蚂蚁金服金融科技产品手册

消息队列

产品版本： V1.1.3

文档版本： V20200528

蚂蚁金服金融科技文档

商标声明

及其他蚂蚁金服金融科技服务相关的商标均为蚂蚁金服金融科技所有。
本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

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。蚂蚁金服金融科技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，并在蚂蚁金服金融科技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蚂蚁金服金融科技授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

目录

1 什么是消息队列

1.1 概述

SOFAStack 消息队列（SOFAStack MQ，简称 SOFAMQ）是基于 Apache RocketMQ 构建的分布式消息中间件，并与金融分布式架构 SOFAStack 深度集成，为分布式应用系统提供异步解耦和削峰填谷的能力，支持事务消息、顺序消息、定时消息等多种消息类型，并具备高可靠、高吞吐、低延时等金融级特性。

应用场景

异步解耦
消息队列的生产消费模型可以解耦上下游业务系统，并支持下游多个消费者对同一消息进行消费和处理。以金融场景为例，支付中心作为支付宝主站最核心的系统，每笔支付数据的产生会引起几百个下游业务系统的关注，包括账户中心、用户中心、权益中心、流计算分析等，整体业务系统庞大而且复杂，在应用强耦合的情况下，任一应用故障都将可能对业务带来影响。通过消息队列进行异步通信和应用解耦，可以很好的提升业务的连续性。

削峰填谷
应用分布式改造后，不同应用能承载的性能情况往往不一致，在诸如双11、店庆、秒杀等大型活动时，将会带来较高的流量脉冲，部分系统可能导致系统超负荷甚至崩溃，影响用户体验，消息队列可提供强大的抗积压能力，实现削峰填谷，生产方生产消息后，消费方可以按照系统自身的承受能力进行消息的消费。

顺序收发
顺序收发指的是消息消费者按照消息发送的顺序进行消费，保证 FIFO。金融场景里需要保证顺序的应用场景非常多，例如证券交易过程中的时间优先原则，交易系统中的订单创建、支付、退款等流程等等。

分布式事务一致性
应用解耦往往带来多个应用之间的事务一致性的问题。例如支付转账成功后，需要生成账单，更新用户积分等，此时通过消息队列的分布式事务处理功能，既可以实现系统之间的解耦，又可以保证最终的数据一致性。

更多信息请参见 应用场景 。

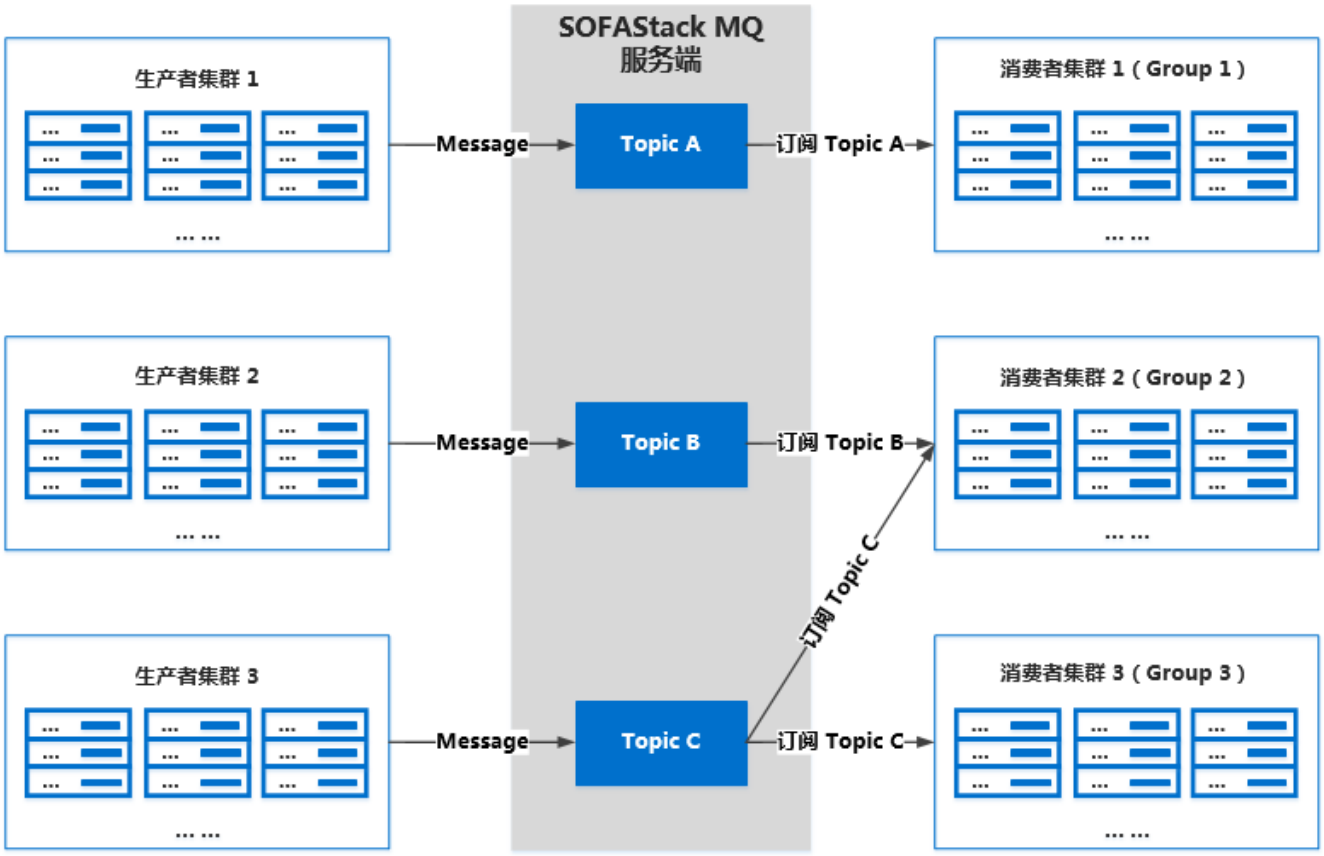
核心概念

- Topic：消息主题，一级消息类型，生产者向其发送消息。
- 生产者：也称为消息发布者，负责生产并发送消息至 Topic。
- 消费者：也称为消息订阅者，负责从 Topic 接收并消费消息。
- 消息：生产者向 Topic 发送并最终传送给消费者的数据和（可选）属性的组合。
- 消息属性：生产者可以为消息定义的属性，包含 Message Key 和 Tag。
- Group：一类生产者或消费者，这类生产者或消费者通常生产或消费同一类消息，且消息发布或订阅的逻辑一致。

更多概念解释请参见 名词解释 。

消息收发模型

消息队列支持发布/订阅模型，消息生产者应用创建 Topic 并将消息发送到 Topic。消费者应用创建对 Topic 的订阅以便从其接收消息。通信可以是一对多（扇出）、多对一（扇入）和多对多。



生产者集群

用来表示发送消息应用，一个生产者集群下包含多个生产者实例，可以是多台机器，也可以是一台机器的多个进程，或者一个进程的多个生产者对象。

一个生产者集群可以发送多个 Topic 消息。发送分布式事务消息时，如果生产者中途意外宕机，Broker 会主动回调生产者集群的任意一台机器来确认事务状态。

消费者集群

用来表示消费消息应用，一个消费者集群下包含多个消费者实例，可以是多台机器，也可以是多个进程，或者是一个进程的多个消费者对象。

一个消费者集群下的多个消费者以均摊方式消费消息。如果设置的是广播方式，那么这个消费者集群下的每个实例都消费全量数据。

一个消费者集群对应一个 Group ID，一个 Group ID 可以订阅多个 Topic，如上图中的 Group 2 所示。Group 和 Topic 的订阅关系可以通过直接在程序中设置即可。

1.2 产品优势

SOFAStack 消息队列的主要优势如下：

功能齐全

- 支持多种消息类型：普通消息、定时消息、分区顺序消息、事务消息
- 支持多种消费模式：Pub/Sub 模式、Tag 过滤
- 支持 TCP Java SDK

运维体系便捷

- 支持多维度消息查询
- 支持全链路消息轨迹

性能优越

- 海量消息堆积能力
- 毫秒级端到端延迟
- 千万级高并发处理能力

服务可靠

- 99.9% 服务高可用
- 99.99999% 数据高可靠
- 支持消息重投机制

1.3 产品架构

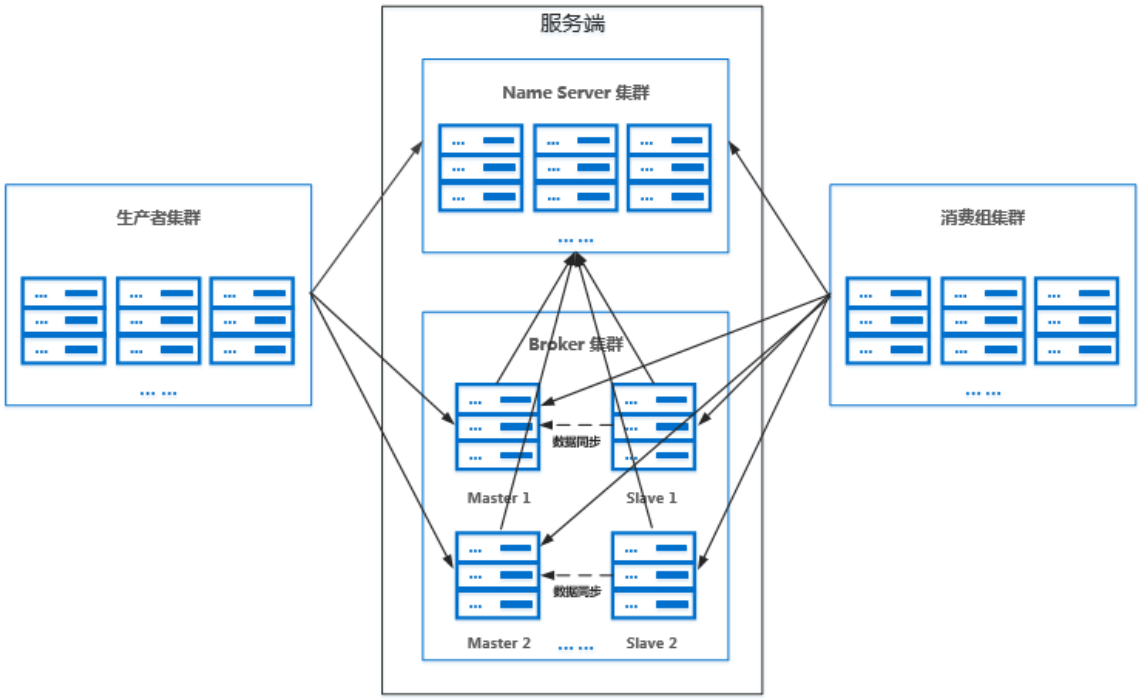
本文介绍 SOFAStack 消息队列的系统部署架构，方便您更好地理解消息队列的高可用性。

消息队列在任何一个环境都是可扩展的，生产者必须是一个集群，消息服务器必须是一个集群，消费者也同样。集群级别的高可用，是消息队列跟其他的消息服务器的主要区别，消息生产者发送一条消息到消息服务器，消息服务器会随机的选择一个消费者，只要这个消费者消费成功就认为是成功了。

> **说明**：文中所提及的消息队列的服务端或者服务器包含 Name Server、Broker 等。服务端不等同于 Broker。

系统部署架构

系统部署架构如下图所示。



图中所涉及到的概念如下所述：

- **Name Server**：是一个几乎无状态节点，可集群部署，在消息队列中提供命名服务，更新和发现 Broker 服务。
- **Broker**：消息中转角色，负责存储消息，转发消息。分为 Master Broker 和 Slave Broker，一个 Master Broker 可以对应多个 Slave Broker，但是一个 Slave Broker 只能对应一个 Master Broker。Broker 启动后需要完成一次将自己注册至 Name Server 的操作；随后每隔 30s 定期向 Name Server 上报 Topic 路由信息。
- **生产者**：与 Name Server 集群中的其中一个节点（随机）建立长链接（Keep-alive），定期从 Name Server 读取 Topic 路由信息，并向提供 Topic 服务的 Master Broker 建立长链接，且定时向 Master Broker 发送心跳。
- **消费者**：与 Name Server 集群中的其中一个节点（随机）建立长连接，定期从 Name Server 拉取 Topic 路由信息，并向提供 Topic 服务的 Master Broker、Slave Broker 建立长连接，且定时向 Master Broker、Slave Broker 发送心跳。Consumer 既可以从 Master Broker 订阅消息，也可以从 Slave Broker 订阅消息，订阅规则由 Broker 配置决定。

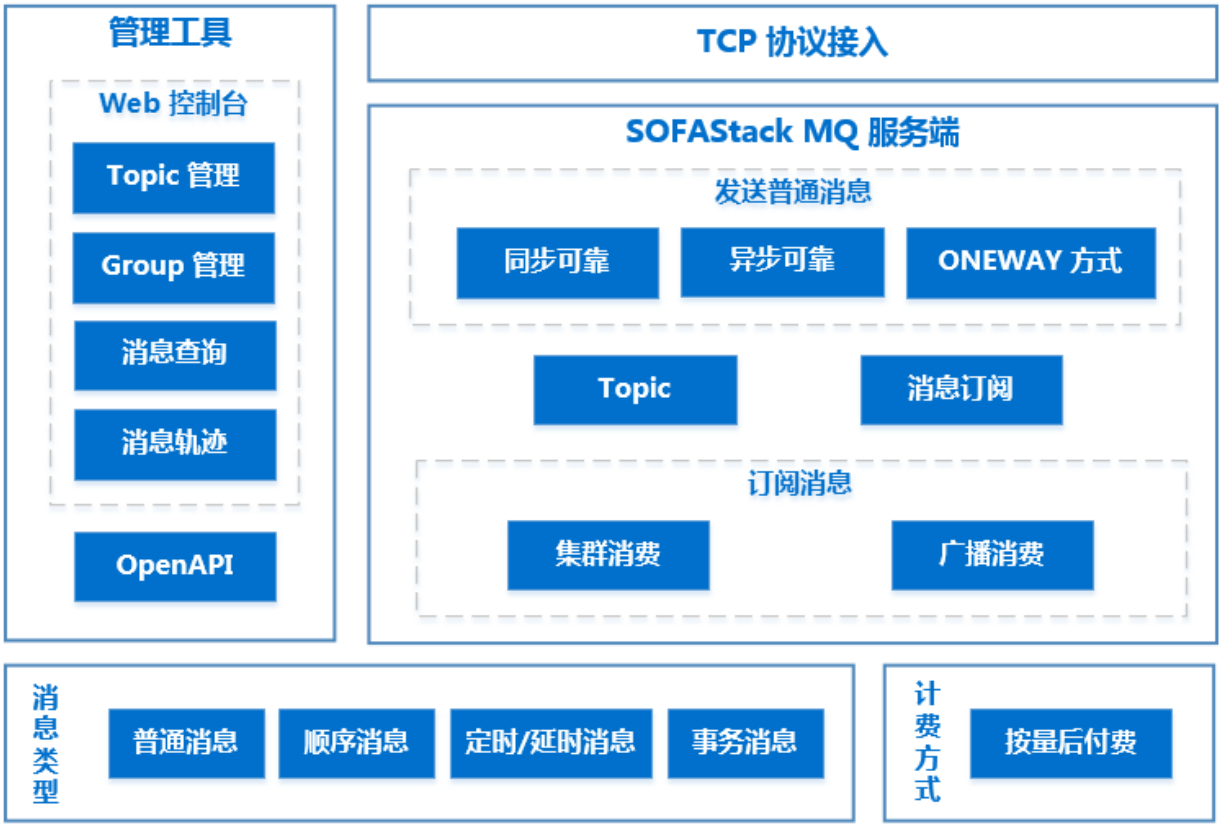
更多信息

消息队列中的概念详情，请参见 基础术语 。

1.4 功能特性

SOFAStack 消息队列在阿里云多个地域（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。

消息队列提供 TCP 协议语言接入方式，方便应用快速接入消息队列消息云服务。在网络联通的情况下，您可以将应用部署在阿里云 ECS、企业自建云，与消息队列建立连接进行消息收发。



TCP 协议接入

提供更为专业、可靠、稳定的 TCP 协议的 SDK 接入服务。支持 Java 语言。

管理工具

- Web 控制台：支持 Topic 管理、Group 管理、消息查询、消息轨迹展示和查询。
- OpenAPI：提供开放的 API 便于将消息队列管理工具集成到自己的控制台。

消息类型

- 普通消息：消息队列中无特性的消息，区别于有特性的定时/延时消息、顺序消息和事务消息。
- 事务消息：实现类似 X/Open XA 的分布事务功能，以达到事务最终一致性状态。
- 定时和延时消息：允许消息生产者对指定消息进行定时（延时）投递，最长支持 40 天。
- 顺序消息：允许消息消费者按照消息发送的顺序对消息进行消费。

有关消息类型的详细信息，参见 消息类型 。

特性功能

- 消息查询 ：消息队列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丢弃堆积消息。

1.5 应用场景

本文为您介绍SOFAStack 消息队列的适用场景，以便您更好地判断如何在业务中使用消息队列。

在互联网金融场景里，其业务涉及广泛，如支付交易、收费计息、商户结算、业务营销、会员积分、风险核查等；同时，也会涉及许多业务峰值时刻，如双11、秒杀、周年庆等。这些活动都对分布性系统中的各项微服务应用的处理性能带来很大的挑战。

消息队列作为分布式系统中的重要组件，可以很好的应对这些场景。

下文先以支付转账为场景说明消息队列如何实现以下功能：

- 异步解耦
- 分布式事务的数据一致性
- 削峰填谷

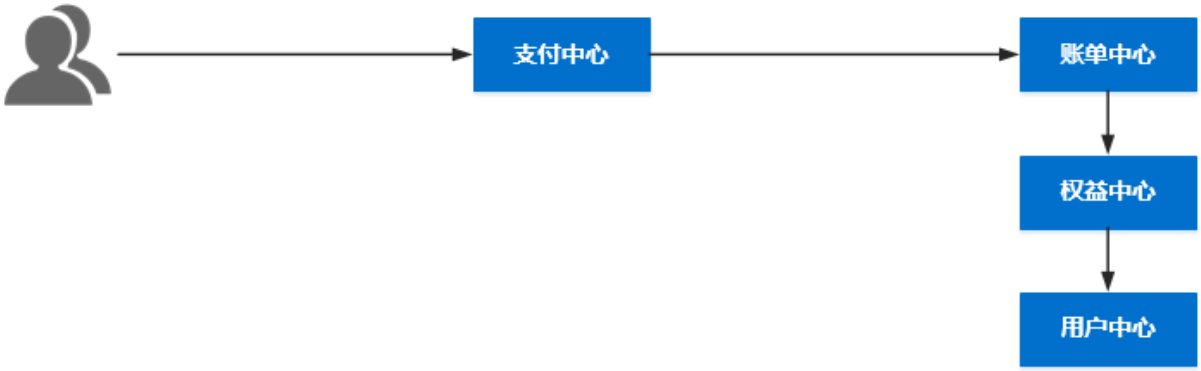
异步解耦

传统处理

最常见的一个场景是支付转账成功后，需要生成交易双方的账单，并更新用户权益，发送用户通知。传统的做法有以下两种：

- 串行方式

串行方式下的流程如下图所示。



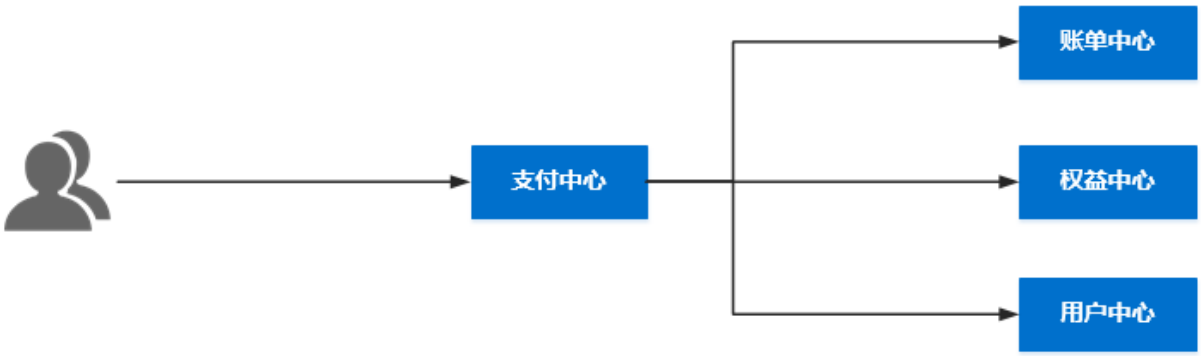
数据流动如下所述：

1. 用户在支付中心，填写金额等相关信息，完成转账操作。
2. 转账成功后，再发送请求至账单中心，生成交易双方账单。

3. 账单生成成功后，再发送请求至权益中心，更新用户积分
4. 积分更新成功后，再发送请求至用户中心，发送用户通知。

- 并行方式

并行方式下的注册流程如下图所示。



数据流动如下所述：

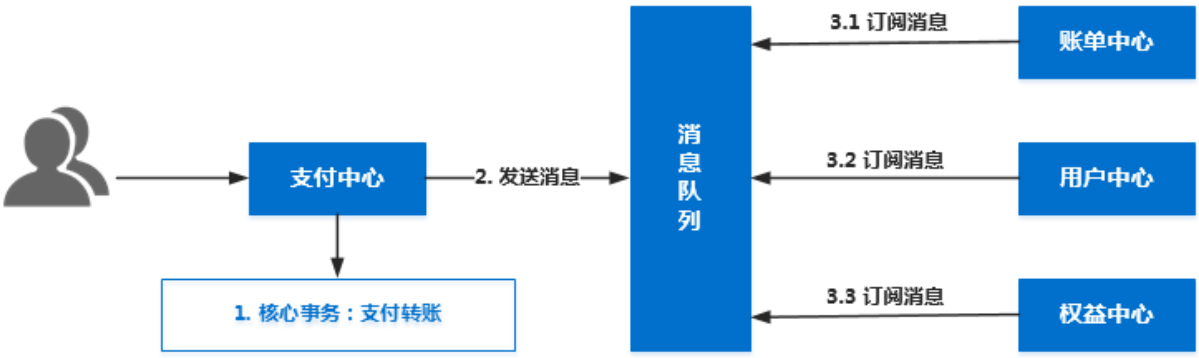
1. 用户在支付中心，填写金额等相关信息，完成转账操作。
2. 转账成功后，同时发送请求至账单中心、用户中心、权益中心完成相应操作。

以下就支付场景中使用了消息队列的效果进行说明。

异步解耦

普通消息处理

对于用户来说，转账操作成功后，后续的账单、权益、积分等不是即时需要关注的步骤，由系统保证即可。



数据流动如下所述：

1. 用户在转账页面，填写金额等相关信息，完成转账操作。
2. 转账成功后，发送一条支付消息至消息队列。消息队列会马上返回响应给支付中心，转账完成。
3. 下游的账单中心、权益中心、用户中心等系统订阅消息队列的支付消息，完成后续的业务流程。

异步解耦是消息队列的主要特点，主要目的是减少请求响应时间和解耦。主要的适用场景就是将比较耗时而且不需要即时（同步）返回结果的操作作为消息放入消息队列。同时，由于使用了消息队列，只要保证消息格式不变，消息的发送方和接收方并不需要彼此联系，也不需要受对方的影响，即解耦和。

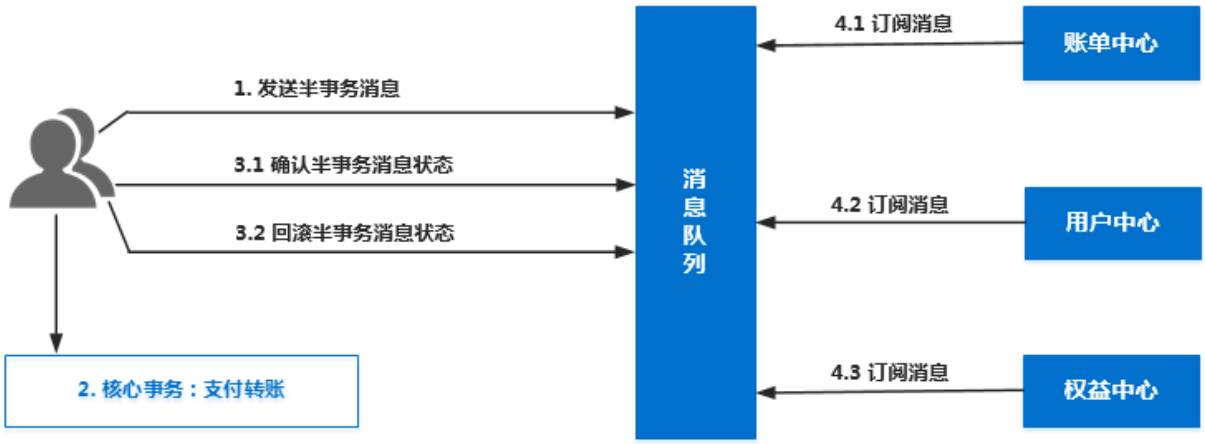
分布式事务的数据一致性

支付的流程中，用户入口在支付中心完成，账单、权益、通知系统在其他系统完成，多个系统之间的数据需要保持最终一致。

在这样的情况下，虽然实现了系统间的解藕，上游系统不需要关心下游系统的业务处理结果；但是数据一致性不好处理，如何保证下游系统状态与支付系统状态的最终一致。

事务消息处理

此时，需要有利用消息队列所提供的事务消息来实现系统间的状态数据一致性。



流程说明如下：

1. 支付中心向消息队列发送半事务消息。
    - 半事务消息发送成功，进入 2。
    - 半事务消息发送失败，支付中心不进行转账，流程结束。（最终支付中心与下游系统数据一致）
2. 支付中心开始转账流程。
    - 转账成功，进入 3.1。
    - 转账失败，进行 3.2。
3. 支付中心向消息队列发送半消息状态。
    - 3.1 提交半事务消息，产生支付成功消息，进入 4。
    - 3.2 回滚半事务消息，未产生支付成功消息，流程结束。（最终支付中心与下游系统数据一致）
4. 下游系统接收消息队列的支付成功的消息。
5. 下游系统处理相关业务逻辑。（最终支付中心与下游系统数据一致）

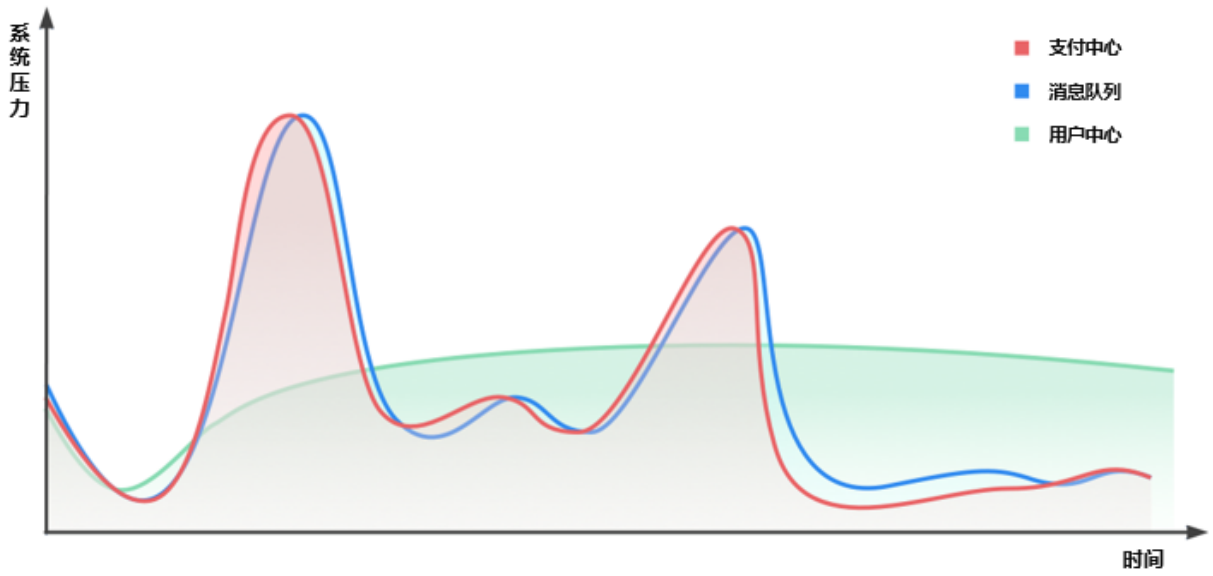
分布式事务消息的更多详细内容请参见 消息类型 > 事务消息 。

削峰填谷

流量削峰也是消息队列的常用场景，一般在秒杀或团队抢购活动中使用广泛。

还是以支付场景举例，在秒杀或团队抢购活动中，由于用户请求量较大，导致流量暴增，支付中心在处理如此大量的访问流量后，下游的应用用户中心可能无法承载海量的调用量，甚至会导致系统崩溃等问题而发生漏通知的情况。

引入消息队列后，用户中心作为消费方可以根据自身应用的能力进行消息的消费，不受大流量的影响。



1.6 基础术语

本文主要对 SOFAStack 消息队列涉及的专有名词及术语进行定义和解析，方便您更好地理解相关概念并使用消息队列。

中文	英文	释义
消息主题	Topic	消息主题，一级消息类型，通过 Topic 对消息进行分类。详情请参见 Topic 与 Tag 。
消息	Message	消息队列中信息传递的载体。
Message ID	Message ID	消息的全局唯一标识，由消息队列系统自动生成，唯一标识某条消息。
Message Key	Message Key	消息的业务标识，由消息生产者（Producer）设置，唯一标识某个业务逻辑。
消息标签	Tag	消息标签，二级消息类型，用来进一步区分某个 Topic 下的消息分类。详情请参见 Topic 与 Tag 。
消息生产者	Producer	消息生产者，也称为消息发布者，负责生产并发送消息。
Producer 实例	Producer instance	Producer 的一个对象实例，不同的 Producer 实例可以运行在不同进程内或者不同机器上。 Producer 实例线程安全，可在同一进程内多线程之间共享。

消息消费者	Consumer	消息消费者，也称为消息订阅者，负责接收并消费消息。
Consumer 实例	Consumer instance	Consumer 的一个对象实例，不同的 Consumer 实例可以运行在不同进程内或者不同机器上。一个 Consumer 实例内配置线程池消费消息。
Group	Group	一类 Producer 或 Consumer，这类 Producer 或 Consumer 通常生产或消费同一类消息，且消息发布或订阅的逻辑一致。
Group ID	Group ID	Group 的标识。
队列	Queue	每个 Topic 下会由一到多个队列来存储消息。每个 Topic 对应队列数与消息类型以及实例所处地域（Region）相关，具体的队列数可 提交工单 咨询。
集群消费	Clustering consumption	一个 Group ID 所标识的所有 Consumer 平均分摊消费消息。例如某个 Topic 有 9 条消息，一个 Group ID 有 3 个 Consumer 实例，那么在集群消费模式下每个实例平均分摊，只消费其中的 3 条消息。详情请参见 集群消费和广播消费 。
广播消费	Broadcasting consumption	一个 Group ID 所标识的所有 Consumer 都会各自消费某条消息一次。例如某个 Topic 有 9 条消息，一个 Group ID 有 3 个 Consumer 实例，那么在广播消费模式下每个实例都会各自消费 9 条消息。详情请参见 集群消费和广播消费 。
定时消息	Scheduled message	Producer 将消息发送到消息队列服务端，但并不期望这条消息立马投递，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。详情请参见 消息类型 > 定时和延时消息 。
延时消息	Delayed message	Producer 将消息发送到消息队列服务端，但并不期望这条消息立马投递，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。详情请参见 消息类型 > 定时和延时消息 。
事务消息	Transactional message	消息队列提供类似 X/Open XA 的分布事务功能，通过消息队列的事务消息能达到分布式事务的最终一致。详情请参见 消息类型 > 事务消息 。
顺序消息	Ordered message	消息队列提供的一种按照顺序进行发布和消费的消息类型，分为全局顺序消息和分区顺序消息，当前仅支持分区顺序消息。详情请参见 消息类型 > 顺序消息 。
分区顺序消息	Partitionally ordered message	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 FIFO 顺序进行发布和消费。Sharding Key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Message Key 是完全不同的概念。详情请参见 消息类型 > 顺序消息 。
消息堆积	Message accumulation	Producer 已经将消息发送到消息队列的服务端，但由于 Consumer 消费能力有限，未能在短时间内将所有消息正确消费掉，此时在消息队列的服务端保存着未被消费的消息，该状态即消息堆积。
消息过滤	Message filtering	Consumer 可以根据消息标签（Tag）对消息进行过滤，确保 Consumer 最终只接收被过滤后的消息类型。消息过滤在消息队列的服务端完成。详情请参见 消息过滤 。
消息轨迹	Message trace	在一条消息从 Producer 发出到 Consumer 消费处理过程中，由各个相关节点的时间、地点等数据汇聚而成的完整链路信息。通过消息轨迹，您能清晰定位消息从 Producer 发出，经由消息队列服务端，投递给 Consumer 的完整链路，方便定位排查问题。详情请参见 查询消息轨迹 。
重置消费位点	Reset consumption offset	以时间轴为坐标，在消息持久化存储的时间范围内（默认 3 天），重新设置 Consumer 对已订阅的 Topic 的消费进度，设置完成后 Consumer 将接收设定时间点之后由 Producer 发送到消息队列服务端的消息。详情请参见 重置消费位点 。

1.7 使用限制

SOFAStack 消息队列对某些具体指标进行了约束和规范，您在使用过程中时注意不要超过相应的限制值，以免程序出现异常。

具体的限制项和限制值请参见下表。

限制项	限制值	说明
Topic 名称长度	64 个字符	Topic 名称长度不得超过该限制，否则会导致无法发送或者订阅。
消息大小	• 普通和顺序消息：4 MB • 事务和定时/延时消息：64 KB	消息大小不得超过其类型所对应的限制，否则消息会被丢弃。
消息保存时间	3 天	消息最多保留 3 天，超过时间将自动滚动删除。
消费位点重置	3 天	支持重置消费 3 天之内任何时间点的消息。
单实例的消息收发 TPS	标准版：5000 条/秒	如需更高规格，请 提交工单。
定时/延时消息的延时时长	40 天	msg.setStartDeliverTime 参数（单位：毫秒）可设置 40 天内的任何时刻，超过 40 天消息发送将失败。

2 消息功能详解

2.1 消息类型

本文介绍 SOFAStack 消息队列各个消息类型的概念、适用场景以及使用过程中的注意事项等。

- 普通消息
- 定时和延时消息
    - 定时消息
    - 延时消息
- 顺序消息
    - 分区顺序消息
- 事务消息

普通消息

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

- TCP Java SDK 收发普通消息的示例代码
    - 发送普通消息（三种方式）
    - 发送消息（多线程）
    - 订阅消息

定时和延时消息

- 定时消息：Producer 将消息发送到消息队列服务端，但并不期望这条消息立马投递，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服务端，但并不期望这条消息立马投递，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。

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：比如在蚂蚁森林场景中，相关低碳行为触发时，会发送一条定时消息，在第二天7点定时投递给消费者，产生绿色能量；或者发送一条延时消息，24小时后产生绿色能量。

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： 收发延时消息

顺序消息

顺序消息（FIFO 消息）是消息队列提供的一种严格按照顺序来发布和消费的消息。顺序发布和顺序消费是指对于指定的一个 Topic，生产者按照一定的先后顺序发布消息；消费者按照既定的先后顺序订阅消息，即先发布

的消息一定会先被客户端接收到。

顺序消息目前支持 分区顺序消息 。

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 FIFO 顺序进行发布和消费。Sharding Key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。

- 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 FIFO 原则进行消息发布和消费的场景。
- 示例
  - 用户注册需要发送发验证码，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。

注意事项

使用顺序消息时，请注意以下几点：

- 顺序消息暂不支持广播模式。
- 建议同一个 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。
- 顺序消息不支持异步发送方式，否则将无法严格保证顺序。

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

事务消息

- 事务消息：消息队列提供类似 X/Open XA 的分布式事务功能，通过消息队列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，发送方已经成功地将消息发送到了消息队列服务端，但是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。

适用场景

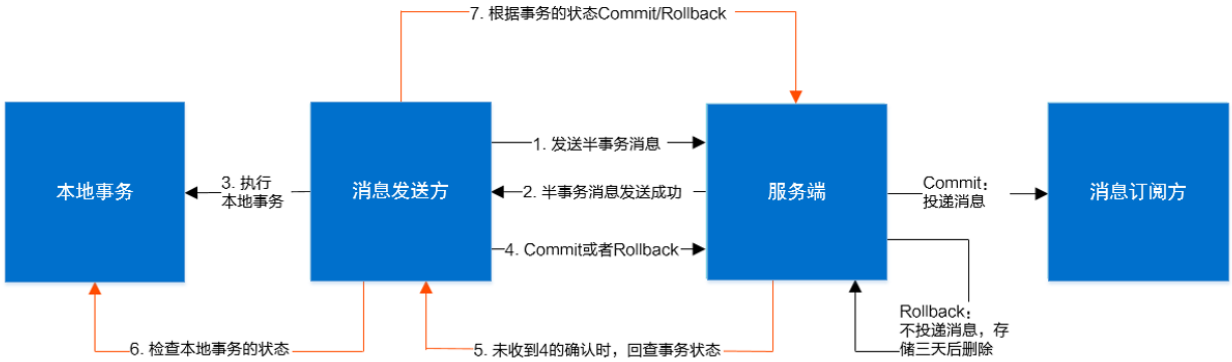
事务消息的适用场景示例：

在转账过程中，比如从支付宝转账到余额宝，两个系统之间的数据需要保持最终一致性，这时可以通过事务消

息进行处理。支付宝扣款执行前，发送一条半事务消息，扣款事务执行成功后，将消息状态更新为 Commit，余额宝系统订阅消息队列的扣款消息，做相应的存款业务处理。

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. 发送方将半事务消息发送至消息队列服务端。
2. 消息队列服务端将消息持久化成功之后，向发送方返回 Ack 确认消息已经发送成功，此时消息为半事务消息。
3. 发送方开始执行本地事务逻辑。
4. 发送方根据本地事务执行结果向服务端提交二次确认（Commit 或是 Rollback），服务端收到 Commit 状态则将半事务消息标记为可投递，订阅方最终将收到该消息；服务端收到 Rollback 状态则删除半事务消息，订阅方将不会接受该消息。

事务消息回查步骤如下：

1. 在断网或者是应用重启的特殊情况下，上述步骤 4 提交的二次确认最终未到达服务端，经过固定时间后服务端将对该消息发起消息回查。
2. 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
3. 发送方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 对半事务消息进行操作。

注意事项

1. 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，回查时消息队列服务端会根据 Group ID 去查询客户端。
2. 通过 AccessPoint.getAccessPoint().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。
3. 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，允许订阅方消费该消息。
    - 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服务端向发送方进行消息回查。

4. 可通过以下方式给每条消息设定第一次消息回查的最快时间：

```
Message message = new Message();
// 在消息属性中添加第一次消息回查的最快时间，单位秒。例如，以下设置实际第一次回查时间为 120 秒 ~ 125 秒之间
message.putUserProperties(PropertyKeyConst.CheckImmunityTimeInSeconds,"120");
// 以上方式只确定事务消息的第一次回查的最快时间，实际回查时间向后浮动 0 秒 ~ 5 秒；如第一次回查后事务仍未提交，后续每隔 5 秒回查一次
```

TCP SDK 示例代码

收发事务消息的示例代码如下：

- TCP Java SDK 收发事务消息

2.2 Topic 与 Tag

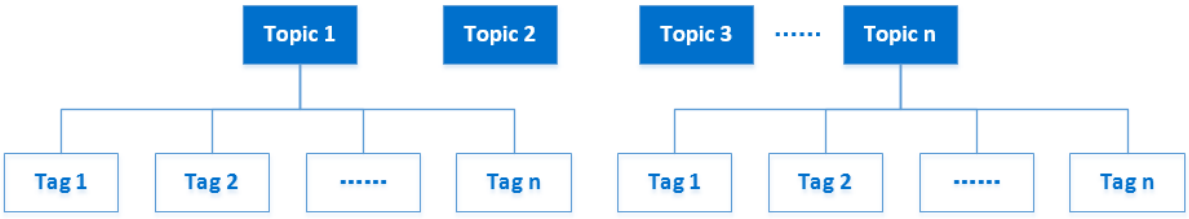
在SOFAStack消息队列中，Topic 与 Tag 都是业务上用来归类的标识，区分在于 Topic 是一级分类，而 Tag 可以理解为是二级分类。您可通过本文了解如何搭配使用 Topic 和 Tag 来实现消息过滤。

背景信息

Topic 和 Tag 的定义如下：

- **Topic**：消息主题，通过 Topic 对不同的业务消息进行分类。
- **Tag**：消息标签，用来进一步区分某个 Topic 下的消息分类，消息从生产者发出即带上的属性。

Topic 和 Tag 的关系如下图所示。



适用场景

您可能会有这样的疑问：到底什么时候该用 Topic，什么时候该用 Tag？

建议您从以下几个方面进行判断：

- 消息类型是否一致：如普通消息、事务消息、定时（延时）消息、顺序消息，不同的消息类型使用不同的 Topic，创建Topic时需要指定消息类型，无法通过 Tag 进行区分。
- 业务是否相关联：没有直接关联的消息，如支付消息、芝麻信用消息、会员消息可以使用不同的 Topic 进行区分；而同样是支付消息，付款、红包到账、转账等消息可以用 Tag 进行区分。

- 消息量级是否相当：有些业务消息虽然量小但是实时性要求高，如果跟某些万亿量级的消息使用同一个 Topic，则有可能会因为过长的等待时间而“饿死”，此时需要将不同量级的消息进行拆分，使用不同的 Topic。

总的来说，针对消息分类，您可以选择创建多个 Topic，或者在同一个 Topic 下创建多个 Tag。但通常情况下，不同的 Topic 之间的消息没有必然的联系，而 Tag 则用来区分同一个 Topic 下相互关联的消息，例如全集和子集的关系、流程先后的关系。

2.3 订阅关系一致

订阅关系一致指的是同一个消费者 Group ID 下所有消费者的处理逻辑必须完全一致。一旦订阅关系不一致，消息消费的逻辑就会混乱，甚至导致消息丢失。本文提供订阅关系不一致的示例代码，帮助您顺畅地订阅消息。

背景信息

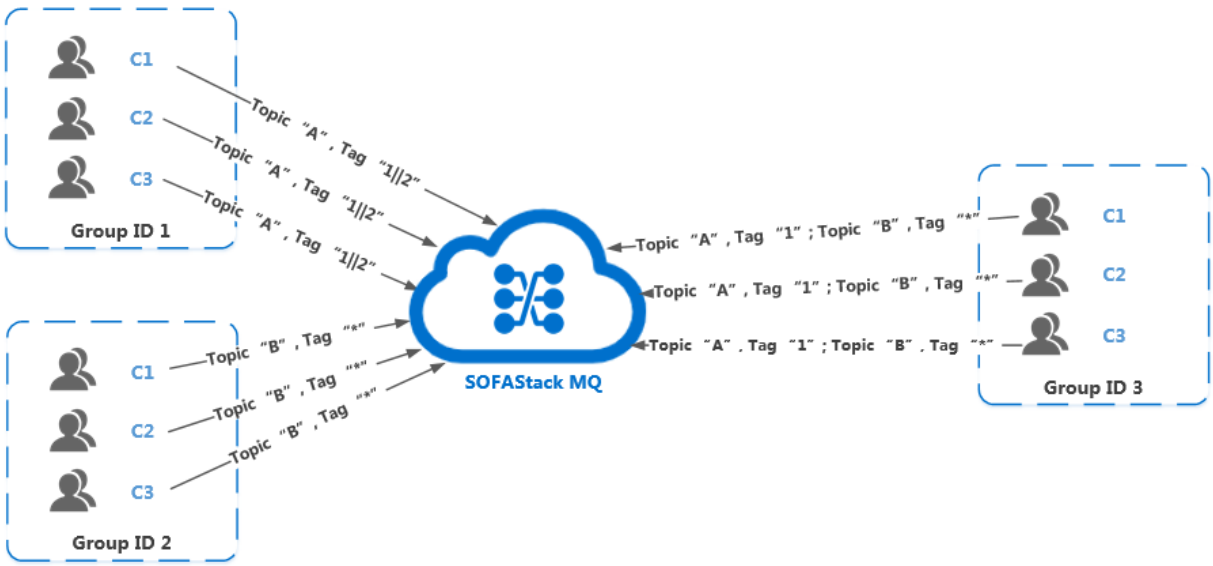
SOFAStack 消息队列里的一个消费者 Group ID 代表一个消费者群组。对于大多数分布式应用来说，一个消费者 Group ID 下通常会挂载多个消费者。

由于消息队列的订阅关系主要由 Topic + Tag 共同组成，因此，保持订阅关系一致意味着同一个消费者 Group ID 下所有的消费者需在以下两方面均保持一致：

- 订阅的 Topic 必须一致
- 订阅的 Topic 中的 Tag 必须一致

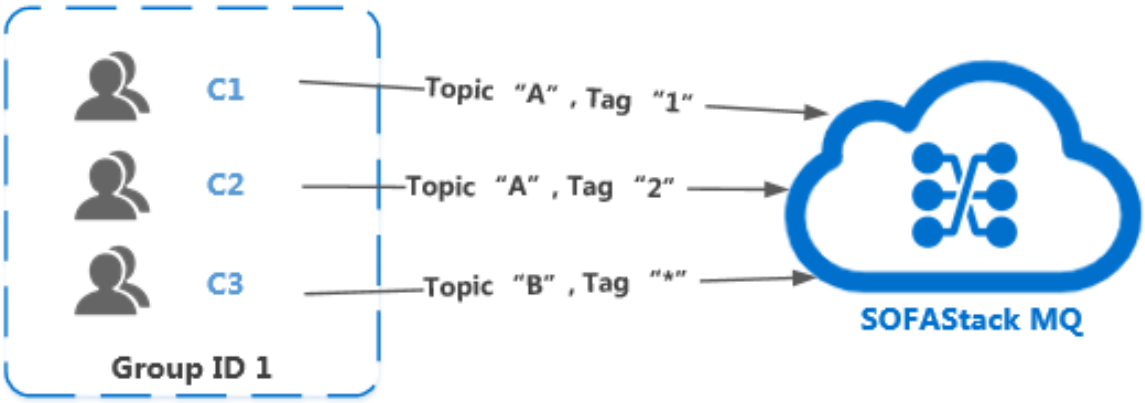
正确订阅关系图片示例

多个 Group ID 订阅了多个 Topic，并且每个 Group ID 里的多个消费者的订阅关系保持了一致。



错误订阅关系图片示例

单个 Group ID 订阅了多个 Topic，但是该 Group ID 里的多个消费者的订阅关系并没有保持一致。



错误订阅关系代码示例一

以下例子中，同一个 Group ID 下的两个消费者订阅的 Topic 不一致。

- 消费者 1-1：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID_jodie_test_1");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
consumer.subscribe("jodie_test_A","*", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

- 消费者 1-2：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID_jodie_test_1");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
consumer.subscribe("jodie_test_B","*", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

错误订阅关系代码示例二

以下例子中，同一个 Group ID 下订阅 Topic 的 Tag 不一致。消费者 2-1 订阅了 TagA，而消费者 2-2 未指定 Tag。

- 消费者 2-1：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID_jodie_test_2");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
consumer.subscribe("jodie_test_A","TagA", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

- 消费者 2-2：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID_jodie_test_2");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
consumer.subscribe("jodie_test_A","*", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

错误订阅关系代码示例三

此例中，错误的原因如下所述：

- 同一个 Group ID 下订阅 Topic 个数不一致。

  同一个 Group ID 下订阅 Topic 的 Tag 不一致。

  消费者 3-1：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID_jodie_test_3");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
consumer.subscribe("jodie_test_A","TagA", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
consumer.subscribe("jodie_test_B","TagB", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

- 消费者 3-2：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID_jodie_test_3");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
consumer.subscribe("jodie_test_A","TagB", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

2.4 集群消费和广播消费

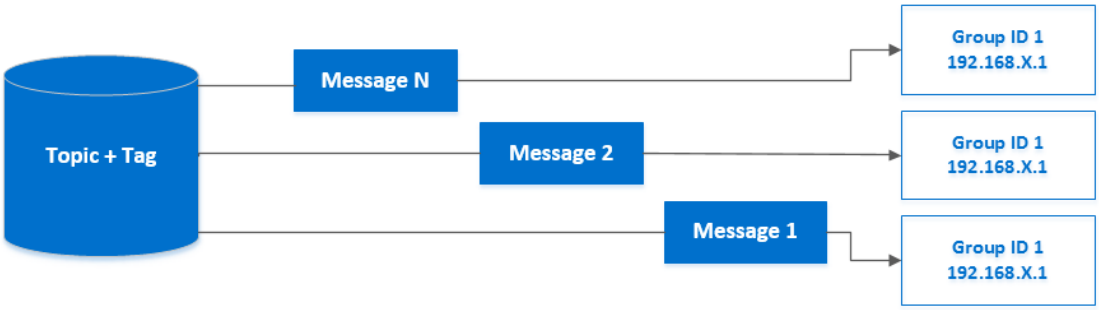
本文介绍 SOFAStack 消息队列的集群消费和广播消费的基本概念、适用场景以及注意事项。

消息队列是基于发布/订阅模型的消息系统。消费者，即消息的订阅方订阅关注的 Topic，以获取并消费消息。由于消费者应用一般是分布式系统，以集群方式部署，因此消息队列约定以下概念：

- 集群：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）。详情请参见 订阅关系一致 。
- 集群消费：当使用集群消费模式时，消息队列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。
- 广播消费：当使用广播消费模式时，消息队列会将每条消息推送给集群内所有注册过的消费者，保证消息至少被每个消费者消费一次。

集群消费模式

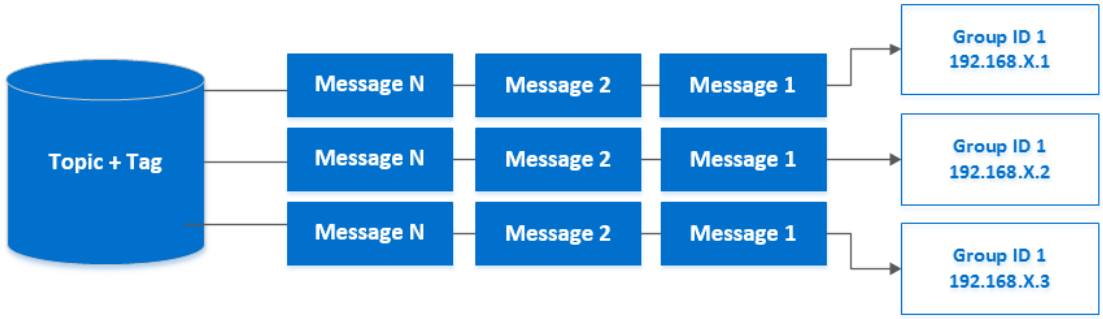
- 适用场景
  适用于消费端集群化部署，每条消息只需要被处理一次的场景。此外，由于消费进度在服务端维护，可靠性更高。具体消费示例如下图所示。



- 注意事项
  - 集群消费模式下，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。
  - 集群消费模式下，不保证每一次失败重投的消息路由到同一台机器上。

广播消费模式

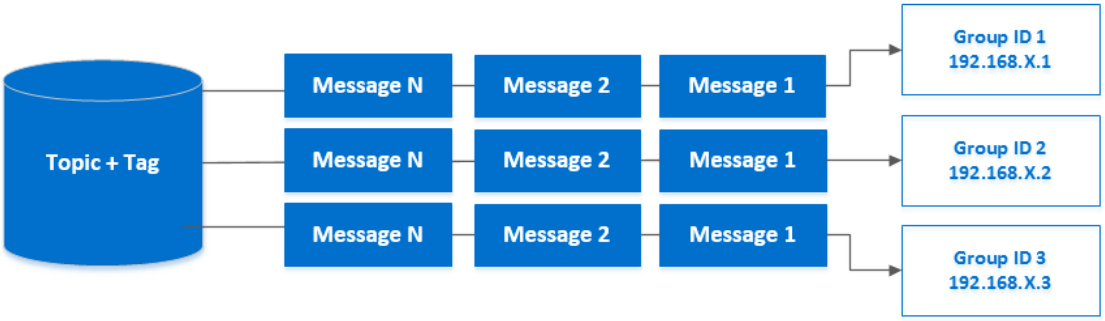
- 适用场景
  适用于消费端集群化部署，每条消息需要被集群下的每个消费者处理的场景。具体消费示例如下图所示。



- 注意事项
  - 广播消费模式下不支持顺序消息。
  - 广播消费模式下不支持重置消费位点。
  - 每条消息都需要被相同订阅逻辑的多台机器处理。
  - 消费进度在客户端维护，出现重复消费的概率稍大于集群模式。
  - 广播模式下，消息队列保证每条消息至少被每台客户端消费一次，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。
  - 广播模式下服务端不维护消费进度，所以消息队列控制台不支持消息堆积查询、消息堆积报警和订阅关系查询功能。

使用集群模式模拟广播

- 适用场景
  适用于每条消息都需要被多台机器处理，每台机器的逻辑可以相同也可以不一样的场景。具体消费示例如下图所示。



如果业务需要使用广播模式，也可以创建多个 Group ID，用于订阅同一个 Topic。

- 注意事项
    - 消费进度在服务端维护，可靠性高于广播模式。
    - 对于一个 Group ID 来说，可以部署一个消费者实例，也可以部署多个消费者实例。当部署多个消费者实例时，实例之间又组成了集群模式（共同分担消费消息）。假设 Group ID 1 部署了三个消费者实例 C1、C2、C3，那么这三个实例将共同分担服务器发送给 Group ID 1 的消息。同时，实例之间订阅关系必须保持一致。详见 订阅关系一致 。

更多信息

集群消费模式和广播消费模式的具体配置方法，请参见以下文档：

- TCP 协议 Java SDK： 订阅消息

2.5 消息重试

本文介绍 SOFAStack 消息队列的消息重试机制和配置方式。

顺序消息的重试

对于顺序消息，当消费者消费消息失败后，消息队列会自动不断地进行消息重试（每次间隔时间为 1 秒），这时，应用会出现消息消费被阻塞的情况。因此，建议您使用顺序消息时，务必保证应用能够及时监控并处理消费失败的情况，避免阻塞现象的发生。

无序消息的重试

对于无序消息（普通、定时、延时、事务消息），当消费者消费消息失败时，您可以通过设置返回状态达到消息重试的结果。

无序消息的重试只针对集群消费方式生效；广播方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。

> 说明：以下内容都只针对无序消息生效。

重试次数

消息队列默认允许每条消息最多重试 16 次，每次重试的间隔时间如下：

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

如果消息重试 16 次后仍然失败，消息将不再投递。如果严格按照上述重试时间间隔计算，某条消息在一直消费失败的前提下，将会在接下来的 4 小时 46 分钟之内进行 16 次重试，超过这个时间范围消息将不再重试投递。

说明：一条消息无论重试多少次，这些重试消息的 Message ID 不会改变。

配置方式

消费失败后，重试配置方式

集群消费方式下，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）：

```
- 返回 Action.ReconsumeLater（推荐）
- 返回 Null
- 抛出异常
```

示例代码：

```
public class MessageListenerImpl implements MessageListener {

@Override
public Action consume(Message message, ConsumeContext context) {
//方法 3：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 exceotion");
}
}
```

消费失败后，无需重试的配置方式

集群消费方式下，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

消息队列允许 Consumer 启动的时候设置最大重试次数，重试时间间隔将按照以下策略：

- 最大重试次数小于等于 16 次，则重试时间间隔同上表描述。
- 最大重试次数大于 16 次，超过 16 次的重试时间间隔均为每次 2 小时。

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
properties.put(PropertyKeyConst.MAX_RECONSUME_TIMES,"20");
Consumer consumer = OMS.builder().driver("sofamq").createConsumer(properties);
```

说明：

- 消息最大重试次数的设置对相同 Group ID 下的所有 Consumer 实例有效。
- 如果只对相同 Group ID 下两个 Consumer 实例中的其中一个设置了 MaxReconsumeTimes，那么该配置对两个 Consumer 实例均生效。
- 配置采用覆盖的方式生效，即最后启动的 Consumer 实例会覆盖之前的启动实例的配置。

获取消息重试次数

消费者收到消息后，可按照以下方式获取消息的重试次数：

```
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;
}
}
```

2.6 消息过滤

本文描述SOFAStack 消息队列的消费者如何根据 Tag 在消息队列服务端完成消息过滤，以确保消费者最终只

消费到其关注的消息类型。

Tag，即消息标签，用于对某个 Topic 下的消息进行分类。消息队列的生产者在发送消息时，已经指定消息的 Tag，消费者需根据已经指定的 Tag 来进行订阅。

示例代码

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 用符号 * 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 || 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;
}
});
```

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，详情请参见 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，详情请参见 Topic 与 Tag 。

2.7 消息幂等

为了防止消息重复消费导致业务处理异常，SOFAStack 消息队列的消费者在接收到消息后，有必要根据业务上的唯一 Key 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。

什么是消息幂等

当出现消费者对某条消息重复消费的情况时，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，那么这整个过程就实现可消息幂等。

例如，在支付场景下，消费者消费扣款消息，对一笔订单执行扣款操作，扣款金额为 100 元。如果因网络不稳定等原因导致扣款消息重复投递，消费者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 100 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费幂等。

适用场景

在互联网应用中，尤其在网络不稳定的情况下，消息队列的消息有可能会出现重复。如果消息重复会影响您的业务处理，请对消息做幂等处理。

消息重复的场景如下：

- 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。 如果此时生产者意识到消息发送失败并尝试再次发送消息，消费者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，消息已投递到消费者并完成业务处理，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，消息队列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，消费者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、Broker 重启以及消费者应用重启）
  当消息队列的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息。

处理方法

因为 Message ID 有可能出现冲突（重复）的情况，所以真正安全的幂等处理，不建议以 Message ID 作为处理依据。最好的方式是以业务唯一标识作为幂等处理的关键依据，而业务的唯一标识可以通过消息 Key 设置。

以支付场景为例，可以将消息的 Key 设置为订单号，作为幂等处理的依据。具体代码示例如下：

```
Message message = new Message();
message.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
```

消费者收到消息时可以根据消息的 Key，即订单号来实现消息幂等：

```
consumer.subscribe("ons_test","*", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
String key = message.getKey()
// 根据业务唯一标识的 Key 做幂等处理
}
});
```

3 快速入门

3.1 快速入门

本文将引导您快速体验 SOFAStack 消息队列，从创建资源、配置接入点到使用 SDK 收发消息。具体操作步骤如下：

1. 创建资源
   - 创建工作空间
   - 创建 Topic
   - 创建 Group ID
   - 创建 AccessKey
2. 获取接入配置
3. 发送消息
4. 订阅消息

创建资源

注意事项

在使用 SOFAStack 消息队列时，请注意以下网络访问限制：

- Topic 和 Group ID 需创建在同一个地域（Region）下的同一个工作空间中才能互通。例如，当某 Topic 创建在华东 1（杭州）下的工作空间 A 中，那么该 Topic 只能被在华东 1（杭州）下的工作空间 A 中创建的 Group ID 对应的生产端和消费端访问。
- 目前不支持公网访问，生产端和消费端需要部署在相同地域的 ECS 上，或者保证网络联通。

创建工作空间

要使用消息队列，您需要确保 SOFAStack 控制台已创建至少一个工作空间。如 SOFAStack 未创建工作空间或您需要创建一个新的工作空间，可参见 管理工作空间 > 添加工作空间 。创建好工作空间后，将为您自动创建一个消息队列实例。

创建 Topic

Topic 是消息队列里对消息的一级归类。消息生产者将消息发送到一个 Topic，而消息消费者则通过订阅该 Topic 来获取和消费消息。

1. 在控制台左侧导航栏，点击 **Topic 管理**。
2. 在 Topic 管理页面上方，点击 **创建 Topic** 按钮。
3. 在 **创建 Topic** 对话框中，输入 Topic 名称，选择该 Topic 对应的消息类型，填写该 Topic 的备注内容。
4. 点击 **确定**，完成创建，您创建的 Topic 将出现在 Topic 列表中。

消息类型的更多信息，请参见 消息类型 。

创建 Group ID

创建完 Topic 后，您需要为消息的消费者（或生产者）创建客户端 ID ，即 Group ID 作为标识。

Group ID 和 Topic 的关系是 N：N，即一个消费者可以订阅多个 Topic，同一个 Topic 也可以被多个消费者订阅；一个生产者可以向多个 Topic 发送消息，同一个 Topic 也可以接收来自多个生产者的消息。

> **说明**：消费者必须有对应的 Group ID，生产者不做强制要求。

1. 在控制台左侧导航栏，点击 **Group 管理**。
2. 在 Group 管理页面上方，点击 创建 **Group ID**。
3. 在创建 Group ID 对话框中，输入 Group ID 和描述，然后点击 **确认**。

创建 AccessKey

阿里云 AccessKey 用于收发消息时进行账户鉴权。

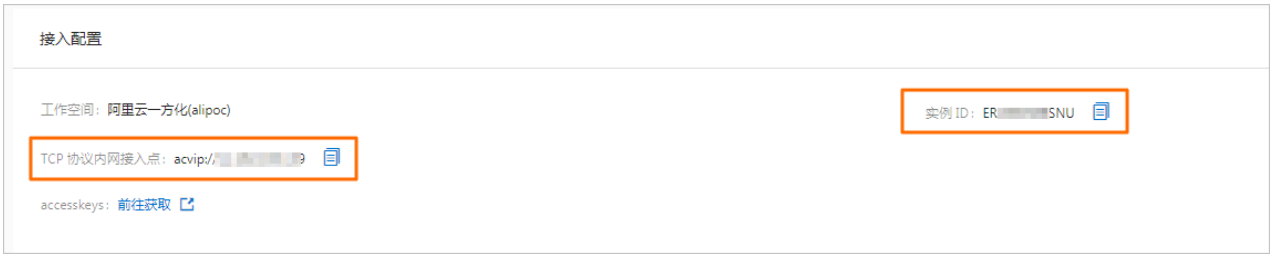
在调用 SDK 发送和订阅消息的时候，除了需要指定创建的 Topic 和 Group ID 以外，还需输入您在 RAM 控制台创建的身份验证信息，即 AccessKey。AccessKey 的信息包含 AccessKeyId 和 AcessKeySecret。

创建 AccessKey 的具体步骤，参见 创建 AccessKey。

获取接入配置

在控制台创建好资源后，您需通过控制台获取工作空间的接入点。在收发消息时，您需要为生产端和消费端配置该接入点，以此接入某个具体工作空间或地域的服务。

1. 进入消息队列控制台页面，选择地域和工作空间。
2. 在概览页底部的 **接入配置** 中，即可找到 **TCP 协议内网接入点** 及 **实例 ID**。
3. 将该 **TCP 协议内网接入点** 配置到您客户端的 SDK 代码的 **ENDPOINT** 参数。
4. 将 **实例 ID** 配置到您客户端的SDK代码的 **INSTANCEID** 参数。



发送消息

您可以通过控制台发送测试消息或通过调用 TCP Java SDK 发送消息。

发送测试消息

用于快速验证 Topic 资源的可用性，主要用作测试。

1. 在控制台左侧导航栏，点击 **Topic 管理**。
2. 在 Topic 管理页面，找到您刚刚创建的 Topic，单击右侧操作列的 **发送测试消息**。
3. 在 **发送测试消息** 对话框中的 **消息体** 一栏，输入消息的具体内容，点击 **确定**。控制台即会返回消息发送成功通知以及相应的 Message ID。

调用 SDK 发送消息

通过 Maven 方式引入依赖。Java SDK 的最新版本号，可参见 SDK 版本说明 。

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofamq-client-all</artifactId>
<version>"XXX"</version>
//设置为 Java SDK 的最新版本号
</dependency>
<repositories>
<repository>
<id>antcloudrelease</id>
<name>Ant Cloud</name>
<url>http://mvn.cloud.alipay.com/nexus/content/groups/open</url>
</repository>
</repositories>
```

根据以下说明设置相关参数，运行示例代码：

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// 鉴权用 AccessKeyId，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// 鉴权用 AccessKeySecret，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
// 您在控制台创建的 Group ID
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);

producer.start();

Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
SendResult sendResult = producer.send(message);
System.out.println(sendResult);
}
}
```

3. 消息发送后，您可以在控制台查看消息发送状态，步骤如下：
    - 在控制台左侧导航栏，选择 **消息查询 > 按 Message ID 查询**。
    - 在搜索框中输入发送消息后返回的 Message ID，点击 **搜索** 查询消息发送状态。

      储存时间 表示消息队列服务端存储这条消息的时间。如果查询到此消息，表示消息已经成功发送到服务端。

      > **注意**：此步骤演示的是第一次使用消息队列的场景，此时消费者从未启动过，所以消息状态显示暂无消费数据。要启动消费者并进行消息订阅请继续下一步操作订阅消息。更多消息状态请参见 消息查询 和 查询消息轨迹 。

订阅消息

消息发送成功后，需要启动消费者来订阅消息。

调用 TCP Java SDK 订阅消息。
您可以运行以下示例代码来启动消费者，并测试订阅消息的功能。请按照说明正确设置相关参数。

```
import java.util.Properties;
import com.alipay.sofa.sofamq.client.PropertyKeyConst;
import io.openmessaging.api.Action;
import io.openmessaging.api.ConsumeContext;
import io.openmessaging.api.Consumer;
import io.openmessaging.api.Message;
import io.openmessaging.api.MessageListener;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// 鉴权用 AccessKeyId，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// 鉴权用 AccessKeySecret，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
// 您在控制台创建的 Group ID
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");

Consumer consumer = accessPoint.createConsumer(properties);
consumer.subscribe("YOUR_TOPIC","YOUR_TAG", new MessageListener() {
@Override
public Action consume(Message message, ConsumeContext context) {
System.out.println(new String(message.getBody()));
return Action.CommitMessage;
}
});
consumer.start();
}
}
```

2. 完成上述步骤后，您可以在控制台查看消费者是否启动成功，即消息订阅是否成功。
    - 在消息队列控制台左侧导航栏，点击 **Group 管理**。
    - 找到要查看的 Group ID，点击该 Group ID 所在行操作列的 **订阅关系**。如果 **是否在线** 显示为 **是**，且订阅关系一致，则说明订阅成功。否则说明订阅失败。

3.2 接入点设置说明

本文提供对应地域的 TCP 内网接入点的详细说明和配置步骤。

背景信息

如需使用SOFAStack 消息队列来收发消息，则需在使用 TCP 协议下的 SDK 时，将相应参数配置为您实际使用的资源信息。需特别注意的是，以下参数所对应的资源应处于同一地域：

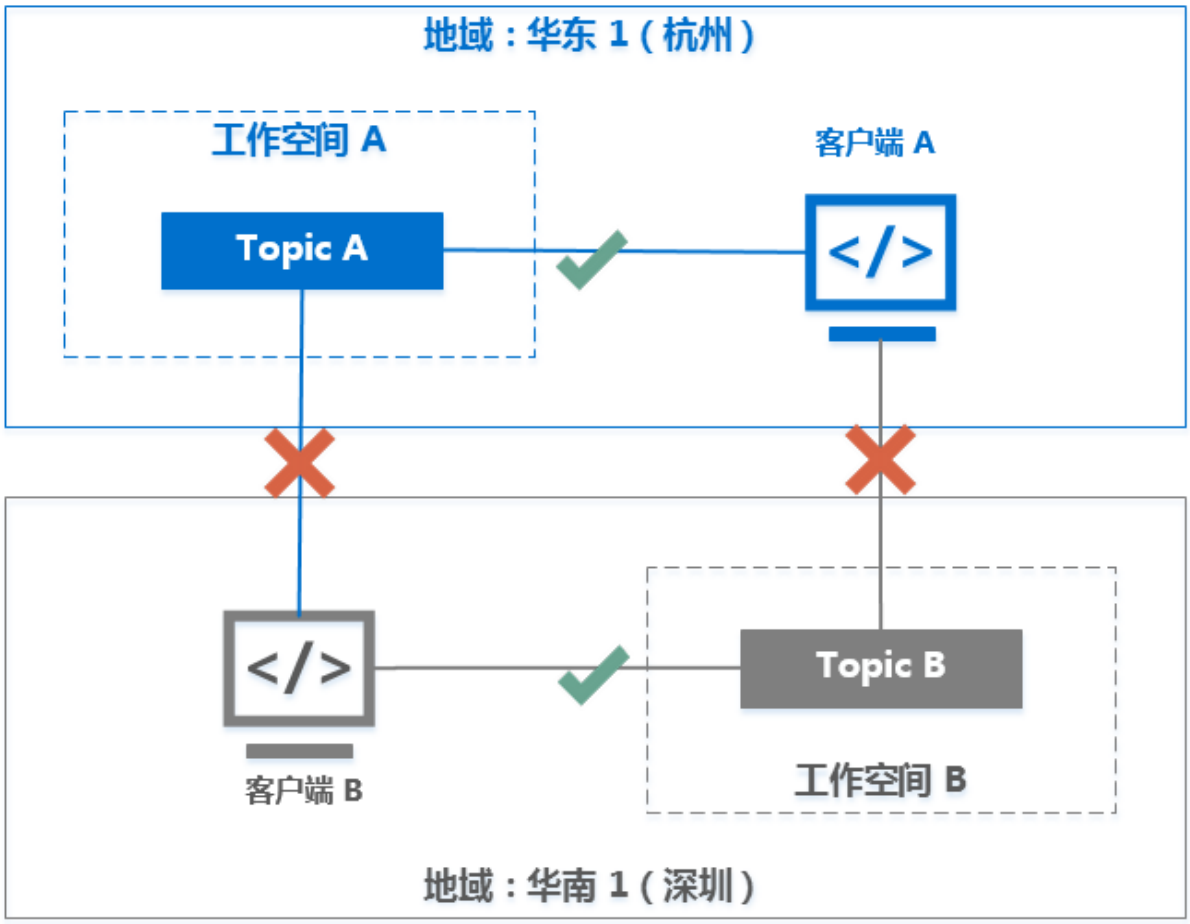
- GROUP_ID：您在消息队列控制台上创建的 Group ID。
- TOPIC：您在消息队列控制台上创建的 Topic。
- ENDPOINT：您从消息队列控制台的 **概览 > 接入配置** 获取的接入点。
- INSTANCE_ID：您从消息队列控制台的 **概览 > 接入配置** 获取的实例 ID。

例如您在华东1（杭州）地域创建了一个消息队列工作空间 A，并在此地域下创建了 Topic A 和 Group ID A，供部署在该地域的应用客户端 A 使用；同时，您也在另一个地域，如华南1（深圳）创建了另一个消息队列工作空间 B，并在此地域下创建了 Topic B 和 Group ID B，供部署在该地域的应用客户端 B 使用。需特别注意的以下几点：

- 针对 Topic A，您只能通过在华东1（杭州）地域的、归属于 Group ID A 的客户端 A 向其发送或接收消息；
- 针对 Topic B，您只能通过在华南1（深圳）地域的、归属于 Group ID B 的客户端 B 向其发送或接收消息。

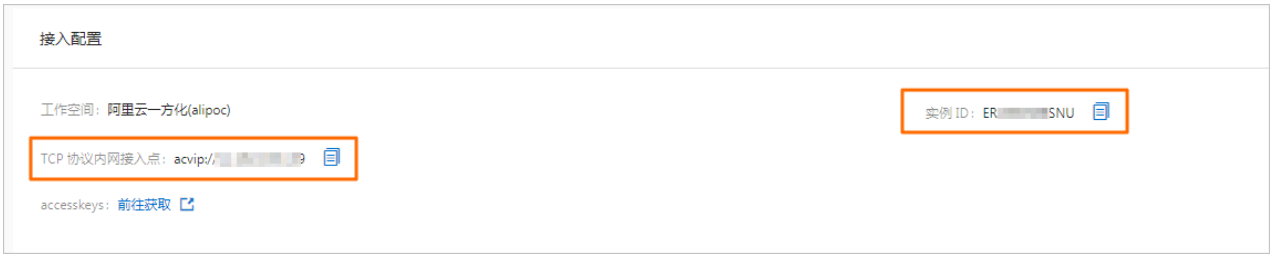
> **注意**：切忌将 Topic A 通过跨地域的方式，与在华南1（深圳）地域的、归属于 Group ID B 的客户端 B 通信，或将 Topic B 通过跨地域的方式，与在华东1（杭州）地域的、归属于 Group ID A 的客户端 A 通信。

具体对应关系如下图所示。



操作步骤

1. 进入消息队列控制台，选择地域和工作空间。
2. 在概览页的接入配置可以找到 TCP 协议接入点和实例 ID 信息。
3. 将该 **TCP 协议内网接入点** 配置到客户端 SDK 代码的 **ENDPOINT** 参数。
4. 将该 **实例 ID** 配置到客户端 SDK 代码的 **INSTANCE_ID** 参数。



4 JAVA SDK 参考

4.1 SDK 版本说明

本文介绍 SOFAStack 消息队列 Java SDK sofamq-client-all 的各个版本信息，包含 SDK 的发布时间、下载链接以及更新点等。

3.0.1 (2020-02-06)

版本号	发布时间	下载链接
3.0.1	2020-02-06	sofamq-client-all 3.0.1

修复

- 修复事务消息轨迹事务状态显示问题。

3.0.0 (2020-01-09)

版本号	发布时间	下载链接
3.0.0	2020-01-09	sofamq-client-all 3.0.0

新特性

- 新增 Producer，支持同步发送、oneway 发送、异步发送。
- 新增 OrderProducer，支持按 shardingKey 顺序发送。
- 新增 TransactionProducer，支持发送事务消息。
- 新增 Consumer，支持以 Tag 过滤、SQL 过滤消费消息。
- 新增 BatchConsumer，支持以 Tag 过滤、SQL 过滤批量消费消息。
- 新增 OrderConsumer，支持以 Tag 过滤、SQL 过滤顺序消费消息。
- 新增 OpenMessaging 2.0.0-pubsub 支持。

4.2 Demo 工程

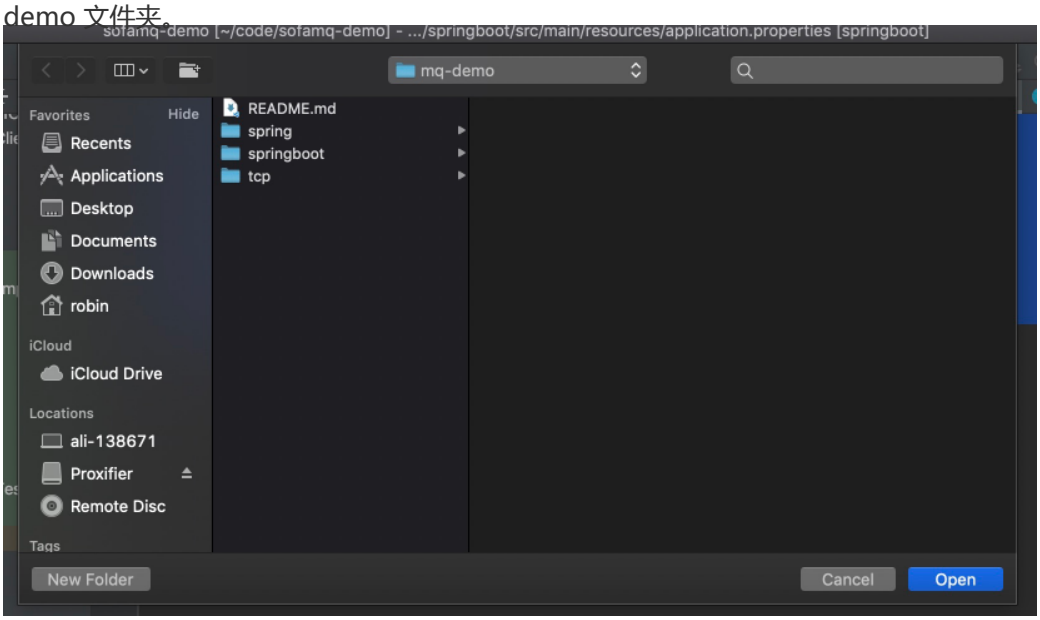
本文以 TCP 协议下的 Java 为例，提供操作示例帮助您从零开始搭建消息队列测试工程，提供Spring 和纯JAVA两种使用方式。Demo 工程包含普通消息、顺序消息、事务消息和定时（延时）消息的测试代码，以及配置。

前提条件

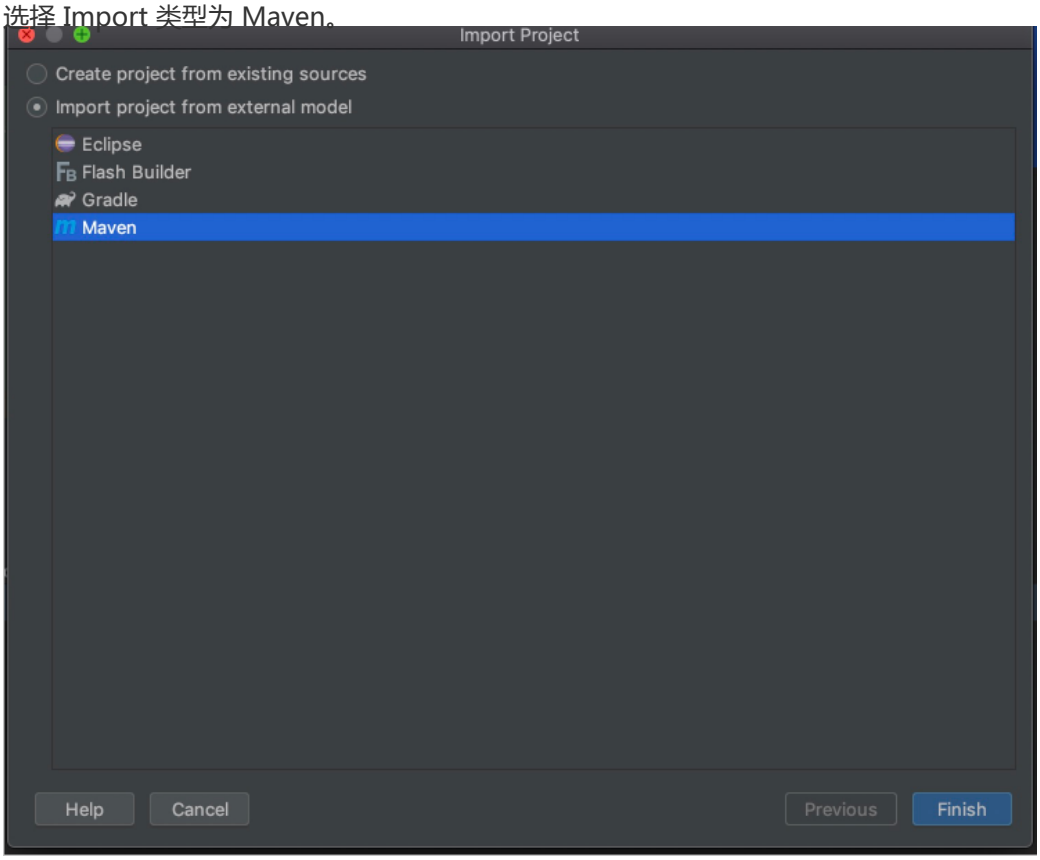
- 安装 IDE。
  您可以使用 IntelliJ IDEA 或者 Eclipse，本文以 IntelliJ IDEA 为例。
  在 https://www.jetbrains.com/idea/ 下载IntelliJ IDEA Ultimate 版本，并参考 IntelliJ IDEA 说明进行安装。
- 下载 Demo 工程。
  在 https://github.com/sofastack-guides/sofamq-demo 下载 Demo 工程到本地，然后解压即可看到本地新增了 sofamq-demo 文件夹。
- 下载安装 JDK。

配置 Demo 工程

1. 将 Demo 工程文件导入 IntelliJ IDEA
   - 在 IntelliJ IDEA 界面，选择 **New > Project From Existing Sources...**，选择 sofamq-demo 文件夹。



   - 选择 Import 类型为 Maven。



   - 默认点击 **Next**，直到导入完成。Demo 工程需要加载依赖的 JAR 包，因此导入过程需要等待 2-3 分钟。

2. 创建资源

您需要先到控制台创建所需资源，包括消息队列的工作空间、Topic、Group ID（GID），以及鉴权需要的 AccessKey（AK）。
更多详细信息和操作指导，请参见 快速入门 > 创建资源 。

3. 配置 Demo
   您需将在步骤二中创建好的资源信息配置到：MqConfig 类和 application.properties。
   - 按以下说明配置 MqConfig 类，适用于非Spring Boot使用方式
     - public static final String TOPIC = “刚才创建的 Topic” ；
     - public static final String GROUP_ID = “刚才创建的 Group ID” ；
     - public static final String ACCESS_KEY = “您的阿里云账号的 AccessKeyId” ；
     - public static final String SECRECT_KEY = “您的阿里云账号的 AccessKeySecret” ；
     - public static final String INSTANCE = “您的实例ID，可在控制台概览页底部接入配置获取实例ID”；
     - public static final String TAG = “您发布订阅时使用的TAG，如没有可为空”；
     - public static final String ENDPOINT = “您的 TCP 接入点，可在控制台概览页底部接入配置获取 TCP 协议接入点 “；

       说明：创建 AccessKey（包括 AccessKeyId 和 AccessKeySecret）的具体步骤，请参见 创建 AccessKey。

   - 配置 application.properties，适用于Spring Boot使用方式

```
#启动测试之前请替换如下 XXX 为您的配置
sofamq.accessKey=XXX
sofamq.secretKey=XX
sofamq.endpoint=XXX
sofamq.instanceId=XXX
sofamq.topic=XXX
sofamq.groupId=XXX
sofamq.tag=*
```

以 Main 方式运行 Demo

1. 发送消息
   - 发送普通消息：
     - 以纯 Java 方式发送普通消息：运行 SimpleProducer 类。
     - 以 Spring 方式发送普通消息：运行 ProducerClient 类。
   - 发送事务消息：运行 SimpleTransactionProducer 类。
     LocalTransactionCheckerImpl 类为本地事务 check 接口类，用于校验事务。详情请参见 收发事务消息 。
   - 发送顺序消息：运行 SimpleOrderProducer 类。

此方式下，消息发布和消费都按顺序进行。详情请参见 收发顺序消息 。

- 发送定时（延时）消息：运行 SimpleDelayProducer 类发送消息。延时 3 秒后投递。
  您也可以指定一个精确的投递时间，最长定时时间为 40 天。
  登录消息队列控制台，在左侧导航栏选择消息查询 > 按 Topic 查询，选择 Topic 名称进行查询。可以看见消息已经发送至 Topic。

2. 接收消息
   - 接收普通消息：
     - 以纯 Java 方式接收普通消息：运行 SimpleConsumer 类。
     - 以 Spring 方式接收普通消息：运行 ConsumerClient 类。
   - 接收事务消息：运行 SimpleConsumer 类。
   - 接收顺序消息：运行 SimpleOrderConsumer 类。
   - 接收定时（延时）消息：运行 SimpleConsumer 类。
     可以看到消息被接收打印的日志。因为有初始化，所以需等待几秒，在生产环境中不会经常初始化。
     从消息队列控制台进入 **Group 管理 > 消费者状态**，可以看到启动的消费端已经在线，并且订阅关系一致。

更多信息

- Spring 集成
- 发送普通消息（三种方式）
- 发送消息（多线程）

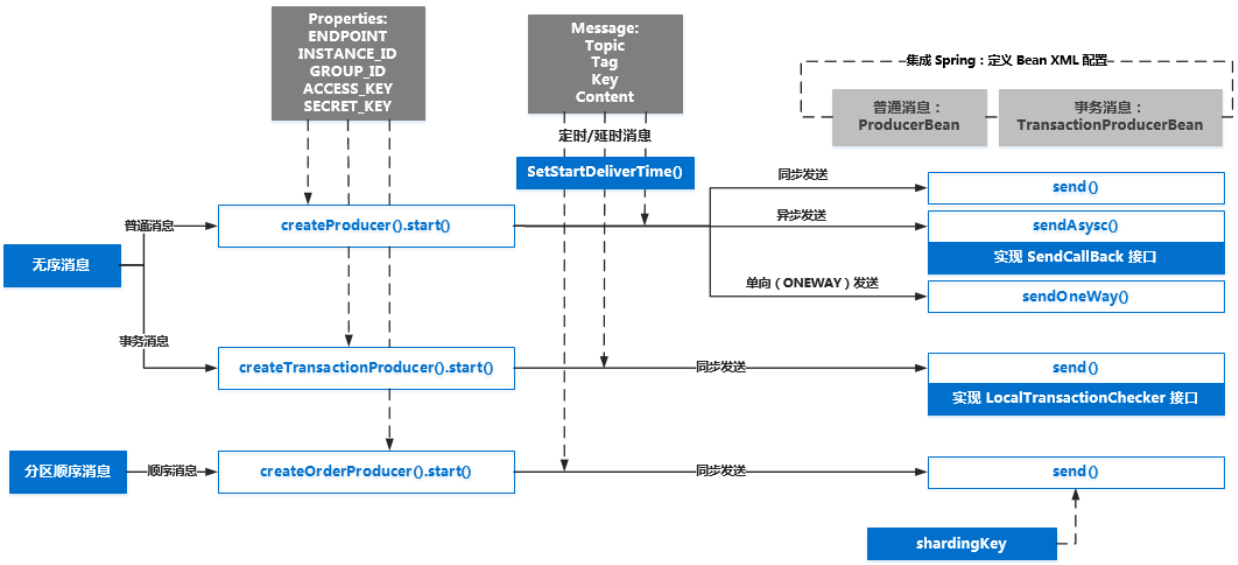
4.3 接口和参数说明

SOFAStack 消息队列提供 Java SDK 实现消息发送与订阅，您可通过本文了解消息发送和订阅相关接口的参数说明。

通用参数

参数名	参数说明
ENDPOINT	设置 TCP 协议接入点，从消息队列控制台的 **概览 > 接入配置** 获取
INSTANCE_ID	设置 实例ID，从消息队列控制台的 **概览 > 接入配置** 获取
ACCESS_KEY	您在阿里云账号管理控制台中创建的 AccessKeyId，用于身份认证
SECRET_KEY	您在阿里云账号管理控制台中创建的 AccessKeySecret，用于身份认证

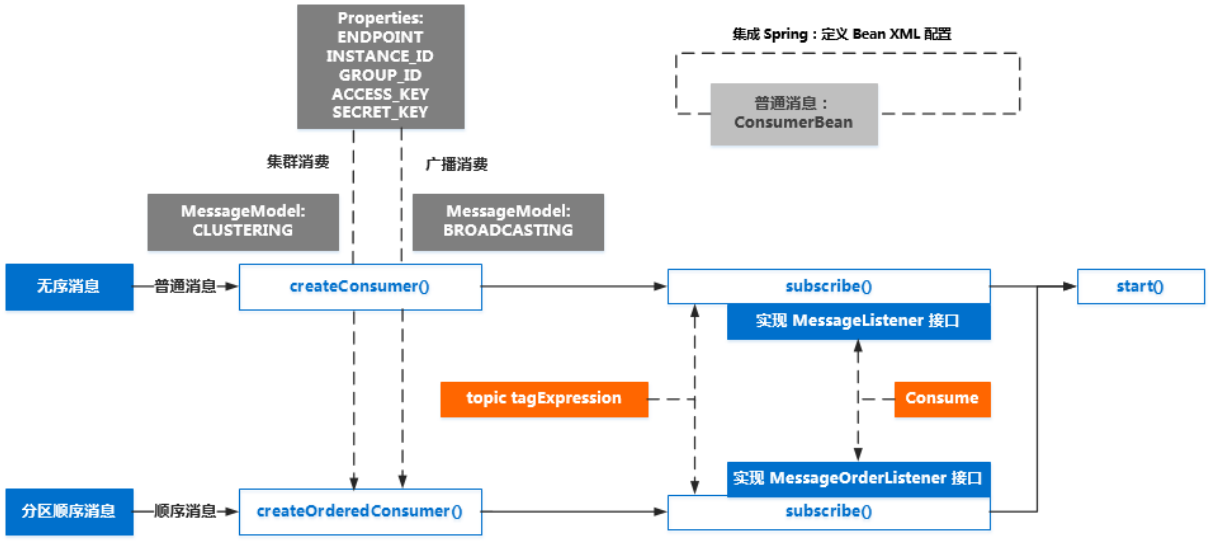
消息发送接口



消息发送参数

参数名	参数说明
SEND_MSG_TIMEOUT_MILLIS	设置消息发送的超时时间，单位：毫秒，默认值：3000
shardingKey（顺序消息）	顺序消息中用来计算不同分区的值

消息订阅接口



消息订阅参数

参数名	参数说明
GROUP_ID	您在消息队列控制台上创建的 Group ID，详情参见 名词解释
MESSAGE_MODEL	设置 Consumer 的消费模式，取值说明如下： CLUSTERING（默认值）：表示集群消费 BROADCASTING：表示广播消费

CONSUME_THREAD_NUMS	设置 Consumer 的消费线程数，默认值：20
MAX_RECONSUME_TIMES	设置消息消费失败的最大重试次数，默认值：16
CONSUME_TIMEOUT	设置每条消息消费的最大超时时间，超过设置时间则被视为消费失败，等下次重新投递再次消费。每个业务需要设置一个合理的值，单位：分钟。 默认值：15
SUSPEND_TIME_MILLIS（顺序消息）	只适用于顺序消息，设置消息消费失败的重试间隔时间
MAX_CACHED_MESSAGE_AMOUNT	客户端本地的最大缓存消息数据，默认值：1000；单位：条
MAX_CACHED_MESSAGE_SIZE_IN_MI_B	客户端本地的最大缓存消息大小，取值范围：16 MB ~ 2 GB；默认值：512 MB

更多信息

消息收发代码示例

- 发送普通消息（三种方式）
- 收发顺序消息
- 收发延时消息
- 收发事务消息
- 订阅消息

4.4 准备环境

在运行 Java 代码收发消息前，您需按照本文提供的步骤来准备环境。

操作步骤

1. 通过 Maven 方式引入依赖。Java SDK 的最新版本号，可参见 SDK 版本说明 。

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofamq-client-all</artifactId>
<version>"XXX"</version>
//设置为 Java SDK 的最新版本号
</dependency>
<repositories>
<repository>
<id>antcloudrelease</id>
<name>Ant Cloud</name>
<url>http://mvn.cloud.alipay.com/nexus/content/groups/open</url>
</repository>
</repositories>
```

2. 代码里涉及到的资源信息，例如 Topic 和 Group ID 等，需要到控制台上创建。具体操作请参见 快速入门 。

后续步骤

日志配置 。

更多信息

按照以上步骤准备好环境后，即可按需收发消息。

- 发送普通消息（三种方式）
- 发送消息（多线程）
- 收发顺序消息
- 收发事务消息
- 收发延时消息
- 订阅消息

4.5 日志配置

客户端日志用于记录客户端运行过程中的异常，帮助快速定位和修复问题。本文介绍 SOFAStack 消息队列的客户端日志的打印方式，以及默认和自定义配置。

打印客户端日志

消息队列的 TCP Java SDK 基于 SLF4J 接口编程。

依赖 log4j 或 logback 作为日志实现的示例代码如下所示。

- **方式一：依赖 log4j 作为日志实现**

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>jcl-over-slf4j</artifactId>
<version>1.7.7</version>
</dependency>
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-log4j12</artifactId>
<version>1.7.7</version>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
```

- **方式二：依赖 logback 作为日志实现**

```
<dependency>
```

```
<groupId>ch.qos.logback</groupId>
<artifactId>logback-core</artifactId>
<version>1.1.2</version>
</dependency>
<dependency>
<groupId>ch.qos.logback</groupId>
<artifactId>logback-classic</artifactId>
<version>1.1.2</version>
</dependency>
```

> **说明**：应用中同时依赖 log4j 和 logback 的日志实现会造成日志冲突导致客户端日志打印混乱。确保应用只依赖其中一个日志实现，是正确打印客户端日志的前提条件。建议通过 mvn clean dependency:tree | grep log 命令排查。

客户端日志配置

消息队列的客户端支持用户自定义日志保存路径、日志级别以及保存历史日志文件的最大个数。考虑到日志传输以及阅读的便利性，消息队列暂不允许自定义单个日志文件大小，保持默认的 64 MB。

各参数的配置说明如下：

- 日志保存路径：请确保应用进程有对该路径写的权限，否则日志不会打印。
- 保存历史日志文件的最大个数：支持 1 到 100 之前的数值；若输入的值超出该范围或格式错误，则系统默认保存 10 个。
- 日志级别：支持 ERROR、WARN、INFO、DEBUG 中任何一种，不配置则默认为 INFO。

默认配置

客户端启动后，会按照如下的默认配置生成日志文件：

- 日志保存路径：/{user.home}/logs/sofamq.log，其中 {user.home} 是指启动当前 Java 进程的用户的根目录
- 保存历史日志文件的最大个数：10 个
- 日志级别：INFO
- 单个日志文件大小：64 MB

自定义配置

> **说明**：若要自定义客户端的日志配置，请升级到 Java SDK 1.2.5 及以上版本。

在 Java SDK 中自定义客户端日志配置，请设置以下系统参数：

- sofamq.client.logRoot：日志保存路径
- sofamq.client.logFileMaxIndex：保存历史日志文件的最大个数
- sofamq.client.logLevel：日志级别

示例配置

您可在启动脚本中或者 IDE 的 VM options 中添加如下系统参数：

- Linux 示例

```
-Dsofamq.client.logRoot=/home/admin/logs -Dsofamq.client.logLevel=WARN -
Dsofamq.client.logFileMaxIndex=20
```

- Windows 示例

```
-Dsofamq.client.logRoot=D:\logs -Dsofamq.client.logLevel=WARN -Dsofamq.client.logFileMaxIndex=20
```

其中，/home/admin/ 和 D:\ 仅为示例，请填写您实际的系统目录。

4.6 Spring 集成

本文介绍如何在 SpringBoot 框架下用 SOFAStack 消息队列收发消息。

背景信息

主要包括以下三部分内容：

- 普通消息生产者和 Spring 集成
- 事务消息生产者和 Spring 集成
- 消息消费者和 Spring 集成

请确保同一个 Group ID 下所有 Consumer 实例的订阅关系保持一致。详情请参见订阅关系一致。

SpringBoot 框架下支持的配置参数和 TCP Java 一致。详情请参见 Java SDK 接口和参数说明 。

生产者与 Spring 集成

申明生产者。

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.alipay.sofa.sofamq.example.springboot.config.MqConfig;
import io.openmessaging.api.OMS;
import io.openmessaging.api.Producer;

@Configuration
public class ProducerClient {
@Autowired
private MqConfig mqConfig;

@Bean(initMethod ="start", destroyMethod ="shutdown")
public Producer buildProducer() {
```

```
Producer producer = OMS.builder().driver("sofamq").build(mqConfig.getMqProperties())
.createProducer(mqConfig.getMqProperties());
return producer;
}
}
```

通过已经与 Spring 集成好的生产者生产消息。

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.alipay.sofa.sofamq.example.springboot.config.MqConfig;

import io.openmessaging.api.Message;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;
import io.openmessaging.api.exception.OMSRuntimeException;

@Component
public class SyncProducerTest {

//普通消息的Producer 已经注册到了spring容器中，后面需要使用时可以直接注入到其它类中
@Autowired
private Producer producer;

@Autowired
private MqConfig mqConfig;

@Test
public void testSend() {
//循环发送消息
for (int i = 0; i < 100; i++) {
Message msg = new Message( //
// Message所属的Topic
mqConfig.getTopic(),
// Message Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤
mqConfig.getTag(),
// Message Body 可以是任何二进制形式的数据， MQ不做任何干预
// 需要Producer与Consumer协商好一致的序列化和反序列化方式
"Hello MQ".getBytes());
// 设置代表消息的业务关键属性，请尽可能全局唯一
// 以方便您在无法正常收到消息情况下，可通过MQ 控制台查询消息并补发
// 注意：不设置也不会影响消息正常收发
msg.setKey("ORDERID_100");
// 发送消息，只要不抛异常就是成功
try {
SendResult sendResult = producer.send(msg);
assert sendResult != null;
System.out.println(sendResult);
} catch (OMSRuntimeException e) {
System.out.println("发送失败");
//出现异常意味着发送失败，为了避免消息丢失，建议缓存该消息然后进行重试。
```

```
}
}
}
}
```

事务消息生产者与 Spring 集成

事务消息的概念详情请参见收发事务消息。

首先需要实现一个 LocalTransactionChecker，如下所示。 一个消息生产者只能有一个 LocalTransactionChecker。

```
import org.springframework.stereotype.Component;

import io.openmessaging.api.Message;
import io.openmessaging.api.transaction.LocalTransactionChecker;
import io.openmessaging.api.transaction.TransactionStatus;

@Component
public class DemoLocalTransactionChecker implements LocalTransactionChecker {
@Override
public TransactionStatus check(Message msg) {
System.out.println("开始回查本地事务状态");
return TransactionStatus.CommitTransaction; //根据本地事务状态检查结果返回不同的TransactionStatus
}
}
```

申明事务生产者。

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.alipay.sofa.sofamq.example.springboot.config.MqConfig;

import io.openmessaging.api.OMS;
import io.openmessaging.api.transaction.TransactionProducer;

@Configuration
public class TransactionProducerClient {

@Autowired
private MqConfig mqConfig;

@Autowired
private DemoLocalTransactionChecker localTransactionChecker;

@Bean(initMethod ="start", destroyMethod ="shutdown")
public TransactionProducer buildTransactionProducer() {
TransactionProducer producer = OMS.builder().driver("sofamq").build(mqConfig.getMqProperties())
.createTransactionProducer(mqConfig.getMqProperties(), localTransactionChecker);
return producer;
```

```
}
}
```

通过已经与 Spring 集成好的生产者生产事务消息。

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.alipay.sofa.sofamq.example.springboot.config.MqConfig;

import io.openmessaging.api.Message;
import io.openmessaging.api.SendResult;
import io.openmessaging.api.transaction.LocalTransactionExecuter;
import io.openmessaging.api.transaction.TransactionProducer;
import io.openmessaging.api.transaction.TransactionStatus;

@RunWith(SpringRunner.class)
@SpringBootTest
public class TransactionProducerTest {

//事务消息的Producer 已经注册到了spring容器中，后面需要使用时可以直接注入到其它类中
@Autowired
private TransactionProducer transactionProducer;

@Autowired
private MqConfig mqConfig;

@Test
public void testSend() {
Message msg = new Message(mqConfig.getTopic(),"TagA","Hello MQ".getBytes());
SendResult sendResult = transactionProducer.send(msg, new LocalTransactionExecuter() {
@Override
public TransactionStatus execute(Message msg, Object arg) {
System.out.println("执行本地事务");
return TransactionStatus.CommitTransaction; //根据本地事务执行结果来返回不同的TransactionStatus
}
}, null);
System.out.println(sendResult);
}
}
```

消费者与 SpringBoot 集成

创建 MessageListener，如下所示。

```
import org.springframework.stereotype.Component;

import io.openmessaging.api.Action;
import io.openmessaging.api.ConsumeContext;
```

```
import io.openmessaging.api.Message;
import io.openmessaging.api.MessageListener;

@Component
public class DemoMessageListener implements MessageListener {

@Override
public Action consume(Message message, ConsumeContext context) {
System.out.println("Receive:" + message);
try {
//do something..
return Action.CommitMessage;
} catch (Exception e) {
//消费失败
return Action.ReconsumeLater;
}
}
}
```

申明消费者。

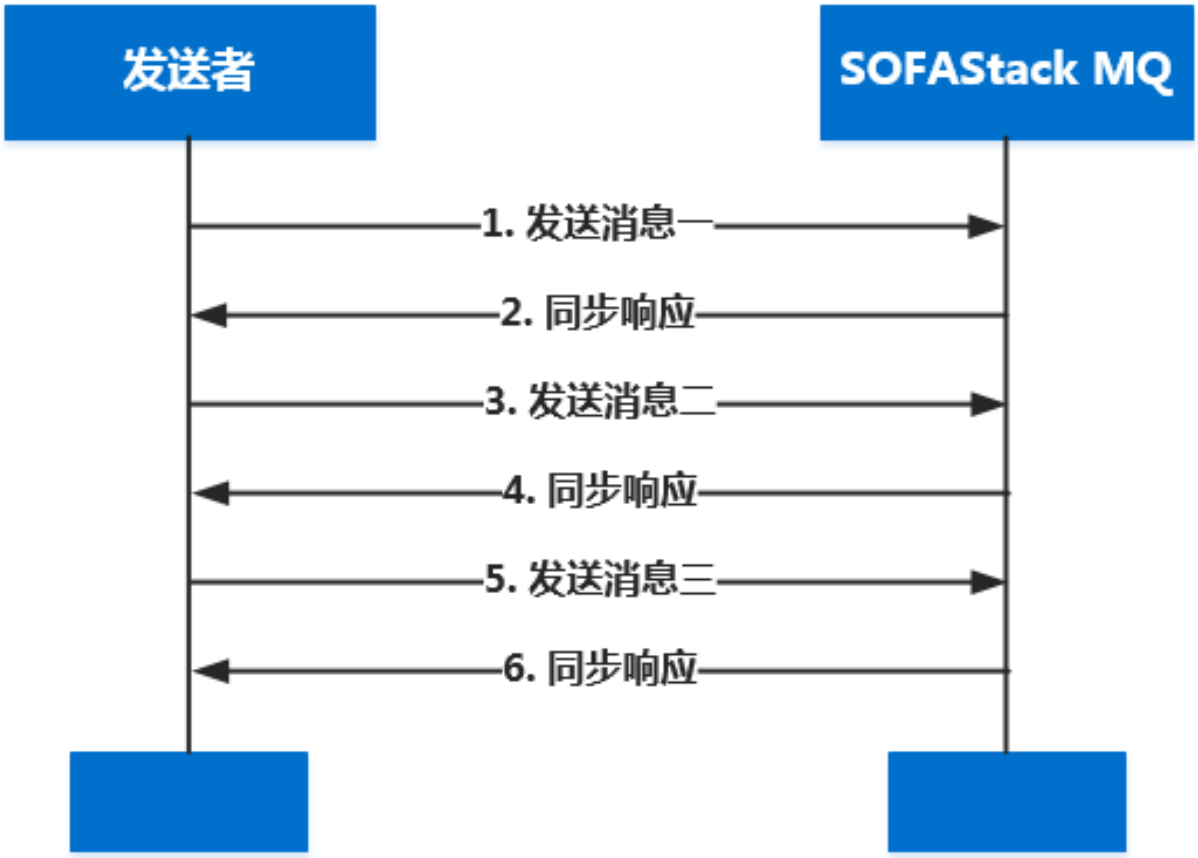
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;

import com.alipay.sofa.sofamq.example.springboot.config.MqConfig;

import io.openmessaging.api.Consumer;
import io.openmessaging.api.OMS;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ConsumerClient {

@Autowired
private MqConfig mqConfig;

@Autowired
private DemoMessageListener messageListener;

@Bean(initMethod ="start", destroyMethod ="shutdown")
public Consumer buildConsumer() {
Consumer consumer = OMS.builder().driver("sofamq").build(mqConfig.getMqProperties())
.createConsumer(mqConfig.getMqProperties());
consumer.subscribe(mqConfig.getTopic(), mqConfig.getTag(), messageListener);
return consumer;
}
}
```

4.7 发送普通消息（三种方式）

SOFAStack 消息队列提供三种方式来发送普通消息：同步发送、异步发送和单向（Oneway）发送。本文介绍了每种发送方式的原理、使用场景、示例代码，以及三种发送方式的对比。

同步发送
原理

同步发送是指消息发送方发出一条消息后，会在收到服务端返回响应之后才发下一条消息的通讯方式。



应用场景

此种方式应用场景非常广泛，例如重要通知邮件、报名短信通知、营销短信系统等。

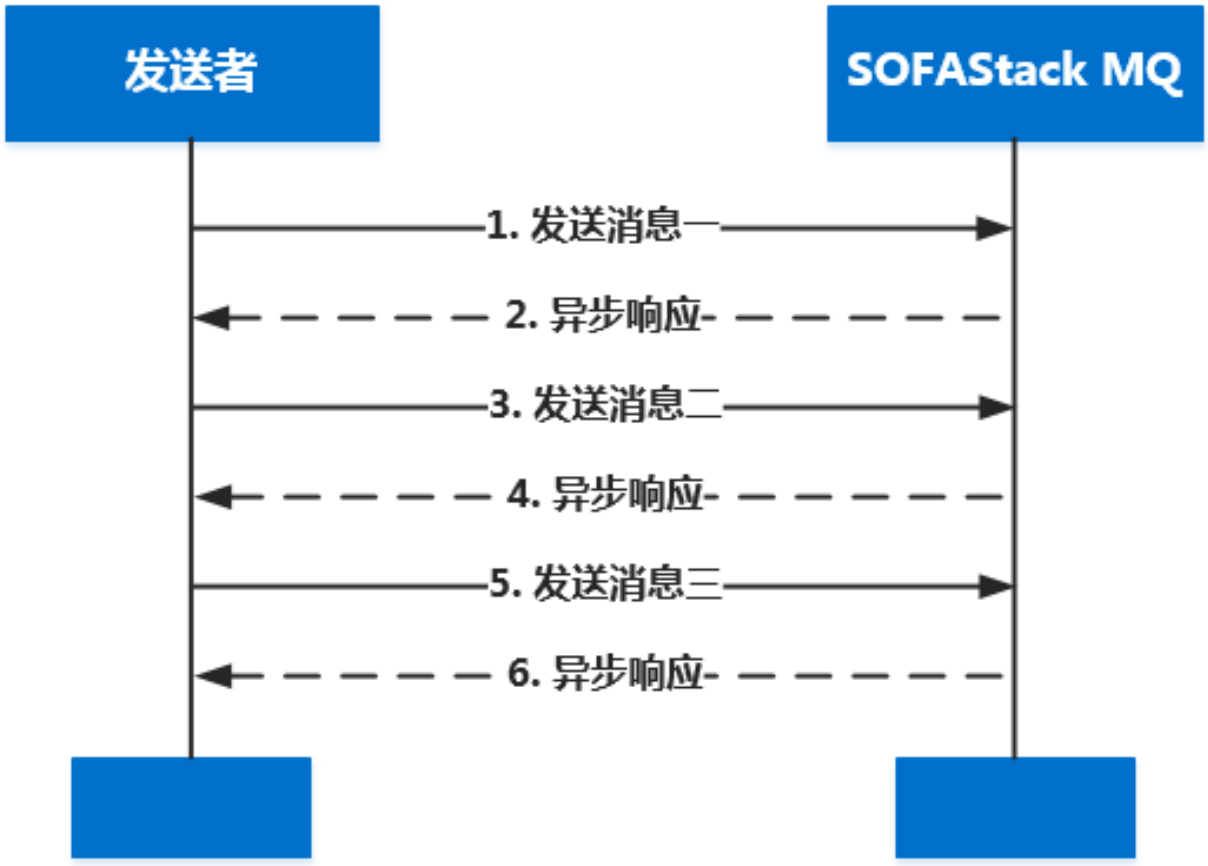
示例代码

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");
```

```
// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
// 您在控制台创建的 Group ID
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);

producer.start();

Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
SendResult sendResult = producer.send(message);
System.out.println(sendResult);
}
}
```

原理

异步发送是指发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。消息队列的异步发送，需要用户实现异步发送回调接口（SendCallback）。消息发送方在发送了一条消息后，不需要等待服务端响应即可发送第二条消息。发送方通过回调接口接收服务端响应，并处理响应结果。



应用场景

异步发送一般用于链路耗时较长，对响应时间较为敏感的业务场景，例如用户视频上传后通知启动转码服务，转码完成后通知推送转码结果等。

示例代码

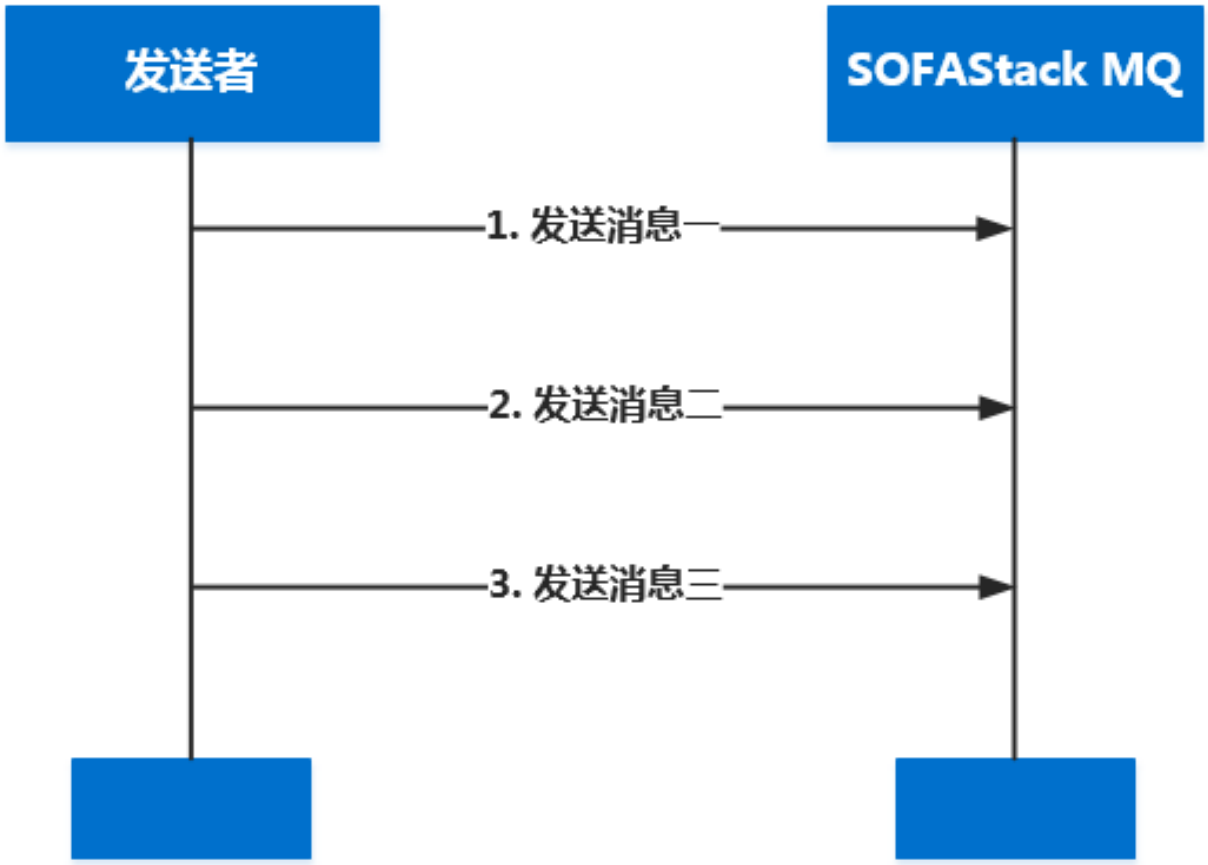
```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
// 您在控制台创建的 Group ID
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);

producer.start();

Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
// 异步发送消息, 发送结果通过 callback 返回给客户端。
producer.sendAsync(msg, new SendCallback() {
@Override
public void onSuccess(final SendResult sendResult) {
// 消费发送成功
System.out.println("send message success. topic="+ sendResult.getTopic() +", msgId="+
sendResult.getMessageId());
}
@Override
public void onException(OnExceptionContext context) {
// 消息发送失败，需要进行重试处理，可重新发送这条消息或持久化这条数据进行补偿处理
System.out.println("send message failed. topic="+ context.getTopic() +", msgId="+ context.getMessageId());
}
});
}
```

```
}
```

单向（Oneway）发送

原理

发送方只负责发送消息，不等待服务端返回响应且没有回调函数触发，即只发送请求不等待应答。此方式发送消息的过程耗时非常短，一般在微秒级别。



应用场景

适用于某些耗时非常短，但对可靠性要求并不高的场景，例如日志收集。

示例代码

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;
```

```
public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
// 您在控制台创建的 Group ID
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);

producer.start();

Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
producer.sendOneway(message);
}
}
```

三种发送方式的对比

下表概括了三者的特点和主要区别。

发送方式	发送 TPS	发送结果反馈	可靠性
同步发送	快	有	不丢失
异步发送	快	有	不丢失
单向发送	最快	无	可能丢失

4.8 发送消息（多线程）

SOFAStack 消息队列的消费者和生产者客户端对象是线程安全的，可以在多个线程之间共享使用。

您可以在服务器上（或者多台服务器）部署多个生产者和消费者实例，也可以在同一个生产者或消费者实例里采用多线程发送或接收消息，从而提高消息发送或接收 TPS。请避免为每个线程创建一个客户端实例。

在多线程之间共享 Producer 的示例代码如下：

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
```

```
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");

// 您在控制台创建的 Group ID
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);

producer.start();

//创建的 Producer 和 Consumer 对象为线程安全的，可以在多线程间进行共享，避免每个线程创建一个实例。

//在 thread 和 anotherThread 中共享 Producer 对象，并发地发送消息至消息队列。
Thread thread = new Thread(new Runnable() {
@Override
public void run() {
try {
Message msg = new Message( //
// Message 所属的 Topic
"TopicTestMQ",
// Message Tag 可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在消息队列的服务器过滤
"TagA",
// Message Body 可以是任何二进制形式的数据，消息队列不做任何干预，
// 需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
"Hello MQ".getBytes());
SendResult sendResult = producer.send(msg);
// 同步发送消息，只要不抛异常就是成功
if (sendResult != null) {
System.out.println(new Date() +" Send mq message success. Topic is:"+ MqConfig.TOPIC +" msgId is:" +
sendResult.getMessageId());
}
} catch (Exception e) {
// 消息发送失败，需要进行重试处理，可重新发送这条消息或持久化这条数据进行补偿处理
System.out.println(new Date() +" Send mq message failed. Topic is:"+ MqConfig.TOPIC);
e.printStackTrace();
}
}
});
thread.start();
```

```
Thread anotherThread = new Thread(new Runnable() {
@Override
public void run() {
try {
Message msg = new Message("TopicTestMQ","TagA","Hello MQ".getBytes());
SendResult sendResult = producer.send(msg);
// 同步发送消息，只要不抛异常就是成功
if (sendResult != null) {
System.out.println(new Date() +" Send mq message success. Topic is:"+ MqConfig.TOPIC +" msgId is:" +
sendResult.getMessageId());
}
} catch (Exception e) {
// 消息发送失败，需要进行重试处理，可重新发送这条消息或持久化这条数据进行补偿处理
System.out.println(new Date() +" Send mq message failed. Topic is:"+ MqConfig.TOPIC);
e.printStackTrace();
}
}
});
anotherThread.start();


// Producer 实例若不再使用时，可将 Producer 关闭，进行资源释放
// producer.shutdown();
}
}
```

4.9 收发顺序消息

顺序消息（FIFO 消息）是 SOFAStack 消息队列提供的一种严格按照顺序来发布和消费的消息类型。本文提供使用 TCP 协议下的 Java SDK 收发顺序消息的示例代码供您参考。

前提条件

您已完成以下操作：

- 通过 Maven 方式引入依赖。Java SDK 的最新版本号，可参见 SDK 版本说明 。

```
<dependencies>
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofamq-client-all</artifactId>
<version>"XXX"</version>
//设置为 Java SDK 的最新版本号
</dependency>
</dependencies>
<repositories>
<repository>
<id>antcloudrelease</id>
<name>Ant Cloud</name>
<url>http://mvn.cloud.alipay.com/nexus/content/groups/open</url>
</repository>
```

```
</repositories>
```

- 准备环境 。
- （ 可选 ） 日志配置 。

背景信息

- 分区顺序：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 FIFO 顺序进行发布和消费。Sharding Key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。

详情请参见 消息类型 > 顺序消息 。

> **说明**：对于新手用户，建议在正式收发消息前，阅读 Demo 工程（ TCP 版 ） 来了解搭建消息队列工程的具体步骤。

发送顺序消息

具体的示例代码，请以 消息队列代码库 为准。

示例代码如下。

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.SendResult;
import io.openmessaging.api.order.OrderProducer;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKeyId 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// AccessKeySecret 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");
// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
OrderProducer producer = accessPoint.createOrderProducer(properties);

producer.start();
```

```
Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
// 分区顺序消息中区分不同分区的关键字段，Sharding Key 与普通消息的 key 是完全不同的概念。
SendResult sendResult = producer.send(message,"YOUR_SHARDING_KEY");
System.out.println(sendResult);
}
}
```

订阅顺序消息

全局顺序消息和分区顺序消息的订阅方式基本一样，示例代码如下。

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.order.ConsumeOrderContext;
import io.openmessaging.api.order.MessageOrderListener;
import io.openmessaging.api.order.OrderAction;
import io.openmessaging.api.order.OrderConsumer;

public class Main {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKeyId 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// AccessKeySecret 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");

OrderConsumer consumer = accessPoint.createOrderedConsumer(properties);
consumer.subscribe("YOUR_TOPIC","YOUR_TAG", new MessageOrderListener() {
@Override
public OrderAction consume(Message message, ConsumeOrderContext context) {
System.out.println(new String(message.getBody()));
return OrderAction.Success;
}
});
consumer.start();
}
}
```

4.10 收发事务消息

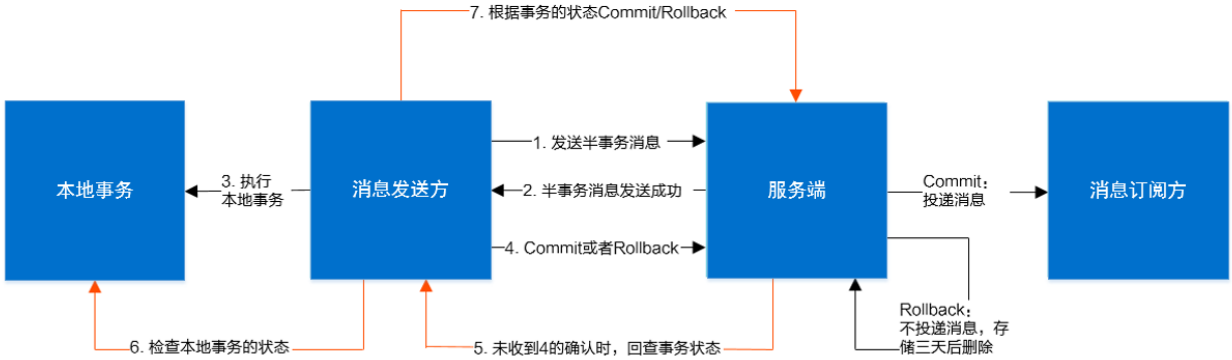
本文提供使用 TCP 协议下的 Java SDK 收发事务消息的示例代码供您参考。

消息队列提供类似 X/Open XA 的分布式事务功能，通过消息队列事务消息，能达到分布式事务的最终一致。

说明：对于新手用户，建议在正式收发消息前，阅读 Demo 工程 来了解搭建消息队列工程的具体步骤。

交互流程

事务消息交互流程如下图所示。



详情请参见 消息类型 > 事务消息 。

前提条件

您已完成以下操作：

- 下载 Java SDK。
- 准备环境 。
- （可选）日志配置 。

发送事务消息

说明：具体的示例代码，请以 消息队列代码库 为准。

发送事务消息包含以下两个步骤：

发送半事务消息（Half Message）及执行本地事务，示例代码如下。

```
import java.util.Properties;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
```

```
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.transaction.LocalTransactionChecker;
import io.openmessaging.api.transaction.LocalTransactionExecuter;
import io.openmessaging.api.transaction.TransactionProducer;
import io.openmessaging.api.transaction.TransactionStatus;

public class TransactionProducerTest {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");

TransactionProducer producer = accessPoint.createTransactionProducer(properties, new
LocalTransactionChecker() {
@Override
public TransactionStatus check(Message msg) {
// check business commit status
return TransactionStatus.CommitTransaction;
}
});
producer.start();

Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
producer.send(message, new LocalTransactionExecuter() {
@Override
public TransactionStatus execute(Message msg, Object arg) {
// if business success, then commit; else rollback
return TransactionStatus.CommitTransaction;
}
}, null);
}
}
```

2. 提交事务消息状态。当本地事务执行完成（执行成功或执行失败），需要通知服务器当前消息的事务状态。通知方式有以下两种：
    - 执行本地事务完成后提交。
    - 执行本地事务一直没提交状态，等待服务器回查消息的事务状态。
      事务状态有以下三种：
    - TransactionStatus.CommitTransaction 提交事务，允许订阅方消费该消息。
    - TransactionStatus.RollbackTransaction 回滚事务，消息将被丢弃不允许消费。

- TransactionStatus.Unknow 无法判断状态，期待消息队列的 Broker 向发送方再次询问该消息对应的本地事务的状态。

事务回查机制说明

发送事务消息为什么必须要实现回查 Check 机制？
当步骤 1 中半事务消息发送完成，但本地事务返回状态为 TransactionStatus.Unknow，或者应用退出导致本地事务未提交任何状态时，从 Broker 的角度看，这条 Half 状态的消息的状态是未知的。因此 Broker 会定期要求发送方能 Check 该 Half 状态消息，并上报其最终状态。

Check 被回调时，业务逻辑都需要做些什么？
事务消息的 Check 方法里面，应该写一些检查事务一致性的逻辑。消息队列发送事务消息时需要实现 LocalTransactionChecker 接口，用来处理 Broker 主动发起的本地事务状态回查请求；因此在事务消息的 Check 方法中，需要完成两件事情：

- 检查该半事务消息对应的本地事务的状态（committed or rollback）。
- 向 Broker 提交该半事务消息本地事务的状态。

订阅事务消息

事务消息的订阅与普通消息订阅一致，详情请参见 订阅消息 。

4.11 收发延时消息

本文提供使用 TCP 协议下的 Java SDK 收发延时消息的示例代码供您参考。

前提条件

您已完成以下操作：

- 下载 Java SDK。
- 准备环境 。
- （可选）日志配置 。

背景信息

延时消息用于指定消息发送到消息队列的服务端后，延时一段时间才被投递到客户端进行消费（例如 3 秒后才被消费），适用于解决一些消息生产和消费有时间窗口要求的场景，或者通过消息触发延迟任务的场景，类似于延迟队列。

延时消息的概念介绍及使用过程中的注意事项，请参见 消息类型 > 定时和延时消息 。

> **说明**：对于新手用户，建议在正式收发消息前，阅读 Demo 工程 来了解搭建消息队列工程的具体步骤。

发送延时消息

具体的示例代码，请以 消息队列代码库 为准。

发送延时消息的示例代码如下。

```
import java.util.Properties;
import java.util.concurrent.TimeUnit;

import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;

public class DelayProducerTest {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKeyId 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// AccessKeySecret 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);

producer.start();

Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
// 延时消息，单位毫秒（ms），在指定延迟时间（当前时间之后）进行投递，例如消息在 5 秒后投递
message.setStartDeliverTime(System.currentTimeMillis() + TimeUnit.SECONDS.toMillis(5));
SendResult sendResult = producer.send(message);
System.out.println(sendResult);
}
}
```

订阅延时消息

延时消息的订阅与普通消息订阅一致，详情请参见 订阅消息 。

4.12 收发定时消息

本文将引导您如何使用 TCP 协议下的 Java SDK 进行定时消息的收发。

前置条件

需要确保您已完成以下操作：

- 下载 Java SDK。
- 准备环境 。
- （ 可选 ） 日志配置 。

背景信息

通过定时消息，在消息发送后，可以在当前时间点之后的某一个时间点，再投递到消费者进行消费，适用于对消息生产和消费有时间窗口要求，或者利用消息触发定时任务的场景。

关于定时消息的更多信息，参见 消息类型 > 定时和延时消息 。

> **说明**：对于新用户，建议在正式收发消息前，阅读 Demo 工程 来了解搭建消息队列工程的具体步骤。

发送定时消息

具体的示例代码，请以 消息队列代码库 为准。

发送定时消息的示例代码如下。

```
import java.util.Properties;
import java.util.concurrent.TimeUnit;
import com.alipay.sofa.sofamq.client.PropertyKeyConst;
import io.openmessaging.api.Message;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendResult;
public class DelayProducerTest {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKeyId 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// AccessKeySecret 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");
// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();
Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");
Producer producer = accessPoint.createProducer(properties);
producer.start();
Message message = new Message("$topic","YOUR_TAG","hello world".getBytes());
// 定时消息，单位毫秒（ ms ），在指定时间戳（ 当前时间之后 ）进行投递，例如 2020-03-13 18:27:00
long timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2020-03-13 18:27:00").getTime();
message.setStartDeliverTime(timeStamp);
SendResult sendResult = producer.send(message);
System.out.println(sendResult);
}
```

```
}
```

订阅定时消息

定时消息的订阅方式与普通消息一致，详见 订阅消息 。

4.13 订阅消息

本文介绍如何通过 SOFAStack 消息队列的 Java SDK 订阅消息。

订阅方式

消息队列支持以下两种订阅方式：

- 集群订阅
  同一个 Group ID 所标识的所有 Consumer 平均分摊消费消息。例如某个 Topic 有 9 条消息，一个 Group ID 有 3 个 Consumer 实例，那么在集群消费模式下每个实例平均分摊，只消费其中的 3 条消息。

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

- 广播订阅
  同一个 Group ID 所标识的所有 Consumer 都会各自消费某条消息一次。例如某个 Topic 有 9 条消息，一个 Group ID 有 3 个 Consumer 实例，那么在广播消费模式下每个实例都会各自消费 9 条消息。

```
// 广播订阅方式设置
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
```

说明：

- 请确保同一个 Group ID 下所有 Consumer 实例的订阅关系保持一致，详情请参见 订阅关系一致 。
- 两种不同的订阅方式有着不同的功能限制，例如，广播模式不支持顺序消息、不维护消费进度、不支持重置消费位点等，详情请参见 集群消费和广播消费 。

示例代码

具体的示例代码，请以 消息队列代码库 为准。

```
import java.util.Properties;
```

```
import com.alipay.sofa.sofamq.client.PropertyKeyConst;

import io.openmessaging.api.Action;
import io.openmessaging.api.ConsumeContext;
import io.openmessaging.api.Consumer;
import io.openmessaging.api.Message;
import io.openmessaging.api.MessageListener;
import io.openmessaging.api.MessagingAccessPoint;
import io.openmessaging.api.OMS;
import io.openmessaging.api.OMSBuiltinKeys;

public class MConsumer {
public static void main(String... args) {
Properties credentials = new Properties();
// AccessKeyId 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.ACCESS_KEY,"$accessKey");
// AccessKeySecret 阿里云身份验证，在阿里云服务器管理控制台创建
credentials.setProperty(OMSBuiltinKeys.SECRET_KEY,"xxxxx");

// 设置 TCP 接入域名，进入控制台的概览页面查看接入点配置
MessagingAccessPoint accessPoint = OMS.builder().driver("sofamq").endpoint("$endpoint")
.withCredentials(credentials).build();

Properties properties = new Properties();
// 设置用户实例，进入控制台的概览页面查看接入点配置
properties.setProperty(PropertyKeyConst.INSTANCE_ID,"$instanceId");
properties.setProperty(PropertyKeyConst.GROUP_ID,"YOUR_GROUP");

// 集群订阅方式 (默认)
// properties.put(PropertyKeyConst.MESSAGE_MODEL, PropertyValueConst.CLUSTERING);
// 广播订阅方式
// properties.put(PropertyKeyConst.MESSAGE_MODEL, PropertyValueConst.BROADCASTING);

Consumer consumer = accessPoint.createConsumer(properties);
consumer.subscribe("YOUR_TOPIC","TAGA||TAGB", new MessageListener() {
@Override
public Action consume(Message message, ConsumeContext context) {
System.out.println(new String(message.getBody()));
return Action.CommitMessage;
}
});
consumer.start();
}
}
```

4.14 单元化开发（仅专有云）

说明：该功能仅适用于专有云开启了 LDC 单元化功能的用户。

本文介绍如何设置 LDC 单元化相关参数。

SOFABoot 生产者

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import io.openmessaging.api.OMS;
import io.openmessaging.api.Producer;
import java.util.Properties;

@Configuration
public class ProducerClient {
@Autowired
Producer producer;

@Bean(initMethod ="start", destroyMethod ="shutdown")
public Producer buildProducer() {
Properties properties = new Properties();
// sofaboot will pass this properties by system property, if not, you can manually set it
// properties.setProperty(PropertyKeyConst.LDC,"true"); // -Dzmode=true
// properties.setProperty(PropertyKeyConst.CELL,"RZXX"); // -Dcom.alipay.ldc.zone=RZ00B
// properties.setProperty(PropertyKeyConst.INSTANCE_ID,"XXX"); // -Dcom.alipay.instanceid=XXX
// properties.setProperty(PropertyKeyConst.DATA_CENTER,"XXX"); // -Dcom.alipay.ldc.datacenter=XXX
// properties.setProperty(PropertyKeyConst.ENDPOINT,"acvip://1.2.X.X"); // -
Dcom.antcloud.antvip.endpoint=1.2.X.X -Dcom.alipay.env=shared
// properties.setProperty(PropertyKeyConst.ACCESS_KEY,"XXX"); // -Dcom.antcloud.mw.access=XXX
// properties.setProperty(PropertyKeyConst.SECRET_KEY,"XXX"); // -Dcom.antcloud.mw.secret=XXX
properties.setProperty(PropertyKeyConst.GROUP_ID,"XXXX");
Producer p = OMS.builder().driver("sofamq").build().createProducer(properties);
return producer;
}

public void send() {
Message message = new Message("TP_XXX","TAGXXX","body".getBytes());
// 如果需要路由到 RZONE，需要设置 UID
message.putUserProperties(UserPropKey.CELL_UID,"XX");
SendResult sendResult = producer.send(message);
}
}
```

SOFABoot 消费者

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import io.openmessaging.api.OMS;
import io.openmessaging.api.Producer;
import java.util.Properties;

@Configuration
public class ConsumerClient {

@Bean(initMethod ="start", destroyMethod ="shutdown")
public Consumer buildConsumer() {
Properties properties = new Properties();
// sofaboot will pass this properties by system property, if not, you can manually set it
// properties.setProperty(PropertyKeyConst.LDC,"true"); // -Dzmode=true
```

```
// properties.setProperty(PropertyKeyConst.CELL,"RZXX"); // -Dcom.alipay.ldc.zone=RZ00B
// properties.setProperty(PropertyKeyConst.INSTANCE_ID,"XXX"); // -Dcom.alipay.instanceid=XXX
// properties.setProperty(PropertyKeyConst.DATA_CENTER,"XXX"); // -Dcom.alipay.ldc.datacenter=XXX
// properties.setProperty(PropertyKeyConst.ENDPOINT,"acvip://1.2.X.X"); // -
Dcom.antcloud.antvip.endpoint=1.2.X.X -Dcom.alipay.env=shared
// properties.setProperty(PropertyKeyConst.ACCESS_KEY,"XXX"); // -Dcom.antcloud.mw.access=XXX
// properties.setProperty(PropertyKeyConst.SECRET_KEY,"XXX"); // -Dcom.antcloud.mw.secret=XXX
properties.setProperty(PropertyKeyConst.GROUP_ID,"XXXX");
properties.setProperty(PropertyKeyConst.LDC_SUB_MODE, LdcSubMode.DEFAULT.name());
Consumer p = OMS.builder().driver("sofamq").build().createConsumer(properties);
consumer.subscribe("TP_XXX","TAGXXX", messageListener);
return consumer;
}

}
```

其中订阅模式 LDC_SUB_MODE 包括：

- DEFAULT：不做任何消息过滤。
- LOCAL：只消费本 CELL 发出的消息。
- RZONE：只在 RZONE 启动消费端，并且只消费目标 RZONE CELL 为本 CELL 的消息。需要在管控台 > **消息路由** 中配置目标单元 RZONE 的消息路由。
- GZONE：只在 GZONE 启动消费端，并且只消费目标 GZONE CELL 为本 CELL 的消息。需要在管控台 > **消息路由** 中配置目标单元 GZONE 的消息路由。
- CZONE：只在 CZONE 启动消费端，并且只消费目标 CZONE CELL 为本 CELL 的消息。需要在管控台 > **消息路由** 中配置目标单元 GZONE 的消息路由。

5 管控指南

5.1 创建和管理 Topic

Topic 是 SOFAStack 消息队列里对消息的一级归类，消息生产者将消息发送到 Topic，而消息消费者则通过订阅该 Topic 来获取和消费消息。

创建 Topic

1. 进入消息队列控制台页面，左侧导航栏选择 **Topic 管理**，进入 Topic 列表页面。
2. 点击列表左上方的 **创建 Topic** 按钮，在新弹出的对话框中，输入或选择 Topic 信息：
    - Topic：Topic 名称，命名不能以 “CID” 和 “GID” 开头，只能包含英文、数字、 短横线（-）和下划线（_），且长度需控制在 3-64 个字符之间。
    - 消息类型：支持的消息类型有普通消息、分区顺序消息、事务消息和定时/延时消息。详细消息类型的说明，可参见 消息类型 。
    - 描述：可选，对该 Topic 的备注内容。

3. 点击 **确定**，即可创建成功。

Topic 创建成功后，您可以选择 **进入详情页** 或 **发送测试消息**。

查看 Topic 详情

在 Topic 列表中，点击您想要查看的 Topic 名称，即可进入该 Topic 的详情页。详情页提供了以下信息：

- **基本信息**：包括 Topic 名称、类型、权限、创建时间和描述。
- **订阅关系**：包括最近发送时间以及订阅了该 Topic 的所有 Group ID 及其相关信息，如消费模式等。

发送测试消息

发送测试消息用于快速验证 Topic 资源的可用性，主要用作测试，发送成功之后可在消息查询查看发送情况。正式生产请使用 调用 SDK 发送消息 。

1. 在 Topic 列表中，找到想要的 Topic，点击其操作列的 **发送测试消息** 按钮。
2. 在 **发送测试消息** 窗口中，输入以下信息：
    - Tag：可选，消息标签，二级消息分类，用来进一步区分某个 Topic 下的消息分类。详见 Topic 与 Tag 。
    - Key：可选，消息的业务标识，由消息生产者设置，唯一标识某个业务逻辑。
    - 消息体:必填，需要发送的具体消息内容。

3. 点击 确定，即可发送测试消息。

消息发送后，您可以直接点击 **进入消息查询** 查看该消息的发送情况，或复制该消息的 MessageID 稍后手动查询该消息。有关消息查询的更多信息，参见 消息查询 。

删除 Topic

不再使用的 Topic 请及时删除，避免产生不必要的费用。

1. 在 Topic 列表中，找到待删除的 Topic，点击其操作列的 删除 按钮。
2. 在弹出的确认窗口中，点击 确定，即可完成删除。

> **注意**：Topic 删除之后，相关的生产者、消费者将会立即停止服务，且将在 10 分钟后完成所有相关资源的清理，请谨慎操作。

更多信息

- 查看订阅关系
- 查看消费者状态
- 消息查询

5.2 创建和管理 Group ID

创建完 Topic 后，您需要为消息的消费者（或生产者）创建客户端 ID ，即 Group ID 作为标识。

Group ID 和 Topic 的关系是 N：N，即一个消费者可以订阅多个 Topic，同一个 Topic 也可以被多个消费者

订阅；一个生产者可以向多个 Topic 发送消息，同一个 Topic 也可以接收来自多个生产者的消息。

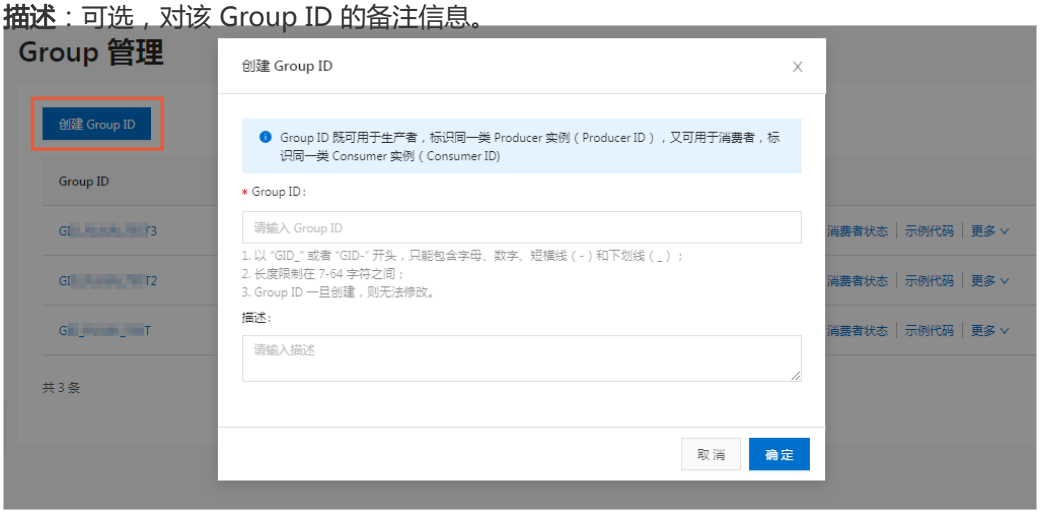
> **说明**：消费者必须有对应的 Group ID，生产者不做强制要求。

创建 Group ID

1. 进入 SOFAStack 消息队列控制台页面，左侧导航栏选择 **Group 管理**，进入 Group ID 列表页面。
2. 点击列表左上方的 **创建 Group ID** 按钮，在新弹出的对话框中，输入或选择 Group ID 信息：
   - **Group ID**：必填，以 “GID_” 或者 “GID-” 开头，只能包含字母、数字、短横线和下划线，且长度限制在 7-64 字符之间。

     > **说明**：Group ID 一旦创建，则无法修改。

   - **描述**：可选，对该 Group ID 的备注信息。



3. 点击 **确定**，即可完成创建。

查看 Group ID 详情

在 Group ID 列表中，点击您想要查看的 Group ID，即可进入该 Group ID 详情页。详情页提供了以下信息：

- **基本信息**：包括 Group ID、创建时间和描述。
- **订阅关系**：包括在线状态、消费模式以及该 Group ID 订阅的 Topic 及其相关信息，如 Topic 名称、过滤规则等。

删除 Group ID

1. 在 Group ID 列表中，找到待删除的 Group ID，点击其操作列的 **删除** 按钮。
2. 在弹出的确认窗口中，点击 **确定**，即可完成删除。

注意：Group 删除之后，由该 Group ID 标识的消费者将立即停止接收消息，且在 10 分钟后所有相关资源将被清理，请谨慎操作。

相关链接

- 查看订阅关系
- 查看消费者状态
- 重置消费位点

5.3 查看订阅关系

在 SOFAStack 消息队列控制台，您可实时查看 Group ID 和 Topic 之间的订阅关系。本文介绍如何在消息队列控制台查看订阅关系。

- 从 Topic 视角，查看即个 Topic 被哪些 Group ID 订阅了。
- 从 Group 视角，查看某个 Group ID 订阅了哪些 Topic。

查看 Topic 被哪些 Group ID 订阅

前提条件：
订阅该 Topic 的 Group ID 至少有一个处于在线状态。

操作步骤:

1. 进入消息队列控制台页面，在左侧导航栏，选择 **Topic 管理**。
2. 在 Topic 列表中，找到目标 Topic，在其操作列点击 **订阅关系**。
3. 在该 Topic 详情页，即可看到所有订阅该 Topic 的在线 Group ID 及其消费模式。

此外，您还可以点击任一 Group ID 的操作列的 **消费者状态**，查看该 Group ID 的消息消费详情。详见 查看消费者状态 。

单元化说明（仅专有云）

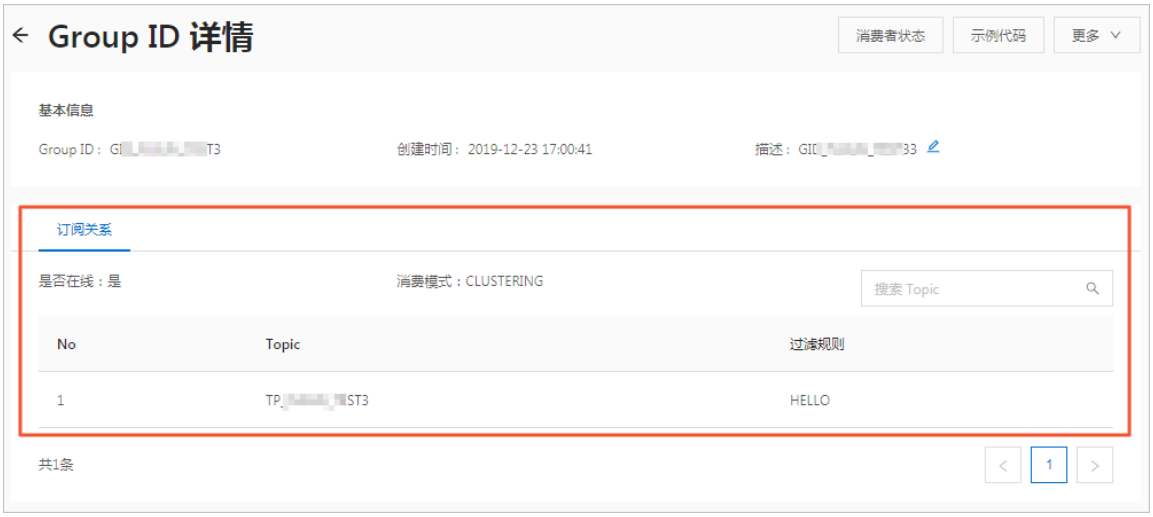
对于专有云开启了单元化功能的用户，在单元化部署环境下，可以自由切换单元，查看指定单元下该 topic 被哪些 Group ID 订阅了，如下图所示。

查看 Group ID 订阅的 Topic

前提条件：
需查询的 Group ID 处于在线状态。

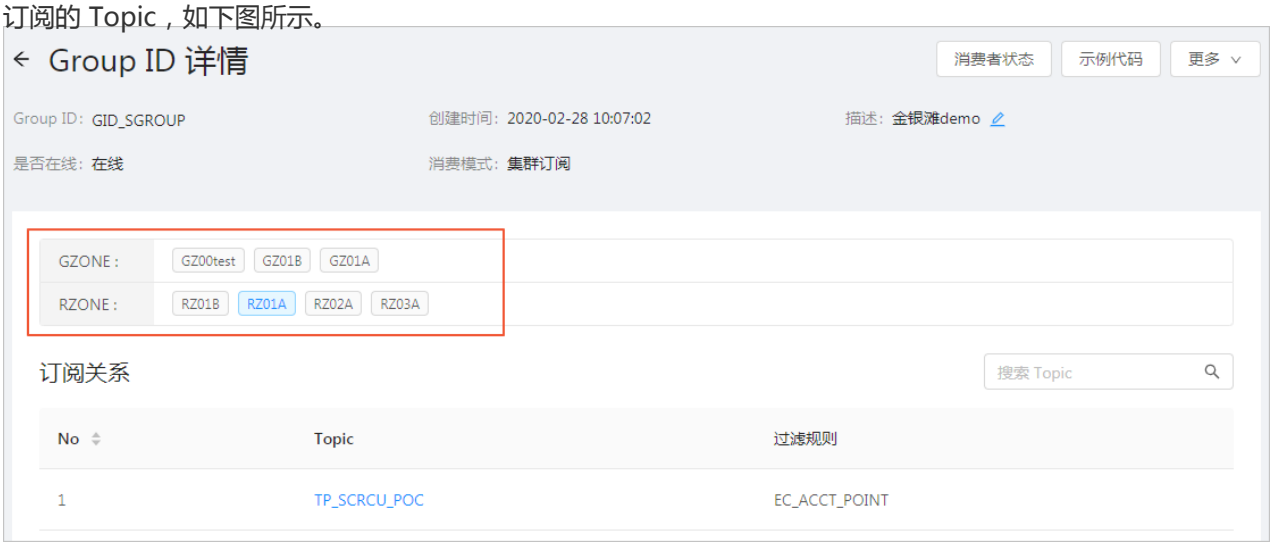
操作步骤:

1. 进入消息队列控制台页面，在左侧导航栏，选择 **Group 管理**。
2. 在 Group ID 列表中，找到目标 Group ID，在其操作列点击 **订阅关系**。
3. 在该 Group ID 的详情页，即可看到该 Group ID 订阅的所有 Topic，以及这些 Topic 的消息过滤规则。



单元化说明（仅专有云）

对于专有云开启了单元化功能的用户，在单元化部署环境下，可以自由切换单元，查看指定单元下该 Group ID 订阅的 Topic，如下图所示。



5.4 查看消费者状态

如果消息消费异常，您可以在SOFAStack 消息队列控制台查看消费者状态，进行问题排查。本文介绍如何查看消费者状态。

- 查看 Group ID 消费者状态
- 查看 Group ID 下单个消费者信息

查看 Group ID 消费者状态

1. 进入消息队列控制台页面，在左侧导航栏选择 **Group 管理**。
2. 在 Group ID 列表中，找到目标 Group ID，点击Group ID进入详情页，或者点击其操作列的 **消费者状态**。
3. 在新窗口，即可查看到消费者群组所有相关的状态信息。

- **在线状态**：
- 只要该 Group ID 下有一个消费者实例在线，就显示在线。
- 若该 Group ID 下所有消费者实例都不在线，则显示离线。

- **实时消费速度**：该 Group ID 下消费者群组接收消息的总 TPS，单位为“条/秒”。
- **实时消息堆积量**：该 Group ID 下消费者群组的未消费消息的总量。
- **最近消费时间**：该 Group ID 下消费者群组最近一次消费消息的时间。
- **消息延迟时间**：该 Group ID 下消费者群组最早的一条未消费消息的生产时间与当前时间的差值。

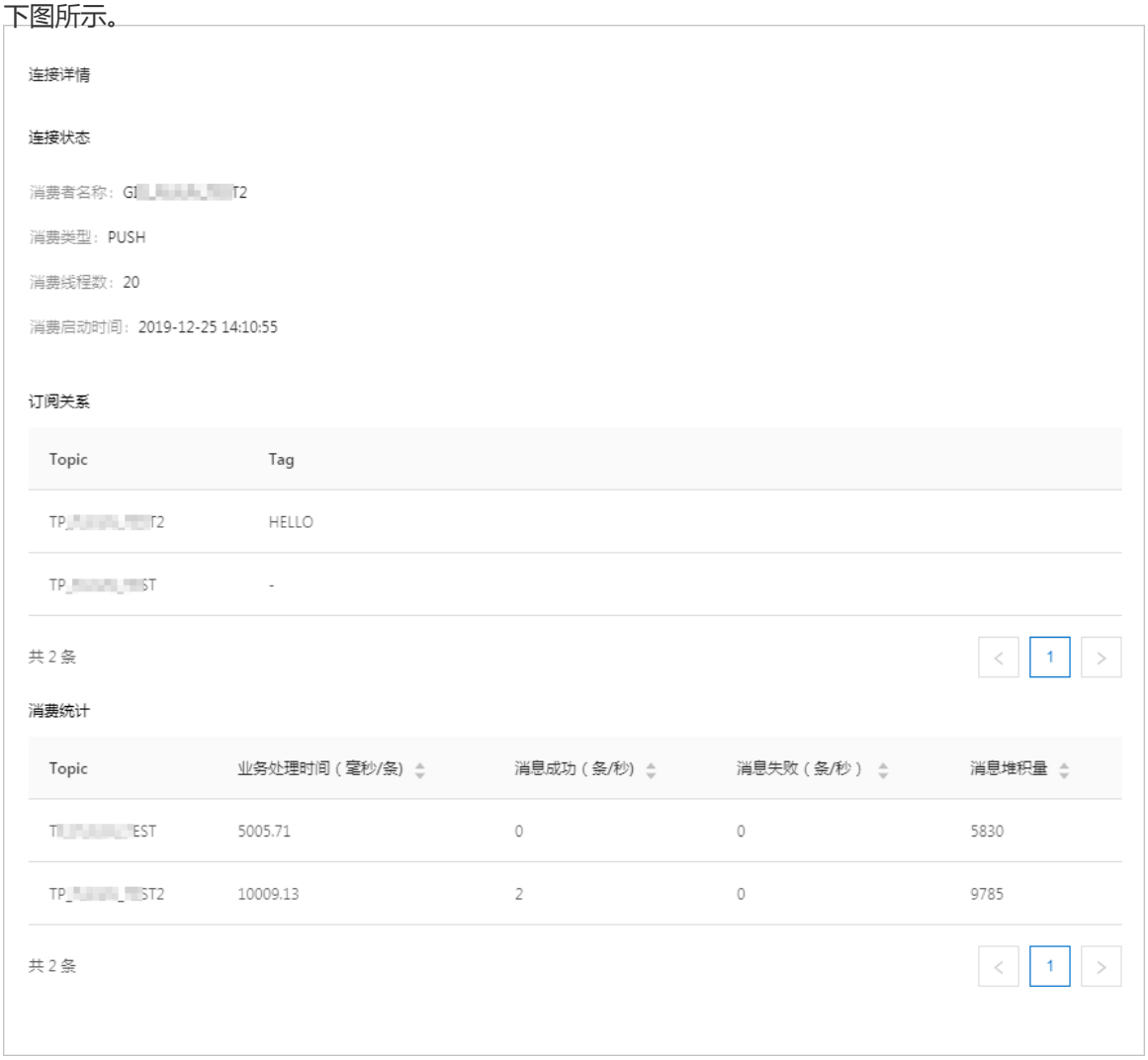
单元化说明（仅专有云）

对于专有云开启了单元化功能的用户，在单元化部署环境下，可以自由切换单元查看消费者状态，如下图所示。

查看 Group ID 下单个消费者信息

如该 Group ID 的在线状态为 **在线**，则在 **连接信息** 区域您可以查看此时在线的每个消费者实例的具体信息，包括客户端 ID、宿主机 IP/公网 IP、当前进程 ID 和消息堆积量等。

- 如需查看更多连接详情，还可在某个消费者信息右侧的 **详细说明** 列，点击 **详细说明**，展现的信息如下图所示。



- 如需查看某消费者实例当前进程的堆栈信息，找到需查看堆栈信息的消费者，在其所在行右侧的 **堆栈信息** 列，点击 **堆栈说明**，展现的信息如下图所示。

5.5 重置消费位点

您可通过重置消费位点，按需清除堆积的或不想消费的这部分消息再从最新位点开始消费，或直接跳转到某个时间点消费该时间点之后的消息。

注意：

- 广播消费模式不支持重置消费位点。
- 目前不支持指定 Message ID、Message Key 和 Tag 来重置消息的消费位点。

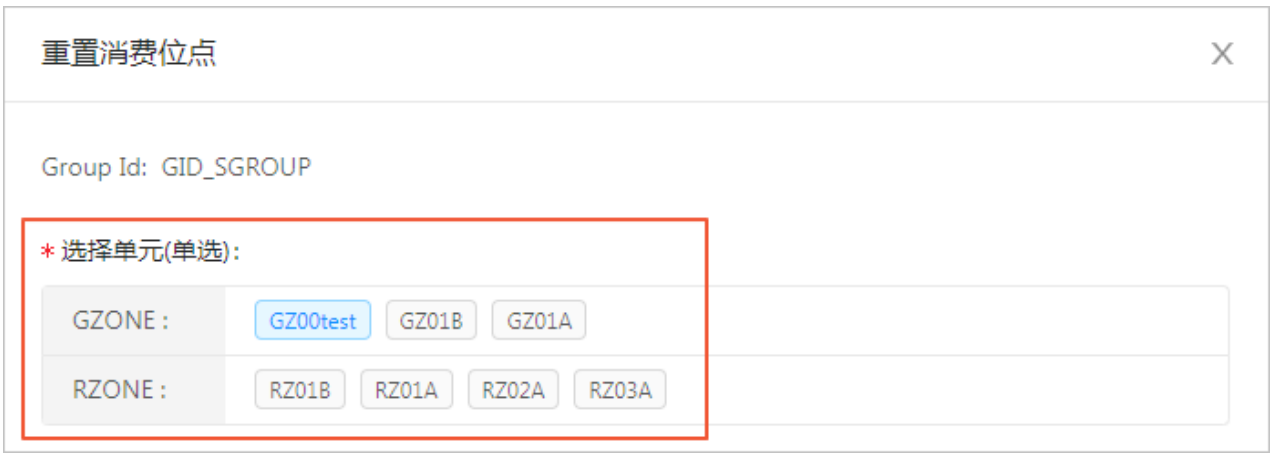
操作步骤

1. 进入 SOFAStack 消息队列控制台页面，在左侧导航栏，点击 **Group 管理**。
2. 找到需要重置消费位点的 Group ID，在其操作列中选择 **更多 > 重置消费位点**。
3. 在 **重置消费位点** 对话框中，按需选择以下选项。
   - Topic：选择重置消费位点针对的 Topic。
   - 重置类型：
     - 清除所有堆积消息，从最新位点开始消费：若选择此项，该 Group ID 在消费该 Topic 下的消息时会跳过当前堆积（未被消费）的所有消息，从这之后发送的最新消息开始消费。
     - 按时间点进行消费位点重置：选择该类型后，您需要选择一个重置时间点，这个时间点之后发送的消息才会被消费。

4. 点击 **确定**。

单元化说明（仅专有云）

对于专有云开启了单元化功能的用户，在进行重置消费位点操作时，需要选择重置消费位点的具体单元，如下图所示。



5.6 消息查询

如遇消息消费有问题，则可通过查询具体发送的消息内容来排查问题。SOFAStack 消息队列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。

操作步骤

1. 进入SOFAStack 消息队列控制台页面，在左侧导航栏，选择 **消息查询**。
2. 在 **消息查询** 页面，您可点击以下任一页签，然后按页面提示输入相应信息。
   - **按 Message ID 查询**：Message ID 是由发送方 Send 方法返回的 32 位 Hex 字符串。按 Message ID 查询，即根据 Topic 和 Message ID，精确定位任意一条消息，获取消息的属性。建议在发送消息成功后将 Message ID 信息打印到日志中，方便问题排查。
     获取 Message ID 的方法如下：

     ```
     SendResult sendResult = producer.send(msg);
     String msgId = sendResult.getMessageId();
     ```

   - **按 Message Key 查询**：Message Key 指发送消息时，设置到消息对象中的 Key 字段，消息队列根据您设置的 Message Key 建立消息的索引信息。按 Message Key 查询，即根据 Topic 和 Message Key，匹配到包含指定 Key 的最近 64 条消息。
     设置 Message Key 的方法如下：

     ```
     Message msg = new Message("Topic","*","Hello MQ".getBytes());
     //Key 值代表消息的业务关键属性，请尽可能全局唯一。
     msg.setKey("TestKey"+System.currentTimeMillis());
     ```

   - **按 Topic 查询**：即根据 Topic 和消息的发送时间范围，批量获取该时间范围内的所有消息，查询量大，不易匹配，一般用在 Message ID 和 Message Key 都无法获得的情况下。
3. 信息输入完毕后，点击 **搜索** 按钮，即可查询消息。

在 **消息查询** 页面，您可以查看到所有查询到的消息及其概览信息，包括 Message ID、Tag、Key 和存储时间。您还可以在下载消息内容、 查询消息轨迹 以及查看消息详情。

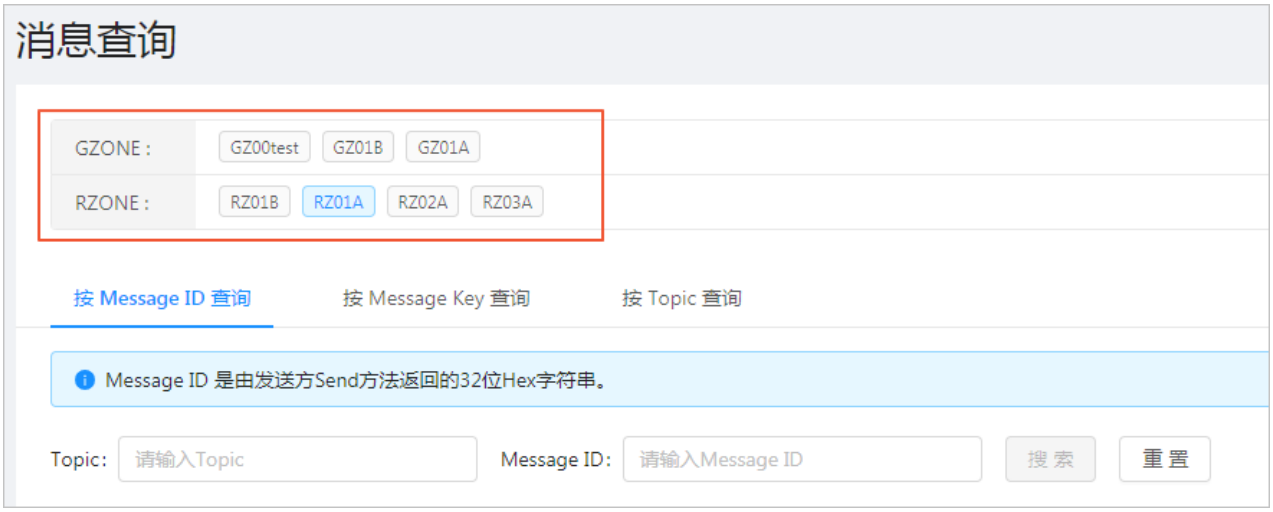
在消息详情页，您可以查看到投递状态。投递状态是消息队列根据各个 Group ID 的消费进度计算出的结果。

> **说明**：投递状态仅仅是根据消费进度估算的结果，如果需要详细的消费信息，请使用消息轨迹查询。详见查询消息轨迹 。

投递状态	说明
已订阅，并且消息至少已被消费一次	该 Group ID 已经正常消费过这条消息。
已订阅，但消息被过滤表达式过滤，请查看 Tag	该消息的 Tag 不符合消费方的订阅关系，消息被过滤，即未被消费。您可以进入控制台中的 Group 管理，然后单击该 Group ID 的列的消费者状态查询其订阅关系。
已订阅，但消息未被消费	该 Group ID 订阅了该消息，但还未消费。有可能是消费过慢，或者消费出现异常导致阻塞。
已订阅，但 Group ID 当前不在线，请通过消息轨迹功能进行精确查询	该 Group ID 订阅了该消息，但是不在线。请检查消费者端应用不在线的原因。
未知异常	出现未收录的异常情况。请 提交工单 进行解决。

单元化说明（仅专有云）

对于专有云开启了单元化功能的用户，在单元化部署环境下，可以自由切换单元，在指定单元内进行消息查询，如下图所示。



5.7 查询消息轨迹

消息轨迹是指一条消息从生产者发送到消息队列服务端，再到消费者消费处理，整个过程中的各个相关节点的时间、状态等数据汇聚而成的完整链路信息。本文介绍如何快速查询消息轨迹。

前提条件

您的消息已从生产者发送。

操作步骤

1. 进入 SOFAStack 消息队列控制台页面，在左侧导航栏，点击 **消息轨迹**。

2. 在消息轨迹的任务列表的右上角，点击 **创建查询任务** 按钮。
3. 在弹出的 **创建查询任务** 对话框中，按需选择查询条件并按提示输入信息。
    - 按 Message ID 查询：该方式属于精确查询，速度快，精确匹配，推荐使用。
    - 按 Message Key 查询：该方式属于模糊查询，最多查询 1000 条轨迹。仅适用于您没有记录 Message ID 但是设置了 Message Key，同时 Message Key 具有区分度的情况。
    - 按 Topic 查询：该方式属于范围查询，适用于没有上述 Message ID 和 Message Key，而且消息量比较小的场景。因为时间范围内消息很多，且不具备区分度。
4. 点击 **确定**。创建完成后，即可在消息轨迹页面查看到刚创建的查询任务，且任务状态显示查询中，说明暂不能查看消息轨迹。
5. 在消息轨迹的任务列表的右上角，点击 **刷新** 按钮，直到状态切换至查询完成。点击展开图标可查看到轨迹的简要信息，主要是消息本身的属性以及接收状态的信息。
6. 在展开的区域框中，点击 **查看轨迹** 即可查看完整的链路图，如下图所示。

该轨迹图提供了以下角色及相关信息：

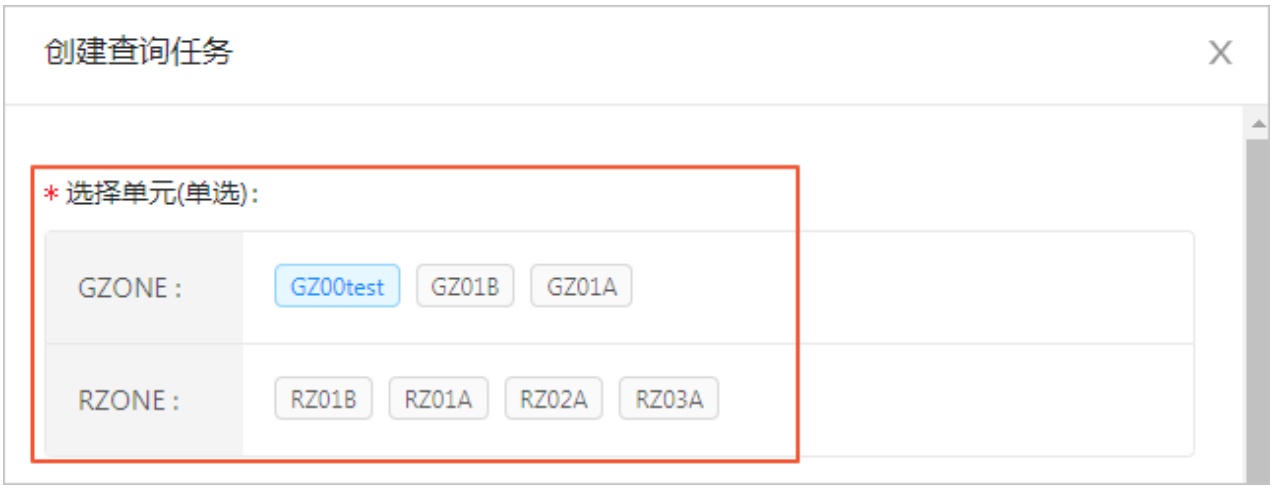
- **生产者**：
    - **发送时间**：消息从生产者发送时的客户端时间戳
    - **发送耗时**：生产者调用 Send 方法发送消息的毫秒耗时
    - **发送状态**：
        - 发送成功：消息发送成功，服务端已经存储成功
        - 发送失败：消息发送失败，服务端没有存储消息，需要重试
        - 消息定时中：该消息是定时或者延时消息，且尚未到达投递时间
        - 事务未提交：该消息是事务消息，且尚未提交状态
        - 事务回滚：该消息是事务消息，并且已经回滚
- Topic:
    - **Key**：消息的业务标识，由消息生产者设置，唯一标识某个业务逻辑。
    - **Tag**：消息标签，二级消息类型，用来进一步区分某个 Topic 下的消息分类。
- **消费者**：
    - **耗时**：消息推送到客户端之后执行 consumeMessage 方法的耗时

- **投递时间**：客户端执行 consumeMessage 方法开始消费消息时的时间戳
- **消费状态**：
  - 全部成功：该消息的所有投递都已成功消费
  - 部分成功：该消息投递中存在消费失败的情况，或消费失败并重试成功的情况
  - 全部失败：该消息的所有投递都消费失败
  - 尚未消费：该消息尚未投递给任何消费方
  - 消费结果未返回：消费消息的方法尚未返回结果，或者被中断，导致本次消费结果未传回服务端
  - 消费成功：该消息已被成功消费
  - 消费失败：消费消息的方法主动返回失败标志，或者是消费方法抛异常

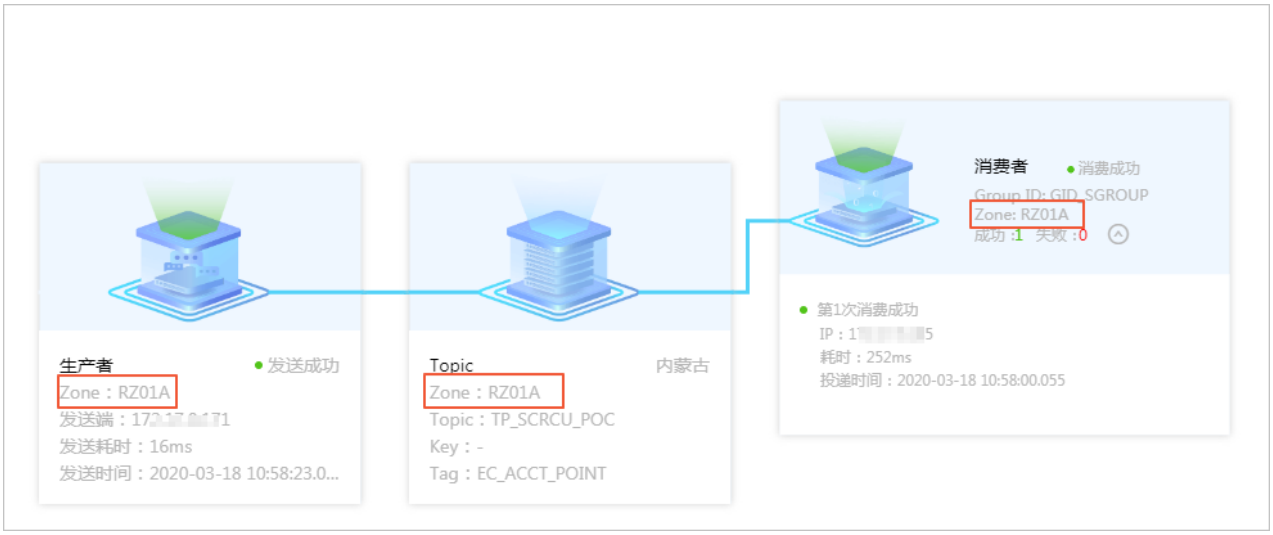
如需删除某个查询任务，可在消息轨迹任务列表页找到需删除的任务，在其操作列，点击 **删除**，按提示完成删除。

单元化说明（仅专有云）

对于专有云开启了单元化功能的用户，在单元化部署环境下，您可以选定单元创建消息查询任务，如下图所示。



查询出的消息轨迹图中，您也可以获取到生产者、Topic 与消费者所在的 Zone 信息，如下图所示。



5.8 消息路由（仅专有云）

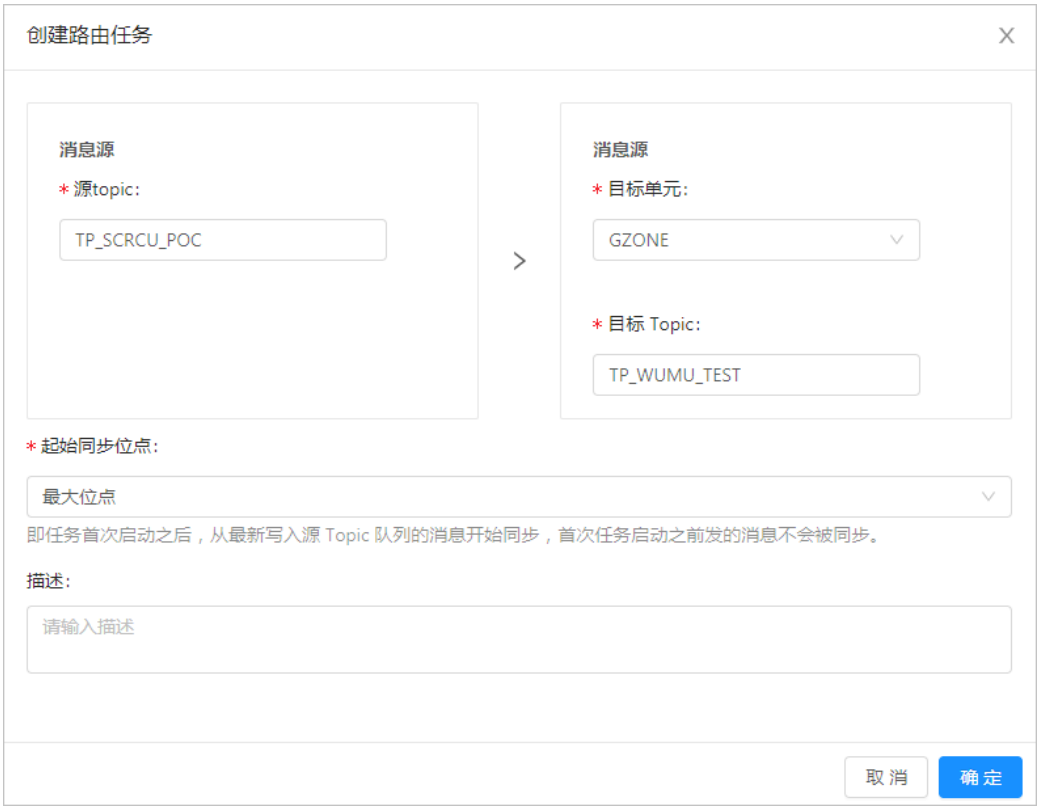
本文将引导您快速创建并管理消息路由任务，实现单元化部署环境下跨单元消息路由和同步。

> **说明**：该功能仅适用于专有云开启了单元化能力的用户。

创建路由任务

您可以登录消息队列控制台，根据业务需求创建跨地域的消息同步任务。操作步骤如下：

1. 进入消息队列控制台页面，在左侧导航栏中选择 **消息路由**。
2. 在消息路由任务列表的左上方，点击 **创建路由任务**。
3. 在 **创建路由任务** 窗口中，配置以下任务信息：
    - **源 Topic**：输入需要同步的消息所属 Topic 名称。
    - **目标单元**：选择消息将被同步到的 Topic 所属单元。
    - **目标 Topic**：输入消息将被同步到的 Topic 名称。
    - **起始同步位点**：选择从源 Topic 中的消息队列的哪个位置开始进行消息同步，即从这个位置之后进入队列的消息都会被同步到目标 Topic。
        - **最小位点**：即任务首次启动之后，从有效期内最早写入源 Topic 队列的消息开始同步，首次任务启动之前发的消息不会被同步。
        - **最大位点**：即任务首次启动之后，从最新写入源 Topic 队列的消息开始同步，首次任务启动之前发的消息不会被同步。
        - **自定义位点**：选择从源 Topic 中的消息队列的哪个位置开始进行消息同步，首次任务启动之前发的消息不会被同步。
    - **描述**：选填，输入对该同步任务的具体描述或备注。



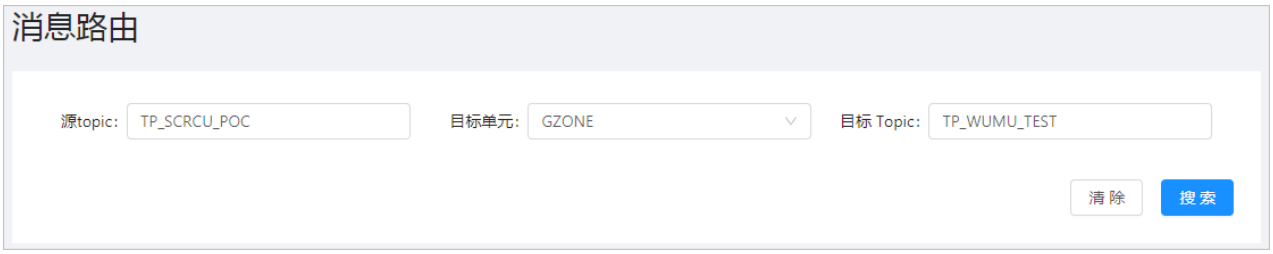
4. 点击 **确定**，完成任务创建。

任务创建完成后，即可在消息路由页面的任务列表中看到刚才创建的任务。

> **说明**：消息路由任务创建成功后，该任务将处于 **初始化** 状态，需要等待片刻，完成初始化后将会进入 **运行中** 状态。

搜索路由任务

在消息路由页面，按需输入源 Topic、目标单元、目标 Topic 后，点击 **搜索** 按钮，即可查询到符合搜索条件的消息路由任务。



启动、停止路由任务

在消息路由页面的任务列表中，找到需要启停的任务，在其操作列点击 **启动** 或 **停止** 按钮，即可启动或停止该任务。

- 任务启动后，该任务的状态会切换至运行中。

- 任务停止后，该任务的状态会切换至已停止。

删除路由任务

说明：仅支持删除已停止的消息路由任务。

在消息路由页面的任务列表中，找到需要删除的任务，在其操作列点击 **删除** 按钮并确认后，即可删除该任务。

6 常见问题

6.1 快速开始相关

与快速开始相关的常见问题如下：

- 消息队列是否可以在公网访问？
- 新创建的 Group ID 从哪里开始消费？
- 消息队列消费失败如何重新消费消息？
- 消息发送了，但是没有收到怎么办？
- 消息队列是否能保证消息不重复？
- 控制台显示的消息堆积量是否包含了 Topic 下所有 Tag 的消息？

消息队列是否可以在公网访问？

暂不支持，消息的生产者和消费者需要和消息队列所在的region网络处于联通状态。

新创建的 Group ID 从哪里开始消费？

- 如果这个 Group ID 是第一次启动，则会忽略启动之前发送的消息，也就是忽略历史消息，从启动之后发送的消息开始消费。
- 如果这个 Group ID 是第二次启动，那么从上次消费的位置开始消费。
- 如果想从特定位置开始消费，可以通过消息队列控制台的消费位点重置功能，指定到具体的时间开始消费。每次重置只针对特定 Group ID 下的特定 Topic，不会影响其他 Group ID。

消息队列消费失败如何重新消费消息？

- **集群消费方式**
  消费业务逻辑代码如果返回 Action.ReconsumerLater，或者 NULL，或者抛出异常，消息都会走重试流程，至多重试 16 次，如果重试 16 次后，仍然失败，则消息丢弃。每次重试的间隔时间如下：

第几次重试	每次重试间隔时间
1	10 秒

2	30 秒
3	1 分钟
4	2 分钟
5	3 分钟
6	4 分钟
7	5 分钟
8	6 分钟
9	7 分钟
10	8 分钟
11	9 分钟
12	10 分钟
13	20 分钟
14	30 分钟
15	1 小时
16	2 小时

可以通过调用 message.getReconsumeTimes() 方法来获取消息的重试次数。

- **广播消费方式**
  广播消费方式仍然能保证一条消息至少被消费一次，但消费失败后不做重试操作。

消息发送了，但是没有收到怎么办？

消息队列提供了多种 消息查询 方式：

- 使用 Topic 按时间范围进行查询，可以查询到一段时间内某 Topic 收到的所有消息。
- 使用 Topic 和 Message ID 对消息进行精准查询。
- 使用 Topic 和 Message Key 较为精准地查询具有相同 Message Key 的一类消息。上述方式可以查询到消息的具体内容以及消费情况，如果需要追踪一条消息从生产者发出到被消费者消费的整个链路中各个相关节点的时间地点，可以使用消息队列最新的消息轨迹查询功能，具体使用方式请参考 查询消息轨迹 。

消息队列是否能保证消息不重复？

绝大多数情况下，消息是不重复的。作为一款分布式消息中间件，在网络抖动、应用处理超时等异常情况下，无法保证消息不重复，但是能保证消息不丢失。

控制台显示的消息堆积量是否包含了 Topic 下所有 Tag 的消息？

是。

消息生产者将所有类型的 Tag 都发送至同一个 Topic 中，消息按照先后顺序在队列中排列，并维护一个消息写入位点；Group ID 启动时会指明需要订阅的 Tag，并从服务端获取当前的消费位点；服务端从当前 Group ID 的消费位点开始遍历队列中的消息，判断如果消息的 Tag 符合 Group ID 订阅的 Tag 即投递给 Group ID，不

符合则跳过该消息。

如下图所示，Group ID 消费位点往前移动，Tag2、Tag3 的消息会在服务端被过滤掉，Tag1 的消息为 Group ID 所需要的，会投递给 Group ID。

因此您在控制台的 **消费者状态 > 消息堆积总量** 看到的是未被过滤的堆积总量，包含了所有 Tag 的消息量。

6.2 消息轨迹

与消息轨迹相关的常见问题如下：

- 为什么查询不到轨迹数据？
- 确定已经消费了消息，但轨迹却显示没有消费，而且客户端 IP 地址和 Producer ID 不符合实际情况？
- Group 列表中为什么没有我的 Group ID？
- 之前的查询任务怎么看不到了？

为什么查询不到轨迹数据？

当输入查询条件查不到轨迹数据时，请参考是否属于以下无法查询情况。

- 消息轨迹目前支持 Java 客户端。
- 查询条件是否正确，如 Topic、Message ID、Message Key 是否输入正确。
- 查询时间区间是否正确，为了提高查询速度，需要您输入消息的发送时间范围。如果查询不到，请尝试扩大时间范围重试。
- 如果确认上述情况无误还是无法查询到结果，请 提交工单 获得技术支持，并附带上日志文件，日志文件位于 /home/{user}/logs/sofamq.log。

确定已经消费了消息，但轨迹却显示没有消费，而且客户端 IP 地址和 Producer ID 不符合实际情况？

出现该现象是由于客户端本身并没有升级到支持轨迹功能的版本，因此消息队列轨迹查询后端仅仅能够拿到部分不完整的轨迹数据，展现结果不正常。建议尽快升级客户端，详情请参见 查询消息轨迹 。

Group 列表中为什么没有我的 Group ID？

出现该情况的原因可能是消息的下游订阅方太多，轨迹图中空间有限，请将鼠标放在滚动条位置，滚动查看完整数据。

之前的查询任务怎么看不到了？

为了防止历史查询任务太多，影响展示效果，消息队列会对用户的历史查询任务做定期清理，默认只保留 7 天以内的查询任务，如果出现历史任务看不到，请重新查询即可。

6.3 顺序消息

与顺序消息相关的常见问题如下：

- 顺序消息是否支持集群消费和广播消费？
- 同一条消息是否可以既是顺序消息，同时又支持定时消息和事务消息？
- 顺序消息支持哪些地域（Region）？
- 顺序消息的使用范围是什么？
- 顺序消息支持哪种消息发送方式？

顺序消息是否支持集群消费和广播消费？

顺序消息支持集群消费，不支持广播消费。

同一条消息是否可以既是顺序消息，同时又支持定时消息和事务消息？

不行。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（Region）？

支持消息队列所有部署的地域。

顺序消息的使用范围是什么？

关于顺序消息的使用范围，包括适用场景和使用限制等，请参见 消息类型 > 顺序消息 。

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。

6.4 消息堆积

常见的消息推挤相关问题如下：

- 如何处理消息堆积？

如何处理消息堆积？

除了异步解耦功能，消息队列还有挡住前端数据洪峰的重要功能，以此保证后端系统的稳定性。这要求消息队列具有一定的消息堆积能力。消息队列能支持 10 亿级别的消息堆积，不会因为消息堆积导致性能明显下降。

问题描述

在消息队列控制台的消费者状态页面，看到 Group ID 的实时消息堆积量的值高于预期，且性能明显下降。

解决方法

面对消息堆积，且有明显性能下降的情况，可采取以下措施处理：

1. 在消息队列控制台，通过 查看消费者状态 获取消息堆积的消费者实例所对应的宿主机 IP 地址，并登录该宿主机或容器。
2. 执行以下任一命令查看进程 pid。
   - ps -ef |grep java
   - jps -lm
3. 执行以下命令查看堆栈信息。
   jstack -l pid > /tmp/pid.jstack
4. 执行以下命令查看 ConsumeMessageThread 的信息，重点关注线程的状态及堆栈。
   cat /tmp/pid.jstack|grep ConsumeMessageThread -A 10 --color
   命令回显如下图所示。

```
"ConsumeMessageThread_28" #906 daemon prio=5 os_prio=0 tid=0x00007f08085d9000 nid=0x42c1 waiting for monitor entry [0x00007f07cc30c000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at com.taobao.txc.a.b.g.c(Unknown Source)
        - waiting to lock <0x0000000702f7d9d0> (a java.lang.Object)
        at com.taobao.txc.a.b.g.a(Unknown Source)
        at com.taobao.txc.a.b.g.b(Unknown Source)
        at com.taobao.txc.a.b.g.a(Unknown Source)
        at com.taobao.txc.client.a.a.a.a(Unknown Source)
        at com.taobao.txc.client.a.a.a.b(Unknown Source)
        at com.taobao.txc.client.a.a.a.a(Unknown Source)
        at com.taobao.txc.client.b.b.a(Unknown Source)
        at com.taobao.txc.client.aop.b.invoke(Unknown Source)
        at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
        at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:207)
        at com.sun.proxy.$Proxy159.orderAutoAdaptation(Unknown Source)
        at com.xtep.o2o.order.adaptation.orderAutoAdaptation.mq.ConsumerOrderAdaptationMQ.consumerMassage(ConsumerOrderAdaptationMQ.java:84)
        at com.xtep.mq.OnsConsumerSpringBean$1.consume(OnsConsumerSpringBean.java:79)
        at com.aliyun.openservices.ons.api.impl.rocketmq.ConsumerImpl$MessageListenerImpl.consumeMessage(ConsumerImpl.java:179)
        at com.alibaba.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentlyService.java:414)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
        at java.util.concurrent.FutureTask.run(FutureTask.java:266)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
```

线程状态的解释说明请参见 Java 官方文档。

更多信息

如按以上操作还未解决因消息堆积而导致的性能下降问题，请 提交工单。提交时，请附带以下信息：

- heap.bin 文件该文件可通过执行 jmap -dump:format=b,file=heap.bin [pid] 命令获取，再执行 gzip heap.bin 命令生成压缩包。[pid] 是消息堆积处理第一步中找到的进程 pid。
- 发生消息堆积的消费者客户端 sofamq.log 本地日志
- 消费者客户端的版本

6.5 使用异常

本文将介绍常见的使用异常问题及相应的解决方案。

- 无法连接 Broker
- 启动 Producer、Consumer 失败，Group ID 重复

- 主动订阅消息，获取队列列表失败
- 消息显示

无法连接 Broker

可能产生的原因：

- 您使用的阿里云云主机（ECS）与消息队列所属服务器不在同一地域（Region）。
- 您可能在非阿里云主机上访问消息队列服务，且您创建的 Topic 不支持非阿里云主机访问。

建议解决方案：
按如下步骤操作：

1. 请确保阿里云主机与创建的 Topic 在同一个地域。

启动 Producer、Consumer 失败，Group ID 重复

可能产生的原因：
在同一个 JVM 进程里面启动多个 Producer 或 Consumer 实例，且这些实例配置了同一个 Group ID，从而导致客户端启动失败。

建议解决方案：
按如下步骤操作：

1. 确保在一个 JVM 进程中只启动了同一个 Group ID 的一个 Producer 或 Consumer 实例，即可以在一个 JVM 进程中同时启动同一个 Group ID 的一个 Producer 和一个 Consumer 实例，但不能同时启动多个 Producer 或 Consumer 实例。
2. 重启应用。

主动订阅消息，获取队列列表失败

可能产生的原因：
可能未在控制台上创建该 Topic，导致订阅方启动时获取 Topic 的队列信息失败。

建议解决方案：
按如下步骤操作：

1. 登录消息队列控制台，在左侧导航栏选择 **Topic 管理 > 创建 Topic**，按提示创建 Topic。
2. 在左侧导航栏选择 **Group 管理 > 创建 Group ID** ，按提示创建 Group ID。
3. 重启应用。

消息显示 *Consumed*, 但消费端未感知到

消息状态显示“Consumed” ，但是消费端业务日志显示没有收到消息。

这种问题一般是下面三个原因导致的。

业务代码在接收到消息后，不立即打印消息收到消息后，如果直接进入业务逻辑，一旦代码遗漏某个逻辑分支，就会导致消息信息没有被留在业务日志里，造成没有收到消息的假象。
建议您收到消息后，立即打印消息信息留存 **messageId, timestamp, reconsumeTime** 等。

消费端部署了多个消费实例
尤其是在调试阶段，消费端不可避免会多次重启，一旦多个消费进程同时存在（进程未退出），那么相当于进入集群的消费模式，多个消费实例会共同分担消费消息。以为没有收到的消息，其实是被另一个消费端接收了。
到消息队列控制台，进入 **Group 管理 > 消费者状态 > 连接状态**，会显示消费端的实例部署情况（有几个消费实例，各自的连接 IP 等等），然后可以自行排查。

消息消费过程出现未被 Catch 的异常，导致消息被重新投递。

```
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);
//如果在 doConsumeMessage()方法中出现未被 Catch 的异常，这行日志将不会打印。
log.info("Receive Message, messageId:", message.getMsgID());
return Action.CommitMessage;
}
}
```

如果问题还未能解决，请提供本地 SDK 日志，联系 售后技术支持。

6.6 状态不一致

本文将介绍常见的状态不一致问题及相应的解决方案。

- 非预期的客户端连接
- 消息不合法
- 参数不合法
- 客户端状态异常
- 订阅关系不一致

非预期的客户端连接

现象:

- 用户侧反馈部分消息发送后未收到。登录消息队列控制台。点击 **消息轨迹 > 创建查询任务 > 按 Message ID 查询**，发现部分消息已发送至 broker，但未投递给下游消费者。
- 登录消息队列控制台。点击 **Group 管理 > 消费者状态**，连接状态栏下方出现不在预期范围内的客户端 IP，同时存在部分消息堆积只在非预期范围内的客户端上。

分析:

原因
在同一套环境中，以错误的方式（AccessKeyId、AccessKeySecret、Topic 等信息配置错误）启动了 Group ID 的客户端，导致该客户端进程占用了 Topic 下的部分消息队列而无法正确消费消息，消息在服务端堆积，无法实时投递给下游正确的消费者。

确认方式
先根据连接状态定位有问题的进程，然后通过 /{user.home}/logs/sofamq.log 以及程序代码确认配置的 AK、SK、Topic 等信息。

恢复方法
可以先将有问题的客户端进程先关闭，此时之前堆积的消息会立马进行 Rebalance 投递至正常的客户端，待定位到问题修复后再重启有问题的进程。（快速恢复）

验证:

1. 登录消息队列控制台。
2. 选择 **Group 管理 > 消费者状态**。连接状态栏下方显示的所有已连接的客户端都能在预期范围内，并能正常消费消息。同时订阅关系是否一致栏显示是。

消息不合法

可能产生的原因：一般是消息属性、消息内容不合法，不合法的情况有：

- 消息为空；
- 消息内容为空；
- 消息内容长度为 0；
- 消息内容超过限定长度。

建议解决方案：
请确保消息没有以上的不合法情况，并根据异常提示进行解决。

参数不合法

可能产生的原因：参数不合法的情况有以下几种：

嵌套的异常说明	异常描述
consumeThreadMin Out of range [1, 1000]	消费端线程数设置不合理
consumeThreadMax Out of range [1, 1000]	消费端线程数设置不合理
messageListener is null	未设置 messageListener
consumerGroup is null	未设置 Group ID
msg delay time more than 40 day	定时消息延时不能超过 40 天

建议解决方案：
按如下步骤操作：

1. 按照异常提示修改客户端对应参数的配置，确保其在合理范围内。
2. 重启应用。

客户端状态异常

可能产生的原因：

- 创建 Consumer、Producer 之后未调用 start() 方法来启动客户端；
- 创建 Consumer、Producer 之后 start() 过程有异常导致客户端启动失败；
- 创建 Consumer、Producer 并成功调用 start() 方法后，显示调用了 shutdown() 方法关闭了客户端。

建议解决方案：
按如下步骤操作：

1. 确保创建 Consumer、Producer 之后调用 start() 保证客户端处于启动状态；
2. 查看 ons.log 判断在客户端在启动过程中是否有异常。

订阅关系不一致

原因描述：

在不同的 JVM 中启动了多个 Consumer，并且给相同的 Group ID 配置了不同的 Topic，或者是相同的 Topic 但 Tag 不同，最终导致订阅关系不一致，消息不符合预期。

错误示例一：同一个 Group ID（GID-MQ-FAQ）订阅的 Topic 不同（分别是 MQ-FAQ-TOPIC-1、MQ-FAQ-TOPIC-2）。
JVM-1上的代码：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1","NM-MQ-FAQ", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println("Receive:" + message);
return Action.CommitMessage;
}
});
consumer.start();
```

JVM-2上的代码：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-2","NM-MQ-FAQ", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println("Receive:" + message);
return Action.CommitMessage;
```

```
}
});

consumer.start();
```

错误示例二：同一 Group ID（GID-MQ-FAQ）订阅的 Topic 相同，但 Tag 不同（分别 NM-MQ-FAQ-1、NM-MQ-FAQ-2）。
JVM-1上的代码：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1","NM-MQ-FAQ-1", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println("Receive:" + message);
return Action.CommitMessage;
}
});
consumer.start();
```

JVM-2上的代码：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID,"GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1","NM-MQ-FAQ-2", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println("Receive:" + message);
return Action.CommitMessage;
}
});

consumer.start();
```

建议解决方案：
请确保在不同 JVM 中使用相同的 Group ID 启动多个 Consumer 时，配置的 Topic 和 Tag 是一致的。

6.7 日志相关

本文将介绍常见的日志相关问题及相应的解决方案。

- 如何根据客户端日志判断当前状态？

如何根据客户端日志判断当前状态？

消息队列客户端日志文件是 sofamq.log，包括 INFO、WARN、ERROR 级别的日志。

此处提供常见的客户端日志打印信息，旨在帮助您更好地从打印的日志中获取信息，并判断当前状态，从而排查故障。

下表列举了 sofamq.log 日志信息说明（持续更新）。

日志级别	打印信息	说明	解决方案
INFO	[persistAll] Group: CID_XXXX ClientId: 10.31.40.100@171374#14159XXX#-2036649XXX#20931314294957XXX updateConsumeOffsetToBroker MessageQueue [topic=XXXX, brokerName=qdinternetorder-XX, queueId=X] 1013XXX	这种现象说明消息已经消费成功，并且在消息队列服务端已持久化消费进度；MessageQueue 里包括了消息主题、对应的 brokerName、消费队列的 ID	不涉及
INFO	[PULL_TPS] [CID_XXXX@CID_XXXX] Stats In One Minute, SUM: 0 TPS: 0.00 AVGPT: 0.00</br>[PULL_RT] [%RETRY%CID_XXXX@CID_XXXX] Stats In One Minute, SUM: 0 TPS: 0.00 AVGPT: 0.00	该类信息打印的是从 consumeQueue 中拉取消息时的 TPS（每秒 Request 的数量）	不涉及
WARN	[TIMEOUT_CLEAN_QUEUE]broker busy, start flow control for a while, period in queue: 905ms, size of queue: 1164	消息队列服务端压力过大，处理不了过多的请求；由于服务端在存储数据时是先写入 pageCache，然后去刷盘，因此每隔 10s 会去清理过期的请求（此过程会判断缓存页是否繁忙）	1. 扩容，增加 Broker，分担压力 2. osPageCacheBusyTimeOutMills 属性值调大
WARN	execute the pull request exceptioncom.aliyun.openservices.shade.com.alibaba.rocketmq.client.exception.MQBrokerException: CODE: 25 DESC: the consumer's subscription not latest	Broker 每隔一段时间就会向 NameServer 上报自己的路由信息，如果此过程网络抖动，拉不到最新的订阅信息，导致消费者消费的时候，会出现该警告	不涉及
WARN	[WRONG]mq is consuming, so can not unlock it, MessageQueue [topic=XX, brokerName=szorder2-02, queueId=1]. maybe hanged for a while, 2	进行负载均衡时，对消息处理队列尝试加锁，如果 1s 内还未加锁成功，说明当前消息处理队列已经有消费者在访问，不能进行解锁	不涉及
WARN	doRebalance, XXX-CID, add a new mq failed, MessageQueue [topic=XXXX, brokerName=szorder2-XX, queueId=X], because lock failed	当前使用的是顺序 Topic，为了保证单个分区中消息的顺序消费，会有个 Lock 的机制。客户端有这个日志说明其中某个分区已经有客户端在消费了。	不涉及
WARN	get Topic [XXXXXX] RouteInfoFromNameServer is not exist valuecom.aliyun.openservices.shade.com.alibaba.rocketmq.client.exception.MQClientException: CODE: 17 DESC: No topic route info in name server for the topic: TOPIC_XXXXX</br>See http://rocketmq.apache.org/docs/faq/ for further details.	• AccessKey（包含 AccessKeyId 和 AccessKeySecret）配置错误 • 没有控制台于当前实例下创建的 Group ID（GID） • 实例化的代码中，NameServerAddr 没有配置正确	• 配置正确的 AccessKey • 在当前实例下创建 GID • Java SDK 1.8.0 及以上版本，推荐配置 NameServerAddr，此参数可从消息队列控制台获取，与之前版本配置的 ONSAddr 是不一致的
WARN	com.aliyun.openservices.ons.api.impl.authority.exception.AuthenticationException: signature validate by dauth failed	AccessKey（包含 AccessKeyId 和 AccessKeySecret）配置错误	AccessKey 要配置创建该 GID 使用的 AccessKey
WARN	NettyClientPublicExecutor_3 - execute the pull request exceptioncom.aliyun.openservices.shade.com.alibaba.rocketmq.client.exception.M	订阅关系没有推送到消息队列 broker 上	subscription.json 文件里直接添加 GID 对应的信息即可

	QBrokerException: CODE: 26 DESC: subscription group [CID_XXX] does not exist,See http://rocketmq.apache.org/docs/faq/ for further details.		
WARN	execute the pull request exceptioncom.aliyun.openservices.shade.com.alibaba.rocketmq.client.exception.MQBrokerException: CODE: 24 DESC: the consumer's subscription not exist	缺少订阅关系	不涉及

6.8 应用内存不足

现象

- 在应用部署的机器上通过查看内存已消耗完。
- 在 /{user.home}/logs/sofamq.log 能搜索到 OutOfMemory 关键字。
- 用户侧反馈消息队列控制台：进入 **Group 管理 > 消费者状态**，**堆积量** 栏显示消息堆积较多，**连接状态** 栏显示了各个已连接客户端的消息堆积。通过 Jstack 排查， ConsumeMessageThread_ 线程无消费卡住现象。

分析

默认客户端最多会给每个 Topic 的每个队列缓存 1000 条消息。假设每个 Topic 的队列数是 16 个（集群 2 主 2 备，每台 broker 上 8 个队列），该 Topic 下单条消息平均大小为 64 KB，那么最终该 Topic 在客户端缓存的消息 Size：16 *1000* 64 KB = 1 GB。如果用户同时订阅了 8 个 Topic 都在客户端内存缓存消息，最终占用内存将超过用户的 JVM 配置，导致 OOM。

- **原因**
  默认最大消耗内存 512 MB（Group ID 订阅的所有 Topic 缓存总和）；如果应用仍然出现 OOM 现象，可在 ConsumerBean 启动时，配置 com.alipay.sofa.sofamq.client.PropertyKeyConst#MAX_CACHED_MESSAGE_SIZE_IN_MI_B 参数自行定义最大消耗内存（范围在 16 MB ~ 2048 MB）。
  确认方式
  确认应用依赖的 ons-client 版本，并通过 JVM 确认给进程分配的内存大小。
  恢复方案
  根据应用机器的内存使用清况给对应的 ConsumerBean 设置 com.alipay.sofa.sofamq.client.PropertyKeyConst#MAX_CACHED_MESSAGE_SIZE_IN_MI_B 参数，然后重启应用。

验证

在 /{user.home}/logs/sofamq.log 中 OutOfMemory 关键字不再出现。登录消息队列控制台，选择 **Group 管理 > 消费者状态**，看到 **实时消费速度** 栏的值上升，同时堆积量的值下降。


蚂蚁金服
ANT FINANCIAL
金融科技
FINANCIAL TECHNOLOGY