ALIBABA CLOUD

阿里云

消息队列 MQ

消息队列RocketMQ版公共云合集

文档版本：20220512

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.功能与特性

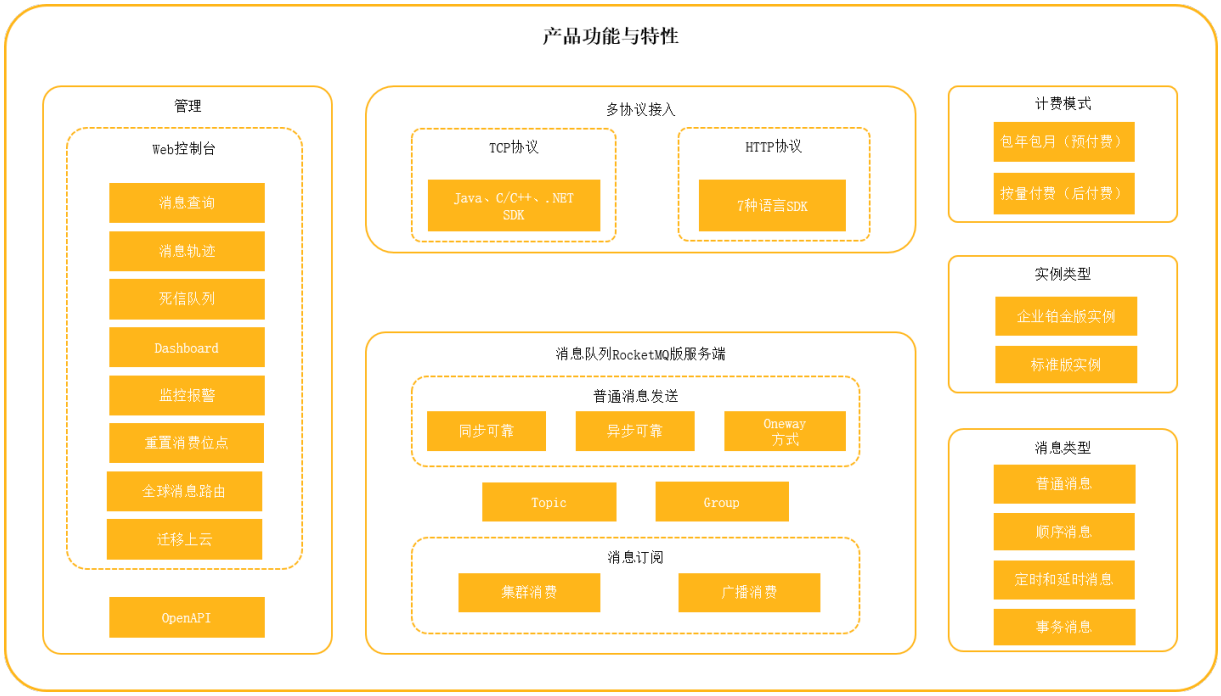
1.1. 功能与特性概述

本文列举
消息队列RocketMQ版
所支持的所有功能与特性。

概览

消息队列RocketMQ版
在阿里云多个地域（Region）提供了高可用消息云服务。单个地域内采用多机房部署，可用性极高，即使整个机房都不可用，仍然可以为应用提供消息发布服务。

消息队列RocketMQ版
提供TCP和HTTP协议的多语言接入方式，方便不同编程语言开发的应用快速接入
消息队列RocketMQ版
消息云服务。您可以将应用部署在阿里云ECS、企业自建云，或者嵌入到移动端、物联网设备中与
消息队列RocketMQ版
建立连接进行消息收发；同时，本地开发者也可以通过公网接入
消息队列RocketMQ版
服务进行消息收发。



多协议接入

- TCP协议：区别于HTTP简单的接入方式，提供更为专业、可靠、稳定的TCP协议的SDK接入服务。支持的语言包括Java、C/C++ 以及.NET。
- HTTP协议：采用RESTful风格，方便易用，快速接入，跨网络能力强。支持Java、C++、.NET、Go、Python、Node.js和PHP七种语言客户端。

管理工具

- Web控制台：支持通过控制台完成资源管理、消息查询、消息轨迹查询、监控报警管理等操作。具体操

作，请参见控制台使用指南。

- OpenAPI：提供开放的API便于将
  消息队列RocketMQ版
  管理工具集成到自己的控制台。
  消息队列RocketMQ版
  的API的更多信息，请参见OpenAPI参考。

消息类型列表

消息类型对比

消息类型	是否支持可靠同步发送	是否支持可靠异步发送	是否支持Oneway发送	是否支持多线程发送	性能
普通消息	是	是	是	是	最高
事务消息					
定时和延时消息					
分区顺序消息	是	否	否	否	高
全局顺序消息					一般

消息特性

- 消息重试：在消费者返回消息重试的响应后，
  消息队列RocketMQ版
  会按照相应的重试规则进行消息重投。
- 至少投递一次（At-least-once）：
  消息队列RocketMQ版
  保证消息成功被消费一次。
  消息队列RocketMQ版
  的分布式特点和瞬变的网络条件，或者用户应用重启发布的情况下，可能导致消费者收到重复的消息。开发人员应将其应用程序设计为多次处理一条消息不会产生任何错误或不一致性。消息幂等最佳实践请参见消费幂等。
- 消息过滤：
  消息队列RocketMQ版
  支持设置消息属性给消息进行分类，
  消息队列RocketMQ版
  服务端会根据您订阅消息时设置的过滤条件对消息进行过滤，您将只消费到需要关注的消息。

消息功能

- 消息查询：
  消息队列RocketMQ版
  提供了三种消息查询的方式，分别是按Message ID、Message Key以及Topic查询。
- 查询消息轨迹：通过消息轨迹，能清晰定位消息从生产者发出，经由
  消息队列RocketMQ版
  服务端，投递给消息消费者的完整链路，方便定位排查问题。
- 集群消费和广播消费：当使用集群消费模式时，
  消息队列RocketMQ版

认为任意一条消息只需要被消费者集群内的任意一个消费者处理即可；当使用广播消费模式时，
消息队列RocketMQ版
会将每条消息推送给消费者集群内所有注册过的消费者，保证消息至少被每台机器消费一次。

- 重置消费位点：根据时间或位点重置消费进度，允许用户进行消息回溯或者跳过堆积的消息从最新位点消费。
- 死信队列：将无法正常消费的消息储存到特殊的死信队列供后续处理。
- Dashboard：提供丰富、全面、多维度的统计指标，您可以分别从实例、Topic和Group维度查看消息生产指标、消息消费指标以及消息堆积等相关指标。
- 监控报警：您可使用
  消息队列RocketMQ版
  提供的监控报警功能，监控某Group ID订阅的某Topic的消息消费状态并接收报警短信，帮助您实时掌握消息消费状态，以便及时处理消费异常。
- 开源RocketMQ迁移上云：支持将开源RocketMQ的元数据迁移到
  消息队列RocketMQ版
  实例上，帮助您完成消息业务的无缝上云。

高级特性

- 全球消息路由：用于全球不同地域之间的消息同步，保证地域之间的数据一致性。

1.2. 普通消息

普通消息是指
消息队列RocketMQ版
中无特性的消息，区别于有特性的定时和延时消息、顺序消息和事务消息。

> **说明** 您在调用SDK收发消息时需注意，
> 消息队列RocketMQ版
> 提供的四种消息类型所对应的Topic不能混用，例如，您创建的普通消息的Topic只能用于收发普通消息，不能用于收发其他类型的消息；同理，事务消息的Topic也只能收发事务消息，不能用于收发其他类型的消息，以此类推。

- TCP SDK收发普通消息的示例代码
  - Java
    - 发送普通消息（三种方式）
    - 发送消息（多线程）
    - 订阅消息
  - C或C++
    - 收发普通消息
    - 订阅消息
  - .NET
    - 收发普通消息
    - 订阅消息
- HTTP SDK收发普通消息的示例代码
  消息队列RocketMQ版
  支持RESTful风格的HTTP协议通信，并提供了以下7种语言的SDK：

- Go SDK版本说明
- Python SDK版本说明
- Node.js SDK版本说明
- PHP SDK版本说明
- Java SDK版本说明
- C++ SDK版本说明
- C# SDK版本说明

更多消息收发示例代码，请参见
消息队列RocketMQ版
HTTP SDK示例代码查看消息收发的示例代码。
HTTP SDK的更多信息请参见HTTP协议。

1.3. 定时和延时消息

本文主要介绍
消息队列RocketMQ版
的定时消息和延时消息的概念、适用场景以及使用过程中的注意事项。

概念介绍

- 定时消息：Producer将消息发送到
  消息队列RocketMQ版
  服务端，但并不期望立马投递这条消息，而是推迟到在当前时间点之后的某一个时间投递到Consumer进行消费，该消息即定时消息。
- 延时消息：Producer将消息发送到
  消息队列RocketMQ版
  服务端，但并不期望立马投递这条消息，而是延迟一定时间后才投递到Consumer进行消费，该消息即延时消息。

定时消息与延时消息在代码配置上存在一些差异，但是最终达到的效果相同：消息在发送到

消息队列RocketMQ版
服务端后并不会立马投递，而是根据消息中的属性延迟固定时间后才投递给消费者。

适用场景

定时消息和延时消息适用于以下一些场景：

- 消息生产和消费有时间窗口要求，例如在电商交易中超时未支付关闭订单的场景，在订单创建时会发送一条延时消息。这条消息将会在30分钟以后投递给消费者，消费者收到此消息后需要判断对应的订单是否已完成支付。如支付未完成，则关闭订单。如已完成支付则忽略。
- 通过消息触发一些定时任务，例如在某一固定时间点向用户发送提醒消息。

使用方式

定时消息和延时消息的使用在代码编写上存在略微的区别：

- 发送定时消息需要明确指定消息发送时间点之后的某一时间点作为消息投递的时间点。
- 发送延时消息时需要设定一个延时时间长度，消息将从当前发送时间点开始延迟固定时间之后才开始投递。

注意事项

- 定时消息的精度会有1s~2s的延迟误差。

- 定时和延时消息的 `msg.setStartDeliverTime` 参数需要设置成当前时间戳之后的某个时刻（单位毫秒）。如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 定时和延时消息的 `msg.setStartDeliverTime` 参数可设置40天内的任何时刻（单位毫秒），超过40天消息发送将失败。
- `StartDeliverTime` 是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时和延时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差。
- 设置定时和延时消息的投递时间后，依然受3天的消息保存时长限制。
  例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第8天被删除。

TCP协议示例代码

收发定时消息和延时消息的示例代码，请参见以下文档：

- Java
  - 收发定时消息
  - 收发延时消息
- C++
  - 收发定时消息
- .NET
  - 收发定时消息

HTTP协议示例代码

收发定时消息和延时消息的示例代码，请参见以下文档：

- 版本说明
- 版本说明
- 版本说明
- 版本说明
- 版本说明
- 版本说明
- 版本说明

1.4. 顺序消息2.0

顺序消息常用于金融证券、电商业务等对消息指令顺序有严格要求的场景。相对于顺序消息1.0版本，2.0版本具有更高的并发性、可用性和抗热点能力。本文介绍
消息队列RocketMQ版
顺序消息2.0的基本概念、实现原理、功能优势以及使用过程中的注意事项。

什么是顺序消息

顺序消息是

消息队列RocketMQ版
提供的一种对消息发送和消费顺序有严格要求的消息。对于一个指定的Topic，消息严格按照先进先出（FIFO）的原则进行消息发布和消费，即先发布的消息先消费，后发布的消息后消费。
顺序消息分为分区顺序消息和全局顺序消息。

分区顺序消息

对于指定的一个Topic，所有消息根据Sharding Key进行区块分区，同一个分区内的消息按照严格的先进先出（FIFO）原则进行发布和消费。同一分区内的消息保证顺序，不同分区之间的消息顺序不做要求。

- **基本概念**
  - Sharding Key：顺序消息中用来区分Topic中不同分区的关键字段，和普通消息的Key是完全不同的概念。
    消息队列RocketMQ版
    会将设置了相同Sharding Key的消息路由到同一个分区下，同一个分区内的消息将按照消息发布顺序进行消费。
  - 分区：即Topic Partition，每个Topic包含一个或多个分区，Topic中的消息会分布在这些不同的分区中。本文中的逻辑分区指的就是Topic的分区，更多信息，请参见名词解释。
  - 物理分区：区别于逻辑分区，消息实际存储的单元，每个物理分区都会分配到某一台机器指定节点上。
- **适用场景**
  适用于性能要求高，以Sharding Key作为分区字段，在同一个区块中严格地按照先进先出（FIFO）原则进行消息发布和消费的场景。
- **示例**
  - 用户注册需要发送验证码，以用户ID作为Sharding Key，那么同一个用户发送的消息都会按照发布的先后顺序来消费。
  - 电商的订单创建，以订单ID作为Sharding Key，那么同一个订单相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息都会按照发布的先后顺序来消费。

阿里巴巴集团内部电商系统均使用分区顺序消息，既保证业务的顺序，同时又能保证业务的高性能。

全局顺序消息

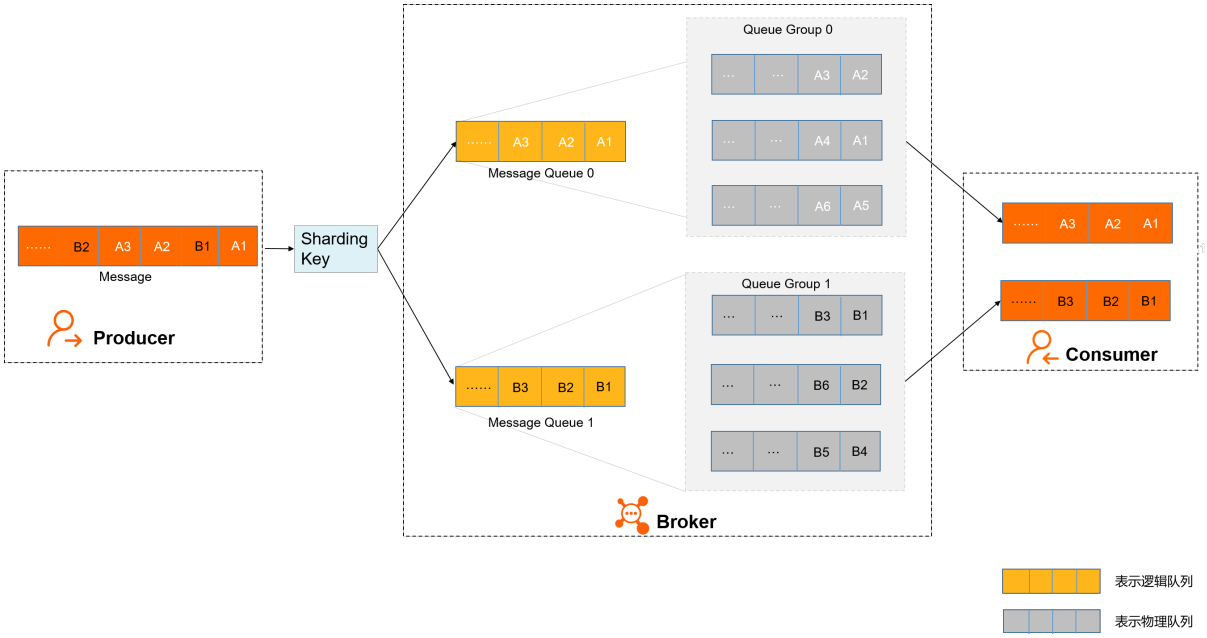
对于指定的一个Topic，所有消息按照严格的先入先出（FIFO）的顺序来发布和消费。

- **适用场景**
  适用于性能要求不高，所有的消息严格按照FIFO原则来发布和消费的场景。
- **示例**
  在证券处理中，以人民币兑换美元为Topic，在价格相同的情况下，先出价者优先处理，则可以按照FIFO的方式发布和消费全局顺序消息。

> **说明** 全局顺序消息实际上是一种特殊的分区顺序消息，即Topic中只有一个分区，因此全局顺序和分区顺序的实现原理相同。因为分区顺序消息有多个分区，所以分区顺序消息比全局顺序消息的并发度和性能更高。

如何保证消息的顺序

全局顺序消息和分区顺序消息原理一样，下文以分区顺序消息为例介绍在
消息队列RocketMQ版
中如何保证消息的顺序。



在

消息队列RocketMQ版
中，消息的顺序需要由以下三个阶段保证：

- **消息发送**
  如上图所示，A1、B1、A2、A3、B2、B3是订单A和订单B的消息产生的顺序，业务上要求同一订单的消息保持顺序，例如订单A的消息发送和消费都按照A1、A2、A3的顺序。如果是普通消息，订单A的消息可能会被轮询发送到不同的队列中，不同队列的消息将无法保持顺序，而顺序消息发送时
  消息队列RocketMQ版
  支持将Sharding Key相同（例如同一订单号）的消息序路由到一个分区中。
- **消息存储**
  顺序消息2.0中，Topic中的每个逻辑分区可以对应多个物理分区，当消息按照顺序发送到Topic中的逻辑分区时，每个分区的消息将按照负载均匀的存储到对应的多个物理分区中，在物理分区中消息的存储可以不用保持顺序，但
  消息队列RocketMQ版
  中会记录消息在逻辑分区和物理分区中的映射关系及存储位置。
- **消息消费**
  即使同一逻辑分区的消息被存储在不同的物理分区中且没有保持消息的顺序，但是基于顺序消息2.0的保序协议，
  消息队列RocketMQ版
  服务端在投递消息时，最终还是会按照消息在逻辑队列中存储的顺序投递给Consumer，Consumer消费消息时，同一Sharding Key的消息使用单线程消费，保证消息消费顺序和存储顺序一致，最终实现消费顺序和发布顺序的一致。

顺序消息2.0功能优势

顺序消息2.0提出了一种创新的分布式消息有序协议，一个逻辑分区可以对应多个物理分区。和顺序消息1.0相比，顺序消息2.0具有以下优势：

- **无热点**
  一个逻辑分区可以对应多个物理分区，消除了因为物理分区性能瓶颈导致的逻辑分区热点问题。即使是全局顺序消息，也不会出现单个逻辑分区的热点问题。
- **高并发无限扩展**

在保持顺序的前提下，每个逻辑分区对应的物理分区可以无限水平扩展，极大提高消息处理的并发性，并且业务侧无感知，也不需额外的业务改造成本。

- **故障秒级恢复**
  单个物理分区故障，其他物理分区可继续处理消息，达到秒级切换，业务基本无感知。

注意事项

使用顺序消息2.0时，请注意以下几点：

- 如需使用顺序消息2.0特性，请确保您的
  消息队列RocketMQ版
  实例是企业铂金版。
- 仅TCP Java 1.8.7.1.Final及以上版本的SDK支持顺序消息2.0。
- 同一个Group ID只对应一种类型的Topic，即不同时用于顺序消息和无序消息的收发。
- 对于全局顺序消息，建议消息不要有阻塞。同时运行多个实例，是为了防止工作实例意外退出而导致业务中断。当工作实例退出时，其他实例可以立即接手工作，不会导致业务中断，实际工作的只会有一个实例。
- 消息队列RocketMQ版
  服务端判定消息产生的顺序性是参照单一生产者、单一线程并发下消息发送的时序。如果发送方有多个生产者或者有多个线程并发发送消息，则此时只能以到达
  消息队列RocketMQ版
  服务端的时序作为消息顺序的依据，和业务侧的发送顺序未必一致。

TCP SDK示例代码

TCP协议下的示例代码，请参见Java SDK收发顺序消息。

存量Topic升级说明

如果您使用的是顺序消息1.0，且需升级至顺序消息2.0，请提交工单交由产品方做迁移支持。

顺序消息常见问题

- 同一条消息是否可以既是顺序消息，又是定时消息和事务消息？
  不可以。顺序消息、定时消息、事务消息是不同的消息类型，三者是互斥关系，不能叠加在一起使用。
- 顺序消息支持哪些地域？
  支持
  消息队列RocketMQ版
  所有公共云地域和金融云地域。
- 为什么全局顺序消息性能一般？
  全局顺序消息是严格按照FIFO的消息阻塞原则，即上一条消息没有被成功消费，那么下一条消息会一直被存储到Topic队列中。如果想提高全局顺序消息的TPS，可以升级实例配置，同时消息客户端应用尽量减少处理本地业务逻辑的耗时。
- 顺序消息支持哪种消息发送方式？
  顺序消息只支持可靠同步发送方式，不支持异步发送方式，否则将无法严格保证顺序。
- 顺序消息是否支持集群消费和广播消费？
  顺序消息暂时仅支持集群消费模式，不支持广播消费模式。

1.5. 顺序消息1.0

顺序消息可以保证消息的消费顺序和发送的顺序一致，即先发送的先消费，后发送的后消费，常用于金融证券、电商业务等对消息指令顺序有严格要求的场景。本文介绍
消息队列RocketMQ版

顺序消息1.0的概念、适用场景、实现原理以及使用过程中的注意事项。

什么是顺序消息

顺序消息是

消息队列RocketMQ版
提供的一种对消息发送和消费顺序有严格要求的消息。对于一个指定的Topic，消息严格按照先进先出（FIFO）的原则进行消息发布和消费，即先发布的消息先消费，后发布的消息后消费。
顺序消息分为分区顺序消息和全局顺序消息。

分区顺序消息

对于指定的一个Topic，所有消息根据Sharding Key进行区块分区，同一个分区内的消息按照严格的先进先出（FIFO）原则进行发布和消费。同一分区内的消息保证顺序，不同分区之间的消息顺序不做要求。

- **基本概念**
  - Sharding Key：顺序消息中用来区分Topic中不同分区的关键字段，和普通消息的Key是完全不同的概念。
    消息队列RocketMQ版
    会将设置了相同Sharding Key的消息路由到同一个分区下，同一个分区内的消息将按照消息发布顺序进行消费。
  - 分区：即Topic Partition，每个Topic包含一个或多个分区，Topic中的消息会分布在这些不同的分区中。本文中的逻辑分区指的就是Topic的分区，更多信息，请参见名词解释。
  - 物理分区：区别于逻辑分区，消息实际存储的单元，每个物理分区都会分配到某一台机器指定节点上。
- **适用场景**
  适用于性能要求高，以Sharding Key作为分区字段，在同一个区块中严格地按照先进先出（FIFO）原则进行消息发布和消费的场景。
- **示例**
  - 用户注册需要发送验证码，以用户ID作为Sharding Key，那么同一个用户发送的消息都会按照发布的先后顺序来消费。
  - 电商的订单创建，以订单ID作为Sharding Key，那么同一个订单相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息都会按照发布的先后顺序来消费。

阿里巴巴集团内部电商系统均使用分区顺序消息，既保证业务的顺序，同时又能保证业务的高性能。

全局顺序消息

对于指定的一个Topic，所有消息按照严格的先入先出（FIFO）的顺序来发布和消费。

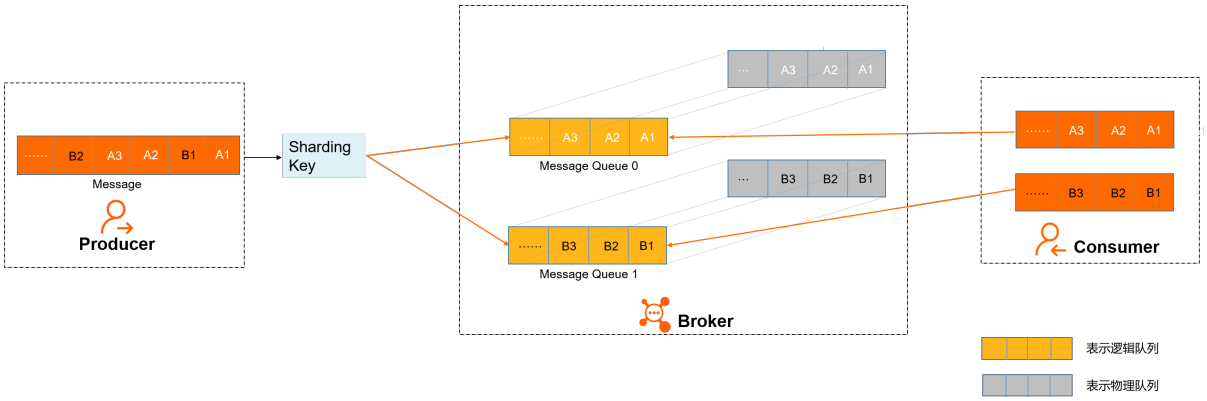
- **适用场景**
  适用于性能要求不高，所有的消息严格按照FIFO原则来发布和消费的场景。
- **示例**
  在证券处理中，以人民币兑换美元为Topic，在价格相同的情况下，先出价者优先处理，则可以按照FIFO的方式发布和消费全局顺序消息。

> **说明** 全局顺序消息实际上是一种特殊的分区顺序消息，即Topic中只有一个分区，因此全局顺序和分区顺序的实现原理相同。因为分区顺序消息有多个分区，所以分区顺序消息比全局顺序消息的并发度和性能更高。

如何实现顺序消息

全局顺序消息和分区顺序消息原理一样，下文以分区顺序消息为例介绍在
消息队列RocketMQ版

中如何保证消息收发的顺序。



在

消息队列RocketMQ版
中，消息的顺序需要由以下三个阶段保证：

- **消息发送**
  如上图所示，A1、B1、A2、A3、B2、B3是订单A和订单B的消息产生的顺序，业务上要求同一订单的消息保持顺序，例如订单A的消息发送和消费都按照A1、A2、A3的顺序。如果是普通消息，订单A的消息可能会被轮询发送到不同的队列中，不同队列的消息将无法保持顺序，而顺序消息发送时
  消息队列RocketMQ版
  支持将Sharding Key相同（例如同一订单号）的消息序路由到一个分区中。
  消息队列RocketMQ版
  服务端判定消息产生的顺序性是参照同一生产者发送消息的时序。不同生产者、不同线程并发产生的消息，
  消息队列RocketMQ版
  服务端无法判定消息的先后顺序。
- **消息存储**
  如上图所示，顺序消息1.0中，Topic中的每个分区（逻辑分区）对应一个物理分区，当消息按照顺序发送到Topic中的逻辑分区时，每个分区的消息将按照同样的顺序存储到对应的物理分区中。
- **消息消费**
  消息队列RocketMQ版
  按照存储的顺序将消息投递给Consumer，Consumer收到消息后也不对消息顺序做任何处理，按照接收到的顺序进行消费。
  Consumer消费消息时，同一Sharding Key的消息使用单线程消费，保证消息消费顺序和存储顺序一致，最终实现消费顺序和发布顺序的一致。

顺序消息1.0中，一个逻辑分区只能对应一个物理分区，且分区数一旦确定无法更改。如果业务出现增长，单物理节点的性能会出现瓶颈，如果强行扩容会影响消息路由策略和顺序，拆分业务也会造成额外的改造成本。因此顺序消息1.0适用于业务比较稳定，消息量波动比较小，后期不需要进行扩容的场景。

如果考虑业务增长可能需要扩容的情况，建议可以使用顺序消息2.0，支持水平无限扩容并对业务透明，避免了分区顺序的热点问题，且单个物理节点故障时支持秒级切换，业务侧无感知。更多信息，请参见顺序消息2.0。

注意事项

使用顺序消息1.0时，请注意以下几点：

- 同一个Group ID只对应一种类型的Topic，即不同时用于顺序消息和无序消息的收发。
- 对于全局顺序消息，建议消息不要有阻塞。同时运行多个实例，是为了防止工作实例意外退出而导致业务中断。当工作实例退出时，其他实例可以立即接手工作，不会导致业务中断，实际工作的只会有一个实

例。

- 消息队列RocketMQ版
  服务端判定消息产生的顺序性是参照单一生产者、单一线程并发下消息发送的时序。如果发送方有多个生产者或者有多个线程并发发送消息，则此时只能以到达
  消息队列RocketMQ版
  服务端的时序作为消息顺序的依据，和业务侧的发送顺序未必一致。

顺序消息常见问题

- 同一条消息是否可以既是顺序消息，又是定时消息和事务消息？
  不可以。顺序消息、定时消息、事务消息是不同的消息类型，三者是互斥关系，不能叠加在一起使用。
- 顺序消息支持哪些地域？
  支持
  消息队列RocketMQ版
  所有公共云地域和金融云地域。
- 为什么全局顺序消息性能一般？
  全局顺序消息是严格按照FIFO的消息阻塞原则，即上一条消息没有被成功消费，那么下一条消息会一直被存储到Topic队列中。如果想提高全局顺序消息的TPS，可以升级实例配置，同时消息客户端应用尽量减少处理本地业务逻辑的耗时。
- 顺序消息支持哪种消息发送方式？
  顺序消息只支持可靠同步发送方式，不支持异步发送方式，否则将无法严格保证顺序。
- 顺序消息是否支持集群消费和广播消费？
  顺序消息暂时仅支持集群消费模式，不支持广播消费模式。

TCP SDK示例代码

TCP协议下的示例代码请参见以下文档：

- Java：收发顺序消息
- C/C++：收发顺序消息
- .NET：收发顺序消息

HTTP SDK示例代码

HTTP协议下的示例代码请参见以下文档：

- Java：收发顺序消息
- Go：收发顺序消息
- Python：收发顺序消息
- Node.js：收发顺序消息
- PHP：收发顺序消息
- C#：收发顺序消息
- C++：收发顺序消息

1.6. 事务消息

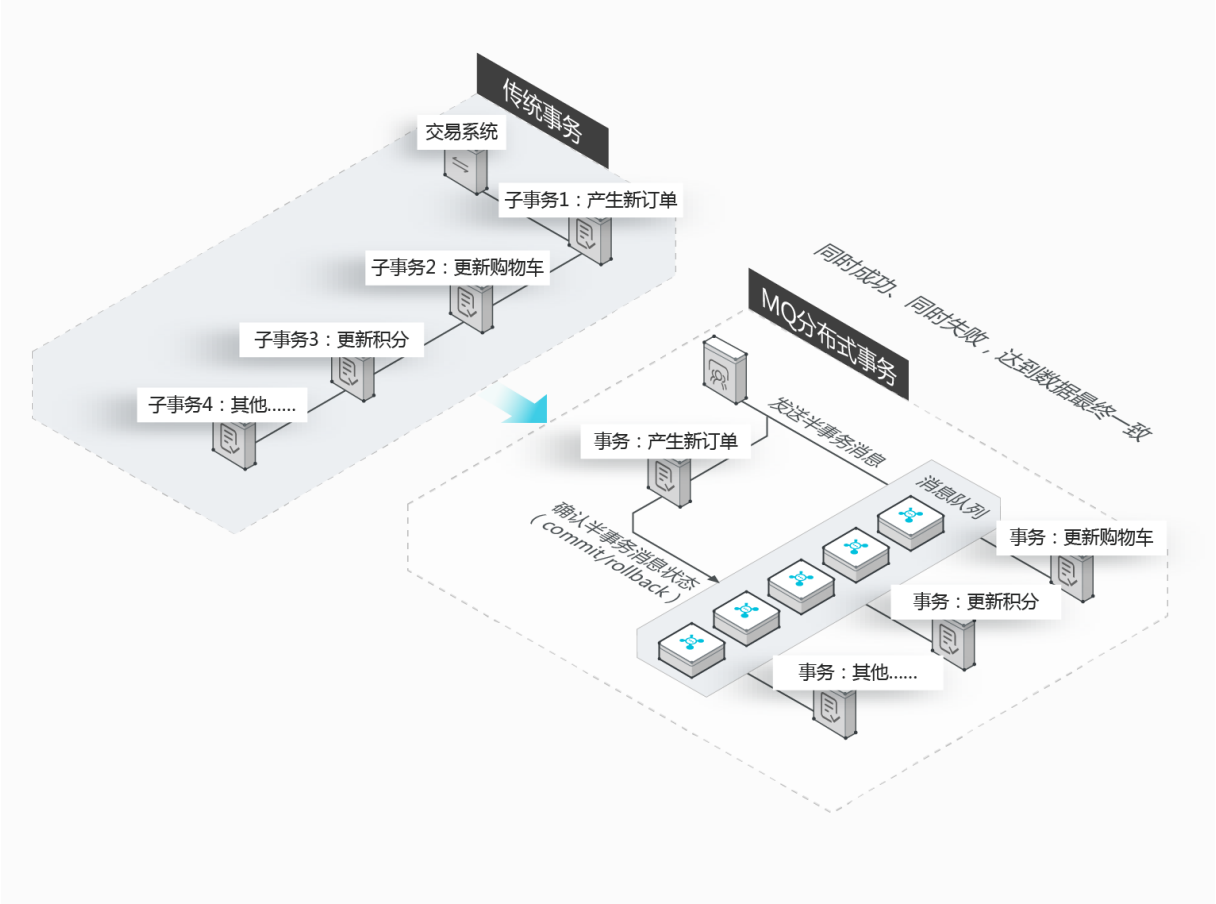
消息队列RocketMQ版
提供的分布式事务消息适用于所有对数据最终一致性有强需求的场景。本文介绍
消息队列RocketMQ版
事务消息的概念、优势、典型场景、交互流程、使用规则以及示例代码。

概念介绍

- 事务消息：
  消息队列RocketMQ版
  提供类似XA或Open XA的分布式事务功能，通过
  消息队列RocketMQ版
  事务消息能达到分布式事务的最终一致。
- 半事务消息：暂不能投递的消息，生产者已经成功地将消息发送到了
  消息队列RocketMQ版
  服务端，但是
  消息队列RocketMQ版
  服务端未收到生产者对该消息的二次确认，此时该消息被标记成“暂不能投递”状态，处于该种状态下的消息即半事务消息。
- 消息回查：由于网络闪断、生产者应用重启等原因，导致某条事务消息的二次确认丢失，
  消息队列RocketMQ版
  服务端通过扫描发现某条消息长期处于“半事务消息”时，需要主动向消息生产者询问该消息的最终状态（Commit或是Rollback），该询问过程即消息回查。

分布式事务消息的优势

消息队列RocketMQ版
分布式事务消息不仅可以实现应用之间的解耦，又能保证数据的最终一致性。同时，传统的大事务可以被拆分为小事务，不仅能提升效率，还不会因为某一个关联应用的不可用导致整体回滚，从而最大限度保证核心系统的可用性。在极端情况下，如果关联的某一个应用始终无法处理成功，也只需对当前应用进行补偿或数据订正处理，而无需对整体业务进行回滚。



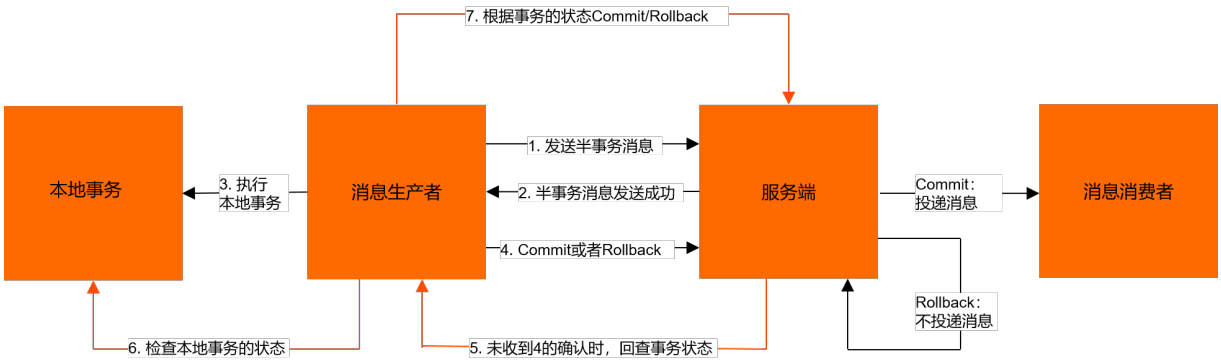
典型场景

在淘宝购物车下单时，涉及到购物车系统和交易系统，这两个系统之间的数据最终一致性可以通过分布式事务消息的异步处理实现。在这种场景下，交易系统是最为核心的系统，需要最大限度地保证下单成功。而购物车系统只需要订阅

消息队列RocketMQ版
的交易订单消息，做相应的业务处理，即可保证最终的数据一致性。

交互流程

事务消息交互流程如下图所示。



事务消息发送步骤如下：

1. 生产者将半事务消息发送至
   消息队列RocketMQ版
   服务端。
2. 消息队列RocketMQ版
   服务端将消息持久化成功之后，向生产者返回Ack确认消息已经发送成功，此时消息为半事务消息。
3. 生产者开始执行本地事务逻辑。
4. 生产者根据本地事务执行结果向服务端提交二次确认结果（Commit或是Rollback），服务端收到确认结果后处理逻辑如下：
   - 二次确认结果为Commit：服务端将半事务消息标记为可投递，并投递给消费者。
   - 二次确认结果为Rollback：服务端将回滚事务，不会将半事务消息投递给消费者。
5. 在断网或者是生产者应用重启的特殊情况下，若服务端未收到发送者提交的二次确认结果，或服务端收到的二次确认结果为Unknown未知状态，经过固定时间后，服务端将对消息生产者即生产者集群中任一生产者实例发起消息回查。

事务消息回查步骤如下：

1. 生产者收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
2. 生产者根据检查得到的本地事务的最终状态再次提交二次确认，服务端仍按照步骤4对半事务消息进行处理。

使用规则

- 事务消息发送完成本地事务后，可在 `execute` 方法中返回以下三种状态：
  - `TransactionStatus.CommitTransaction`：提交事务，允许消费者消费该消息。
  - `TransactionStatus.RollbackTransaction`：回滚事务，消息将被丢弃不允许消费。
  - `TransactionStatus.Unknow`：暂时无法判断状态，等待固定时间以后
    消息队列RocketMQ版
    服务端根据回查规则向生产者进行消息回查。

- 通过 `ONSFactory.createTransactionProducer` 创建事务消息的Producer时必须指定 `LocalTransactionChecker` 的实现类，处理异常情况下事务消息的回查。
- 回查规则：本地事务执行完成后，若
  消息队列RocketMQ版
  服务端收到的本地事务返回状态为 `TransactionStatus.Unknow` ，或生产者应用退出导致本地事务未提交任何状态。则
  消息队列RocketMQ版
  服务端会向消息生产者发起事务回查，第一次回查后仍未获取到事务状态，则之后每隔一段时间会再次回查。
  - 回查间隔时间：系统默认每隔30秒发起一次定时任务，对未提交的半事务消息进行回查，共持续12小时。
  - 第一次消息回查最快时间：该参数支持自定义设置。若指定消息未达到设置的最快回查时间前，系统默认每隔30秒一次的回查任务不会检查该消息。
    以Java为例，以下设置表示：第一次回查的最快时间为60秒。

    ```
    Message message = new Message();
    message.putUserProperties(PropertyKeyConst.CheckImmunityTimeInSeconds,"60");
    ```

    > **说明**
    > 因为系统默认的回查间隔，第一次消息回查的实际时间会向后有0秒~30秒的浮动。
    > 例如：指定消息的第一次消息最快回查时间设置为60秒，系统在第58秒时达到定时的回查时间，但设置的60秒未到，所以该消息不在本次回查范围内。等待间隔30秒后，下一次的系统回查时间在第88秒，该消息才符合条件进行第一次回查，距设置的最快回查时间延后了28秒。
- 事务消息的Group ID不能与其他类型消息的Group ID共用。与其他类型的消息不同，事务消息有回查机制，回查时
  消息队列RocketMQ版
  服务端会根据Group ID去查询生产者客户端。

生产消息规则

消费消息规则

示例代码

收发事务消息的示例代码如下：

- TCP协议
  - Java SDK收发事务消息
  - C/C++ SDK收发事务消息
  - .NET SDK收发事务消息
- HTTP协议（多语言）
  - Go SDK收发事务消息
  - Python SDK收发事务消息
  - Node.js SDK收发事务消息
  - PHP SDK收发事务消息
  - Java SDK收发事务消息
  - C++ SDK收发事务消息

- C# SDK收发事务消息

1.7. 消息重试

消息队列RocketMQ版
支持消息重试功能，即Consumer消费某条消息失败后，
消息队列RocketMQ版
会根据消息重试机制重新投递消息。TCP协议和HTTP协议的重试机制有所差异，本文介绍
消息队列RocketMQ版
分别在HTTP协议和TCP协议下的消息重试机制和配置方式。

注意事项

- 一条消息无论重试多少次，这些重试消息的Message ID都不会改变。
- 消息重试只针对集群消费模式生效；广播消费模式不提供失败重试特性，即消费失败后，失败消息不再重试，继续消费新的消息。

消息重试机制

消息队列RocketMQ版
消息收发过程中，若Consumer消费某条消息失败，则
消息队列RocketMQ版
会在重试间隔时间后，将消息重新投递给Consumer消费，若达到最大重试次数后消息还没有成功被消费，则消息将被投递至死信队列。

- **重试间隔**：消息消费失败后再次被
  消息队列RocketMQ版
  投递给Consumer消费的间隔时间。
- **最大重试次数**：消息消费失败后，可被
  消息队列RocketMQ版
  重复投递的最大次数。

消息队列RocketMQ版
中，顺序消息和无序消息（包括普通消息、延时消息、定时消息和事务消息）的重试机制如下：

协议	消息类型	重试间隔	最大重试次数	配置方式
	顺序消息	间隔时间可通过自定义参数suspendTimeMillis取值进行配置。参数取值范围：10～30000，单位：毫秒，默认值：1000毫秒，即1秒。	最大重试次数可通过自定义参数MaxReconsumeTimes取值进行配置。该参数取值无最大限制。若未设置参数值，默认最大重试次数为Integer.MAX。	请参见配置方式

TCP协议 协议	消息类型	重试间隔	最大重试次数	请参见配置方式（TCP协议）。 配置方式
	无序消息	间隔时间根据重试次数阶梯变化，取值范围：1秒～2小时。不支持自定义配置。 • 若最大重试次数小于等于16次，则每次重试的间隔时间会阶梯变化，具体时间，请参见无序消息重试间隔。 • 若最大重试次数大于16次，则超过16次的间隔时间均为2小时。	最大重试次数可通过自定义参数MaxReconsumeTimes取值进行配置。默认值为16次，该参数取值无最大限制，建议使用默认值。	
HTTP协议	顺序消息	1分钟	288次	系统预设，不支持修改。
	无序消息	5分钟	288次	系统预设，不支持修改。

TCP协议无序消息重试间隔

第几次重试	与上次重试的间隔时间	第几次重试	与上次重试的间隔时间
1	10秒	9	7分钟
2	30秒	10	8分钟
3	1分钟	11	9分钟
4	2分钟	12	10分钟
5	3分钟	13	20分钟
6	4分钟	14	30分钟
7	5分钟	15	1小时
8	6分钟	16	2小时

配置方式（TCP协议）

> **注意** 以下配置方式仅适用于TCP协议，HTTP协议不涉及。

- **消息投递失败后需要重试**
  集群消费模式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置（三种方式任选一种）：

- 方式1：返回Action.ReconsumeLater（推荐）
- 方式2：返回Null
- 方式3：抛出异常

示例代码

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        //消息处理逻辑抛出异常，消息将重试。
        doConsumeMessage(message);
        //方式1：返回Action.ReconsumeLater，消息将重试。
        return Action.ReconsumeLater;
        //方式2：返回null，消息将重试。
        return null;
        //方式3：直接抛出异常，消息将重试。
        throw new RuntimeException("Consumer Message exception");
    }
}
```

- **消费投递失败后无需重试**
  集群消费模式下，消息失败后期望消息不重试，需要捕获消费逻辑中可能抛出的异常，最终返回Action.CommitMessage，此后这条消息将不会再重试。
  示例代码

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        try {
            doConsumeMessage(message);
        } catch (Throwable e) {
            //捕获消费逻辑中的所有异常，并返回Action.CommitMessage;
            return Action.CommitMessage;
        }
        //消息处理正常，直接返回Action.CommitMessage;
        return Action.CommitMessage;
    }
}
```

- **自定义消息最大重试次数和重试间隔**

> **说明** 自定义
> 消息队列RocketMQ版
> 的客户端日志配置，请升级TCP Java SDK到1.2.2或以上版本。更多信息，请参见版本说明。

消息队列RocketMQ版
允许Consumer实例启动的时候设置最大重试次数和重试间隔，无序消息重试间隔时间不支持自定义，以无序消息重试间隔为准。
配置方式如下：

```
Properties properties = new Properties();
//配置对应Group ID的最大消息重试次数为20次，最大重试次数为字符串类型。
properties.put(PropertyKeyConst.MaxReconsumeTimes,"20");
//配置对应Group ID的消息重试间隔时间为3000毫秒，重试间隔时间为字符串类型。
properties.put(PropertyKeyConst.suspendTimeMillis,"3000");
Consumer consumer = ONSFactory.createConsumer(properties);
```

> **注意** 配置采用覆盖的方式生效，即最后启动的Consumer实例会覆盖之前启动的实例的配置。因此，请确保同一Group ID下的所有Consumer实例设置的最大重试次数和重试间隔相同，否则各实例间的配置将会互相覆盖。

- **获取消息重试次数**
  Consumer收到消息后，可按照以下方式获取消息的重试次数，消息重试间隔时间一般不需要获取。

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        //获取消息的重试次数。
        System.out.println(message.getReconsumeTimes());
        return Action.CommitMessage;
    }
}
```

1.8. 消息过滤

消费者订阅了某个Topic后，
消息队列RocketMQ版
会将该Topic中的所有消息投递给消费端进行消费。若消费者只需要关注部分消息，可通过设置过滤条件在消息队列RocketMQ版
服务端完成消息过滤，只消费需要关注的消息。本文介绍消息过滤的功能描述、应用场景、使用限制、配置方式及示例代码。

功能描述

消息过滤功能指消息生产者向Topic中发送消息时，设置消息属性对消息进行分类，消费者订阅Topic时，根据消息属性设置过滤条件对消息进行过滤，只有符合过滤条件的消息才会被投递到消费端进行消费。

消费者订阅Topic时若未设置过滤条件，无论消息发送时是否有设置过滤属性，Topic中的所有消息都将被投递到消费端进行消费。

消息队列RocketMQ版
支持的消息过滤方式如下：

过滤方式	说明	场景	实例限制	协议限制

过滤方式	说明	场景	实例限制	协议限制
Tag过滤（默认过滤方式）	• 发送者：设置消息Tag。 • 订阅者：订阅消息Tag。 订阅的Tag和发送者设置的消息Tag一致，则消息被投递给消费端进行消费。	简单过滤场景。 一条消息支持设置一个Tag，仅需要对Topic中的消息进行一级分类并过滤时可以使用此方式。	无。	无。
SQL属性过滤	• 发送者：设置消息的自定义属性。 • 订阅者：订阅时自定义SQL过滤表达式，根据自定义属性过滤消息。 满足过滤表达式的消息被投递给消费端进行消费。	复杂过滤场景。 一条消息支持设置多个自定义属性，可根据SQL语法自定义组合多种类型的表达式对消息进行多级分类并实现多维度的过滤。	仅企业铂金版实例支持该功能。	仅商业版TCP协议的SDK支持该功能。

Tag过滤

Tag，即消息标签，用于对某个Topic下的消息进行分类。

消息队列RocketMQ版
的生产者在发送消息时，指定消息的Tag，消费者需根据已经指定的Tag来进行订阅。

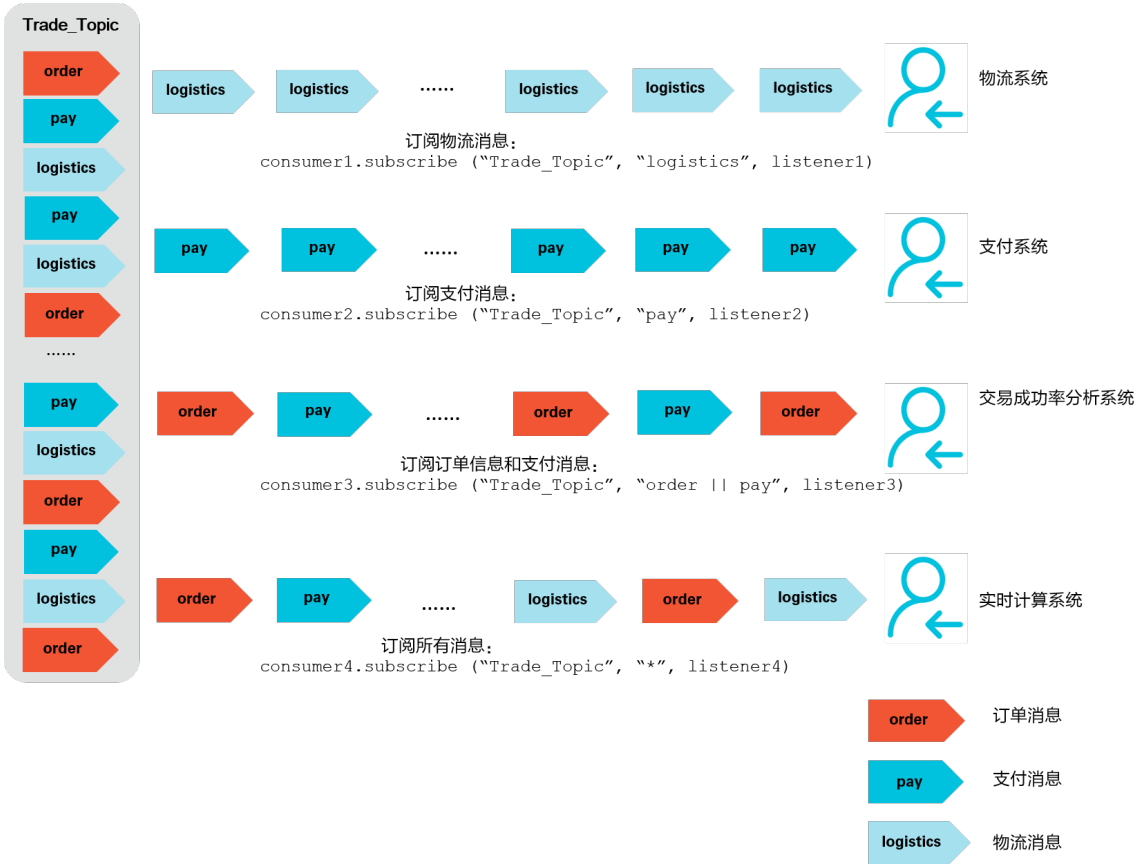
以下图电商交易场景为例，从客户下单到收到商品这一过程会生产一系列消息，以如下消息为例：

- 订单消息
- 支付消息
- 物流消息

这些消息会发送到名称为**Trade_Topic**的Topic中，被各个不同的系统所订阅，以如下系统为例：

- 支付系统：只需订阅支付消息。
- 物流系统：只需订阅物流消息。
- 交易成功率分析系统：需订阅订单和支付消息。
- 实时计算系统：需要订阅所有和交易相关的消息。

过滤示意图如下所示。



消息队列RocketMQ版
支持通过SDK配置Tag过滤功能，分别在消息发送和订阅代码中设置消息Tag和订阅消息Tag。SDK详细信息，请参见SDK参考概述。消息发送端和消费端的代码配置方法如下：

- 发送消息
  发送消息时，每条消息必须指明Tag。

```
Message msg = new Message("MQ_TOPIC","TagA","Hello MQ".getBytes());
```

- 订阅所有Tag
  消费者如需订阅某Topic下所有类型的消息，Tag用星号（*）表示。

```
consumer.subscribe("MQ_TOPIC", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

- 订阅单个Tag
  消费者如需订阅某Topic下某一种类型的消息，请明确标明Tag。

```
consumer.subscribe("MQ_TOPIC", "TagA", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

- 订阅多个Tag
  消费者如需订阅某Topic下多种类型的消息，请在多个Tag之间用两个竖线（||）分隔。

```
consumer.subscribe("MQ_TOPIC", "TagA||TagB", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

- 错误示例
  同一个消费者多次订阅某个Topic下的Tag，以最后一次订阅的Tag为准。

```
//如下错误代码中，Consumer只能订阅到MQ_TOPIC下TagB的消息，而不能订阅TagA的消息。
consumer.subscribe("MQ_TOPIC", "TagA", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
consumer.subscribe("MQ_TOPIC", "TagB", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

场景示例

配置方式

SQL属性过滤

SQL属性过滤是在消息发送时设置消息的自定义属性，消费者订阅时使用SQL语法设置过滤表达式，根据自定义属性过滤消息，

消息队列RocketMQ版
根据表达式的逻辑进行计算，将符合条件的消息投递到消费端。

> **说明** Tag属于一种特殊的消息属性，所以SQL过滤方式也兼容Tag过滤方法，支持通过Tag属性过滤消息。在SQL语法中，Tag的属性值为TAGS。

使用SQL属性过滤消息时，有以下限制：

- 只有企业铂金版实例支持SQL属性过滤，标准版实例不支持该功能。
- 只有TCP协议的客户端支持SQL属性过滤，HTTP协议的客户端不支持该功能。
- 若服务端不支持SQL过滤时，客户端使用SQL过滤消息，则客户端启动会报错或收不到消息。

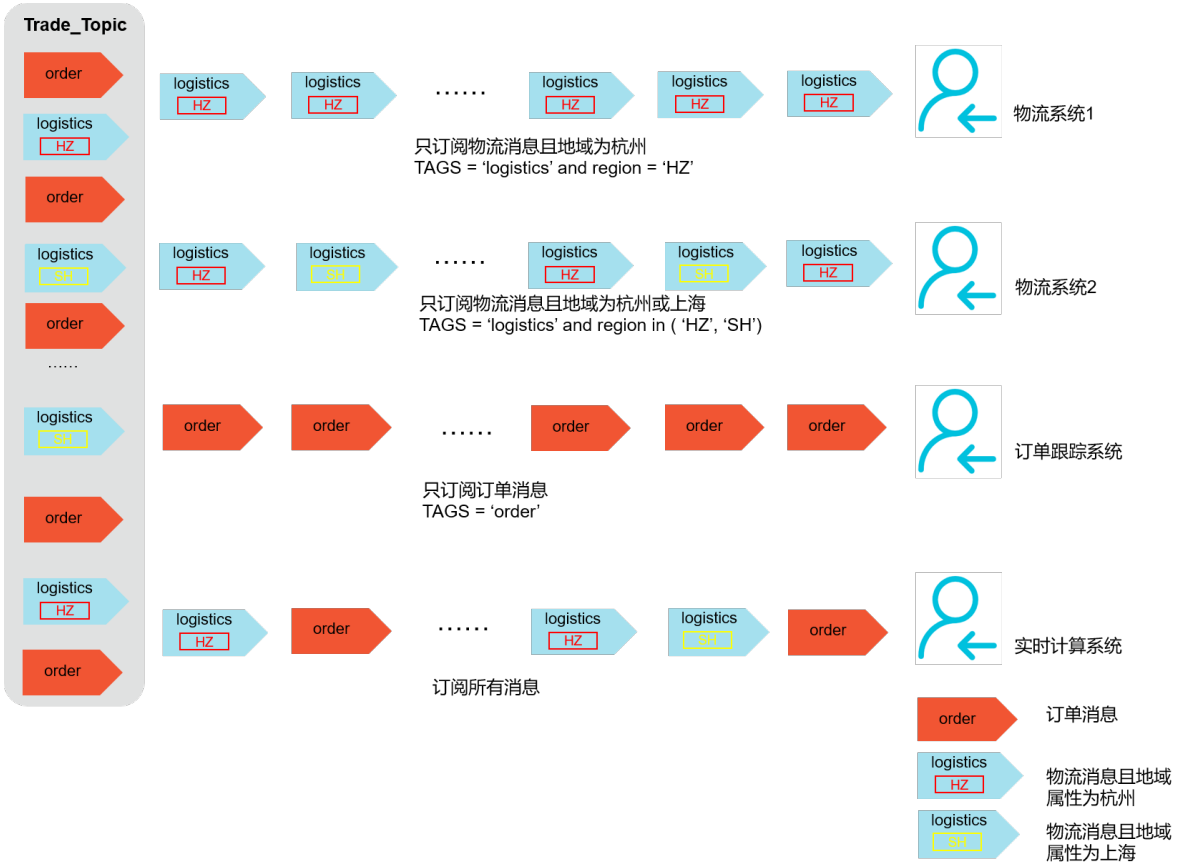
以下图电商交易场景为例，从客户下单到收到商品这一过程会生产一系列消息，按照类型将消息分为订单消息和物流消息，其中给物流消息定义地域属性，按照地域分为杭州和上海：

- 订单消息
- 物流消息
  - 物流消息且地域为杭州
  - 物流消息且地域为上海

这些消息会发送到名称为Trade_Topic的Topic中，被各个不同的系统所订阅，以如下系统为例：

- 物流系统1：只需订阅物流消息且消息地域为杭州。
- 物流系统2：只需订阅物流消息且消息地域为杭州或上海。
- 订单跟踪系统：只需订阅订单消息。
- 实时计算系统：需要订阅所有和交易相关的消息。

过滤示意图如下所示。



消息队列RocketMQ版

支持通过SDK配置SQL属性过滤。发送端需要在发送消息的代码中设置消息的自定义属性；消费端需要在订阅消息代码中设置SQL语法的过滤表达式。SDK详细信息，请参见SDK参考概述。消息发送端和消费端的代码配置方法如下：

- 消息发送端：
  设置消息的自定义属性。

```
Message msg = new Message("topic", "tagA", "Hello MQ".getBytes());
// 设置自定义属性A，属性值为1。
msg.putUserProperties("A", "1");
```

- 消息消费端
  使用SQL语法设置过滤表达式，并根据自定义属性过滤消息。

  > **注意** 使用属性时，需要先判断属性是否存在。若属性不存在则过滤表达式的计算结果为NULL，消息不会被投递到消费端。

```
// 订阅自定义属性A存在且属性值为1的消息。
consumer.subscribe("topic", MessageSelector.bySql("A IS NOT NULL AND TAGS IS NOT NULL AND A = '1'"), new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.printf("Receive New Messages: %s %n", message);
        return Action.CommitMessage;
    }
});
```

SQL语法如下：

语法	说明	示例
IS NULL	判断属性不存在。	`a IS NULL` ：属性a不存在。
IS NOT NULL	判断属性存在。	`a IS NOT NULL` ：属性a存在。
• > • >= • < • <=	用于比较数字，不能用于比较字符串，否则Consumer客户端启动会报错。 **说明** 可转化为数字的字符串也被认为是数字。	• `a IS NOT NULL AND a > 100` ：属性a存在且属性a的值大于100。 • `a IS NOT NULL AND a > 'abc'` ：错误示例，abc为字符串，不能用于比较大小。
BETWEEN xxx AND xxx	用于比较数字，不能用于比较字符串，否则Consumer客户端启动会报错。等价于>= xxx AND <= xxx。表示属性值在两个数字之间。	`a IS NOT NULL AND (a BETWEEN 10 AND 100)` ：属性a存在且属性a的值大于等于10且小于等于100。
NOT BETWEEN xxx AND xxx	用于比较数字，不能用于比较字符串，否则Consumer客户端启动会报错。等价于< xxx OR > xxx，表示属性值在两个值的区间之外。	`a IS NOT NULL AND (a NOT BETWEEN 10 AND 100)` ：属性a存在且属性a的值小于10或大于100。
IN (xxx, xxx)	表示属性的值在某个集合内。集合的元素只能是字符串。	`a IS NOT NULL AND (a IN ('abc', 'def'))` ：属性a存在且属性a的值为abc或def。
• = • <>	等于和不等于。可用于比较数字和字符串。	`a IS NOT NULL AND (a = 'abc' OR a<>'def')` ：属性a存在且属性a的值为abc或a的值不为def。

语法	说明	示例
• AND • OR	逻辑与和逻辑或。可用于组合任意简单的逻辑判断，需要将每个逻辑判断内容放入括号内。	`a IS NOT NULL AND (a > 100) OR (b IS NULL)` ：属性a存在且属性a的值大于100或属性b不存在。

由于SQL属性过滤是发送端定义消息属性，消费端设置SQL过滤条件，因此过滤条件的计算结果具有不确定性，服务端的处理方式为：

- 如果过滤条件的表达式计算抛异常，消息默认被过滤，不会被投递给消费端。例如比较数字和非数字类型的值。
- 如果过滤条件的表达式计算值为null或不是布尔类型（true和false），则消息默认被过滤，不会被投递给消费端。例如发送消息时未定义某个属性，在订阅时过滤条件中直接使用该属性，则过滤条件的表达式计算结果为null。
- 如果消息自定义属性为浮点型，但过滤条件中使用整数进行判断，则消息默认被过滤，不会被投递给消费端。
- 发送消息
  同时设置消息Tag和自定义属性。

```
Producer producer = ONSFactory.createProducer(properties);
// 设置Tag的值为tagA。
Message msg = new Message("topicA", "tagA", "Hello MQ".getBytes());
// 设置自定义属性region为hangzhou。
msg.putUserProperties("region", "hangzhou");
// 设置自定义属性price为50。
msg.putUserProperties("price", "50");
SendResult sendResult = producer.send(msg);
```

- 根据单个自定义属性订阅消息。

```
Consumer consumer = ONSFactory.createConsumer(properties);
// 只订阅属性region为hangzhou的消息，若消息中未定义属性region或属性值不是hangzhou，则消息不会被投递到消费端。
consumer.subscribe("topicA", MessageSelector.bySql("region IS NOT NULL AND region = 'hangzhou'"), new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.printf("Receive New Messages: %s %n", message);
        return Action.CommitMessage;
    }
});
```

  预期结果：示例中发送的消息属性符合订阅的过滤条件，消息被投递到消费端。

- 同时根据Tag和自定义属性订阅消息。

```
Consumer consumer = ONSFactory.createConsumer(properties);
// 只订阅Tag的值为tagA且属性price大于30的消息。
consumer.subscribe("topicA", MessageSelector.bySql("TAGS IS NOT NULL AND price IS NOT NUL
L AND TAGS = 'tagA' AND price > 30 "), new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.printf("Receive New Messages: %s %n", message);
        return Action.CommitMessage;
    }
});
```

预期结果：示例中发送的消息Tag和自定义属性符合订阅的过滤条件，消息被投递到消费端。

- 同时根据多个自定义属性订阅消息。

```
Consumer consumer = ONSFactory.createConsumer(properties);
// 只订阅region为hangzhou且属性price小于20的消息。
consumer.subscribe("topicA", MessageSelector.bySql("region IS NOT NULL AND price IS NOT N
ULL AND region = 'hangzhou' AND price < 20"), new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.printf("Receive New Messages: %s %n", message);
        return Action.CommitMessage;
    }
});
```

预期结果：消息不会被投递到消费端。订阅的过滤条件中price小于20，发送的消息中price属性值为50，不符合订阅过滤条件。

- 订阅Topic中的所有消息，不进行过滤。

```
Consumer consumer = ONSFactory.createConsumer(properties);
// 若需要订阅Topic中的所有消息，需要将SQL表达式的值设置为“TRUE”。
consumer.subscribe("topicA", MessageSelector.bySql("TRUE"), new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.printf("Receive New Messages: %s %n", message);
        return Action.CommitMessage;
    }
});
```

预期结果：Topic中的所有消息都将被投递到消费端进行消费。

- 错误示例
  消息发送时未自定义某属性，消费端在订阅时未判断该属性是否存在直接使用，则消息不会被投递给消费端。

```
Consumer consumer = ONSFactory.createConsumer(properties);
// 属性product在发送消息时未定义，过滤失败，消息不会被投递至消费端。
consumer.subscribe("topicA", MessageSelector.bySql("product = 'MQ'"), new MessageListener
() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.printf("Receive New Messages: %s %n", message);
        return Action.CommitMessage;
    }
});
```

使用限制

场景示例

配置方式

示例代码

更多信息

- 同一个Group ID下的消费者实例与Topic的订阅关系需保持一致，更多信息，请参见订阅关系一致。
- 合理使用Topic和Tag来过滤消息可以让业务更清晰，更多信息，请参见Topic与Tag最佳实践。

1.9. Exactly-Once投递语义

本文主要介绍
消息队列RocketMQ版
的Exactly-Once投递语义的概念和典型使用场景，以便您理解如何使得消息只被消费端处理且仅处理一次。

什么是Exactly-Once投递语义

Exactly-Once是指发送到消息系统的消息只能被消费端处理且仅处理一次，即使生产端重试消息发送导致某消息重复投递，该消息在消费端也只被消费一次。

Exactly-Once语义是消息系统和流式计算系统中消息流转的最理想状态，但是在业界并没有太多理想的实现。因为真正意义上的Exactly-Once依赖消息系统的服务端、消息系统的客户端和用户消费逻辑这三者状态的协调。例如，当您的消费端完成一条消息的消费处理后出现异常宕机，而消费端重启后由于消费的位点没有同步到消息系统的服务端，该消息有可能被重复消费。

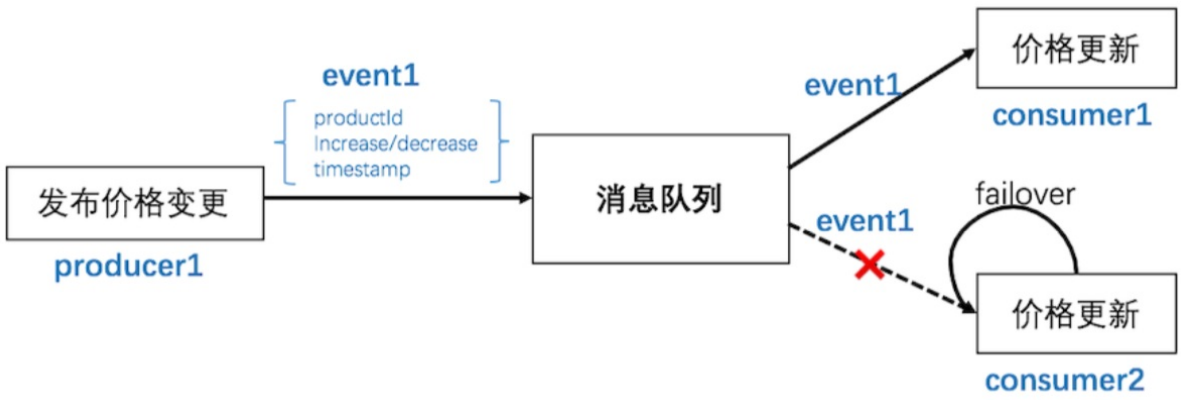
业界对于Exactly-Once投递语义存在很大的争议，很多人会拿出“FLP不可能理论”或者其他一致性定律对此议题进行否定，但事实上，特定场景的Exactly-Once语义实现并不是非常复杂，只是因为通常大家没有精确的描述问题的本质。

如果您要实现一条消息的消费结果只能在业务系统中生效一次，您需要解决的只是如何保证同一条消息的消费幂等问题。

消息队列RocketMQ版
的Exactly-Once语义就是解决业务中最常见的一条消息的消费结果（消息在消费端计算处理的结果）在数据库系统中有且仅生效一次的问题。

典型使用场景

在电商系统中，上游实时计算模块发布商品价格变更的信息，异步通知到下游商品管理模块进行价格变更。此时，需要保证每一条信息的消费幂等，即重复的价格变更信息只会生效一次，这样便不会发生价格多次重复修改的情况，确保实现了消息消费的幂等。



更多信息

如何使用Exactly-Once投递语义的具体步骤，请参见使用Exactly-Once投递语义收发消息。

1.10. 集群消费和广播消费

消息队列RocketMQ版
支持集群消费和广播消费，本文介绍集群消费和广播消费的基本概念、适用场景、功能差异、注意事项以及设置方式。

背景信息

消息队列RocketMQ版
是基于发布和订阅模式的消息系统。消费者，即消息的订阅方，从订阅的Topic中接收消息并消费。
消费者应用一般是分布式系统，以集群方式部署。消费者在订阅Topic时，可根据实际业务选择集群消费或广播消费，集群中的多个消费者将按照实际选择的消费模式消费消息。

使用相同Group ID的消费者属于同一个集群。同一个集群下的消费者消费逻辑必须完全一致（包括Tag的使用）。更多信息，请参见订阅关系一致。

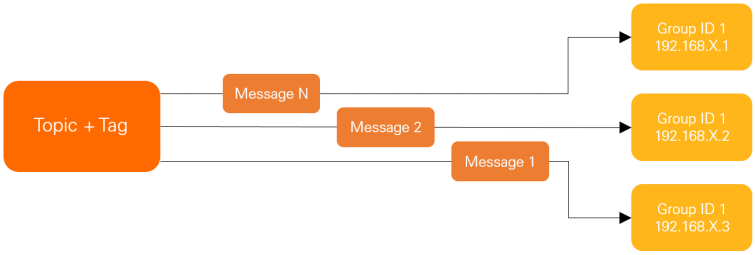
集群消费模式

基本概念

集群消费：当使用集群消费模式时，
消息队列RocketMQ版
认为任意一条消息只需要被集群内的任意一个消费者处理即可。

适用场景

适用于消费端集群化部署，每条消息只需要被处理一次的场景。具体示例如下图所示：



广播消费模式

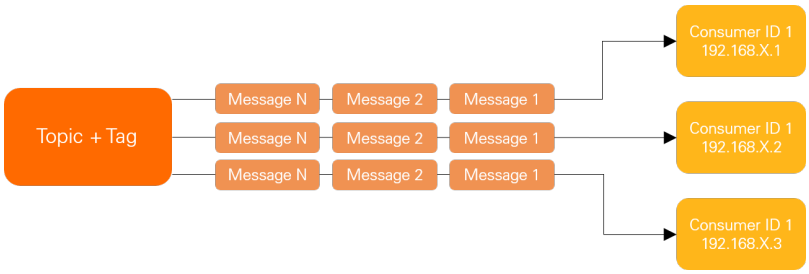
基本概念

广播消费：当使用广播消费模式时，

消息队列RocketMQ版
会将每条消息推送给集群内所有的消费者，保证消息至少被每个消费者消费一次。

应用场景

适用于消费端集群化部署，每条消息需要被集群下的每个消费者处理的场景。具体消费示例如下图所示：



集群消费和广播消费差异

集群消费和广播消费模式下，各功能的支持情况如下：

功能	集群消费	广播消费
TCP协议SDK	✓	✓
HTTP协议SDK	✓	✗
顺序消息	✓	✗
重置消费位点	✓	✗
消息重试	✓	✗
消息堆积查询、报警	✓	✗
订阅关系查询	✓	✗
消费进度	服务端维护 • 可靠性更高，客户端重启后，将按照上次的消费进度继续消费。 • 支持服务端消费重试机制，详细信息，请参见消息重试。	客户端维护 出现重复消费的概率稍大于集群模式，客户端每次重启都会从最新消息消费。

> **说明**
>
> ✓：表示支持该功能。
>
> ✗：表示不支持该功能。

注意事项

集群消费

- 集群模式下，每一条消息都只会被分发到一台机器上处理。如果需要被集群下的每一台机器都处理，请使用广播模式。
- 集群模式下，不保证每一次失败重投的消息路由到同一台机器上。

广播消费

- 广播模式下，
  消息队列RocketMQ版
  保证消息至少被客户端消费一次，但是并不会重投消费失败的消息，因此业务方需要关注消费失败的情况。
- 广播模式下，客户端每一次重启都会从最新消息消费。客户端在被停止期间发送至服务端的消息将会被自动跳过，请谨慎选择。
- 广播模式下，每条消息都会被大量的客户端重复处理，因此推荐尽可能使用集群模式。

设置方式

设置消息的消费方式，需要在订阅消息SDK中设置以下参数：

TCP协议SDK

- **集群消费**

```
// MessageModel设置为CLUSTERING（不设置的情况下，默认为集群订阅方式）。
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
```

- **广播消费**

```
// MessageModel设置为BROADCASTING。
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
```

以上示例以Java语言为例，更多信息，请参见以下文档：

- Java：订阅消息
- .NET：订阅消息
- C/C++：订阅消息

HTTP协议SDK

HTTP协议仅支持集群消费，无需设置。更多信息，请参见HTTP SDK使用须知。

1.11. 批量消费

如需提高消息的处理效率，或降低下游资源的API调用频率，您可使用批量消费功能。本文介绍批量消费的定义、优势与场景、使用限制和示例代码等信息。

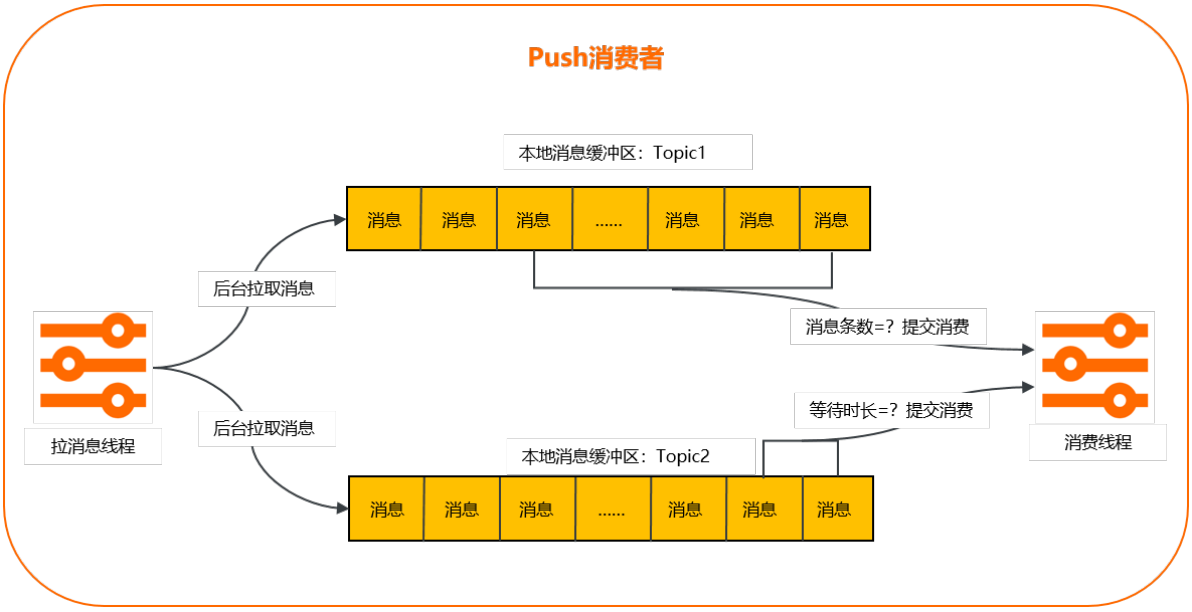
什么是批量消费

- 定义
  批量消费是
  消息队列RocketMQ版
  通过Push消费者提供的、将消息分批次消费的功能。

> **说明** 根据消息获取方式，
> 消息队列RocketMQ版
> 提供Push和Pull两种类型的消费者，更多信息，请参见名词解释。

- 功能原理
  批量消费主要分为以下两个阶段：
  i. 消息从生产者发布至消息队列RocketMQ版后，Push消费者中的拉消息线程通过长轮询将消息拉到后台缓存。
  ii. Push消费者根据缓存情况是否满足任一批量条件，判断是否将消息提交给消费线程完成消费。

  具体示意图如下所示。



使用限制

- 请确保您使用的SDK是商业版TCP Java SDK，且版本在1.8.7.3.Final或以上，详细的版本说明和获取方式，请参见商业版TCP Java SDK版本说明。
- 支持一次提交最多1024条消息，支持攒批等待最多450秒。

功能优势及场景示例

批量消费的功能优势和场景示例说明如下：

- 优势一：提高消息的吞吐能力和处理效率
  场景示例：上游订单系统和下游Elasticsearch系统间通过消息队列RocketMQ版解耦，Elasticsearch消费订单系统的10条日志消息，每一条消息对于Elasticsearch系统而言都是一次RPC请求，假设一次RPC请求耗时10毫秒，那么不使用批量消费的耗时为10×10=100毫秒；理想状态下，使用批量消费的耗时可缩短至10毫秒，因为10条消息合并为一次消费，大大提高消息的处理效率。
- 优势二：降低下游资源的API调用频率
  场景示例：给数据库中插入数据，每更新一条数据执行一次插入任务，如果数据更新较频繁，可能会对数据库造成较大压力。此时，您可以设置每10条数据批量插入一次或每5秒执行一次插入任务，降低系统运行压力。

示例代码

批量消费的示例代码如下所示。

```
import com.aliyun.openservices.ons.api.Action;
import com.aliyun.openservices.ons.api.ConsumeContext;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.batch.BatchConsumer;
import com.aliyun.openservices.ons.api.batch.BatchMessageListener;
import java.util.List;
import java.util.Properties;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.tcp.example.MqConfig;
public class SimpleBatchConsumer {
    public static void main(String[] args) {
        Properties consumerProperties = new Properties();
        consumerProperties.setProperty(PropertyKeyConst.GROUP_ID, MqConfig.GROUP_ID);
        consumerProperties.setProperty(PropertyKeyConst.AccessKey, MqConfig.ACCESS_KEY);
        consumerProperties.setProperty(PropertyKeyConst.SecretKey, MqConfig.SECRET_KEY);
        consumerProperties.setProperty(PropertyKeyConst.NAMESRV_ADDR, MqConfig.NAMESRV_ADDR
);
        // 设置批量消费最大消息数量，当指定Topic的消息数量已经攒够128条，SDK立即执行回调进行消费。默认值：32，取值范围：1~1024。
        consumerProperties.setProperty(PropertyKeyConst.ConsumeMessageBatchMaxSize, String.
valueOf(128));
        // 设置批量消费最大等待时长，当等待时间达到10秒，SDK立即执行回调进行消费。默认值：0，取值范围：0~450，单位：秒。
        consumerProperties.setProperty(PropertyKeyConst.BatchConsumeMaxAwaitDurationInSecon
ds, String.valueOf(10));
        BatchConsumer batchConsumer = ONSFactory.createBatchConsumer(consumerProperties);
        batchConsumer.subscribe(MqConfig.TOPIC, MqConfig.TAG, new BatchMessageListener() {
             @Override
            public Action consume(final List<Message> messages, ConsumeContext context) {
                System.out.printf("Batch-size: %d\n", messages.size());
                // 批量消息处理。
                return Action.CommitMessage;
            }
        });
        //启动batchConsumer。
        batchConsumer.start();
        System.out.println("Consumer start success.");
        //等待固定时间防止进程退出。
        try {
            Thread.sleep(200000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

参数描述如下表所示。

参数名	参数类型	是否必选	描述
ConsumeMessageBatchMaxSize	String	否 **说明** 如未指定参数值，则使用默认值。	批量消费的最大消息数量，缓存的消息数量达到参数设置的值，Push消费者SDK会将缓存的消息统一提交给消费线程，实现批量消费。取值范围：[1,1024]，默认值：32，单位：条。
BatchConsumeMaxAwaitDurationInSeconds	String		批量消费的最大等待时长，等待时长达到参数设置的值，会将缓存的消息统一推送给消费者进行批量消费。取值范围：[1,1024]，默认值：0，单位：秒。

说明

- 具体的示例代码，请以消息队列RocketMQ版代码库为准。
- 更多参数信息，请参见接口和参数说明。

最佳实践

请合理设置ConsumeMessageBatchMaxSize和BatchConsumeMaxAwaitDurationInSeconds参数的取值，只要达到任一参数设置的批量条件，即会触发提交批量消费。例如ConsumeMessageBatchMaxSize设置为128，BatchConsumeMaxAwaitDurationInSeconds设置为1，1秒内虽然没有积攒到128条消息，仍然会触发批量消费，此时返回的Batch-size会小于128。

此外，为了获得更好的批量消费效果，强烈推荐您实现消息幂等，保证消息有且仅被处理1次。幂等处理的具体信息，请参见消费幂等。

更多信息

商业版TCP Java SDK订阅消息

1.12. 异地双活

依托于阿里云高速通道专线、事件总线EventBridge和MSHA（Multi-Site High Availability）多活容灾平台，消息队列RocketMQ版
提供异地双活功能，通过跨实例间数据的双向同步和业务切流能力，实现业务恢复和故障恢复解耦，保障故障场景下的业务连续性。本文介绍异地双活的概念、应用场景、功能优势、使用限制和计费说明。

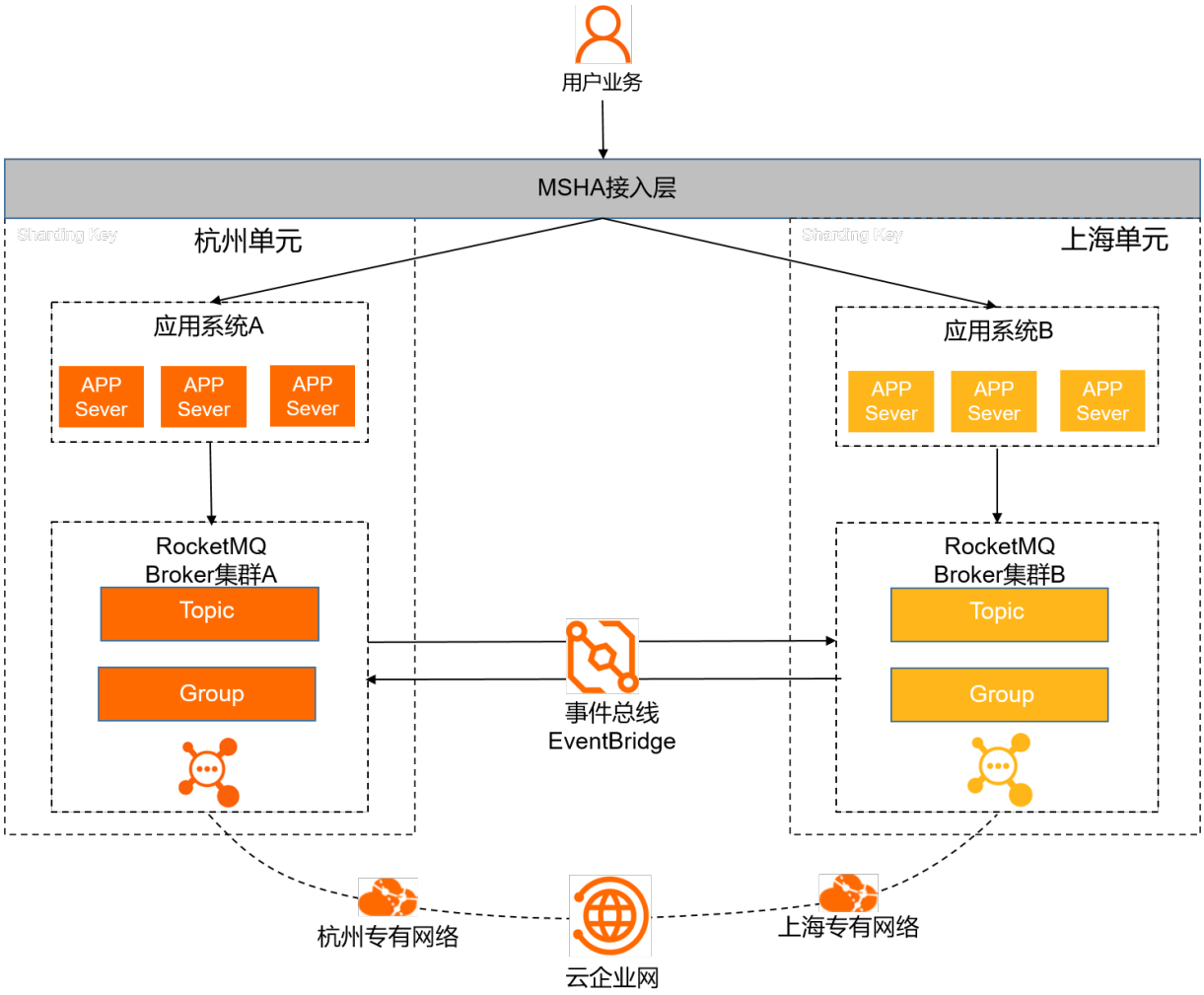
什么是异地双活

多活容灾MSHA是在阿里巴巴电商业务环境演进出来的多活容灾架构解决方案，可以将业务恢复和故障恢复解耦。基于灵活的流量规则调度、跨域跨云管控、数据保护等能力，实现故障场景下的业务快速切换及恢复。

消息队列RocketMQ版
依赖于阿里云高速通道专线、事件总线EventBridge和多活容灾MSHA实现跨地域实例间的消息数据双向同

步。和传统容灾不同的是，异地双活方案中，部署在不同数据中心的
消息队列RocketMQ版
并行地为业务访问提供服务，实现了资源的充分利用，避免一个或多个备份中心处于闲置状态。当一个数据中心发生故障而另一个数据中心正常运行时，可通过MSHA实现流量的切换，提供正常的消息服务，实现用户对故障无感知。异地双活功能不仅解决了容灾本身问题，还提升了业务连续性，并且实现了资源的异地扩展。

消息队列RocketMQ版
的异地双活具体实现如下：



- 杭州单元和上海单元分别部署了一套完整的业务系统。
- MSHA接入层按照数据分流规则将业务数据路由到两个业务单元，杭州单元和上海单元的应用系统和消息队列RocketMQ版
  Broker集群各自处理对应地域的业务。
- 两个单元的Broker集群配置了异地双活功能，依赖于事件总线EventBridge提供的事件路由能力，以及云企业网CEN（Cloud Enterprise Network）提供的专有网络间的私网通信通道，Broker集群A和Broker集群B间的数据实现了双向同步，包括Topic数据、Group数据及消费位点等信息。正常情况下，杭州单元和上海单元只负责各自单元的业务，并且同时将各自单元的消息数据同步到对方单元的集群中进行容灾备份。
- 假设杭州单元发生灾害，整个业务系统出现故障，此时通过MSHA切流操作将整个杭州的业务切到上海单元，因为配置了异地双活，上海单元的Broker集群存储了杭州单元的业务数据，可以继续处理未完成的消息数据。在保证业务不中断的前提下排查故障并修复，做到先恢复业务再恢复故障。
- 当杭州单元故障恢复后，通过回切流量，将杭州单元的业务重新切回到杭州单元的系统处理，整个过程用

户不感知故障，不影响用户业务体验。

使用限制

- **实例类型限制**：异地双活功能仅企业铂金版实例支持，标准版实例不支持。
- **地域限制**：目前仅华东1（杭州）、华东2（上海）、华北2（北京）和华南1（深圳）地域支持使用异地双活功能，并且使用前您需要提交工单申请，其他地域暂不支持该功能。

计费说明

异地双活为

消息队列RocketMQ版
的高级功能。若您开通了异地双活功能，
消息队列RocketMQ版
会按照包年包月方式进行收费；未开通则不收取异地双活功能费用。具体计费信息，请参见异地双活定价详情。

应用场景

异地双活功能适用于以下典型业务场景：

- 按地域划分单元的业务场景，如物流业务。可以通过物流订单的地域将业务进行划分，将业务引流到不同地域的生产中心同时处理，提高资源利用率和业务并发度。
- 对业务数据可靠性有严格要求的业务场景，如金融证券等。当系统出现故障会对交易结果产生较大影响，通过异地双活可以将业务快速切换到容灾站点，并且根据同步的数据继续处理未完成的消息。

功能优势

- **可用性**
  和传统容灾方案相比，异地双活方案中所有生产中心数据实现双向同步，并且均可对外提供服务，各中心分担业务流量，提高了资源使用率。
- **故障快速恢复**
  异地双活功能够有效保障业务连续性。当其中一个生产中心发生故障，区别于传统的解决思路，不是去排查、处理和修复故障，而是立即使用切流将业务切换到其他生产中心，保证业务的连续性，将业务恢复和故障恢复解耦。
- **异地容量扩容**
  业务高速发展，受限于单地有限资源，也存在资源存储、计算网络瓶颈等问题。在
  消息队列RocketMQ版
  水平拓展能力的支撑下，使业务具备其他机房或者其它地域快速扩建的特性，减少成本浪费。

更多信息

配置异地双活功能的具体操作，请参见异地双活实例管理。

2.常见问题

- 快速入门
- 计费FAQ
- 消息负载均衡策略
- 顺序消息FAQ

订阅/消费异常

- 客户端中常见的日志说明
- 订阅关系不一致
- 客户端无法连接消息队列RocketMQ版实例
- 客户端的状态出现异常
- 消费者连接Broker时提示Group ID不存在
- Producer（生产者）出现有关消息不合法的异常信息
- 启动客户端时提示Group ID重复
- 启动客户端时提示获取Topic队列失败
- 启动客户端时提示“UnknownHostException”
- 客户端消费的内存过多
- 启动.NET客户端时提示DLL加载失败或提示其他运行错误
- Consumer实例未收到预期的消息
- 消息状态显示“Consumed”但消费端业务日志显示没有收到消息
- 消息消费失败时能否重试消费
- Producer出现有关消息不合法的异常信息
- Consumer提示有关参数不合法的报错
- message类的userproperties属性如何自定义

消息堆积

- Java进程的消息堆积
- 收到信息堆积告警

消息查询

- 无法找到历史查询记录

消息轨迹

- 查询不到轨迹数据
- 已经消费的消息在消息轨迹中显示未消费
- 消费者列表中没有Group ID

最新发布

- Group ID的消费位点介绍