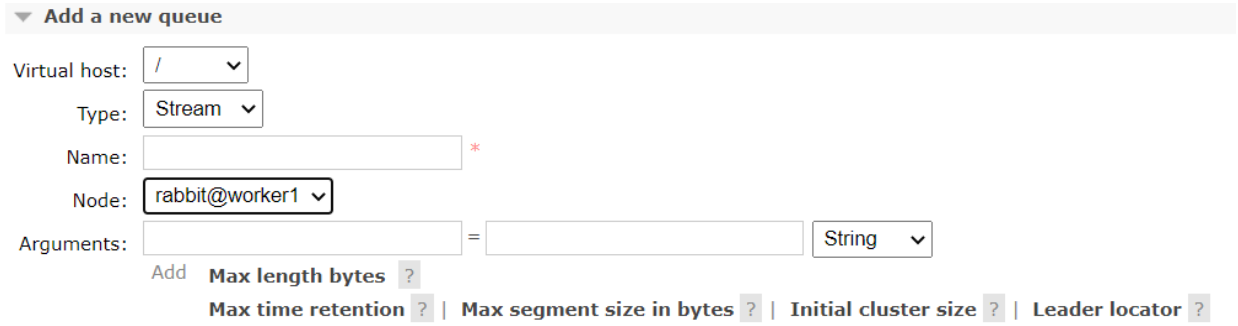
Quorum队列更适合于 队列长期存在，并且对容错、数据安全方面的要求比低延迟、不持久等高级队列更能要求更严格的场景。例如 电商系统的订单，引入MQ后，处理速度可以慢一点，但是订单不能丢失。

也对应以下一些不适合使用的场景：

1、一些临时使用的队列：比如transient临时队列，exclusive独占队列，或者经常会修改和删除的队列。
2、对消息低延迟要求高： 一致性算法会影响消息的延迟。
3、对数据安全性要求不高：Quorum队列需要消费者手动通知或者生产者手动确认。
4、队列消息积压严重： 如果队列中的消息很大，或者积压的消息很多，就不要使用Quorum队列。Quorum队列当前会将所有消息始终保存在内存中，直到达到内存使用极限。

3、Stream流式队列

Stream队列是RabbitMQ自3.9.0版本开始引入的一种新的数据队列类型。这种队列类型的消息是持久化到磁盘并且具备分布式备份的，更适合于消费者多，读消息非常频繁的场景。



> Stream队列的官方文档地址: https://www.rabbitmq.com/streams.html

Stream队列的核心是以append-only只添加的日志来记录消息，整体来说，就是消息将以append-only的方式持久化到日志文件中，然后通过调整每个消费者的消费进度offset，来实现消息的多次分发。下方有几个属性也都是来定义日志文件的大小以及保存时间。如果你熟悉Kafka或者RocketMQ，会对这种日志记录消息的方式非常熟悉。这种队列提供了RabbitMQ已有的其他队列类型不太好实现的四个特点：

1、large fan-outs 大规模分发

当想要向多个订阅者发送相同的消息时，以往的队列类型必须为每个消费者绑定一个专用的队列。如果消费者的数量很大，这就会导致性能低下。而Stream队列允许任意数量的消费者使用同一个队列的消息，从而消除绑定多个队列的需求。

2、Replay/Time-travelling 消息回溯

RabbitMQ已有的这些队列类型，在消费者处理完消息后，消息都会从队列中删除，因此，无法重新读取已经消费过的消息。而Stream队列允许用户在日志的任何一个连接点开始重新读取数据。

3、Throughput Performance 高吞吐性能

Strem队列的设计以性能为主要目标，对消息传递吞吐量的提升非常明显。

4、Large logs 大日志

RabbitMQ一直以来有一个让人诟病的地方，就是当队列中积累的消息过多时，性能下降会非常明显。但是Stream队列的设计目标就是以最小的内存开销高效地存储大量的数据。使用Stream队列可以比较轻松的在队列中积累百万级别的消息。

整体上来说，RabbitMQ的Stream队列，其实有很多地方借鉴了其他MQ产品的优点，在保证消息可靠性的基础上，着力提高队列的消息吞吐量以及消息转发性能。因此，Stream也是在视图解决一个RabbitMQ一直以来，让人诟病的缺点，就是当队列中积累的消息过多时，性能下降会非常明显的问题。RabbitMQ以往更专注于企业级的内部使用，但是从这些队列功能可以看到，Rabbitmq也在向更复杂的互联网环境靠拢，未来对于RabbitMQ的了解，也需要随着版本推进，不断更新。

4、如何使用不同类型的队列

这几种不同类型的队列，虽然实现方式各有不同，但是本质上都是一种存储消息的数据结构。在之前章节，已经对Classic队列的各种编程模型进行了详细分析。而Quorum队列和Stream队列的使用方式也是大同小异的。

1、Quorum队列

Quorum队列与Classic队列的使用方式是差不多的。最主要的差别就是在声明队列时有点不同。

如果要声明一个Quorum队列，则只需要在后面的arguments中传入一个参数，**x-queue-type**，参数值设定为**quorum**。

```
Map<String,Object> params = new HashMap<>();
params.put("x-queue-type","quorum");
//声明Quorum队列的方式就是添加一个x-queue-type参数，指定为quorum。默认是classic
channel.queueDeclare(QUEUE_NAME, true, false, false, params);
```

Quorum队列的消息是必须持久化的，所以durable参数必须设定为true，如果声明为false，就会报错。同样，exclusive参数必须设置为false。这些声明，在Producer和Consumer中是要保持一致的。

2、Stream队列

Stream队列相比于Classic队列，在使用上就要稍微复杂一点。

如果要声明一个Stream队列，则 **x-queue-type**参数要设置为 **stream** 。

```
  Map<String,Object> params = new HashMap<>();
        params.put("x-queue-type","stream");
        params.put("x-max-length-bytes", 20_000_000_000L); // maximum stream size:
20 GB
        params.put("x-stream-max-segment-size-bytes", 100_000_000); // size of
segment files: 100 MB
        channel.queueDeclare(QUEUE_NAME, true, false, false, params);
```

与Quorum队列类似， Stream队列的durable参数必须声明为true，exclusive参数必须声明为false。

这其中，x-max-length-bytes 表示日志文件的最大字节数。x-stream-max-segment-size-bytes 每一个日志文件的最大大小。这两个是可选参数，通常为了防止stream日志无限制累计，都会配合stream队列一起声明。

然后，当要消费Stream队列时，要重点注意他的三个必要的步骤：

- channel必须设置basicQos属性。 与Spring框架集成使用时，channel对象可以在@RabbitListener声明的消费者方法中直接引用，Spring框架会进行注入。
- 正确声明Stream队列。 在Queue对象中传入声明Stream队列所需要的参数。
- 消费时需要指定offset。 与Spring框架集成时，可以通过注入Channel对象，使用原生API传入offset属性。

例如用原生API创建Stream类型的Consumer时，还必须添加一个参数x-stream-offset，表示从队列的哪个位置开始消费。

```
    Map<String,Object> consumeParam = new HashMap<>();
    consumeParam.put("x-stream-offset","last");
    channel.basicConsume(QUEUE_NAME, false,consumeParam, myconsumer);
```

x-stream-offset的可选值有以下几种：

- first: 从日志队列中第一个可消费的消息开始消费
- last: 消费消息日志中最后一个消息

- next: 相当于不指定offset，消费不到消息。
- Offset: 一个数字型的偏移量
- Timestamp:一个代表时间的Data类型变量，表示从这个时间点开始消费。例如 一个小时前 Date timestamp = new Date(System.currentTimeMillis() - 60 * 60 * 1_000)

由于在Consumer中必须传入x-stream-offset这个参数，所以在与SpringBoot集成时，stream队列目前暂时无法正常消费。在目前版本下，使用RabbitMQ的SpringBoot框架集成，可以正常声明Stream队列，往Stream队列发送消息，但是无法直接消费Stream队列了。

关于这个问题，还是需要从Stream队列的三个重点操作入手。SpringBoot框架集成RabbitMQ后，为了简化编程模型，就把channel，connection等这些关键对象给隐藏了，目前框架下，无法直接接入这些对象的注入过程，所以无法直接使用。

如果非要使用Stream队列，那么有两种方式，一种是使用原生API的方式，在SpringBoot框架下自行封装。另一种是使用RabbitMQ的Stream 插件。在服务端通过Strem插件打开TCP连接接口，并配合单独提供的Stream客户端使用。这种方式对应用端的影响太重了，并且并没有提供与SpringBoot框架的集成，还需要自行完善，因此选择使用的企业还比较少。

这里就不详细介绍使用方式了。关于Stream插件的使用和配置方式参见官方文档：https://www.rabbitmq.com/stream.html。配合Stream插件使用的客户端有Java和GO两个版本。其中Java版本客户端参见git仓库：https://github.com/rabbitmq/rabbitmq-stream-java-client 。

最后，在企业中，目前用的最多的还是Classic经典队列。而从RabbitMQ的官网就能看出，RabbitMQ目前主推的是Quorum队列，甚至有传言未来会用Quorum队列全面替代Classic经典队列。至于Stream队列，虽然已经经历了几个版本的完善修复，但是目前还是不太稳定，企业用得还比较少。

二、死信队列

> 文档地址： https://www.rabbitmq.com/dlx.html

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。

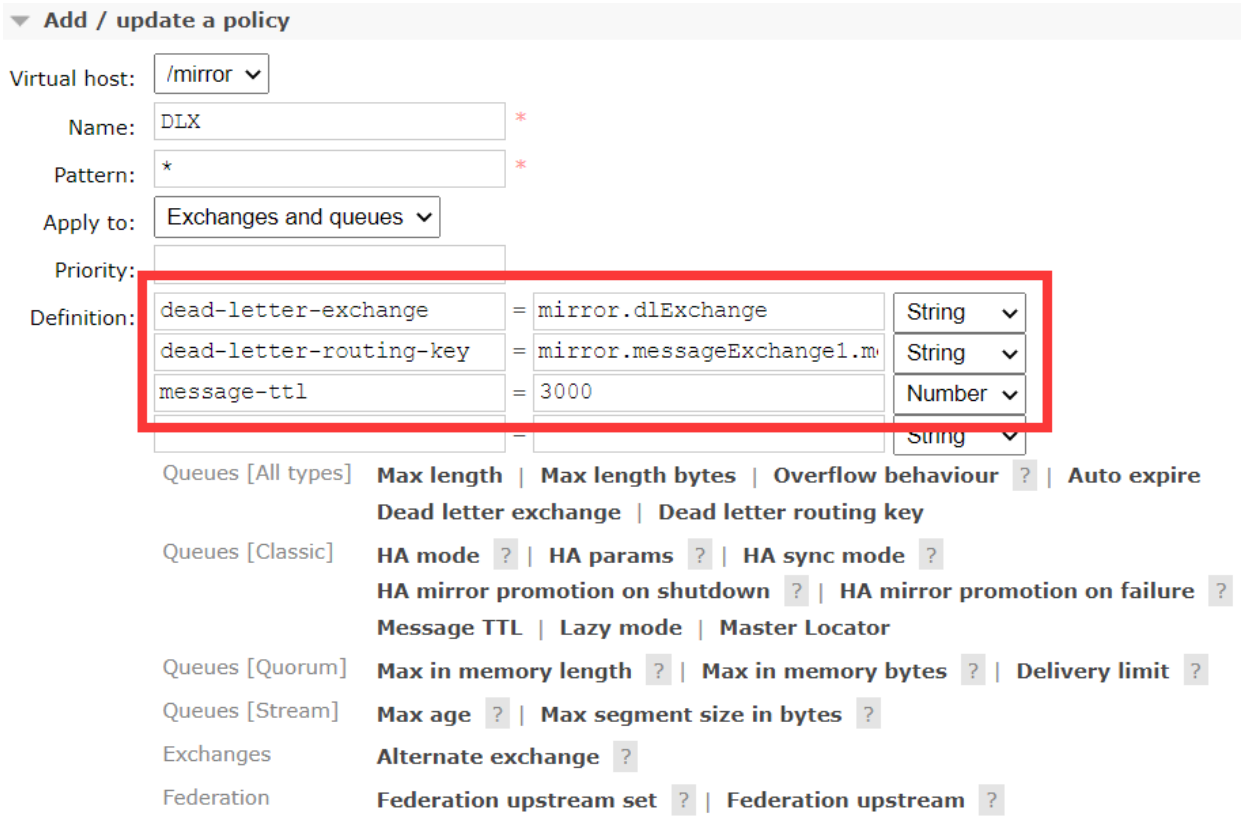
针对多个队列，可以使用策略方式，配置统一的死信队列。、

```
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、基于死信队列实现延迟队列

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。

三、懒队列

> 文档地址： https://www.rabbitmq.com/lazy-queues.html

RabbitMQ从3.6.0版本开始，就引入了懒队列(Lazy Queue)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。

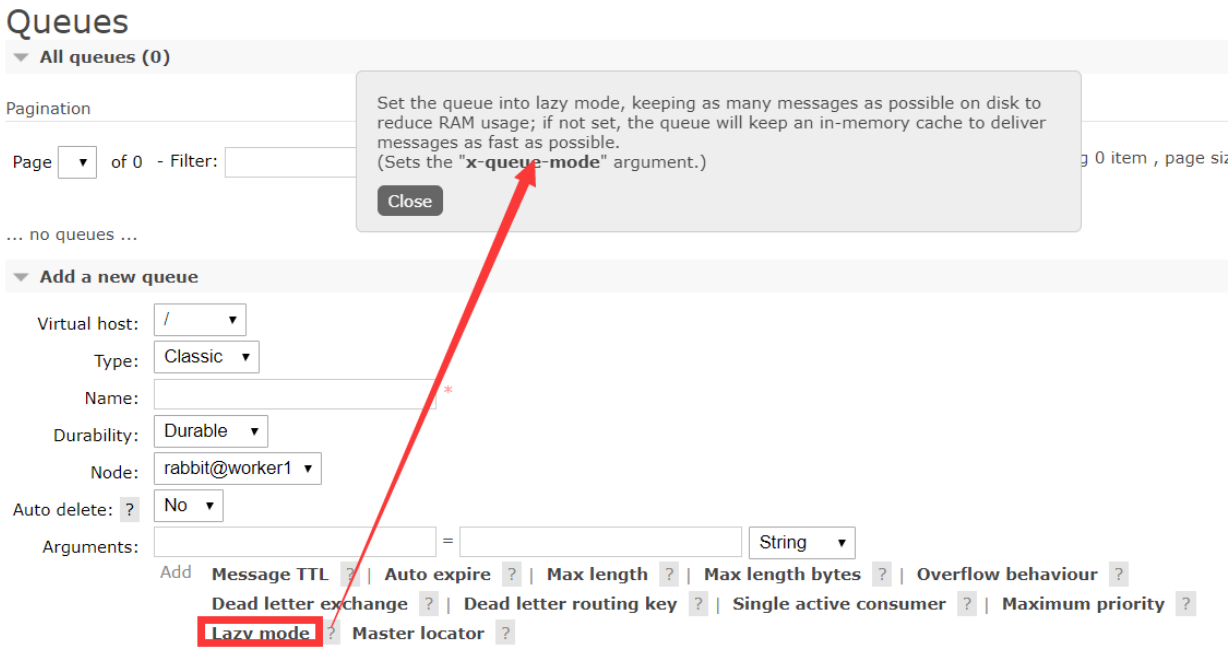
- 消费者服务宕机了
- 有一个突然的消息高峰，生产者生产消息超过消费者
- 消费者消费太慢了

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。如果是在集群模式中使用，这会给集群资源带来很大的负担。

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。**

四、联邦插件

1、插件的作用

在企业中有很多大型的分布式场景，在这些业务场景下，希望服务也能够同样进行分布式部署。这样即可以提高数据的安全性，也能够提升消息读取的性能。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？搭建一个跨度这么大的内部子网显然就不太划算。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。

> 文档地址： https://www.rabbitmq.com/federation.html

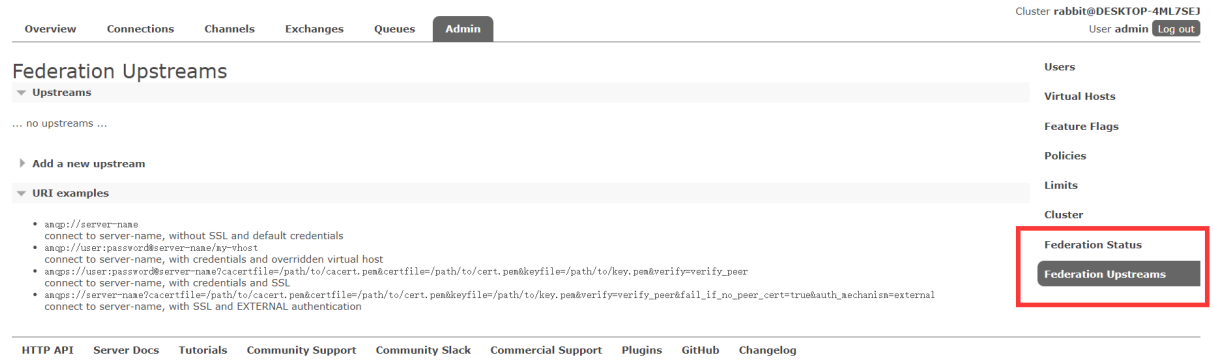
2、使用步骤

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。

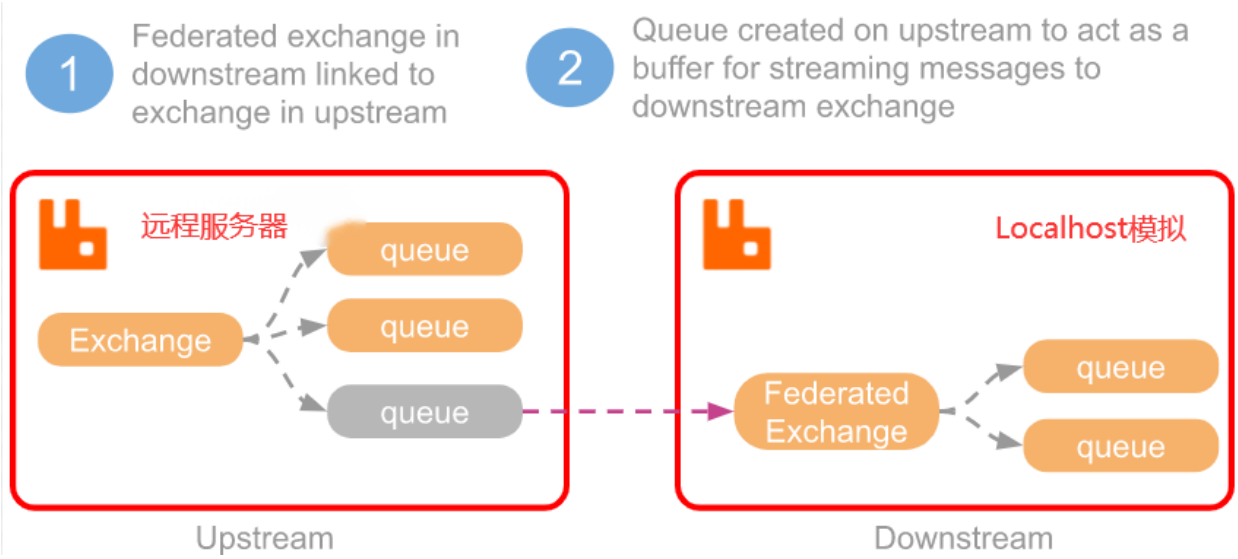
```
# 确认联邦插件
rabbitmq-plugins list|grep federation

[  ] rabbitmq_federation               3.11.10
[  ] rabbitmq_federation_management    3.11.10

# 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。



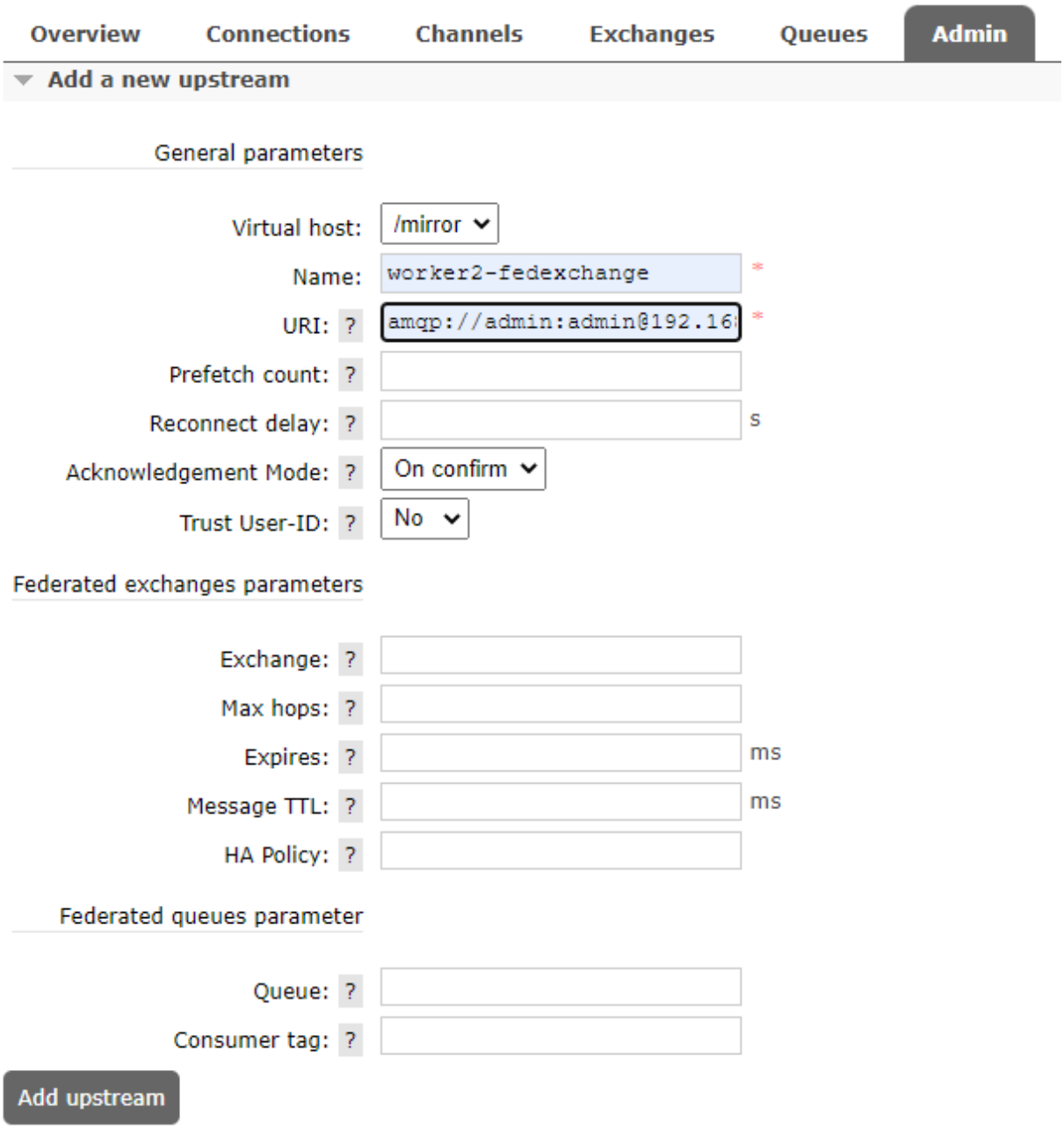
接下来我们用本地localhost的RabbitMQ服务来模拟DownStream下游服务，去指向一个192.168.65.112服务器上搭建的RabbitMQ服务，搭建一个联邦交换机Federation Exchange。

首先要在本地RabbitMQ中声明一个交换机和交换队列，用来接收远端的数据。这里我们直接用客户端API来快速进行声明。

```
public class DownStreamConsumer {
    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.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);
    }
}
```

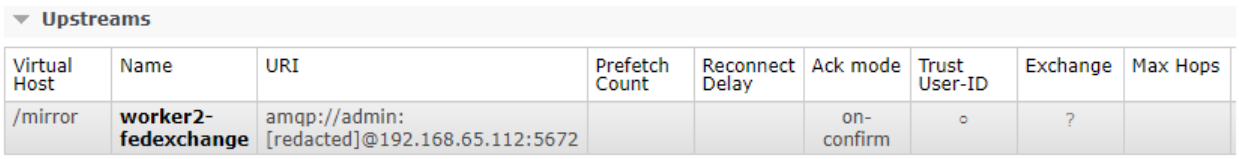
然后在本地RabbitMQ中配置一个上游服务。



服务的名字Name属性随意，URI指向远程服务器(配置方式参看页面上的示例)：amqp://admin:admin@192.168.65.112:5672/

> URI的详细配置方式见页面下方的示例。只不过需要注意下，DownStream和UpStream建议使用相同的虚拟机。

下面的Federated exchanges parameters和Federated queues parameters分别指定Upstream(也就是远程服务器)的Exchange和Queue。如果不指定，就是用和DownStream中相同的Exchange和Queue。如果UpStream里没有，就创建新的Exchange和Queue。

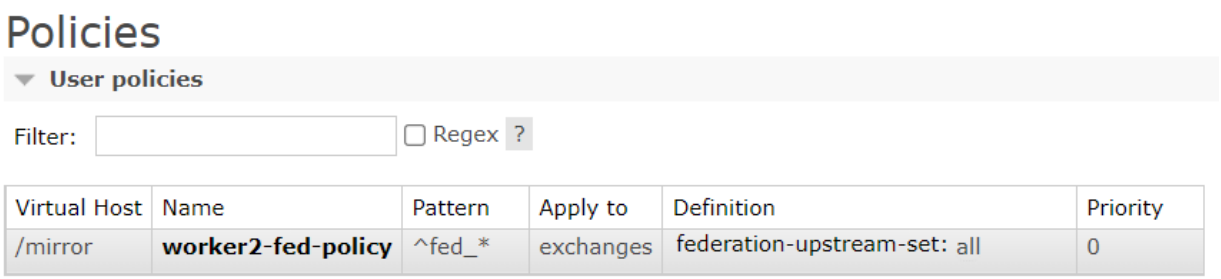


> 注意： 1、其他的相关参数，可以在页面上查看帮助。

> 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Virtual Host配置了，默认会在Upstream上使用相同的Virtual Host。

3、配置Federation策略

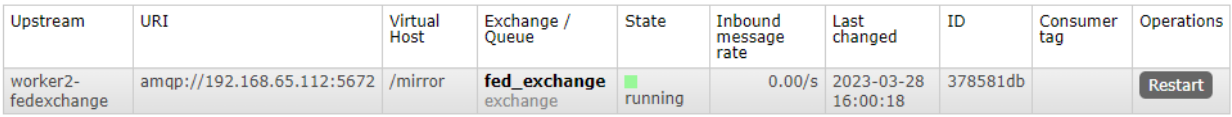
接下来需要配置一个指向上游服务的Federation策略。在配置策略时可以选择是针对Exchange交换机还是针对Queue队列。配置策略时，同样有很多参数可以选择配置。最简化的一个配置如下：



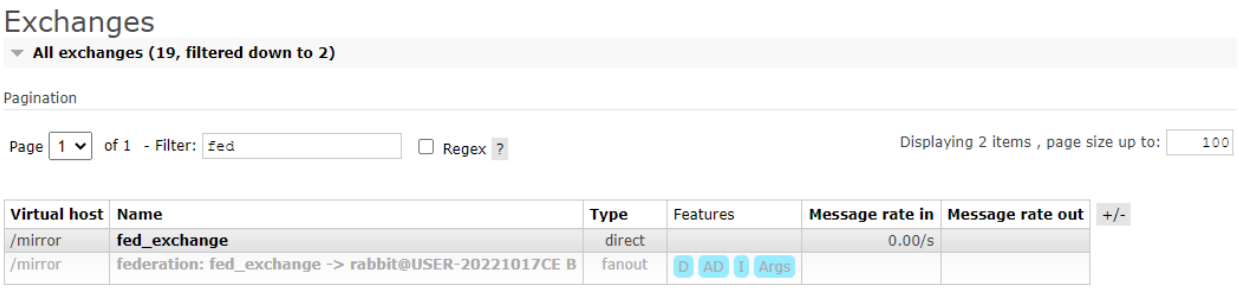
> 注意：每个策略的Definition部分，至少需要指定一个Federation目标。federation-upstream-set参数表示是以set集合的方式针对多个Upstream生效，all表示是全部Upstream。而federation-upstream参数表示只对某一个Upstream生效。

4、测试

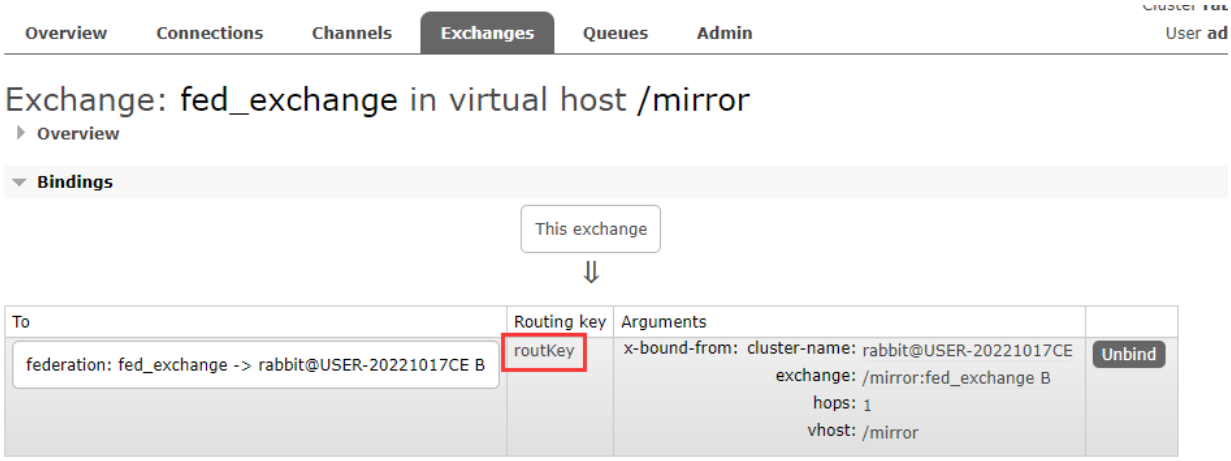
配置完Upstream和对应的策略后，进入Federation Status菜单就能看到Federation插件的执行情况。状态为running表示启动成功，如果配置出错，则会提示失败原因 这个提示非常非常简单



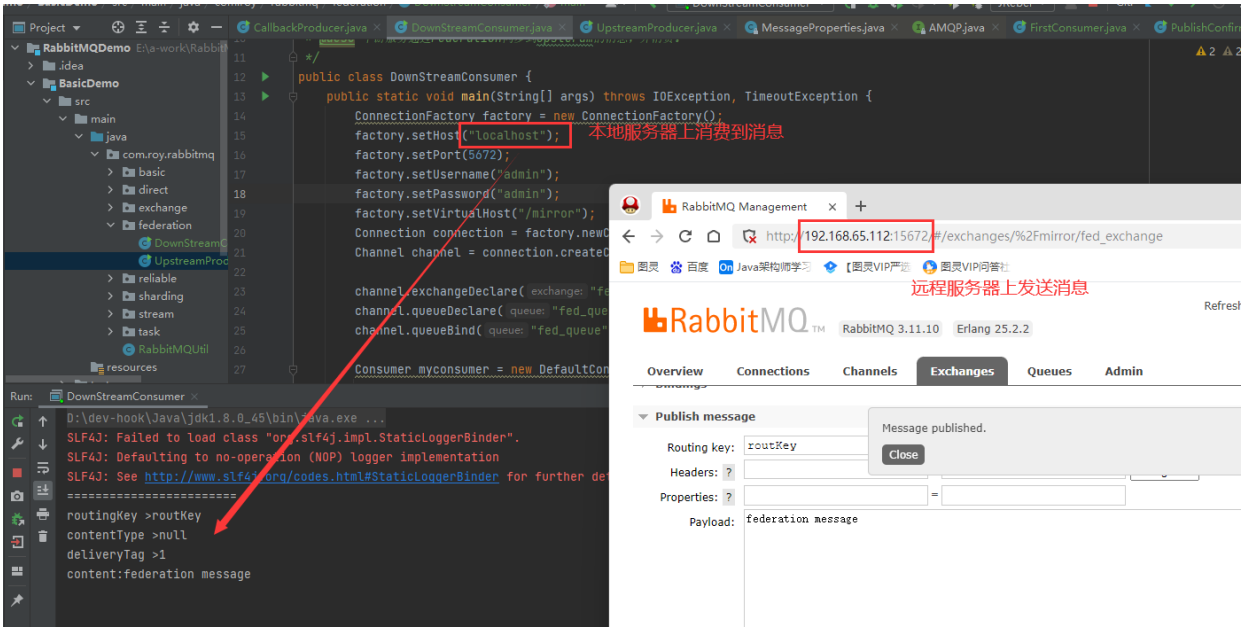
然后，在远程服务Worker2的RabbitMQ服务中，可以看到对应生成的Federation交换机。



并且在fed_exchange的详情页中也能够看到绑定关系。这里要注意一下他给出了一个默认的Routing_key。



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。



渔与鱼：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？你可以自己试试。

五、消息分片存储插件

1、插件的作用

Lazy Queue懒队列机制提升了消息在RabbitMQ中堆积的能力，但是最终，消息还是需要消费者处理消化。但是如何在消费者的处理能力有限的前提下提升消费者的消费速度呢？RabbitMQ提供的Sharding插件，就提供了一种思路。

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，针对单个队列，如何增加吞吐量呢？ 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。

2、使用步骤

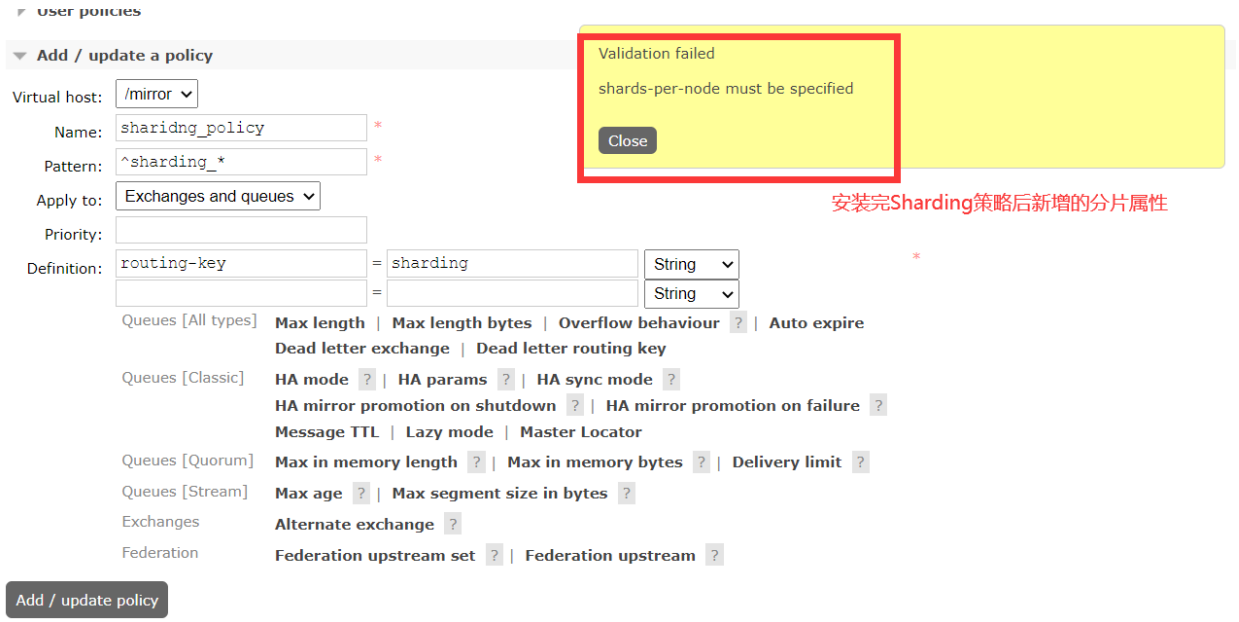
1、启用Sharding插件

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。

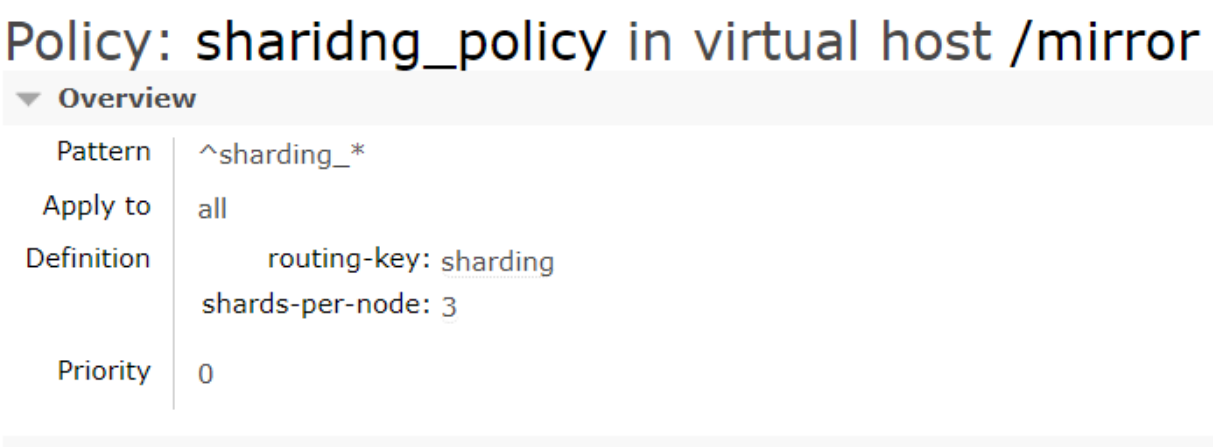
```
rabbitmq-plugins enable rabbitmq_sharding
```

2、配置Sharding策略

启用完成后，需要配置Sharding的策略。

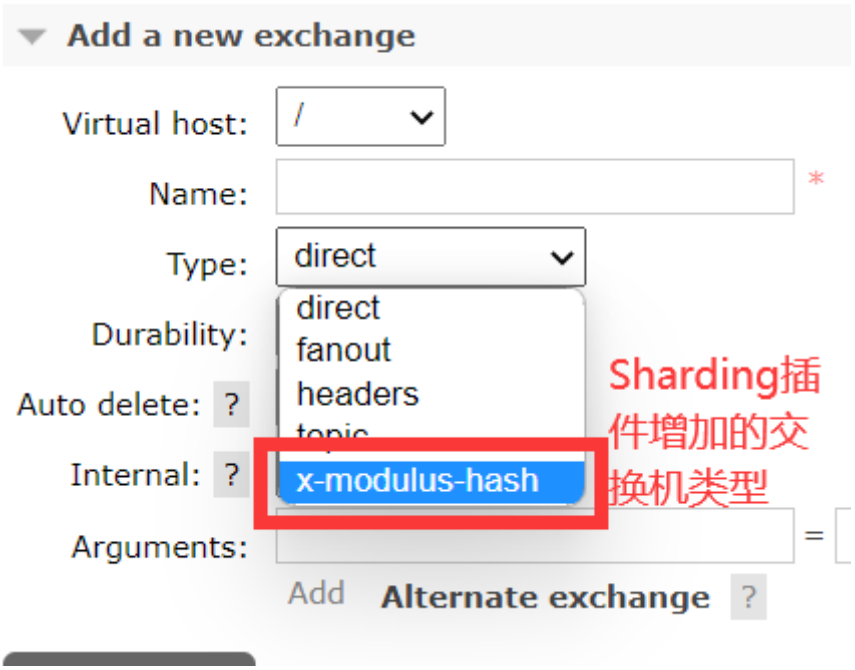


按照要求，就可以配置一个针对sharding_开头的交换机和队列的策略。



3、新增带Sharding的Exchange交换机

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

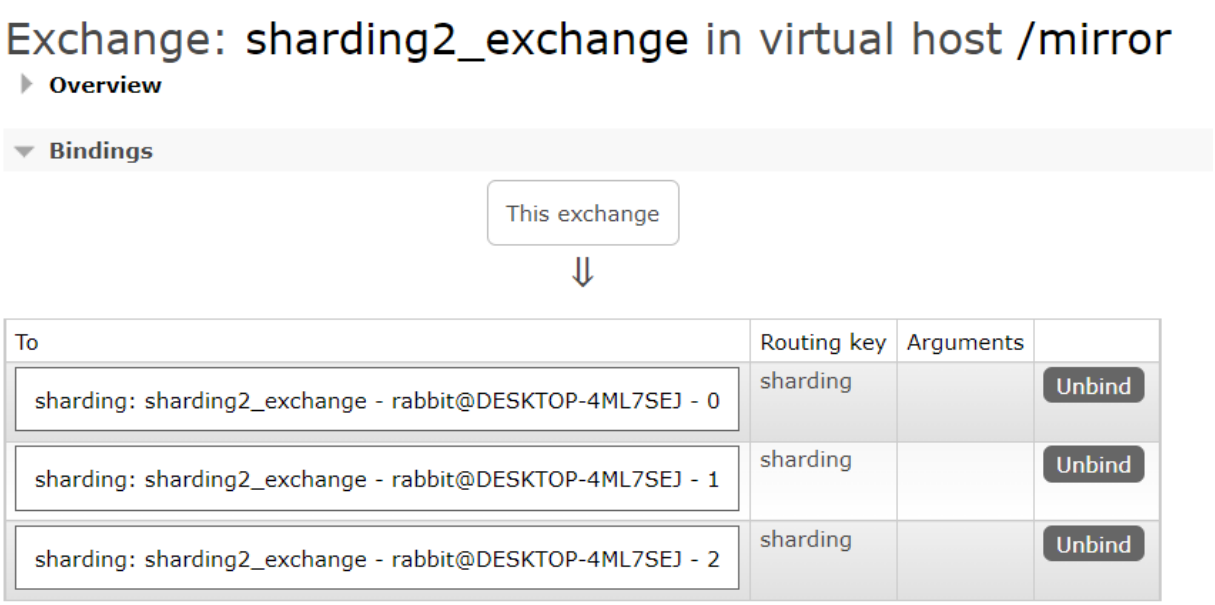


4、往分片交换机上发送消息

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 = "sharding_exchange";

    public static void main(String[] args) throws Exception{
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("192.168.65.112");
        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");
        for(int i = 0 ; i < 10000 ; i ++){
            String message = "Sharding message "+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);
            }
        };
        //三个分片要消费三次。其实相当于每一次basicConsumer去消费一个分片队列的消息。
        String consumeerFlag1 = channel.basicConsume(QUEUENAME, true, myconsumer);
        System.out.println("c1:"+consumeerFlag1);
        String consumeerFlag2 = channel.basicConsume(QUEUENAME, true, myconsumer);
        System.out.println("c2:"+consumeerFlag2);
        String consumeerFlag3 = channel.basicConsume(QUEUENAME, true, myconsumer);
        System.out.println("c3:"+consumeerFlag3);
    }
}
```

3、注意事项

使用Sharding插件后，Producer发送消息时，只需要指定虚拟Exchange，并不能确定消息最终会发往哪一个分片队列。而Sharding插件在进行消息分散存储时，虽然尽量是按照轮询的方式，均匀的保存消息。但是，这并不能保证消息就一定是均匀的。

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。**

六、章节总结

这一章节主要是在基础编程模型的基础上，了解一些RabbitMQ在面向一些具体的业务问题时提供的解决工具。这些虽然不是每个开发人员都必须要掌握的开发技巧，但是理解并熟悉这些工具，对于深入掌握RabbitMQ是必须的，同时也是深入理解MQ业务场景所必须的。但是这并不是全部。以RabbitMQ的插件举例，这里我们已经介绍了好几个RabbitMQ的插件。可是目前RabbitMQ官方提供的插件有好几十个。这每一个插件都是针对具体业务场景的一种扩展。这些都是RabbitMQ的核心精髓。任何一个技术产品，真正产生价值的不是他提供的实现方式，而是他解决实际问题的思路。这些解决具体问题的思路才是技术人员需要掌握的核心。

> 有道云笔记链接： https://note.youdao.com/s/DdGK0tRU