分布式消息服务RabbitMQ

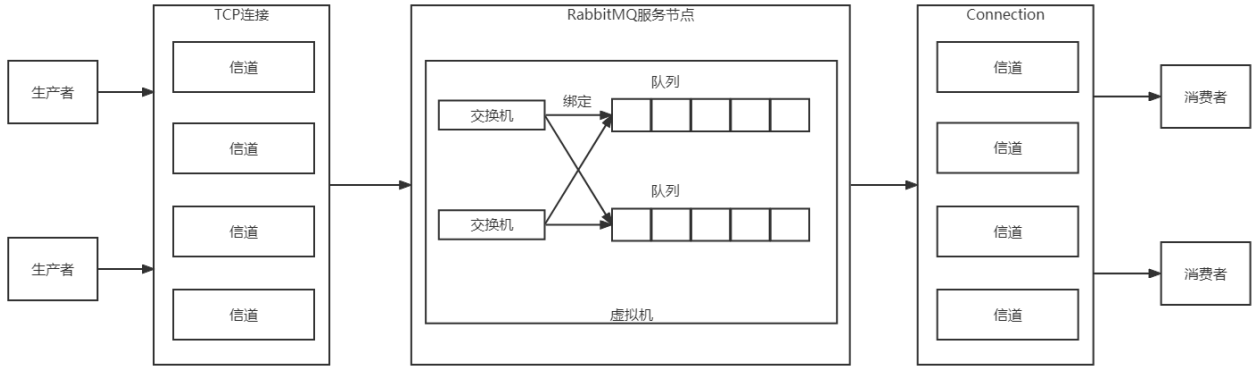
目录

产品介绍

产品定义

分布式消息服务RabbitMQ是基于高可用、分布式集群技术，完全兼容RabbitMQ开源社区，支持消息路由、事务消息、优先级队列、延迟队列、死信队列、镜像队列等功能的消息云服务。用户可开箱即用，无需部署免运维，从而实现快速上云。

产品示意图



分布式消息服务RabbitMQ发布订阅基本流程如下：

1、生产者生产的消息，通过TCP连接的信道首先发布到指定的交换机上；

2、交换机通过路由键(RoutingKey)的匹配，选择对应的队列进行投递；

3、消费者订阅队列，消费队列的消息。

核心概念

对照产品示意图，分布式消息服务RabbitMQ的核心概念如下：

- Producer：消息生产者，即消息的发布方, 生产者生产的消息，首先发布到指定的交换机上，交换机通过路由键(RoutingKey)的匹配，选择对应的队列进行投递。消费者订阅队列，消费队列的消息。
- Connection：客户端与Broker间的TCP连接。
- Channel：信道，每个连接采用多路复用，包含多个信道。Producer与Broker间采用信道传递数据。
- Broker：RabbitMQ服务节点，集群由多个节点组成。
- Vhost：虚拟机，一个节点下包含多个Vhost，Vhost间的Exchange，Queue相互隔离。就好比一台物理机上(Broker)部署多台虚拟机(Vhost)，虚拟机采用不同的用户名密码登录，实现多租户。
- Exchange：交换机，消息首先会传递到交换机，由交换机匹配路由键(RoutingKey)决定投递到哪个Queue。
- Queue：队列，存储消息的数据结构。类比小区的快递柜。

- Binding：绑定，交换机与队列间通过路由键(RoutingKey)进行绑定起来。
- Consumer，消费者，即消息的接受方。

更多信息请参见 名词解释 。

开源对比

相较于开源自建RabbitMQ，分布式消息服务RabbitMQ在低成本运维、堆积性、可观测性、集群可用性、安全保证、操作日志、OpenApi访问等方面更具优势。

对比项	开源自建	分布式消息服务RabbitMQ版
低成本运维	需要专业人员资源规划、部署、运维	一键开通，全托管。提供多种规格，按需使用，支持一键式节点数、磁盘存储空间和节点规格扩容。
堆积性	抗堆积性差，容易引发内存问题导致宕机	提供云原生引擎类型，支持海量消息堆积不受影响
可观测性	通过Management UI能够获取丰富的指标，但需要自建指标存储及展示的系统	提供监控告警能力，指标丰富，维度可精确到Vhost、Exchange和Queue，便于您快速发现和定位问题
集群可用性	需配套解决集群多节点负载均衡设置	集群内置负载均衡，支持跨AZ部署
安全保证	需要自行进行安全加固	VPC隔离，支持SSL通道加密
操作日志	无，需要自己开发	可追溯租户管理操作的记录
OpenApi访问	无，需要自己开发	提供简单的实例管理RESTFul API，使用门槛低

功能特性

分布式消息服务RabbitMQ的功能特性主要体现在以下几个方面：

- 访问接口

支持通过API调用，提供交换器、队列增删查改等管控工作。管理控制台上进行的操作用于对交换器、队列、用户、策略等增删查改等管控工作。

- 队列能力

（1）优先级队列：相比低优先级的消息，要优先投递给消费者进行处理。

（2）延迟队列：延时消息，实现秒级精准定时；简单易用，在代码上只需一个参数设置即可完成，解决开源RabbitMQ无延时队列的痛点。

（3）死信队列：支持被拒绝消息、TTL 过期消息、队列达到最大长度（消息队列 AMQP队列长度无上限）等 3 种类型消息自动进入死信队列的能力，确保消息不丢失。

- 消息能力

（1）广播消息：在同一个消费组内对所有消费者投递相同消息。

（2）事务消息：支持事务消息，可用于分布式应用。

（3）定时消息：支持消息延迟发送，解决开源 RabbitMQ 无延时队列的痛点。

- 安全防护

可追溯租户管理操作的记录。 起源于金融系统，支持权限控制和SSL协议。

- 运维监控

提供集群、交换器、队列的管理；集群、信道、连接、交换器、队列多维度指标监控。

更多信息请参见 功能特性 。

应用场景

分布式消息服务RabbitMQ适用于电子商务、金融服务、电信、物流和供应链管理、社交媒体、游戏开发、科学和研究领域等行业，通常用于业务的应用解耦、错峰流控与流量削峰、异步通信等场景。更多信息请参见 应用场景 。

使用限制

分布式消息服务RabbitMQ对连接数、通道数等信息进行限制，使用时注意不要超过限制，以免程序出现异常。更多信息请参见 使用限制 。

产品优势

分布式消息服务RabbitMQ的产品优势主要包括以下几个方面：

高可用性

分布式消息服务RabbitMQ主要通过以下三种方式保证服务的连续性和可靠性：

支持生产消费自动负载均衡

- 多个消费者：在RabbitMQ中，可以创建多个消费者来同时消费同一个队列中的消息。RabbitMQ会将消息平均地分配给这些消费者，从而实现负载均衡。
- 发布/订阅模式：使用RabbitMQ的发布/订阅模式可以实现自动负载均衡。在这种模式下，生产者将消息发送到交换机，然后交换机将消息广播给绑定到它的所有队列。每个队列上都可以有多个消费者，它们会共享接收到的消息负载。
- 优先级队列：RabbitMQ支持创建优先级队列，可以根据消息的优先级进行排序和传递。通过使用优先级队列，可以确保高优先级的消息能够更快地被处理，从而实现负载均衡。

lvs节点故障时的自动主备切换

- 配置镜像队列：首先，你需要在RabbitMQ集群的各个节点之间配置镜像队列。镜像队列会将消息复制到备用节点上，在主节点发生故障时，备用节点会接管主节点的工作。
- 心跳检测：RabbitMQ会通过心跳机制监测节点之间的连接状态。如果主节点无法进行正常通信（比如网络故障或节点崩溃），备用节点会被选举为新的主节点。

- 节点选举：当主节点无法通信时，RabbitMQ集群中的其他节点会开始进行选举，选择一个备用节点来替代主节点。选举过程中，节点会相互通信并比较彼此的状态和能力。最终，一个节点会被选举为新的主节点。
- 自动切换：一旦新的主节点被选举出来，RabbitMQ集群就会自动切换到新的主节点，并开始处理消息。客户端可以通过相同的连接重新连接到新的主节点，并继续发送和接收消息。

镜像队列安全备份

- 数据复制: RabbitMQ的镜像队列通过在多个节点之间复制队列的消息来提供冗余备份。每个节点都维护自己的完整队列副本，这样当一个节点发生故障时，其他节点将能够接管并继续处理消息。
- 同步复制: 镜像队列支持同步复制和异步复制两种模式。在同步复制模式下，所有的写入操作都会等待所有镜像节点都完成相同的操作，这样可以确保数据的一致性。但是这会增加写入操作的延迟。
- 容错机制: 当一个节点失效时，RabbitMQ会自动将该节点上的队列重新分配给其他正常工作的节点。这个过程是自动的，无需人工干预。因此，即使整个节点失效，消息也不会丢失。
- 高可用性: 镜像队列提供了高可用性的保证。如果一个节点故障，其他节点将自动接管该节点上的队列并继续处理消息，从而确保系统的可靠性和可用性。
- 故障恢复: 当一个节点失效后重新启动时，RabbitMQ会自动将该节点上的队列重新同步到新的镜像节点上，以确保数据的完整性和一致性。

高安全性

起源于金融系统，分布式消息服务RabbitMQ主要支持权限控制和SSL协议实现高安全性。

支持权限控制

- 用户（User）：在RabbitMQ中，用户用于标识连接RabbitMQ的客户端。每个用户都分配了一个用户名和密码。
- 虚拟主机（Virtual Host）：虚拟主机是RabbitMQ中的逻辑隔离单位，用于将不同的应用程序或服务隔离开。每个虚拟主机都有一个名称，并且可以有不同的权限设置。
- 权限（Permission）：权限定义了对RabbitMQ资源的操作权限。这些资源可以是交换机、队列、绑定等。权限包括读写、发布、接收等操作。
- 角色（Role）：角色是一组权限的集合。通过为用户分配角色，可以简化权限管理。

支持SSL协议

- SSL协议：SSL（Secure Sockets Layer）是一种用于安全传输数据的协议，其目标是通过使用加密技术来保护网络通信的安全性。
- RabbitMQ的SSL支持：RabbitMQ可以配置为使用SSL协议来保护数据传输。它支持使用自签名证书或由受信任的证书颁发机构（CA）签名的证书。
- 配置SSL证书：要在RabbitMQ中启用SSL协议，需要为服务器和客户端生成SSL证书。可以使用openssl工具生成自签名证书或请求一个CA签名的证书。
- RabbitMQ服务器配置：在RabbitMQ服务器上，需要在配置文件中指定SSL相关的参数，如证书路径、私钥路径、密码等。还可以配置是否要求客户端验证证书以及是否启用TLS版本的选择。
- 客户端配置：在连接RabbitMQ服务器之前，客户端也需要配置SSL相关的参数，包括证书路径、私钥路径、密码等。客户端还可以配置是否验证服务器的证书。
- 客户端连接：一旦RabbitMQ服务器和客户端都配置好了SSL相关的参数，客户端可以使用SSL连接RabbitMQ服务器。客户端通过指定SSL选项来建立SSL连接。
- 数据传输加密：通过SSL协议建立的连接可以保证数据传输的安全性。所有通过SSL连接发送和接收的数据都将通过加密算法进行加密和解密，以防止中间人攻击和数据泄漏。

高可靠性

分布式消息服务RabbitMQ使用了持久化、传输确认、发布确认等机制来保证可靠性。

持久化

- 消息持久化：默认情况下，RabbitMQ将消息存储在内存中，这意味着如果RabbitMQ服务器关闭或发生故障，未被消费的消息将会丢失。为了解决这个问题，可以将消息设置为持久化的。当消息被标记为持久化时，RabbitMQ会将消息存储到磁盘上的文件中，以确保消息在服务器重启后仍然可用。
- 队列持久化：除了消息持久化外，还可以将队列标记为持久化。当队列被标记为持久化时，RabbitMQ会将队列的元数据和消息都存储到磁盘上的文件中。这样，在服务器重启后，队列和其中的消息将会被重新创建。
- 交换机持久化：交换机也可以被标记为持久化。当交换机被标记为持久化时，RabbitMQ会将交换机的元数据存储到磁盘上的文件中。然而，交换机本身并不存储消息，因此即使交换机持久化了，如果没有持久化的队列与其绑定，未被消费的消息仍然会丢失。
- 持久化模式：在RabbitMQ中，可以选择将消息和队列都标记为持久化，或者只将其中一项标记为持久化。通过将消息和队列都标记为持久化，可以最大程度地确保消息的安全性。但是，需要注意的是，将所有消息都持久化可能会导致性能下降。

传输确认

- 发布确认模式（Publish Confirm）：当生产者发送消息给RabbitMQ后，可以通过设置“Confirm模式”来确保消息已被RabbitMQ接收和处理。可以使用事务或者确认模式来实现。
- 事务模式：在事务模式下，生产者可以将消息发送到RabbitMQ，并进行事务提交操作。如果发送成功，则事务会被提交，消息会被RabbitMQ接收和处理，否则事务会被回滚。
- 确认模式：在确认模式下，生产者将消息发送到RabbitMQ，并等待RabbitMQ发送一个确认消息给生产者来表示该消息已经被成功接收和处理。
- 多个消息的批量确认：可以以批量的方式进行消息的确认，即同时确认多个消息，而不是逐个确认。
- 异步确认：可以使用异步的方式进行确认，即发送消息后不需要等待确认消息的返回，而是通过回调函数来处理确认结果。
- 应答超时设置：为了避免消息发送失败或者长时间无响应造成的阻塞，可以设置应答超时时间，超过该时间还未收到确认消息，则认为消息发送失败。

发布确认

RabbitMQ发布确认机制是一个确保消息在发送到队列之后被成功接收的机制。它提供了两种发布确认模式：简单模式和批量模式。

- 简单模式：

在发送每条消息之后，生产者会等待来自RabbitMQ的确认回复。

如果RabbitMQ成功接收到消息并将其存储在队列中，它会返回一个确认回复给生产者。

如果由于某种原因消息未能成功发送到队列，RabbitMQ将返回一个拒绝回复给生产者。

生产者可以根据具体情况来处理确认和拒绝回复。

- 批量模式：

在发送一批消息后，生产者会等待来自RabbitMQ的一个确认回复。

如果RabbitMQ成功接收到批量消息并将它们存储在队列中，它会返回一个确认回复给生产者。

如果其中任何一条消息未能成功发送到队列，RabbitMQ将返回一个拒绝回复给生产者，并且不会存储整个批量消息。

生产者可以根据具体情况来处理确认和拒绝回复。

开箱即用

分布式消息服务RabbitMQ支持一键部署，用户可开箱即用，无需部署免运维，从而实现快速上云。

功能特性

消息收发

- 消息发送：支持指定同步发送、异步发送；支持身份认证，客户端连接RabbitMQ服务端时使用SSL或账号密码验证；支持广播发送；支持发送持久化消息、非持久化消息。
- 消息消费：采用pull方式和push方式消费，多消费者间轮询消费。

消息发送可靠性

RabbitMQ消息发送可靠性有两个方法，一个是事务，另一个是Confirm机制。事务能够解决Producer与Broker之间消息确认的问题，只有消息成功被Broker接受，事务提交才能成功，但会降低RabbitMQ的性能，为此RabbitMQ提供了一种低消耗的事务管理方式，将Channel设置成Confirm模式。Confirm模式的Channel，通过该Channel发出的消息会生成一个唯一的有序ID（从1开始），一旦消息成功发送到相应的队列之后，RabbitMQ服务端会发送给生产者一个确认标志，包含消息的ID，这样生产者就知道该消息已经发送成功了。Confirm支持普通发送方确认模式、批量确认模式、异步监听发送方确认模式，可以满足发送可靠性和高性能需求。

消息确认

为了保证消息从队列可靠地到达消费者，RabbitMQ提供了消息确认机制。

- 消费者订阅队列的时候，可以指定autoAck参数，当autoAck为true的时候，RabbitMQ采用自动确认模式，RabbitMQ自动把发送出去的消息设置为确认，然后从内存或者硬盘中删除，而不管消费者是否真正消费到了这些消息。
- 当autoAck为false的时候，RabbitMQ会等待消费者回复的确认信号，收到确认信号之后才从内存或者磁盘中删除消息。

消息确认机制是RabbitMQ消息可靠性投递的基础，只要设置autoAck参数为false，消费者就有足够的时间处理消息，不用担心处理消息的过程中消费者进程挂掉后消息丢失的问题。

消息TTL

Time To Live，也就是生存时间，是一条消息在队列中的最大存活时间，单位是毫秒。

- RabbitMQ可以对消息和队列设置TTL。

- RabbitMQ支持设置消息的过期时间，在消息发送的时候可以进行指定，每条消息的过期时间可以不同。
- RabbitMQ支持设置队列的过期时间，从消息入队列开始计算，直到超过了队列的超时时间配置，那么消息会变成死信，自动清除。
- 如果两种方式一起使用，则过期时间以两者中较小的那个数值为准。
- 当然也可以不设置TTL，不设置表示消息不会过期；如果设置为0，则表示除非此时可以直接将消息投递到消费者，否则该消息将被立即丢弃。

持久化

RabbitMQ的持久化分为三个部分：交换器持久化、队列持久化和消息的持久化。

- 交换器持久化可以通过在声明队列时将durable参数设置为true。如果交换器不设置持久化，那么在RabbitMQ服务重启之后，相关的交换器元数据会丢失，不过消息不会丢失，只是不能将消息发送到这个交换器了。
- 队列的持久化能保证其本身的元数据不会因异常情况而丢失，但是不能保证内部所存储的消息不会丢失。要确保消息不会丢失，需要将其设置为持久化。队列的持久化可以通过在声明队列时将durable参数设置为true。
- 设置了队列和消息的持久化，当RabbitMQ服务重启之后，消息依然存在。如果只设置队列持久化或者消息持久化，重启之后消息都会消失。

对于可靠性不是那么高的消息可以不采用持久化处理以提高整体的吞吐量。在实际中，需要根据实际情况在可靠性和吞吐量之间做一个权衡。

延迟队列

一般的队列，消息一旦进入队列就会被消费者立即消费。延迟队列就是进入该队列的消息会被消费者延迟消费，延迟队列中存储的对象是的延迟消息，“延迟消息”是指当消息被发送以后，等待特定的时间后，消费者才能拿到这个消息进行消费。RabbitMQ提供TTL配合死信队列和延时队列插件两种方式来满足延时消息业务场景。

死信队列

当消息在一个队列中变成死信之后，他能被重新发送到另一个交换器中，这个交换器成为死信交换器，与该交换器绑定的队列称为死信队列。消息变成死信有下面几种情况：

- 消息被拒绝
- 消息过期
- 队列达到最大长度

DLX也是一个正常的交换器，和一般的交换器没有区别，他能在任何的队列上面被指定，实际上就是设置某个队列的属性。当这个队列中有死信的时候，RabbitMQ会自动将这个消息重新发送到设置的交换器上，进而被路由到另一个队列。当发生异常的时候，消息不能够被消费者正常消费，被加入到了死信队列中。后续的程序可以根据死信队列中的内容分析当时发生的异常，进而改善和优化系统。

优先级队列

优先级队列是指优先级高的消息往往放在队列的head头部，相比低优先级的消息，要优先投递给消费者进行处理。

在RabbitMQ中，我们可以设置消息的优先级，在发送消息时指定消息的优先级，可以分为10个级别，级别越高优先级越大。当多个消息拥有相同的优先级时，它们按照FIFO的顺序排序。

镜像队列

Rabbitmq支持镜像队列，队列的多个副本保存在不同实例节点。队列分为主备角色，对某个Queue来说，只有master对外提供服务，而其他slave只提供备份服务，在master所在节点不可用时，选出一个slave作为新的master继续对外提供服务。无论客户端的请求打到master还是slave最终数据都是从master节点获取。

- 当请求打到master节点时，master节点直接将消息返回给client，同时master节点会通过GM（Guaranteed Multicast）协议将Queue的最新状态广播到slave节点。GM保证了广播消息的原子性，即要么都更新要么都不更新。
- 当请求打到slave节点时，slave节点需要将请求先重定向到master节点，master节点将将消息返回给client，同时master节点会通过GM协议将queue的最新状态广播到slave节点。

应用场景

行业应用

RabbitMQ在需要高效、可靠的消息传递和处理的任何行业都有广泛的应用，常见行业及其实际业务场景如下。

- 电子商务：RabbitMQ可用于处理订单和库存管理，处理支付通知及其它与物流相关的消息。
- 金融服务：RabbitMQ可以用于处理实时交易数据、通知和报价，并支持金融机构之间的异步通信。
- 电信：RabbitMQ可用于处理电话呼叫记录、短信和多媒体消息的分发等。
- 物流和供应链管理：RabbitMQ可以用于跟踪货物的位置和状态，以及协调供应链中各个环节的消息传递。
- 社交媒体：RabbitMQ可以用于实现实时消息推送、聊天和通知功能，以及处理用户生成内容。
- 游戏开发：RabbitMQ可用于处理游戏中的多人互动、玩家间的消息传递和协作。
- 科学和研究领域：RabbitMQ可以用于分布式计算、任务队列和数据流处理。

RabbitMQ通常用于业务的应用解耦、错峰流控与流量削峰和异步通信场景。

应用解耦

RabbitMQ可以将应用程序之间的耦合度降低，使得系统更加灵活和可扩展。以下是一些使用RabbitMQ实现应用解耦的举例：

- 订单和库存管理系统：假设有一个在线商店，订单系统负责接收和处理用户的订单，而库存管理系统则负责跟踪库存并更新库存状态。通过使用RabbitMQ，订单系统可以将订单信息发送到一个名

为"order_queue"的消息队列中，而库存管理系统则监听该队列，并在收到订单消息时进行库存更新。这样，订单系统和库存管理系统可以解耦，并且可以独立地进行扩展和维护。

- 日志处理系统：在一个大规模的分布式系统中，各个服务都会生成大量的日志信息。为了对这些日志进行集中管理和分析，可以使用RabbitMQ作为日志消息的中间代理。每个服务将其产生的日志消息发送到RabbitMQ的一个名为"log_queue"的消息队列中，然后日志处理系统从该队列中消费日志消息，并进行相应的处理和存储。这样，每个服务的日志处理可以独立进行，互不影响。
- 流量控制系统：在一个微服务架构中，可能会有多个服务实例同时运行，当某个服务实例过载时，传统的负载均衡器无法有效地控制流量分发。通过使用RabbitMQ，可以实现基于消息的流量控制。每个服务实例将其当前的负载情况发送到一个名为"load_queue"的消息队列中，一个负载控制器订阅该队列，并根据各个服务实例的负载情况来动态调整流量分发。这样，流量控制仍然可以在应用层面进行解耦。

错峰流控与流量削峰

RabbitMQ 是一个开源的消息代理，它支持多种不同的消息协议，并提供了强大的消息传递功能。在 RabbitMQ 中，可以通过错峰流控和流量削峰来管理消息传递的速率和负载。

- 错峰流控： 错峰流控是指通过限制消息的传递速率，使得消息的发送和接收能够在一个可控制的范围内进行。例如，假设有一个消息队列，每秒最多只能处理1000条消息，当消息的发送速率超过此限制时，消息将被暂时缓存起来，直到消息队列可以处理更多的消息为止。
- 流量削峰： 流量削峰是指通过调整消息传递的速率，使得消息的发送和接收能够适应系统的负载情况。例如，在高峰期间，系统的负载可能会非常高，如果继续以相同的速率发送消息，可能会导致系统崩溃。因此，通过流量削峰，可以减少消息的发送速率，避免系统过载。

举例来说，假设有一个在线购物网站，用户在下单后需要生成订单消息并发送到 RabbitMQ 中。为了避免订单量过大导致系统崩溃，可以采取如下策略：

- 错峰流控：设置 RabbitMQ 的每秒最大处理订单数为1000，当用户下单速度超过1000单/秒时，系统将对订单进行暂时缓存，直到系统可以处理更多订单。
- 流量削峰：在特定时间段（例如促销活动期间），预计订单量会非常高，为了避免系统过载，可以调整 RabbitMQ 的每秒最大处理订单数为500，即降低消息的发送速率，使得系统能够更好地处理订单。

通过这种方式，可以在保证系统稳定性的同时，有效地管理消息传递的速率和负载。

异步通信

RabbitMQ是一个消息中间件，可以用于异步通信。在用户注册场景中，RabbitMQ可以用于发送和接收注册相关的消息。

下面是一个异步用户注册流程：

1. 用户提交注册表单。
2. 服务器接收到注册请求后，将用户提交的数据写入数据库，并生成一个唯一的用户ID。
3. 服务器将用户ID封装成一个消息，发送到RabbitMQ的注册队列中。
4. 注册队列中的消息被一个或多个消费者监听。
5. 消费者接收到注册消息后，执行注册相关的逻辑，比如发送确认邮件、生成用户账号等操作。
6. 在完成注册逻辑后，消费者可以将结果封装成一个消息，发送到另一个队列中，比如发送注册成功通知给用户。
7. 另一个队列中的消息可以由一个或多个消费者监听，负责处理注册成功通知的逻辑。

使用RabbitMQ实现异步用户注册可以带来以下好处：

- 解耦：通过将注册请求和注册逻辑解耦，使得系统组件之间的依赖减少。
- 异步处理：用户不需要等待注册逻辑完成，而是可以立即返回一个成功的响应。真正的注册逻辑可以在后台完成，提高系统的并发能力。
- 可伸缩性：通过添加更多的消费者来处理注册消息，可以轻松地提高系统的吞吐量。
- 可靠性：RabbitMQ具备持久化消息的能力，即使在发生故障时也不会丢失消息。

需要注意的是，使用RabbitMQ进行异步通信需要对消息的可靠性进行处理，比如使用事务或者消息确认机制，以确保消息能够被成功处理。

高可用

RabbitMQ提供了多种方式来实现高可用性，确保消息队列的稳定和可靠性。以下是一些常用的方法：

1. 集群模式：RabbitMQ支持将多个节点组成集群，通过在多个节点之间复制消息队列和交换器，实现数据的冗余和高可用性。当一个节点发生故障时，其他节点可以接管其工作，确保消息的连续传递。
2. 镜像队列：RabbitMQ的镜像队列功能可以将队列的数据在多个节点之间进行复制。这样，当一个节点发生故障时，其他节点上的镜像队列可以继续提供服务，确保消息的可靠传递。
3. 自动化故障转移：RabbitMQ支持自动故障转移，当一个节点发生故障时，可以自动将其上的队列和交换器迁移到其他正常的节点上，确保消息的连续处理。
4. 心跳机制：RabbitMQ通过心跳机制来监测节点的健康状态。当一个节点长时间没有响应时，其他节点可以将其标记为不可用，并进行故障转移。
5. 负载均衡：通过在多个节点之间分配消息的处理负载，可以提高系统的吞吐量和可用性。RabbitMQ支持多种负载均衡算法，如轮询、随机等。
6. 数据备份和恢复：为了防止数据丢失，可以定期备份RabbitMQ的数据，并在节点故障时进行数据恢复。

通过以上的方法，可以实现RabbitMQ的高可用性，确保消息队列的稳定运行和数据的可靠传递。但是需要注意的是，高可用性的实现需要根据具体的需求和场景进行配置和调优。

产品选型

特性	Kafka	RabbitMQ	RocketMQ
功能	支持功能较少，不支持延迟发送，消息重试等功能	功能丰富，支持多个队列种类（优先级队列、延迟队列、死信队列镜像队列等），提供丰富的策略分配	功能完善，支持事务消息、定时消息、事务消息等
单机吞吐量	十万级	万级	几万级
稳定性	队列/分区多时性能不稳定	消息堆积时，性能不稳定	队列较多、消息堆积时性能保持稳定
可用性	非常高（分布式）具有主备故障自动切换	较高，基于主从架构实现高可用性	非常高（分布式）具有主备故障自动切换

特性	Kafka	RabbitMQ	RocketMQ
选型建议	性能要求高，数据量大，适合产生大量数据的互联网服务的数据收集业务，如日志采集处理、需对接大数据应用等，kafka是首选	数据量少，吞吐量需求不大；数据可靠性要求较高，对功能丰富性要求极高	可靠性要求很高且性能要求较高的场景以及业务削峰场景，如电商、订单处理等

产品规格

（一）以下规格选型适用于14个节点

华东1、华北2、西南1、华南2、上海36、青岛20、长沙42、南昌5、武汉41、杭州7、西南2-贵州、太原4、郑州5、西安7、呼和浩特3

注意

- 通用型规格已调整为白名单特性，如需了解该规格参数请联系技术支持。
- 云原生引擎已调整为白名单特性，如需了解该引擎请联系技术支持。
- 单机版实例面向用户体验和业务测试场景，无法保证性能和高可用。如果需要在生产环境使用RabbitMQ实例，建议购买集群版实例。
- TPS参考值，是指以2K大小的消息为例的每秒处理消息条数，测试场景为不开启持久化的非镜像队列，实时生产实时消费，队列无积压。此数据仅供参考，在实际业务中，队列数、消息堆积、连接数、channel、消费者数、镜像队列、优先级队列、消息持久化和exchange类型等因素会对TPS造成较大影响，不一定能够达到上述性能。

1、RabbitMQ实例规格

RabbitMQ兼容开源RabbitMQ 3.8版本，提供集群和单机两种部署架构实例，支持X86和ARM计算，主机类型为Intel、海光和鲲鹏计算增强型，节点可选择3节点、5节点、7节点、9节点。

存储类型可选择高IO（SAS）、超高IO（SSD），默认选择超高IO。

2、目前所有的可选规格如下：

集群版

Intel计算增强型

引擎类型	规格	节点数	单个代理最大消费者数	单个代理最大连接数	单个代理建议队列数	TPS参考值
RabbitMQ	2C4G	3	4000	1000	100	3000
		5	4000	1000	100	5000
		7	4000	1000	100	7000
		9	4000	1000	100	9000

	4C8G	3	8000	2000	200	6000
		5	8000	2000	200	10000
		7	8000	2000	200	14000
		9	8000	2000	200	18000
	8C16G	3	16000	4000	400	12000
		5	16000	4000	400	20000
		7	16000	4000	400	28000
		9	16000	4000	400	36000
	12C24G	3	24000	6000	600	24000
		5	24000	6000	600	40000
		7	24000	6000	600	56000
		9	24000	6000	600	72000
	16C32G	3	32000	8000	800	48000
		5	32000	8000	800	80000
		7	32000	8000	800	112000
		9	32000	8000	800	144000
	24C48G	3	40000	10000	1000	60000
		5	40000	10000	1000	100000
		7	40000	10000	1000	140000
		9	40000	10000	1000	180000
	32C64G	3	40000	10000	1000	72000
		5	40000	10000	1000	120000
		7	40000	10000	1000	168000
		9	40000	10000	1000	216000

海光计算增强型

引擎类型	规格	节点数	单个代理最大消费者数	单个代理最大连接数	单个代理建议队列数	TPS参考值
RabbitMQ	2C4G	3	4000	1000	100	1500
		5	4000	1000	100	2500
		7	4000	1000	100	3500
		9	4000	1000	100	4500
	4C8G	3	8000	2000	200	3000
		5	8000	2000	200	5000
		7	8000	2000	200	7000

		9	8000	2000	200	9000
	8C16G	3	16000	4000	400	6000
		5	16000	4000	400	10000
		7	16000	4000	400	14000
		9	16000	4000	400	18000
	12C24G	3	24000	6000	600	12000
		5	24000	6000	600	20000
		7	24000	6000	600	28000
		9	24000	6000	600	36000
	16C32G	3	32000	8000	800	30000
		5	32000	8000	800	50000
		7	32000	8000	800	70000
		9	32000	8000	800	90000
	24C32G	3	40000	10000	1000	36000
		5	40000	10000	1000	60000
		7	40000	10000	1000	84000
		9	40000	10000	1000	108000
	32C64G	3	40000	10000	1000	45000
		5	40000	10000	1000	75000
		7	40000	10000	1000	105000
		9	40000	10000	1000	135000

鲲鹏计算增强型

引擎类型	规格	节点数	单个代理最大消费者数	单个代理最大连接数	单个代理建议队列数	TPS参考值
RabbitMQ	2C4G	3	4000	1000	100	2400
		5	4000	1000	100	4000
		7	4000	1000	100	5600
		9	4000	1000	100	7200
	4C8G	3	8000	2000	200	4500
		5	8000	2000	200	7500
		7	8000	2000	200	10500
		9	8000	2000	200	13500
	8C16G	3	16000	4000	400	9000
		5	16000	4000	400	15000

		7	16000	4000	400	21000
		9	16000	4000	400	27000
	12C24G	3	24000	6000	600	21000
		5	24000	6000	600	35000
		7	24000	6000	600	49000
		9	24000	6000	600	63000
	16C32G	3	32000	8000	800	45000
		5	32000	8000	800	75000
		7	32000	8000	800	105000
		9	32000	8000	800	135000
	24C32G	3	40000	10000	1000	54000
		5	40000	10000	1000	90000
		7	40000	10000	1000	126000
		9	40000	10000	1000	162000
	32C64G	3	40000	10000	1000	60000
		5	40000	10000	1000	100000
		7	40000	10000	1000	140000
		9	40000	10000	1000	180000

单机版

Intel计算增强型

引擎类型	规格	单个代理最大消费者数	单个代理最大连接数	单个代理建议队列数	TPS参考值
RabbitMQ	2C4G	4000	1000	100	1000
	4C8G	8000	2000	200	2000
	8C16G	16000	4000	400	4000
	16C32G	32000	8000	800	16000
	24C48G	40000	10000	1000	20000

海光计算增强型

引擎类型	规格	单个代理最大消费者数	单个代理最大连接数	单个代理建议队列数	TPS参考值
RabbitMQ	2C4G	4000	1000	100	500
	4C8G	8000	2000	200	1000

	8C16G	16000	4000	400	2000
	16C32G	32000	8000	800	10000
	24C48G	40000	10000	1000	12000

鲲鹏计算增强型

引擎类型	规格	单个代理最大消费者数	单个代理最大连接数	单个代理建议队列数	TPS参考值
RabbitMQ	2C4G	4000	1000	100	800
	4C8G	8000	2000	200	1500
	8C16G	16000	4000	400	3000
	16C32G	32000	8000	800	15000
	24C48G	40000	10000	1000	18000

（二）以下规格选型适用于6个节点

芜湖2、南京3、上海7、福州25、重庆2、北京5。

1、实例规格

分布式消息服务RabbitMQ兼容开源RabbitMQ 3.8.9，实例规格分为基础版，高级版。具体的规格分为基础版3节点4C8G，高级版3节点8C16G。

存储类型为普通I/O。

2、目前基础版和高级版规格如下：

实例规格	节点	参考性能（TPS）	建议队列数	最大连接数
4核 8GB	3	45000	600	3000*3
8核16GB	3	85000	1200	5000*3

安全方案

安全价值

RabbitMQ的安全性对用户具有重要的价值，以下是一些方面的价值：

1. 数据保护：RabbitMQ的安全性可以保护用户的消息和数据免受未经授权的访问和篡改。通过合适的安全配置，可以确保消息在传输和存储过程中的机密性和完整性。
2. 身份认证和授权：RabbitMQ的安全性可以确保只有经过身份验证的用户可以访问和操作消息队列。通过身份认证和授权机制，可以限制用户的访问权限，防止未经授权的访问和操作。
3. 防止拒绝服务攻击：RabbitMQ的安全性可以帮助用户防止拒绝服务（DoS）攻击。通过限制资源的使用和设置合适的配额，可以防止恶意用户或程序对消息队列进行过载和耗尽资源的攻击。
4. 监控和审计：RabbitMQ的安全性可以提供监控和审计功能，帮助用户跟踪和记录对消息队列的访问和操作。这可以帮助用户发现异常行为和安全事件，并进行及时的响应和调查。

5. 合规性要求：对于一些行业和法规，如金融、医疗等，安全性是必要的要求。RabbitMQ的安全性可以帮助用户满足合规性要求，确保消息和数据的保密性和完整性。

综上所述，RabbitMQ的安全性对用户具有重要的价值，可以保护用户的数据和消息的安全，防止未经授权的访问和篡改，防止拒绝服务攻击，并满足合规性要求。

身份认证

- CTIAM

  统一身份认证（Identity and Access Management， 简称：CTIAM）是提供用户进行权限管理的基础服务，可以帮助您安全的控制天翼云服务和资源的访问及操作权限，包括：用户身份认证、权限分配、访问控制等功能。具体介绍请参考 统一身份认证-产品介绍 。

  您可以创建IAM用户，并为其设置关联分布式消息服务RabbitMQ实例权限，该用户就可以通过用户名和密码访问授权的实例资源。具体请参见 统一身份认证-快速入门-创建IAM用户 。

访问控制

- 权限控制

  购买分布式消息服务RabbitMQ实例之后，您可以使用CTIAM为企业中的员工设置不同的访问权限，以达到不同员工之间的权限隔离，通过CTIAM进行精细的权限管理。
- VPC和子网

  虚拟私有云（Virtual Private Cloud，VPC）为分布式消息服务RabbitMQ构建隔离、私密的虚拟网络环境，提升数据库的安全性，并简化用户的网络部署。您可以完全掌控自己的专有网络，VPC丰富的功能帮助您灵活管理云上网络，包括创建子网、设置安全组和网络ACL、管理路由表、申请弹性公网IP和带宽等。通过子网与其他网络隔离，独享网络资源，提高网络安全性。具体内容请参见 虚拟私有云-用户指南-创建虚拟私有云和子网 。
- 安全组

  安全组是一个逻辑上的分组，可以为同一个虚拟私有云内具有相同安全保护需求并相互信任的RabbitMQ实例提供相同的访问策略。您可以通过为数据库实例设置安全组，开通需访问RabbitMQ实例的IP地址和端口，来保证保障其运行环境的安全性和稳定性。具体请参见 修改实例安全组 。

数据保护技术

RabbitMQ提供了多种数据保护技术，以确保数据在传输和存储过程中的机密性和完整性。

1. 跨AZ容灾：在不同的可用区部署多个RabbitMQ节点，确保节点之间的数据复制和同步。这样，当一个可用区发生故障时，其他可用区上的节点可以继续提供服务。
2. 副本冗余：RabbitMQ副本冗余是一种保证消息队列的高可用性和数据冗余的策略。通过在多个节点上创建副本，可以确保即使一个节点发生故障，其他节点上的副本仍然可以提供服务。
3. 数据持久化：RabbitMQ数据持久化是通过将队列、消息和交换器进行持久化，确保消息队列中的数据在节点重启或故障时不会丢失。

服务韧性

RabbitMQ服务的韧性是指其在面对各种故障和异常情况时能够保持可用性和可靠性的能力。以下是保障RabbitMQ服务韧性的关键方面：

1. AZ内实例容灾：在不同的可用区内部署多个RabbitMQ节点，使它们能够相互复制和同步消息。这样即使一个可用区发生故障，其他可用区的节点仍然可以提供服务。
2. 数据容灾：RabbitMQ数据容灾是通过持久化消息、队列和交换器、备份和复制以及高可用性集群等策略，保护数据免受损失和故障影响的措施。

使用限制

限制项	约束和限制	描述
版本	3.8.35	兼容AMQP 0-9-1协议的客户端版本。
连接数	RabbitMQ单机和集群实例，不同实例规格的连接数上限不一致，具体限制，请参考产品规格。	
通道数	<= 2047	单条连接可以建立的通道数。
消息大小	50MB	请勿发送大于此长度的消息，否则生产失败。
内存高水位阈值	<=60%	内存使用率超过60%会触发内存高水位，生产者流程被阻塞。
磁盘高水位阈值	<=5GB	磁盘剩余空间低于5GB会触发磁盘高水位，生产者流程被阻塞。
cluster_partition_handling	pause_minority	当集群发生网络分区时，代理会检查自己是否处于“少数派”（存储分区的代理数小于等于总代理数的一半称为少数派）。少数派中的代理将会自动关闭服务并定期检测网络状态，待分区恢复之后重新启动服务。如果未开启镜像队列，发生分区时少数派上的队列将无法生产消费。此策略相当于放弃了可用性而选择了数据一致性。
rabbitmq_delayed_message_exchange	插件延迟时间存在1%左右的误差，可能提前或者推迟发送消息给消费者。	实例是否开启消息延迟功能。
RabbitMQ插件	RabbitMQ插件功能可用于测试和迁移业务等场景，不建议用于生产业务。因使用插件导致的可靠性问题，不在服务承诺的SLA范围内。	

名词解释

Vhost

虚拟主机（Virtual Host），类似于Namespace命名空间的概念，逻辑隔离，每个用户里可以创建多个Vhost，每个Vhost可以创建若干个Exchange和Queue。

Queue

消息队列，每个消息都会被投入到一个或者多个Queue里。

Producer

消息生产者，即投递消息的程序。

Consumer

消息消费者，即接受消息的程序。

Connection

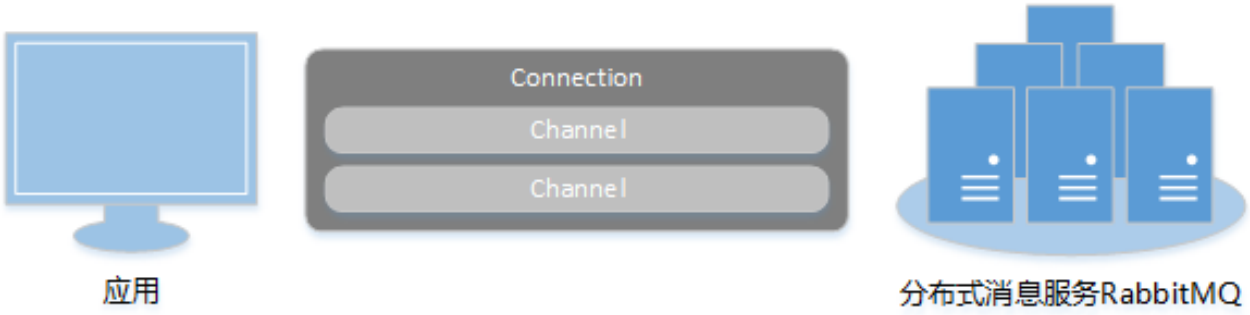
TCP 连接，Producer 或 Consumer 与消息队列间的物理 TCP 连接。

Connection将应用与分布式消息服务RabbitMQ连接在一起。Connection会执行认证、IP解析、路由等底层网络任务。应用与分布式消息服务RabbitMQ建立Connection需要多个TCP报文交互，因而会消耗较多的网络资源和分布式消息服务RabbitMQ资源。大量的Connection会对分布式消息服务RabbitMQ造成巨大压力，甚至触发分布式消息服务RabbitMQ SYN洪水攻击防护，导致分布式消息服务RabbitMQ无响应，进而影响业务。

Channel

在客户端的每个物理TCP连接里，可建立多个Channel，每个Channel代表一个会话任务。

Channel是物理TCP连接中的虚拟连接。当应用通过Connection与分布式消息服务RabbitMQ建立连接后，所有的AMQP协议操作（例如创建队列、发送消息、接收消息等）都会通过Connection中的Channel完成。Channel可以复用Connection，即一个Connection下可以建立多个Channel。Channel不能脱离Connection独立存在，而必须存活在Connection中。当某个Connection断开时，该Connection下的所有Channel都会断开。当大量应用需要与分布式消息服务RabbitMQ建立多个连接时，建议您使用Channel来复用Connection，从而减少网络资源和分布式消息服务RabbitMQ资源消耗。



Connection和Channel的使用建议

保持Connection长连接，请勿频繁开启或关闭Connection。如果确实需要频繁开启或关闭连接，请使用Channel。

一个进程对应一个Connection，一个进程中的多个线程则分别对应一个Connection中的多个Channel。

Producer和Consumer分别使用不同的Connection进行消息发送和消费。

Exchange

Producer将消息发送到Exchange，由 Exchange将消息路由到一个或多个Queue中（或者丢弃），Exchange按照相应的Binding逻辑将消息路由到Queue。

Exchange 类型

- Fanout：该类型路由规则非常简单，会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中，相当于广播功能。

适用于广播消息的场景。

路由示例：

Message	Routing Key	Binding Key	Queue
Message A	key.a	key.c.#key.e	Queue AQueue B
Message B	key.b	key.c.#key.e	Queue AQueue B

- Direct：该类型路由规则会将消息路由到 Binding key 与 Routing key 完全匹配的 Queue 中。

适用于通过简单字符标识符区分消息的场景，常用于单播路由。

匹配示例：

Message	Routing Key	Binding Key	Queue
Message A	key.a	key.a	Queue A
Message B	key.b	key.b	Queue B

- Topic：该类型与Direct类型相似，只是规则没有那么严格，可以模糊匹配和多条件匹配，即该类型Exchange使用Routing key模式匹配和字符串比较的方式将消息路由至绑定的Queue；

支持的通配符包括星号（）和井号（#）。星号（*）代表一个英文单词（例如cn）。井号（#）代表零个、一个或多个英文单词，英文单词间通过英文句号（.）分隔；适用于通过通配符区分消息的场景，常用于多播路由。

路由示例：

Message	Routing Key	Binding Key	Queue
Message A	key.a.b	key.a.b.#	Queue A
Message B	key.a.b.c	key.a.b.#key.a.*.c	Queue AQueue B
Message C	key.a.d.c	key.a.*.c	Queue B

Binding

一套绑定规则，用于告诉Exchange消息应该被存储到哪个Queue。它的作用是把Exchange和Queue按照路由规则绑定起来。

Routing Key

Producer在发送消息给Exchange时，需要指定一个Routing key来设定该消息的路由规则，而Routing key需要与Exchange类型及Binding key联合使用才能生效；一般情况下，Exchange类型与Binding key提前配置好，Producer在发送消息给Exchange时，可以通过指定Routing key来决定消息投放到哪个Queue。

与其他服务关系

虚拟私有云(CT-VPC，Virtual Private Cloud)

虚拟私有云为分布式消息服务RabbitMQ提供一个逻辑隔离的区域，构建一个安全可靠、 可配置和管理的虚拟网络环境。更多信息请参见 虚拟私有云 。

弹性云主机（CT-ECS，Elastic Cloud Server）

分布式消息服务RabbitMQ订购后，默认按照用户选择的实例规格开通弹性云主机，云主机由 CPU、内存、镜像、云硬盘组成，同时结合VPC、安全组、数据多副本保存等能力，打造一个既高效又可靠安全的计算环境，确保分布式消息服务RabbitMQ持久稳定运行。更多信息请参见 弹性云主机 。

云硬盘（CT-EVS，Elastic Volume Service）

分布式消息服务RabbitMQ订购后，默认按照用户选择的存储大小开通云硬盘。云硬盘是一种可弹性扩展的块存储设备，可以为分布式消息服务RabbitMQ提供高性能、高可靠的块存储服务。更多信息请参见 云硬盘 。

弹性IP（Elastic IP，EIP）

弹性IP是可以独立申请的公网 IP 地址，包括公网IP地址与公网出口带宽服务。可以与分布式消息服务RabbitMQ动态绑定和解绑，实现云资源的互联网访问。针对需要公网访问分布式消息服务RabbitMQ实例的需求，用户可开通弹性IP后，在RabbitMQ实例开通页面进行绑定。更多信息请参见 弹性IP 。

计费说明

产品资费

新资费

说明

分布式消息RabbitMQ新资费产品支持RabbitMQ3.8版本引擎，提供集群和单机两种规格实例，支持X86和ARM计算CPU架构类型的计算增强型主机，可选3、5、7、9代理数量。

新资费目前在 华东1、华北2、西南1、华南2、上海36、青岛20、长沙42、南昌5、武汉41、杭州7、西南2-贵州、太原4、郑州5、西安7、呼和浩特3 资源池开放订购。

上述资源池实例新购和续订可享受1年83折，2年7折，3年5折优惠。

价格计算公式

分布式消息服务RabbitMQ费用由实例费用和存储费用两部分组成，两者单价如下表所示，计费公式为：

- 实例费用=实例规格单价 * 代理数量，单机版总节点数量为1。
- 存储费用=存储类型单价 * 代理数量 * 单节点存储空间GB大小，单机版代理数量为1。

实例规格单价

Intel计算增强型

规格名称	实例单价（单个节点）	
	按需标准价格(元/小时)	包月标准价格(元/月)
RabbitMQ.2u4g.cluster	0.98	441
RabbitMQ.4u8g.cluster	2.24	1008
RabbitMQ.8u16g.cluster	4.86	2187
RabbitMQ.12u24g.cluster	7.38	3321
RabbitMQ.16u32g.cluster	9	4050
RabbitMQ.24u48g.cluster	15.12	6804
RabbitMQ.32u64g.cluster	20.16	9072
RabbitMQ.48u96g.cluster	30.24	13608
RabbitMQ.64u128g.cluster	40.32	18144

海光计算增强型

规格名称	实例单价（单个节点）

	按需标准价格(元/小时)	包月标准价格(元/月)
RabbitMQ.hg.2u4g.cluster	1.2152	546.84
RabbitMQ.hg.4u8g.cluster	2.7776	1249.92
RabbitMQ.hg.8u16g.cluster	5.3568	2410.56
RabbitMQ.hg.16u32g.cluster	11.16	5022
RabbitMQ.hg.32u64g.cluster	24.9984	11249.28

鲲鹏计算增强型

规格名称	实例单价（单个节点）	
	按需标准价格(元/小时)	包月标准价格(元/月)
RabbitMQ.kp.2u4g.cluster	1.3916	626.22
RabbitMQ.kp.4u8g.cluster	3.1808	1431.36
RabbitMQ.kp.8u16g.cluster	6.9012	3105.54
RabbitMQ.kp.16u32g.cluster	12.78	5751
RabbitMQ.kp.32u64g.cluster	28.6272	12882.24

存储类型单价

磁盘类型	标准资费（按月）元/GB/月	标准资费（按天）元/GB/天	标准资费（按年）元/GB/年	标准资费（按需）元/GB/小时
高IO（SAS）	0.4	0.013151	4.8	0.000900
超高IO（SSD）	1.2	0.039452	14.4	0.001700

旧资费

说明

分布式消息RabbitMQ旧资费产品根据实例规格分为基础版和高级版，存储空间按照不同类型收费。

旧资费产品目前在 芜湖2、南京3、上海7、重庆2、北京5 资源池开放订购。

包周期

RabbitMQ当前支持包年/包月的计费方式。且按天翼云的包年订购优惠策略享受相关的一次性包年订购优惠。

RabbitMQ队列实例价格如下：

实例类型	实例规格	节点数量	标准资费(元/月)
基础版（300G）	4C8G	3	3248
高级版（300G）	8C16G	3	6321

其中，RabbitMQ队列额外存储空间价格如下：

存储类型	标准资费（元/100GB/月）
普通IO	30

按需计费

RabbitMQ当前支持按需的计费方式，计费颗粒度为按小时。

RabbitMQ队列实例价格如下：

实例类型	实例规格	节点数量	标准资费(元/小时)
基础版（300G）	4C8G	3	6.73
高级版（300G）	8C16G	3	13.13

其中，RabbitMQ队列额外存储空间价格如下：

存储类型	标准资费（元/100GB/小时）
普通IO	0.05

计费项

RabbitMQ包括2个计费项：RabbitMQ队列实例费用、RabbitMQ队列存储空间费用，具体费用详情可参考 产品资费 。

目前天翼云对分布式消息服务RabbitMQ产品支持包年包月、按需两种计费方式。

计费项	计费项说明	使用的计费模式	计费公式
实例费用	计费因子：代理规格和代理数量	包年/包月、按需计费	实例规格单价* 代理数量 * 购买时长
存储空间费用	计费因子：云硬盘类型、容量大小	包年/包月、按需计费	云硬盘规格单价 * 单个代理存储大小 * 代理数量 * 购买时长

计费模式

包周期（包年/包月）、按需2种计费模式供您灵活选择，使用越久越便宜。

按包周期实例计费

天翼云提供包月和包年的购买模式。这种购买方式相对于按需付费则能够提供更大的折扣，对于长期使用者，推荐该方式。包周期计费按照订单的购买周期来进行结算。

按需实例计费

这种购买方式比较灵活，可以即开即停，支持秒级计费。实例从“开通”开启计费到“删除”结束计费，按实际购买时长（精确到秒）计费。

下表列出两种模式的区别：

计费模式	包年/包月	按需计费
付费方式	预付费按照订单的购买周期结算。	后付费按照云服务器实际使用时长计费。
计费周期	按订单的购买周期计费。	按小时结算。
实例升级	支持扩容，工单施工完生效，但是施工过程中服务不可用；不支持缩容。	支持扩容，工单施工完生效，但是施工过程中服务不可用；不支持缩容。
更改计费模式	支持变更为按需资源。	支持变更为包周期资源。
变更规格	支持变更实例规格。	支持变更实例规格。
适用场景	适用于可预估资源使用周期的场景，价格比按需计费模式更优惠。对于长期使用者，推荐该方式。	适用于消息资源需求波动的场景，可以随时开通，随时删除。

变更配置：天翼云分布式消息服务支持计费模式变更，更多操作内容请参考 按需转包周期 。

续费、到期与欠费

到期前续费

手动续订：对于包年/包月订购的分布式消息RabbitMQ服务，用户在资源到期前进行续费操作，可以延长原有资源到期时间，避免资源到期后冻结或超过保留期后被系统回收。详细操作请参考 费用中心-续订管理-手动续订 。

自动续订：自动续订仅针对采用包月、包年计费模式的资源，详细操作请参考 费用中心-续订管理-自动续订 。

到期处理

到期后，分布式消息服务RabbitMQ进入保留期，您将不能正常访问及使用天翼云分布式消息服务RabbitMQ服务，已开通的实例资源将予以保留。

- 若您在到期后15天内续费，续订规则请参考 费用中心-续订管理-续订规则说明 ；
- 若到期15天后您仍未续费，RabbitMQ实例资源将被释放。

欠费原因

在按需计费的模式下帐号的余额不足。

按需欠费资源冻结规则

欠费后，资源进入保留期，您将不能正常访问及使用分布式消息服务RabbitMQ，已开通的实例资源将予以保留。

- 若您在保留期内充值，充值后系统会自动扣减欠费金额。
- 若保留期到期您仍未充值，RabbitMQ实例资源将被释放。

查看欠费账单

欠费后，可以查看欠费详情或者消费账单，具体请参见 费用中心-账单管理-账单预览 。

充值

为防止相关资源不被停止或者释放，请及时进行充值，为避免帐号将进入欠费状态，需要在约定时间内支付欠款，详细操作请参考 费用中心-资金管理-余额充值-在线充值 。

退订

针对分布式消息服务RabbitMQ退订操作，具体请参见文档 费用中心-退订-退订流程 。

变更配置

当需要处理大量消息时，RabbitMQ实例的扩容是一种常见的解决方案。扩容可以增加RabbitMQ集群的吞吐量、存储能力和高可用性。分布式消息服务RabbitMQ提供三类扩容方案，分别为节点、规格和磁盘扩容，更好满足用户不同场景下的扩容需求。

- 节点扩容：指向RabbitMQ集群中添加更多的节点以增加系统的吞吐量和可靠性。通过扩容，可以将消息的发送和消费负载分摊到更多的节点上，从而提高系统的并发处理能力。
- 规格扩容：指通过增加RabbitMQ的资源配置来提升系统的处理能力和性能。
- 磁盘扩容：指增加磁盘的存储容量，以满足更多消息的存储需求。

快速入门

入门指引

本章节将为您介绍分布式消息服务RabbitMQ入门的基本流程，主要包括环境准备、创建资源、编译工程生产消费等环节，帮助您快速上手RabbitMQ。

步骤说明

1、 环境准备

创建RabbitMQ实例先要准备好虚拟私有云、子网和安全组，可选弹性公网IP。

2、 创建实例

在订购分布式消息RabbitMQ填写和确认实例名称、引擎类型、计费模式等信息，确认费用后点击下一步，等待开通流程结果通知成功后完成创建实例。

3、 创建资源

一个新的应用接入消息队列需要先创建相关资源，包括：Vhost、User、Exchange、Queue。

4、 生产消费

以上工作完成后，在客户端应用进行生产消费，包括引入依赖、绑定BindingKey、生产消息和消费消息。

环境准备

概述

在创建天翼云分布式消息服务RabbitMQ实例之前，您需要做一些准备工作。

首先，您需要设置一个虚拟私有云（VPC），这是一个隔离的网络环境，用于托管RabbitMQ实例。

接下来，您需要创建一个子网，它是VPC内部的一个子网络，用于划分不同的部分和区域。

最后，您需要配置一个安全组，用于控制入站和出站的流量规则，以保证RabbitMQ实例的安全性。

每个分布式消息服务RabbitMQ实例都会被部署在特定的VPC中，并与特定的子网和安全组相关联。这种方式可以让您自主配置和管理RabbitMQ实例的网络环境，并提供安全保护策略。如果您已经有了现成的VPC、子网和安全组，可以重复使用它们，无需重复创建。这样可以节省时间和资源，并确保一致性和可靠性。

VPC和子网

VPC和子网可重复使用，您也可以使用不同的VPC和子网来配置RabbitMQ实例，您可根据实际需要进行配置。在创建VPC和子网时应注意如下要求：

- VPC与使用的天翼云分布式消息服务RabbitMQ服务应在相同的区域。
- 如无特殊需求，创建VPC和子网的配置参数使用默认配置即可。

创建VPC和子网的操作请参考 虚拟私有云-创建VPC、子网搭建私有网络 。

若需要在已有VPC上创建和使用新的子网，请参考 虚拟私有云-子网管理-创建子网 。

安全组

安全组可重复使用，您也可以根据实际情况使用不同的安全组，请根据实际需要进行配置。

创建安全组的操作指导，请参考 虚拟私有云-创建安全组 。

若需要为安全组添加规则，请参考 虚拟私有云-安全组-添加安全组规则 。

弹性公网IP（可选）

若需要通过公网访问RabbitMQ实例，则需要申请弹性公网IP，否则不需要申请弹性公网IP。申请弹性公网IP的操作指导请参考 购买弹性IP 。

购买实例

实例介绍

RabbitMQ实例订购支持用户自定义规格和自定义特性，采用物理隔离的方式部署。租户独占RabbitMQ实例，可根据业务需要可定制相应规格的RabbitMQ实例。在新的资源池节点上，还支持选择主机类型和存储规格等丰富用户选项。

操作步骤

1. 登录管理控制台。
2. 进入RabbitMQ管理控制台。
3. 在管理控制台右上角单击“地域名称”，选择区域。此处请选择与您的应用服务相同的区域。
4. 点击“购买实例”跳转到购买页面，根据页面订购说明进行产品开通。

实例规格选择说明

节点可选择3节点、5节点、7节点、9节点，实例规格可选择4C8G、8C16G、16C32G。

存储类型可选择普通IO（SATA）、高IO（SAS）、超高IO（SSD）。

（1）填写实例名称，长度在 4 到 64个字符，必须以字母开头，不区分大小写，可以包含字母、数字、中划线或下划线，不能包含其他特殊字符。

（2）选择引擎类型，默认选择云原生引擎，海量消息堆积能力，支持更多连接和队列数，稳定性高。也可选择rabbitmq引擎，完全支持开源RabbitMQ生态，功能完备。

（3）选择计费模式：包年包月/按需计费，两种模式说明参见计费模式。

（4）购买时长按照计费模式选择变化：

- 计费模式为包年包月，可选择购买时长1-6个月、1-3年。该模式提供自动续期功能，勾选后可以自动续期购买时长：1-6个月、1-3年。
- 计费模式为按需计费，则该选项隐藏无需选择。

（5）部署方式有单可用区和多可用区两个选项，目前仅支持单可用区和3可用区部署,单可用区部署请选中任意一个AZ；多可用区部署请选中3个AZ，系统会自动将Broker节点平均分配至各可用区。

（6）设置节点数，可选择3/5/7/9。RabbitMQ 的节点数是指 RabbitMQ 集群中的节点数量。在 RabbitMQ 集群中，可以有多个节点组成一个集群，每个节点都是一个独立的 RabbitMQ 服务器实例。

（7）下拉选择主机类型，可选择通用型和计算增强型。通用型云主机共享宿主机的CPU资源，主要提供基本水平的vCPU性能、平衡的计算、内存和网络资源，具有较高性价比，支持通用的业务运行。计算增强型云主机独享宿主机的CPU资源，实例间无CPU争抢，并且没有进行资源超配，同时搭载全新网络加速引擎，实现接近物理服务器的强劲稳定性能。

（8）选择实例规格，分布式消息服务RabbitMQ提供通用型和计算增强型各3类规格，各规格详细说明参见 弹性云主机规格 。

（9）选择存储空间，包括磁盘类型和空间。

- 磁盘类型提供高IO/超高IO三类。普通IO适用于大容量、读写速率中等、事务性处理较少的应用场景。高IO：适用于主流的高性能、高可靠应用场景。超高IO：适用于超高IOPS、超大带宽需求的读写密集型应用场景。了解更多磁盘类型说明参见 云硬盘规格 。
- 磁盘空间以100G起步，可以以100倍数增加磁盘空间。

（10）选择已有虚拟私有云，若无虚拟私有云，点击创建跳转到虚拟私有云页面新增，了解更多内容参见 虚拟私有云 。

（11）选择已有子网，若无子网，点击创建跳转到子网页面新增。

（12）选择已有安全组，若无安全组，点击创建跳转到安全组页面新增。

创建资源

背景信息

一个新的应用接入消息队列需要先创建相关资源，包括：Vhost、User、Exchange、Queue。

操作步骤

创建Vhost

1.登录天翼云，进入分布式消息服务RabbitMQ的控制台。

2.进入相应实例的管理页面。

3.点击左侧选项卡的集群管理，然后进入主界面的虚拟主机。

4.点击新建，输入虚拟主机名称即可新增Vhost。

设置Vhost名称时，请注意有如下要求：

- Vhost名称只能包含字母、数字、短划线（-）、下划线（_）。
- Vhost名称长度限制在1~64个字符，长度超过64个字符将被自动截取。
- Vhost创建成功后，Vhost名称不可修改。

创建User并且配置Vhost的权限

1.登录天翼云，进入分布式消息服务RabbitMQ的控制台。

2.进入相应实例的管理页面。

3.点击左侧选项卡的集群管理，然后进入主界面的“用户界面”。

4.创建成功后给用户配置Vhost的权限，选择目标虚拟主机。

5.创建成功后给用户配置Vhost的权限，选择目标虚拟主机，然后进入主机详情，点击权限tab页，点击新增权限。

6.创建成功后给用户配置Vhost的权限，选择目标虚拟主机，然后进入主机详情，点击权限tab页，点击新增权限。

Tips：权限配置规则为正则表达式。例如 .* 表示所有权限。

如'^(amq.gen.*|amq.default) '可以匹配server生成的和默认的Exchange，'^ '不匹配任何资源。

创建Exchange

当创建Vhost时，会创建5个默认的Exchange，如需要额外创建Exchange，可进入实例列表的交换器选项卡进行新建。

参数说明如下表所示

参数	描述
虚拟主机	选择创建交换器所属的虚拟主机
名称	交换器名称。以amq.开头的为保留字段，因此不能使用。例如：。
类型	Exchange类型。取值：
类型	direct：该类型路由规则会将消息路由到Binding Key与Routing Key完全匹配的Queue中。
类型	topic：该类型与direct类型相似。Topic Exchange路由规则没有Direct Exchange那么严格， 支持模糊匹配和多条件匹配，即该类型Exchange使用Routing Key模式匹配和字符串比较的方式将消息路由至绑定的Queue中。
类型	fanout：该类型路由规则非常简单，会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中，相当于广播功能。
类型	headers：该类型与direct类型相似。Headers Exchange使用Headers属性代替Routing Key进行路由匹配，在绑定Headers Exchange和Queue时，设置绑定属性的键值对；在向Headers Exchange发送消息时，设置消息的Headers属性键值对，使用消息Headers属性键值对和绑定属性键值对比较的方式将消息路由至绑定的Queue。

参数	描述
类型	x-delayed-message：通过声明该类Exchange，您可以自定义消息的Header属性x-delay来指定消息延时投递的时间段，单位为毫秒。消息将在x-delay中定义的时间段后，根据路由规则被投递到对应的Queue。路由规则取决于x-delayed-type中指定的Exchange路由类型。
x-delayed-type	当Exchange类型为x-delayed-message时，需要配置此参数，以指定Exchange的路由类型。
是否持久化	交换器是否持久化到磁盘
是否自动删除	如果是，交换器将在至少一个队列或交换器绑定到该交换器，然后所有队列或交换器都已解除绑定时删除。
是否内置	如果是，客户端不能直接发布到这个交换器。它只能与其他交换器绑定使用。
其他参数	Alternate exchange：备份交换器是为了实现没有路由到队列的消息，声明交换机的时候添加属性alternate-exchange，声明一个备用交换机，一般声明为fanout类型，这样交换机收到路由不到队列的消息就会发送到备用交换机绑定的队列中。

创建Queue

可以在代码中声明，会自动创建队列。也可以进入控制台，在实例列表的队列选项卡进行新建。

参数说明如下表所示：

参数	描述
虚拟主机	选择创建队列所属的虚拟主机
名称	队列的名称。以amq.开头的为保留字段，因此不能使用。例如：amq.test。
存储节点	队列数据存储节点

参数	描述
是否持久化	队列元数据是否持久化到磁盘
是否自动删除	最后一个Consumer取消订阅后，Queue是否自动删除。
其他参数	Message TTL-消息过期时间：number型(单位:ms)Auto expire-队列过期时间，过期后队列自动删除：number型(单位:ms)Max length-队列能保存的最大消息数：number型(单位:个)Max length bytes-队列能保存的最大消息量：number型(单位:字节)Overflow behaviour超过队列的最大设定值后消息接收策略：drop-head,reject-publishdrop-head：删除头部消息,一般就是最早发送的消息，保证队列可用reject-publish：拒绝接受新的消息，保证消息不丢失Dead letter exchange死信交换器名称Dead letter routing key死信路由键Maximum priority队列最大优先级：要开启消息的优先级，必须设置消息所在队列的优先级Lazy mode队列惰性模式：default、lazydefault：默认值，普通队列lazy：惰性队列，尽可能将消息存到磁盘中，会引起I/O操作比较多，内存消耗极少（有大量堆积的持久化消息建议使用）Master Locator队列保存位置：client-local、min-masters、randomclient-local：队列创建时所用连接的节点min-masters： 集群中节点主数量最少的节点random：由rabbitmq服务器随机指定一个节点

生产消费

前提条件

创建分布式消息服务RabbitMQ实例。

操作步骤

RabbitMQ是一个开源的消息队列中间件，支持生产者和消费者之间的异步通信。在上述资源准备完成后，接下来需要编译工程生产消费，主要分以下几个步骤：

1、编写生产者代码：编写一个生产者程序。该程序将连接到RabbitMQ服务器，并将消息发送到队列中。

2、编写消费者代码：编写一个消费者程序。该程序将连接到RabbitMQ服务器，并从队列中接收消息。

3、运行生产者和消费者：运行生产者程序，它将发送消息到队列中。然后运行消费者程序，它将从队列中接收并处理消息。

4、验证结果：检查生产者和消费者程序的输出，确保消息被正确发送和接收。

引入依赖

在使用RabbitMQ时，你需要在你的项目中引入相应的依赖。具体的依赖项可能会因你的项目和需求而有所不同。在使用RabbitMQ之前，请确保查阅官方文档以获取最新的依赖项和使用说明。

以Java编程语言为例，可以使用RabbitMQ的Java客户端库。你可以在Maven或Gradle构建工具中添加以下依赖项：

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.7.0</version>
</dependency>
```

也可以通过下载JAR包来引入依赖。

绑定BindingKey

在RabbitMQ中，绑定键（Binding Key）是用于绑定交换机（Exchange）和队列（Queue）的关键字。当一个消息被发送到交换机时，交换机会根据绑定键将消息路由到相应的队列中。

绑定键是在创建绑定（Binding）时指定的，它定义了消息应该如何被路由到队列。绑定键通常与消息的属性或内容进行匹配，以确定消息应该发送到哪个队列。

绑定键可以具有不同的形式，取决于使用的交换机类型。以下是一些常见的绑定键形式：

- 直接匹配（Direct Match）：绑定键与消息的路由键（Routing Key）完全匹配时，消息会被路由到相应的队列。
- 通配符匹配（Wildcard Match）：绑定键可以使用通配符进行模式匹配。常见的通配符有和#，其中表示匹配一个单词，#表示匹配零个或多个单词。
- 主题匹配（Topic Match）：绑定键可以使用主题模式进行匹配。主题模式使用.分隔的单词，可以包含*和#通配符。例如，stock.#可以匹配stock.price、stock.quantity等。

绑定键的选择取决于你的需求和消息的路由策略。通过正确设置绑定键，你可以确保消息被正确地路由到相应的队列中，以便消费者进行处理。

代码示例：

```
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
public class RabbitmqBindingKey {
    private final static String EXCHANGE_NAME = "YOUR EXCHANGE NAME";
    private final static String QUEUE_NAME = "YOUR QUEUE NAME";
    private final static String ROUTING_KEY = "YOUR ROUTING KEY";
    public static void main(String[] args) throws IOException,
 TimeoutException {
        // ######
        ConnectionFactory factory = new ConnectionFactory();
        // ####ip
        factory.setHost("YOUR HOST IP");
        // ##amqp####
        factory.setPort(YOUR PORT);
        // #######
        factory.setUsername("YOUR USER NAME");
```

```
        factory.setPassword("YOUR USER PASSWORD");
        // ##Vhost##########
        factory.setVirtualHost("YOUR VHOST");
        //##############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
          Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.
DIRECT, true);
        // ## ${QueueName}#Queue #############API##
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
        // Queue # Exchange####### BindingKeyTest
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, ROUTING_KEY);
        connection.close();
    }
 }
```

完成后，可以在实例列表的交换器选项卡和队列选项卡查看结果。

生产消息

生产者需要创建一个连接到RabbitMQ服务器，然后创建一个通道（Channel）来进行消息的发布。在发布消息之前，生产者通常需要先声明一个队列，以确保消息能够被正确地路由和接收。

一旦连接和通道建立完成，生产者可以使用basicPublish()方法将消息发布到指定的队列。在发布消息时，需要指定目标队列的名称、消息内容以及其他的属性。

发布消息后，RabbitMQ将会将消息存储在队列中，等待消费者来接收。消费者可以使用相同的客户端库来创建连接和通道，并使用basicConsume()方法来订阅队列并接收消息。一旦有消息到达队列，消费者就会收到消息并进行相应的处理。

通过使用RabbitMQ，生产者和消费者可以实现解耦，即它们可以独立地进行开发和部署。生产者可以按照自己的节奏和需求发布消息，而消费者可以根据自己的处理能力和负载来接收和处理消息。

代码示例：

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
public class RabbitmqProducer {
    // private final static String EXCHANGE_NAME = "YOUR EXCHANGE NAME";
    private final static String QUEUE_NAME = "YOUR QUEUE NAME";
    // private final static String ROUTING_KEY = "YOUR ROUTING KEY";
    public static void main(String[] args) throws IOException, TimeoutException, Interrup
tedException {
        // ######
        ConnectionFactory factory = new ConnectionFactory();
          // ####ip
        factory.setHost("YOUR HOST IP");
        // ##amqp####
        factory.setPort(YOUR PORT);
        // #######
        factory.setUsername("YOUR USER NAME");
        factory.setPassword("YOUR USER PASSWORD");
          // ##Vhost##########
        factory.setVirtualHost("YOUR VHOST");
          //##############
```

```
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
          // ######
        Connection connection = factory.newConnection();
          // ######
        Channel channel = connection.createChannel();
         // ########channel.confirmSelect();
        // ##########channel.txSelect();
        // ######
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        for (int i = 0; i < 100; i++) {
          // #####
          String message = "Hello RabbitMQ!_" + i;
          // ###################
          channel.basicPublish("", QUEUE_NAME, null, message.getBytes(Standard
Charsets.UTF_8));
          // ########################routing key
          // channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY, null, message.
getBytes(StandardCharsets.UTF_8));
          System.out.println(" [x] Sent '" + message + "'");
          TimeUnit.MILLISECONDS.sleep(100);
        }
          //#######
        channel.close();
        connection.close();
      }
}
```

消息发送后，可以进入控制台，在实例列表的队列选项卡查看消息发送状态。

消费消息

消费者需要创建一个连接到RabbitMQ服务器，然后创建一个通道（Channel）来进行消息的订阅。在订阅消息之前，消费者通常需要先声明一个队列，以确保能够正确地接收和处理消息。

一旦连接和通道建立完成，消费者可以使用basicConsume()方法来订阅指定的队列，并注册一个回调函数来处理接收到的消息。当有消息到达队列时，RabbitMQ会将消息推送给消费者，消费者的回调函数将被调用，从而可以对消息进行处理。

消费者可以根据自己的需求设置消息的确认机制。在默认情况下，消费者在接收到消息后，会自动向RabbitMQ发送一个确认（ack）消息，表示已成功接收并处理该消息。如果消费者在处理消息时发生错误，可以选择不发送确认消息，从而使消息重新进入队列，以便其他消费者重新处理。

通过使用RabbitMQ，消费者可以实现解耦，即它们可以独立地进行开发和部署。消费者可以根据自己的处理能力和负载来接收和处理消息，从而实现负载均衡和水平扩展。

代码示例：

```
import com.rabbitmq.client.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;
public class RabbitmqConsumer {
    //####
    private final static String QUEUE_NAME = "Hello#RabbitMQ";
    public static void main(String[] args) throws IOException, TimeoutException {
     //######
      ConnectionFactory factory = new ConnectionFactory();
     //####ip
     factory.setHost("YOUR HOST IP");
     //##amqp####
     factory.setPort(YOUR PORT);
```

```
      //#######
      factory.setUsername("YOUR USER NAME");
      factory.setPassword("YOUR USER PASSWORD");
      //##Vhost##########
      factory.setVirtualHost("YOUR VHOST");
      //###############
      factory.setConnectionTimeout(30 * 1000);
      factory.setHandshakeTimeout(30 * 1000);
      factory.setShutdownTimeout(0);
      Connection connection = factory.newConnection();
      Channel channel = connection.createChannel();
       //###################################
      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
      System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
      Consumer consumer = new DefaultConsumer(channel) {
       @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.basicConsume(QUEUE_NAME, true, consumer);
    }
  }
```

完成上述步骤后，可以在控制台查看消费者是否启动成功。

完成以上所有步骤后，就成功接入了RabbitMQ服务，可以用消息队列进行消息发送和订阅了。

用户指南

创建实例

订购前准备

在创建天翼云分布式消息服务RabbitMQ实例之前，您需要做一些准备工作。

首先，您需要设置一个虚拟私有云（VPC），这是一个隔离的网络环境，用于托管RabbitMQ实例。

接下来，您需要创建一个子网，它是VPC内部的一个子网络，用于划分不同的部分和区域。

最后，您需要配置一个安全组，用于控制入站和出站的流量规则，以保证RabbitMQ实例的安全性。

每个分布式消息服务RabbitMQ实例都会被部署在特定的VPC中，并与特定的子网和安全组相关联。这种方式可以让您自主配置和管理RabbitMQ实例的网络环境，并提供安全保护策略。如果您已经有了现成的VPC、子网和安全组，可以重复使用它们，无需重复创建。这样可以节省时间和资源，并确保一致性和可靠性。

VPC和子网

VPC和子网可重复使用，您也可以使用不同的VPC和子网来配置RabbitMQ实例，您可根据实际需要进行配置。在创建VPC和子网时应注意如下要求：

- VPC与使用的天翼云分布式消息服务RabbitMQ服务应在相同的区域。
- 如无特殊需求，创建VPC和子网的配置参数使用默认配置即可。

创建VPC和子网的操作请参考 虚拟私有云-创建VPC、子网搭建私有网络

若需要在已有VPC上创建和使用新的子网，请参考 虚拟私有云-子网管理-创建子网

安全组

安全组可重复使用，您也可以根据实际情况使用不同的安全组，请根据实际需要进行配置。

创建安全组的操作指导，请参考 虚拟私有云-创建安全组

若需要为安全组添加规则，请参考 虚拟私有云-安全组-添加安全组规则

弹性公网IP（可选）

若需要通过公网访问RabbitMQ实例，则需要申请弹性公网IP，否则不需要申请弹性公网IP。申请弹性公网IP的操作指导请参考 购买弹性IP 。

实例规格选择说明

节点可选择3节点、5节点、7节点、9节点，实例规格可选择4C8G、8C16G、16C32G。

存储类型可选择普通IO（SATA）、高IO（SAS）、超高IO（SSD）。

（1）填写实例名称，长度在 4 到 64个字符，必须以字母开头，不区分大小写，可以包含字母、数字、中划线或下划线，不能包含其他特殊字符。

（2）选择引擎类型，默认选择云原生引擎，海量消息堆积能力，支持更多连接和队列数，稳定性高。也可选择rabbitmq引擎，完全支持开源RabbitMQ生态，功能完备。

（3）选择计费模式：包年包月/按需计费，两种模式说明参见计费模式。

（4）购买时长按照计费模式选择变化：

- 计费模式为包年包月，可选择购买时长1-6个月、1-3年。该模式提供自动续期功能，勾选后可以自动续期购买时长：1-6个月、1-3年。
- 计费模式为按需计费，则该选项隐藏无需选择。

（5）部署方式有单可用区和多可用区两个选项，目前仅支持单可用区和3可用区部署,单可用区部署请选中任意一个AZ；多可用区部署请选中3个AZ，系统会自动将Broker节点平均分配至各可用区。

（6）设置节点数，可选择3/5/7/9。RabbitMQ 的节点数是指 RabbitMQ 集群中的节点数量。在 RabbitMQ 集群中，可以有多个节点组成一个集群，每个节点都是一个独立的 RabbitMQ 服务器实例。

（7）下拉选择主机类型，可选择通用型和计算增强型。通用型云主机共享宿主机的CPU资源，主要提供基本水平的vCPU性能、平衡的计算、内存和网络资源，具有较高性价比，支持通用的业务运行。计算增强型云主机独享宿主机的CPU资源，实例间无CPU争抢，并且没有进行资源超配，同时搭载全新网络加速引擎，实现接近物理服务器的强劲稳定性能。

（8）选择实例规格，分布式消息服务RabbitMQ提供通用型和计算增强型各3类规格，各规格详细说明参见 弹性云主机规格 。

（9）选择存储空间，包括磁盘类型和空间。

- 磁盘类型提供高IO/超高IO三类。普通IO适用于大容量、读写速率中等、事务性处理较少的应用场景。高IO：适用于主流的高性能、高可靠应用场景。超高IO：适用于超高IOPS、超大带宽需求的读写密集型应用场景。了解更多磁盘类型说明参见 云硬盘规格 。
- 磁盘空间以100G起步，可以以100倍数增加磁盘空间。

（10）选择已有虚拟私有云，若无虚拟私有云，点击创建跳转到虚拟私有云页面新增，了解更多内容参见 虚拟私有云 。

（11）选择已有子网，若无子网，点击创建跳转到子网页面新增。

（12）选择已有安全组，若无安全组，点击创建跳转到安全组页面新增。

实例管理

查看实例

实例列表

1、进入分布式消息服务RabbitMQ管理控制台查看已购买的实例列表，若列表为空，可点击右上角【创建实例】进入购买页面，创建实例详情见具体操作步骤。

2、支持按照实例ID查询，输入查询内容，点击【查询】按钮即可展示需要的实例数据。

3、查看实例基本信息，包括实例ID、规格、VPC、计费模式、创建时间、到期时间、状态。其中状态说明见下文。

运行状态

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）当前页面会列出所购买的RabbitMQ实例，并查看状态，状态说明如下

状态	说明
运行中	RabbitMQ实例正常运行状态。
已关闭	RabbitMQ实例处于离线的状态。
变更中	RabbitMQ实例正在进行规格变更操作。
变更失败	RabbitMQ实例处于规格变更失败状态。
暂停	RabbitMQ专享版实例处于已冻结状态，用户可以在“更多”中续费开启冻结的RabbitMQ实例。
注销	RabbitMQ实例已经过期并关闭，需要重新购买实例。

实例概览

场景描述

实例概览展示分布式消息服务RabbitMQ实例详情信息，包括基本信息、统计信息、节点、端口等，并提供导出服务、元数据导入服务等功能。

- 基本信息：展示实例订购选择的参数信息，支持安全组修改和绑定公网弹性IP。
- 统计信息：统计实例连接数、信道数、交换器数、队列数、消费者数等数据。
- 节点：列表展示实例节点信息。
- 端口：列表展示实例端口信息。
- 导出服务：下载ssl文件和broker元数据。
- 元数据导入服务：支持本地元数据文件Json格式导入。

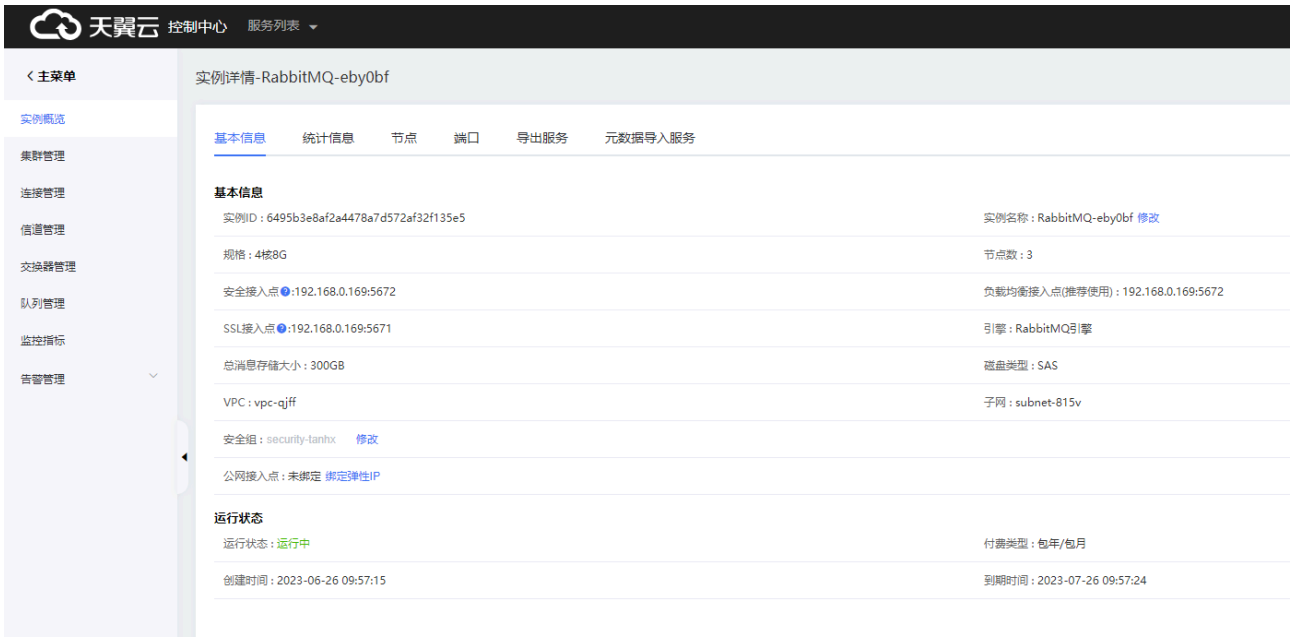
操作步骤

基本信息

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）在实例列表页在操作列，对目标实例点击“管理”。

（3）点击“实例概览”后点击“基本信息”。

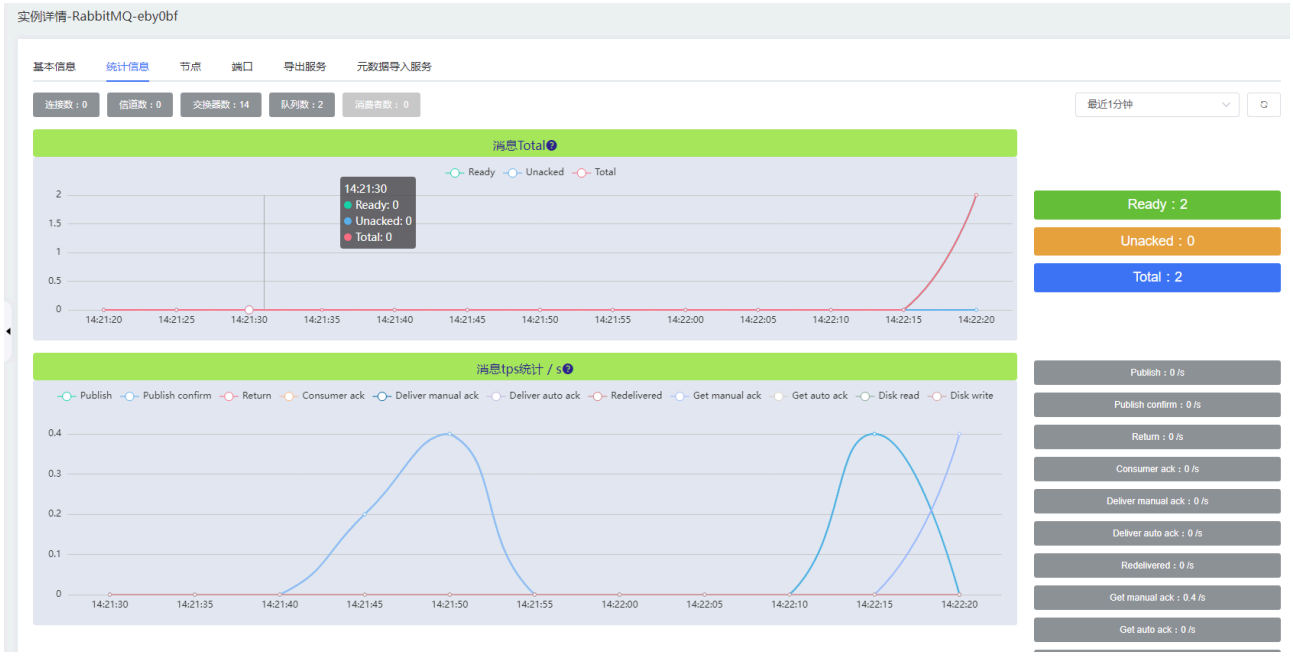


统计信息

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）在实例列表页在操作列，对目标实例点击“管理”。

（3）点击“实例概览”后点击“统计信息”。



节点

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）在实例列表页在操作列，对目标实例点击“管理”。

（3）点击“实例概览”后点击“节点”。

端口

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）在实例列表页在操作列，对目标实例点击“管理”。

（3）点击“实例概览”后点击“端口”。

导出服务

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）在实例列表页在操作列，对目标实例点击“管理”。

（3）点击“实例概览”后点击“导出服务”。点击“下载ssl文件”按钮即可下载ssl文件。点击“下载broker配置”按钮即可下载broker元数据。

元数据导入服务

（1）登录天翼云，进入分布式消息服务RabbitMQ管理控制台。

（2）在实例列表页在操作列，对目标实例点击“管理”。

（3）点击“实例概览”后点击“元数据导入服务”。点击“选择文件”按钮即可选择本地元数据文件。点击“导入配置”按钮即可导入元数据。

连接实例

场景描述

连接RabbitMQ实例的场景包括：

1. 消息队列通信：连接RabbitMQ实例可以用于构建分布式系统中的消息队列通信。不同的应用程序或服务可以通过RabbitMQ实例发送和接收消息，实现解耦和异步通信。
2. 任务队列：连接RabbitMQ实例可以用于构建任务队列，将任务提交到RabbitMQ中，然后由消费者从队列中获取任务并进行处理。这样可以实现任务的分发和负载均衡，提高系统的处理能力和可伸缩性。
3. 发布/订阅模式：连接RabbitMQ实例可以用于实现发布/订阅模式，其中发布者将消息发布到交换器，然后订阅者可以从交换器中订阅感兴趣的消息。这样可以实现消息的广播和多个消费者的并行处理。
4. 日志收集：连接RabbitMQ实例可以用于实现日志收集系统，应用程序可以将日志消息发送到RabbitMQ中，然后由日志消费者从队列中获取日志消息并进行处理和存储。
5. 系统集成：连接RabbitMQ实例可以用于实现不同系统之间的集成，通过将消息发送到RabbitMQ中，其他系统可以从队列中获取消息并进行处理，实现系统之间的数据交换和通信。

总之，连接RabbitMQ实例可以应用于各种场景，包括消息队列通信、任务队列、发布/订阅模式、日志收集和系统集成，提供了一种可靠和灵活的消息传递机制。

操作步骤

RabbitMQ是一个开源的消息队列中间件，支持生产者和消费者之间的异步通信。在上述资源准备完成后，接下来需要编译工程生产消费，主要分以下几个步骤：

1、编写生产者代码：使用编程语言编写一个生产者程序。该程序将连接到RabbitMQ服务器，并将消息发送到队列中。

2、编写消费者代码：同样使用编程语言编写一个消费者程序。该程序将连接到RabbitMQ服务器，并从队列中接收消息。

3、运行生产者和消费者：运行生产者程序，它将发送消息到队列中。然后运行消费者程序，它将从队列中接收并处理消息。

4、验证结果：检查生产者和消费者程序的输出，确保消息被正确发送和接收。

引入依赖

在使用RabbitMQ时，你需要在你的项目中引入相应的依赖。具体的依赖项可能会因你的项目和需求而有所不同。在使用RabbitMQ之前，请确保查阅官方文档以获取最新的依赖项和使用说明。

以Java编程语言为例，可以使用RabbitMQ的Java客户端库。你可以在Maven或Gradle构建工具中添加以下依赖项：

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.7.0</version>
</dependency>
```

也可以通过下载JAR包来引入依赖。

绑定BindingKey

在RabbitMQ中，绑定键（Binding Key）是用于绑定交换机（Exchange）和队列（Queue）的关键字。当一个消息被发送到交换机时，交换机会根据绑定键将消息路由到相应的队列中。

绑定键是在创建绑定（Binding）时指定的，它定义了消息应该如何被路由到队列。绑定键通常与消息的属性或内容进行匹配，以确定消息应该发送到哪个队列。

绑定键可以具有不同的形式，取决于使用的交换机类型。以下是一些常见的绑定键形式：

- 接匹配（Direct Match）：绑定键与消息的路由键（Routing Key）完全匹配时，消息会被路由到相应的队列。
- 通配符匹配（Wildcard Match）：绑定键可以使用通配符进行模式匹配。常见的通配符有和#，其中表示匹配一个单词，#表示匹配零个或多个单词。
- 主题匹配（Topic Match）：绑定键可以使用主题模式进行匹配。主题模式使用.分隔的单词，可以包含*和#通配符。例如，stock.#可以匹配stock.price、stock.quantity等。

绑定键的选择取决于你的需求和消息的路由策略。通过正确设置绑定键，你可以确保消息被正确地路由到相应的队列中，以便消费者进行处理。

代码示例：

```
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
public class RabbitmqBindingKey {
    private final static String EXCHANGE_NAME = "YOUR EXCHANGE NAME";
    private final static String QUEUE_NAME = "YOUR QUEUE NAME";
    private final static String ROUTING_KEY = "YOUR ROUTING KEY";
    public static void main(String[] args) throws IOException,
 TimeoutException {
        // ######
        ConnectionFactory factory = new ConnectionFactory();
         // ####ip
        factory.setHost("YOUR HOST IP");
        // ##amqp####
```

```
        factory.setPort(YOUR PORT);
        // #######
        factory.setUsername("YOUR USER NAME");
        factory.setPassword("YOUR USER PASSWORD");
          // ##Vhost##########
        factory.setVirtualHost("YOUR VHOST");
         //###############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
          Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.exchangeDeclare(EXCHANGE_NAME,      BuiltinExchangeType.
DIRECT, true);
          // ## ${QueueName}#Queue #############API##
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
          // Queue # Exchange####### BindingKeyTest
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, ROUTING_KEY);
          connection.close();
      }
 }
```

完成后，可以在实例列表的交换器选项卡和队列选项卡查看结果。

生产消息

生产者需要创建一个连接到RabbitMQ服务器，然后创建一个通道（Channel）来进行消息的发布。在发布消息之前，生产者通常需要先声明一个队列，以确保消息能够被正确地路由和接收。

一旦连接和通道建立完成，生产者可以使用basicPublish()方法将消息发布到指定的队列。在发布消息时，需要指定目标队列的名称、消息内容以及其他的属性。

发布消息后，RabbitMQ将会将消息存储在队列中，等待消费者来接收。消费者可以使用相同的客户端库来创建连接和通道，并使用basicConsume()方法来订阅队列并接收消息。一旦有消息到达队列，消费者就会收到消息并进行相应的处理。

通过使用RabbitMQ，生产者和消费者可以实现解耦，即它们可以独立地进行开发和部署。生产者可以按照自己的节奏和需求发布消息，而消费者可以根据自己的处理能力和负载来接收和处理消息。

代码示例：

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
public class RabbitmqProducer {
    // private final static String EXCHANGE_NAME = "exchangeTest";
    private final static String QUEUE_NAME = "YOUR QUEUE NAME";
    // private final static String ROUTING_KEY = "YOUR ROUTING KEY";
    public static void main(String[] args) throws IOException, TimeoutException, Interrup
tedException {
        // ######
        ConnectionFactory factory = new ConnectionFactory();
          // ####ip
        factory.setHost("YOUR HOST IP");
        // ##amqp####
        factory.setPort(YOUR PORT);
        // #######
        factory.setUsername("YOUR USER NAME");
        factory.setPassword("YOUR USER PASSWORD");
          // ##Vhost##########
        factory.setVirtualHost("YOUR VHOST");
```

```
          //##############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
          // ######
        Connection connection = factory.newConnection();
          // ######
        Channel channel = connection.createChannel();
         // ########channel.confirmSelect();
        // ##########channel.txSelect();
        // ######
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        for (int i = 0; i < 100; i++) {
          // #####
          String message = "Hello RabbitMQ!_" + i;
          // ###################
          channel.basicPublish("", QUEUE_NAME, null, message.getBytes(Standard
Charsets.UTF_8));
          // ########################routing key
          // channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY, null, message.
getBytes(StandardCharsets.UTF_8));
          System.out.println(" [x] Sent '" + message + "'");
          TimeUnit.MILLISECONDS.sleep(100);
        }
          //#######
        channel.close();
        connection.close();
      }
}
```

消息发送后，可以进入控制台，在实例列表的队列选项卡查看消息发送状态。

消费消息

消费者需要创建一个连接到RabbitMQ服务器，然后创建一个通道（Channel）来进行消息的订阅。在订阅消息之前，消费者通常需要先声明一个队列，以确保能够正确地接收和处理消息。

一旦连接和通道建立完成，消费者可以使用basicConsume()方法来订阅指定的队列，并注册一个回调函数来处理接收到的消息。当有消息到达队列时，RabbitMQ会将消息推送给消费者，消费者的回调函数将被调用，从而可以对消息进行处理。

消费者可以根据自己的需求设置消息的确认机制。在默认情况下，消费者在接收到消息后，会自动向RabbitMQ发送一个确认（ack）消息，表示已成功接收并处理该消息。如果消费者在处理消息时发生错误，可以选择不发送确认消息，从而使消息重新进入队列，以便其他消费者重新处理。

通过使用RabbitMQ，消费者可以实现解耦，即它们可以独立地进行开发和部署。消费者可以根据自己的处理能力和负载来接收和处理消息，从而实现负载均衡和水平扩展。

代码示例：

```
import com.rabbitmq.client.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;
public class RabbitmqConsumer {
    //####
    private final static String QUEUE_NAME = "YOUR QUEUE NAME";
    public static void main(String[] args) throws IOException, TimeoutException {
     //######
      ConnectionFactory factory = new ConnectionFactory();
     //####ip
     factory.setHost("YOUR HOST IP");
     //##amqp####
     factory.setPort(YOUR PORT);
```

```
      //#######
      factory.setUsername("YOUR USER NAME");
      factory.setPassword("YOUR USER PASSWORD");
      //##Vhost##########
      factory.setVirtualHost("YOUR VHOST");
      //###############
      factory.setConnectionTimeout(30 * 1000);
      factory.setHandshakeTimeout(30 * 1000);
      factory.setShutdownTimeout(0);
      Connection connection = factory.newConnection();
      Channel channel = connection.createChannel();
       //###################################
      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
      System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
      Consumer consumer = new DefaultConsumer(channel) {
       @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.basicConsume(QUEUE_NAME, true, consumer);
    }
  }
```

完成上述步骤后，可以在控制台查看消费者是否启动成功。

完成以上所有步骤后，就成功接入了RabbitMQ服务，可以用消息队列进行消息发送和订阅了。

修改实例

场景描述

在使用RabbitMQ时，可能会遇到需要修改实例的场景，例如：

- 业务变更：当业务需求发生变化，需要修改RabbitMQ实例的名称以反映新的业务逻辑时，可以进行实例名称的修改。例如，当新增或调整了某个业务模块，需要将其对应的实例名称修改为更准确的名称，以便于管理和监控。
- 网络配置变更：当需要变更RabbitMQ实例的网络配置时，可能需要修改弹性 IP。例如，当需要更改子网、VPC 或安全组的配置时，可能需要重新绑定弹性 IP，以确保 RabbitMQ 实例能够正确地与其他组件通信。
- 安全配置修改：当需要变更RabbitMQ实例的安全配置时，可以进行相应的修改。

需要注意的是，在进行实例的修改时，应该谨慎操作，并根据实际需求和系统情况进行调整。同时，修改实例时应该遵循RabbitMQ的最佳实践和建议，以确保系统的稳定性和性能。

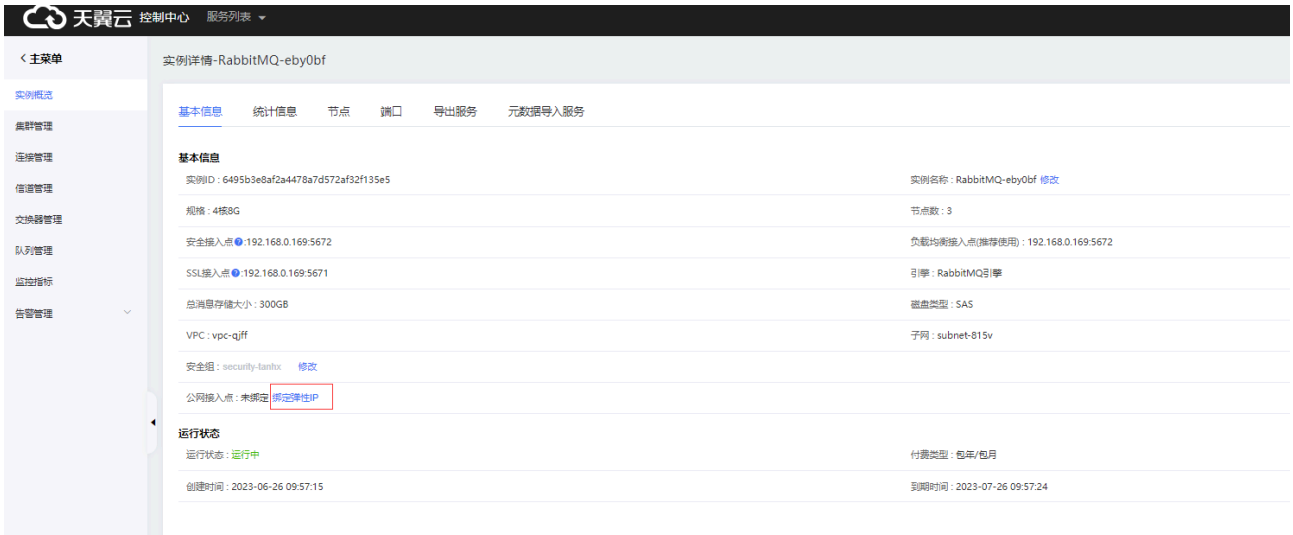
修改实例名称

（1）在基本信息的实例名称后点击"修改"

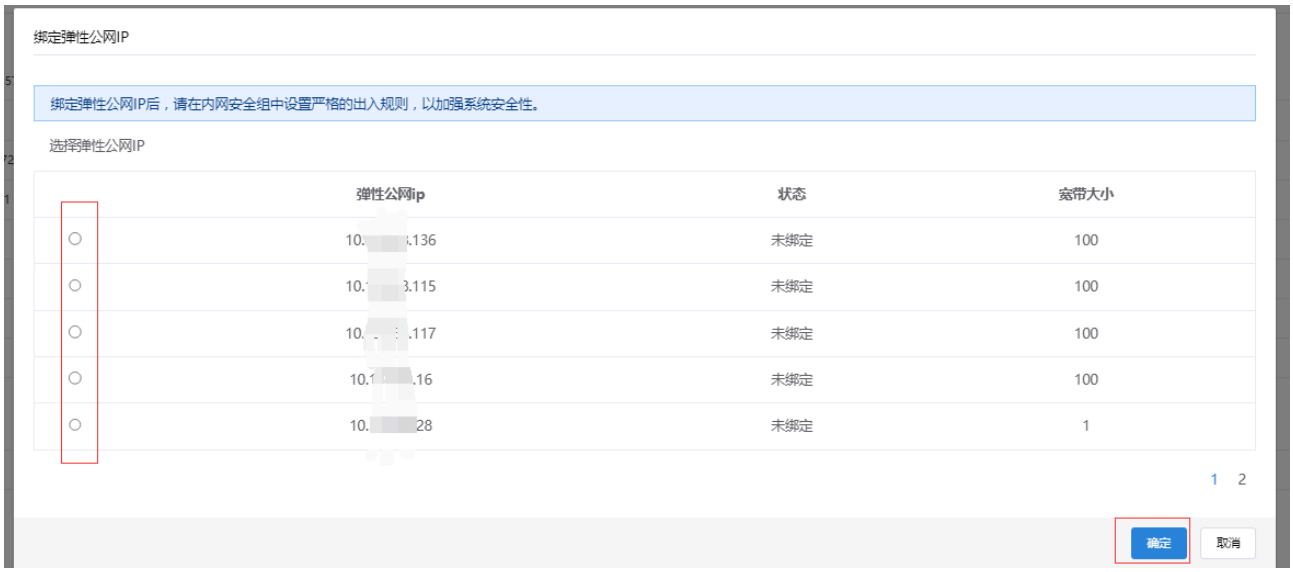
（2）在弹出窗口中输入新的实例名称，点击“确定”

设置公网访问的弹性IP

（1）在基本信息的公网接入点后点击"绑定弹性IP"

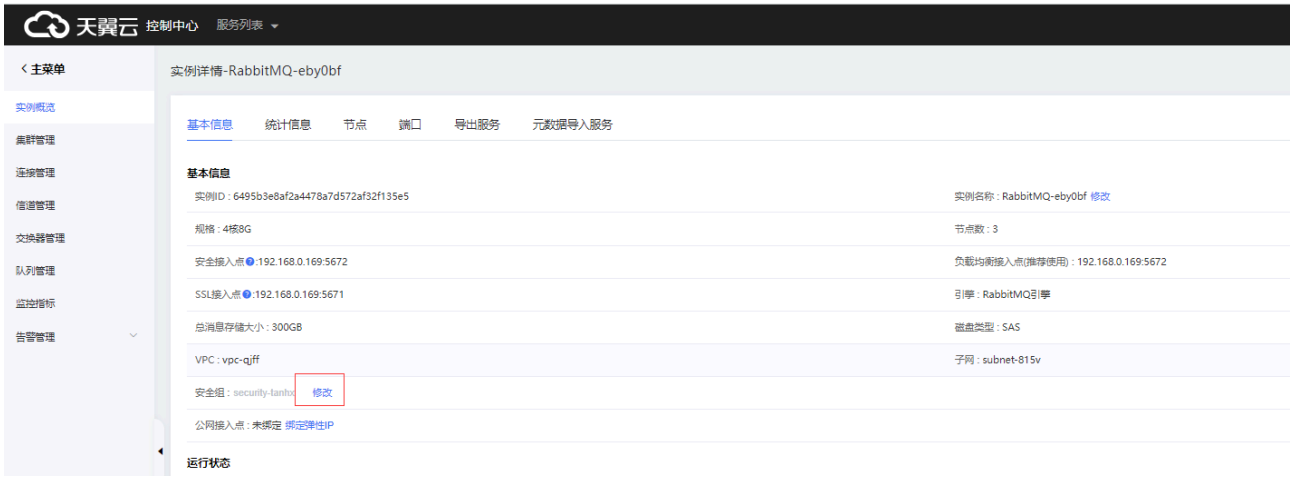


（2）在弹出窗口中选择弹性IP，点击“确定”

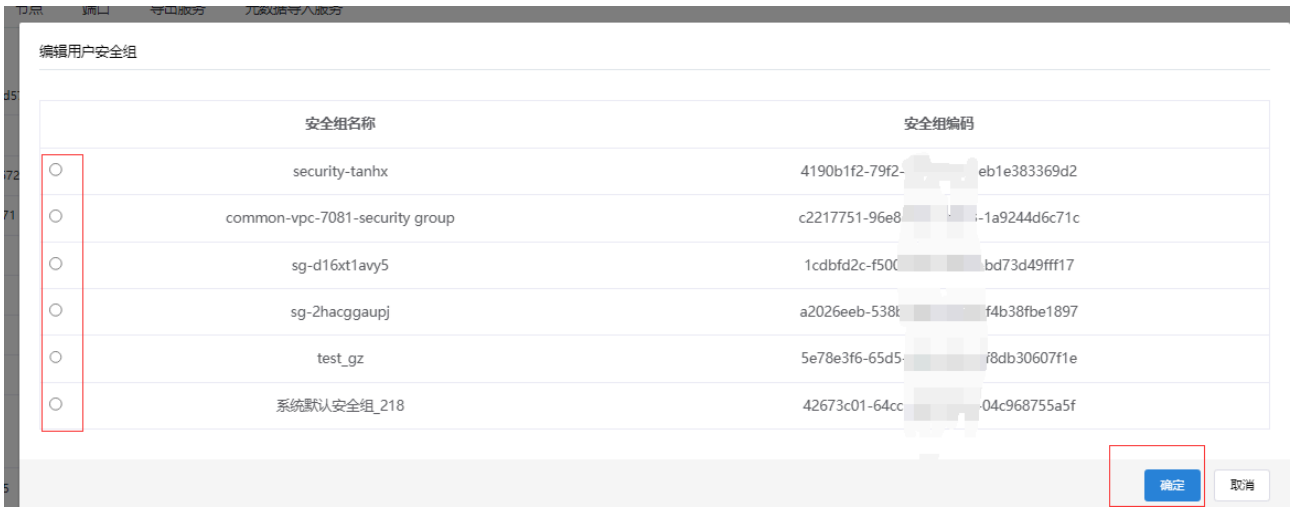


修改安全组

（1）在基本信息的安全组后点击"修改"



（2）在弹出窗口中选择安全组，点击“确定”



实例退订

背景信息

分布式消息服务RabbitMQ为用户提供全面周到的服务，支持用户退订需求。用户如不需要继续使用该分布式消息服务RocketMQ实例，可进行删除实例操作，即退订操作。

操作步骤

（1）登录管理控制台。

（2）进入RabbitMQ管理控制台。

（3）当前页面会列出所购买的RabbitMQ实例。选择需要退订的实例，点击更多-退订。

注意：

- 退订的实例处于冻结状态，请务必在实例退订前停止全部的应用。
- 在申请退订前，请做好数据备份工作，退订后数据将保留15个自然日，15天后相关数据将不予保留，且不会进行备份，务必谨慎操作。

实例扩容

背景信息

当需要处理大量消息时，RabbitMQ实例的扩容是一种常见的解决方案。扩容可以增加RabbitMQ集群的吞吐量、存储能力和高可用性。分布式消息服务RabbitMQ提供三类扩容方案，分别为节点、规格和磁盘扩容，更好满足用户不同场景下的扩容需求。

- 节点扩容：指向RabbitMQ集群中添加更多的节点以增加系统的吞吐量和可靠性。通过扩容，可以将消息的发送和消费负载分摊到更多的节点上，从而提高系统的并发处理能力。
- 规格扩容：指通过增加RabbitMQ的资源配置来提升系统的处理能力和性能。
- 磁盘扩容：指增加磁盘的存储容量，以满足更多消息的存储需求。

操作步骤

以下操作适用于华东1、上海36节点。

（1）登录管理控制台。

（2）进入RabbitMQ管理控制台。

（3）当前页面会列出所购买的RabbitMQ实例。选择需要扩容的实例，点击更多-磁盘扩容/规格扩容/节点扩容.

扩容说明

- 磁盘扩容：增加实例磁盘空间。选择需要的磁盘空间后点击确定下单扩容。

- 规格扩容：将实例的规格升配，选择需要的规格点击确定下单扩容。

- 节点扩容：在3/5/7/9四种节点数目中增加节点数量。选择需要的节点数点击确定下单扩容。

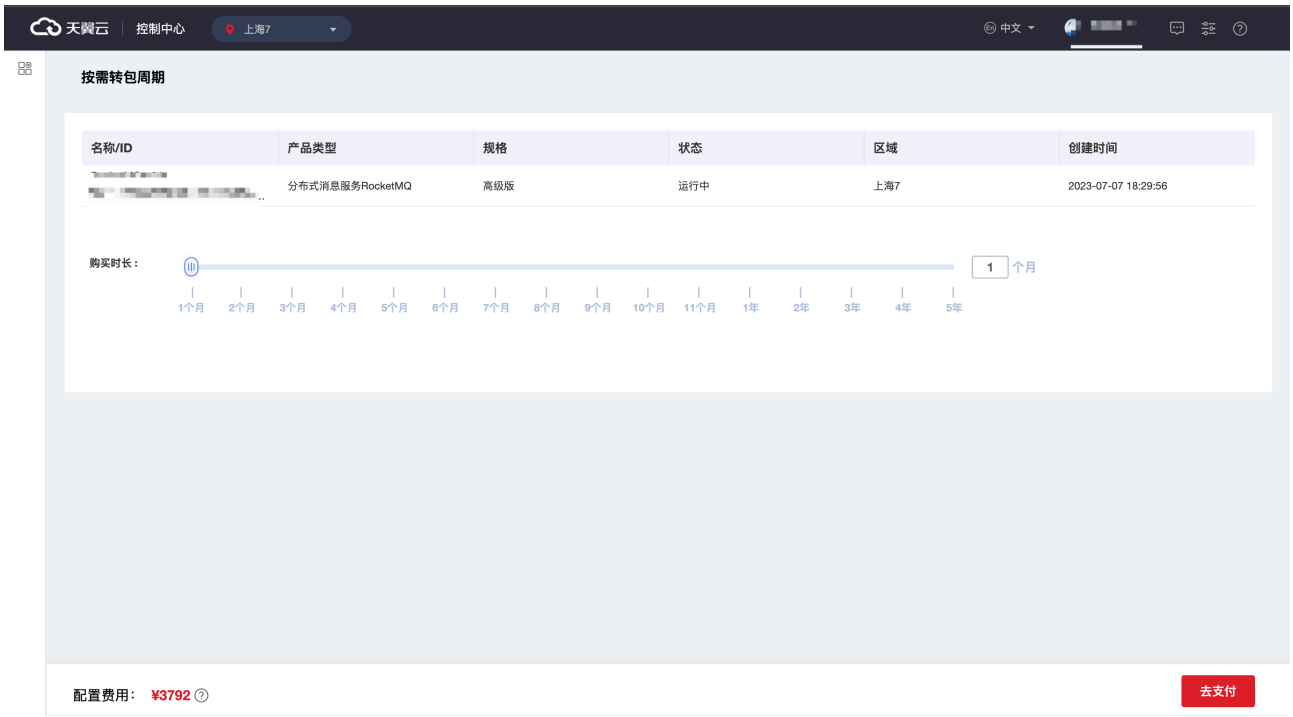
按需转包周期

操作步骤

1、登录分布式消息服务RabbitMQ消息控制台，可以看到当前租户下面的实例列表。

2、点击需要变更实例栏 > 更多 > 按需转包周期。

3、进入到按需转包周期页面，在弹出来的确认窗口选择续订时长，点击确认即可。



修改RabbitMQ实例配置参数

1、登录天翼云分布式消息服务RabbitMQ控制台。

2、点击RabbitMQ实例名称，进入实例详情页面。

3、在“智能运维>配置管理”页面，修改配置参数。

参数说明：

参数	参数说明	参数范围	默认值
heartbeat	心跳超时时间（秒）	0～300	300
frame_max	与客户端协商的允许最大frame大小	0～1048576	131072
initial_frame_max	在连接之前服务器将接受的最大帧大小	0～1048576	4096
max_message_size	最大消息大小	1～52428800	52428800
vm_memory_high_watermark	流程控制触发的内存阈值	0～1	0.8
vm_memory_high_watermark_paging_ratio	高水位限制的分数，当达到阈值时，队列中消息消息会转移到磁盘上以释放内存	0～1	0.5
memory_monitor_interval	执行内存检查的间隔（毫秒）	0～60000	2500
collect_statistics_interval	统计信息收集间隔，增加此值将减少管理数据库的负载（毫秒）	5000～3600000	5000

回收站管理

分布式消息服务RabbitMQ实例在被删除后，会被临时存入回收站中，此时实例中的数据尚未被彻底删除，在保留天数内支持从回收站中恢复此实例。超过保留天数的实例会被彻底删除，无法恢复。

约束与限制

- 回收站中的实例不会收取实例的费用，但是会收取存储空间的费用。
- 包年/包月的实例退订后会存入回收站中，此时不会收取实例的费用，但是收取存储空间的费用。
- 回收站中的按需实例，不支持实例恢复。

回收站管理

1. 登录分布式消息服务RabbitMQ管理控制台。
2. 在左侧导航栏选择“回收站”，进入“回收站”页面。

恢复RabbitMQ实例

1. 登录分布式消息服务RabbitMQ管理控制台。
2. 在左侧导航栏选择“回收站”，进入“回收站”页面。
3. 在待恢复RabbitMQ实例所在行，单击“恢复”。

删除回收站中的实例

1. 登录分布式消息服务RabbitMQ管理控制台。
2. 在左侧导航栏选择“回收站”，进入“回收站”页面。
3. 在待删除RabbitMQ实例所在行，单击“销毁”。

注意

回收站中的RabbitMQ实例删除后，实例中原有的数据将被删除，且没有备份，请谨慎操作。

虚拟主机管理

创建虚拟主机

背景信息

虚拟主机，用作逻辑隔离，分别管理各自的交换器、队列和绑定，使得应用安全地运行在不同的虚拟主机上，相互之间不会干扰。一个实例下可以有多个虚拟主机，一个虚拟主机里面可以有若干个交换器和队列。生产者和消费者连接分布式消息服务 RabbitMQ 版需要指定一个虚拟主机。

操作步骤

（1）登录管理控制台。

（2）进入RabbitMQ管理控制台。

（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“集群管理”后点击“虚拟主机”到达虚拟主机管理页面，点击“新建”按钮。

（5）点击“新建”后出现以下创建，输入虚拟主机名称后点击确定。

注意：设置虚拟主机名称时，有如下要求：

- 虚拟主机名称只能包含字母、数字、短划线（-）、下划线（_）。
- 虚拟主机名称长度限制在1~64个字符。
- 虚拟主机创建成功后，Vhost名称不可修改。

查看虚拟主机

场景描述

在RabbitMQ中，虚拟主机（Virtual Host）是一个逻辑隔离的消息代理环境，用于分隔不同应用或不同团队之间的消息流。当需要查看虚拟主机的信息时，可能有以下场景描述：

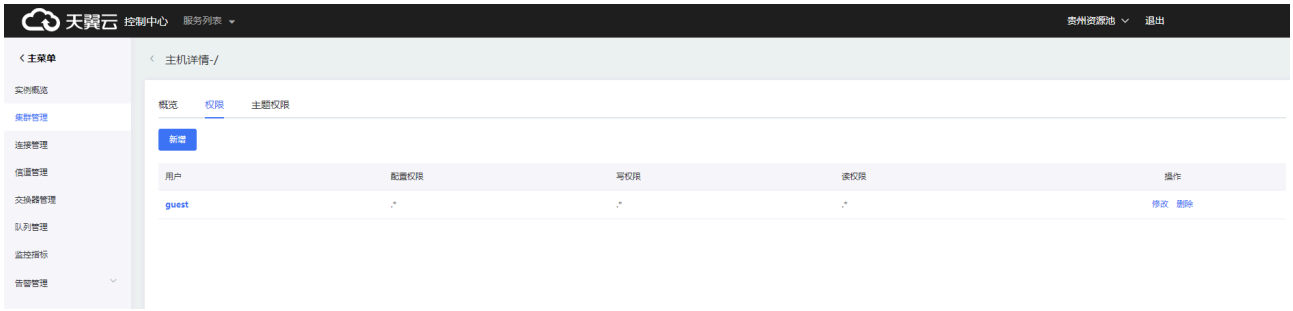
- 管理员查看虚拟主机配置：管理员需要查看虚拟主机的配置信息，包括虚拟主机的名称、权限设置、连接数限制等。这可以帮助管理员了解当前系统的虚拟主机情况，进行配置管理和优化。
- 开发人员查看虚拟主机队列信息：开发人员需要查看虚拟主机中队列的信息，包括队列的名称、消息数量、消费者数量等。这可以帮助开发人员了解当前队列的状态，监控消息的流动情况，进行故障排查和性能调优。
- 运维人员查看虚拟主机连接信息：运维人员需要查看虚拟主机的连接信息，包括连接的客户端IP、连接数、协议等。这可以帮助运维人员监控连接的使用情况，检测异常连接，进行资源管理和安全审计。

操作步骤

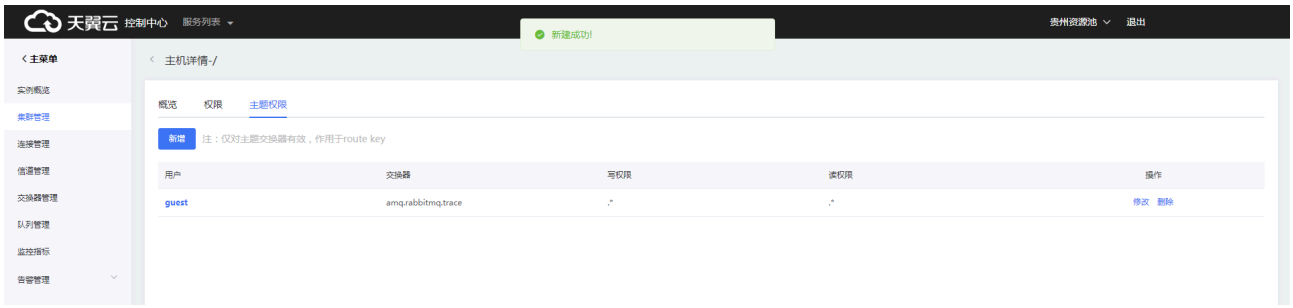
（1）在虚拟主机管理页面，点击目标虚拟主机名称，即可参看虚拟主机详情。

（2）查看虚拟主机概览，主要展示最近时间段各类型消息数量和消息tps统计。

（3）参看用户配置权限及读写权限信息。



（4）查看用户主题权限信息。



删除虚拟主机

场景描述

在RabbitMQ中，删除虚拟主机（Virtual Host）的场景描述如下：

- 应用迁移或清理：当需要迁移应用或进行系统清理时，可能需要删除不再使用的虚拟主机。例如，如果某个应用已经停用或迁移到其他环境，可以删除相关的虚拟主机，以释放资源和减少管理工作。
- 安全审计和合规性要求：根据安全审计和合规性要求，可能需要删除不再需要的虚拟主机。例如，当某个虚拟主机涉及敏感数据或权限配置存在问题时，可以选择删除该虚拟主机以避免安全风险。
- 故障处理和恢复：在某些情况下，当虚拟主机发生故障或出现严重问题时，可能需要删除虚拟主机并重新创建。这可以作为一种故障处理和恢复的手段，以解决虚拟主机相关的问题并恢复正常运行。
- 系统资源管理和优化：如果虚拟主机过多或占用了过多的系统资源，可能需要删除一些不再使用或不需要的虚拟主机。这有助于优化系统资源的利用和提高整体性能。

需要注意的是，在删除虚拟主机之前，务必确保相关的队列、交换机和绑定等对象已经被删除或迁移。否则，删除虚拟主机可能会导致数据丢失或系统异常。

操作步骤

（1）在虚拟主机管理页面，在目标虚拟主机行点击“删除”，即可删除虚拟主机。

注意事项： 删除虚拟主机会删除该虚拟主机内所有数据且不可恢复，请谨慎操作。

虚拟主机用户权限管理

介绍分布式消息服务RabbitMQ虚拟主机权限管理内容。

场景描述

在RabbitMQ中，虚拟主机（Virtual Hosts, vhosts）是一种逻辑隔离机制，允许单个 RabbitMQ 实例划分为多个独立的环境。每个虚拟主机都有自己的交换机、队列、绑定和权限设置。虚拟主机的用户权限管理，支持对每个用户配置三种权限，分别是配置权限、写权限、读权限，用户在未配置这三种权限之前，无法访问对应的虚拟主机。三个权限的详情如下：

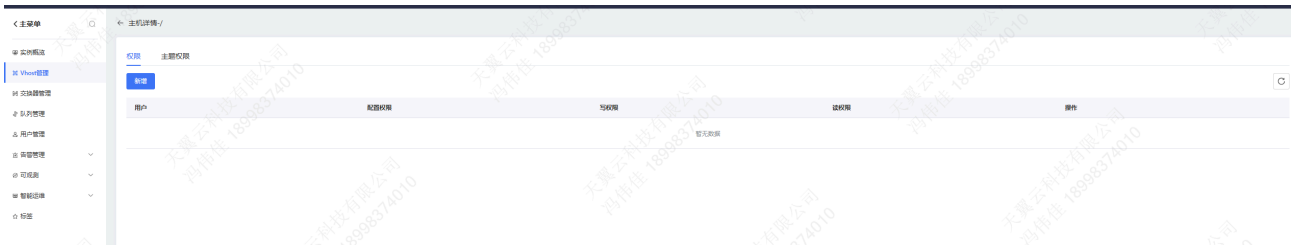
- 配置权限：允许用户创建、修改或删除交换器（Exchange）、队列（Queue）和绑定关系。
- 写权限：允许用户向交换器发布消息。
- 读权限：允许用户消费队列中的消息或获取队列状态。

权限配置

在RabbitMQ中，虚拟主机的用户权限配置，采用的是正则表达式匹配方式， 只有满足正则表达式的资源才有权限。比如配置写权限为：".*"，表示该用户可以向虚拟主机下全部的交换器发布消息。配置写权限为："^$”，表示该用户不允许向任何交换器发布消息。配置写权限为："^exchange_.*"，表示该用户可以向虚拟主机下以"exchange_"开头的交换器发布消息。

最佳实践

1. 登录管理控制台。
2. 进入RabbitMQ管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 进入具体的实例管理页面，点击Vhost管理，在vhost列表页，针对指定的vhost，点击vhost进入具体的权限管理页：



5. 在具体的权限管理页，点击新建，选择指定用户，配置权限为".*"、".*"、".*"，则该用户拥有对虚拟主机下全部资源的配置权限、写权限和读权限：

6. 点击权限列表的操作列的“修改”或“删除”按钮，可以修改或删除对应用户的权限：

虚拟主机限制管理

背景信息

RabbitMQ虚拟主机限制管理是一种用于管理RabbitMQ虚拟主机的功能，它允许管理员对虚拟主机的资源和权限进行限制和管理。

虚拟主机是RabbitMQ中的逻辑隔离单位，它允许在同一台RabbitMQ服务器上创建多个独立的消息队列环境。每个虚拟主机都有自己的队列、交换机、绑定和权限设置。通过虚拟主机限制管理，管理员可以对RabbitMQ服务器上的虚拟主机进行细粒度的控制和管理。这有助于确保不同的应用程序或用户之间的隔离性、安全性和资源利用率，提高整个消息队列系统的可靠性和性能。

操作步骤

1.登录管理控制台。

2.进入RabbitMQ管理控制台。

3.在实例列表页在操作列，目标实例行点击“管理”。

4.点击“集群管理”后点击“虚拟主机限制”到达虚拟主机页面，点击“新建”按钮。

5.点击“新建”后出现以下界面，选择虚拟主机，添加限制策略内容。

限制项参数	说明
max-connections	最大TCP连接。
max-queues	最大队列数。

6.在目标虚拟主机限制行点击“删除”或“修改”，即可删除当前虚拟主机策略。

用户管理

背景信息

客户端访问分布式消息服务RabbitMQ 版服务端时，需要传入用户名和密码进行权限认证，认证通过才允许访问服务端。

操作步骤

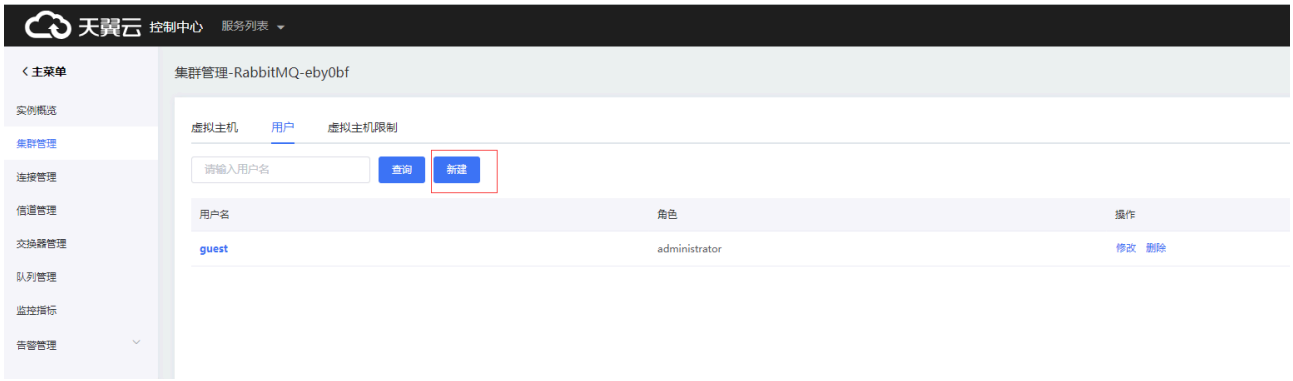
创建用户

1.登录管理控制台。

2.进入RabbitMQ管理控制台。

3.在实例列表页在操作列，目标实例行点击“管理”。

4.点击“集群管理”后点击“用户”到达用户管理页面，点击“新建”按钮。



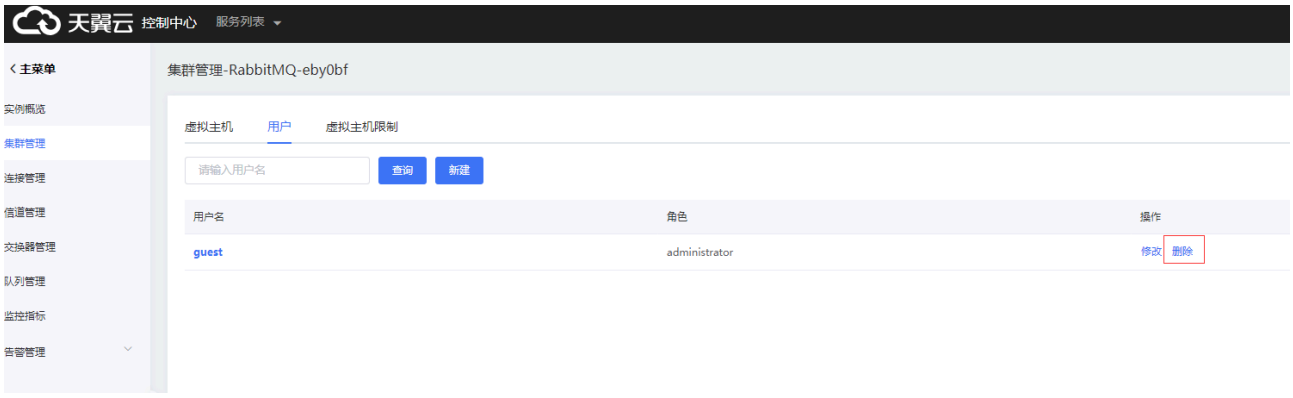
5.点击“新建”后出现以下画面，输入用户密码后点击“确定”即可创建。

修改用户

1.在用户管理页面，在目标用户行点击“修改”，即可重置用户密码。

删除用户

1.在用户管理页面，在目标用户行点击“删除”，即可删除用户。



获取用户token

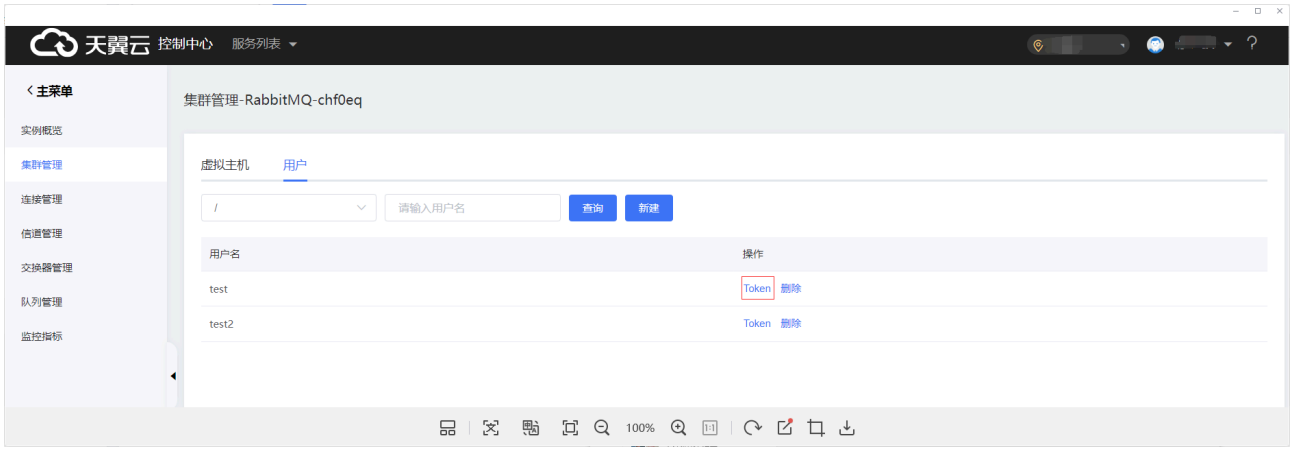
仅云原生引擎支持

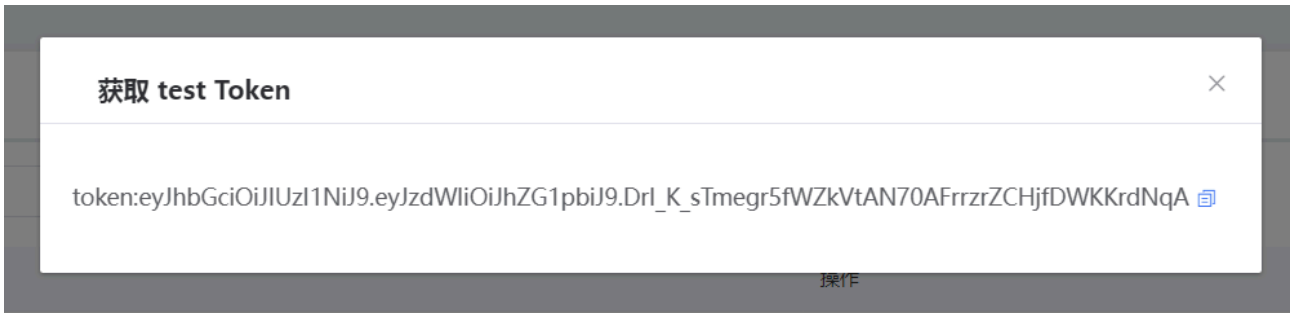
（1）登录管理控制台。

（2）进入RabbitMQ管理控制台。

（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“集群管理”后点击“用户”到达用户管理页面，点击“Token”按钮。





在RabbitMQ中，获取用户令牌（token）的作用如下：

- 认证和授权：用户令牌用于认证和授权用户对RabbitMQ的访问权限。通过获取有效的用户令牌，可以验证用户的身份，并根据其权限配置来限制或授予其对虚拟主机、队列、交换机等资源的访问权限。
- 安全性：通过使用用户令牌进行认证，可以增加RabbitMQ系统的安全性。只有具有有效令牌的用户才能访问和执行相应的操作，从而减少了未经授权的访问和潜在的安全风险。
- 资源管理：用户令牌可以用于管理和限制用户对RabbitMQ资源的使用。通过为每个用户分配独立的令牌，可以控制其对虚拟主机、队列、交换机等资源的使用情况，避免资源滥用或过度消耗。
- 追踪和审计：通过用户令牌，可以追踪和记录用户对RabbitMQ的操作和访问历史。这对于安全审计、故障排查和性能优化等方面非常有用，可以帮助管理员了解系统的使用情况和问题定位。

总之，获取用户令牌是为了实现认证、授权和安全性，以及对RabbitMQ资源的管理和追踪。通过令牌，可以确保只有经过授权的用户才能访问和操作RabbitMQ，提高系统的安全性和可管理性。

连接管理

背景信息

RabbitMQ 连接管理的背景信息如下：

- 生产者连接管理：在 RabbitMQ 中，生产者负责将消息发送到队列中。在高并发的场景下，可能会有大量的生产者与 RabbitMQ 建立连接。因此，连接管理变得至关重要。在这种场景下，需要确保连接的稳定性和可用性，以确保生产者能够持续地发送消息，并避免连接的过度消耗。
- 消费者连接管理：消费者负责从队列中接收消息并进行处理。在高负载的情况下，可能会有大量的消费者连接到 RabbitMQ。在这种场景下，需要进行连接的管理和优化，以确保消费者能够高效地处理消息，并避免过多的连接占用资源。
- 连接池管理：为了更好地管理连接，可以使用连接池来对 RabbitMQ 连接进行复用和管理。连接池可以帮助控制连接的数量，避免频繁地创建和销毁连接，提高系统的性能和资源利用率。通过连接池管理，可以确保连接的可用性，并有效地管理连接的生命周期。
- 连接超时和重连：在网络不稳定或 RabbitMQ 实例发生故障时，连接可能会中断或超时。在这种情况下，需要实现连接的超时和重连机制，以确保连接能够及时地恢复，并保持与 RabbitMQ 的正常通信。通过合理设置连接超时和重连策略，可以提高系统的可靠性和稳定性。

总之，RabbitMQ 连接管理的场景涉及生产者连接管理、消费者连接管理、连接池管理以及连接超时和重连。通过合理的连接管理和优化，可以提高系统的性能、可用性和稳定性，确保消息的可靠传递和处理。

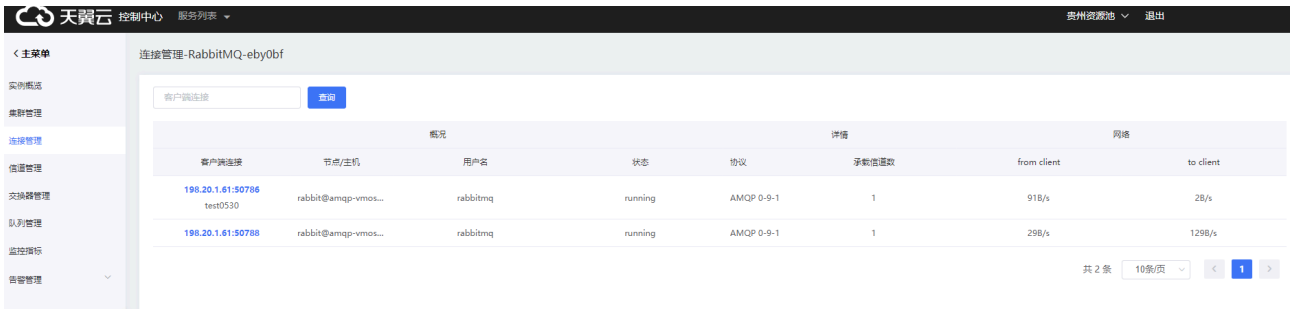
连接列表

（1）登录管理控制台。

（2）进入RabbitMQ管理控制台。

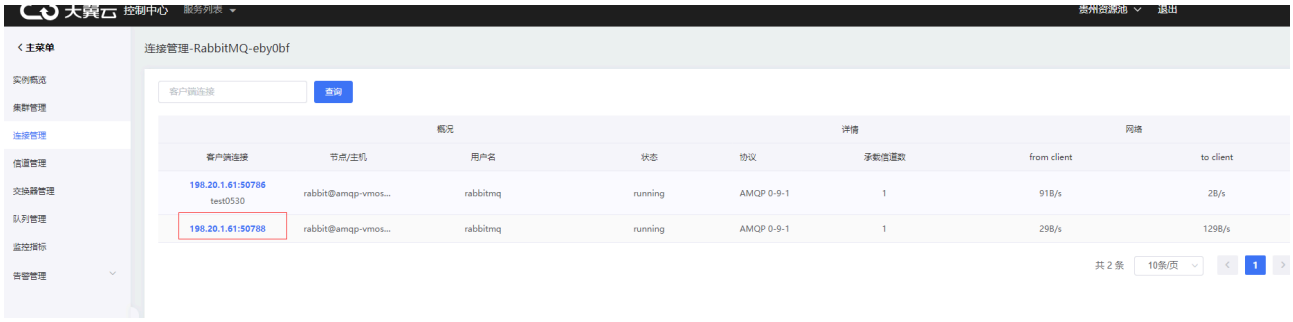
（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“连接管理”进入连接列表。

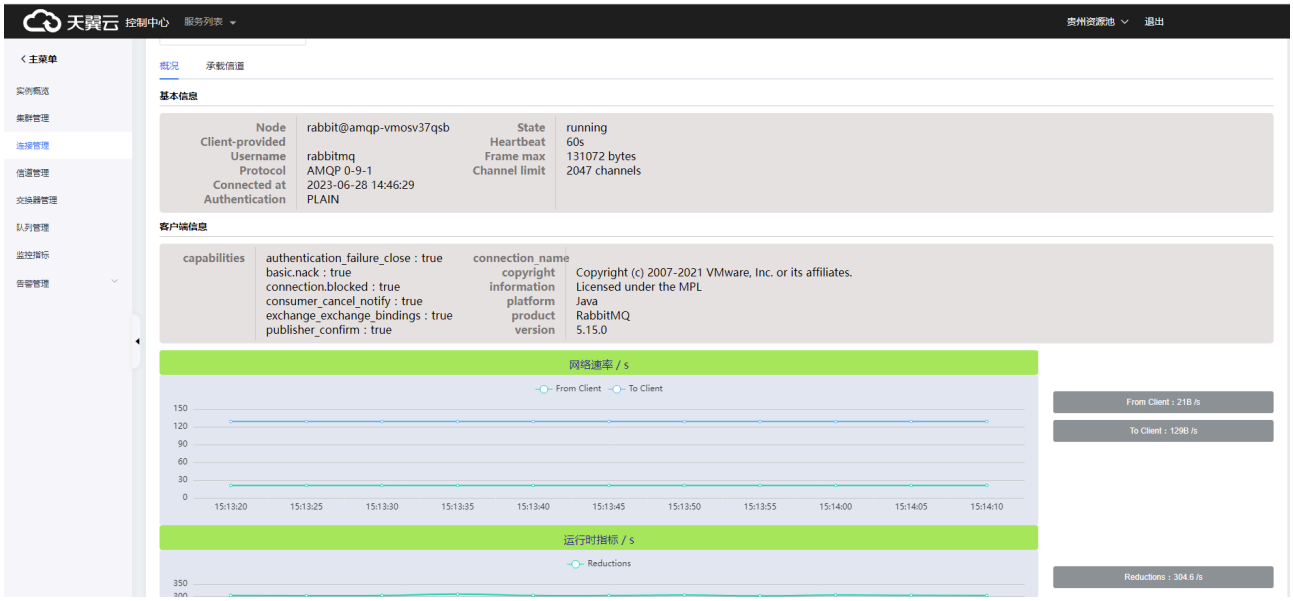


连接概况

（1）点击需要查看的客户端连接名称，进入连接概况。



（2）查看连接概况，包括基本信息、客户端信息、网络速率统计、运行时指标等。



（3）点击“承载信道”，查看连接的信道列表。

信道管理

场景描述

RabbitMQ 信道管理的场景描述如下：

- 并发处理：在高并发的场景下，有大量的消息需要发送或接收。为了提高性能和效率，可以使用多个信道来同时处理消息。通过合理地管理信道，可以实现消息的并发处理，提高系统的吞吐量和响应速度。

- 事务管理：在某些情况下，需要确保消息的原子性操作，即要么全部成功，要么全部失败。通过使用信道的事务机制，可以将多个操作封装在一个事务中，以确保消息的一致性和可靠性。在需要严格的数据一致性的场景下，信道的事务管理非常重要。
- 消息确认机制：在消息的发送和接收过程中，可能会出现网络故障或其他错误导致消息丢失或处理失败。为了确保消息的可靠性，可以使用信道的消息确认机制。通过确认机制，消息的发送者可以收到关于消息是否已经被成功处理的确认信息，从而可以进行相应的处理。
- 限流控制：在消息的发送和接收过程中，可能会出现瞬时的高负载情况，导致系统资源的过度消耗。为了避免这种情况，可以使用信道的限流控制机制。通过限制每个信道的消息数量或速率，可以控制系统的负载，保护系统的稳定性和可用性。

总之，RabbitMQ 信道管理的场景涉及并发处理、事务管理、消息确认机制和限流控制。通过合理地管理和优化信道，可以提高系统的性能、可靠性和稳定性，确保消息的可靠传递和处理。

信道列表

（1）登录管理控制台。

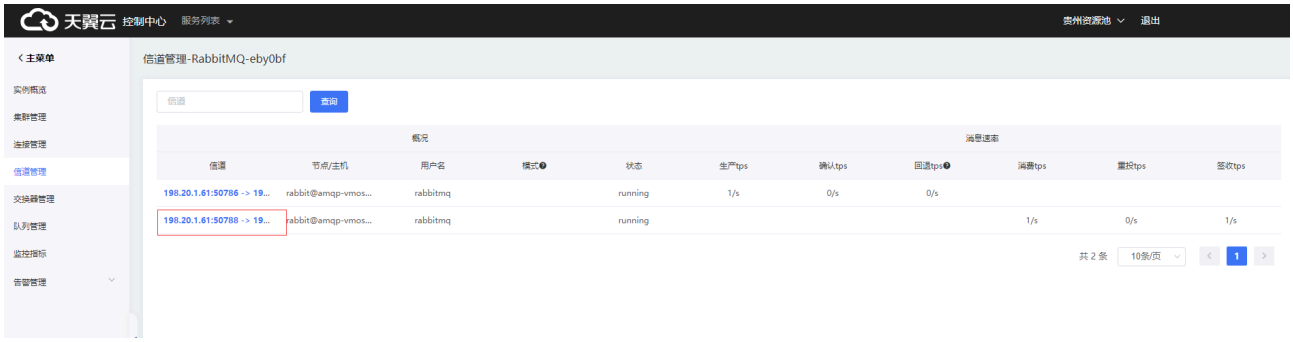
（2）进入RabbitMQ管理控制台。

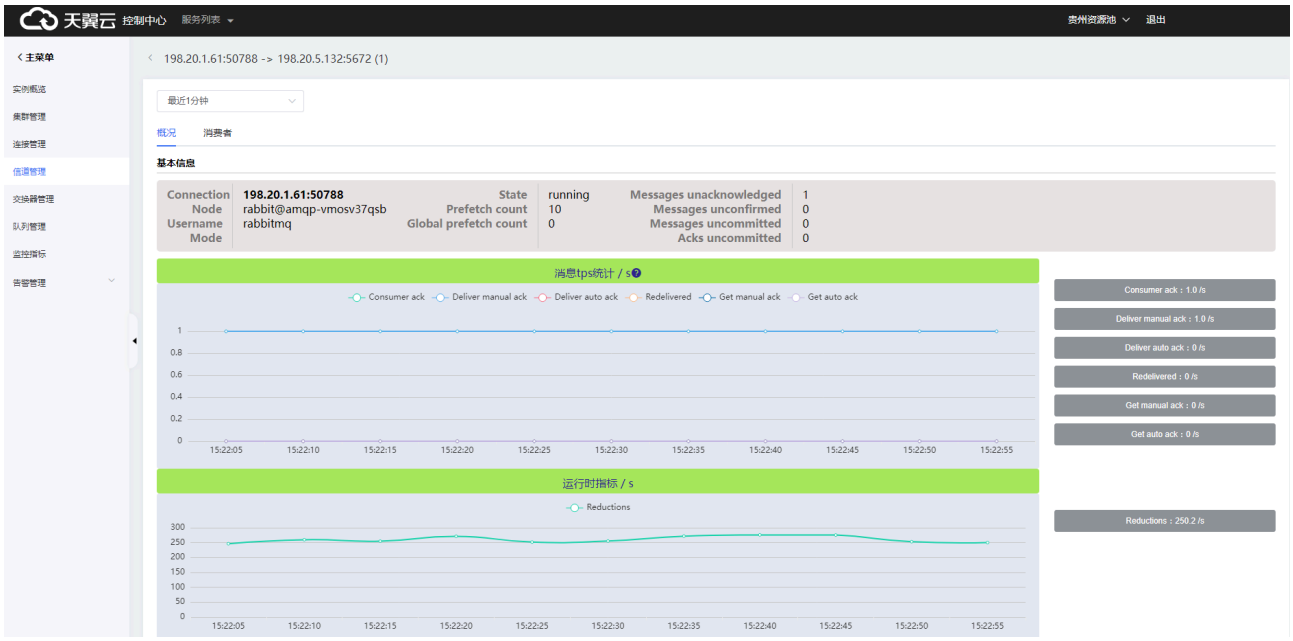
（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“信道管理”进入信道列表。

信道概况

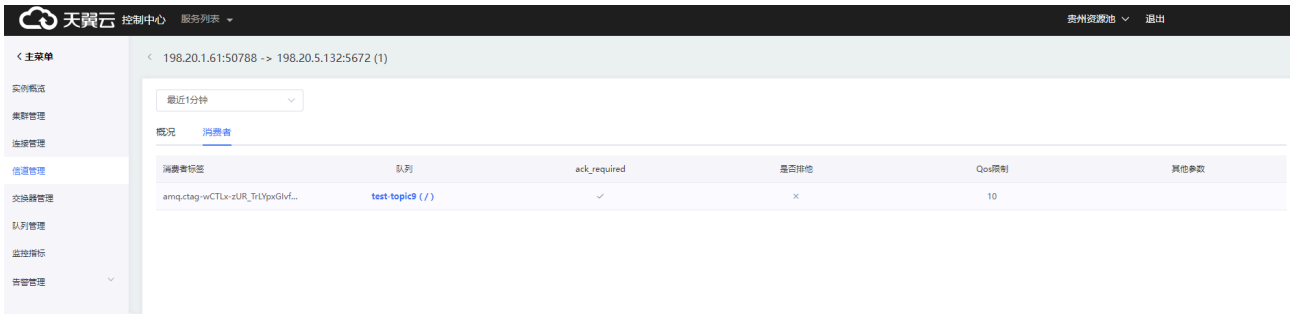
（1）点击需要查看的信道名称，进入信道概况。





消费者

（1）点击“消费者”，查看信道的消费者列表。



对用户而言，在RabbitMQ中，查看信道的消费者有以下作用：

监控消费者状态：通过查看信道的消费者，可以监控消费者的状态和活动情况。你可以了解到哪些消费者正在活动中，以及它们消费消息的速度和效率。这有助于实时监控系统的消费者负载和性能状况。

故障排查：如果出现消息消费问题或故障，查看信道的消费者可以帮助你定位问题所在。你可以检查消费者是否正常连接到队列，并查看消费者的消费进度和处理速度。这有助于排查消费者故障、网络问题或处理逻辑错误等情况。

性能优化：通过查看信道的消费者，你可以了解到消费者的消费速度和负载情况。如果某个消费者的消费速度较慢或负载较高，你可以考虑进行性能优化，例如增加消费者数量、调整消费者的并发处理能力或优化消费者的处理逻辑。

资源管理：查看信道的消费者可以帮助你管理系统资源。你可以了解到消费者的数量和资源占用情况，以便合理分配系统资源和调整系统配置。这有助于避免资源过度消耗和优化系统的性能。

总之，通过查看信道的消费者，用户可以监控消费者状态、进行故障排查、优化性能和管理资源。这对于保证系统的可靠性、性能和可管理性非常重要。

操作策略管理

背景信息

RabbitMQ操作策略管理是一种用于管理RabbitMQ服务器的功能，它允许管理员定义和控制各种操作策略以满足特定的需求和约束。

操作策略可以应用于多个方面，包括队列、交换机、绑定和连接等。通过操作策略管理，管理员可以定义以下内容：

● 长度限制：管理员可以设置队列的最大长度，以防止队列无限增长导致资源耗尽。当队列达到指定的长度限制时，可以选择丢弃新的消息或者拒绝新的连接。

● 时间限制：管理员可以设置队列中消息的最大存活时间，以防止消息在队列中长时间滞留。一旦消息超过指定的时间限制，可以选择将其丢弃或者转发到其他队列。

● 内存限制：管理员可以设置队列或交换机的最大内存使用量，以防止RabbitMQ服务器的内存资源被过度消耗。当达到指定的内存限制时，可以选择丢弃消息或者拒绝新的连接。

● 消息优先级：管理员可以为消息设置优先级，以确保重要的消息能够优先处理。可以通过操作策略管理来定义消息优先级的规则和行为。

● 连接限制：管理员可以设置连接的最大数量，以限制同时连接到RabbitMQ服务器的客户端数量。这可以用于控制系统的负载和资源消耗。

通过操作策略管理，管理员可以根据实际需求和约束对RabbitMQ服务器进行细粒度的控制和管理。这有助于提高系统的可靠性、性能和可伸缩性，并确保消息队列系统能够适应各种场景和负载。

操作步骤

操作策略仅以下29节点支持配置：

上海6、北京4、内蒙5、西安3、重庆2、拉萨3、南京3、雄安2、晋中、郴州2、成都4、杭州2、上海7、西安4、福州3、泉州1、芜湖2、北京5、中卫2、贵州3、九江、内蒙6、

武汉4、佛山3、福州4、昆明2、海口2、保定、辽阳1。

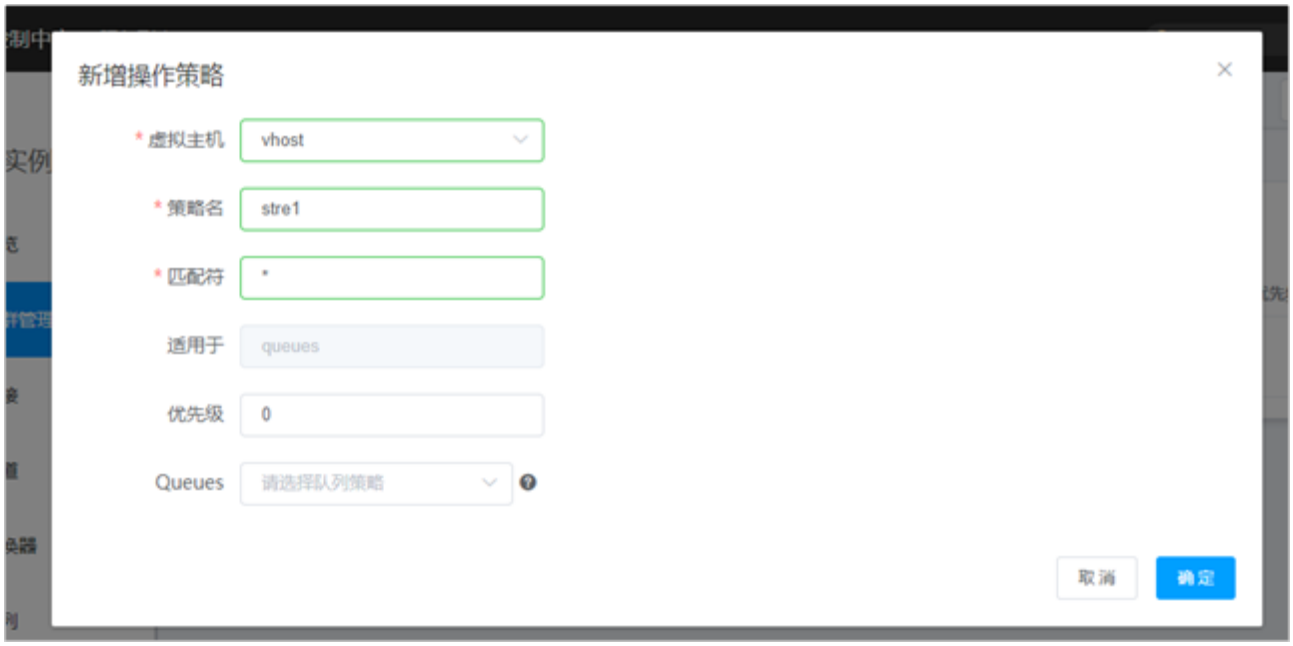
1.登录管理控制台。

2.进入RabbitMQ管理控制台。

3.在实例列表页在操作列，目标实例行点击“管理”。

4.点击“集群管理”后点击“操作策略”到达操作策略管理页面，点击“新建”按钮。

5.点击“新建”后出现以下界面，选择虚拟主机，添加策略名、匹配符号，和策略内容。



Queues参数	说明
Message TTL	消息过期时间：number型(单位:ms)
Auto expire	队列过期时间，过期后队列自动删除：number型(单位:ms)
Max length	队列能保存的最大消息数：number型(单位:个)
Max length bytes	队列能保存的最大消息量：number型(单位:字节)
Overflow behaviour	超过队列的最大设定值后消息接收策略： • drop-head：删除头部消息,一般就是最早发送的消息，保证队列可用 • reject-publish：拒绝接受新的消息，保证消息不丢失

6.在目标操作策略所在行，点击“删除”或“修改”即可删除或修改当前操作策略。

交换器管理

背景信息

生产者将消息发送到交换器，由交换器将消息路由到一个或多个队列中（或者丢弃）。交换器根据Routing Key和Binding Key将消息路由到Queue。不同类型的交换器的路由规则不同。

操作步骤

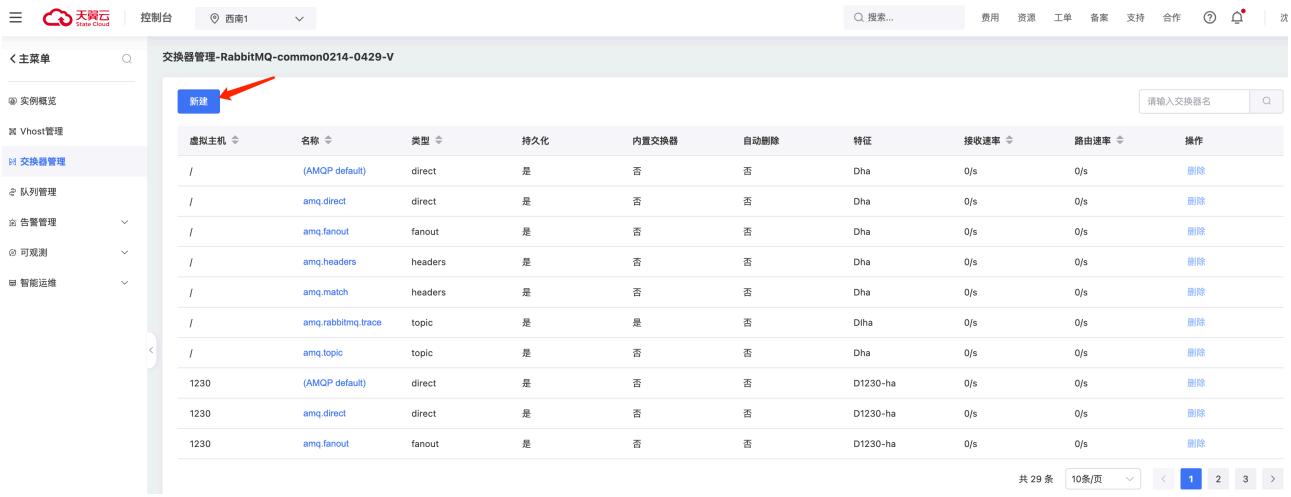
新建交换器

1.登录管理控制台。

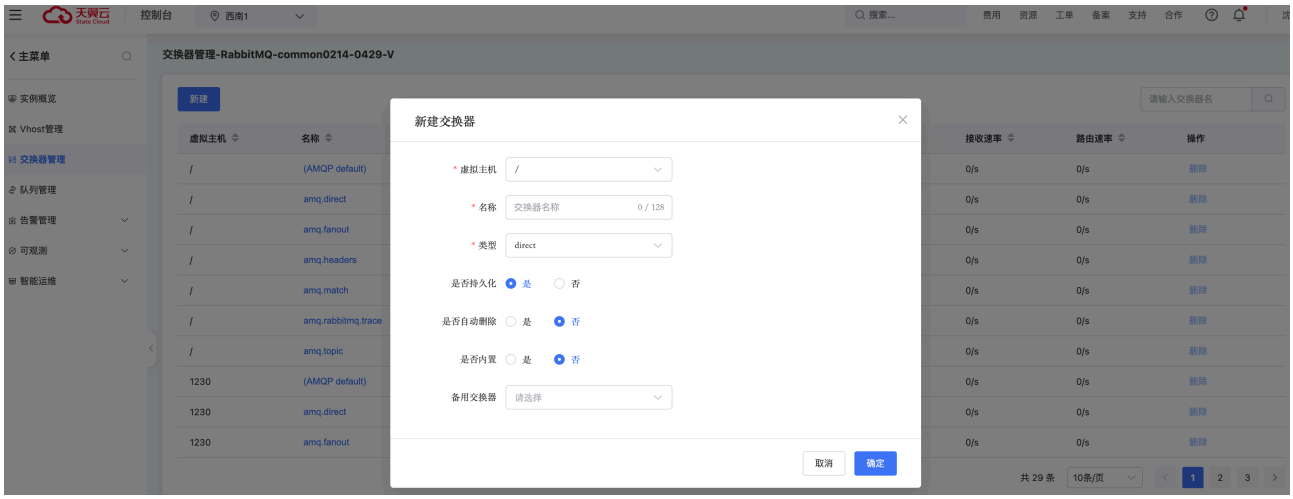
2.进入RabbitMQ管理控制台。

3.在实例列表页在操作列，目标实例行点击“管理”。

4.点击“交换器管理”后，点击“新建”按钮。



5.点击“新建”后出现以下窗口，选择虚拟主机，添加交换器名字，选择交换器类型和其他参数，然后点击“确定”即可新建交换器。



各参数说明如下

参数	描述
虚拟主机	选择创建交换器所属的虚拟主机
名称	交换器名称。以amq.开头的为保留字段，因此不能使用。例如：。
类型	Exchange类型。
是否持久化	交换器是否持久化到磁盘
是否自动删除	如果是，交换器将在至少一个队列或交换器绑定到该交换器，然后所有队列或交换器都已解除绑定时删除。
是否内置	如果是，客户端不能直接发布到这个交换器。它只能与其他交换器绑定使用。
其他参数	Alternate exchange：备份交换器是为了实现没有路由到队列的消息，声明交换机的时候添加属性alternate-exchange，声明一个备用交换机，一般声明为fanout类型，这样交换机收到路由不到队列的消息就会发送到备用交换机绑定的队列中。

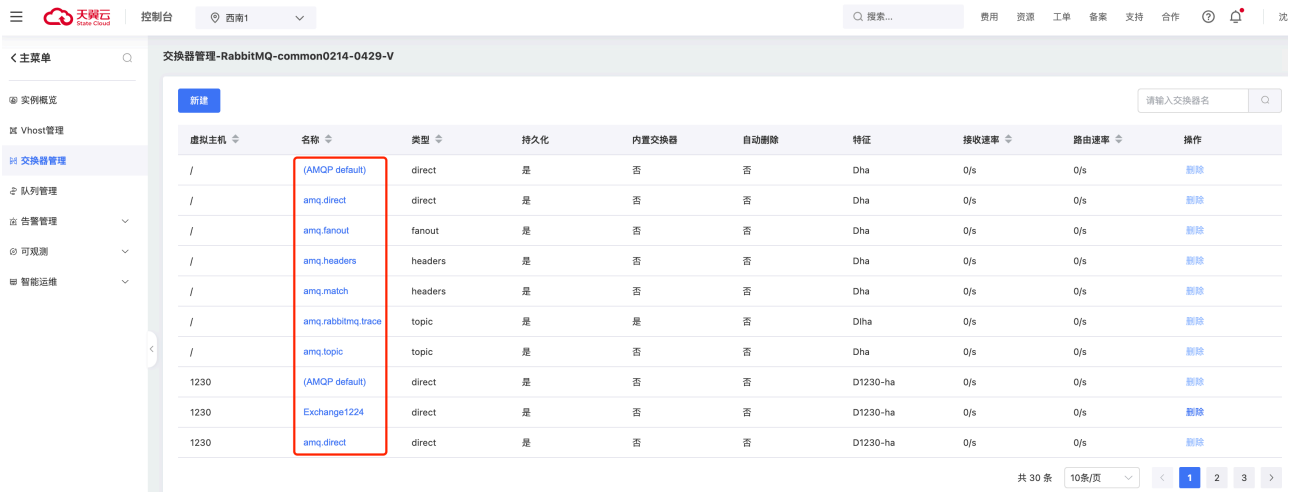
其中交换机类型如下表所示

交换机类型	说明
Direct	完全根据key进行投递的叫做Direct交换机。如果Routing key匹配, 那么Message就会被传递到相应的queue中。其实在queue创建时，它会自动的以queue的名字作为routing key来绑定那个exchange。例如，绑定时设置了Routing key为”abc”，那么客户端提交的消息，只有设置了key为”abc”的才会投递到队列。
Fanout	不需要key的叫做Fanout交换机。它采取广播模式，一个消息进来时，投递到与该交换机绑定的所有队列。

交换机类型	说明
Topic	对key进行模式匹配后进行投递的叫做Topic交换机。比如符号”#”匹配一个或多个词，符号”“匹配正好一个词。例如”abc.#”匹配”abc.def.ghi”，”abc.”只匹配”abc.def”。

查看交换器

（1）点击目标交换器名称，即可查看交换器概况。

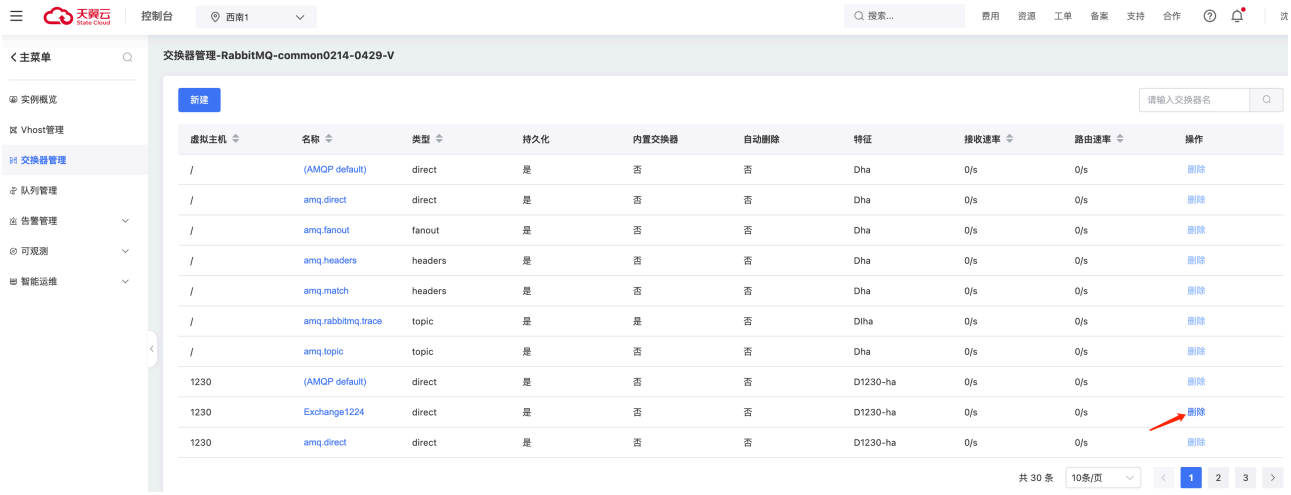


（2）点击“绑定信息”，即可查看交换器的绑定信息。

（3）点击“被绑定信息”，即可查看交换器的被绑定信息。

删除交换器

（1）在目标交换器点击“删除”，即可删除交换器。



队列管理

背景信息

分布式消息服务RabbitMQ版的消息都会被投入到一个或多个队列中。Consumer Tag是消费者客户端的标识符。您可以在分布式消息服务 RabbitMQ 版的消费者客户端设置Consumer Tag。

操作步骤

创建队列

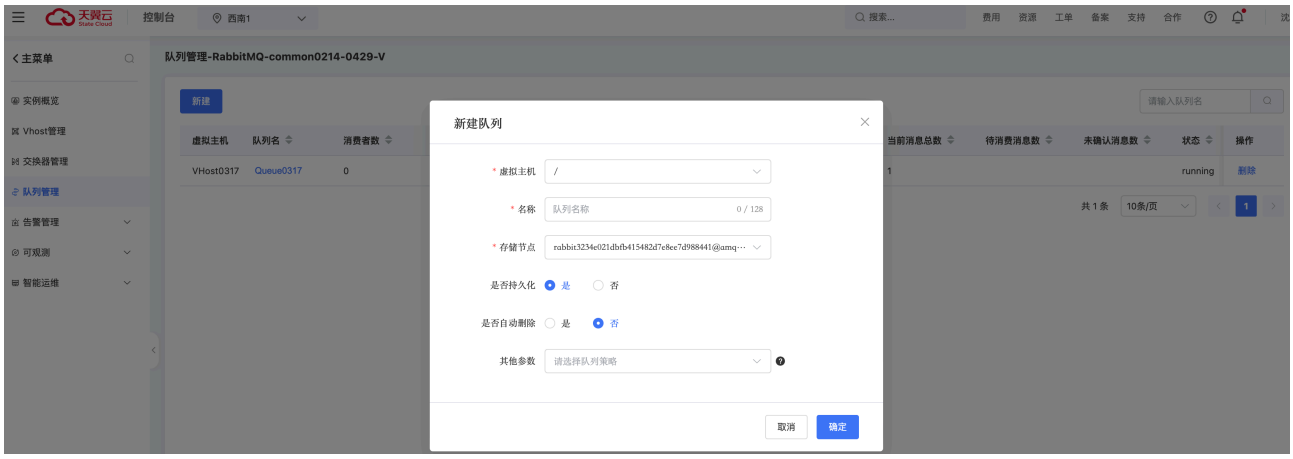
1.登录管理控制台。

2.进入RabbitMQ管理控制台。

3.在实例列表页在操作列，目标实例行点击“管理”。

4.点击“队列管理”后，点击“新建”按钮。

5.点击“新建”后出现以下窗口，选择虚拟主机，输入队列名字，选择存储节点，然后点击确定即可创建队列。



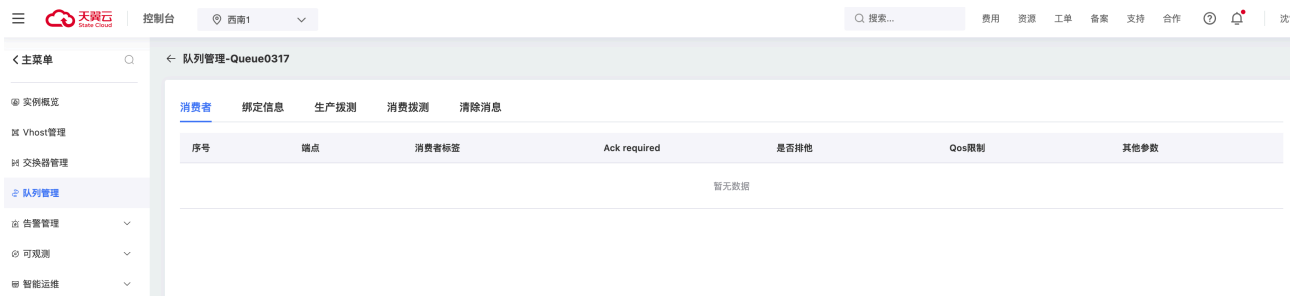
参数说明如下表所示：

参数	描述
虚拟主机	选择创建队列所属的虚拟主机
名称	队列的名称。以amq.开头的为保留字段，因此不能使用。例如：amq.test。
存储节点	队列数据存储节点
是否持久化	队列元数据是否持久化到磁盘
是否自动删除	最后一个Consumer取消订阅后，Queue是否自动删除。
其他参数	Message TTL-消息过期时间：number型(单位:ms)Auto expire-队列过期时间，过期后队列自动删除：number型(单位:ms)Max length-队列能保存的最大消息数：number型(单位:个)Max length bytes-队列能保存的最大消息量：number型(单位:字节)Overflow behaviour超过队列的最大设定值后消息接收策略：drop-head,reject-publishdrop-head：删除头部消息,一般就是最早发送的消息，保证队列可用reject-publish：拒绝接受新的消息，保证消息不丢失Dead letter exchange死信交换器名称Dead letter routing key死信路由键Maximum priority队列最大优先级：要开启消息的优先级，必须设置消息所在队列的优先级Lazy mode队列惰性模式：default、lazydefault：默认值，普通队列lazy：惰性队列，尽可能将消息存到磁盘中，会引起I/O操作比较多，内存消耗极少（有大量堆积的持久化消息建议使用）Master Locator队列保存位置：client-local、min-masters、randomclient-local：队列创建时所用连接的节点min-masters： 集群中节点主数量最少的节点random：由rabbitmq服务器随机指定一个节点

查看队列

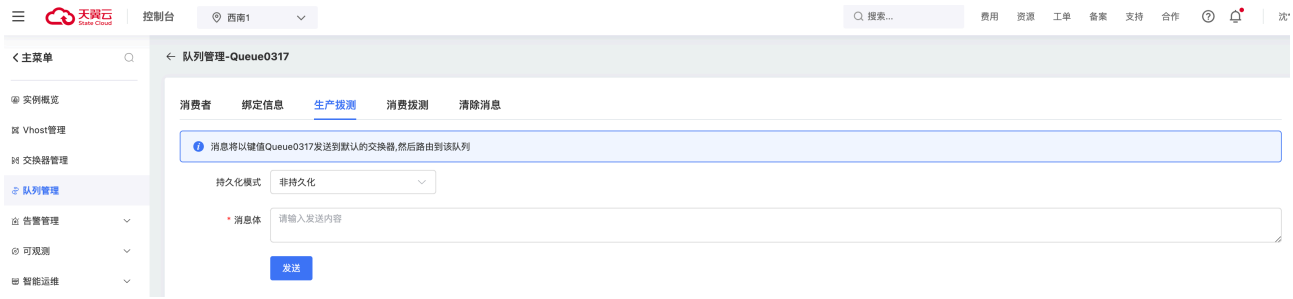
（1）点击目标队列名称，即可查看队列概况。

（2）点击“消费者”，即可查看队列的消费者。

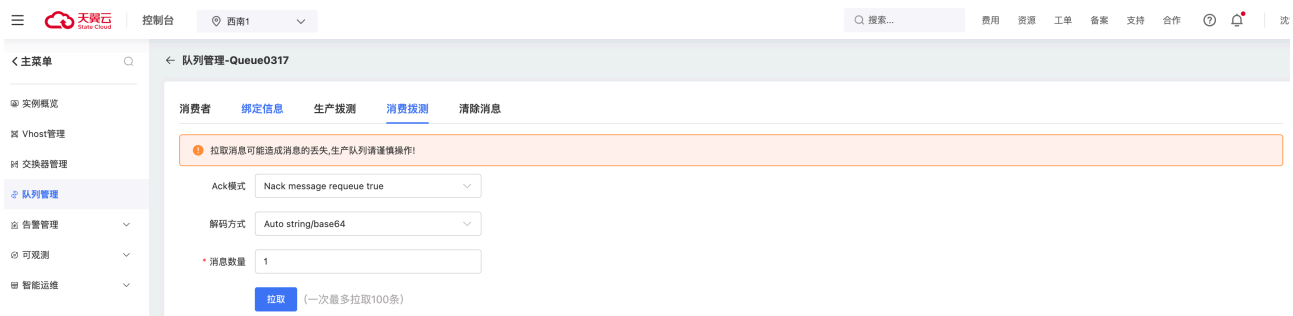


（3）点击“绑定信息”，即可查看队列的绑定信息。

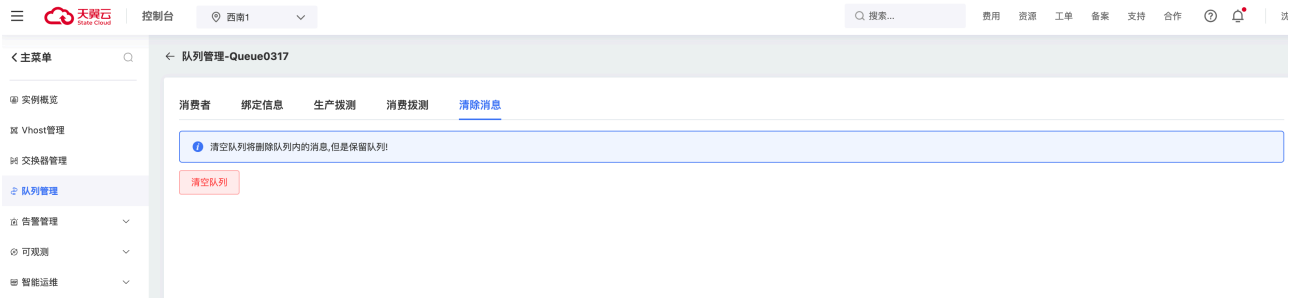
（4）点击“生产拨测”，即可进入队列的生产拨测页面。可通过生产拨测发送消息到队列。



（5）点击“消费拨测”，即可进入队列的消费拨测页面。通过消费拨测可以拉取队列的消息。



（6）点击清空消息”，即可进入清空消息页面，再点击清空队列，可以清除队列的消息。



删除队列

在目标队列点击删除，即可删除队列。

注意事项： 删除队列时，队列中未被消费的消息会被同时删除，且不可恢复。请您谨慎操作。

监控指标

监控项

分布式消息服务RabbitMQ 版监控指标支持以下监控项：

资源类型	监控项	单位	指标名	维度
实例	每秒消息流入数	个/秒	publishRate	实例
实例	每秒消息流出数	个/秒	deliverRate	实例
实例	消费者数量	个	consumers	实例
实例	连接数量	个	connections	实例
实例	通道数量	个	channels	实例
虚拟主机	每秒消息流入数	个/秒	publishRate	实例、虚拟主机
虚拟主机	每秒消息流出数	个/秒	deliverRate	实例、虚拟主机
虚拟主机	消费者数量	个	consumers	实例、虚拟主机
虚拟主机	连接数量	个	connections	实例、虚拟主机
虚拟主机	通道数量	个	channels	实例、虚拟主机
队列	每秒消息流入数	个/秒	publishRate	实例、虚拟主机、队列

资源类型	监控项	单位	指标名	维度
队列	每秒消息流出数	个/秒	deliverRate	实例、虚拟主机、队列
队列	消费者数量	个	consumers	实例、虚拟主机、队列
队列	消息堆积量	个	messages	实例、虚拟主机、队列
交换器	每秒消息流入数	个/秒	publishRate	实例、虚拟主机、交换器
交换器	每秒消息流出数	个/秒	deliverRate	实例、虚拟主机、交换器

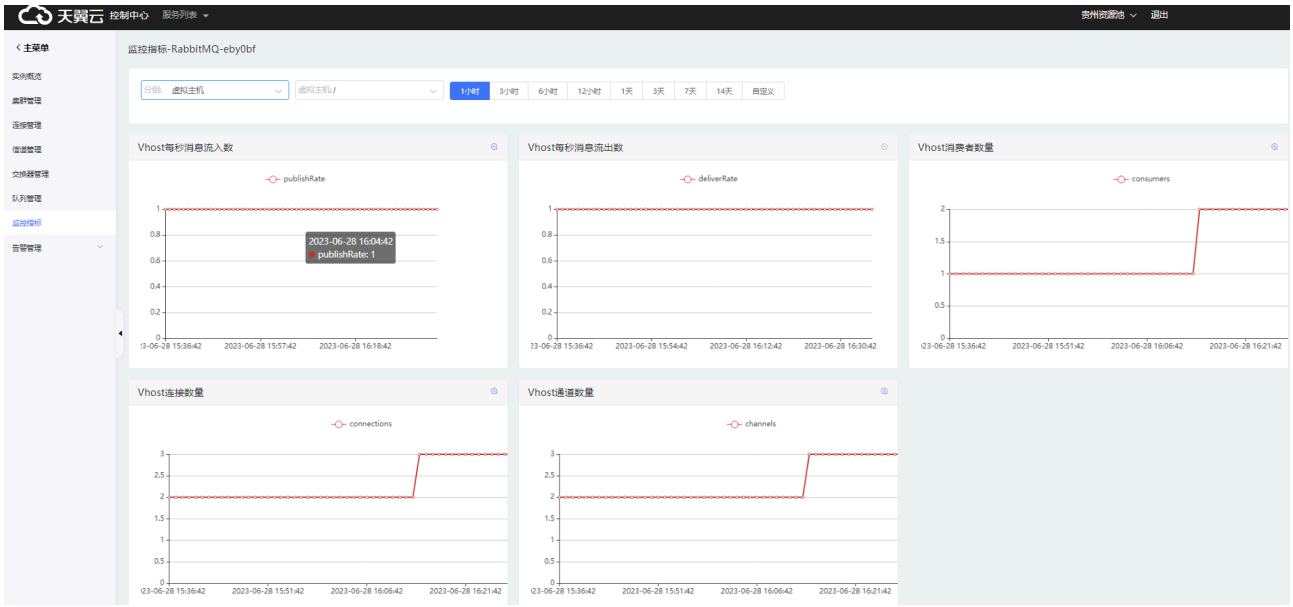
查看监控数据

（1）登录管理控制台。

（2）进入RabbitMQ管理控制台。

（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“监控指标”后，选择对应的监控项。



仪表盘

使用场景

RabbitMQ仪表盘支持实时监控消息流、排查生产消费异常、配置与维护队列 / 交换机 / 绑定、权限管理及集群状态查看，是 RabbitMQ 运维与开发调试的核心工具。

指标说明

指标	指标含义
可消费消息数	队列中已经准备好等待消费者去获取和处理的消息数量。
连接数	当前与 RabbitMQ 服务器建立的 TCP 连接总数。每个生产者或消费者客户端都需要与 RabbitMQ 建立一个 TCP 连接。
信道数	在所有 TCP 连接上打开的 AMQP 信道总数。信道是在 TCP 连接内部建立的虚拟连接，用于多路复用，以减少建立和关闭 TCP 连接的开销。
消费者数	当前所有队列上注册的消费者总数。一个消费者可以监听一个或多个队列。
交换器生产速率	单位时间内发送到某个交换器的消息数量。
交换器消费速率	单位时间内从某个交换器路由到其绑定队列的消息数量。
队列生产速率	单位时间内发送到某个队列的消息数量。
队列消费速率	单位时间内从某个队列成功投递给消费者并收到确认（ACK）的消息数量。
队列可消费消息数	某个队列中当前处于 "Ready" 状态的消息数量。
队列消费者数	当前正在监听某个队列的消费者数量。
VHost连接数	当前连接到某个特定 VHost 的 TCP 连接总数。
VHost信道数	当前在某个特定 VHost 的所有连接上打开的信道总数。
VHost每个连接的信道数	某个 VHost 的信道总数除以其连接总数得到的平均值。

如何通过仪表盘发现问题

以下指标是最基础也是最重要的指标，用于快速判断RabbitMQ实例健康状况。

指标1：可消费消息数 （Ready Messages）

- 正常情况：这个数值应该保持在一个相对稳定的水平，或者有轻微波动。
- 异常情况：如果这个数值持续快速增长，通常意味着生产速度远大于消费速度，消费者处理能力不足、数量不够，或者出现了故障。这是消息堆积的直接体现。

指标2：连接数 （Connections）

- 正常情况：连接数应与系统设计的客户端数量匹配。
- 异常情况1：连接数过多：可能导致服务器资源（内存、文件描述符）耗尽，影响性能甚至崩溃。这通常是由于客户端没有正确关闭连接（连接泄漏）。
- 异常情况2：连接数骤降：可能表示大量客户端离线或网络中断。

指标3：信道数 （Channels）

- 正常情况：信道数通常远大于连接数。
- 异常情况1：信道数过多：即使连接数不多，过多的信道也会增加服务器的内存消耗和管理负担。
- 异常情况2：信道数与连接数比例异常：如果每个连接只使用一个信道，可能没有充分利用 TCP 连接的多路复用优势。

指标4：消费者数（Consumers）

- 正常情况：消费者数量应足以处理预期的消息流量。
- 异常情况1：消费者数为零：对于需要即时处理的队列来说是严重问题，会导致消息堆积。
- 异常情况2：消费者数不足：会导致消息处理缓慢，Ready 消息数增长。
- 异常情况3：消费者数过多：可能导致资源竞争，反而降低整体效率，需要根据实际情况进行压测和调优。

高级特性

惰性队列

使用场景

RabbitMQ从3.6.0版本开始引入了惰性队列（Lazy Queue）的概念。惰性队列会尽可能的将消息存入磁盘中，而在消费者消费到相应的消息时才会被加载到内存中，它的一个重要的设计目标是能够支持更长的队列，即支持更多的消息存储。当消费者由于各种各样的原因（比如消费者下线、宕机亦或者是由于维护而关闭等）而致使长时间内不能消费消息造成堆积时，惰性队列就很有必要了

默认情况下，当生产者将消息发送到RabbitMQ的时候，队列中的消息会尽可能的存储在内存之中，这样可以更加快速的将消息发送给消费者。即使是持久化的消息，在被写入磁盘的同时也会在内存中驻留一份备份。当RabbitMQ需要释放内存的时候，会将内存中的消息换页至磁盘中，这个操作会耗费较长的时间，也会阻塞队列的操作，进而无法接收新的消息。虽然RabbitMQ的开发者们一直在升级相关的算法，但是效果始终不太理想，尤其是在消息量特别大的时候

惰性队列会将接收到的消息直接存入文件系统中，而不管是持久化的或者是非持久化的，这样可以减少了内存的消耗，但是会增加I/O的使用，如果消息是持久化的，那么这样的I/O操作不可避免，惰性队列和持久化消息可谓是“最佳拍档”。注意如果惰性队列中存储的是非持久化的消息，内存的使用率会一直很稳定，但是重启之后消息一样会丢失。

设置惰性队列

队列具备两种模式：default和lazy，默认模式为default。lazy模式即为惰性队列的模式，可以通过调用channel.queueDeclare方法的时候在参数中设置。

以下示例演示在Java客户端通过调用channel.queueDeclare设置惰性队列。

```
Map args = new HashMap();

args.put("x-queue-mode","lazy"); channel.queueDeclare("test_queue", false,
 false, false, args);
```

消息持久化

使用场景

在生产过程中，难免会发生服务器宕机的事情，RabbitMQ也不例外，可能由于某种特殊情况下的异常而导致RabbitMQ宕机从而重启，那么这个时候对于消息队列里的数据，包括交换机、队列以及队列中存在消息恢复就显得尤为重要了。RabbitMQ本身带有持久化机制，包括交换机、队列以及消息的持久化。持久化的主要机制就是将信息写入磁盘，当RabbtiMQ服务宕机重启后，从磁盘中读取存入的持久化信息，恢复数据。

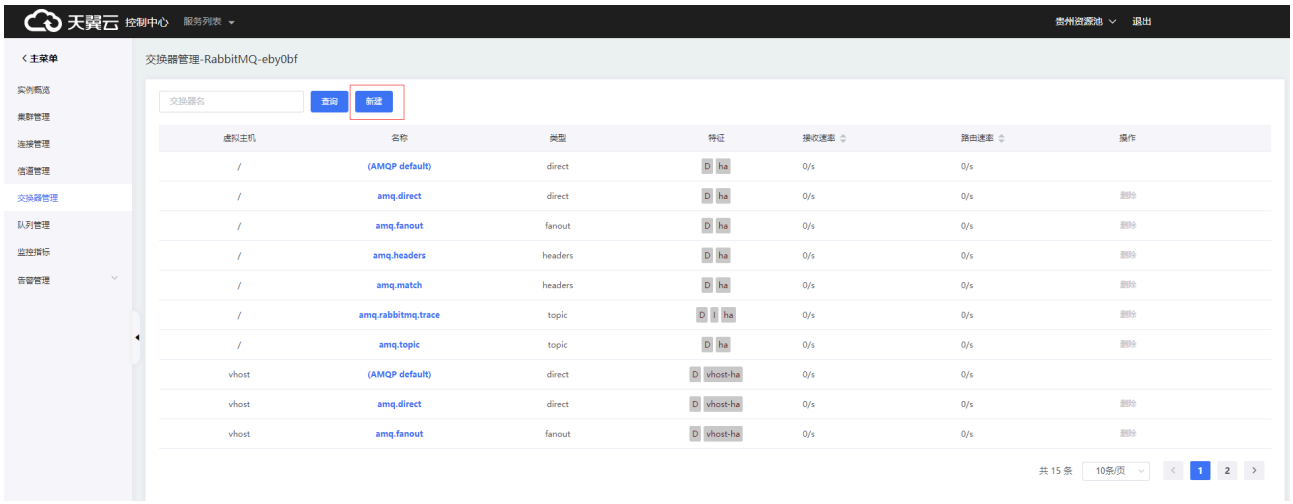
设置交换器持久化

（1）登录管理控制台。

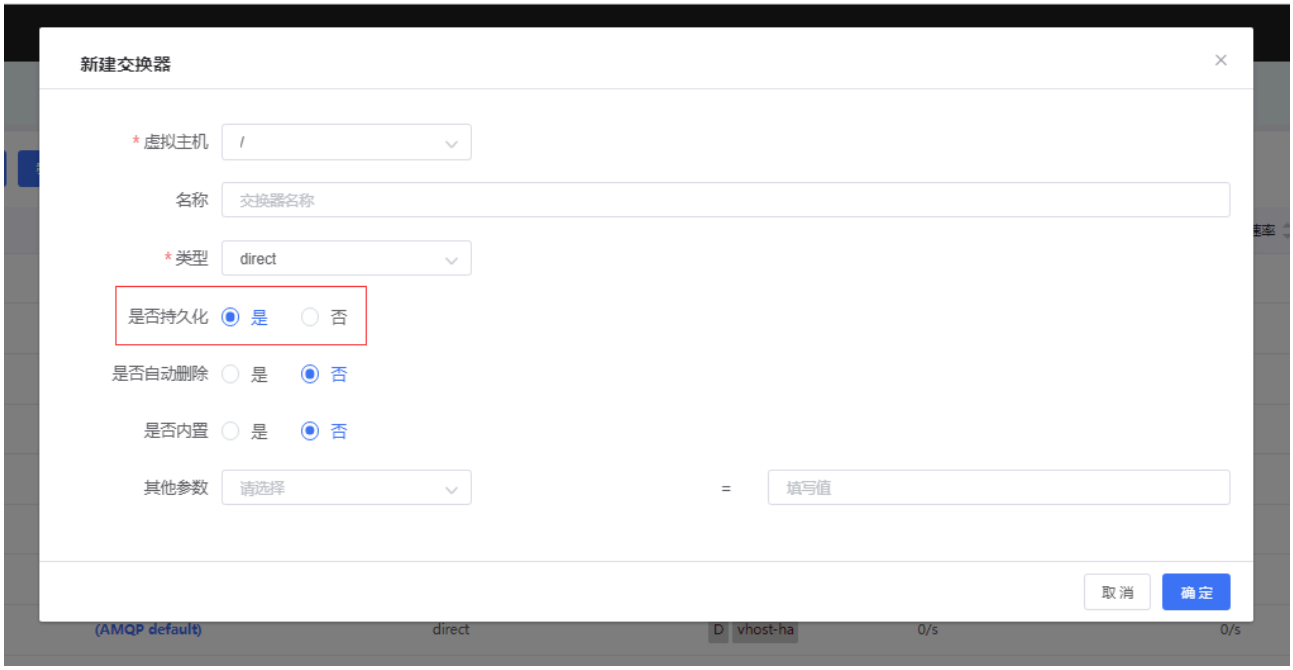
（2）进入RabbitMQ管理控制台。

（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“交换器管理”后，点击“新建”按钮。



（5）点击“新建”后出现以下窗口，是否持久化选择是。



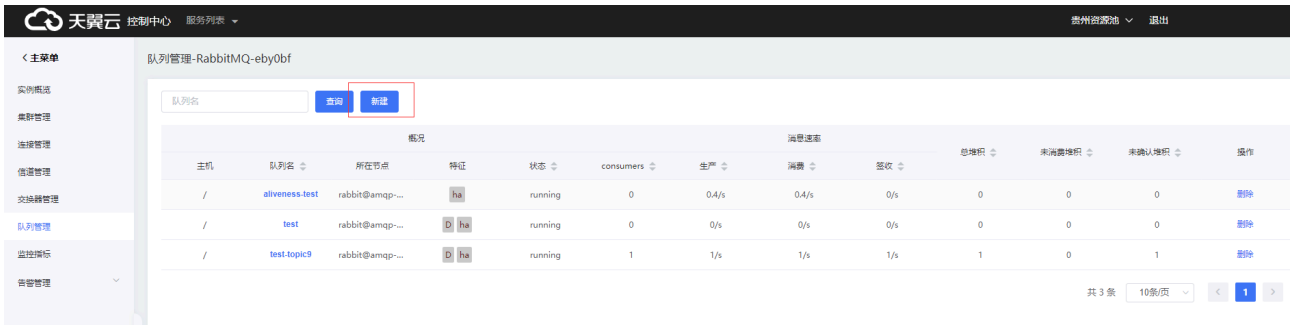
设置队列持久化

（1）登录管理控制台。

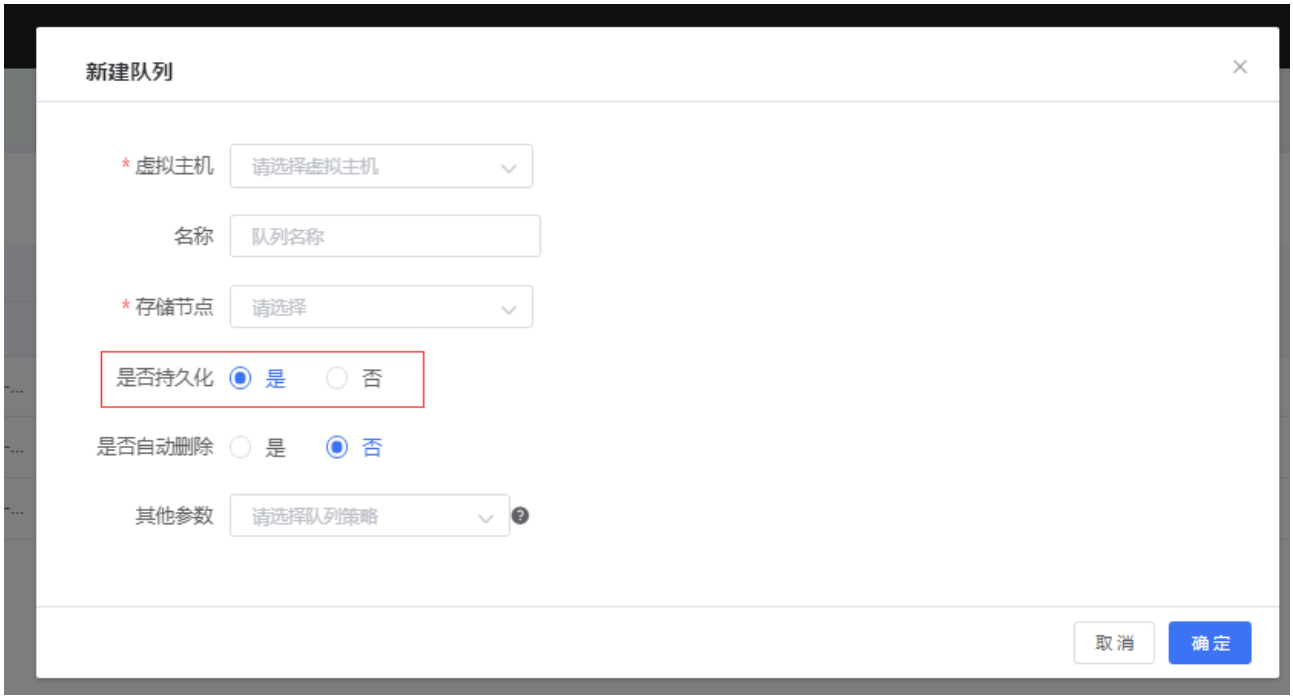
（2）进入RabbitMQ管理控制台。

（3）在实例列表页在操作列，目标实例行点击“管理”。

（4）点击“队列管理”后，点击“新建”按钮。



（5）点击“新建”后出现以下窗口，是否持久化选择是。



设置消息持久化

Queue设置为持久化后，通过设置“MessageProperties”为“PERSISTENT_TEXT_PLAIN”来向Queue发送持久消息。

java客户端示例为：

```
import com.rabbitmq.client.MessageProperties;
channel.basicPublish("", "my_queue",MessageProperties.PERSISTENT_TEXT_PLAIN,
 message.getBytes());
```

死信和TTL

死信

称为死信的信息，需要如下几个条件：

- 消息被消费者拒绝（通过basic.reject 或者 back.nack），并且设置 requeue=false。
- 消息过期，队列设置了TTL（Time To Live）时间并且消息过期。
- 超过了队列的长度限制消息被丢弃。
- 为队列配置死信交换机，并在申明队列时指定“x-dead-letter-exchange”和“x-dead-letter-routing-key”参数。队列根据“x-dead-letter-exchange”将死信消息发送到死信交换机中，并根据“x-dead-letter-routing-key”为死信消息设置死信路由Key。

以下示例演示在Java客户端配置死信交换机和路由

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

TTL

RabbitMQ可以对消息和队列设置TTL. 目前有两种方法可以设置。第一种方法是通过队列属性设置，队列中所有消息都有相同的过期时间。第二种方法是对消息进行单独设置，每条消息TTL可以不同。如果上述两种方法同时使用，则消息的过期时间以两者之间TTL较小的那个数值为准。消息在队列的生存时间一旦超过设置的TTL值，就称为dead message， 消费者将无法再收到该消息。

设置队列TTL

通过队列属性设置消息TTL的方法是在queue.declare方法中加入x-message-ttl参数，单位为ms.

以下示例演示在Java客户端设置队列TTL。

```
Map<String, Object>  argss = new HashMap<String, Object>();
argss.put("x-message-ttl",6000);
channel.queueDeclare(queueName, durable, exclusive, autoDelete, argss);
```

设置消息TTL

针对每条消息设置TTL的方法是在basic.publish方法中加入expiration的属性参数，单位为ms.

以下示例演示在Java客户端通过队列属性设置消息TTL。

```
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();
builder.deliveryMode(2);
builder.expiration("6000");
AMQP.BasicProperties  properties = builder.build();
channel.basicPublish(exchange
Name,routingKey,mandatory,properties,"ttlTestMessage".getBytes());
```

RabbitMQ消息确认机制

消息确认

RabbitMQ消息确认机制分为发送消息确认和消息接收确认。

发送消息确认

生产者确认，即为生产者发送消息后，等待服务端确认。

以下示例演示在Java客户端配置生产者确认过程：

```
try {
 channel.confirmSelect();
 channel . basicPublish( "test_exchange" , " routingKey" , null , "publisher
 confirm message".getBytes());
 if (!channel.waitForConfirms()) {
  System.out.println( "send message failed") ;
  // do something else
 }else{
  // do something
 }
} catch (InterruptedException e) {
 e.printStackTrace() ;
}
```

消息接收确认(ACK)

消费者收到消息后需要对 RabbitMQ Server 进行消息ACK确认，RabbitMQ根据确认信息决定是删除队列中的该信息还是重新发送。

重点在于消费者的下面两个方法

channel.basicAck 消费者签收

channel.basicNAck 消费者拒绝签收

以下示例演示在Java客户端配置生产者确认过程：

```
QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(ConfirmConfig.queueName, false, consumer);

QueueingConsumer.Delivery delivery = consumer.nextDelivery();
String msg = new String(delivery.getBody());
// do something with msg.
channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
```

预取值

使用场景

所谓消息预取机制，它定义了在一个信道上，消费者允许的最大未确认的消息数量。

一旦未确认的消息数量达到了设置的预取值，RabbitMQ就停止传递更多消息，除非至少有一条未完成的消息得到了确认。

如何设置合适的预取值

通常，增加预取将提高向消费者传递消息的速度。虽然自动应答传输消息速率是最佳的，但是，在这种情况下已传递但尚未处理的消息的数量也会增加，从而增加了消费者的 RAM 消耗(随机存取存储器)应该小心使用具有无限预处理的自动确认模式或手动确认模式，消费者消费了大量的消息如果没有确认的话，会导致消费者连接节点的内存消耗变大，所以找到合适的预取值是一个反复试验的过程，不同的负载该值取值也不同 100 到 300 范围内的值通常可提供最佳的吞吐量，并且不会给消费者带来太大的风险。

预取值为 1 是最保守的。当然这将使吞吐量变得很低，特别是消费者连接延迟很严重的情况下，特别是在消费者连接等待时间较长的环境 中。对于大多数应用来说，稍微高一点的值将是最佳的。

设置预取值

设置预取值的java示例代码如下

```
ConnectionFactory factory = new ConnectionFactory();
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
channel.basicQos(20, false);
```

心跳检测

使用场景

网络在很多情况下会失败，有时情况很微妙（比如 丢包率很高）。操作系统检测到 TCP 断开是一个适中的时间（在 Linux 中默认时长是 11 分钟）。AMQP 0-9-1 提供心跳检测功能来确保应用层及时发现中断的连接（或者是完全没有工作的连接）。 心跳检测还能保护连接不会在一段时间内没有活动而被终止。

心跳超时时间

心跳 timeout 值决定了 TCP 相互连接的最大时间，超过这个时间，该连接将被 RabbitMQ 和 客户端当作不可到达。这个值是在 RabbitMQ 服务器和客户端连接的时候协商的。客户端需要配置请求心跳检测。

心跳帧

心跳帧每隔 timeout/2 时间会发送一次。连续两次心跳失败后，连接将会当作不可到达。不同客户端对此的表现不同，但是 TCP 连接都会关闭。当客户端检测到 RabbitMQ 服务节点不可到达，它需要重新发起连接。

任何连接数据交换（例如 协议操作、发布消息、消息确认）都会计入有效的心跳。客户端可能也会发送心跳包，在连接中有其他数据交换，但有些只在需要时发送心跳包。

在客户端设置心跳超时时间

Java 客户端中设置心跳时间

```
ConnectionFactory cf = new ConnectionFactory();

cf.setRequestedHeartbeat(30);
```

单一活跃消费者

使用场景

在默认direct模式下, 多个消费者订阅一个queue,消息会轮流发送至每个消费者。单一消费者模式下,消息只由一个消费者消费，除非该改消费者出现异常。由此可以保证消息消费的有序性。

使用示例

以下为java客户端示例，通过在声明队列时配置x-single-active-consumer参数实现单一活跃消费者。

```
Map arguments = newHashMap();

arguments.put("x-single-active-consumer", true);

channel.queueDeclare("test_queue", false, false, false, arguments);
```

仲裁队列

使用场景

仲裁队列（Quorum Queue）是基于Raft一致性协议实现的一种新型的分布式消息队列，针对RabbitMQ的镜像模式设计。仲裁队列中的消息需要有集群中过半节点同意确认后，才会写入到队列中。

仲裁队列与镜像队列的差异

使用镜像队列，当生产者发送一条消息，主队列会将消息同步给镜像队列，所有的镜像队列都保存消息后，主队列才会向生产者发送确认。

仲裁队列的算法是基于Raft共识算法的一个实现。仲裁队列包含一个主副本和多个从副本，当生产者向主副本发送消息时，主副本会将消息同步给从副本，当超过半数的副本保存消息后，主副本才会向生产者发送确认。并且，主副本的选举也需要超过半数的副本同意，这会避免出现网络分区时队列元数据不一致。所以仲裁队列有更高的一致性。

配置方法

在声明队列时，将队列的“x-queue-type”参数设置为“quorum”。

以下为java示例

```
Map arguments = newHashMap<>();

arguments.put("x-queue-type", "quorum");

channel.queueDeclare("test_quorum_queue", true, false, false, arguments);
```

设置仲裁队列的长度

通过设置队列属性的方式可以限制仲裁队列的长度和在内存中保存的长度。

x-max-length：仲裁队列的最大消息数。如果超过则丢弃消息，或者发送到死信交换器。

x-max-length-bytes：仲裁队列的最大总消息大小（字节数）。如果超过则丢弃消息，或者发送到死信交换器。

x-max-in-memory-length：限制仲裁队列的内存中最大消息数量。

x-max-in-memory-bytes：限制仲裁队列的内存中的最大总消息大小（字节数）。

权限管理

主子账号

本章介绍如何使用主子账号体系管理子账号权限

概述

分布式消息服务RabbitMQ已接入主子账号体系，可区分两种账号权限，实现主账号对子账号的数据权限管理与功能权限管理，支持系统策略和自定义策略的授权方式。本章介绍如何使用主子账号体系管理子账号权限。

背景

1. 主账号：用户在天翼云注册后自动创建，该账号对其所拥有的资源具有完全的访问权限，可以重置用户密码、分配用户权限等。如果需要多人共同使用天翼云资源，由于账号是付费主体为了确保账号安全，建议创建子用户来进行日常管理工作。
2. 子账号：主账号认证为企业账号后，在天翼云用户中心页面创建出来的账号。子账号的用户名、密码统一由主账号创建管理。子账号同样可以登录访问天翼云控制台，登录入口与主账号相同，受主账号赋予的权限限制。
3. 企业项目：将云资源、企业成员按项目进行管理，通过企业项目将云资源、带有权限的用户组绑定到一起，用户使用项目内云资源的权限受用户组的授权限制。

注意

一个实例只能归属一个企业项目（可变更），一个子账号可以同时在多个企业项目中。

4. 策略： 是描述一组权限集的语言，它可以精确地描述被授权的资源集和操作集，通过策略，用户可以自由搭配需要授予的权限集。通过给用户组授予策略，用户组中的用户就能获得策略中定义的权限。
5. 系统策略： 系统预置的常用权限集，主要针对不同云服务的只读权限或管理员权限，比如对组件的只读权限、普通用户权限和管理员权限等等；系统策略只能用于授权，不能编辑和修改。
6. 数据权限： 看到的数据不一样。主账号看到所有实例，子账号只能看到所属项目中的实例。
7. 功能权限： 主账号可以进行所有控制台操作，子账号对单个组件实例拥有的操作权限由主账号授权。
8. 功能权限授权： 给子账号在企业项目A下增加一个策略，即代表该子账号对企业项目A下的实例拥有了策略中定义的权限，策略以外的操作会被禁止。

数据权限控制

数据权限的权限码为统一值instance-list，策略中包含instance-list的权限码才具备查看实例的权限。如果在用户组直接授权包含instance-list的策略，则该用户组的子用户能看到所有实例。如果在企业项目中的用户组授权包含instance-list的策略，则该用户组的子用户只能看到该企业项目下的实例。

操作步骤：

1. 进入IAM管理页面：

- 登录天翼云官网，鼠标悬浮至右上角个人信息，点击个人信息进入账号中心页面。
- 在账号中心页面中，点击统一认证服务进入IAM管理页面。

2. 创建用户组：

- 在IAM管理页面中，点击左侧菜单栏中用户组进入用户组管理页面。
- 在用户组管理页面中，点击创建用户组按钮，在弹出框中填写用户组名称与描述，点击确定即可。

3. 创建子用户：

- 在IAM管理页面中，点击左侧菜单栏中用户进入用户管理页面。
- 在用户管理页面中，点击创建用户按钮，进入创建页面。
- 在创建页面中，首先需要配置用户基本信息。填写用户名称、手机号、邮箱和密码等信息，点击下一步加入用户组。
- 在加入用户组页面，选择对应的用户组，点击添加按钮加入已选用户组，点击下一步即可。

4. 创建企业项目：

- 在IAM管理页面中，点击左侧菜单栏中企业项目进入企业项目管理页面。
- 在企业项目管理页面中，点击创建企业项目按钮，在弹出框中填写企业项目名称与描述，点击确定即可。

5. 在企业项目中添加用户组：

- 在企业项目管理页面中，点击企业项目的查看用户组按钮进入企业项目的用户组管理页面。
- 在企业项目的用户组管理页面中，点击设置用户组按钮，弹出设置用户组弹框。
- 在弹出框中，选择用户组再点击>按钮加入已选用户组，最后点击确定按钮即可。

6. 对企业项目中的用户组设置策略：

- 在企业项目管理页面中，点击企业项目的查看用户组按钮进入企业项目的用户组管理页面。
- 在企业项目的用户组管理页面中，点击用户组的设置策略按钮弹出设置策略弹框。
- 在弹出框中选择对应策略，点击>按钮加入已选策略，最后点击确定按钮即可。

功能权限控制

主账号对子账号的功能权限控制是通过给用户组授权策略实现的，策略包括系统策略和自定义策略。策略中可定义“允许”的操作和“拒绝”的操作，“拒绝”的优先级大于“允许”。

操作步骤：

1. 进入IAM管理页面：

- 登录天翼云官网，鼠标悬浮至右上角个人信息，点击“个人信息”进入账号中心页面。
- 在账号中心页面中，点击“统一认证服务”进入IAM管理页面。

2. 创建用户组：

- 在IAM管理页面中，点击左侧菜单栏中“用户组”进入用户组管理页面。
- 在用户组管理页面中，点击“创建用户组”按钮，在弹出框中填写用户组名称与描述，点击确定即可。

3. 创建子用户：

- 在IAM管理页面中，点击左侧菜单栏中“用户”进入用户管理页面。
- 在用户管理页面中，点击“创建用户”按钮，进入创建页面。
- 在创建页面中，首先需要配置用户基本信息。填写用户名称、手机号、邮箱和密码等信息，点击下一步加入用户组。
- 在加入用户组页面，选择对应的用户组，点击“添加”按钮加入已选用户组，点击下一步即可。

4. 授权：

- 在IAM管理页面中，点击左侧菜单栏中“用户组”进入用户组管理页面。
- 在用户组管理页面中，点击对应用户组的“授权”按钮，进入授权页面。
- 在授权页面中，勾选对应策略，点击下一步进入设置授权范围页面。
- 在设置授权范围页面中，选择授权范围，点击“确定”按钮就可完成授权。

云审计服务支持的关键操作

云审计服务支持的RabbitMQ操作列表

本页面主要介绍分布式消息服务RabbitMQ对接的云审计服务使用和查看方法。

操作场景

本服务现已对接天翼云 云审计服务 ，云审计服务提供对各种云资源操作的记录和查询功能，用于支撑合规审计、安全分析、操作追踪和问题定位等场景，同时提供事件跟踪功能，将操作日志转储至对象存储实现永久保存。

云审计可提供的功能服务具体如下：

● 记录审计日志：支持用户通过管理控制台或API接口发起的操作，以及各服务内部自触发的操作。

● 审计日志查询：支持在管理控制台对7天内操作记录按照事件类型、事件来源、资源类型、筛选类型、操作用户和事件级别等多个维度进行组合查询。

使用限制

● 云审计服务本身免费，包括时间记录以及7天内时间的存储和检索。

● 用户通过云审计能查询到多久前的操作事件：7天。

● 用户操作后多久可以通过云审计查询到数据：5分钟。

● 其它限制请参考 使用限制-云审计。

操作步骤

1. 开通云审计服务。

参见 开通云审计服务-云审计 。

2. 查看云审计事件。

参见 查看审计事件-云审计 。

3. 在事件列表中，选择事件来源为“容器与中间件”，资源类型选择“分布式消息服务Kafka”，上方时间选择需要筛选的时间段。点击查询即可。

4. 在审计事件右侧点击详情，可以看到更详细的事件信息。

更多云审计相关使用说明和常见问题请参考 用户指南、常见问题 。

关键操作列表

操作事件	读写类型
创建虚拟主机	写事件
删除虚拟主机	写事件
创建交换器	写事件
删除交换器	写事件
创建队列	写事件
删除队列	写事件
清空队列消息	写事件
队列生产拨测	写事件
队列消费拨测	写事件
新建用户	写事件
删除用户	写事件
创建绑定	写事件
删除绑定	写事件
配置虚拟主机主题权限	写事件
删除虚拟主机主题权限	写事件
配置虚拟主机限制	写事件
删除虚拟主机限制	写事件
配置虚拟主机权限	写事件
删除虚拟主机权限	写事件
插件启停	写事件
更新配置	写事件
实例开通	写事件
实例退订	写事件
实例冻结	写事件
实例注销	写事件
实例按需转包周期	写事件
实例包周期转按需	写事件
实例扩容	写事件
实例缩容	写事件

开发指南

概述

本指南主要介绍RabbitMQ实例连接信息的收集，如获取RabbitMQ实例连接地址与端口、访问实例的用户名和密码，然后提供Python语言和Spring Boot的连接示例。

RabbitMQ实例完全兼容开源RabbitMQ协议，Python以外的语言，请参考RabbitMQ官网提供的不同语言的连接和使用向导。

开源SDK列表

分布式消息服务RabbitMQ版支持所有开源版本的SDK，常见的开源SDK如下表所示。

开源SDK列表

编程语言	SDK
Java	rabbitmq-java-client
Spring Framework	SpringAMQP
.Net	rabbitmq-dotnet-client
Python	pika
PHP	php-amqplib
C	rabbitmq-c
Go	amqp091-go

客户端可以通过以下方式访问RabbitMQ实例：

1. VPC内子网地址访问

   如果客户端与RabbitMQ实例处于同region同VPC，则可以直接访问RabbitMQ实例提供的VPC内子网地址。
2. VPC对等连接方式访问

   如果客户端与RabbitMQ实例处于相同region但不同VPC，则可以通过建立VPC对等连接后，访问RabbitMQ实例提供的VPC内子网地址。

   关于创建和使用VPC对等连接，可参考 对等连接 。
3. 公网访问

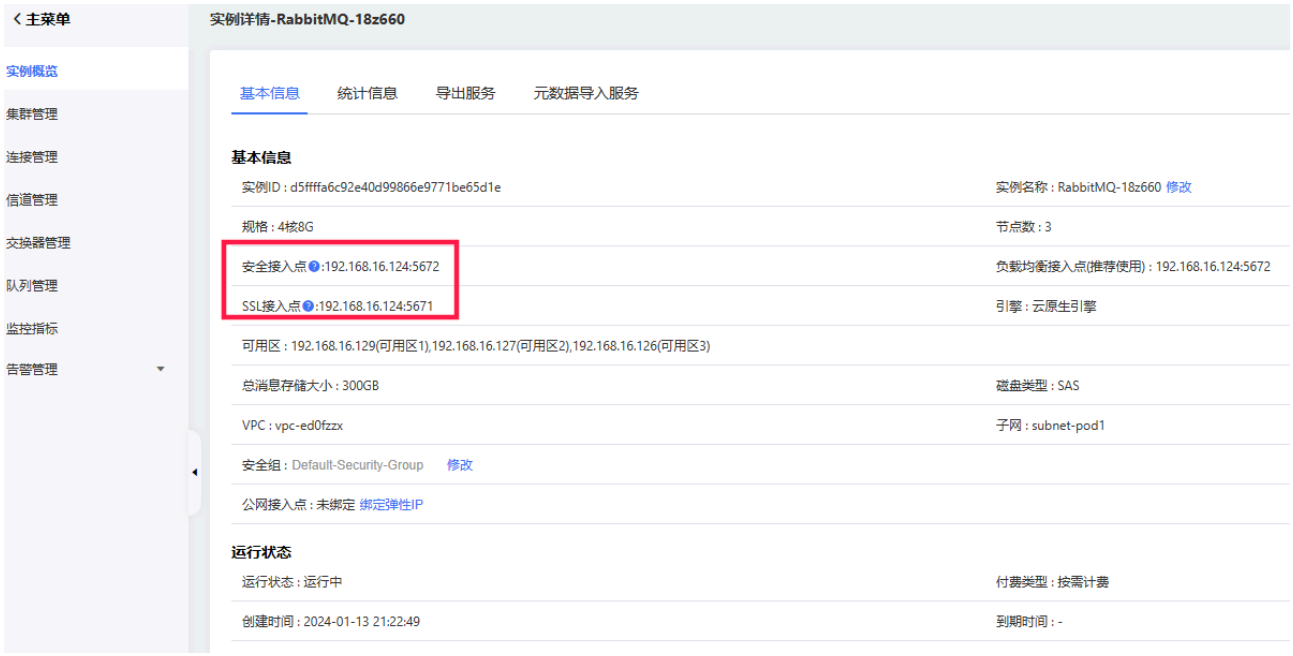
   客户端在其他网络环境，或者与RabbitMQ实例处于不同region，则访问实例的公网地址。

收集连接信息

实例连接地址与端口

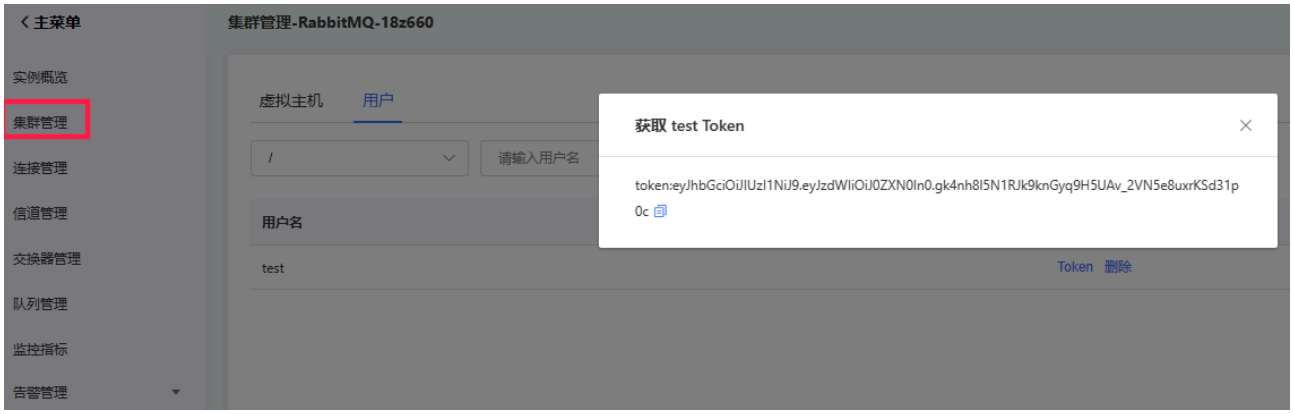
实例创建后，从实例的“基本信息”页签的“连接信息”中获取。

查看RabbitMQ实例连接地址与端口



访问实例的用户名和token

实例创建后，从实例的“集群管理”页签的“用户”中获取用户名及token。如果没有则新建。



Java

使用Maven方式引入依赖

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.7.0</version>
</dependency>
```

生产消息

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
public class RabbitmqProducer {
    //  private  final  static  String  EXCHANGE_NAME  =  "exchangeTest";
    private final static String QUEUE_NAME = "helloMQ";
    //  private  final  static  String  ROUTING_KEY  =  "test";
    public static void main(String[] args) throws IOException, TimeoutException, Interrup
tedException {
        //  ######
        ConnectionFactory factory = new ConnectionFactory();
        //  ####ip
        factory.setHost("127.0.0.1");
        //  ##amqp#tcp###
        factory.setPort(5672);
        //  #######
        factory.setUsername("YOUR USERNAME");
        factory.setPassword("YOUR PASSWORD");
        //  ##Vhost##########
        factory.setVirtualHost("/");
        //  ###############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
        //  ######
        Connection connection = factory.newConnection();
        //  ######
        Channel channel = connection.createChannel();
        //  ######
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        for (int i = 0; i < 100; i++) {
            //  #####
            String message = "Hello  rabbitMQ!_" + i;
            //  ###################
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes(
StandardCharsets.UTF_8));
            System.out.println(" Sent message: '" + message + "'");
            TimeUnit.MILLISECONDS.sleep(100);
        }
        //#######
        channel.close();
        connection.close();
    }
```

```
}
```

消费消息

```
import com.rabbitmq.client.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;
public class RabbitmqConsumer {
    //####
    private final static String QUEUE_NAME = "helloMQ";
    public static void main(String[] args) throws IOException, TimeoutException {
        //######
        ConnectionFactory factory = new ConnectionFactory();
        //####ip
        factory.setHost("127.0.0.1");
        //##amqp#tcp###
        factory.setPort(5672);
        //#######
        factory.setUsername("YOUR USERNAME");
        factory.setPassword("YOUR PASSWORD");
        //##Vhost
        factory.setVirtualHost("/");
        //###############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        //####################################
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        System.out.println("  [*]  Waiting  for  messages.  To  exit  press  CTRL
+C");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(
String consumerTag, Envelope envelope, AMQP.BasicPro
perties properties, byte[] body) throws IOException {
                String message = new String(body, StandardCharsets.UTF_8);
                System.out.println("Received message: '" + message + "'");
            }
        };
        channel.basicConsume(QUEUE_NAME, true, consumer);
    }
}
```

SSL生产消息

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DefaultSaslConfig;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManagerFactory;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.security.cert.CertificateException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
```

```
public class RabbitmqProducerSsl {
    //  private  final  static  String  EXCHANGE_NAME  =  "exchangeTest";
    private final static String QUEUE_NAME = "helloMQ";
    //  private  final  static  String  ROUTING_KEY  =  "test";
    public static void main(String[] args) throws IOException, TimeoutException, Interrup
tedException {
        //  ######
        ConnectionFactory factory = new ConnectionFactory();
        //  ####ip
        factory.setHost("127.0.0.1");
        //  ##amqp#ssl###
        factory.setPort(5671);
        String ksFile = "/sslpath/ssl/client_rabbitmq_key.p12";
        String tksFile = "/sslpath/ssl/truststore";
        SSLContext c = null;
        try {
            char[] keyPassphrase = "YOUR PASSPHRASE".toCharArray();
            KeyStore ks = KeyStore.getInstance("PKCS12");
            ks.load(new FileInputStream(ksFile), keyPassphrase);
            KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
            kmf.init(ks, keyPassphrase);
            char[] trustPassphrase = "YOUR PASSPHRASE".toCharArray();
            KeyStore tks = KeyStore.getInstance("JKS");
            tks.load(new FileInputStream(tksFile), trustPassphrase);
            TrustManagerFactory tmf = TrustManagerFactory.getInsta
nce("SunX509");
            tmf.init(tks);
            c = SSLContext.getInstance("tlsv1.2");
            c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
        } catch (KeyStoreException e) {
            throw new RuntimeException(e);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        } catch (CertificateException e) {
            throw new RuntimeException(e);
        } catch (UnrecoverableKeyException e) {
            throw new RuntimeException(e);
        } catch (KeyManagementException e) {
            throw new RuntimeException(e);
        }
        factory.setSaslConfig(DefaultSaslConfig.EXTERNAL);
        factory.useSslProtocol(c);
        //  ##Vhost##########
        factory.setVirtualHost("/");
        //  ##############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
        //  ######
        Connection connection = factory.newConnection();
        //  ######
        Channel channel = connection.createChannel();
        //  ######
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        for (int i = 0; i < 100; i++) {
            //  #####
            String message = "Hello  rabbitMQ!_" + i;
            //  ##################
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes(
StandardCharsets.UTF_8));
            System.out.println(" Sent message: '" + message + "'");
            TimeUnit.MILLISECONDS.sleep(100);
        }
        //#######
        channel.close();
        connection.close();
    }
}
```

SSL消费消息

```
import com.rabbitmq.client.*;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManagerFactory;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.security.cert.CertificateException;
import java.util.concurrent.TimeoutException;
public class RabbitmqConsumerSsl {
    //####
    private final static String QUEUE_NAME = "helloMQ";
    public static void main(String[] args) throws IOException, TimeoutException {
        //######
        ConnectionFactory factory = new ConnectionFactory();
        //####ip
        factory.setHost("127.0.0.1");
        //  ##amqp#ssl###
        factory.setPort(5671);
        String ksFile = "/sslpath/ssl/client_rabbitmq_key.p12";
        String tksFile = "/sslpath/ssl/truststore";
        SSLContext c = null;
        try {
            char[] keyPassphrase = "YOUR PASSPHRASE".toCharArray();
            KeyStore ks = KeyStore.getInstance("PKCS12");
            ks.load(new FileInputStream(ksFile), keyPassphrase);
            KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
            kmf.init(ks, keyPassphrase);
            char[] trustPassphrase = "YOUR PASSPHRASE".toCharArray();
            KeyStore tks = KeyStore.getInstance("JKS");
            tks.load(new FileInputStream(tksFile), trustPassphrase);
            TrustManagerFactory tmf = TrustManagerFactory.getInsta
nce("SunX509");
            tmf.init(tks);
            c = SSLContext.getInstance("tlsv1.2");
            c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
        } catch (KeyStoreException e) {
            throw new RuntimeException(e);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        } catch (CertificateException e) {
            throw new RuntimeException(e);
        } catch (UnrecoverableKeyException e) {
            throw new RuntimeException(e);
        } catch (KeyManagementException e) {
            throw new RuntimeException(e);
        }
        factory.setSaslConfig(DefaultSaslConfig.EXTERNAL);
        factory.useSslProtocol(c);
        //##Vhost##########
        factory.setVirtualHost("vhost");
        //###############
        factory.setConnectionTimeout(30 * 1000);
        factory.setHandshakeTimeout(30 * 1000);
        factory.setShutdownTimeout(0);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        //###################################
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        System.out.println("  [*]  Waiting  for  messages.  To  exit  press  CTRL
+C");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
```

```
            public void handleDelivery(
String consumerTag, Envelope envelope, AMQP.BasicPro
perties properties, byte[] body) throws IOException {
                String message = new String(body, StandardCharsets.UTF_8);
                System.out.println("Received message: '" + message + "'");
            }
        };
        channel.basicConsume(QUEUE_NAME, true, consumer);
    }
}
```

Python

本文介绍Python版本的RabbitMQ客户端连接指导，包括RabbitMQ客户端安装，以及生产、消费消息。

使用前请参考 收集连接信息 收集RabbitMQ所需的连接信息。

准备环境

1. 安装Python3
2. 安装kombu

```
pip3 install kombu
```

生产消息

SSL认证方式

```
import ssl
import sys
from kombu import Connection, Exchange, Producer
def Main(argv):
    ca_certfile = "certs\\ca_certificate.pem"
    certfile = "certs\\client_rabbitmq_certificate.pem"
    private_key = "certs\\client_rabbitmq_key.pem"
    conn = Connection('amqp://XXX:xxx/', login_method='EXTERNAL', ssl={
        'ca_certs': ca_certfile,
        'keyfile': private_key,
        'certfile': certfile,
        'cert_reqs': ssl.CERT_REQUIRED,
        'ssl_version': ssl.PROTOCOL_TLSv1_2,
    })
    channel = conn.channel()
    exchange = Exchange("example-exchange", type="direct")
    producer = Producer(exchange=exchange, channel=channel, routing_key="test-
key")
    message = "Hello Rabbimtq"
    producer.publish(message, retry=True)
    print('send message: %s' % message)
if __name__ == '__main__':
    Main(sys.argv)
```

非SSL认证方式

```
import sys
from kombu import Connection, Exchange, Producer
def Main(argv):
    rabbitmq_url = 'amqp://USERNAME:PASSWORD@XXX:xxx/'
    conn = Connection(rabbitmq_url, login_method='PLAIN')
    channel = conn.channel()
    exchange = Exchange("example-exchange", type="direct")
    producer = Producer(exchange=exchange, channel=channel, routing_key="test-
key")
    message = "Hello Rabbimtq"
    producer.publish(message, retry=True)
    print('send message: %s' % message)
if __name__ == '__main__':
    Main(sys.argv)
```

消费消息

SSL认证方式

```
import sys
import ssl
from kombu import Connection, Exchange, Queue, Consumer
def Main(argv):
    ca_certfile = "certs\\ca_certificate.pem"
    certfile ="certs\\client_rabbitmq_certificate.pem"
    private_key = "certs\\client_rabbitmq_key.pem"
    conn = Connection('amqp://XXX:xxx/', login_method='EXTERNAL', ssl={
        'ca_certs': ca_certfile,
        'keyfile': private_key,
        'certfile': certfile,
        'cert_reqs': ssl.CERT_REQUIRED,
        'ssl_version': ssl.PROTOCOL_TLSv1_2,
    })
    exchange = Exchange("example-exchange", type="direct")
    queue = Queue(name="example-queue", exchange=exchange, routing_key="test-
key")
    def process_message(body, message):
        print("receive message: %s" % format(body))
        message.ack()
    with Consumer(conn, queues=queue, callbacks=[process_message], accept=["text/
plain"]):
        conn.drain_events(timeout=2)
if __name__ == '__main__':
    Main(sys.argv)
```

非SSL认证方式

```
import sys
from kombu import Connection, Exchange, Queue, Consumer
def Main(argv):
    rabbitmq_url = 'amqp://USERNAME:PASSWORD@XXX:xxx/'
    conn = Connection(rabbitmq_url, login_method='PLAIN')
    exchange = Exchange("example-exchange", type="direct")
    queue = Queue(name="example-queue", exchange=exchange, routing_key="test-
key")
```

```
    def process_message(body, message):
        print("receive message: %s" % format(body))
        message.ack()
    with Consumer(conn, queues=queue, callbacks=[process_message], accept=["text/
plain"]):
        conn.drain_events(timeout=2)
if __name__ == '__main__':
    Main(sys.argv)
```

.net

编译工程生产消费

引入依赖

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <LangVersion>latest</LangVersion>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
    <WarningsAsErrors />
    <AnalysisMode>Default</AnalysisMode>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="RabbitMQ.Client" Version="6.2.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\..\Common\Rabbit.Common.csproj" />
  </ItemGroup>
</Project>```
```

生产消息

```
using System.Collections.Immutable;
using System.Drawing;
using System.Threading.Tasks;
using Rabbit.Common.Data.Trades;
using Rabbit.Common.Display;
using RabbitMQ.Client;

namespace Rabbit.Example4.Producer
{
    internal sealed class Program
    {
        private static async Task Main()
        {
            var connectionFactory = new ConnectionFactory
            {
                HostName = "YOUR HOST IP",
                UserName = "YOUR USERNAME",
                Password = "YOUR PASSWORD",
                Port = 5672
            };
```

```
            using var connection = connectionFactory.CreateConnection();

            using var channel = connection.CreateModel();

            const string ExchangeName = "dotnet_exchange";
            const string QueueName = "dotnet_queue";

            channel.ExchangeDeclare(
                exchange: ExchangeName,
                type: ExchangeType.Direct,
                durable: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            var queue = channel.QueueDeclare(
                queue: QueueName,
                durable: false,
                exclusive: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            channel.QueueBind(
                queue: queue.QueueName,
                exchange: ExchangeName,
                routingKey: QueueName,
                arguments: ImmutableDictionary<string, object>.Empty);


            while (true)
            {
                var trade = TradeData.GetFakeTrade();

                string routingKey = QueueName;

                channel.BasicPublish(
                    exchange: ExchangeName,
                    routingKey: routingKey,
                    body: trade.ToBytes()
                );

                DisplayInfo<Trade>
                    .For(trade)
                    .SetExchange(ExchangeName)
                    .SetRoutingKey(routingKey)
                    .SetVirtualHost(connectionFactory.VirtualHost)
                    .Display(Color.Cyan);

                await Task.Delay(millisecondsDelay: 3000);
            }
        }
    }
}
```

消费消息

```
using System;
using System.Collections.Immutable;
using System.Drawing;
using Rabbit.Common.Data.Trades;
using Rabbit.Common.Display;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

namespace Rabbit.Example4.Consumer
```

```
{
    internal sealed class Program
    {
        private static void Main()
        {
            var connectionFactory = new ConnectionFactory
            {
                HostName = "YOUR HOST IP",
                UserName = "YOUR USER",
                Password = "YOUR PASSWORD",
                Port = 5672
            };

            using var connection = connectionFactory.CreateConnection();
            using var channel = connection.CreateModel();

            const string ExchangeName = "dotnet_exchange";
            const string QueueName = "dotnet_queue";

            channel.ExchangeDeclare(
                exchange: ExchangeName,
                type: ExchangeType.Direct,
                durable: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            var queue = channel.QueueDeclare(
                queue: QueueName,
                durable: false,
                exclusive: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            channel.QueueBind(
                queue: queue.QueueName,
                exchange: ExchangeName,
                routingKey: QueueName);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (sender, eventArgs) =>
            {
                var messageBody = eventArgs.Body.ToArray();
                var trade = Trade.FromBytes(messageBody);

                DisplayInfo<Trade>
                    .For(trade)
                    .SetExchange(eventArgs.Exchange)
                    .SetQueue(queue.QueueName)
                    .SetRoutingKey(eventArgs.RoutingKey)
                    .SetVirtualHost(connectionFactory.VirtualHost)
                    .Display(Color.Yellow);

                channel.BasicAck(eventArgs.DeliveryTag, multiple: false);
            };

            channel.BasicConsume(
                queue: queue.QueueName,
                autoAck: false,
                consumer: consumer);

            Console.ReadLine();
        }
    }
}
```

ssl生产消息

```
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Drawing;
using System.Threading.Tasks;
using Rabbit.Common.Data.Trades;
using Rabbit.Common.Display;
using RabbitMQ.Client;

namespace Rabbit.dotnet.Producer
{
    internal sealed class Program
    {
        private static async Task Main()
        {
            var connectionFactory = new ConnectionFactory
            {
                HostName = "YOUR HOST IP",
                UserName = "YOUR USERNAME",
                Password = "YOUR PASSWORD",
                Port = 5671
            };
            connectionFactory.AuthMechanisms = new List<IAuthMec
hanismFactory>() { new ExternalMechanismFactory() };
            connectionFactory.Ssl.CertPath = @"D:\tmp\hzmq-test-0520_rabbitmq_
ssl_client\client_rabbitmq_key.p12";
            // Ssl.CertPassphrase##########
            connectionFactory.Ssl.CertPassphrase = "YOUR PASSPHRASE";
            connectionFactory.Ssl.Enabled = true;
            connectionFactory.Ssl.CertificateValidationCallback = (_, _, _, _)
 => { return true; };

            using var connection = connectionFactory.CreateConnection();

            using var channel = connection.CreateModel();

            const string ExchangeName = "dotnet_exchange";
            const string QueueName = "dotnet_queue";

            channel.ExchangeDeclare(
                exchange: ExchangeName,
                type: ExchangeType.Direct,
                durable: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            var queue = channel.QueueDeclare(
                queue: QueueName,
                durable: false,
                exclusive: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            channel.QueueBind(
                queue: queue.QueueName,
                exchange: ExchangeName,
                routingKey: QueueName,
                arguments: ImmutableDictionary<string, object>.Empty);

            while (true)
            {
                var trade = TradeData.GetFakeTrade();

                string routingKey = QueueName;
```

```
                channel.BasicPublish(
                    exchange: ExchangeName,
                    routingKey: routingKey,
                    body: trade.ToBytes()
                );

                DisplayInfo<Trade>
                    .For(trade)
                    .SetExchange(ExchangeName)
                    .SetRoutingKey(routingKey)
                    .SetVirtualHost(connectionFactory.VirtualHost)
                    .Display(Color.Cyan);

                await Task.Delay(millisecondsDelay: 1000);
            }
        }
    }
}
```

ssl消费消息

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Drawing;
using Rabbit.Common.Data.Trades;
using Rabbit.Common.Display;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

namespace Rabbit.example5.Consumer
{
    internal sealed class Program
    {
        private static void Main()
        {
            var connectionFactory = new ConnectionFactory
            {
                HostName = "YOUR HOST IP",
                UserName = "YOUR USERNAME",
                Password = "YOUR PASSWORD",
                Port = 5671
            };
            connectionFactory.AuthMechanisms = new List<IAuthMec
hanismFactory>() { new ExternalMechanismFactory() };
            connectionFactory.Ssl.CertPath = @"D:\tmp\hzmq-test-0520_rabbitmq_
ssl_client\client_rabbitmq_key.p12";
            // Ssl.CertPassphrase##########
            connectionFactory.Ssl.CertPassphrase = "YOUR PASSPHRASE";
            connectionFactory.Ssl.Enabled = true;
            connectionFactory.Ssl.CertificateValidationCallback = (_, _, _, _)
 => { return true; };

            using var connection = connectionFactory.CreateConnection();

            using var channel = connection.CreateModel();

            const string ExchangeName = "dotnet_exchange";
            const string QueueName = "dotnet_queue";

            channel.ExchangeDeclare(
                exchange: ExchangeName,
                type: ExchangeType.Direct,
                durable: false,
                autoDelete: false,
```

```
                arguments: ImmutableDictionary<string, object>.Empty);

            var queue = channel.QueueDeclare(
                queue: QueueName,
                durable: false,
                exclusive: false,
                autoDelete: false,
                arguments: ImmutableDictionary<string, object>.Empty);

            channel.QueueBind(
                queue: queue.QueueName,
                exchange: ExchangeName,
                routingKey: QueueName);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (sender, eventArgs) =>
            {
                var messageBody = eventArgs.Body.ToArray();
                var trade = Trade.FromBytes(messageBody);

                DisplayInfo<Trade>
                    .For(trade)
                    .SetExchange(eventArgs.Exchange)
                    .SetQueue(queue.QueueName)
                    .SetRoutingKey(eventArgs.RoutingKey)
                    .SetVirtualHost(connectionFactory.VirtualHost)
                    .Display(Color.Yellow);

                channel.BasicAck(eventArgs.DeliveryTag, multiple: false);
            };

            channel.BasicConsume(
                queue: queue.QueueName,
                autoAck: false,
                consumer: consumer);

            Console.ReadLine();
        }
    }
}
```

PHP

编译工程生产消费

引入依赖

composer.json

```
{
  "require": {
    "php-amqplib/php-amqplib":"v3.6.2"
  }
}
```

生产消息

```
<?php

require __DIR__."/../vendor/autoload.php";

use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Exchange\AMQPExchangeType;
use PhpAmqpLib\Message\AMQPMessage;

$exchange = 'php-exchange';
$queue = 'php-queue';


$connection = new AMQPStreamConnection("IP", 5672, "YOUR USERNAME", "YOUR
 PASSWORD", "/");
$channel = $connection->channel();


$channel->queue_declare($queue, false, true, false, false);

$channel->exchange_declare($exchange, AMQPExchangeType::DIRECT, false, true,
 false);

$channel->queue_bind($queue, $exchange);

$messageBody = implode(' ', array_slice($argv, 1));
$message = new AMQPMessage($messageBody, array('content_type' => 'text/plain',
 'delivery_mode' => AMQPMessage::DELIVERY_MODE_PERSISTENT));
$channel->basic_publish($message, $exchange);

$channel->close();
$connection->close();
```

消费消息

```
<?php

require __DIR__."/../vendor/autoload.php";
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Exchange\AMQPExchangeType;

$exchange = 'php-exchange';
$queue = 'php-queue';
$consumerTag = 'php-consumer-tag';

$connection = new AMQPStreamConnection("33.0.1.35", 5672, "user", "password",
 "/");
$channel = $connection->channel();

$channel->queue_declare($queue, false, true, false, false);

$channel->exchange_declare($exchange, AMQPExchangeType::DIRECT, false, true,
 false);

$channel->queue_bind($queue, $exchange);

function process_message($message)
{
    echo "\n--------\n";
    echo $message->body;
    echo "\n--------\n";
```

```
    $message->ack();

    // Send a message with the string "quit" to cancel the consumer.
    if ($message->body === 'quit') {
        $message->getChannel()->basic_cancel($message->getConsumerTag());
    }
}

$channel->basic_consume($queue, $consumerTag, false, false, false, false,
 'process_message');

/**
 * @param \PhpAmqpLib\Channel\AMQPChannel $channel
 * @param \PhpAmqpLib\Connection\AbstractConnection $connection
 */
function shutdown($channel, $connection)
{
    $channel->close();
    $connection->close();
}

register_shutdown_function('shutdown', $channel, $connection);

$channel->consume();
```

ssl生产消息

```
<?php

require __DIR__."/../vendor/autoload.php";

use PhpAmqpLib\Connection\AMQPSSLConnection;
use PhpAmqpLib\Exchange\AMQPExchangeType;
use PhpAmqpLib\Message\AMQPMessage;

define('CERTS_PATH', realpath(__DIR__ . '/certs'));

$exchange = 'php-exchange';
$queue = 'php-queue';

$sslOptions = array(
    'cafile' => CERTS_PATH . '/ca_certificate.pem',
    'local_cert' => CERTS_PATH . '/client_rabbitmq_certificate.pem',
    'local_pk' => CERTS_PATH . '/client_rabbitmq_key.pem',
    'verify_peer' => true,
    'verify_peer_name' => false,
);

$connection = new AMQPSSLConnection("10.10.33.196", 5671, "YOUR USERNAME",
 "YOUR PASSWORD", "/" , $sslOptions);

$channel = $connection->channel();

$channel->queue_declare($queue, false, true, false, false);

$channel->exchange_declare($exchange, AMQPExchangeType::DIRECT, false, true,
 false);

$channel->queue_bind($queue, $exchange);

$channel->queue_declare($queue, false, true, false, false);

$channel->exchange_declare($exchange, AMQPExchangeType::DIRECT, false, true,
 false);
```

```
$channel->queue_bind($queue, $exchange);

$messageBody = implode(' ', array_slice($argv, 1));
$message = new AMQPMessage($messageBody, array('content_type' => 'text/plain',
 'delivery_mode' => AMQPMessage::DELIVERY_MODE_PERSISTENT));
$channel->basic_publish($message, $exchange);

$channel->close();
$connection->close();
```

ssl消费消息

```
<?php

require __DIR__."/../vendor/autoload.php";

use PhpAmqpLib\Connection\AMQPSSLConnection;

use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Exchange\AMQPExchangeType;
use PhpAmqpLib\Message\AMQPMessage;

define('CERTS_PATH', realpath(__DIR__ . '/certs'));

$exchange = 'php-exchange';
$queue = 'php-queue';
$consumerTag = 'php-consumer-tag';

$sslOptions = array(
    'cafile' => CERTS_PATH . '/ca_certificate.pem',
    'local_cert' => CERTS_PATH . '/client_rabbitmq_certificate.pem',
    'local_pk' => CERTS_PATH .   '/client_rabbitmq_key.pem',
    'verify_peer' => false,
    'verify_peer_name' => false,
);

$connection = new AMQPSSLConnection("10.10.33.196", 5671, "YOUR USERNAME",
 "YOUR PASSWORD", "/" , $sslOptions);
$channel = $connection->channel();

$channel->queue_declare($queue, false, true, false, false);

$channel->exchange_declare($exchange, AMQPExchangeType::DIRECT, false, true,
 false);

$channel->queue_bind($queue, $exchange);

function process_message($message)
{
    echo "\n--------\n";
    echo $message->body;
    echo "\n--------\n";

    $message->ack();

    // Send a message with the string "quit" to cancel the consumer.
    if ($message->body === 'quit') {
        $message->getChannel()->basic_cancel($message->getConsumerTag());
    }
}

$channel->basic_consume($queue, $consumerTag, false, false, false, false,
 'process_message');
```

```
function shutdown($channel, $connection)
{
    $channel->close();
    $connection->close();
}

register_shutdown_function('shutdown', $channel, $connection);

$channel->consume();
```

Go

编译工程生产消费

引入依赖

go.mod

```
module test_go

require (
github.com/rabbitmq/amqp091-go v1.10.0
golang.org/x/net v0.26.0
)

go 1.20
```

生产消息

```
package main

import (
"flag"
"fmt"
amqp "github.com/rabbitmq/amqp091-go"
"log"
)

var (
uri          = flag.String("uri", "amqp://USERNAME:PASSWORD@33.0.1.35:5672",
 "AMQP URI")
exchangeName = flag.String("exchange", "test-exchange", "Durable AMQP exchange
 name")
exchangeType = flag.String("exchange-type", "direct", "Exchange type - direct|
fanout|topic|x-custom")
routingKey   = flag.String("key", "test-key", "AMQP routing key")
body         = flag.String("body", "foobar", "Body of message")
reliable     = flag.Bool("reliable", true, "Wait for the publisher
 confirmation before exiting")
)

func init() {
flag.Parse()
}

func main() {
```

```
if err := publish(*uri, *exchangeName, *exchangeType, *routingKey, *body,
 *reliable); err != nil {
log.Fatalf("%s", err)
}
log.Printf("published %dB OK", len(*body))
}

func publish(amqpURI, exchange, exchangeType, routingKey, body string,
 reliable bool) error {

// This function dials, connects, declares, publishes, and tears down,
// all in one go. In a real service, you probably want to maintain a
// long-lived connection as state, and publish against that.

log.Printf("dialing %q", amqpURI)
connection, err := amqp.Dial(amqpURI)
if err != nil {
return fmt.Errorf("Dial: %s", err)
}
defer connection.Close()

log.Printf("got Connection, getting Channel")
channel, err := connection.Channel()
if err != nil {
return fmt.Errorf("Channel: %s", err)
}

log.Printf("got Channel, declaring %q Exchange (%q)", exchangeType, exchange)
if err := channel.ExchangeDeclare(
exchange,     // name
exchangeType, // type
true,         // durable
false,        // auto-deleted
false,        // internal
false,        // noWait
nil,          // arguments
); err != nil {
return fmt.Errorf("Exchange Declare: %s", err)
}

// Reliable publisher confirms require confirm.select support from the
// connection.
if reliable {
log.Printf("enabling publishing confirms.")
if err := channel.Confirm(false); err != nil {
return fmt.Errorf("Channel could not be put into confirm mode: %s", err)
}

confirms := channel.NotifyPublish(make(chan amqp.Confirmation, 1))

defer confirmOne(confirms)
}

log.Printf("declared Exchange, publishing %dB body (%q)", len(body), body)
if err = channel.Publish(
exchange,   // publish to an exchange
routingKey, // routing to 0 or more queues
false,      // mandatory
false,      // immediate
amqp.Publishing{
Headers:         amqp.Table{},
ContentType:     "text/plain",
ContentEncoding: "",
Body:            []byte(body),
DeliveryMode:    amqp.Transient, // 1=non-persistent, 2=persistent
Priority:        0,              // 0-9
// a bunch of application/implementation-specific fields
},
```

```
); err != nil {
return fmt.Errorf("Exchange Publish: %s", err)
}

return nil
}

// One would typically keep a channel of publishings, a sequence number, and a
// set of unacknowledged sequence numbers and loop until the publishing
 channel
// is closed.
func confirmOne(confirms <-chan amqp.Confirmation) {
log.Printf("waiting for confirmation of one publishing")

if confirmed := <-confirms; confirmed.Ack {
log.Printf("confirmed delivery with delivery tag: %d", confirmed.DeliveryTag)
} else {
log.Printf("failed delivery of delivery tag: %d", confirmed.DeliveryTag)
}
}
```

消费消息

```
package main

import (
"flag"
"fmt"
amqp "github.com/rabbitmq/amqp091-go"
"log"
"time"
)

var (
uri          = flag.String("uri", "amqp://USERNAME:PASSWORD@10.10.33.196:
5672", "AMQP URI")
exchange     = flag.String("exchange", "test-exchange", "Durable, non-auto-
deleted AMQP exchange name")
exchangeType = flag.String("exchange-type", "direct", "Exchange type - direct|
fanout|topic|x-custom")
queue        = flag.String("queue", "test-queue", "Ephemeral AMQP queue name")
bindingKey   = flag.String("key", "test-key", "AMQP binding key")
consumerTag  = flag.String("consumer-tag", "simple-consumer", "AMQP consumer
 tag (should not be blank)")
lifetime     = flag.Duration("lifetime", 5*time.Second, "lifetime of process
 before shutdown (0s=infinite)")
)

func init() {
flag.Parse()
}

func main() {
c, err := NewConsumer(*uri, *exchange, *exchangeType, *queue, *bindingKey,
 *consumerTag)
if err != nil {
log.Fatalf("%s", err)
}

if *lifetime > 0 {
log.Printf("running for %s", *lifetime)
time.Sleep(*lifetime)
} else {
log.Printf("running forever")
select {}
}

log.Printf("shutting down")
```

```
if err := c.Shutdown(); err != nil {
log.Fatalf("error during shutdown: %s", err)
}
}

type Consumer struct {
conn    *amqp.Connection
channel *amqp.Channel
tag     string
done    chan error
}

func NewConsumer(amqpURI, exchange, exchangeType, queueName, key, ctag string)
 (*Consumer, error) {
c := &Consumer{
conn:    nil,
channel: nil,
tag:     ctag,
done:    make(chan error),
}

var err error

log.Printf("dialing %q", amqpURI)
c.conn, err = amqp.Dial(amqpURI)
if err != nil {
return nil, fmt.Errorf("Dial: %s", err)
}

go func() {
fmt.Printf("closing: %s", <-c.conn.NotifyClose(make(chan *amqp.Error)))
}()

log.Printf("got Connection, getting Channel")
c.channel, err = c.conn.Channel()
if err != nil {
return nil, fmt.Errorf("Channel: %s", err)
}

log.Printf("got Channel, declaring Exchange (%q)", exchange)
if err = c.channel.ExchangeDeclare(
exchange,     // name of the exchange
exchangeType, // type
true,         // durable
false,        // delete when complete
false,        // internal
false,        // noWait
nil,          // arguments
); err != nil {
return nil, fmt.Errorf("Exchange Declare: %s", err)
}

log.Printf("declared Exchange, declaring Queue %q", queueName)
queue, err := c.channel.QueueDeclare(
queueName, // name of the queue
true,      // durable
false,     // delete when unused
false,     // exclusive
false,     // noWait
nil,       // arguments
)
if err != nil {
return nil, fmt.Errorf("Queue Declare: %s", err)
}

log.Printf("declared Queue (%q %d messages, %d consumers), binding to Exchange
 (key %q)",
```

```
queue.Name, queue.Messages, queue.Consumers, key)

if err = c.channel.QueueBind(
queue.Name, // name of the queue
key,        // bindingKey
exchange,   // sourceExchange
false,      // noWait
nil,        // arguments
); err != nil {
return nil, fmt.Errorf("Queue Bind: %s", err)
}

log.Printf("Queue bound to Exchange, starting Consume (consumer tag %q)",
 c.tag)
deliveries, err := c.channel.Consume(
queue.Name, // name
c.tag,      // consumerTag,
false,      // noAck
false,      // exclusive
false,      // noLocal
false,      // noWait
nil,        // arguments
)
if err != nil {
return nil, fmt.Errorf("Queue Consume: %s", err)
}

go handle(deliveries, c.done)

return c, nil
}

func (c *Consumer) Shutdown() error {
// will close() the deliveries channel
if err := c.channel.Cancel(c.tag, true); err != nil {
return fmt.Errorf("Consumer cancel failed: %s", err)
}

if err := c.conn.Close(); err != nil {
return fmt.Errorf("AMQP connection close error: %s", err)
}

defer log.Printf("AMQP shutdown OK")

// wait for handle() to exit
return <-c.done
}

func handle(deliveries <-chan amqp.Delivery, done chan error) {
for d := range deliveries {
log.Printf(
"got %dB delivery: [%v] %q",
len(d.Body),
d.DeliveryTag,
d.Body,
)
d.Ack(false)
}
log.Printf("handle: deliveries channel closed")
done <- nil
}
```

ssl生产消息

```
package main
```

```
import (
"crypto/tls"
"crypto/x509"
"flag"
"fmt"
amqp "github.com/rabbitmq/amqp091-go"
"io/ioutil"
"log"
)

var (
uri          = flag.String("uri", "amqps://USERNAME:PASSWORD@10.10.33.196:
5671", "AMQP URI")
exchangeName = flag.String("exchange", "go-exchange", "Durable AMQP exchange
 name")
exchangeType = flag.String("exchange-type", "direct", "Exchange type - direct|
fanout|topic|x-custom")
routingKey   = flag.String("key", "test-key", "AMQP routing key")
body         = flag.String("body", "foobar", "Body of message")
reliable     = flag.Bool("reliable", true, "Wait for the publisher
 confirmation before exiting")
)

func init() {
flag.Parse()
}

func main() {
if err := publish(*uri, *exchangeName, *exchangeType, *routingKey, *body,
 *reliable); err != nil {
log.Fatalf("%s", err)
}
log.Printf("published %dB OK", len(*body))
}

func publish(amqpsURI, exchange, exchangeType, routingKey, body string,
 reliable bool) error {
caCert, err := ioutil.ReadFile("D:\\tmp\\hzmq-test-0520_rabbitmq_ssl_client\
\ca_certificate.pem")
if err != nil {
return err
}

cert, err := tls.LoadX509KeyPair("D:\\tmp\\hzmq-test-0520_rabbitmq_ssl_client
\\client_rabbitmq_certificate.pem", "D:\\tmp\\hzmq-test-0520_rabbitmq_ssl_
client\\client_rabbitmq_key.pem")
if err != nil {
return err
}

rootCAs := x509.NewCertPool()
rootCAs.AppendCertsFromPEM(caCert)

tlsConf := &tls.Config{
RootCAs:      rootCAs,
Certificates: []tls.Certificate{cert},
//ServerName:   "localhost", // Optional
InsecureSkipVerify: true,
}

connection, err := amqp.DialTLS(amqpsURI, tlsConf)
if err != nil {
return fmt.Errorf("Dial: %s", err)
}
defer connection.Close()

log.Printf("got Connection, getting Channel")
```

```
channel, err := connection.Channel()
if err != nil {
return fmt.Errorf("Channel: %s", err)
}

log.Printf("got Channel, declaring %q Exchange (%q)", exchangeType, exchange)
if err := channel.ExchangeDeclare(
exchange,     // name
exchangeType, // type
true,         // durable
false,        // auto-deleted
false,        // internal
false,        // noWait
nil,          // arguments
); err != nil {
return fmt.Errorf("Exchange Declare: %s", err)
}

// Reliable publisher confirms require confirm.select support from the
// connection.
if reliable {
log.Printf("enabling publishing confirms.")
if err := channel.Confirm(false); err != nil {
return fmt.Errorf("Channel could not be put into confirm mode: %s", err)
}

confirms := channel.NotifyPublish(make(chan amqp.Confirmation, 1))

defer confirmOne(confirms)
}

log.Printf("declared Exchange, publishing %dB body (%q)", len(body), body)
if err = channel.Publish(
exchange,   // publish to an exchange
routingKey, // routing to 0 or more queues
false,      // mandatory
false,      // immediate
amqp.Publishing{
Headers:         amqp.Table{},
ContentType:     "text/plain",
ContentEncoding: "",
Body:            []byte(body),
DeliveryMode:    amqp.Transient, // 1=non-persistent, 2=persistent
Priority:        0,              // 0-9
},
); err != nil {
return fmt.Errorf("Exchange Publish: %s", err)
}

return nil
}

func confirmOne(confirms <-chan amqp.Confirmation) {
log.Printf("waiting for confirmation of one publishing")

if confirmed := <-confirms; confirmed.Ack {
log.Printf("confirmed delivery with delivery tag: %d", confirmed.DeliveryTag)
} else {
log.Printf("failed delivery of delivery tag: %d", confirmed.DeliveryTag)
}
}
```

ssl消费消息

```
package main
```

```
import (
"crypto/tls"
"crypto/x509"
"flag"
"fmt"
amqp "github.com/rabbitmq/amqp091-go"
"io/ioutil"
"log"
"time"
)

var (
uri          = flag.String("uri", "amqps://USERNAME:PASSWORD@10.10.33.196:
5671", "AMQP URI")
exchange     = flag.String("exchange", "test-exchange", "Durable, non-auto-
deleted AMQP exchange name")
exchangeType = flag.String("exchange-type", "direct", "Exchange type - direct|
fanout|topic|x-custom")
queue        = flag.String("queue", "test-queue", "Ephemeral AMQP queue name")
bindingKey   = flag.String("key", "test-key", "AMQP binding key")
consumerTag  = flag.String("consumer-tag", "simple-consumer", "AMQP consumer
 tag (should not be blank)")
lifetime     = flag.Duration("lifetime", 5*time.Second, "lifetime of process
 before shutdown (0s=infinite)")
)

func init() {
flag.Parse()
}

func main() {
c, err := NewConsumer(*uri, *exchange, *exchangeType, *queue, *bindingKey,
 *consumerTag)
if err != nil {
log.Fatalf("%s", err)
}

if *lifetime > 0 {
log.Printf("running for %s", *lifetime)
time.Sleep(*lifetime)
} else {
log.Printf("running forever")
select {}
}

log.Printf("shutting down")

if err := c.Shutdown(); err != nil {
log.Fatalf("error during shutdown: %s", err)
}
}

type Consumer struct {
conn    *amqp.Connection
channel *amqp.Channel
tag     string
done    chan error
}

func NewConsumer(amqpsURI, exchange, exchangeType, queueName, key, ctag
 string) (*Consumer, error) {
c := &Consumer{
conn:    nil,
channel: nil,
tag:     ctag,
done:    make(chan error),
}
```

```
var err error

log.Printf("dialing %q", amqpsURI)

caCert, err := ioutil.ReadFile("D:\\codeTest\\amqp\\_examples\\ssl\
\ca_certificate.pem")
if err != nil {
return nil, err
}

cert, err := tls.LoadX509KeyPair("D:\\codeTest\\amqp\\_examples\\ssl\\client_
openstack_certificate.pem", "D:\\codeTest\\amqp\\_examples\\ssl\\client_
openstack_key.pem")
if err != nil {
return nil, err
}

rootCAs := x509.NewCertPool()
rootCAs.AppendCertsFromPEM(caCert)

tlsConf := &tls.Config{
RootCAs:      rootCAs,
Certificates: []tls.Certificate{cert},
//ServerName:   "localhost", // Optional
InsecureSkipVerify: true,
}

c.conn, err = amqp.DialTLS(amqpsURI, tlsConf)
if err != nil {
return nil, fmt.Errorf("Dial: %s", err)
}

go func() {
fmt.Printf("closing: %s", <-c.conn.NotifyClose(make(chan *amqp.Error)))
}()

log.Printf("got Connection, getting Channel")
c.channel, err = c.conn.Channel()
if err != nil {
return nil, fmt.Errorf("Channel: %s", err)
}

log.Printf("got Channel, declaring Exchange (%q)", exchange)
if err = c.channel.ExchangeDeclare(
exchange,     // name of the exchange
exchangeType, // type
true,         // durable
false,        // delete when complete
false,        // internal
false,        // noWait
nil,          // arguments
); err != nil {
return nil, fmt.Errorf("Exchange Declare: %s", err)
}

log.Printf("declared Exchange, declaring Queue %q", queueName)
queue, err := c.channel.QueueDeclare(
queueName, // name of the queue
true,      // durable
false,     // delete when unused
false,     // exclusive
false,     // noWait
nil,       // arguments
)
if err != nil {
return nil, fmt.Errorf("Queue Declare: %s", err)
}
```

```
log.Printf("declared Queue (%q %d messages, %d consumers), binding to Exchange
 (key %q)",
queue.Name, queue.Messages, queue.Consumers, key)

if err = c.channel.QueueBind(
queue.Name, // name of the queue
key,        // bindingKey
exchange,   // sourceExchange
false,      // noWait
nil,        // arguments
); err != nil {
return nil, fmt.Errorf("Queue Bind: %s", err)
}

log.Printf("Queue bound to Exchange, starting Consume (consumer tag %q)",
 c.tag)
deliveries, err := c.channel.Consume(
queue.Name, // name
c.tag,      // consumerTag,
false,      // noAck
false,      // exclusive
false,      // noLocal
false,      // noWait
nil,        // arguments
)
if err != nil {
return nil, fmt.Errorf("Queue Consume: %s", err)
}

go handle(deliveries, c.done)

return c, nil
}

func (c *Consumer) Shutdown() error {
// will close() the deliveries channel
if err := c.channel.Cancel(c.tag, true); err != nil {
return fmt.Errorf("Consumer cancel failed: %s", err)
}

if err := c.conn.Close(); err != nil {
return fmt.Errorf("AMQP connection close error: %s", err)
}

defer log.Printf("AMQP shutdown OK")

// wait for handle() to exit
return <-c.done
}

func handle(deliveries <-chan amqp.Delivery, done chan error) {
for d := range deliveries {
log.Printf(
"got %dB delivery: [%v] %q",
len(d.Body),
d.DeliveryTag,
d.Body,
)
d.Ack(false)
}
log.Printf("handle: deliveries channel closed")
done <- nil
}
```

性能白皮书

常见问题

计费类

支持哪些付费方式?

支持包年包月和按需计费。

产品规格可选择哪些?

分布式消息服务RabbitMQ是基于高可用、分布式集群技术，完全兼容RabbitMQ开源社区，支持消息路由、事务消息、优先级队列、延迟队列、死信队列、镜像队列等功能的消息云服务，更多产品信息请参见 产品规格 。

如何选择磁盘空间?

在集群模式中，RabbitMQ需要对消息持久化写入到磁盘中，因此，在创建RabbitMQ实例选择存储空间时，建议根据业务消息体积预估以及镜像队列副本数量选择合适的存储空间。镜像队列副本数最大为集群的节点数。

例如：业务消息体积预估100GB，则磁盘容量最少应为100GB*节点数 + 预留磁盘大小100GB。

购买类

到期后如何续费?

在集群列表中点击“续费”，进入购买时长页面，购买成功续费成功。

产品订购时可选资源池节点不一致?

已上线资源池节点的剩余容量达到一定比例后，为确保老客户权益，将不再面向新客户开放，产品订购时的可选资源节点范围以实际为准。

收费依据有哪些?

根据规格和磁盘容量收费。

操作类

无法被路由的消息，去了哪里?

如果没有任何设置，无法路由的消息会被直接丢弃。

无法路由的情况：Routing key不正确。

解决方案：

1.使用mandatory=true配合ReturnListener，实现消息回发。

2.声明交换机时，指定备份交换机。

多个消费者监听一个队列时，消息如何分发?

1.Round-Robin（轮询）

默认的策略，消费者轮流、平均地收到消息。

2.Fair dispatch（公平分发）

如果要实现根据消费者的处理能力来分发消息，给空闲地消费者发送更多消息，可以用basicQos(int prefetch_count)来设置。prefetch_count含义：当消费者有多条消息没有响应ACK时，不再给这个消费者发送消息。

消息在什么时候会变成Dead Letter(死信)?

1.消息被拒绝并且没有设置重新入队：(NACK || Reject) && requeue == false

2.消息过期(消息或者队列的TTL设置)

3.消息堆积，并且队列达到最大长度，先入队的消息编程DL。

解决方案：可以在声明队列时，指定一个Dead Letter Exchange，来实现Dead Letter的转发，保证消息不会丢失。

如何进行消息持久化?

所谓持久化，就是RabbitMQ将内存中的数据（比如交换机、队列、消息等）固化到磁盘，以防止异常情况的发生时造成数据丢失。

持久化分类	说明
交换机持久化	在创建Exchange时设置durable参数参数。channel.exchangeDeclare(EXCHANGE_NAME, "direct", true);

持久化分类	说明
队列持久化	同样也是设置设置durable参数。持久化的队列会存盘，在服务器重启的时候可以保证不丢失相关信息。channel.queueDeclare(QUEUE_NAME, true, false, false, null);
消息持久化	即使交换机、队列持久化不会因为重启丢失，但是存储在队列中的消息仍然会丢失。解决的办法就是设置消息的投递模式为2，即代表持久化(JAVA)。理论上，可以将所有的消息都设置为持久化，但是这会严重影响RabbitMQ性能，因为写入到磁盘的速度可比写入到内存的速度慢非常多。因此，在选择是否要持久化消息时，需要在可靠性和吞吐量之间做一个权衡。

RabbitMQ专享实例是否支持公网访问?

RabbitMQ专享实例支持公网访问。

在创建RabbitMQ专享实例的“更多”选项中，选择开启“公网访问”可自主控制是否进行公网访问，并选择已购买的弹性IP及带宽。或创建完后，在实例详情页中将公网访问开关打开。

RabbitMQ实例是否支持跨VPC和跨子网访问?

RabbitMQ实例支持跨VPC和子网访问，可以通过创建VPC对等连接，将两个VPC的网络打通，实现跨VPC访问实例。

SSL方式连接RabbitMQ实例失败?

首先排查安全组的入方向规则，是否放开了端口5671（SSL方式访问）或5672（非SSL访问）。

其次，参考如下内容配置SSL单向认证：

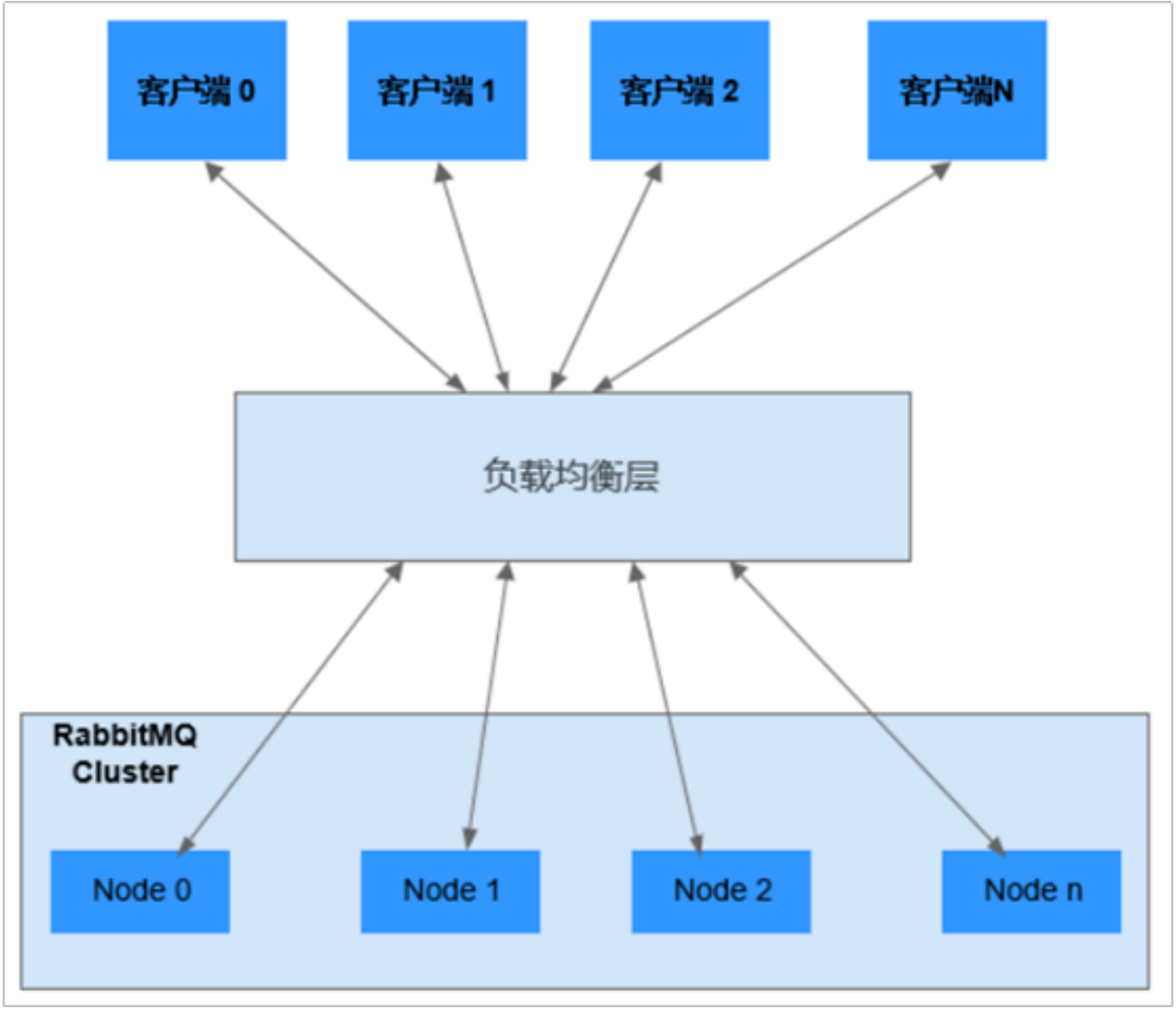
```
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(host);
    factory.setPort(port);
    factory.setUsername(user);
    factory.setPassword(password);
    factory.useSslProtocol();
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
```

客户端是否可以通过DNAT方式访问RabbitMQ实例?

可以。

为什么RabbitMQ集群只有一个连接地址?

RabbitMQ集群实例的连接地址，实际上是实例的LVS节点地址（负载均衡地址），客户端连接实例时，通过负载均衡器将客户端请求分发到集群实例的各个节点。



RabbitMQ实例集群的队列是否有备份?

RabbitMQ实例默认开启了镜像队列，会在集群中多个代理上备份队列的副本，当某个代理故障，集群会从其他正常的代理中选择一个代理，用来同步队列数据。

RabbitMQ支持双向认证吗?

不支持。

RabbitMQ实例是否支持扩容?

不支持。

RabbitMQ实例是否支持修改可用区?

不支持，您可以重新购买实例，以满足可用区要求。

RabbitMQ客户端连接报错原因分析?

RabbitMQ客户端连接失败，可能原因包括地址、端口填错、用户名或者密码填错。

连接地址不正确

```
Exception in thread "main"java.net.SocketTimeoutException: connect timed out
at java.net.PlainSocketImpl.socketConnect(NativeMethod)
at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:
350)
```

端口不正确

```
Exception in thread "main"java.net.ConnectException:Connecti
on refused (Connection refused)
at java.net.PlainSocketImpl.socketConnect(NativeMethod)
at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:
350)
at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.
java:206)
```

用户名或密码错误

```
Exception in thread "main"com.rabbitmq.client.
AuthenticationFailureException: ACCESS_REFUSED -
Login was refused using authentication mechanism PLAIN. For details
 see the broker logfile.
at com.rabbitmq.client.impl.AMQConnection.start(AMQConnection.java:351)
at com.rabbitmq.client.impl.recovery.RecoveryAwareAMQConnectionFactory.
newConnection(RecoveryAwareAMQConnectionFactory.java:64)
```

RabbitMQ实例是否支持不同的子网?

支持。

客户端与实例在相同VPC内，可以跨子网段访问。同一个VPC内的子网默认可以进行通信。

客户端与实例在不同VPC时，需建立VPC对等连接。

客户端是否可以连接同个RabbitMQ下多个Vhost?

客户端可以连接同个RabbitMQ下多个Vhost。

Vhost（Virtual Hosts虚拟主机）是RabbitMQ的基本特性，每个Vhost相当于一个独立的虚拟消息服务器，每个Vhost数据目录不同，拥有自己的队列、交换器和权限控制机制。性能上，连接多个Vhost和单独使用一个Vhost差别不大。

消息创建时间在哪设置?

消息创建时间是在生产消息时由生产客户端设置。

RabbitMQ是否支持跨Region部署?

不支持跨Region部署。

如何清空队列数据?

（1）进入“分布式消息服务RabbitMQ”控制中心；

（2）在“实例列表”页面点击对应的RabbitMQ实例；

（3）在“队列管理”页面点击要清空的队列；

（4）在“清除消息”页面点击清空队列。

RabbitMQ支持升级CPU和内存吗?

RabbitMQ支持扩容规格。

如何设置Message ID?

如需追踪和识别消息，可以在分布式消息服务RabbitMQ的Producer客户端设置Message ID属性，为每条消息设置唯一标识符。

在分布式消息服务RabbitMQ的Producer客户端设置Basic.Properties的messageid属性。示例代码如下：

```
AMQP.BasicProperties props =newAMQP.BasicProperties.
Builder().messageId("messageid").build();
channel.basicPublish("ExchangeName","RoutingKey",true, props,("####
Body").getBytes());
```

Message ID（消息标识符）是消息的可选属性，类型为String。Message ID在业务上通常被设置为唯一，适用于追踪和识别销售单、工单等需要保证消息唯一的场景。分布式消息服务RabbitMQ服务端不会对消息进行幂等处理。如需实现消息幂等，即如果消息重试多次，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，并且多次消费没有对系统产生副作用，在为每条消息设置唯一Message ID的基础上，还需要在分布式消息服务RabbitMQ的Consumer客户端对消息进行幂等处理。

RabbitMQ使用的版本是多少?

服务端RabbitMQ的版本有3.8.9和3.8.35。

RabbitMQ实例SSL连接的协议版本号是多少?

服务端支持协议版本号：SSL v3、TLS v1、TLS v1.1、TLS v1.2、TLS v1.3。

管理类

重启RabbitMQ实例时，若其中一台RabbitMQ重启失败，会如何处理?

重启RabbitMQ实例时，不会重启实例所在虚拟机，仅重启RabbitMQ进程。

重启集群实例时，若其中一台RabbitMQ进程重启失败，则重启后实例状态依然为“运行中”，并提示“部分节点故障”。在每台虚拟机上都有RabbitMQ的守护进程，定时检查RabbitMQ进程是否存在，当进程不存在时会自动拉起RabbitMQ进程。

如果RabbitMQ实例异常持续超过1分钟，会上报告警。

RabbitMQ集群实例如何均衡分发请求到每个虚拟机?

集群内部使用LVS做负载均衡，由LVS将请求均衡分发到每个虚拟机节点。

RabbitMQ实例是否支持持久化，如何定时备份数据?

RabbitMQ支持消息数据持久化，可从客户端连接RabbitMQ并设置消息持久化，也可在RabbitMQ集群管理工具界面创建队列时设置消息持久化。

不支持客户自定义定时备份数据，或从界面触发备份数据。

RabbitMQ实例开启SSL开关后，证书如何获取?

RabbitMQ实例开启SSL后只做单向认证，不需要证书。

RabbitMQ实例的SSL开关是否支持修改?

不支持动态修改，即如果实例创建时没有选择开启，创建完成之后，不支持修改，建议在实例创建时将开关打开。

RabbitMQ实例支持延迟消息队列么?

RabbitMQ可以通过设置消息的有效期、和死信队列来实现延迟消息。

同时，也提供安装插件实现延迟消息。

消息堆积对业务有什么影响?

为了保证服务的稳定可靠，分布式消息服务RabbitMQ版采用了默认的40%高水位配置。当内存占用率达到40%高水位后，会触发流控，生产者流程会被阻塞。消息堆积可能造成内存高水位，为了避免高水位的产生，请及时消费积压在队列中的消息。

消费的最长保留时间是多久?

一般情况下消息如果未被消费会一直保留，只有被消费后，才会被删除。但是如果设置了过期时间（TTL），则以TTL时间为准。

最佳实践

RabbitMQ元数据迁移

RabbitMQ元数据迁移指将用户线下实例的rabbitmq实例元数据迁移到线上实例。

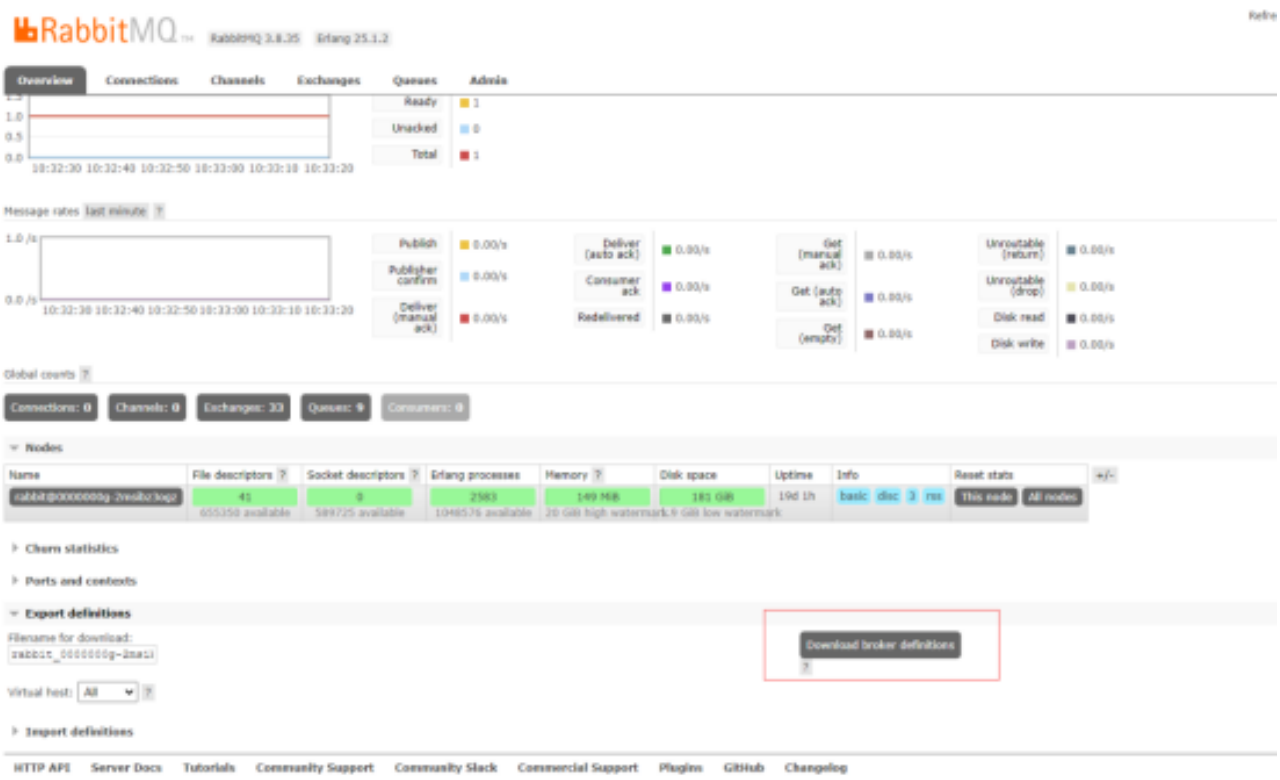
背景信息

RabbitMQ集群元数据是指RabbitMQ集群的信息，包括User、Vhost、Queue、Exchange、Binding Key、Permission、Parameter等信息。RabbitMQ集群元数据存储于RabbitMQ集群的内部数据库，在集群的各个节点之间自动复制。集群中的每个节点都有自己的元数据副本。当某个节点的元数据变更时，所有节点的元数据都会同步更新。因此，集群的各个节点的元数据被导出时都是相同的。RabbitMQ集群元数据可以被导出成一份JSON文件，然后被导入另一个RabbitMQ集群，实现RabbitMQ集群元数据备份。

迁移元数据上云是指将开源RabbitMQ集群的元数据迁移到天翼云分布式消息服务RabbitMQ实例。分布式消息服务RabbitMQ是天翼云提供的全托管消息队列服务，兼容开源RabbitMQ。您可以将RabbitMQ集群元数据导出，然后导入分布式消息服务RabbitMQ实例，分布式消息服务RabbitMQ会根据成功导入的元数据在目标分布式消息服务RabbitMQ实例中创建对应的Vhost、Queue、Exchange、Binding，实现RabbitMQ集群元数据迁移上云。您可以将全部Vhost信息导入分布式消息服务RabbitMQ实例，也可以根据需要将某个Vhost信息导入分布式消息服务RabbitMQ实例中的Vhost。

迁移元数据

（1）在RabbitMQ WebUI页面查看，如图所示。



Overview视图中，点击“Download broker definitions”按钮下载集群元数据。得到rabbitmq元数据的json文件。

（2）上线rabbitmq实例导入元数据。

如上图所示，先点击选择文件，选择上一步骤导出的json文件，再点击导入配置。

注意：在线下的Rabbitmq集群中不能含有用户名rabbitmq，否则会被覆盖。用户名rabbitmq是天翼云Rabbitmq内部使用的管理账号。

如何实现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。

在队列声明时使用对应参数设置。

```
//####

HashMap map = new HashMap<>();

//########

map.put("x-max-length",10 );

//###########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。

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。

RabbitMQ接入

获取SDK

引入依赖

```
<dependencies>
    <dependency>
```

```
        <groupId>com.rabbitmq</groupId>
        <artifactId>amqp-client</artifactId>
        <version>5.8.0</version>
    </dependency>
</dependencies>
```

接入方式

安全接入点

RabbitMQ 安全接入点支持 "PLAIN"、"AMQPLAIN" 授权机制。

1、访问控制

RabbitMQ "PLAIN"、"AMQPLAIN"授权机制需要创建用户，从而获得对应虚拟主机的访问权限。

2、接入步骤

（1）新建用户（集群管理->用户->新建用户）

（2）运行demo

客户端关键参数设置

"PLAIN"、"AMQPLAIN" 授权机制的客户端关键参数配置

```
        String host = "192.168.0.0"; //#####ip
        Integer port = 5672;        //#####port
        String username = "xxx";       //############
        String password = "xxx";
        String vhost = "/";

        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setHost(host);
        connectionFactory.setPort(port);
        connectionFactory.setUsername(username);
        connectionFactory.setPassword(password);
        connectionFactory.setVirtualHost(vhost);
```

SSL接入点

RabbitMQ 安全接入点支持 "EXTERNAL" 授权机制

1、访问控制

无

2、接入步骤

（1）下载SSL证书（实例概览->导出服务->下载SSL文件）

（2）运行demo

客户端关键参数设置

"EXTERNAL" 授权机制的客户端关键参数配置

```
        String host = "192.168.0.0";  //SSL###ip
        int port = 5671;    //SSL###port
        //##2#ssl#############, ###########2.2.1#########
        String ksFile = "D:\\tmp\\ssl\\client_rabbitmq_key.p12";
        String tksFile = "D:\\tmp\\ssl\\truststore";
        String vhost = "/";
```

```
        char[] keyPassphrase = "W3zT_98Zz9Io".toCharArray();
        KeyStore ks = KeyStore.getInstance("PKCS12");
        ks.load(new FileInputStream(ksFile), keyPassphrase);

        KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks, keyPassphrase);

        char[] trustPassphrase = null;
        trustPassphrase = "W3zT_98Zz9Io".toCharArray();
        KeyStore tks = KeyStore.getInstance("JKS");
        tks.load(new FileInputStream(tksFile), trustPassphrase);

        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init(tks);
        SSLContext c = SSLContext.getInstance("tlsv1.2");
        c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setHost(host);
        connectionFactory.setPort(port);
        connectionFactory.setVirtualHost(vhost);
        connectionFactory.setSaslConfig(DefaultSaslConfig.EXTERNAL);
        connectionFactory.useSslProtocol(c);
```

代码示例

安全接入点(PLAIN、AMQPLAIN授权机制)

```
import com.rabbitmq.client.*;
import java.io.IOException;

public class RabbitmqAmqpDemo {
    public static void main(String[] args) throws Exception {
        String host = "192.168.0.0"; //#####ip
        Integer port = 5672;        //#####port
        String username = "xxx";       //#############
        String password = "xxx";       //############
        String vhost = "/";
        String exchangeName = "ex_test";
        String queueName = "qu_test";

        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setHost(host);
        connectionFactory.setPort(port);
        connectionFactory.setUsername(username);
        connectionFactory.setPassword(password);
        connectionFactory.setVirtualHost(vhost);
        Connection connection = connectionFactory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT,
 true);
        channel.queueDeclare(queueName, true, false, false, null);
        channel.queueBind(queueName, exchangeName, "test");

        String message = "Hello Aop";
        for (int i = 0; i < 10; i++) {
            channel.basicPublish(exchangeName, "test", null,
 message.getBytes());
            System.out.println("######");
        }
        Channel consumeChannel = connection.createChannel();
        Consumer consumer = new DefaultConsumer(consumeChannel) {
            @Override
```

```
            public void handleDelivery(String consumerTag, Envelope envelope,
 AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {
                String messageGet = new String(body, "UTF-8");
                if (messageGet.equals(message)) {
                    System.out.println("######");
                }
            }
        };
        consumeChannel.setDefaultConsumer(consumer);
        consumeChannel.basicConsume(queueName, false, consumer);
        Thread.sleep(10000);
    }
}
```

SSL接入点(EXTERNAL授权机制)

```
import com.rabbitmq.client.*;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManagerFactory;
import java.io.FileInputStream;
import java.io.IOException;
import java.security.KeyStore;

public class RabbitmqExternalDemo {
    public static void main(String[] args) throws Exception {
        String host = "192.168.0.0";  //SSL###ip
        int port = 5671;    //SSL###port
        //##2#ssl##############, ############2.2.1#########
        String ksFile = "D:\\tmp\\ssl\\client_rabbitmq_key.p12";
        String tksFile = "D:\\tmp\\ssl\\truststore";
        String vhost = "/";
        String exchangeName = "ex_test";
        String queueName = "qu_test";

        char[] keyPassphrase = "W3zT_98Zz9Io".toCharArray();
        KeyStore ks = KeyStore.getInstance("PKCS12");
        ks.load(new FileInputStream(ksFile), keyPassphrase);

        KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks, keyPassphrase);

        char[] trustPassphrase = null;
        trustPassphrase = "W3zT_98Zz9Io".toCharArray();
        KeyStore tks = KeyStore.getInstance("JKS");
        tks.load(new FileInputStream(tksFile), trustPassphrase);

        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init(tks);
        SSLContext c = SSLContext.getInstance("tlsv1.2");
        c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setHost(host);
        connectionFactory.setPort(port);
        connectionFactory.setVirtualHost(vhost);
        connectionFactory.setSaslConfig(DefaultSaslConfig.EXTERNAL);
        connectionFactory.useSslProtocol(c);

        Connection connection = connectionFactory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT,
 true);
        channel.queueDeclare(queueName, true, false, false, null);
        channel.queueBind(queueName, exchangeName, "test");
```

```
        String message = "Hello Aop";
        for (int i = 0; i < 10; i++) {
            channel.basicPublish(exchangeName, "test", null,
 message.getBytes());
            System.out.println("######");
        }
        Channel consumeChannel = connection.createChannel();
        Consumer consumer = new DefaultConsumer(consumeChannel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
 AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {
                String messageGet = new String(body, "UTF-8");
                if (messageGet.equals(message)) {
                    System.out.println("######");
                }
            }
        };
        consumeChannel.setDefaultConsumer(consumer);
        consumeChannel.basicConsume(queueName, false, consumer);
        Thread.sleep(10000);
    }
}
```

消息幂等

如果消息重复消费会影响您的业务处理，要对消息做幂等处理。本文介绍消息幂等的概念、适用场景以及处理方法。

概念

在消息领域，幂等是指Consumer重复消费某条消息时，重复消费的结果与消费一次的结果是相同的，并且多次消费并未对业务系统产生任何负面影响。

例如，在支付场景下，Consumer消费扣款消息，对一笔订单执行扣款操作，扣款金额为500元。如果因网络不稳定等原因导致扣款消息重复投递，Consumer重复消费了该扣款消息，但最终的业务结果是只扣款一次，扣费500元，且用户的扣款记录中对应的订单只有一条扣款流水，不会多次扣除费用。那么这次扣款操作是符合要求的，整个消费过程实现了消息幂等。

适用场景

在互联网应用中，尤其在网络不稳定的情况下，分布式消息服务RabbitMQ的消息有可能会出现重复。如果消息重复消费会影响您的业务处理，请对消息做幂等处理。消息重复的可能原因如下：

- 发送时消息重复

当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或者客户端宕机，导致服务端对客户端应答失败。 如果此时Producer意识到消息发送失败并尝试再次发送消息，Consumer后续会收到两条内容相同并且Message ID也相同的消息。

- 投递时消息重复

消息消费的场景下，消息已投递到Consumer并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。为了保证消息至少被消费一次，分布式消息服务RabbitMQ的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，Consumer后续会收到两条内容相同并且Message ID也相同的消息。

- 负载均衡时消息重复（包括但不限于网络抖动、服务端重启以及Consumer应用重启）

当分布式消息服务RabbitMQ的服务端或客户端重启、扩容或缩容时，会触发Rebalance，此时Consumer可能会收到重复消息。

处理方法

以Message ID为幂等键对消息进行幂等处理的步骤如下：

（1）在数据库中创建一张unique key索引为唯一Message ID的表。

（2）在Producer客户端为每条消息设置唯一Message ID。

设置唯一Message ID的示例代码如下：

```
AMQP.BasicProperties props =newAMQP.BasicProperties.Builder().messageId(UUID.
randomUUID().toString()).build();
channel.basicPublish("ExchangeName","RoutingKey",true, props,("###
#"+ i).getBytes());
```

（3）在Consumer客户端根据唯一Message ID对消息进行幂等处理。

根据唯一Message ID进行幂等处理的示例代码如下：

```
channel.basicConsume(Producer.QueueName, false, "MyConsumerTag",
    new DefaultConsumer(channel) {
    @Override public void handleDelivery(String consumerTag, Envelope env,
                AMQP.BasicProperties properties, byte[] body) throws
 IOException {
        // 1. ############
        try{
            String messageId = properties.getMessageId();
            // Message ID######unique key####
            // 2. ########
            idempTable.insert(messageId);
            // 3. ##################
            // 4. ########// ########ACK#######ACK#
            channel.basicAck(env.getDeliveryTag(), false);
        }
        catch (######### e){
            // ############
            channel.basicAck(env.getDeliveryTag(), false);
        }
    }
}
);
```

网络异常自动恢复

由于服务端升级、服务端重启、网络抖动等原因，服务端和客户端的网络连接可能会断开。本文介绍如何在客户端设置Connection和Topology自动恢复，使Connection和Topology在网络连接断开后自动恢复，避免断连对业务造成影响。

触发原因

触发Connection自动恢复的原因如下：

- Connection的I/O抛出异常。
- Socket读取操作超时。
- 检测到服务端心跳丢失。

恢复方法

4.0.0及以上版本Java客户端默认开启Connection和Topology自动恢复，无需在代码中设置。

在客户端开启Connection和Topology（Queue、Exchange、Binding、Consumer）自动恢复的方法如下：

- factory.setAutomaticRecoveryEnabled(boolean)：用于开启或关闭Connection自动恢复。
- factory.setNetworkRecoveryInterval(long)：用于设置重试时间间隔。如果Connection自动恢复异常，设置了Connection自动恢复的客户端将在一段固定时间间隔（默认为5秒）后重试。
- factory.setTopologyRecoveryEnabled(boolean)：用于开启Topology自动恢复。Topology包括Queue、Exchange、Binding、Consumer。

示例代码

开启Connection和Topology自动恢复的客户端示例代码如下：

```
ConnectionFactory factory =newConnectionFactory();
// ##############RabbitMQ############
factory.setHost("192.168.x.x");
// ##Vhost##
factory.setVirtualHost("/");
// ##
factory.setPort(5672);
// ################
factory.setConnectionTimeout(30*1000);
factory.setHandshakeTimeout(30*1000);
factory.setShutdownTimeout(0);
// ##Connection#####
factory.setAutomaticRecoveryEnabled(true);
// ##Connection#######10##
factory.setNetworkRecoveryInterval(10000);
// ##Topology#####
factory.setTopologyRecoveryEnabled(true);
Connection connection = factory.newConnection();
```

恢复限制

Connection自动恢复的限制如下：

- Connection断开需要一定的时间检测。要确保这段时间内发送的消息不丢失，需使用Publisher Confirms实现可靠发送。
- Channel异常导致Connection断开时，不会触发Connection自动恢复。Channel异常通常为应用级别的问题，需要使用方自行处理。
- Connection自动恢复不会使Channel也自动恢复。

节点重启后消费者如何重连

本文介绍节点重启后消费者如何重连，以Java中使用的RabbitMQ客户端amqp-client为例。

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，能够解决上面的两种错误，实现消费者的持续消费。

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

public class MyRabbitConsumer {

    public static void main(String... args) throws IOException,
 TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("192.168.x.x");
        factory.setPort(5672);
        factory.setUsername("name");
        factory.setPassword("password");
        Connection connection = factory.newConnection();
        createNewConnection(connection);
    }

    public static void createNewConnection(Connection connection) {
        try {
            Channel channel = connection.createChannel();
            channel.basicQos(64);
            channel.basicConsume("queue-1", false, new CustomConsumer(channel,
 connection));
        } catch (Exception e) {
            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 {
            createNewConnection(_connection);
        }

        @Override
        public void handleShutdownSignal(String consumerTag, Shutdown
SignalException sig) {
            createNewConnection(_connection);
        }

        @Override
        public void handleRecoverOk(String consumerTag) {}

        @Override
        public void handleDelivery(String consumerTag, Envelope env, AMQP.
BasicProperties prop, byte[] body) throws IOException {
            String message = new String(body, StandardCharsets.UTF_8);
            System.out.println("####: " + message);
            _channel.basicAck(env.getDeliveryTag(), false);
        }
    }
}
```

使用AMQProxy解决PHP等客户端Connection复用问题

AMQProxy是一款开源AMQP代理服务，具备复用AMQP Connection的能力。可以通过该代理服务使原本只能使用短连接的客户端（例如PHP客户端）使用长连接，从而减少网络资源消耗和分布式消息服务RabbitMQ资源消耗。

前提条件

如果您要使用SSL连接AMQProxy和分布式消息服务RabbitMQ，请确保您的客户端服务器已安装OpenSSL。

背景

部分语言的客户端，例如PHP客户端，无法使用长连接，会频繁地开启或关闭Connection，消耗大量的网络资源和分布式消息服务RabbitMQ资源，从而对分布式消息服务RabbitMQ造成巨大压力。

AMQProxy

AMQProxy是Cloud AMQP提供的开源AMQP代理服务。客户端可以通过该代理服务与分布式消息服务RabbitMQ保持长连接。当客户端服务器部署AMQProxy后，客户端和分布式消息服务RabbitMQ之间的请求都会先发送到AMQProxy，然后由AMQProxy转发到对方。

AMQProxy处理客户端发起的Connection相关请求的逻辑如下：

如果客户端发送开启Connection的请求，AMQProxy将根据用户名、密码、Vhost查找当前是否有合适的Connection可以复用，如果有就复用该Connection，如果没有就由AMQProxy代替客户端和分布式消息服务RabbitMQ开启Connection。

如果客户端发送关闭Connection的请求，AMQProxy会直接应答OK，但并不会关闭与分布式消息服务RabbitMQ的Connection，当该客户端下次再请求开启Connection时，AMQProxy会直接使用该Connection。

部署AMQProxy

（1）下载和安装AMQProxy。

开源项目地址：https://github.com/cloudamqp/amqproxy

releases目录下有安装包，下载后可以在本地解压

（2）启动AMQProxy

./amqproxy -l LISTEN_ADDRESS -p LISTEN_PORT AMQP_URL

参数	描述
LISTEN_ADDRESS	AMQProxy IP地址。由于是在客户端服务器部署AMQProxy，因此可以直接使用本机地址127.0.0.1。
LISTEN_PORT	AMQProxy监听端口。客户端请求通过该端口发送到AMQProxy。该端口可以为任何可用的端口，例如5673。
AMQP_URL	分布式消息服务RabbitMQ实例的URL。格式为{amqp|amqps}://{endpoint}。amqp：AMQP协议。不使用SSL连接时使用。amqps：AMQP/SSL协议。使用SSL连接时使用。endpoint：分布式消息服务RabbitMQ实例的接入点。可以在分布式消息服务RabbitMQ控制台的实例详情页面查看。

示例命令如下：

./amqproxy -l 127.0.0.1 -p 5673 amqps://192.168.0.100:5672

返回示例如下：

Proxy upstream: 192.168.0.100:5672 TLS

Proxy listening on 127.0.0.1:5673

0 clients 0 upstreams

参数	描述
clients	客户端和AMQProxy的Connection数量。
upstreams	AMQProxy和分布式消息服务RabbitMQ实例的Connection数量。

（3）在客户端代码中将Host和端口修改为AMQProxy IP地址和监听端口。

factory.setHost("127.0.0.1");

```
factory.setPort(5673);
```

延迟消息

延时消息定义

延时消息是指消息在发送后不会立即投递给消费者，而是在指定的延迟时间之后才可被消费。

开源 RabbitMQ 本身不直接支持原生延时消息语义，但可以通过现有机制组合或官方插件实现类似能力。

应用场景

延时消息适用于需要定时触发或延迟处理的业务场景，例如：

- 订单超时关闭

  下单成功后发送延时消息，30 分钟后检查支付状态，未支付则关闭订单。
- 提醒通知

  预约活动开始前，通过延时消息发送提醒。
- 任务调度

  延迟执行数据清理、报表生成、状态检查等后台任务。

实现方案对比

方案	实现原理	特点
死信队列（DLX）+ TTL	利用消息或队列 TTL 过期后转入死信交换机，再重新路由到消费队列	无需插件，兼容性好；配置复杂，延迟精度有限
延时消息插件	使用官方延时交换机，消息到期后才进入正常路由流程	使用简单、精度高；需安装插件，存在版本依赖

方案一：死信队列（DLX）+ TTL

实现原理

RabbitMQ 支持为：

- 消息 设置 TTL（`expiration`）
- 队列 设置 TTL（`x-message-ttl`）

当消息到达 TTL 后不会立即删除，而是被投递到死信交换机（Dead Letter Exchange, DLX），再由 DLX 路由到目标队列，从而实现“延迟消费”。

流程说明

1. 创建 死信交换机（DLX） 和 死信队列

2. 创建业务队列，并配置：
   - `x-dead-letter-exchange`
   - `x-dead-letter-routing-key`
3. 消息进入业务队列并等待 TTL 到期
4. 消息过期后转入 DLX
5. 消费者从最终队列中消费消息

代码示例

```
// #######
channel.exchangeDeclare("dlx_exchange", "direct", true);
// ######
channel.queueDeclare("dlx_queue", true, false, false, null);
channel.queueBind("dlx_queue", "dlx_exchange", "dlx_routing_key");
// #############Map<String, Object> args = new HashMap<>();
args.put("x-dead-letter-exchange", "dlx_exchange");
args.put("x-dead-letter-routing-key", "dlx_routing_key");
args.put("x-message-ttl", 10000); //###TTL#10#
channel.queueDeclare("business_queue", true, false, false, args);
//######## TTL#
AMQP.BasicProperties props = new AMQP.BasicProperties.Builder()
        .expiration("5000") // 5 ####
        .build();
channel.basicPublish("","business_queue",props,"####".getBytes(Standard
Charsets.UTF_8));
```

特点说明

- TTL 优先级
  - 同时设置消息 TTL 和队列 TTL 时，取较小值
- 延迟精度
  - 依赖队列头部过期检查机制，精度非严格实时
- 实现成本
  - 无需额外组件，但配置相对复杂

方案二：延时消息插件方案

实现原理

通过声明特殊类型的交换机，使消息在 Exchange 层面延迟投递。消息在延迟时间到期前不会进入正常路由流程。

前提条件

- RabbitMQ 版本 ≥ 3.8
- 启用官方插件：

rabbitmq-plugins enable rabbitmq_delayed_message_exchange

天翼云RabbitMQ默认启用延迟交换器交换器插件，无需操作

使用步骤

1. 声明 x-delayed-message 类型交换机
2. 指定底层路由类型（x-delayed-type）
3. 发送消息时在 Header 中设置 x-delay（毫秒）

代码示例

```
//#######
Map<String, Object> args = new HashMap<>();
args.put("x-delayed-type","direct");
channel.exchangeDeclare("delayed_exchange","x-delayed-
message",true,false,args);
// #######
channel.queueDeclare("target_queue",true,false,false,null);
channel.queueBind("target_queue","delayed_exchange","routing_key");
//######### 5 ##
Map<String, Object> headers = new HashMap<>();
headers.put("x-delay",5000);
AMQP.BasicProperties props = new AMQP.BasicProperties.Builder()
        .headers(headers)
        .build();
channel.basicPublish("delayed_exchange","routing_key",props,"####".getBytes(
StandardCharsets.UTF_8));
```

特点说明

- 延迟精度高，使用方式直观
- 延迟逻辑在 Exchange 层完成
- 依赖插件及 RabbitMQ 版本兼容性

注意事项

精度与可靠性

- DLX + TTL
  - 可能存在毫秒级到秒级误差
- 插件方案
  - 延迟精度更高，更适合严格定时场景

资源消耗

- 延时消息在到期前会占用 Broker 内存
- 大量长延时消息需重点监控：
  - 队列堆积
  - 内存水位
  - 磁盘使用率

消息顺序

- 延时消息可能打乱原始发送顺序
- 对顺序敏感的业务需在消费端自行处理

总结

- 开源 RabbitMQ 不提供原生延时语义
- 可通过两种方式实现延时消息能力：
  - 死信队列 + TTL：通用、稳定，但灵活性有限
  - 延时消息插件：精度高、使用简单，但依赖部署环境
- 实际选型需综合考虑：
  - 延迟精度要求
  - 运维复杂度
  - 消息规模与资源成本

API参考

API使用说明

翼云OpenAPI门户提供了产品的API 文档、API调试、SDK中心等。

关于用户如何使用分布式消息服务RabbitMQ产品API的详细介绍，请参见 使用API 。您可以在OpenAPI门户可以了解到具体的调用前必知、API概览、如何调用API以及具体的API的接口详细说明。

说明

分布式消息服务RabbitMQ提供两种版本接口供用户调用

- V3版本：适用于 华东1、华北2、西南1、华南2、上海36、青岛20、长沙42、南昌5、武汉41、杭州7、西南2-贵州、太原4、郑州5、西安7、呼和浩特3 资源池。
- V2版本：适用于 芜湖2、南京3、上海7、重庆2、乌鲁木齐27、石家庄20、内蒙6、保定、北京5 资源池。部分资源池已不再开放新实例订购，仅存量实例调用。

如何调用API

认证鉴权

获取天翼云AK/SK:

1. 点击天翼云门户首页右上角的“登录”，输入登录的用户名和密码。
2. 点击右上角的“我的”， 点击个人中心。
3. 在左侧下拉列表中单击“安全设置”。
4. 在安全设置页面的“用户AccessKey”列，点击“新建”按钮，获取访问密钥（AK/SK）。新建成功后，“新建”按钮变为“查看”，您可以点击并查看并复制AK/SK。

认证鉴权介绍：

AK/SK认证：通过AK（Access Key ID）/SK（Secret Access Key）加密调用请求。

AK/SK认证就是使用AK/SK对请求进行签名，在请求时将签名信息添加到消息头，从而通过身份认证。

参数	描述
AK（Access Key ID）	访问密钥ID。与私有访问密钥关联的唯一标识符；访问密钥ID和私有访问密钥一起使用，对请求进行加密签名

参数	描述
SK（Secret Access Key）	私有访问密钥。与访问密钥ID结合使用，对请求进行加密签名，可标识发送方，并防止请求被修改

使用AK/SK认证时，您可以基于签名算法使用AK/SK对请求进行签名，也可以使用专门的签名SDK对请求进行签名。详细的签名方法和SDK使用方法请参见API签名指南。

说明： 签名SDK只提供签名功能，与服务提供的SDK不同，使用时请注意。

构造请求

本节介绍REST API请求的组成，以调用云主机产品的"根据regionID查询用户资源"举例说明如何调用该API。

请求示例

```
POST https://ctecs-xxx.ctapi.ctyun.cn/v4/region/customerResources?
prodInstId=11&startTime=2021-04-04T06:01:46ZHTTP/1.1
Content-Type:application/json
ctyun-eop-request-id:0ffb9b07-d5a8-4e19-b3ce-12dfb9705a1d
Eop-Authorization:4a4bdc57e06542199b5f98d4cd107be2 Headers=ctyun-eop-request-
id;eop-date Signature=b2WEo4nh9ewn6SWOP0ii5g8L0z9CT5gprpDWntlCX9I=
Eop-date:20221107T093029Z
```

请求URI

{URI-scheme}://{Endpoint}/{resource-path}?{query-string}

参数	描述
URI-scheme	表示用于传输请求的协议，当前所有API均采用HTTPS协议
Endpoint	指定承载REST服务端点的服务器域名或IP，不同服务不同区域的Endpoint不同，您可以从地区和终端节点获取。例如云主机产品服务在某个区域的Endpoint为“ctecs-xxx.ctapi.ctyun.cn”
resource-path	资源路径，即API访问路径。从具体API的URI模块获取，例如"根据regionID查询用户资源"API的resource-path为“/v4/region/customerResources”
query-string	查询参数，是可选部分，并不是每个API都有查询参数。查询参数前面需要带一个“?”，形式为“参数名=参数取值”，例如“?limit=10”，表示查询不超过10条数据

示例：

云主机-根据regionID查询用户资源，则需使用云主机产品在对应区域的Endpoint（ctecs-xxx.ctapi.ctyun.cn），并在"根据regionID查询用户资源"的URI部分找到resource-path（/v4/region/customerResources），拼接起来如下所示。

https://ctecs-xxx.ctapi.ctyun.cn/v4/region/customerResources

说明： 为方便查看，在每个具体API的URI部分，只给出resource-path部分，并将请求方法写在一起。这是因为URI-scheme都是HTTPS，而Endpoint在同一个区域也相同，所以简洁起见将这两部分省略。

{resource-path}资源路径，使用编码的规范URI示例：

```
##
/v4/region/customerResources api/code
##
/v4/region/customerResources%20api/code
```

说明：

资源路径{resource-path}是从HTTP Host结束到查询字符串参数（如果有）的问号字符（“?”）中间的部分。 需要根据RFC 3986标准化URI路径规范移除冗余和相对路径部分。路径中每个部分都必须进行URI编码。

{query-string}查询参数，使用规范查询字符串示例：

```
##
prodInstId=11&startTime=2021-04-04T06:01:46Z
##
prodInstId=11&startTime=2021-04-04T06%3A01%3A46Z
```

说明：

{query-string}为键值对，使用等号字符(=)拼接。值需进行URI编码，同样需要满足RFC 3986关于URL编码规范的定义要求。键则不需要。对没有值的参数使用空字符串。

在每个参数值后追加与字符(&)，列表中最后一个值除外。

使用 %XY 对所有其他字符进行百分比编码，其中“X”和“Y”为十六进制字符（0-9和大写字母A-F）。例如，空格字符必须编码为 %20（不像某些编码方案那样使用“+”），扩展UTF-8字符必须采用格式 %XY%ZA %BC。

RFC 3986规范的具体要求：

字符A-Z、a-z、0-9 以及字符-、_、.、不编码。

对其它ASCII码字符进行编码。编码格式为%加上16进制的ASCII码。例如半角双引号（"）将被编码为%22。空格编码成%20，而不是加号（+）。

非ASCII码通过UTF-8编码。

请求方法

方法	说明
GET	请求服务器返回指定资源
PUT	请求服务器更新指定资源
POST	请求服务器新增资源或执行特殊操作
DELETE	请求服务器删除指定资源，如删除对象等
HEAD	请求服务器资源头部
PATCH	请求服务器更新资源的部分内容

当资源不存在的时候，PATCH可能会去创建一个新的资源。

在"根据regionID查询用户资源"的URI部分，您可以看到其请求方法为“POST”，则其请求为：

POST https://ctecs-xxx.ctapi.ctyun.cn/v4/region/customerResources

请求消息头

名称	描述	是否必选	示例
Content-Type	消息体的类型	是	application/json
ctyun-eop-request-id	流水号（32位随机数）	是	0ffb9b07-d5a8-4e19-b3ce-12dfb9705a1d
Eop-Authorization	签名认证信息	是	4a4bdc57e06542199b5f98d4cd107be2 Headers=ctyun-eop-request-id;eop-date Signature=b2WEo4nh9ewn6SWOP0ii5g8L0z9CT5gprpDWnt
Eop-date	时间，15分钟有效期(实际传时间为北京东八区UTC+8时间。TZ仅为格式，非UTC时间。)	是	20221107T093029Z

Eop-Authorization签名认证信息：

步骤一：构造代签字符串sigture

sigture=需要进行签名的Header排序后的组合列表+"\n"+encode的query+"\n"+toHex(sha256(原封的body))

名称	描述
需要进行签名的Header排序后的组合列表	将ctyun-eop-request-id、eop-date以 “header_name:header_value”的形式、以“\n”作为每个header的结尾符、以英文字母表作为header_name的排序依据将它们拼接起来。注意：EOP强制要求ctyun-eop-request-id、eop-date必须进行签名。其他字段是否需要签名看自身需求。例子(假设你需要将ctyun-eop-request-id、eop-date、host都要签名)：ctyun-eop-request-id:123456789\neop-date:20210531T100101Z\n
encode的query	query以&作为拼接，key和value以=连接，值需要encode，Query参数全部都需要进行签名
toHex(sha256(原封的body))	传进来的body参数进行sha256摘要，对摘要出来的结果转十六进制

sigture示例1（假设query为空、需要进行签名的Header排序后的组合列表为“ctyun-eop-request-id:27cfe4dc-e640-45f6-92ca-492ca73e8680\neop-date:20220525T160752Z\n”、body参数做sha256摘要后转十六进制为e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855）：
ctyun-eop-request-id:27cfe4dc-e640-45f6-92ca-492ca73e8680\neop-date:20220525T160752Z\n\n
\ne3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

sigture示例2（假设query不为空、需要进行签名的Header排序后的组合列表为“ctyun-eop-request-id:27cfe4dc-e640-45f6-92ca-492ca73e8680\neop-date:20220525T160752Z\n”、body参数做sha256摘要后转十六进制为e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855）：

ctyun-eop-request-id:27cfe4dc-e640-45f6-92ca-492ca73e8680\neop-date:20220525T160930Z\n\naa=1&bb=2\ne3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

步骤二：构造动态秘钥kdate

1.使用eop-date作为数据，sk作为密钥，算出ktime。

2.使用ak作为数据，ktime作为密钥，算出kAk。

3.使用eop-date的年月日值作为数据，kAk作为密钥，算出kdate。

名称	描述
eop-date	yyyyMMdd'T'HHmmss'Z'(20211221T163614Z)(年月日T时分秒Z)(实际传时间为北京东八区UTC+8时间，TZ仅为格式，非UTC时间)
ktime	使用eop-date作为数据，sk作为密钥，算出ktime。ktime=hmacSHA256(eop-date,sk)
kAk	使用ak作为数据，ktime作为密钥，算出kAk。kAk=hmacSHA256(ak,ktime)
kdate	使用eop-date的年月日值作为数据，kAk作为密钥，算出kdate

步骤三：构造signature

使用kdate作为密钥、sigture作为数据，将其得到的结果进行base64编码得出signature

名称	描述
Signature	hmacSHA256(sigture,kdate)将上一步的结果进行base64加密得出signature

步骤四：构造Eop-Authorization

1.构造Headers。

2.得到Eop-Authorization。

Eop-Authorization:ak Headers=xxx Signature=xxx。

名称	描述
Headers	将需要进行签名的请求头字段以“header_name”的形式、以“;”作为间隔符、以英文字母表作为header_name的排序依据将它们拼接起来。例子(假设你需要将ctyun-eop-request-id、eop-date都要签名)：Headers=ctyun-eop-request-id;eop-date

名称	描述
Eop-Authorization	Eop-Authorization:ak Headers=xxx Signature=xxx。注意，ak、Headers、Signature之间以空格隔开。例如：Eop-Authorization:ak Headers=ctyun-eop-request-id;eop-date Signature=NlMHOhk5bVfZ9MwDSSJydcZjjENmDtpNYigJGVb。注意：如果你需要进行签名的Header不止默认的ctyun-eop-request-id和eop-date，那么你需要在http_client的请求头部中加上，并且Eop-Authorization中也需要增加

温馨提示：

API可进入 API调试 页面，进行在线调试。

API

2022-04-06

队列管理

查询队列

接口功能介绍

查询队列v3

接口约束

无

URI

GET /v3/queue/query

路径参数 无

Query参数

参数	是否必填	参数类型	说明	示例	下级对象
prodInstId	是	String	实例ID	6b10c8b962244f1f921d1a48d0f15cca	
vhost	否	String	vhost名称	vhost1	
name	否	String	队列名称		
pageNum	否	Integer	分页的页序号	1	
pageSize	否	Integer	分页的大小	100	

请求参数

请求头header参数

参数	是否必填	参数类型	说明	示例	下级对象
regionId	是	String	资源池id	6b10c8b9 62244f1f921d1a48d0f15cca	

请求体body参数 无

响应参数

参数	参数类型	说明	示例	下级对象
returnObj	Object	返回对象		returnObj
message	String	描述状态	success	
statusCode	String	接口系统层面状态码。成功：800，失败：900	800	
error	String	错误码，描述错误信息，只有失败才显示	AMQP_ AMQP_	

表 returnObj

参数	参数类型	说明	示例	下级对象
data	Object	返回数据		data

表 data

参数	参数类型	说明	示例	下级对象
filtered_count	Integer	按name过滤后的总队列数	2	
item_count	Integer	当前页面上的队列数量	2	
items	Array of Objects	队列详细信息		items
page	Integer	当前页数	1	
page_count	Integer	总页数	1	
page_size	Integer	分页设置的每个页面的最多队列数	100	
total_count	Integer	总队列数	2	

表 items

参数	参数类型	说明	示例	下级对象
vhost	String	虚拟机名称	/	
durable	Boolean	是否持久化，默认持久化	true	

参数	参数类型	说明	示例	下级对象
name	String	队列名称	qu1	
auto_delete	Boolean	是否自动删除	false	
consumers	Integer	消费者数量	0	

枚举参数

无

请求示例

请求url

```
#
```

请求头header

```
#
```

请求体body

```
1
```

响应示例

```
{
    "returnObj":{
        "data":{
            "filtered_count":2,
            "item_count":2,
            "items":[
                {
                    "vhost":"/",
                    "durable":true,
                    "name":"qu1",
                    "auto_delete":false,
                    "consumers":0
                },
                {
                    "vhost":"/",
                    "durable":true,
                    "name":"test2",
                    "auto_delete":false,
                    "consumers":0
                }
            ],
            "page":1,
            "page_count":1,
            "page_size":100,
            "total_count":2
        }
    },
    "message":"success",
    "statusCode":"800"
}
```

状态码

请参考 状态码

错误码

请参考 错误码

SDK参考

SDK概述

本文介绍了分布式消息服务RabbitMQ提供的SDK语言版本。

SDK列表

下表提供了分布式消息服务RabbitMQ支持的SDK列表。

编程语言	参考文档
Java	开发指南-Java
Python	开发指南-Python