SOFAStack

消息队列 SOFAMQ
使用指南

产品版本：AntStack Plus 1.11.0

文档版本：20230215

蚂蚁集团
ANT GROUP

法律声明

蚂蚁集团版权所有©2022，并保留一切权利。

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

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。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。蚂蚁集团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，并在蚂蚁集团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集团授权渠道下载、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。

通用约定

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

目录

1.什么是消息队列 07
1.1. 概述 07
1.2. 产品优势 08
1.3. 产品架构 09
1.4. 功能特性 10
1.5. 应用场景 12
1.6. 基础术语 15
1.7. 使用限制 17
2.消息功能详解 19
2.1. 消息类型 19
2.2. Topic 与 Tag 22
2.3. 订阅关系一致 23
2.4. 集群消费和广播消费 26
2.5. 消息重试 28
2.6. 消息过滤 31
2.7. 消息幂等 34
3.快速入门 36
3.1. 操作链路概述 36
3.2. 接入点设置说明 43
4.JAVA SDK 参考 45
4.1. SDK 版本说明 45
4.2. Demo 工程 47
4.3. 接口和参数说明 51
4.4. 准备环境 53
4.5. 日志配置 54
4.6. Spring 集成 56

4.7. 发送普通消息（三种方式） 61
4.8. 发送消息（多线程） 67
4.9. 收发顺序消息 69
4.10. 收发事务消息 71
4.11. 收发延时消息 74
4.12. 收发定时消息 75
4.13. 订阅消息 77
4.14. 单元化开发 78
5.管控指南 82
5.1. 创建和管理 Topic 82
5.2. 创建和管理 Group ID 85
5.3. 查看订阅关系 88
5.4. 查看消费者状态 91
5.5. 重置消费位点 94
5.6. 消息查询 95
5.7. 查询消息轨迹 96
5.8. 死信队列 99
5.9. 资源大盘 103
5.10. 报警管理 105
5.11. 消息路由 108
5.12. 启用 TLS 通信加密 110
6.权限说明 118
7.常见问题 119
7.1. SOFAStack 常见问题 119
7.2. SOFAMQ 常见问题 119
7.3. 配置相关 121
7.4. 消息轨迹 122
7.5. 顺序消息 123

7.6. 消息负载均衡策略 123
7.7. 消息堆积 127
7.8. 日志相关 128
7.9. 使用异常 131
7.10. 资源不存在 133
7.11. 状态不一致 134
7.12. 应用内存不足 137

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，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）。
- 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。

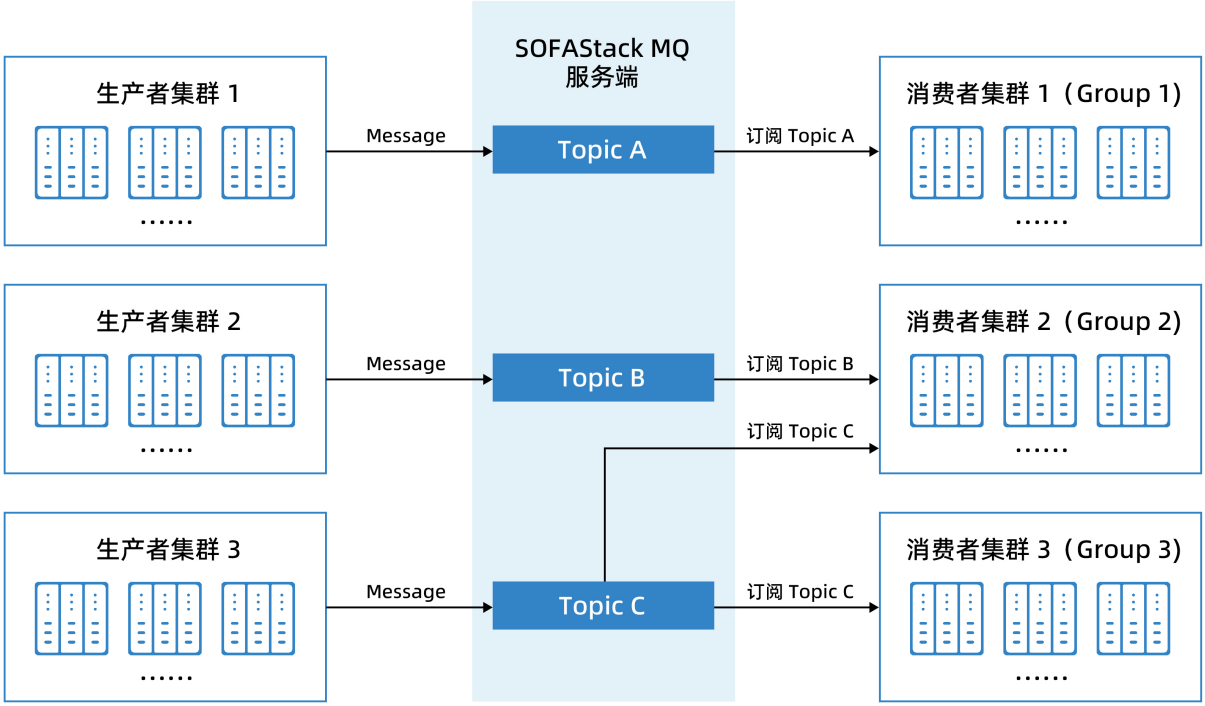
核心概念

- 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 基础术语。

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

- **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

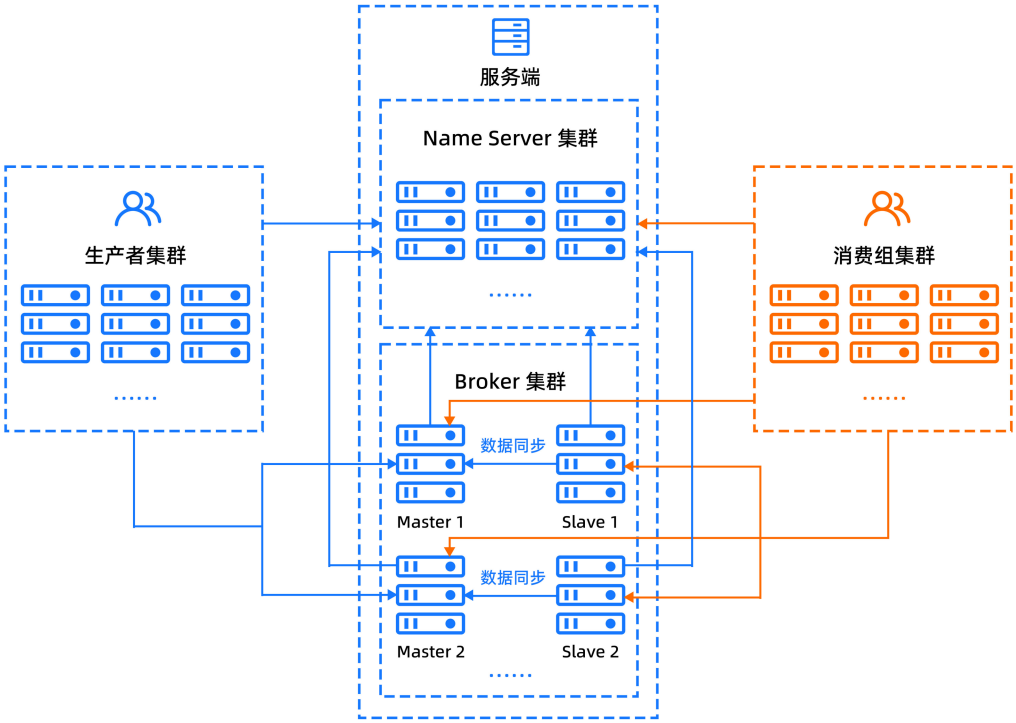
本文介绍 SOFAStack 消息队列的系统部署架构，方便您更好地理解消息队列的高可用性。

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：

- **NameServer**：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，该组件内置 zookeeper 用来负责 Broker 的选主和集群管理。
- **Broker**：消息中转角色，负责存储和转发消息。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 Producer**：与 Name Server 集群中的其中一个节点（随机）建立长连接（Keep-alive），定期从 Name Server 读取 Topic 路由信息，并与提供 Topic 服务的 Master Broker 建立长连接，且定时向 Master Broker 发送心跳。
- **消费者 Consumer**：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。

SOFAMQ 还包含如下两个未在图中展示的组件，主要功能如下：

- **Console**：SOFAMQ 的图形化管理控制台。
- **Router**：LDC 场景下，根据配置的 Router 规则，进行逻辑单元之间消息转发，使 SOFAMQ 支持 LDC 能力的组件。

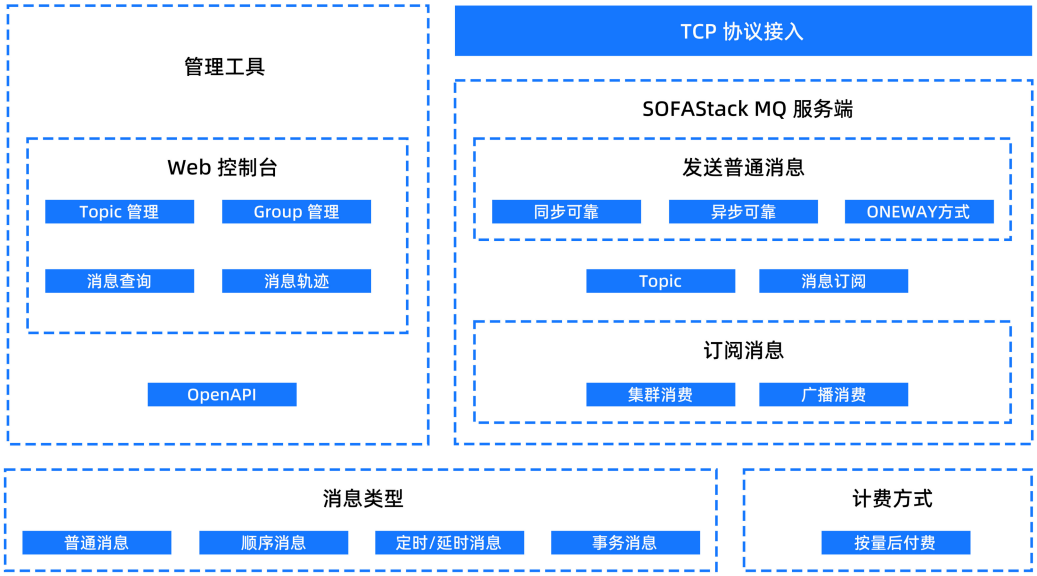
更多信息

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。
- 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：

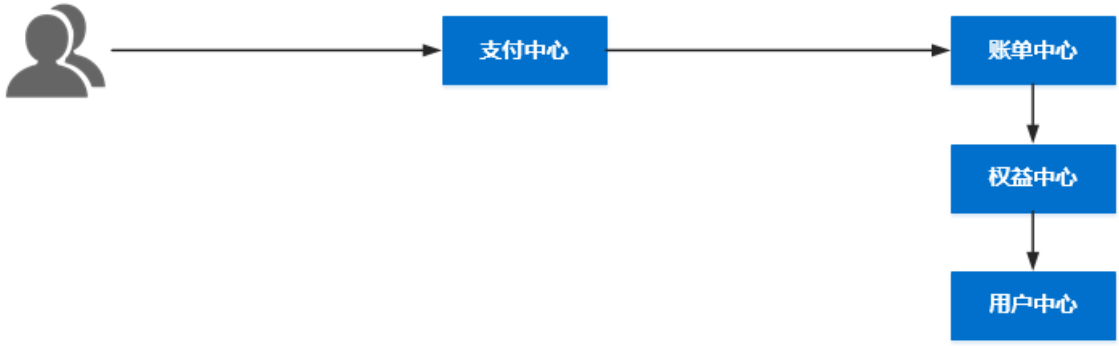
- 异步解耦
- 分布式事务的数据一致性
- 削峰填谷

异步解耦

传统处理方式

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：

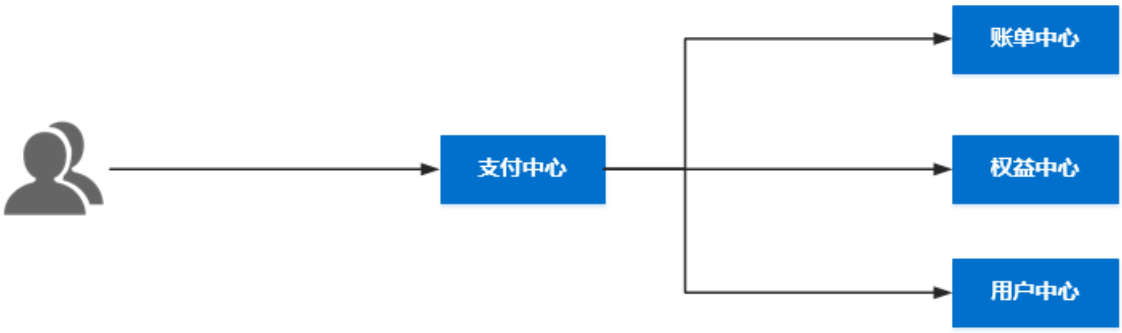
- 串行方式



流程说明如下：

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

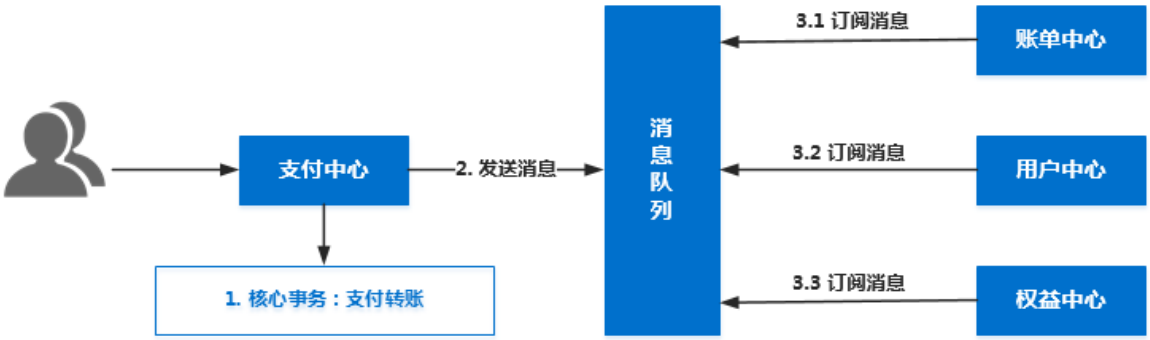


流程说明如下：

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。

异步解耦方式

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。



流程说明如下：

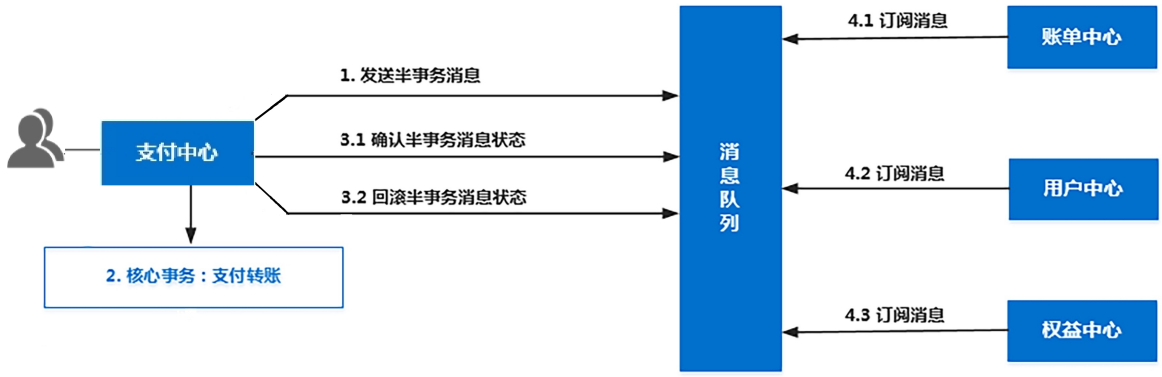
1. 用户在转账页面，填写金额等相关信息，完成转账操作。
2. 转账成功后，发送一条支付消息至消息队列。消息队列会马上返回响应给支付中心，转账完成。
3. 下游的账单中心、权益中心、用户中心等系统订阅消息队列的支付消息，完成后续的业务流程。

异步解耦是消息队列的主要特点和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耦，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。此时，需要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）

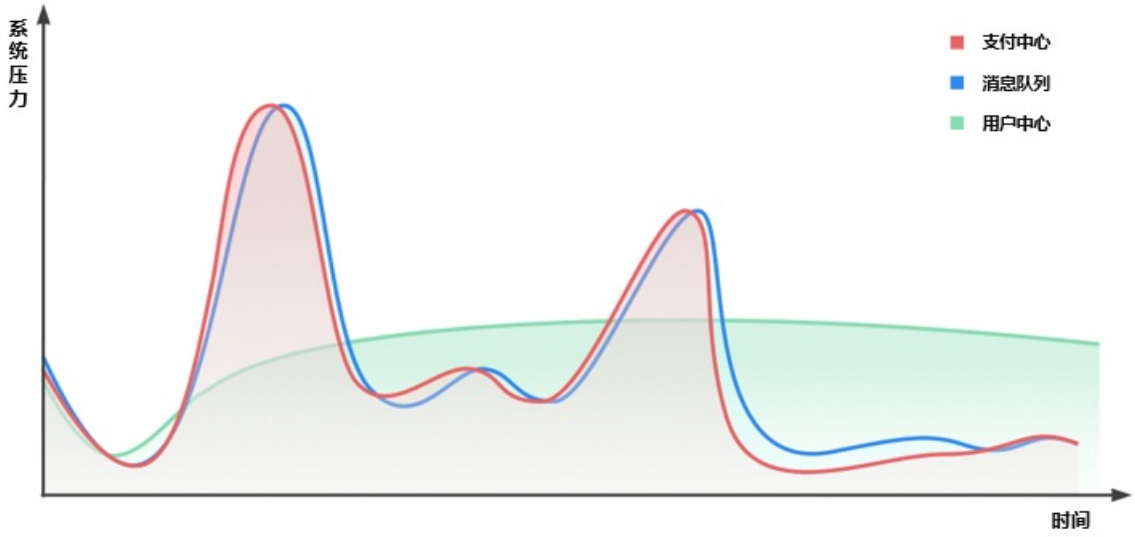
分布式事务消息的更多详细内容请参见 事务消息。

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。

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，具体的队列数可咨询技术支持。
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 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 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 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 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 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，在使用过程中请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：4 MB	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	4000 条/秒 说明 消息收发计算 TPS 规格是按照消息大小 1 KB 为基准计算消息收发之和。	如需更高规格，请 提交工单。
定时/延时消息的延时时长	40 天	• 默认最大限制 3 天，超过 3 天消息将发送失败。您可以修改 `timerMaxDelaySec` 参数（单位：秒）增加延时时间限制至 40 天，即 3456000。 • 您可以在 `msg.setStartDeliverTime` 参数（单位：毫秒）设置消息延时时间。

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
- 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 2 天内的任何时刻（单位毫秒），超过 2 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 2 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（Java），请参见 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。
- 示例说明如下：

  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 收发顺序消息。

事务消息

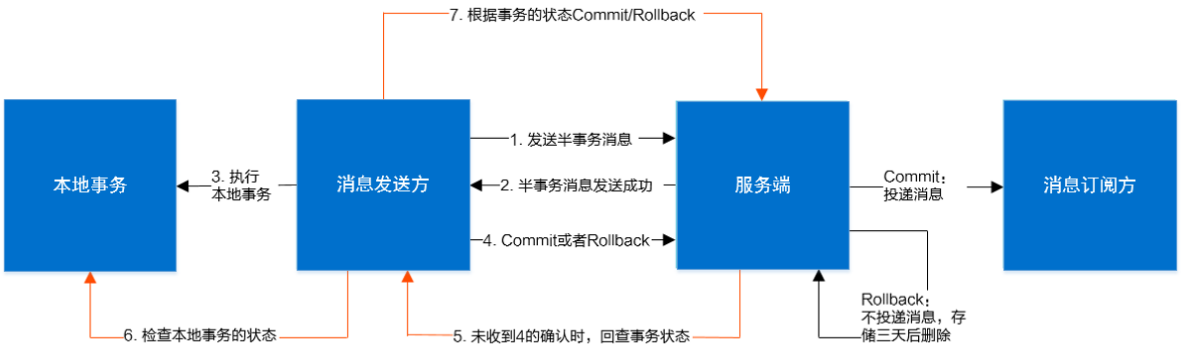
- 事务消息：消息队列提供类似 X/Open XA 的分布式事务功能，通过消息队列事务消息能达到分布式事务的最终一致。
- 半事务消息：暂不能投递的消息，发送方已经成功地将消息发送到了消息队列服务端，但是服务端未收到生产者对该消息的二次确认，此时该消息被标记成“暂不能投递”状态，处于该种状态下的消息即半事务消息。
- 消息回查：由于网络闪断、生产者应用重启等原因，导致某条事务消息的二次确认丢失，消息队列服务端通过扫描发现某条消息长期处于“半事务消息”时，需要主动向消息生产者询问该消息的最终状态（Commit 或 Rollback），该询问过程即消息回查。

适用场景

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。



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 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

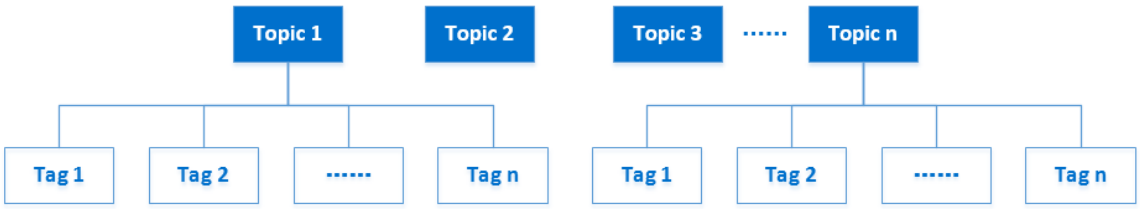
在 SOFAStack 消息队列中，Topic 与 Tag 都是业务上用来归类的标识，区分在于 Topic 是一级分类，而 Tag 可以理解为是二级分类。您可通过本文了解如何搭配使用 Topic 和 Tag 来实现消息过滤。

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、定时或延时消息、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，如付款、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。

注意事项

- 同一个 Topic 下有多个 Tag 的情况时，无法区分每个 Tag 的消费进度、消息积压数量等，且 Tag 间相互影响。例如 Topic 下分为 Tag 1 和 Tag 2，假如 Tag 2 消费较慢会导致同一 Topic 下的 Tag 1 同样消费速度减慢；假如实际 Tag 1 积压 100，Tag 2 积压 200，最终只能确定该 Topic 的整体消费进度、积压量为 300，无法准确区分出每个 Tag 的消费进度和积压数量。
- 如为避免 Tag 间相互影响需要申请接入多个 Topic，请自行评估整体峰值，超过 5000 Quota 需向消息队列提供机器资源。

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 下所有消费者实例所订阅的 Topic、Tag 以及对消息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

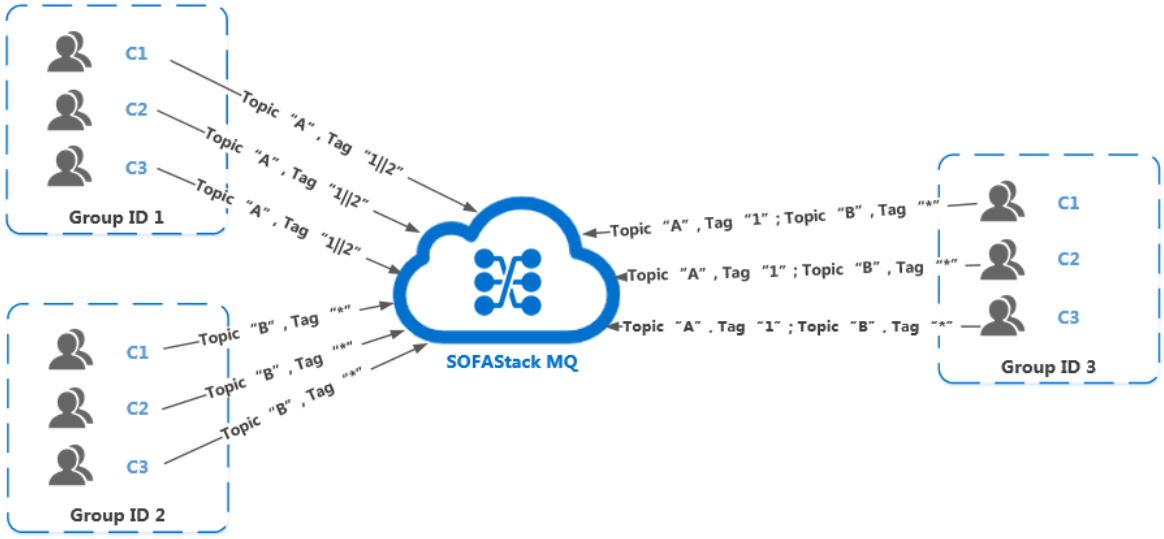
SOFAStack 消息队列里的一个消费者 Group ID 代表一个消费者群组。对于大多数分布式应用来说，一个消费者 Group ID 下通常会挂载多个消费者。

由于消息队列的订阅关系主要由 Topic 和 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

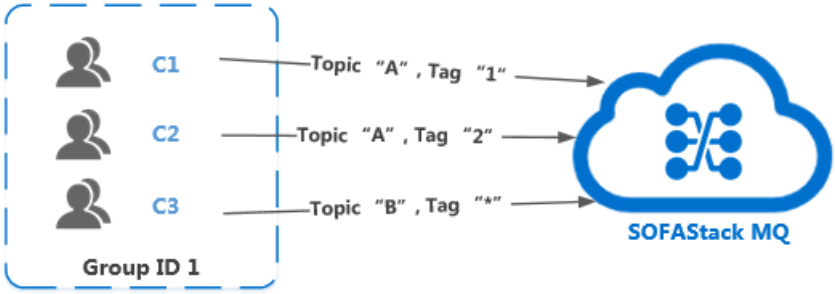
正确订阅关系图片示例

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

示例一

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;
    }
});
```

示例二

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;
    }
});
```

示例三

同一个 Group ID 下订阅 Topic 个数不一致。

- 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;
    }
});
```

同一个 Group ID 下订阅 Topic 的 Tag 不一致。

- 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

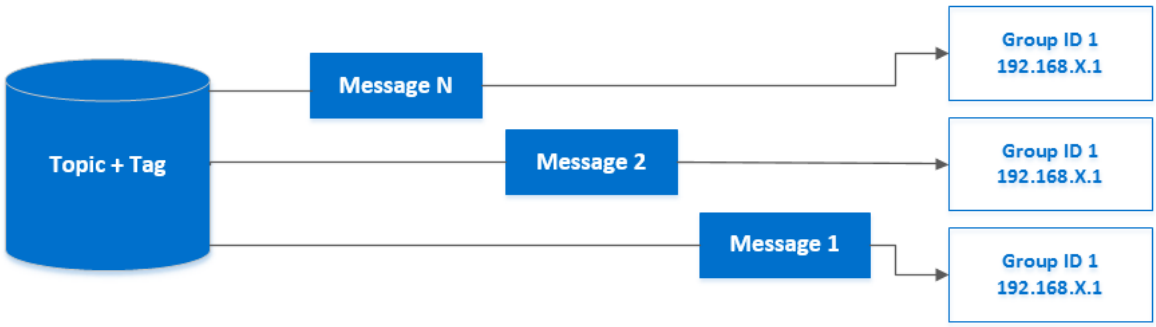
本文介绍 SOFAStack 消息队列的集群消费和广播消费的基本概念、适用场景以及注意事项。

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（单 Zone 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（单 Zone 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。

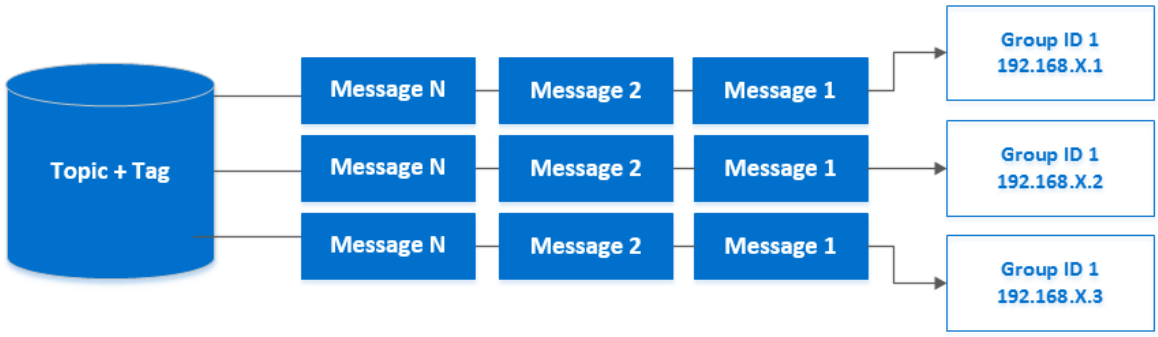


注意事项

- 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。
- 广播模式下不支持线下联调分组消息。
- 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。

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

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
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：

- SOFABOOT

```
// 配置对应 Group ID 的最大消息重试次数为 20 次
@MessageConsumer(group = "GID_xxx", topic = "TP_xxx", maxRetryTimes = 20)
```

- 非 SOFABOOT

```
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：

```
producer.messageBuilder().withTopic("TP_XXX").withTags("TAGA").withValue(orderPojo).build()
```

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：

- SOFABOOT 示例

```
import com.alipay.sofa.sofamq.api.MessageConsumer;
import com.alipay.sofa.sofamq.api.Messaging;

// 请使用 xml 或注解将该类配置为 Bean，只有 @Messaging 扫描不到
@Messaging
public class SomeClass {
    @MessageConsumer(group = "GID_XXX", topic = "TP_XXX", filter = "*")
    public void someMethodReceivePojo(OrderPojo somePojo) {
        // do something
    }
}
```

- 非 SOFABOOT 示例

```
consumer.subscribe("TP_XXX", "*", new GenericMessageListener<OrderPojo>() {
    @Override
    public Class<OrderPojo> payloadClass() {
        return OrderPojo.class;
    }

    @Override
    public Action consume(GenericMessage<OrderPojo> message, MessageConsumeContext cont
ext) {
        System.out.println(new Date() + " Receive message: " + message.getMsgID() + " " + messag
e.getValue());
        // 如果想测试消息重投的功能,可以将Action.CommitMessage 替换成Action.ReconsumeLater
        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：

- SOFABOOT 示例

```
import com.alipay.sofa.sofamq.api.MessageConsumer;
import com.alipay.sofa.sofamq.api.Messaging;

// 请使用 xml 或注解将该类配置为 Bean，只有 @Messaging 扫描不到
@Messaging
public class SomeClass {
    @MessageConsumer(group = "GID_XXX", topic = "TP_XXX", filter = "TAGA")
    public void someMethodReceivePojo(OrderPojo somePojo) {
        // do something
    }
}
```

- 非 SOFABOOT 示例

```
consumer.subscribe("TP_XXX", "TAGA", new GenericMessageListener<OrderPojo>() {
    @Override
    public Class<OrderPojo> payloadClass() {
        return OrderPojo.class;
    }

    @Override
    public Action consume(GenericMessage<OrderPojo> message, MessageConsumeContext context)
{
        System.out.println(new Date() + " Receive message: " + message.getMsgID() + " " + message.ge
tValue());
        // 如果想测试消息重投的功能,可以将Action.CommitMessage 替换成Action.ReconsumeLater
        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 GenericMessageListener<OrderPojo>() {
    @Override
    public Class<OrderPojo> payloadClass() {
        return OrderPojo.class;
    }

    @Override
    public Action consume(GenericMessage<OrderPojo> message, MessageConsumeContext context) {
        System.out.println(new Date() + " Receive message: " + message.getMsgID() + " " + message.getV
alue());
        // 如果想测试消息重投的功能,可以将Action.CommitMessage 替换成Action.ReconsumeLater
        return Action.CommitMessage;
    }
});
```

SQL92 过滤

过滤表达式中的 TAGS 特指消息的 TAG，其他的变量取自消息属性 `Properties` ，可通过 `Message#putUserProperties` 设置。

- SOFABOOT 示例

```
import static com.alipay.sofa.sofamq.api.MessageConsumer.SQL_FILTER;

import com.alipay.sofa.sofamq.api.MessageConsumer;
import com.alipay.sofa.sofamq.api.Messaging;

// 请使用 xml 或注解将该类配置为 Bean，只有 @Messaging 扫描不到
@Messaging
public class SomeClass {
    @MessageConsumer(group = "GID_XXX", topic = "TP_XXX", filter = "(TAGS in ('tag')) and a > 5", filte
rType = SQL_FILTER)
    public void someMethodReceivePojo(OrderPojo somePojo) {
        // do something
    }
}
```

- 非 SOFABOOT 示例

```
consumer.subscribe(MqConfig.TOPIC, MessageSelector.bySql("(TAGS in ('tag')) and a > 5"), new Mes
sageListenerImpl());
```

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 GenericMessageListener<OrderPojo>() {
    @Override
    public Class<OrderPojo> payloadClass() {
        return OrderPojo.class;
    }

    @Override
    public Action consume(GenericMessage<OrderPojo> message, MessageConsumeContext context) {
        System.out.println(new Date() + " Receive message: " + message.getMsgID() + " " + message.getV
alue());
        // 如果想测试消息重投的功能,可以将Action.CommitMessage 替换成Action.ReconsumeLater
        return Action.CommitMessage;
    }
});

consumer.subscribe("MQ_TOPIC", "TagB", new GenericMessageListener<OrderPojo>() {
    @Override
    public Class<OrderPojo> payloadClass() {
        return OrderPojo.class;
    }

    @Override
    public Action consume(GenericMessage<OrderPojo> message, MessageConsumeContext context) {
        System.out.println(new Date() + " Receive message: " + message.getMsgID() + " " + message.getV
alue());
        // 如果想测试消息重投的功能,可以将Action.CommitMessage 替换成Action.ReconsumeLater
        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了消息幂等。

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 也会收到相同的消息。
- 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：

```
OrderPojo orderPojo = getPojo();
Message message = producer.messageBuilder().withTopic("TP_XXX").withValue(orderPojo).build();
// set unique key
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 GenericMessageListener<OrderPojo>() {
    @Override
    public Class<OrderPojo> payloadClass() {
        return OrderPojo.class;
    }

    @Override
    public Action consume(GenericMessage<OrderPojo> message, MessageConsumeContext context) {
        String key = message.getKey();
        // 根据业务唯一标识的 Key 做幂等处理
        return Action.CommitMessage;
    }
});
```

3.快速入门

3.1. 操作链路概述

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
   i. 创建工作空间
   ii. 创建 Topic
   iii. 创建 Group ID
   iv. 获取 AK（AccessKey ID）和 SK（AccessKey Secret）
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 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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Topic 管理**。
3. 单击 **创建 Topic**，然后在 **创建 Topic** 对话框配置 Topic 信息：

参数	是否必填	说明
Topic	必填	Topic 格式要求如下： ○ Topic 只能包含英文、数字、 短横线（-）和下划线（_），其中英文和数字必须要有一种， 短横线（-）和下划线（_）可选。 ○ 长度需控制在 3~64 个字符之间。 ○ 命名不能以“CID”和“GID”开头。
消息类型	必填	支持的消息类型有普通消息、分区顺序消息、事务消息和定时消息。详细消息类型的说明，可参见 消息类型。
描述	选填	对该 Topic 的备注信息，长度限制在 256 个字符以内。

4. 单击 **确定**。

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. 登录消息队列控制台。

2. 在左侧导航栏，单击 **Group 管理**。
3. 单击 **创建 Group ID**，然后在 **创建 Group ID** 对话框配置 Group ID 信息：

参数	是否必填	说明
Group ID	必填	Group ID 格式要求如下： ○ 命名以 “GID_” 或者 “GID-” 开头。 ○ 只能包含字母、数字、短横线和下划线。 ○ 长度限制在 7~64 字符之间。 **说明** Group ID 一旦创建，则无法修改。
协议类型	必填	可选 **HTTP** 或 **TCP**。
描述	选填	对该 Group ID 的备注信息，长度限制在 256 个字符以内。

4. 单击 **确定**。

获取 AK（AccessKey ID）和 SK（AccessKey Secret）

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 AccessKeySecret。

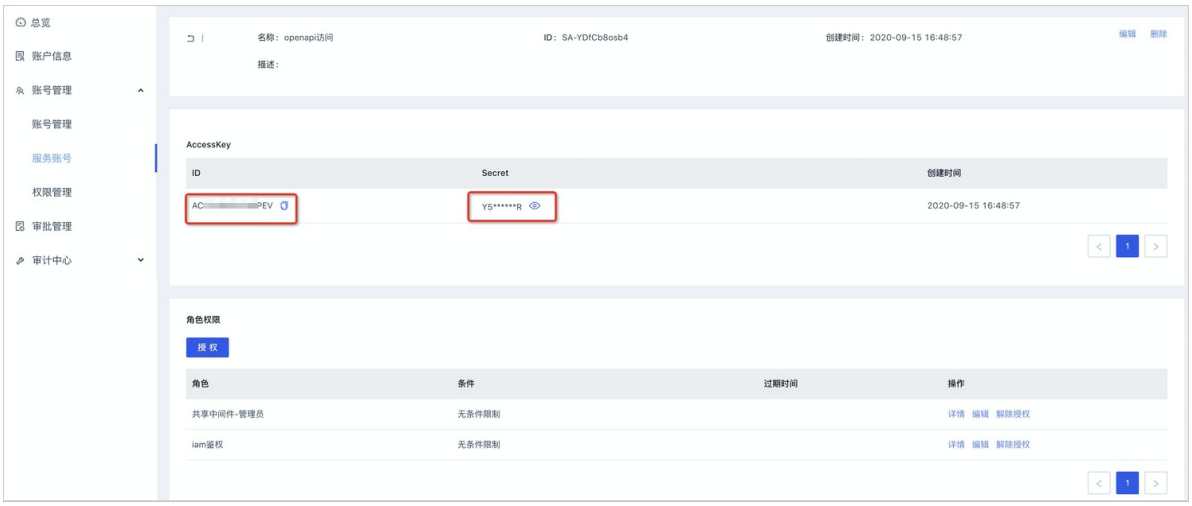
> **说明**
>
> 由于 RAM 是阿里云产品，非阿里云飞天底座输出的环境中可使用蚂蚁 IAM 创建访问密钥（AccessKey）做身份验证。

在 IAM 上获取 AccessKey 的具体步骤如下：

1. 生成服务账号。
   i. 登录 IAM 控制台。
   ii. 在左侧导航栏，选择 **账号管理 > 服务账号**。
   iii. 单击 **新增服务账号**，并在弹出的对话框中输入服务账号的名称和描述。
   iv. 单击 **OK**。
2. 为账号授予权限。
   i. 单击刚刚创建的账号右侧的 **授权**。
   ii. 按照步骤给账号授予合适的中间件权限。

   您可以给账号授予中间件开发者或管理员权限，或根据自己需要创建合适的权限授予该服务账号。
3. 获取 AK 和 SK。

   单击账号名称或右侧的 **详情**，进入服务账号详情页。在该页面的 **AccessKey** 区域查看 **ID** 和 **Secret** 信息。



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. 在左侧导航栏，单击 **概览**。

2. 在页面底部的 **接入配置** 找到 **实例 ID** 及 **内网接入点**。

3. 将 **内网接入点** 配置到客户端的 SDK 代码的 **ENDPOINT** 参数。
4. 将 **实例 ID** 配置到客户端的 SDK 代码的 **INSTANCE_ID** 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. 在左侧导航栏，单击 **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，单击 **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

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. 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：
   i. 在左侧导航栏，单击 **消息查询**。
   ii. 单击 **按 Message ID 查询**，在搜索框中输入发送消息后返回的 Message ID，单击 **搜索** 查询消息发送状态。

   存储时间 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。

   > **重要**
   >
   > 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。

1. 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。
   i. 在左侧导航栏，单击 **Group 管理**。
   ii. 单击目标 Group ID 名称进入详情页。
   iii. 单击 **订阅关系**。

      如果 **是否在线** 显示为 **在线**，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：

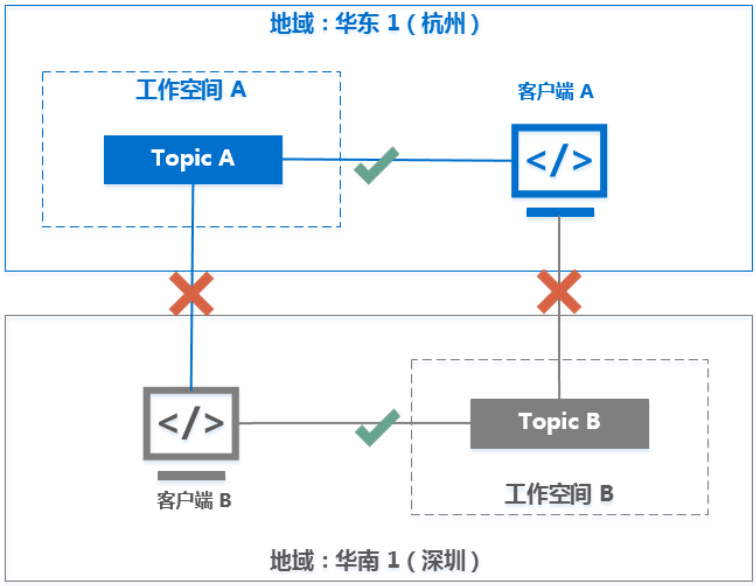
- GROUP_ID：您在消息队列控制台上创建的 Group ID。
- TOPIC：您在消息队列控制台上创建的 Topic。
- ENDPOINT：您从消息队列控制台的 **概览** > **接入配置** 获取的接入点。
- INSTANCE_ID：您从消息队列控制台的 **概览** > **接入配置** 获取的实例 ID。

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。

> **重要**
>
> 切忌将 Topic A 通过跨地域的方式，与客户端 B 通信，或将 Topic B 通过跨地域的方式，与客户端 A 通信。

具体对应关系如下图所示。



操作步骤

1. 登录消息队列控制台，选择工作空间。
2. 在左侧导航栏，单击 **概览**。
3. 在页面底部的 **接入配置** 找到 **内网接入点** 及 **实例 ID**。

4. 将该 **TCP 协议内网接入点** 配置到客户端 SDK 代码的 **ENDPOINT** 参数。
5. 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。

3.1.7（2021-04-10）

版本号	发布时间	下载链接
3.1.7	2021-04-10	sofamq-client-all 3.1.7

新特性

- 集成 Metrics/Lookout 2.0。
- 新增支持全站广播消费注解。
- 新增支持消费异步确认注解。
- 新增支持消费者注解动态订阅。
- 新增消费限流支持单元/Zone TPS 小于消费端机器数。

修复

- 修复积压日志有概率为负数问题。

3.1.6（2021-03-19）

版本号	发布时间	下载链接
3.1.6	2021-03-19	sofamq-client-all 3.1.6

新特性

- 支持 JDK 1.6。
- 支持全站广播消息。
- 支持消费异步确认。
- 升级 ANTVIP 至 1.5.4，支持跨域连接。
- 支持注解 noop schema url。

修复

- 修复消费端列表变动（发布、重启、扩缩容）后一段时间限流值偏小问题。
- 修复在特定流量和限流情况下限流值偏小问题。
- 修复 Tracer 不携带 rpcid 问题。

引入问题

- OrderConsumer 不可用。

3.1.5（2021-03-08）

版本号	发布时间	下载链接
3.1.5	2021-03-08	sofamq-client-all 3.1.5

新特性

- 支持 SOFABOOT 和 SOFA4 注解。
- 支持 JDK 1.8。

修复

- 修复限流值初次设置概率性可见性问题。

3.1.4（2021-01-19）

版本号	发布时间	下载链接
3.1.4	2021-01-19	sofamq-client-all 3.1.4

新特性

- 支持弹性 LDC：非弹性 Consumer 自动连接弹性 CELL。
- 支持消费端限流。
- 支持自动 Rebalance PullConsumer。
- 适配 sofa-common-tools 标准蚂蚁日志。
- 支持新版开发分组 2.0。
- CELL 强制转为大写。

修复

- 修复 3.1.1 引入 OrderConsumer 不能消费的问题。
- 修复 MAX_CACHED_MESSAGE_AMOUNT、MAX_CACHED_MESSAGE_SIZE_IN_MI_B 默认值不生效问题。

3.1.1（2020-08-13）

版本号	发布时间	下载链接
3.1.1	2020-08-13	sofamq-client-all 3.1.1

新特性

- 无

修复

- 修改 guava 包 shade 后的包名，避免冲突。
- 修复使用短域名 antvip://sofamqnamesrv-pool 时 AntVip Client 重复创建问题。
- 修复 antvip 客户端重复创建导致 FullGC 问题。

3.0.1（2021-02-06）

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（2021-01-09）

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、延时消息的配置以及测试代码。

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。

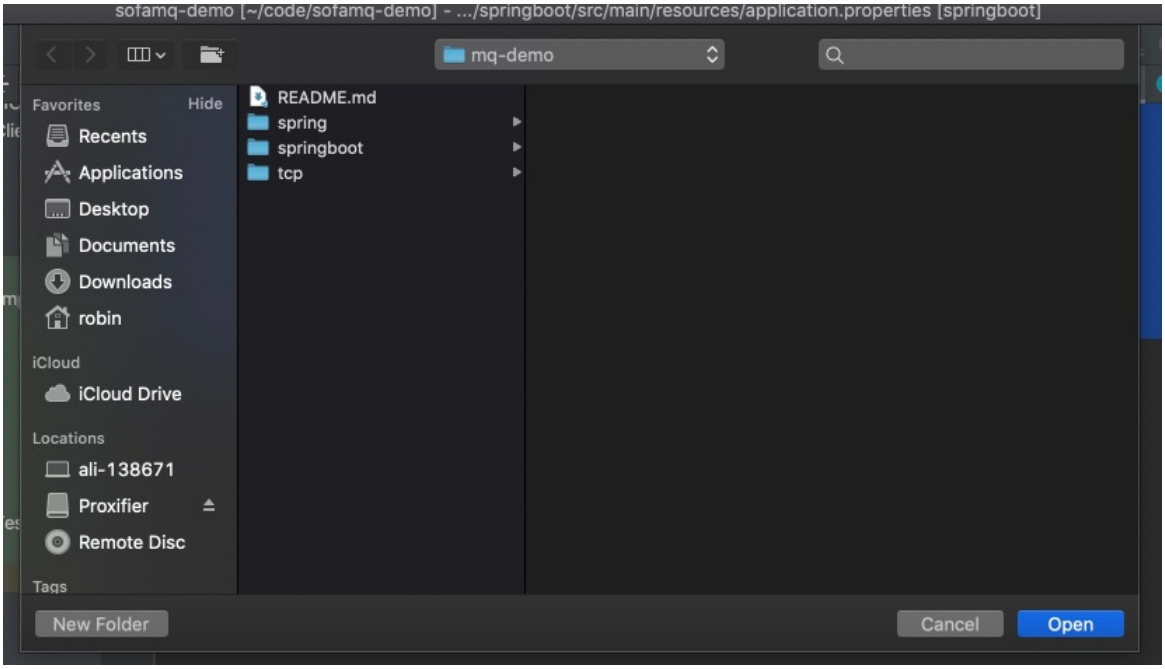
  在 https://github.com/sofastack-guides/sofamq-demo 下载 Demo 工程到本地，然后解压即可看到本地新增了 sofamq-demo 文件夹。

- 下载安装 JDK。

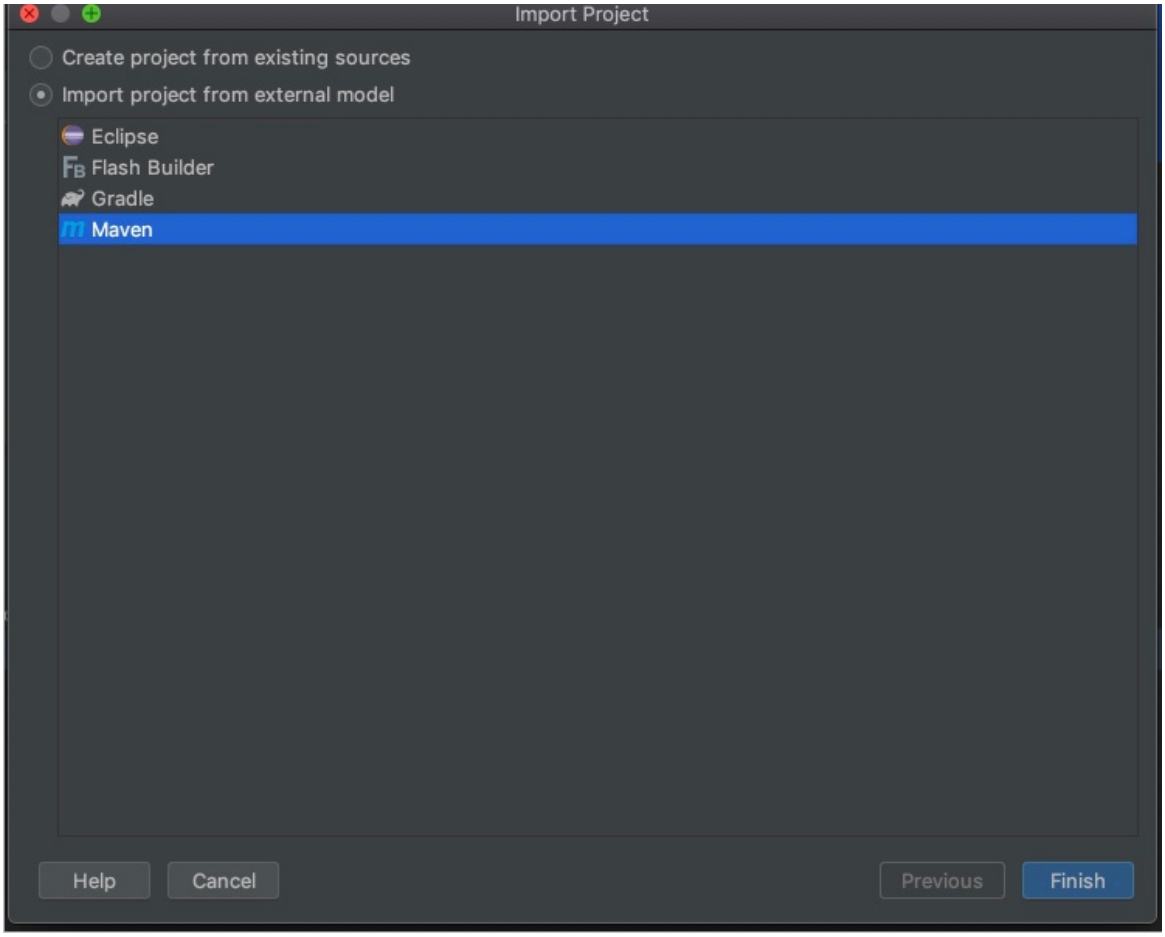
配置 Demo 工程

1. 将 Demo 工程文件导入 IntelliJ IDEA。

   i. 在 IntelliJ IDEA 界面，单击 **New** > **Project From Existing Sources...**，选择 sofamq-demo 文件夹。



ii. 选择 Import 类型为 Maven。



iii. 默认单击 Finish，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）。

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` 。

   i. 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";
```

ii. 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。

   检查方法消息发送成功的步骤如下：

   i. 登录消息队列控制台。

   ii. 在左侧导航栏，单击 **消息查询**。

   iii. 在 **消息查询** 页面，您可单击 **按 Topic 查询**。

   iv. 单击目标 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。

   检查方法消息发送成功的步骤如下：

   i. 登录消息队列控制台。

   ii. 在左侧导航栏，单击 **Group 管理**。

   iii. 单击目标 Group ID 进入详情页。

iv. 单击 **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）

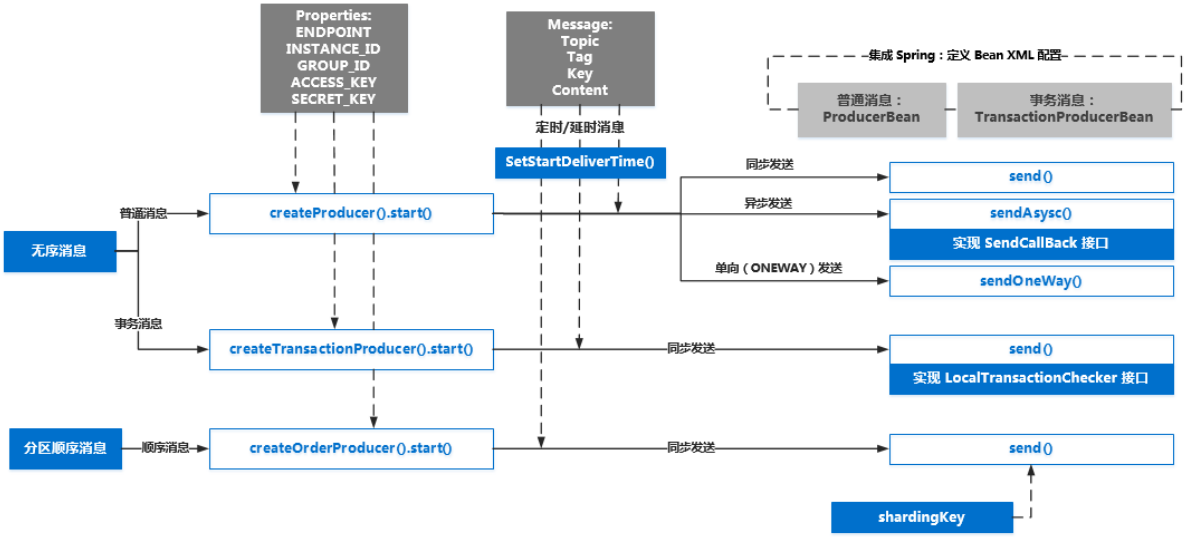
4.3. 接口和参数说明

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。

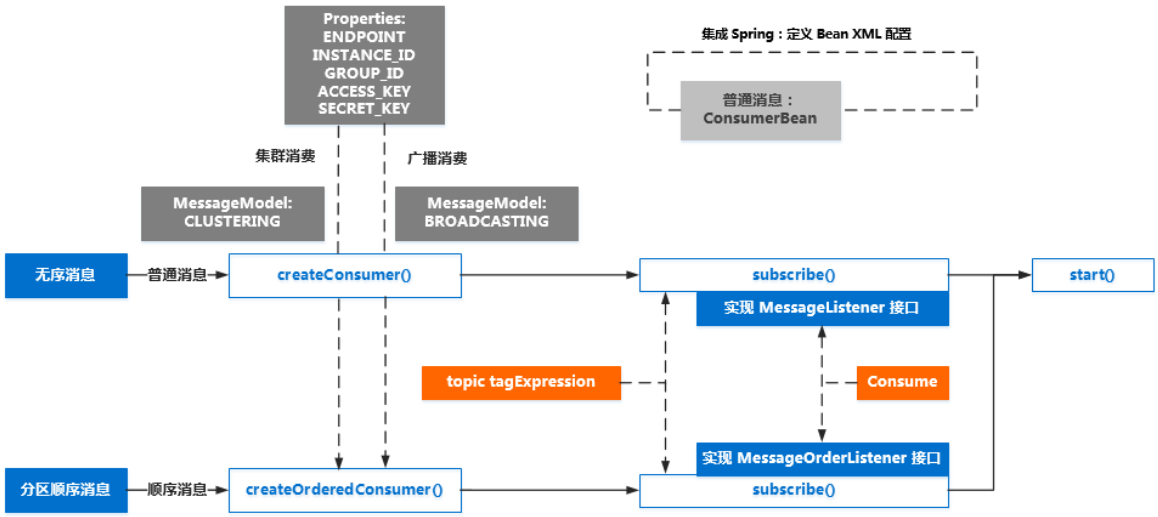
消息发送接口



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。默认值为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 基础术语。
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，默认值为 20。
MAX_RECONSUME_TIMES	设置消息消费失败的最大重试次数，默认值为 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。 默认值为 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，单位：条。默认值为 1000。
MAX_CACHED_MESSAGE_SIZE_IN_MI_B	客户端本地的最大缓存消息大小，默认值：512 MB。取值范围为 16 MB ~ 2 GB。

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

在运行收发消息的 Java 代码前，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

1. 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;
   }
}
```

2. 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。

1. 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TransactionSt
atus
   }
}
```

2. 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.getMqPropertie
s())
      .createTransactionProducer(mqConfig.getMqProperties(), localTransactionChecker);
      return producer;
   }
}
```

3. 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 =newMessage(mqConfig.getTopic(),"TagA","Hello MQ".getBytes());
      SendResult sendResult = transactionProducer.send(msg,newLocalTransactionExecuter(){
         @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TransactionSt
atus
         }
      },null);
      System.out.println(sendResult);
   }
}
```

消费者与 SpringBoot 集成

1. 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;
      }
   }
}
```

2. 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。

三种发送方式的对比

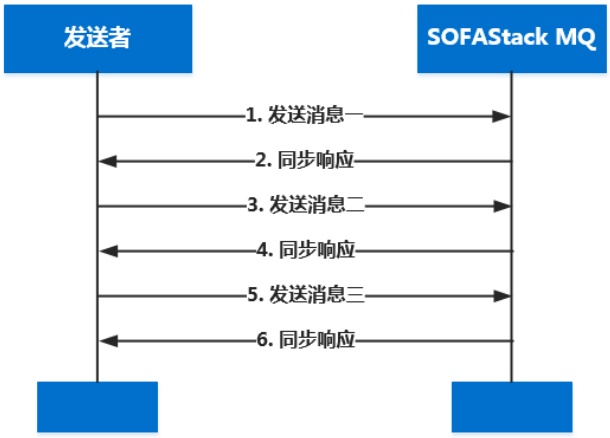
三者的特点和主要区别如下：

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

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。



示例代码

- **SOFABOOT**

```
import com.alibaba.fastjson.JSON;
import com.alipay.sofa.sofamq.api.Messaging;
import com.alipay.sofa.sofamq.api.MessageProducer;
import com.alipay.sofa.sofamq.api.Producer;

// 请使用 xml 或注解将该类配置为 Bean，只有 @Messaging 扫描不到
@Messaging
public class SomeClass {
    @MessageProducer(group = "GID_XXX", topic = "TP_XXX", payload = OrderPojo.class)
    public Producer<OrderPojo> producer;

    public void someMethod() {
        OrderPojo orderPojo = getPojo();
        producer.send(producer.messageBuilder().withTags("TAGA").withValue(orderPojo).build());
    }
}
```

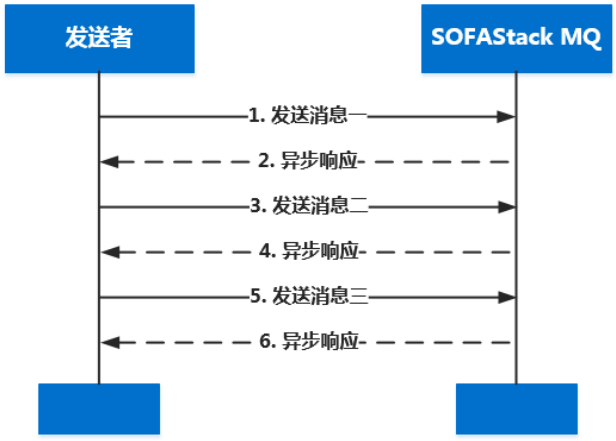
- 非 SOFABOOT

```
import java.util.Properties;

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

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

public class SomeClass {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(PropertyKeyConst.GROUP_ID, "GID_XXX");
        properties.setProperty(PropertyKeyConst.CELL, "GZ00B");
        Producer producer = OMS.builder().driver("sofamq").endpoint("antvip://sofamqnamesrv-pool.g
z00b.alipay.net")
            .schemaRegistryUrl("antvip://openmeta-pool.gz00b.alipay.net")
            .build().createProducer(properties);
        producer.start();

        OrderPojo orderPojo = getPojo();
        SendResult sendResult = producer.send(producer.messageBuilder().withTopic("TP_XXX").withT
ags("TAGA")
                                .withProperty("propkey", "propvalue").withValue(orderPojo).build());
        System.out.println(sendResult);
    }
}
```

异步发送

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。



示例代码

- **SOFABOOT**

```
import com.alipay.sofa.sofamq.api.MessageProducer;
import com.alipay.sofa.sofamq.api.Messaging;
import com.alipay.sofa.sofamq.api.Producer;

import io.openmessaging.api.OnExceptionContext;
import io.openmessaging.api.SendCallback;
import io.openmessaging.api.SendResult;

// 请使用 xml 或注解将该类配置为 Bean，只有 @Messaging 扫描不到
@Messaging
public class SomeClass {
   @MessageProducer(group = "GID_XXX", topic = "TP_XXX", payload = OrderPojo.class)
   public Producer<OrderPojo> producer;

   public void someMethod() {
      OrderPojo orderPojo = getPojo();
      Message message = producer.messageBuilder().withTags("TRADE").withValue(orderPojo).build
();
      producer.sendAsync(message,
         new SendCallback() {
            @Override
            public void onSuccess(SendResult sendResult) {
               System.out.println("send success: " + sendResult);
            }

            @Override
            public void onException(OnExceptionContext context) {
               System.out.println("send exception: " + context);
            }
         });
   }
}
```

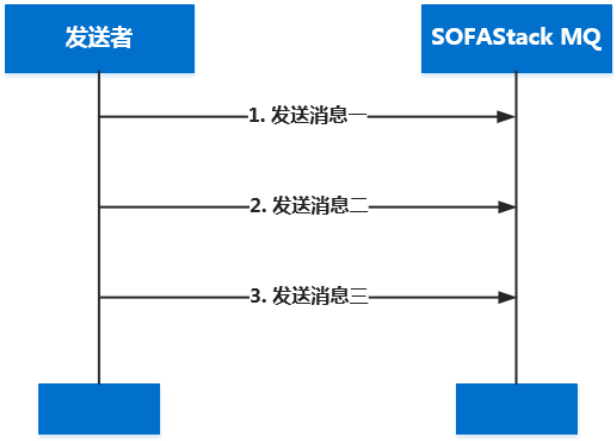
- 非 SOFABOOT

```
import java.util.Properties;

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

import io.openmessaging.api.OMS;
import io.openmessaging.api.OnExceptionContext;
import io.openmessaging.api.Producer;
import io.openmessaging.api.SendCallback;
import io.openmessaging.api.SendResult;

public class SomeClass {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(PropertyKeyConst.GROUP_ID, "GID_XXX");
        properties.setProperty(PropertyKeyConst.CELL, "GZ00B");
        Producer producer = OMS.builder().driver("sofamq").endpoint("antvip://sofamqnamesrv-pool.g
z00b.alipay.net")
            .schemaRegistryUrl("antvip://openmeta-pool.gz00b.alipay.net")
            .build().createProducer(properties);
        producer.start();

        OrderPojo orderPojo = getPojo();
        Message message = producer.messageBuilder().withTopic("TP_XXX").withTags("TRADE").withV
alue(orderPojo).build();
        producer.sendAsync(message, new SendCallback() {
                @Override
                public void onSuccess(SendResult sendResult) {
                    System.out.println("send success: " + sendResult);
                }

                @Override
                public void onException(OnExceptionContext context) {
                    System.out.println("send fail: " + context);
                }
            });

    }
}
```

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。



示例代码

- **SOFABOOT**

```
import com.alipay.sofa.sofamq.api.MessageProducer;
import com.alipay.sofa.sofamq.api.Messaging;
import com.alipay.sofa.sofamq.api.Producer;

// 请使用 xml 或注解将该类配置为 Bean，只有 @Messaging 扫描不到
@Messaging
public class SomeClass {
   @MessageProducer(group = "GID_XXX", topic = "TP_XXX", payload = OrderPojo.class)
   public Producer<OrderPojo> producer;

   public void someMethod() {
      OrderPojo orderPojo = getPojo();
      Message message = producer.messageBuilder().withTags("TRADE").withValue(orderPojo).build
();
      producer.sendOneway(message);
   }
}
```

- 非 SOFABOOT

```
import java.util.Properties;

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

import io.openmessaging.api.OMS;
import io.openmessaging.api.Producer;

public class SomeClass {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(PropertyKeyConst.GROUP_ID, "GID_XXX");
        properties.setProperty(PropertyKeyConst.CELL, "GZ00B");
        Producer producer = OMS.builder().driver("sofamq").endpoint("antvip://sofamqnamesrv-pool.gz00b.alipay.net")
            .schemaRegistryUrl("antvip://openmeta-pool.gz00b.alipay.net")
            .build().createProducer(properties);
        producer.start();

        OrderPojo orderPojo = getPojo();
        Message message = producer.messageBuilder().withTopic("TP_XXX").withTags("TRADE").withValue(orderPojo).build();
        producer.sendOneway(message);
    }
}
```

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 的能力。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");
        finalProducer 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(newRunnable() {
            @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(newString(message.getBody()));
                return OrderAction.Success;
            }
        });
        consumer.start();
    }
}
```

4.10. 收发事务消息

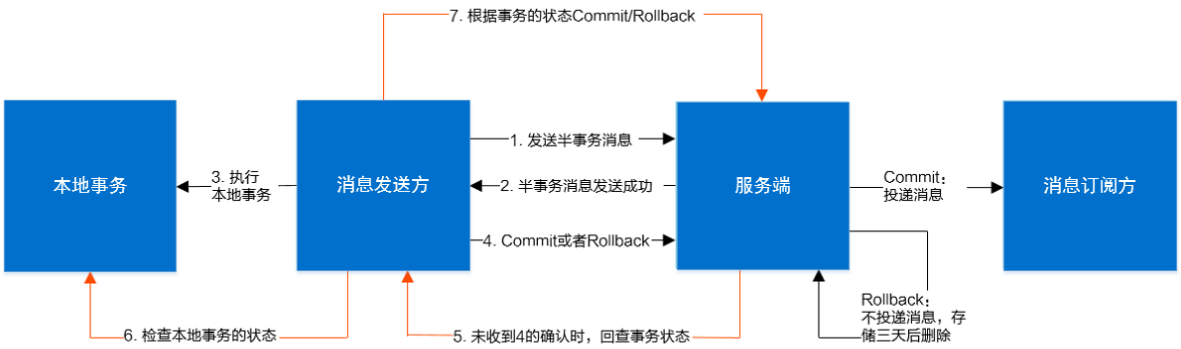
本文提供使用 TCP 协议下的 Java SDK 收发事务消息的示例代码供您参考。

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：

1. 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").wit
hCredentials(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, newLocalT
ransactionChecker() {
         @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：
  i. 检查该半事务消息对应的本地事务的状态（committed or rollback）。
  ii. 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

本文介绍如何设置 LDC 单元化相关参数。

说明

该功能仅适用于支持 LDC 单元化架构的环境。

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.datace
nter=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.acces
s=XXX
      //      properties.setProperty(PropertyKeyConst.SECRET_KEY, "XXX"); // -Dcom.antcloud.mw.secre
t=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.datace
nter=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.acces
s=XXX
      //      properties.setProperty(PropertyKeyConst.SECRET_KEY, "XXX"); // -Dcom.antcloud.mw.secre
t=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。

消息队列
概览
Topic 管理
Group 管理
消息查询
消息轨迹
消息路由
消息路由
创建路由任务
消息源
* 源topic：
TPTEST
消息源
* 目标单元：
RZONE
* 目标 Topic：
TPTEST
* 起始同步位点：
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。
描述：
请输入描述
取 消
确 定
清 除
搜 索
创建时间
操作

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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Topic 管理**。
3. 单击 **创建 Topic**，然后在 **创建 Topic** 对话框配置 Topic 信息：
   - **Topic**：Topic 名称。
     - Topic 只能包含英文、数字、 短横线（-）和下划线（_），其中英文和数字必须要有一种， 短横线（-）和下划线（_）可选。
     - 长度需控制在 3~64 个字符之间。
     - 命名不能以“CID”和“GID”开头。
   - **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，长度限制在 256 个字符以内。

4. 单击 **确定**。

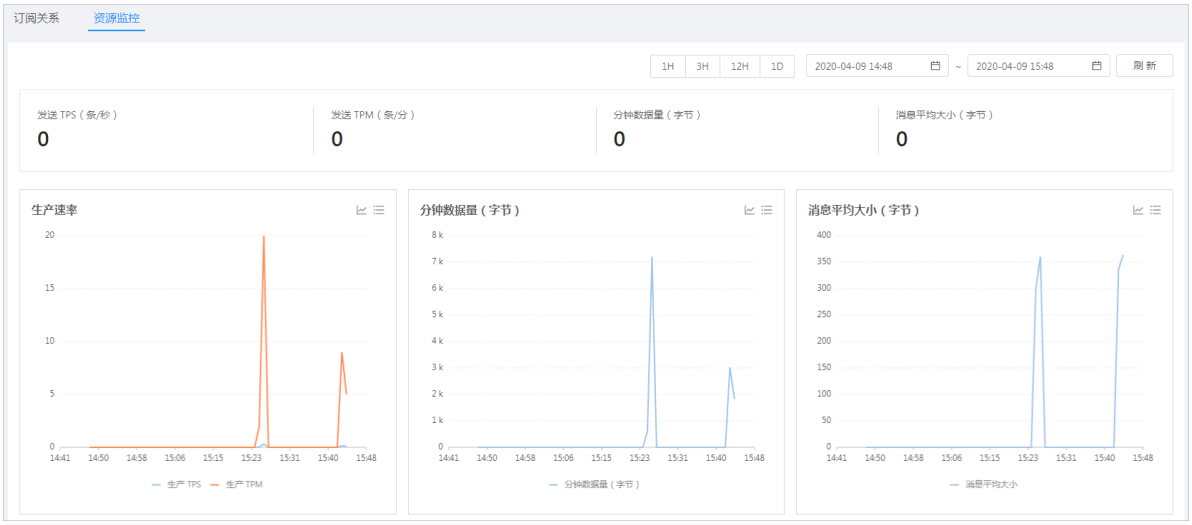
Topic 创建成功后，您可以选择进入详情页或 **发送测试消息**。

查看 Topic 详情

在 Topic 列表中，单击目标 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、最近发现时间和描述。
- **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。

- **资源监控**：可查看选定时段内该 Topic 的每分钟与每秒钟的消息生产数据、每分钟的数据量、消息平均大小等相关数据变化信息，如下图所示。



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，单击目标 Topic 操作列的 **发送测试消息**。
2. 在 **发送测试消息** 对话框配置以下信息：
   - **Tag**：可选，消息标签，用于对 Topic 下的消息进行二级分类。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。

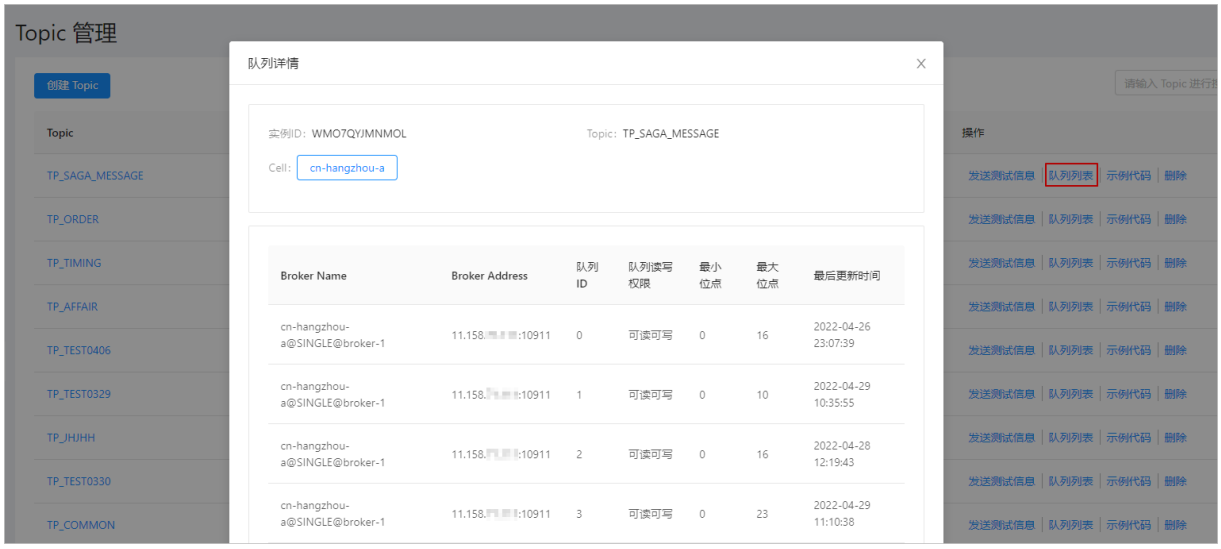
- **消息体**：必填，需要发送的具体消息内容。

3. 单击 **确定**，即可发送测试消息。

消息发送后，您可以直接单击 **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。

查看队列列表

在 Topic 列表中，单击目标 Topic 操作列的 **队列列表** 即可查看该 Topic 下各个队列信息详情，包括 Broker Name、 Broker Address、队列 ID、队列读写权限等。



说明

您也可以单击目标 Topic 名称，进入 Topic 详情页，单击页面右上角 **队列列表** 查看队列详情。

删除 Topic

1. 在 Topic 列表中，单击目标 Topic 操作列的 **删除**。
2. 在弹出的对话框中，单击 **确定**。

> **重要**
>
> 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。

后续操作

- 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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Group 管理**。
3. 单击 **创建 Group ID**，然后在 **创建 Group ID** 对话框配置 Group ID 信息：

- **Group ID**：必填。
  - 命名以 “GID_” 或者 “GID-” 开头。
  - 只能包含字母、数字、短横线和下划线。
  - 长度限制在 7-64 字符之间。

> **说明**
>
> Group ID 一旦创建，则无法修改。

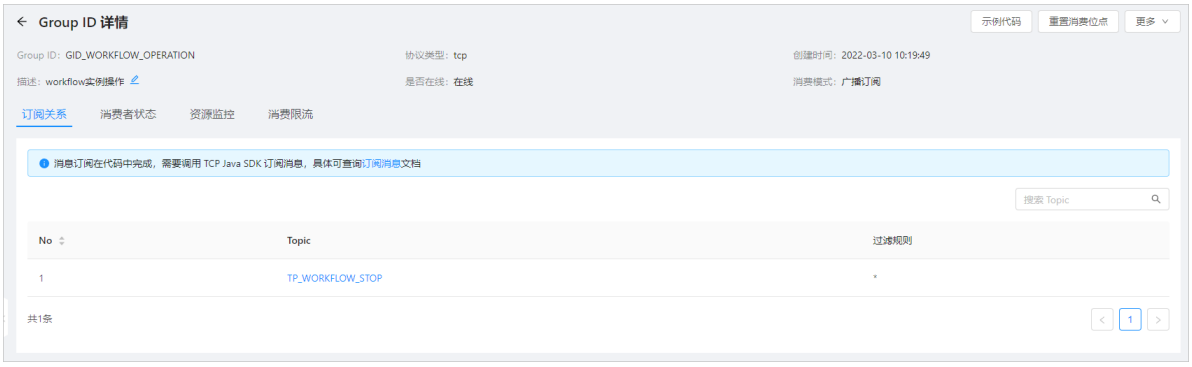
- **协议类型**：必填，支持选择 HTTP 或 TCP 协议类型。
- **描述**：可选，对该 Group ID 的备注信息，长度限制在 256 个字符以内。

4. 单击 **确定**。

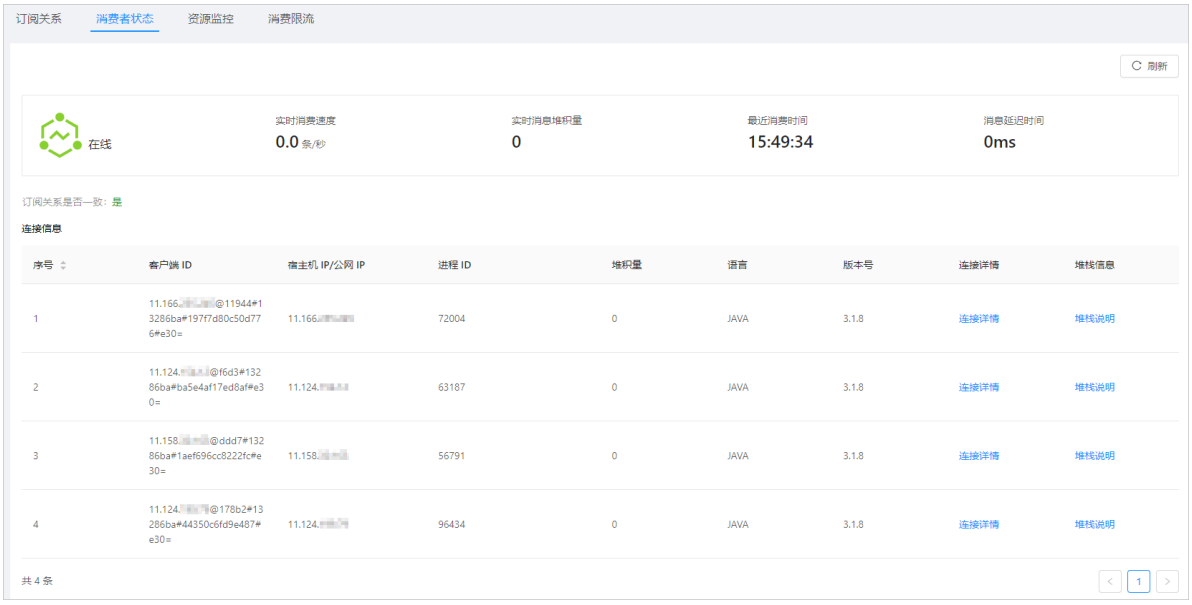
查看 Group ID 详情

在 Group ID 列表中，单击目标 Group ID 名称，即可进入该 Group ID 详情页。详情页提供了以下信息：

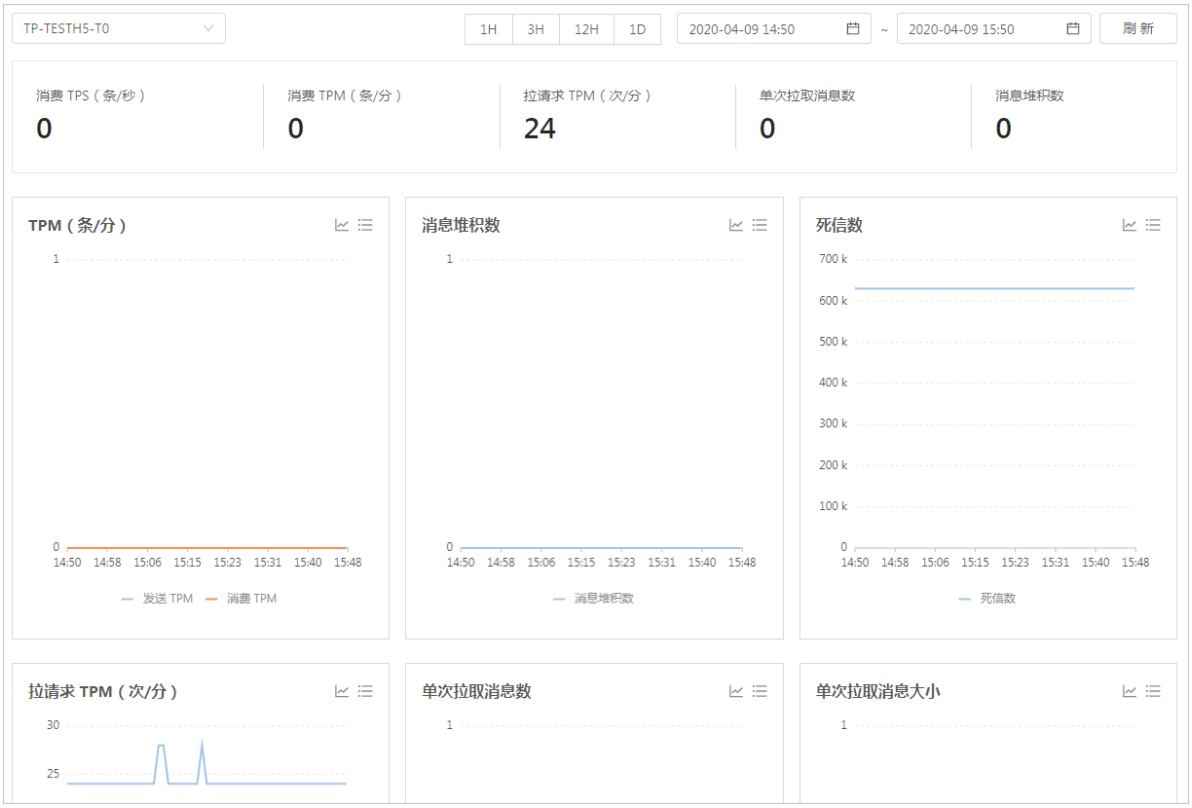
- **基本信息**：包括 Group ID、协议类型、创建时间、在线状态、消费模式和描述。
- **订阅关系**：包括该 Group ID 订阅的 Topic 及其相关信息，如 Topic 名称、过滤规则等。



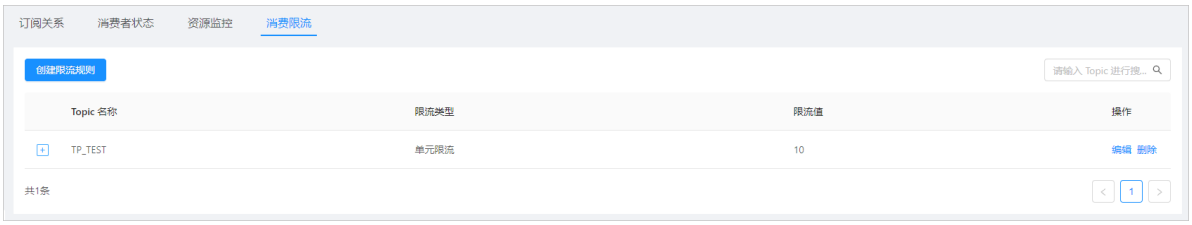
- **消费者状态：**可查看实时消息速度、实时消息堆积量、最近消费时间、消息延迟时间以及连接信息等，如下图所示。详细消费者状态说明，可参见 查看消费者状态。



- **资源监控：**可查看选定时段内指定 Topic 的消息消费数据（TPS/TPM）、每分钟拉请求数、单次拉取数量以及消息堆积数量等，如下图所示。



- **消费限流**：可查看限流类型和限流值，如下图所示。



删除 Group ID

1. 在 Group ID 列表中，单击目标 Group ID 操作列的 **删除**。
2. 在弹出的对话框中，单击 **确定**。

> **重要**
>
> 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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Topic 管理**。
3. 单击目标 Topic 进入详情页，即可看到所有订阅该 Topic 的在线 Group ID 及其消费模式。

此外，您还可以单击任一 Group ID 操作列的 **消费状态**，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

在 LDC 单元化架构环境下，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. 在左侧导航栏，单击 **Group 管理**。
2. 单击目标 Group ID 名称进入详情页，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

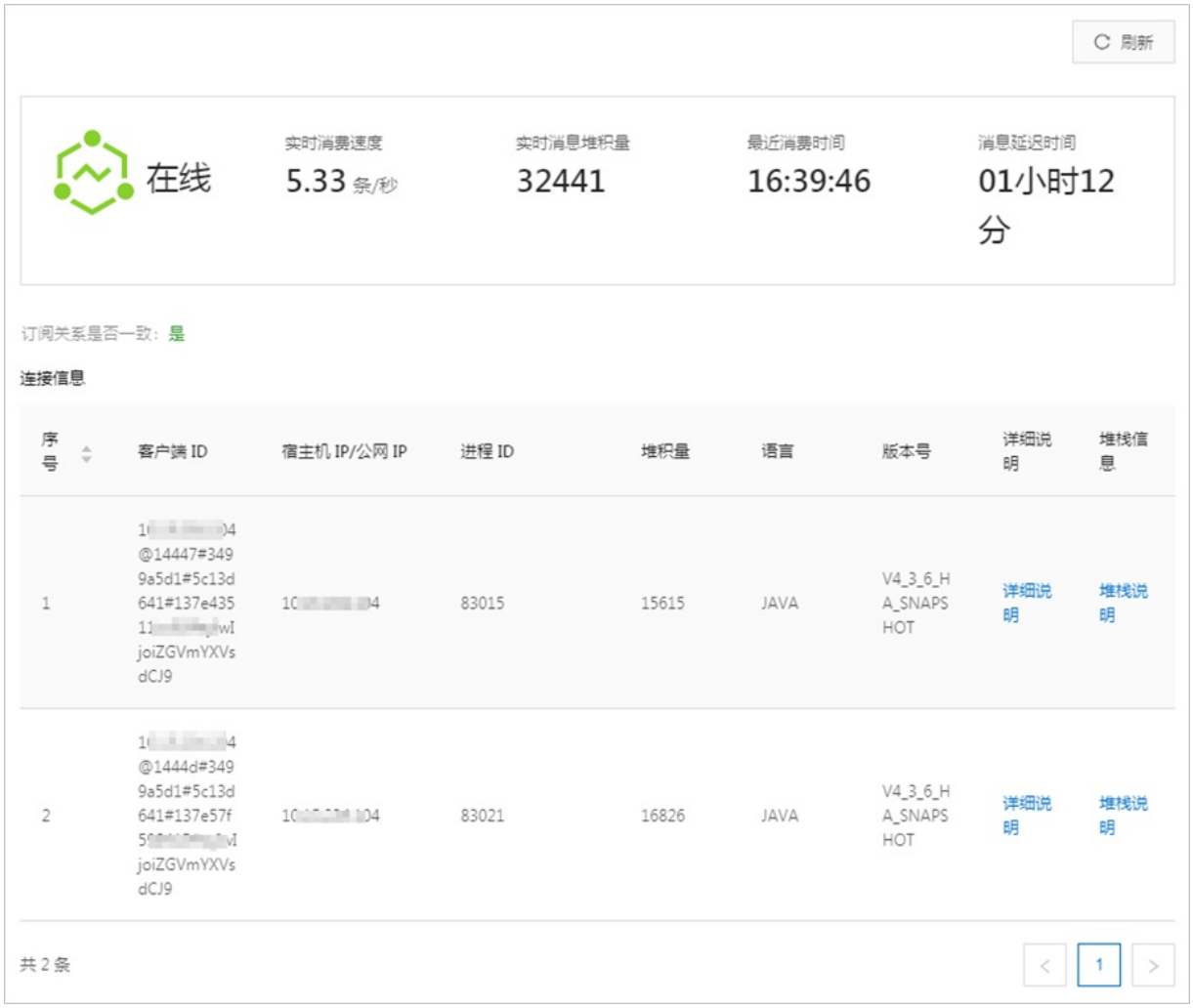
在 LDC 单元化架构环境下，可以自由切换单元，查看指定单元下该 Group ID 订阅的 Topic，如下图所示。

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

1. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Group 管理**。
3. 单击目标 Group ID 进入详情页。
4. 单击 **消费者状态**，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。



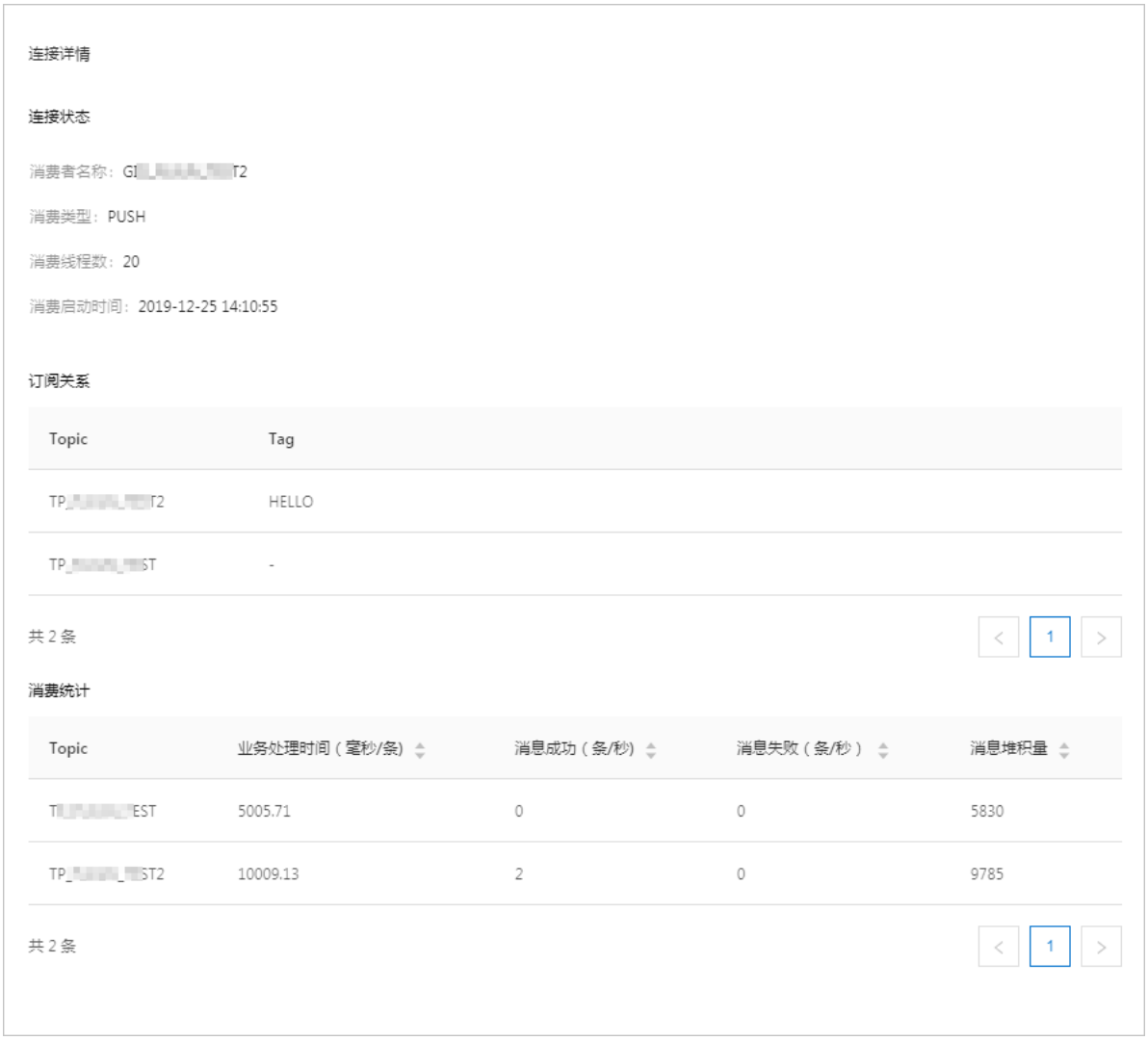
单元化说明

在 LDC 单元化架构环境下，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** 页面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。

- 单击目标消费者右侧的 **连接详情**，即可查看更多连接详情。



- 单击目标消费者右侧的 **堆栈说明**，即可查看该消费者实例当前进程的堆栈信息。

连接堆栈

Thread	Stack
ConsumeMessageThread_7	TID: 59 STATE: TIMED_WAITING java.lang.Thread.sleep(Native Method) com.alipay.fujun.SofamqConsumer2$2.consume(SofamqConsumer2.java:54) com.alipay.sofa.sofamq.client.ConsumerImpl$MessageListenerImpl.consumeMessage(ConsumerImpl.java:89) com.alipay.sofa.sofamq.shade.org.apache.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentlyService.java:420) java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) java.util.concurrent.FutureTask.run(FutureTask.java:266) java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142) java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) java.lang.Thread.run(Thread.java:745)
ConsumeMessageThread_11	TID: 63 STATE: TIMED_WAITING java.lang.Thread.sleep(Native Method) com.alipay.fujun.SofamqConsumer2$2.consume(SofamqConsumer2.java:54) com.alipay.sofa.sofamq.client.ConsumerImpl$MessageListenerImpl.consumeMessage(ConsumerImpl.java:89) com.alipay.sofa.sofamq.shade.org.apache.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentlyService.java:420) java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) java.util.concurrent.FutureTask.run(FutureTask.java:266) java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142) java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) java.lang.Thread.run(Thread.java:745)
ConsumeMessageThread_2	TID: 53 STATE: TIMED_WAITING java.lang.Thread.sleep(Native Method) com.alipay.fujun.SofamqConsumer2$2.consume(SofamqConsumer2.java:54) com.alipay.sofa.sofamq.client.ConsumerImpl$MessageListenerImpl.consumeMessage(ConsumerImpl.java:89) com.alipay.sofa.sofamq.shade.org.apache.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentlyService.java:420) java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) java.util.concurrent.FutureTask.run(FutureTask.java:266) java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142) java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) java.lang.Thread.run(Thread.java:745)
ConsumeMessageThread_15	TID: 67 STATE: TIMED_WAITING java.lang.Thread.sleep(Native Method) com.alipay.fujun.SofamqConsumer2$2.consume(SofamqConsumer2.java:54) com.alipay.sofa.sofamq.client.ConsumerImpl$MessageListenerImpl.consumeMessage(ConsumerImpl.java:89) com.alipay.sofa.sofamq.shade.org.apache.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentlyService.java:420) java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) java.util.concurrent.FutureTask.run(FutureTask.java:266) java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)

关闭

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。

> **重要**
>
> 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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Group 管理**。
3. 单击目标 Group ID 操作列的 **重置消费位点**，然后在 **重置消费位点** 对话框配置以下信息：

- **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. 单击 **确定**。

单元化说明

在 LDC 单元化架构环境中，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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **消息查询**。
3. 在 **消息查询** 页面，您可单击以下任一页签，然后按页面提示输入相应信息。
   - **按 Message ID 查询**：Message ID 是由服务端生成的 32 位字符串，用于标识一条消息。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。
4. 信息输入完毕后，单击 **搜索**。

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 查询消息轨迹 以及查看消息详情。

单元化说明

在 LDC 单元化架构环境下，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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **消息轨迹**。
3. 单击 **创建查询任务**，您可选择以下查询维度，然后按页面提示输入相应信息。
   - **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 64 条消息轨迹。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. 单击 **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. 在消息轨迹页面，单击 **刷新**，直到状态切换至查询完成。单击展开图标可查看到轨迹的简要信息，包括消息本身的属性以及接收状态的信息。
6. 单击 **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：

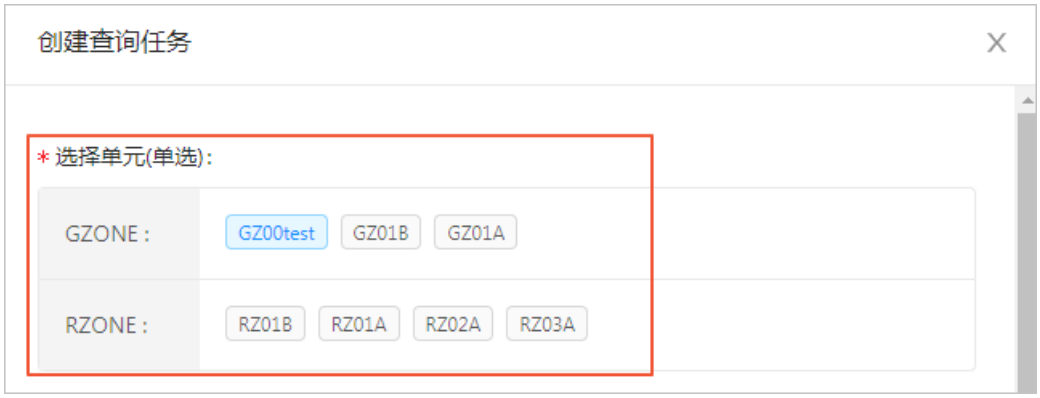
- **生产者**：
  - **发送时间**：消息从生产者发送时的客户端时间戳。
  - **发送耗时**：生产者调用 Send 方法发送消息的毫秒耗时。
  - **发送状态**：
    - 发送成功：消息发送成功，服务端已经存储成功。
    - 发送失败：消息发送失败，服务端没有存储消息，需要重试。
    - 消息定时中：该消息是定时或者延时消息，且尚未到达投递时间。
    - 事务未提交：该消息是事务消息，且尚未提交状态。
    - 事务回滚：该消息是事务消息，并且已经回滚。

- **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。

> **说明**
>
> 如需删除某个查询任务，可在消息轨迹任务列表页找到目标任务，单击其操作列的 **删除**，按提示完成删除。

单元化说明

在 LDC 单元化架构环境下，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. 死信队列

当一条消息初次消费失败，消息队列会自动进行消息重试。达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息。此时，消息队列不会立刻将消息丢弃，而是将其发送到该消费者对应的特殊队列中。在消息队列中，这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），死信队列（Dead-Letter Queue）用于存储及处理这些无法被正常消费的死信消息。

本文通过介绍如何查询和重新发送进入死信队列的死信消息，以便您按需管理死信消息，避免消息漏处理。

特性说明

死信队列

- 一个死信队列对应一个 Group ID， 而不是对应单个消费者实例。
- 如果一个 Group ID 未产生死信消息，消息队列不会为其创建相应的死信队列。
- 一个死信队列包含了对应 Group ID 产生的所有死信消息，不论该消息属于哪个 Topic。

死信消息

- 不会再被消费者正常消费。
- 有效期与正常消息相同，均为 3 天，3 天后会被自动删除。因此，请在死信消息产生后的 3 天内及时处理。

查询死信消息

消息队列提供以下两种查询死信消息的方式：

- 按 Group ID 查询：根据 Group ID 和时间范围，批量获取符合条件的所有消息；查询量大，不易匹配。
- 按 Message ID 查询：根据 Group ID 和 Message ID 可以精确定位任意一条消息。

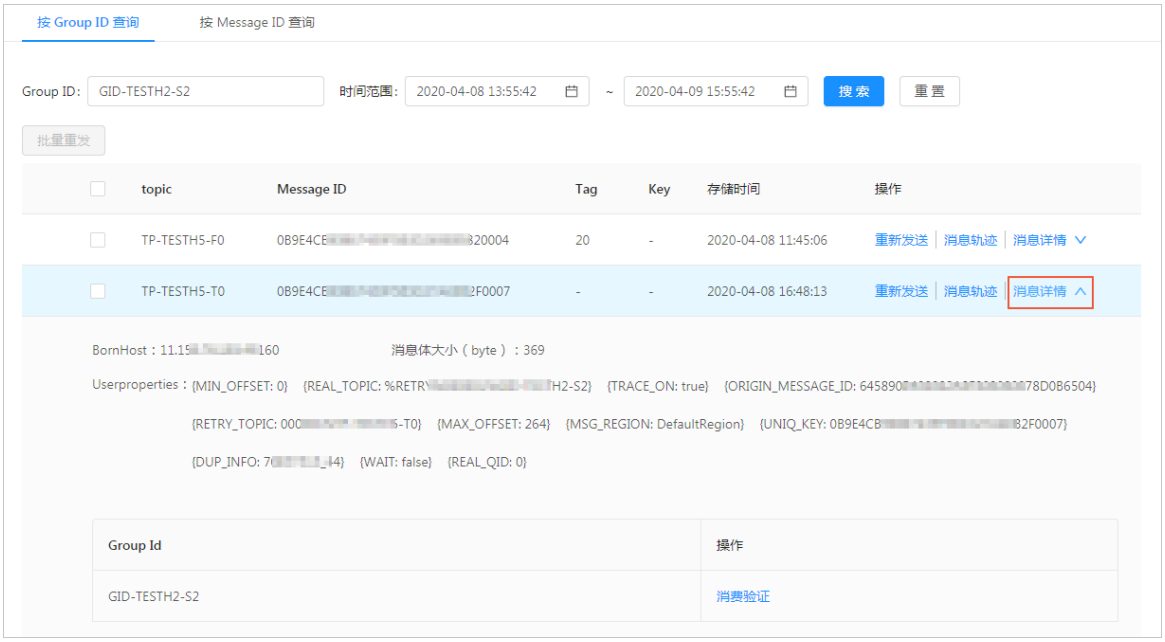
按 Group ID 查询

您可以根据 Group ID 和死信消息产生的时间范围，批量查询该 Group ID 在某段时间内产生的所有死信消息。

> **说明**
>
> 死信消息产生的时间是指一条消息在投递重试达到最大次数后被发送到死信队列的时间。

操作步骤

1. 登录消息队列控制台。
2. 在左侧导航栏，单击 **死信队列**。
3. 单击 **按 Group ID 查询**。在 Group ID 的输入框中，输入关键词搜索想要查找的 Group ID，然后选中，或直接在下拉列表中选择想要查找的 Group ID。
4. 单击时间选择框，选择想要查询的时间范围的起止日期和具体时间点。
5. 单击 **搜索**，页面中会显示所有符合以上条件的死信消息。
   - 单击目标死信消息操作列的 **消息详情**，即可查看该消息的详细信息，包括消息体大小、基本属性等。



   - 单击目标死信消息操作列中的 **消息轨迹**，即可进入 **消息轨迹** 页面，查询该消息轨迹。详见 查询消息轨迹。

按 Message ID 查询

按 Message ID 查询消息属于精确查询。您可以根据 Group ID 与 Message ID 精确查询到任意一条消息。

操作步骤

1. 登录消息队列控制台。
2. 在左侧导航栏，单击 **死信队列**。
3. 单击 **按 Message ID 查询**，在 Group ID 的输入框中，输入关键词搜索想要查找的 Group ID，然后选中，或直接在下拉列表中选择想要查找的 Group ID。
4. 在 Message ID 的输入框中，输入想要查找的消息的 Message ID。
5. 单击 **搜索**，页面中会显示所有符合以上条件的死信消息。

- 单击目标死信消息操作列的 **消息详情**，即可查看该消息的详细信息，包括基本属性、消息体的下载链接等。

- 单击目标死信消息操作列中的 **消息轨迹**，即可进入 **消息轨迹** 页面，查询该消息轨迹。详见 查询消息轨迹。

消费验证

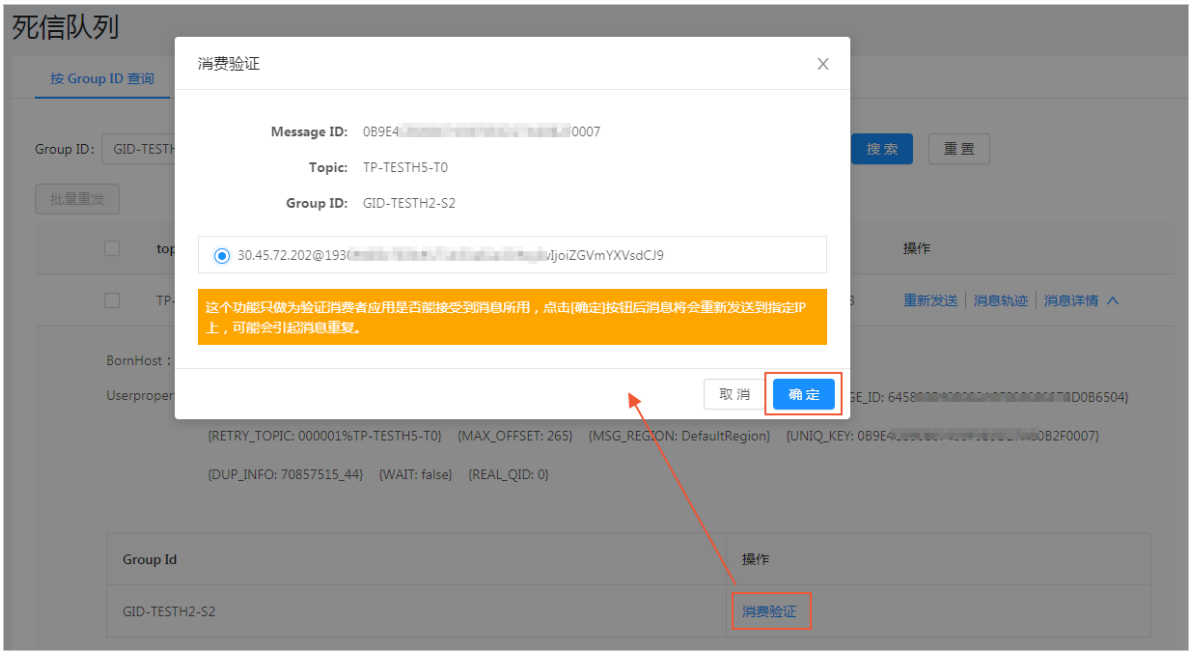
您还可以通过消费验证功能来验证指定的消费者应用是否能接受到消息。

> **说明**
>
> 消费验证功能仅仅是用于验证客户端的消费逻辑是否正常，并不会影响正常的收消息流程，因此消息的消费状态等信息在消费验证后并不会改变。

操作步骤：

1. 在展开的死信消息详情页面中，单击 Group Id 操作列的 **消费验证** 。
2. 在消费验证窗口中，选择目标消费者应用 IP。

3. 单击 **确定**，该消息将会重新发送到指定 IP 上。

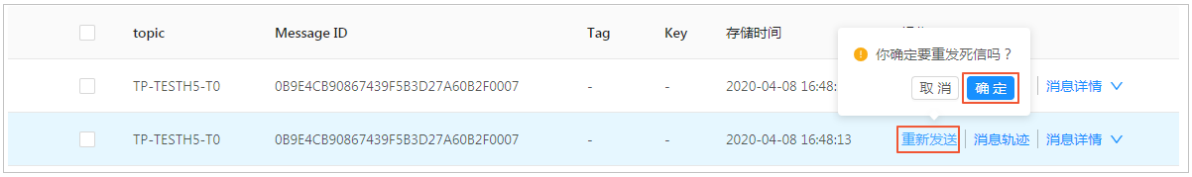


重新发送死信消息

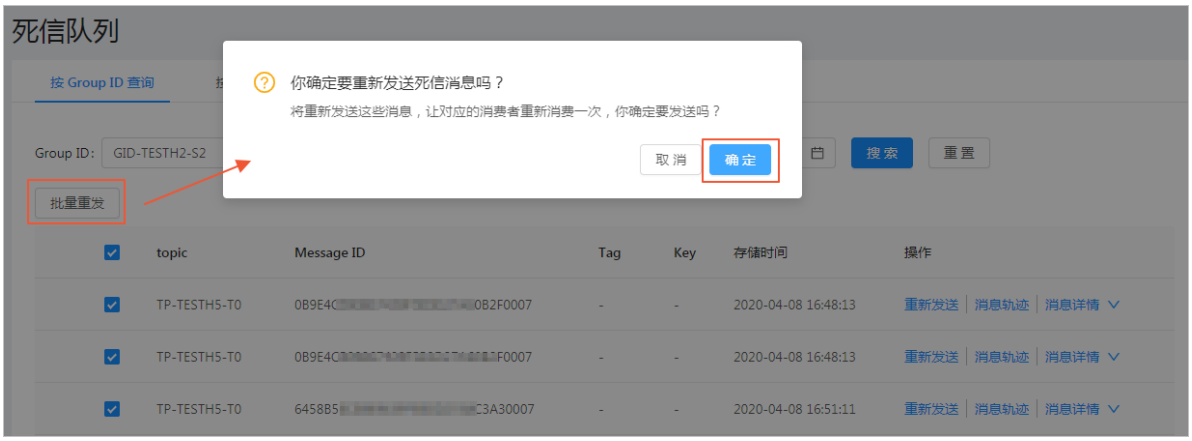
一条消息进入死信队列，意味着某些因素导致消费者无法正常消费该消息，因此，通常需要您对其进行特殊处理。排查可疑因素并解决问题后，您可以在消息队列控制台重新发送该消息，让消费者重新消费一次。

> **说明**
>
> 死信消息被重新发送后，不会在死信队列中被立即删除。

- **单条重发**在消息队列控制台按任意方式查询到死信消息后，在目标死信消息的操作列，单击 **重新发送**，重发该条死信消息。



- **批量重发**在消息队列控制台按 Group ID 查询到死信消息后，勾选目标死信消息，然后单击页面下方的 **批量重发** ，重新发送所有勾选的死信消息。



5.9. 资源大盘

资源大盘页面主要展示了消息生成和消费的数据统计及各项监控数据变化趋势等。通过该页面，您可以查看到一段时间范围内 Topic 等资源的统计数据、发送至某 Topic 的消息总量或者消息生产数据，以及某个时段内某 Topic 投递给某 Group ID 的消息总量和消息消费数据等。

操作步骤

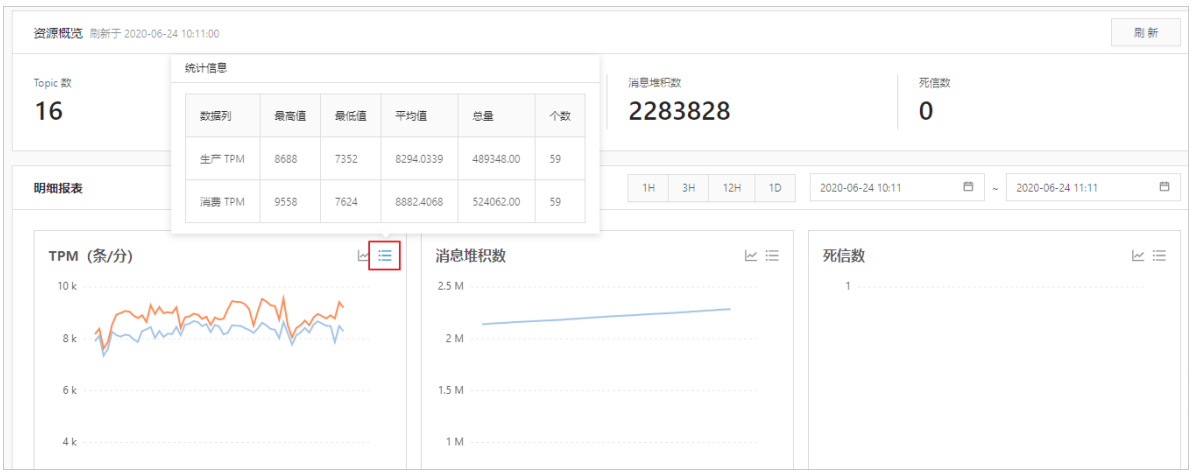
1. 登录消息队列控制台。
2. 在左侧导航栏，单击 **资源大盘**。
3. 选择或自定义时间范围，默认范围是最近 1 小时，支持的最长时间间隔为 1 天。
4. 单击 **刷新**，即可获取并查看目标时间范围内的各项资源数据：
   - 资源统计数据
   - Topic 生产报表
   - Group 消费报表

资源统计数据

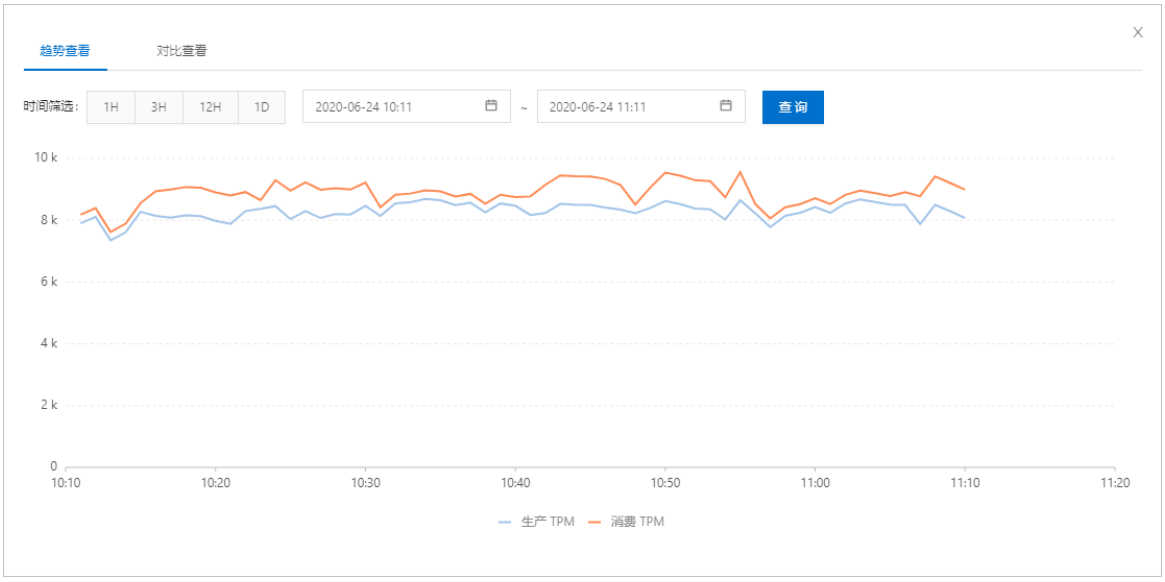
资源大盘 首先展示的是当前环境内消息相关资源的数据统计概览信息，包括 Topic 总数、Group 总数、消息堆积量与死信总数，如下图所示。

此外，您还可以查看到消息生产数据 TPM、消息堆积数以及死信数的数据变化趋势。

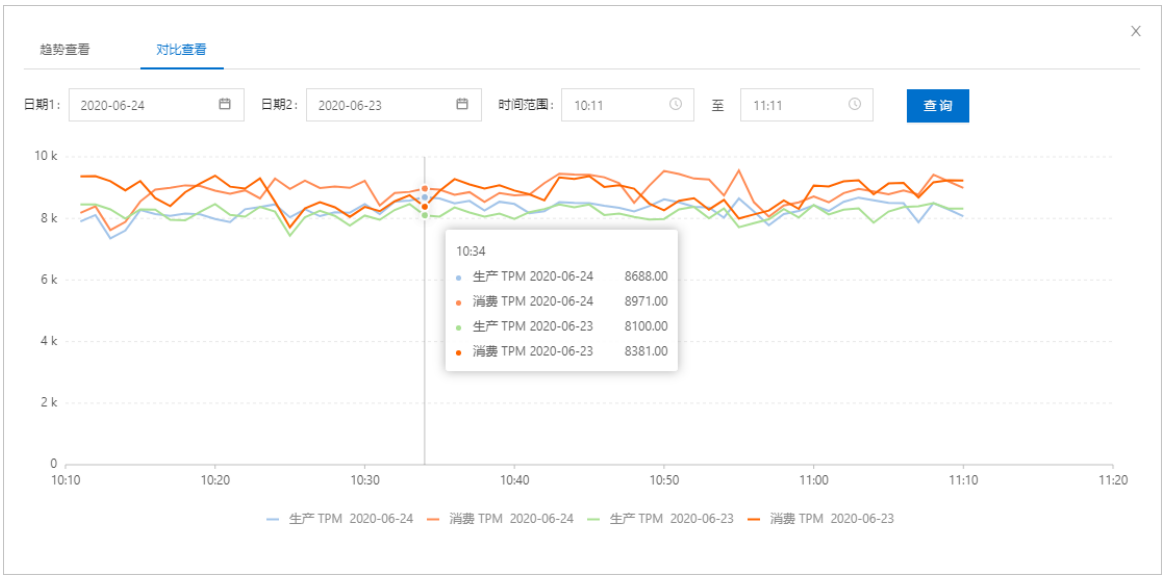
- 单击各监控图表右上角的 ☰ 图标，即可查看监控指标目标时间范围内的详细统计信息，如下图所示。



- 单击各监控图表右上角的 📈 图标，即可在新窗口中进一步查看该监控指标的详细趋势变化。
  - **趋势查看**：在 **趋势查看** 页签下，重新选择您想要查看的趋势的时间范围后，单击 **查询** 即可，如下图所示。



- **对比查看：**在 **对比查看** 页签下，选择您想要对比查看趋势变化的两个日期及目标时间范围，单击 **查询** 即可，如下图所示。



Topic 生产报表

Topic 生产报表 默认展示了当前环境所有 Topic 各项数据信息，包括 Topic 名称、消息生产数据（TPS/TPM）、每分钟数据量（字节）以及平均大小（字节），如下图所示。

- 单击目标 Topic 名称，即可进入该 Topic 的详情页查看具体的资源监控数据。
- 单击各项指标数据，即可进一步查看趋势以及对比查看两个不同时段的趋势变化。

Topic 生产报表

Topic	生产 TPS	生产 TPM	每分钟数据量（字节）	消息平均大小（字节）
TP_SOFAMQ_TEST	0.15	9	3025	336.11

共 1 条　　1　10 条/页

Group 消费报表

Group 消费报表 默认展示了当前环境所有 Group 各项数据信息，包括 Group ID、对应的 Topic、消息消费数据（TPS/TPM）、每分钟拉请求数、单次拉取数量以及当前积压量。

- 单击目标 Group ID 或 Topic 名称，即可进入 Group ID 或 Topic 的详情页查看具体的资源监控数据。
- 单击各项指标数据，即可进一步查看趋势以及对比查看两个不同时段的趋势变化。

Group 消费报表

Group ID	Topic	消费 TPS	消费 TPM	拉请求 TPM	单次拉取消息数	消息堆积数	死信数
GID-TESTH2-S1	%RETRY%GID-TESTH2-S1	2.50	150	78000	0.00	1	562448.00
GID-TESTH2-S1	TP-TESTH3-C0	0.83	50	50	1.00	1	562448.00
GID_SOFAMQ_TEST|dg	rmq_sys_TP_DEV_GROUP	0.00	0	0	0	7	0
GID-TESTH2-S2	TP-TESTH3-T0	0.00	0	0	0	1530	265.00
GID_SOFAMQ_TEST	TP_SOFAMQ_TEST	0.00	0	0	0	3	0
GID-TESTH1-S6	%RETRY%GID-TESTH1-S6	0.00	0	0	0	95	23812.00

5.10. 报警管理

您可通过 SOFAStack 消息队列的报警管理功能来监控某 Group ID 订阅的某 Topic 的消息消费状态并接收报警短信，帮助您实时掌握消息堆积情况，以便及时处理消费异常。

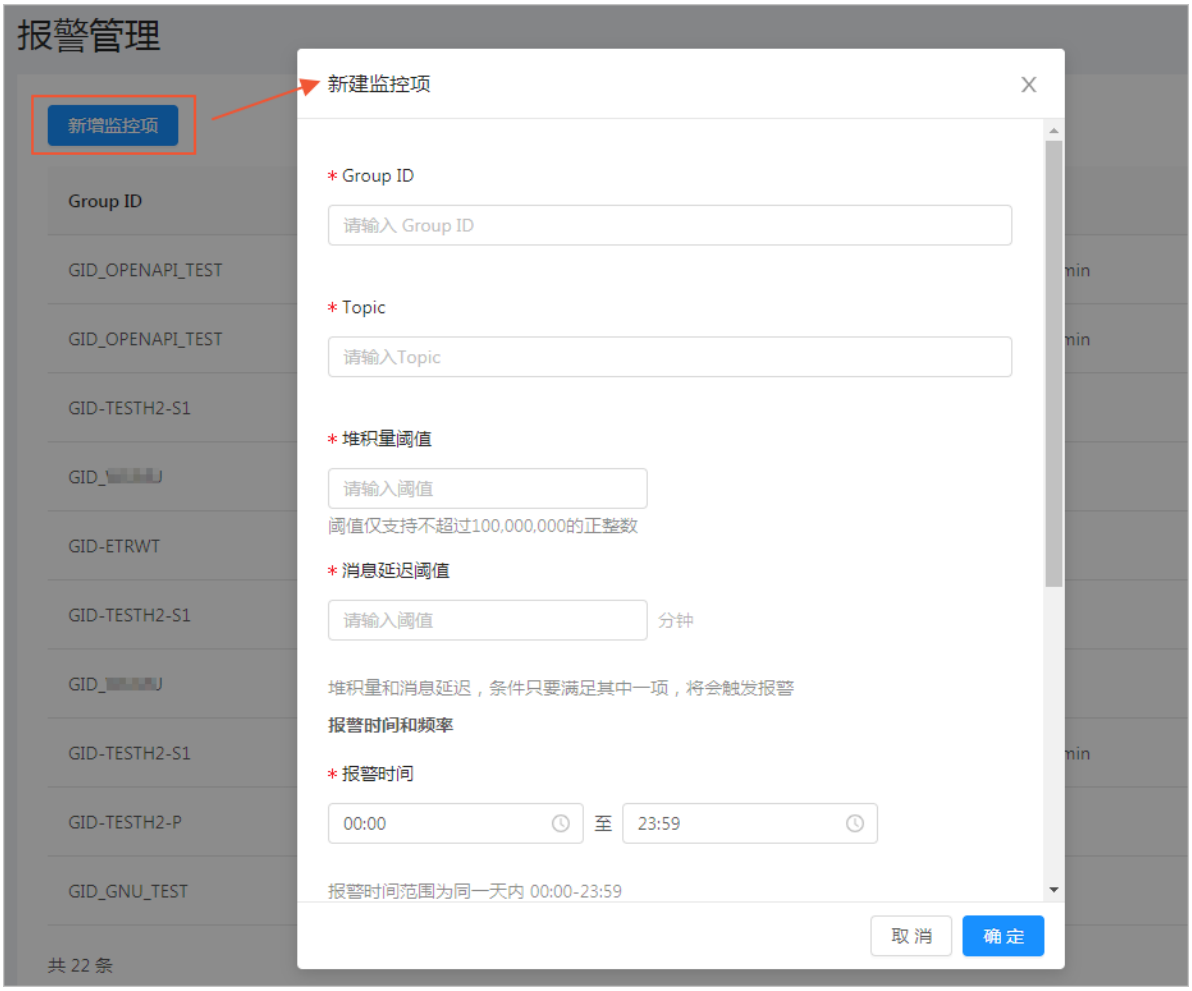
前提条件

确保您需监控的 Group ID 已订阅相应 Topic。订阅消息的详细步骤，请参见 订阅消息。

新增监控项

1. 登录消息队列控制台。
2. 在左侧导航栏，单击 **Group 管理**。
3. 单击目标 Group ID 操作列的 **更多** > **监控报警配置** 进入 **监控报警** 页面。
4. 单击 **新增监控项**，然后在 **新增监控项** 对话框配置以下信息：
   - **Group ID**：要监控的 Group ID。
   - **Topic**：要监控的该 Group ID 订阅的 Topic。
   - **堆积量阈值**：消费堆积的报警阈值。阈值的可填范围是 1 到 100,000,000 的正整数。如果您选择的 Group ID 在消费对应的 Topic 时产生了消息堆积，并且堆积超过了报警阈值，消息队列 MQ 就会给报警接收人发送短信通知。
   - **消费延迟阈值**：消费延迟的值是指该 Group ID 最近一次消费该 Topic 消息的时间和该 Topic 最近一次投递消息的时间之间的差值。可设置的消费延迟阈值的最小值是 1 分钟。
   - **报警时间**：时间精确到分钟，最大范围为 00:00 ~ 23:59。您只会在设置的接收时间范围内才会收到报警短信。
   - **报警频率**：每隔 5 分钟、15 分钟或 30 分钟。
   - **报警接收方式**：目前仅支持 **钉钉** 的方式。需输入具体的联系人组名及钉钉机器人令牌。

> **说明**
>
> 如果您已取消某个 Group ID 对某个 Topic 的订阅，请删除对应的监控项。



5. 单击 **确定**，新建的监控项会出现在监控项管理页面。

管理监控项

在 **报警管理** 页面，您可以自由管理监控项：

- 单击 **禁用** 并确认后，可禁用监控项。
- 单击 **启用**，可以重新启用已禁用的监控项。
- 单击 **编辑**，即可修改更新监控项配置信息。
- 单击 **删除** 并确认后，即可彻底删除一个无用的监控项。

> **说明**
>
> 删除或禁用任一监控项后，其消费者将无法接受堆积报警信息。请谨慎操作。

Group ID	Topic	堆积量阈值	状态	联系人	操作
GID_OPENAPI_TEST	m[illegible]nm	11111111	已启用	an[illegible]nin	告警历史 | 禁用 | 编辑 | 删除
GID_OPENAPI_TEST	n[illegible]n	100000000	已启用	an[illegible]in	告警历史 | 禁用 | 编辑 | 删除
GID-TESTH2-S1	TP-TESTH3-C0	20	已禁用	a[illegible]y	告警历史 | 启用 | 编辑 | 删除

5.11. 消息路由

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。

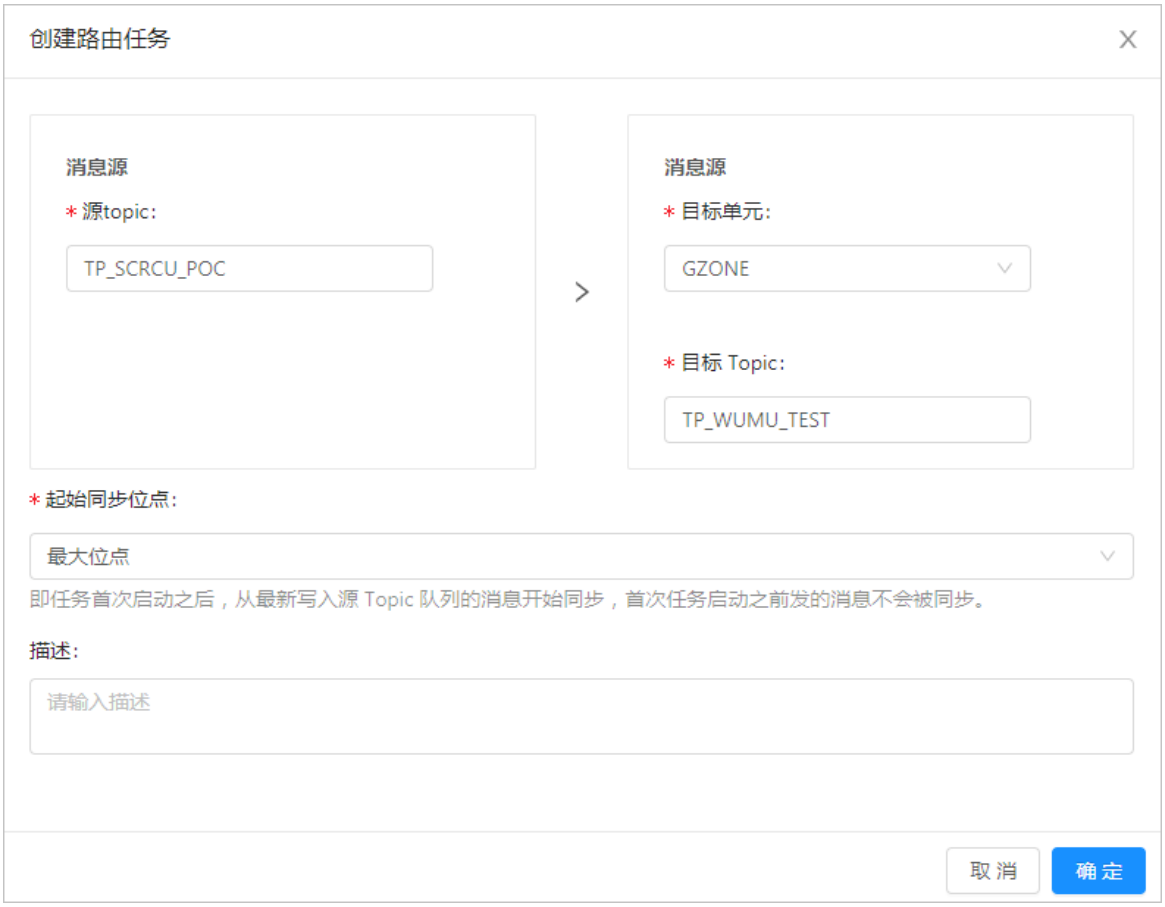
> **说明**
>
> 该功能仅适用于支持 LDC 单元化架构的环境。

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. 登录消息队列控制台。
2. 在左侧导航栏，单击 **消息路由**。
3. 单击 **创建路由任务**，然后在 **创建路由任务** 对话框配置任务信息：
   - **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。



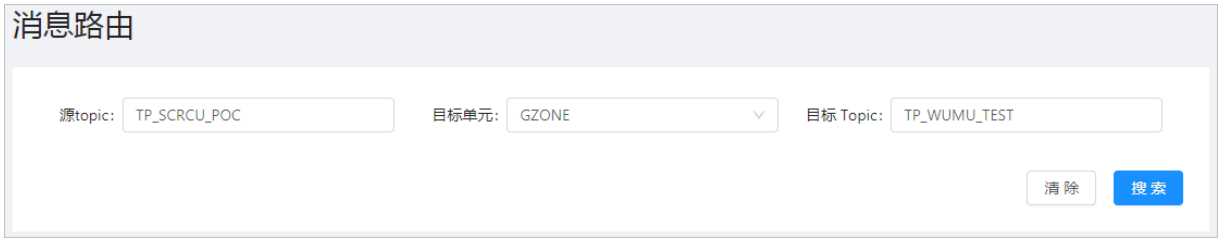
4. 单击 **确定**。

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，输入 **源 Topic**、**目标单元**、**目标 Topic** 后，单击 **搜索** ，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，单击目标任务操作列的 **启动** 或 **停止**，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，单击目标任务操作列的 **删除**，即可删除该任务。

5.12. 启用 TLS 通信加密

为了保证通信的安全性，消息的生产和消费要支持 TLS 加密。对于金融系统而言，信息安全尤为重要，而 TLS 加密是比较公认的通信安全加密方式，具备 TLS 通信加密能力且易于使用，是 SOFAMQ 必备的能力。本文介绍使用 TLS 消息加密的操作指导。

> 重要
>
> NameServer、Broker、Client 的认证证书 CA 要保持一致。

配置 NameServer

1. 准备工作。

   确认 NameServer 是否安装 OpenSSL。推荐安装，可使用 `openssl version` 命令查看。

2. 生成证书。

   以 OpenSSL 方式为例生成证书，步骤如下：

   > 说明
   >
   > 执行以下命令，过程中均要输入密码，本示例中均使用 123456。

i. 安装认证证书 CA。本文列出以下两种安装方式：

- 从专门的证书机构申请（推荐）。
- 生成自签名根证书。

  文件可命名为 `ca.crt` 。步骤如下：

  a. 生成 CA 密钥。

  ```
  openssl genrsa -des3 -out ca.key 2048
  ```

  b. 生成证书请求文件。

  ```
  openssl req -new -out ca.csr -key ca.key -keyform PEM -subj "/C=CN/ST=省/L=城市/O=单位/OU=部门/CN=域名或ip"
  ```

  c. 生成自签名根证书。

  ```
  openssl x509 -req -days 365 -in ca.csr -signkey ca.key -out ca.crt -CAcreateserial
  ```

ii. 生成应用公私密钥对，命令示例如下。

```
openssl genrsa -des3 -out server.pem 2048
```

文件可命名为 `server.pem` 。

iii. 生成证书签名请求文件，命令示例如下。

```
openssl req -new -key server.pem -out server.csr -subj "/C=CN/ST=省/L=城市/O=单位/OU=部门/CN=域名或ip"
```

文件可命名为 `server.csr` 。

iv. 使用 CA 签名生成应用证书，命令示例如下。

```
openssl x509 -req -in server.csr -out server.crt -signkey server.pem -CA ca.crt -CAkey ca.key -days 365 -CAcreateserial
```

文件可命名为 `server.crt` 。

3. 配置 `tls.properties` 配置文件。参数说明如下：

配置项	说明	示例值	建议值
`tls.test.mode.enable`	是否使用测试模式。	false	false
`tls.server.need.client.auth`	服务端是否请求客户端证书。	optional	optional
`tls.server.keyPath`	服务端私钥文件路径。	`/home/admin/store/ca/server.pem`	-

配置项	说明	示例值	建议值
`tls.server.keyPassword`	服务端私钥文件访问密码。	123456	-
`tls.server.certPath`	服务端应用证书路径。	`/home/admin/store/ca/server.crt`	-
`tls.server.authClient`	服务端是否认证客户端。	true	true
`tls.server.trustCertPath`	服务端认证证书路径。	`/home/admin/store/ca/ca.crt`	-

重要

证书建议安装在 `/home/admin/store` 目录下，该目录正常情况下不会被清理。

4. 设置环境变量。参数说明如下：

配置项	说明	默认值
`tls.server.mode`	枚举值如下： ○ Disable：不支持加密。 ○ Permissive：兼容模式。 ○ Enforcing：强制加密。	Permissive
`tls.config.file`	tls.properties 文件路径。	`/etc/rocketmq/tls.properties`

说明

如果使用云游部署，tls.properties 路径已固定为 `/home/admin/store/ca/tls.properties` 。

5. 重启 NameServer。

配置 Broker

1. 准备工作。

   确认 Broker 是否安装 OpenSSL。推荐安装，可使用 `openssl version` 命令查看。

2. 生成证书。

以 OpenSSL 方式为例生成证书，步骤如下：

> **说明**
>
> 执行以下命令，过程中均要输入密码，本示例中均使用 123456。

i. 安装认证证书 CA。本文列出以下两种安装方式：
   - 从专门的证书机构申请（推荐）。
   - 自签名生成根证书。

     文件可命名为 `ca.crt` 。生成根证书命令，可参考 NameServer 自签名根证书。

ii. 生成服务端应用公私密钥对，命令示例如下。

```
openssl genrsa -des3 -out server.pem 2048
```

文件可命名为 `server.pem` 。

iii. 生成服务端证书签名请求文件，命令示例如下。

```
openssl req -new -key server.pem -out server.csr -subj "/C=CN/ST=省/L=城市/O=单位/OU=部门/CN=域名或ip"
```

文件可命名为 `server.csr` 。

iv. 使用 CA 签名生成服务端应用证书，命令示例如下。

```
openssl x509 -req -in server.csr -out server.crt -signkey server.pem -CA ca.crt -CAkey ca.key -days 365 -CAcreateserial
```

文件可命名为 `server.crt` 。

v. 生成客户端应用公私密钥对，命令示例如下。

```
openssl genrsa -des3 -out client.pem 2048
```

文件可命名为 `client.pem` 。

vi. 生成客户端证书签名请求文件，命令示例如下。

```
openssl req -new -key client.pem -out client.csr -subj "/C=CN/ST=省/L=城市/O=单位/OU=部门/CN=域名或ip"
```

文件可命名为 `client.csr` 。

vii. 使用 CA 签名生成客户端应用证书，命令示例如下。

```
openssl x509 -req -in client.csr -out client.crt -signkey client.pem -CA ca.crt -CAkey ca.key -days 365 -CAcreateserial
```

文件可命名为 `client.crt` 。

3. 配置 `tls.properties` 配置文件。参数说明如下：

配置项	说明	示例值	建议值
tls.test.mode.enable	是否使用测试模式。	false	false
tls.server.need.client.auth	服务端是否请求客户端证书。	optional	optional
tls.server.keyPath	服务端私钥文件路径。	/home/admin/store/ca/server.pem	-
tls.server.keyPassword	服务端私钥文件访问密码。	123456	-
tls.server.certPath	服务端应用证书路径。	/home/admin/store/ca/server.crt	-
tls.server.authClient	服务端是否认证客户端。	true	true
tls.server.trustCertPath	服务端认证证书路径。	/home/admin/store/ca/ca.crt	-
tls.client.keyPath	客户端私钥文件路径。	/home/admin/store/ca/client.pem	-
tls.client.keyPassword	客户端私钥文件访问密码。	Abc123456	-
tls.client.certPath	客户端应用证书路径。	/home/admin/store/ca/client.crt	-
tls.client.authServer	客户是否认证服务端。	true	true
tls.client.trustCertPath	客户端认证证书路径。	/home/admin/store/ca/ca.crt	-

重要

建议将证书安装在 `/home/admin/store` 目录下，该目录在正常情况下不会被清理。

4. 设置环境变量。参数说明如下：

配置项	说明	默认值
tls.server.mode	枚举值如下： ○ Disable：不支持加密。 ○ Permissive：兼容模式。 ○ Enforcing：强制加密。	Permissive
tls.config.file	tls.properties 文件路径。	/etc/rocketmq/tls.properties

> **说明**
>
> 如果使用云游部署，tls.properties 路径已固定为 `/home/admin/store/ca/tls.properties` 。

5. 重启 Broker。

配置 Client

1. 准备工作。

   通过查看 Client 端 sofamq-client jar 包版本，确认当前 sofamq-client 的版本不低于 3.1.6.1。

2. 生成证书。

   以 OpenSSL 方式为例生成证书，步骤如下：

   > **说明**
   >
   > 执行以下命令，过程中均要输入密码，本示例中均使用 123456。

   i. 安装认证证书 CA。本文列出以下两种安装方式：
      - 从专门的证书机构申请（推荐）。
      - 自签名生成根证书。

        文件可命名为 `ca.crt` 。生成根证书命令，可参考 NameServer 自签名根证书。

   ii. 生成应用公私密钥对。命令示例如下。

   ```
   openssl genrsa -des3 -out client.pem 2048
   ```

   文件可命名为 `client.pem` 。

   iii. 生成证书签名请求文件。命令示例如下。

   ```
   openssl req -new -key client.pem -out client.csr -subj "/C=CN/ST=省/L=城市/O=单位/OU=部门/CN=域名或ip"
   ```

   文件可命名为 `client.csr` 。

iv. 使用 CA 签名生成应用证书。命令示例如下。

```
openssl x509 -req -in client.csr -out client.crt -signkey client.pem -CA ca.crt -CAkey ca.key -days 365 -CAcreateserial
```

文件可命名为 `client.crt` 。

3. 配置 `tls.properties` 配置文件。参数说明如下：

配置项	说明	示例值	建议值
`tls.test.mode.enable`	是否使用测试模式。	false	false
`tls.client.keyPath`	客户端私钥文件路径。	`/home/admin/store/ca/client.pem`	-
`tls.client.keyPassword`	客户端私钥文件访问密码。	123456	-
`tls.client.certPath`	客户端应用证书路径。	`/home/admin/store/ca/client.crt`	-
`tls.client.authServer`	客户是否认证服务端。	true	true
`tls.client.trustCertPath`	客户端认证证书路径。	`/home/admin/store/ca/ca.crt`	-

4. 设置环境变量。参数说明如下：

配置项	说明	默认值
`tls.config.file`	tls.properties 文件路径。	`/etc/rocketmq/tls.properties`
`tls.enable`	Client 是否开启加密。	false

5. 重启 Client。

   开启 Client TLS 前请务必确保 Broker TLS 和 NameServer TLS 已开启。

6. 创建 Producer 或 Consumer 实例时指定是否加密（可选）。

   通过环境变量 `tls.enable` 指定默认是否加密。但 API 的优先级更高。

   API 示例如下：

```
Properties properties = new Properties();

properties.setProperty(PropertyKeyConst.GROUP_ID, "GID_LFQ");

properties.setProperty(PropertyKeyConst.USE_TLS, "true"); // 开启加密

Producer producer = AccessPoint.getAccessPoint().createProducer(properties);

producer.start();
```

6.权限说明

说明

说明

专有云环境若需开启鉴权，请联系技术支持工程师。开启后，您可前往 IAM 控制台获取 AK（AccessKey ID）和 SK（AccessKey Secret）信息。获取 AK 和 SK 操作，请参见 获取 AK 和 SK。

7.常见问题

7.1. SOFAStack 常见问题

SOFAStack 是否可以部署在虚拟机、物理机上？

可以，SOFA 部署支持物理机、虚机 VM。

SOFAStack 当前是否只支持阿里云飞天 ACK？

SOFA 部署支持多云异构、目前支持华为云、开源 Openstack 等。

SOFAStack 是否可以被客户侧现有系统集成？

可以，SOFAStack 提供开放标准 API 接口，供客户侧系统调用集成。

7.2. SOFAMQ 常见问题

- 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，最多重试 16 次。如果重试 16 次后，仍然失败，则消息丢弃。您可以通过调用 `message.getReconsumeTimes()` 方法来获取消息的重试次数。

  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

- **广播消费方式**

  广播消费方式可以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 按时间范围进行消息查询，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，可以保证消息不丢失，但是无法保证消息不重复。

控制台显示的消息堆积量是否包含了 Topic 下所有 Tag 的消息？

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。

因此消费者状态页面显示的消费堆积量是未被过滤的堆积总量，包含了所有 Tag 的消息量。

7.3. 配置相关

与消息队列配置相关的常见问题如下：

- 消息队列消息在服务器保存多长时间？
- 消息队列消息 Body 的长度限制是多少？
- 消息队列客户端如何设置消费线程数？

消息队列消息在服务器保存多长时间？

存储的消息最多保存 3 天，超过 3 天未消费的消息会被删除。

消息队列消息 Body 的长度限制是多少？

消息大小根据消息类型来限制。具体限制如下所述：

- 普通消息和顺序消息：4 MB
- 事务消息和定时/延时消息：4 MB

消息队列客户端如何设置消费线程数？

在启动 Consumer 时，设置一个 ConsumeThreadNums 属性即可。具体示例如下所示：

```
public static void main(String[] args) {
    Properties properties = new Properties();
    properties.put(PropertyKeyConst.GROUP_ID, "GID_001");
    properties.put(PropertyKeyConst.AccessKey, "xxxxxxxxxxxx");
    properties.put(PropertyKeyConst.SecretKey, "xxxxxxxxxxxx");
    /**
     * 设置消费端线程数固定为 20
     */
    properties.put(PropertyKeyConst.ConsumeThreadNums, 20);
    Consumer consumer = ONSFactory.createConsumer(properties);
    consumer.subscribe("TestTopic", "*", new MessageListener() {
      publicAction consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
      }
    });
    consumer.start();
    System.out.println("Consumer Started");
}
```

7.4. 消息轨迹

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。

查询不到之前的巡检任务？

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。

7.5. 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 顺序消息。

顺序消息支持哪种消息发送方式？

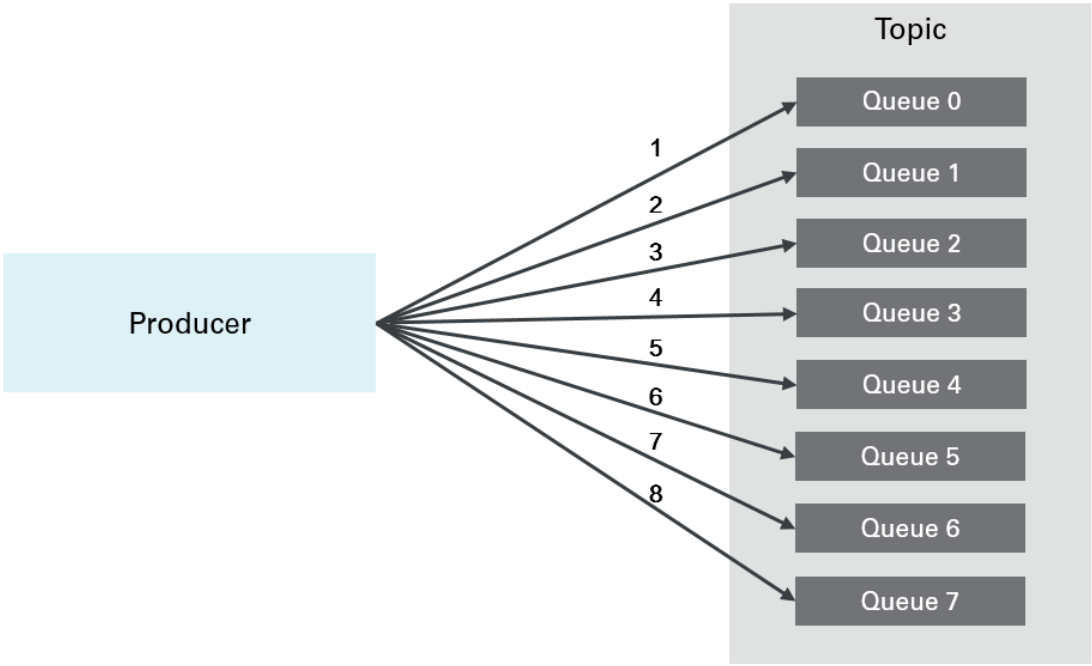
顺序消息只支持可靠同步发送方式。

7.6. 消息负载均衡策略

消息队列的消息负载策略针对发布方和订阅方有所差异。对订阅方而言，消息负载策略在一定程度上影响消息堆积。

发布方消息负载均衡策略

消息队列针对发布方采取的是轮询制，即 Producer 的消息以轮询的方式发送至 Queue，如下图所示。



图中箭头线条上的标号代表顺序，发布方会把第一条消息发送至 Queue 0，然后第二条消息发送至 Queue 1，以此类推，第八条消息发送至 Queue 7，第九条消息又发送至 Queue 0，循环往复。

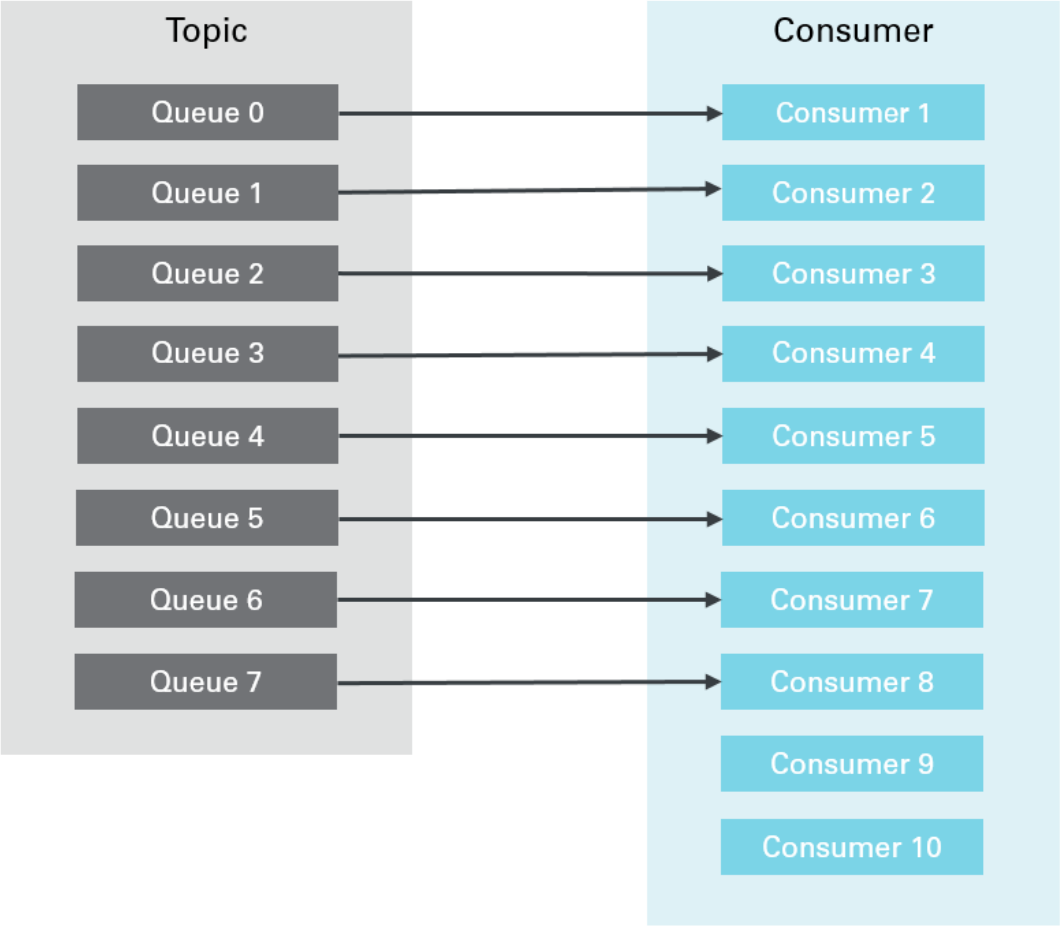
订阅方消息负载均衡策略

消息队列包含 Broker 和 Name Server 等节点，其中 Broker 节点负责将 Topic 的路由信息上报至 Name Server 节点。

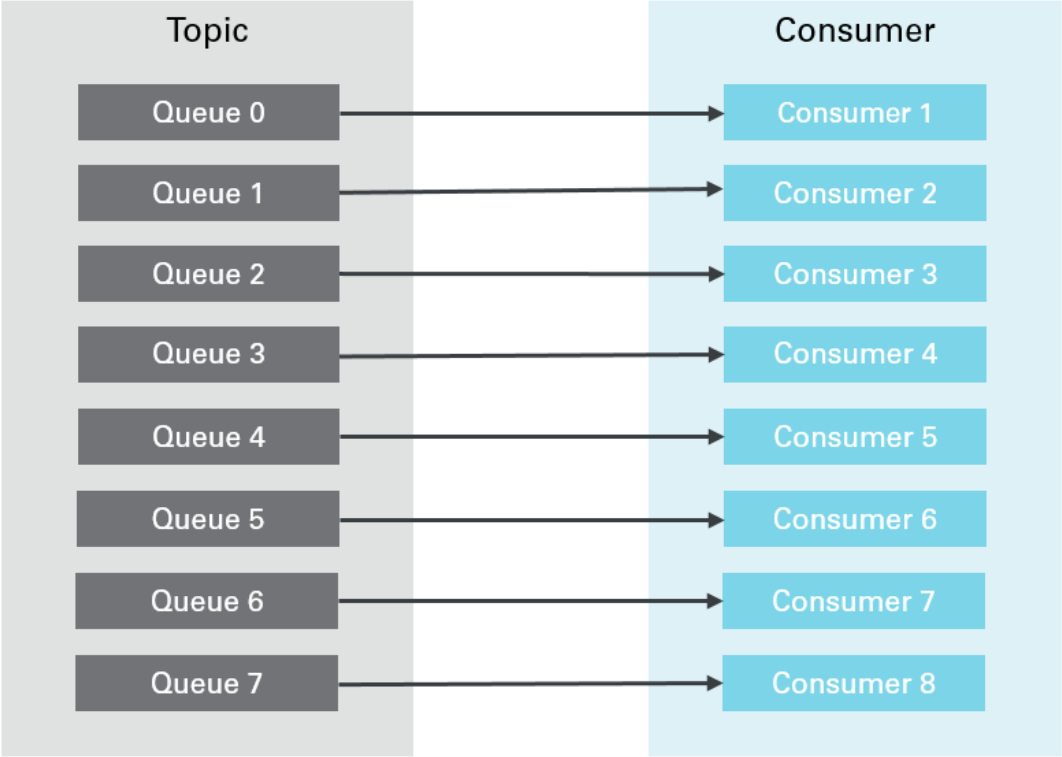
假设目前消息队列只有一个 Broker 节点，消息从 Producer 发送至消息队列的 Topic，默认会将这些 Topic 下的消息均衡负载至 8 个 Queue（逻辑概念）。消息队列 Broker 会将这些 Queue 再平均分配至属于同一个 Group ID 的订阅方集群。

因此，每台订阅方机器处理的 Queue 的数量有以下几种可能：

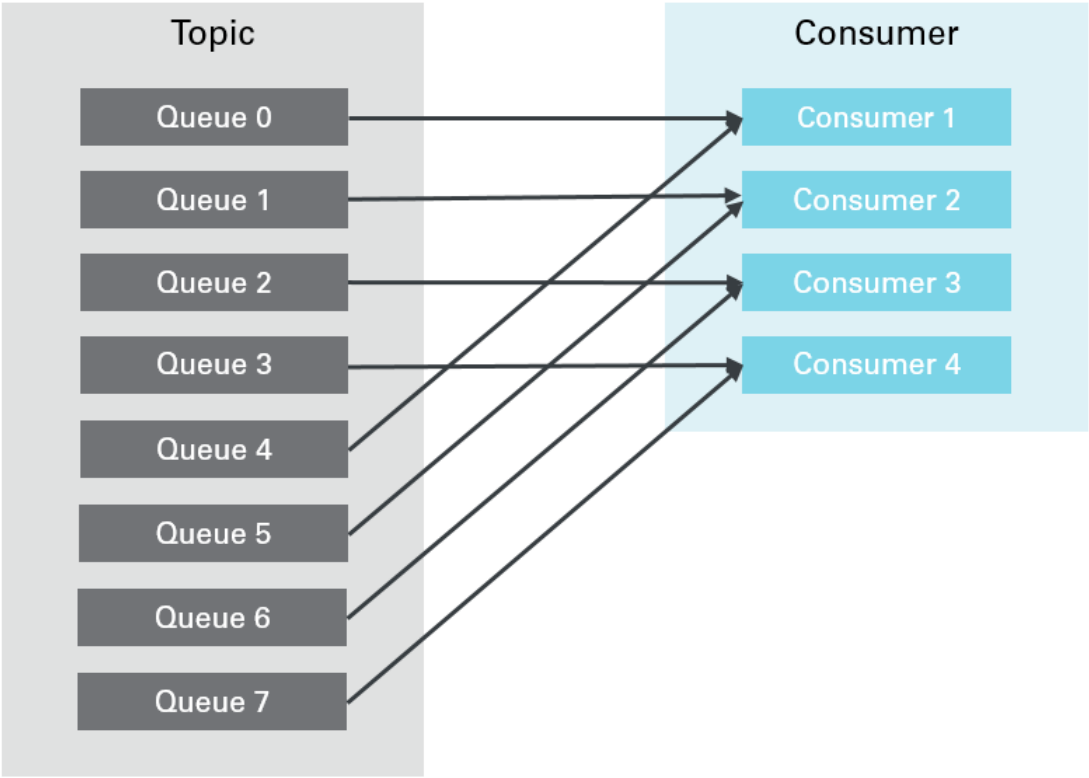
- 若订阅方机器数量大于 Queue 的数量，则超出 Queue 数量的机器将不会处理 Queue 上的消息，如下图所示。



- 若订阅方机器数量等于 Queue 的数量，则每台机器会处理 1 个 Queue 上的消息，如下图所示。



- 若订阅方机器数量小于 Queue 的数量，则每台机器会处理多个 Queue 上的消息，如下图所示。



如果其中一台机器处理变慢，可能是机器硬件、系统、远程 RPC 调用或 Java GC 等原因导致分配至此机器上的 Queue 的消息不能及时处理；此外，消息队列的消息负载是按 Queue 为粒度维护，所以，整个 Queue 上的消息都会堆积。

7.7. 消息堆积

常见的消息堆积相关问题如下：

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 `jstack -l pid > /tmp/pid.jstack` 命令查看堆栈信息。
4. 执行 `cat /tmp/pid.jstack|grep ConsumeMessageThread -A 10 --color` 命令查看 `ConsumeMessageThread` 的信息，重点关注线程的状态及堆栈。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，您可以 提交工单 获得技术支持。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。

  > **说明**
  >
  > [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。

7.8. 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
	`[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
	`[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。
	`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 上报自身路由信息，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

日志级别	打印信息	说明	解决方案
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
	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
	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。
	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。

日志级别	打印信息	说明	解决方案
	`NettyClientPublicExecutor_3 - execute the pull request exceptioncom.aliyun.openservices.shade.com.alibaba.rocketmq.client.exception.MQBrokerException: CODE: 26 DESC: subscription group [CID_XXX] does not exist,See http://rocketmq.apache.org/docs/faq/ for further details.`	订阅关系没有推送到消息队列 Broker 上。	在subscription.json 文件里直接添加 GID 对应的信息即可。
	`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
	`Can not find name server with endpoint`	一般是用户设置的 instanceId、endpoint 参数不正确。	配置正确的 instanceId、endpoint。
ERROR	`No route info of this topic,TP_XXXXX`	一般是连接了错误的 NameServer。	登录 NameServer 机器，执行 `netstat -an|grep 9876` 命令检查是否连接了正确的 NameServer，并配置正确的 NameServer。

7.9. 使用异常

本节将介绍消息队列常见的使用异常问题及相应的解决方案。

- 无法连接 Broker
- 客户端启动失败，出现“Can not find name server with endpoint”报错
- 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 Consumed, 但消费端未感知到

无法连接 Broker

可能的原因：

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

解决方案：确保阿里云主机与创建的 Topic 在同一个地域。

客户端启动失败，出现“Can not find name server with endpoint”报错

可能的原因：一般是设置的 instanceId、dataCenter（客户端会根据 dataCenter 参数过滤 ACVIP 返回的 sofamqnamesrv 地址）和 endpoint 参数不正确。

解决方案：

1. 检查 instanceId、dataCenter 和 endpoint 参数设置。
2. 您可以尝试直接请求 acvip，检查返回的数据，如果出现以下两种情况说明数据有问题：

```
curl --location --request POST 'http://{替换成用户设置的 acvip endpoint}:9003/antcloud/antvip/instances/get' \ --header 'Content-Type: application/json' \ --data '{"allowPolling" : false,"from": "sofamq-client","clientVersion": "direct_http_json","vipDomainName2ChecksumMap": {"{替换成用户设置的 instanceId}-SOFAMQ": "N"}}'
```

   - 返回的地址列表为空，说明该用户没有在 portal 里面开通 SOFAMQ，所以地址没有加到对应的 ACVIP 域名。
   - 返回的地址列表不为空，但是没有任何地址的 dataCenter 属性和用户设置的 dataCenter 一致，两者必有一个是错的。具体的 dataCenter 值请登录云游 Local，在 **环境设置** > **环境变量** > **datacenter** 中查看。

启动 Producer、Consumer 失败造成 Group ID 重复

可能的原因：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。

解决方案：

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。

解决方案：

1. 登录消息队列控制台，在左侧导航栏选择 **Topic 管理 > 创建 Topic**，然后创建 Topic。具体操作步骤请参见 创建 Topic。
2. 在左侧导航栏选择 **Group 管理 > 创建 Group ID**，然后创建 Group ID。具体操作步骤请参见 创建 Group ID。
3. 重启应用。

消息显示 Consumed，但消费端未感知到

可能的原因：

- 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。消费端以为没有收到的消息，其实是被另一个消费端接收了。

  您可以到消息队列控制台，进入 **Group 管理 > 消费者状态 > 连接状态**，查看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，然后 提交工单 获得技术支持。

7.10. 资源不存在

本文将介绍常见的资源不存在问题及相应的解决方案。

Group ID 不存在

可能的原因：未在控制台上创建该 Group ID，导致使用该 Group ID 与 Broker 创建连接的时候，服务器校验不通过。

解决方案：

1. 登录消息队列的控制台，进入 **Group 管理**，然后创建 Group ID。具体操作步骤请参见 创建 Group ID。
2. 重启应用。

主机名不存在

可能的原因：可能是无法正确获取主机名或者主机 IP 地址导致，请尝试执行 `hostname` 命令。

- 如果无法正常输出，就说明确实是此原因。
- 如果可以正常输出，可能是其他原因，请咨询售后技术支持。

解决方案：

1. 在报错机器上执行查看主机名命令：

   ```
   [root@iZ231wxgt6mZ ~]# hostname
   iZ231wxgt6mZ
   ```

   如果执行命令报错，请检查是否给 hostname 定义了别名，比如在 `.bash_profile` 或者 `.bashrc` 中 `alias xxx='hostname'`；或者命令路径不在 `$PATH` 下面。

2. Ping 主机：

   ```
   [root@iZ231wxgt6mZ ~]# ping iZ231wxgt6mZ
   ```

   如果无法正常 ping 通主机名，则需要将本机地址绑定到 `/etc/hosts` 文件中。默认 ECS 机器都会有一个本地地址和主机名的绑定关系，切勿手动的将其去除。

3. 确认系统配置检查 `/etc/sysconfig/network` 中的记录的 hostname 是否和 `/etc/hosts` 中的主机名绑定一致。

   如果不一致请修改 `/etc/sysconfig/network` 中的内容，修改后需要重启机器才能生效。

   > **重要**
   >
   > 推荐不要随意修改系统文件里的配置，可以会引发一些其他异常问题。

以上三个步骤确认完毕后，客户端启动就不会再报 `UnknownHostException` 的异常了。

7.11. 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

现象：

- 部分消息发送后客户端未收到。按 Message ID 查询消息（具体操作方法详见 消息查询）后，发现部分消息已发送至 Broker，但未投递给下游消费者。
- 查看目标 Group ID 的消费者状态（具体操作方法详见 查看消费者状态），连接信息中出现了不在预期范围内的客户端 IP，同时存在部分消息只堆积在非预期范围内的客户端上的情况。

可能的原因：

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。

解决方案：

1. 先根据连接状态定位有问题的进程，然后查看 `/{user.home}/logs/sofamq.log` 日志以及程序代码确认配置的 AK、SK、Topic 等信息是否配置正确。
2. 快速恢复。

   将有问题的客户端进程先关闭，此时之前堆积的消息会马上进行 Rebalance 投递至正常的客户端，待问题修复后再重启有问题的进程。

结果验证：

登录消息队列控制台查看目标 Group ID 的消费者状态，确认连接信息中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。同时订阅关系一致栏显示 **是**。

消息不合法

可能的原因：

消息属性、消息内容不合法，有以下 4 种情况：

- 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。

解决方案：

请确认消息没有以上不合法情况，并根据异常提示进行解决。

参数不合法

可能的原因：

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

解决方案：

1. 请确认客户端对应参数没有以上不合法情况，并根据异常提示进行解决。
2. 重启应用。

客户端状态异常

可能的原因：

- 创建 Consumer、Producer 之后未调用 `start()` 方法启动客户端。
- 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。

解决方案：

查看 `ons.log` 日志判断在客户端在启动过程中是否有异常。

订阅关系不一致

可能的原因：

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上的代码示例如下：

```
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 上的代码示例如下：

```
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（分别是 NM-MQ-FAQ-1、NM-MQ-FAQ-2）。

- JVM-1 上的代码示例如下：

```
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 上的代码示例如下：

```
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();
```

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。

7.12. 应用内存不足

本文介绍在使用消息队列时出现应用内存不足时的一些常见问题及解决方法。

现象

- 在应用部署的机器上查看内存已消耗完。
- 在 `/{user.home}/logs/sofamq.log` 能搜索到 `OutOfMemory` 关键字。
- 在消息队列控制台查看目标 Group ID 的消费者状态，**堆积量** 栏显示消息堆积较多，**连接信息** 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。

可能的原因

默认情况下，客户端的最大消耗内存为 512 MB（Group ID 订阅的所有 Topic 缓存总和），且最多会给每个 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 为 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。

解决方案

1. 确认应用依赖的 ons-client 版本。

2. 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` 参数自行定义最大消耗内存（范围在 16 MB ~ 2048 MB）。
3. 重启应用。

结果验证

- 在 `/{user.home}/logs/sofamq.log` 中 `OutOfMemory` 关键字不再出现。
- 登录消息队列控制台查看消费者状态，看到实时消费速度上升，同时堆积量下降。