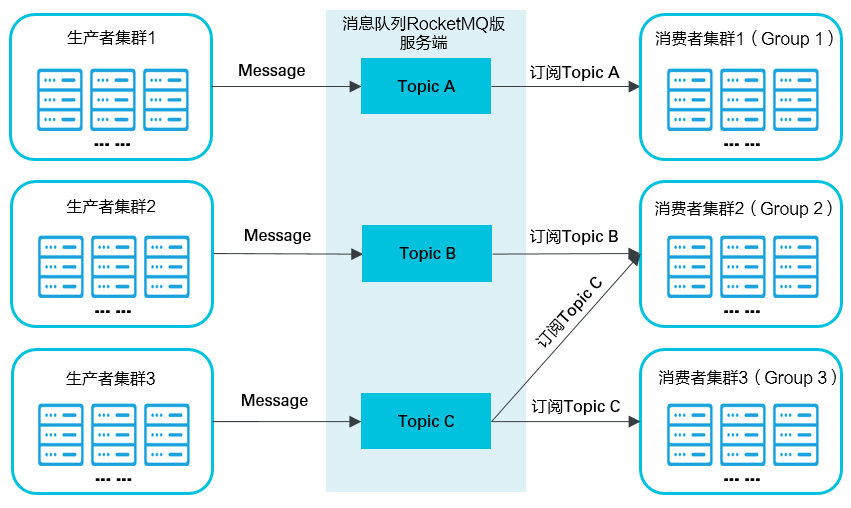
**核心概念**

* *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。

**消息收发模型**

消息队列 RocketMQ 版支持发布/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。

具体通信如下图所示。图 1. 消息收发模型[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6718273851/p68843.png)

* 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[图 1](https://help.aliyun.com/document_detail/29532.html?spm=a2c4g.11186623.6.547.2c9165fetCPA6c#fig-rys-aqb-674) 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，具体设置方法可参见[产品更新日志](https://help.aliyun.com/document_detail/102895.html#concept-2047061)中的资源申请流程优化部分。

**应用场景**

* 削峰填谷

诸如秒杀、抢红包、企业开门红等大型活动时皆会带来较高的流量脉冲，或因没做相应的保护而导致系统超负荷甚至崩溃，或因限制太过导致请求大量失败而影响用户体验，消息队列 RocketMQ 版可提供削峰填谷的服务来解决该问题。

* 异步解耦

交易系统作为淘宝/天猫主站最核心的系统，每笔交易订单数据的产生会引起几百个下游业务系统的关注，包括物流、购物车、积分、流计算分析等等，整体业务系统庞大而且复杂，消息队列 RocketMQ 版可实现异步通信和应用解耦，确保主站业务的连续性。

* 顺序收发

细数日常中需要保证顺序的应用场景非常多，例如证券交易过程时间优先原则，交易系统中的订单创建、支付、退款等流程，航班中的旅客登机消息处理等等。与先进先出（First In First Out，缩写 FIFO）原理类似，消息队列 RocketMQ 版提供的顺序消息即保证消息 FIFO。

* 分布式事务一致性

交易系统、支付红包等场景需要确保数据的最终一致性，大量引入消息队列 RocketMQ 版的分布式事务，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。

* 大数据分析

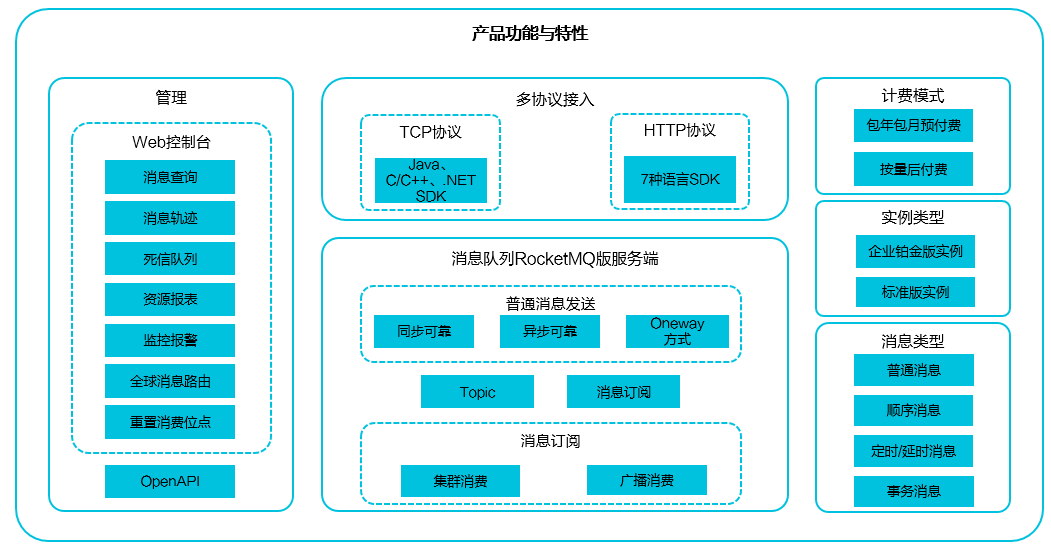
数据在“流动”中产生价值，传统数据分析大多是基于批量计算模型，而无法做到实时的数据分析，利用阿里云消息队列 RocketMQ 版与流式计算引擎相结合，可以很方便的实现将业务数据进行实时分析。

* 分布式缓存同步

天猫双 11 大促，各个分会场琳琅满目的商品需要实时感知价格变化，大量并发访问数据库导致会场页面响应时间长，集中式缓存因为带宽瓶颈限制商品变更的访问流量，通过消息队列 RocketMQ 版构建分布式缓存，实时通知商品数据的变化。

**概览**

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

消息队列 RocketMQ 版提供 TCP 和 HTTP 协议的多语言接入方式，方便不同编程语言开发的应用快速接入消息队列 RocketMQ 版消息云服务。您可以将应用部署在阿里云 ECS、企业自建云，或者嵌入到移动端、物联网设备中与消息队列 RocketMQ 版建立连接进行消息收发；同时，本地开发者也可以通过公网接入消息队列 RocketMQ 版服务进行消息收发。[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6771879851/p68860.png)

**多协议接入**

* [HTTP 协议](https://help.aliyun.com/document_detail/102996.html#concept-2047113)：采用 RESTful 风格，方便易用，快速接入，跨网络能力强。支持 Java、C++、.NET、Go、Python、Node.js 和 PHP 七种语言客户端。
* [TCP 协议](https://help.aliyun.com/document_detail/114448.html#concept-2335081)：区别于 HTTP 简单的接入方式，提供更为专业、可靠、稳定的 TCP 协议的 SDK 接入服务。支持的语言包括 Java、C/C++ 以及 .NET。

**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、资源报表以及监控报警管理。
* OpenAPI：提供开放的 API 便于将消息队列 RocketMQ 版管理工具集成到自己的控制台。消息队列 RocketMQ 版的 API 详情请参见[OpenAPI 参考](https://help.aliyun.com/document_detail/44418.html#concept-1351495)。

**消息类型**

* [普通消息](https://help.aliyun.com/document_detail/96359.html#concept-2047064)：消息队列 RocketMQ 版中无特性的消息，区别于有特性的定时和延时消息、顺序消息和事务消息。
* [事务消息](https://help.aliyun.com/document_detail/43348.html#concept-2047067)：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](https://help.aliyun.com/document_detail/43349.html#concept-2047065)：允许消息生产者对指定消息进行定时（延时）投递，最长支持 40 天。
* [顺序消息](https://help.aliyun.com/document_detail/49319.html#concept-2047066)：允许消息消费者按照消息发送的顺序对消息进行消费。

**特性功能**

* [消息查询](https://help.aliyun.com/document_detail/29540.html#concept-2047150)：消息队列 RocketMQ 版提供了三种消息查询的方式，分别是按 Message ID、Message Key 以及 Topic 查询。
* [查询消息轨迹](https://help.aliyun.com/document_detail/43357.html#concept-2335151)：通过消息轨迹，能清晰定位消息从生产者发出，经由消息队列 RocketMQ 版服务端，投递给消息消费者的完整链路，方便定位排查问题。
* [集群消费和广播消费](https://help.aliyun.com/document_detail/43163.html#concept-2047071)：当使用集群消费模式时，消息队列 RocketMQ 版认为任意一条消息只需要被消费者集群内的任意一个消费者处理即可；当使用广播消费模式时，消息队列 RocketMQ 版会将每条消息推送给消费者集群内所有注册过的消费者，保证消息至少被每台机器消费一次。
* [重置消费位点](https://help.aliyun.com/document_detail/63390.html#task-2047153)：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。
* [死信队列](https://help.aliyun.com/document_detail/87277.html#concept-2047154)：将无法正常消费的消息储存到特殊的死信队列供后续处理。
* [全球消息路由](https://help.aliyun.com/document_detail/90483.html#concept-2047155)：用于全球不同地域之间的消息同步，保证地域之间的数据一致性。
* [资源报表](https://help.aliyun.com/document_detail/43733.html#concept-2047157)：消息生产和消费数据的统计功能。通过该功能，您可查询在一段时间范围内发送至某 Topic 的消息总量或者 TPS（消息生产数据），也可查询在一个时间段内某 Topic 投递给某 Group ID 的消息总量或 TPS（消息消费数据）。
* [监控报警](https://help.aliyun.com/document_detail/43732.html#concept-2047158)：您可使用消息队列 RocketMQ 版提供的监控报警功能，监控某 Group ID 订阅的某 Topic 的消息消费状态并接收报警短信，帮助您实时掌握消息消费状态，以便及时处理消费异常。

# 定时和延时消息

## 概念介绍

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

定时消息与延时消息在代码配置上存在一些差异，但是最终达到的效果相同：消息在发送到消息队列 RocketMQ 版服务端后并不会立马投递，而是根据消息中的属性延迟固定时间后才投递给消费者。

## 适用场景

定时消息和延时消息适用于以下一些场景：

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

## 使用方式

定时消息和延时消息的使用在代码编写上存在略微的区别：

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

## 注意事项

* 定时消息的精度会有 1s ~ 2s 的延迟误差。
* 定时和延时消息的 msg.setStartDeliverTime 参数需要设置成当前时间戳之后的某个时刻（单位毫秒）。如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
* 定时和延时消息的 msg.setStartDeliverTime 参数可设置 40 天内的任何时刻（单位毫秒），超过 40 天消息发送将失败。
* StartDeliverTime 是服务端开始向消费端投递的时间。 如果消费者当前有消息堆积，那么定时和延时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
* 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差。
* 设置定时和延时消息的投递时间后，依然受 3 天的消息保存时长限制。

例如，设置定时消息 5 天后才能被消费，如果第 5 天后一直没被消费，那么这条消息将在第 8 天被删除。

# 顺序消息

顺序消息（FIFO 消息）是消息队列 RocketMQ 版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。

顺序消息分为[全局顺序消息](https://help.aliyun.com/document_detail/49319.html?spm=a2c4g.11186623.6.554.756d3631jcGwjY#section-9q7-b7b-nua)和[分区顺序消息](https://help.aliyun.com/document_detail/49319.html?spm=a2c4g.11186623.6.554.756d3631jcGwjY#section-pdd-h4p-aax)。

## 全局顺序消息

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

* 适用场景

适用于性能要求不高，所有的消息严格按照 FIFO 原则来发布和消费的场景。

* 示例

在证券处理中，以人民币兑换美元为 Topic，在价格相同的情况下，先出价者优先处理，则可以按照 FIFO 的方式发布和消费全局顺序消息。

## 分区顺序消息

对于指定的一个 Topic，所有消息根据 Sharding Key 进行区块分区。同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。Sharding Key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。

* 适用场景

适用于性能要求高，以 Sharding Key 作为分区字段，在同一个区块中严格地按照 FIFO 原则进行消息发布和消费的场景。

* 示例
  + 用户注册需要发送发验证码，以用户 ID 作为 Sharding Key，那么同一个用户发送的消息都会按照发布的先后顺序来消费。
  + 电商的订单创建，以订单 ID 作为 Sharding Key，那么同一个订单相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息都会按照发布的先后顺序来消费。

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

## 全局顺序与分区顺序对比

在控制台创建顺序消息使用的不同类型 Topic 对比如下。

| *表 1. 消息类型对比* | | | |
| --- | --- | --- | --- |
| **Topic 的消息类型** | **是否支持事务消息** | **是否支持定时/延时消息** | **性能** |
| 无序消息（普通、事务、定时/延时消息） | 是 | 是 | 最高 |
| 分区顺序消息 | 否 | 否 | 高 |
| 全局顺序消息 | 否 | 否 | 一般 |

| *表 2. 发送方式对比* | | | |
| --- | --- | --- | --- |
| **消息类型** | **是否支持可靠同步发送** | **是否支持可靠异步发送** | **是否支持 Oneway 发送** |
| 无序消息（普通、事务、定时/延时消息） | 是 | 是 | 是 |
| 分区顺序消息 | 是 | 否 | 否 |
| 全局顺序消息 | 是 | 否 | 否 |

## 注意事项

使用顺序消息时，请注意以下几点：

* 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。
* 对于全局顺序消息，建议消息不要有阻塞。同时运行多个实例，是为了防止工作实例意外退出而导致业务中断。当工作实例退出时，其他实例可以立即接手工作，不会导致业务中断，实际工作的只会有一个实例。

## 顺序消息常见问题

* 同一条消息是否可以既是顺序消息，又是定时消息和事务消息？

不行。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？

支持消息队列 RocketMQ 版所有公共云地域和金融云地域。

* 为什么全局顺序消息性能一般？

全局顺序消息是严格按照 FIFO 的消息阻塞原则，即上一条消息没有被成功消费，那么下一条消息会一直被存储到 Topic 队列中。如果想提高全局顺序消息的 TPS，可以升级实例配置，同时消息客户端应用尽量减少处理本地业务逻辑的耗时。

* 顺序消息支持哪种消息发送方式？

顺序消息只支持可靠同步发送方式，不支持异步发送方式，否则将无法严格保证顺序。

* 顺序消息是否支持集群消费和广播消费？

顺序消息暂时仅支持集群消费模式，不支持广播消费模式。

# 事务消息

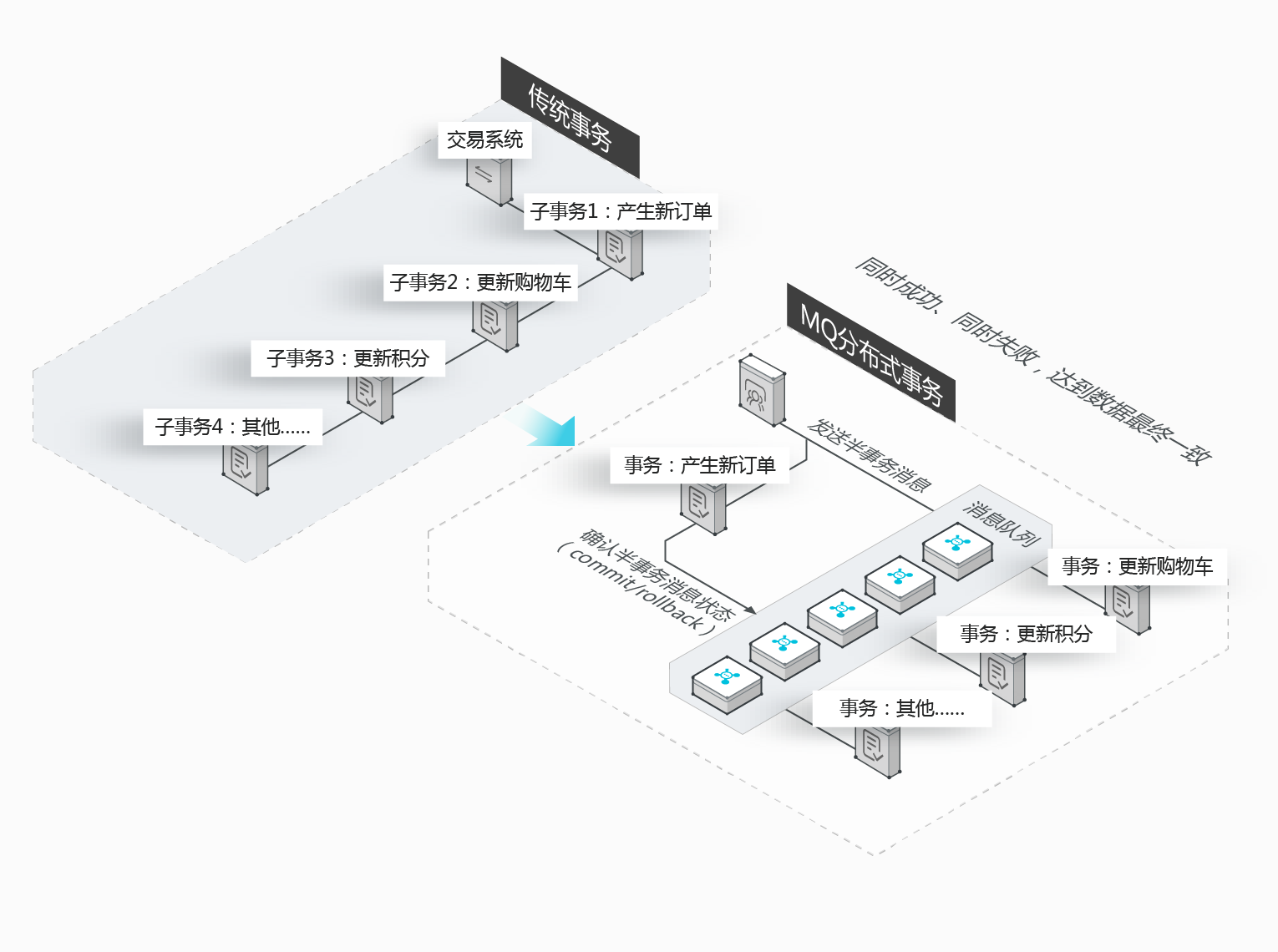
消息队列 RocketMQ 版提供的分布式事务消息适用于所有对数据最终一致性有强需求的场景。本文介绍消息队列 RocketMQ 版事务消息的概念、优势、典型场景、交互流程以及使用过程中的注意事项。

## 概念介绍

* 事务消息：消息队列 RocketMQ 版提供类似 X/Open XA 的分布式事务功能，通过消息队列 RocketMQ 版事务消息能达到分布式事务的最终一致。
* 半事务消息：暂不能投递的消息，发送方已经成功地将消息发送到了消息队列 RocketMQ 版服务端，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 RocketMQ 版服务端通过扫描发现某条消息长期处于“半事务消息”时，需要主动向消息生产者询问该消息的最终状态（Commit 或是 Rollback），该询问过程即消息回查。

## 分布式事务消息的优势

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

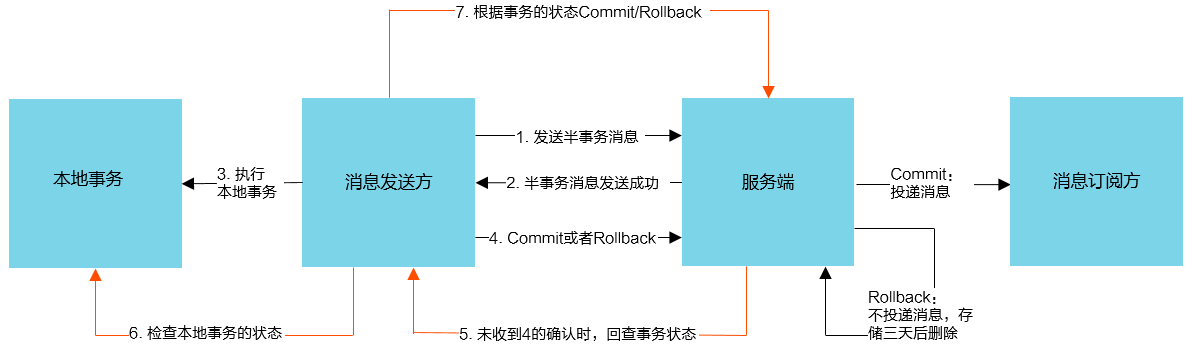
[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6087385851/p96619.png)

## 典型场景

在淘宝购物车下单时，涉及到购物车系统和交易系统，这两个系统之间的数据最终一致性可以通过分布式事务消息的异步处理实现。在这种场景下，交易系统是最为核心的系统，需要最大限度地保证下单成功。而购物车系统只需要订阅消息队列 RocketMQ 版的交易订单消息，做相应的业务处理，即可保证最终的数据一致性。

## 交互流程

事务消息交互流程如下图所示。

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6087385851/p69094.png)

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

事务消息回查步骤如下：

1. 在断网或者是应用重启的特殊情况下，上述步骤 4 提交的二次确认最终未到达服务端，经过固定时间后服务端将对该消息发起消息回查。
2. 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
3. 发送方根据检查得到的本地事务的最终状态再次提交二次确认，服务端仍按照步骤 4 对半事务消息进行操作。

## 注意事项

1. 事务消息的 Group ID 不能与其他类型消息的 Group ID 共用。与其他类型的消息不同，事务消息有回查机制，回查时消息队列 RocketMQ 版服务端会根据 Group ID 去查询客户端。
2. 通过 ONSFactory.createTransactionProducer 创建事务消息的 Producer 时必须指定 LocalTransactionChecker 的实现类，处理异常情况下事务消息的回查。
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 RocketMQ 版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：
5. Message message = **new** Message();
6. // 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
7. 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

# 消息重试

# 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 RocketMQ 版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。

**注意** 以下内容都只针对无序消息生效。

## 重试次数

消息队列 RocketMQ 版默认允许每条消息最多重试 16 次，每次重试的间隔时间如下：

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

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

**注意** 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 exception");

}

}

* 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

**说明** 自定义消息队列 RocketMQ 版的客户端日志配置，请升级 TCP Java SDK 版本到 1.2.2 及以上。

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

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

配置方式如下：

Properties properties = **new** Properties();

//配置对应 Group ID 的最大消息重试次数为 20 次

properties.put(PropertyKeyConst.MaxReconsumeTimes,"20");

Consumer consumer =ONSFactory.createConsumer(properties);

**注意**

* + 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;

}

}

# 消息过滤

本文描述消息队列 RocketMQ 版的消费者如何根据 Tag 在消息队列 RocketMQ 版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 RocketMQ 版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。

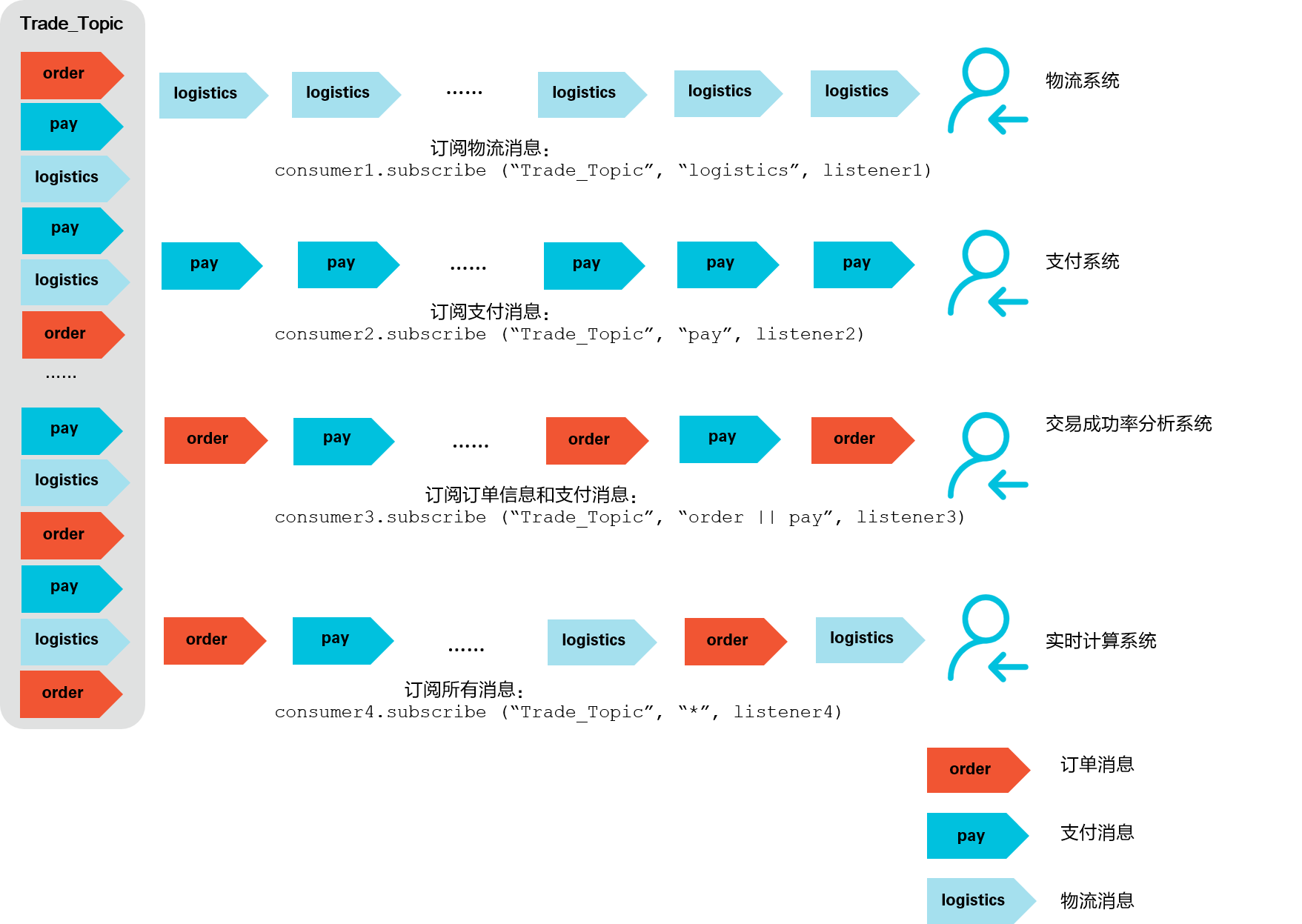
## 场景示例

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

* 订单创建消息
* 支付消息
* 物流消息

这些消息会发送到 **Trade\_Topic** Topic 中，被各个不同的系统所订阅，以以下系统为例：

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

过滤示意图如下所示。[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/1629885751/p69168.png)

## 示例代码

* 发送消息

发送消息时，每条消息必须指明 Tag：

Message msg = **new** Message("MQ\_TOPIC","TagA","HelloMQ".getBytes());

* 订阅所有 Tag

消费者如需订阅某 Topic 下所有类型的消息，Tag 用符号 \* 表示：

consumer.subscribe("MQ\_TOPIC", "\*", **new** MessageListener() {

**public** Action **consume**(Message message, ConsumeContext context) {

System.out.println(message.getMsgID());

**return** Action.CommitMessage;

}

});

* 订阅单个 Tag

消费者如需订阅某 Topic 下某一种类型的消息，请明确标明 Tag：

consumer.subscribe("MQ\_TOPIC", "TagA", **new** MessageListener() {

**public** Action **consume**(Message message, ConsumeContext context) {

System.out.println(message.getMsgID());

**return** Action.CommitMessage;

}

});

* 订阅多个 Tag

消费者如需订阅某 Topic 下多种类型的消息，请在多个 Tag 之间用 || 分隔：

consumer.subscribe("MQ\_TOPIC", "TagA||TagB",

**new** MessageListener() {

**public** Action **consume**(Message message, ConsumeContext context) {

System.out.println(message.getMsgID());

**return** Action.CommitMessage;

}

});

* 错误示例

同一个消费者多次订阅某个 Topic 下的 Tag，以最后一次订阅的 Tag 为准：

//如下错误代码中，Consumer 只能订阅到 MQ\_TOPIC 下 TagB 的消息，而不能订阅 TagA 的消息。

consumer.subscribe("MQ\_TOPIC", "TagA", **new** MessageListener() {

**public** Action **consume**(Message message, ConsumeContext context) {

System.out.println(message.getMsgID());

**return** Action.CommitMessage;

}

});

consumer.subscribe("MQ\_TOPIC", "TagB", **new** MessageListener() {

**public** Action **consume**(Message message, ConsumeContext context) {

System.out.println(message.getMsgID());

**return** Action.CommitMessage;

}

});

## 更多信息

* 同一个 Group ID 下的消费者实例与 Topic 的订阅关系需保持一致，详情请参见[订阅关系一致](https://help.aliyun.com/document_detail/43523.html#concept-2047148)。
* 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 最佳实践](https://help.aliyun.com/document_detail/95837.html#concept-2047146)。

# Exactly-Once 投递语义

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

## 什么是 Exactly-Once 投递语义

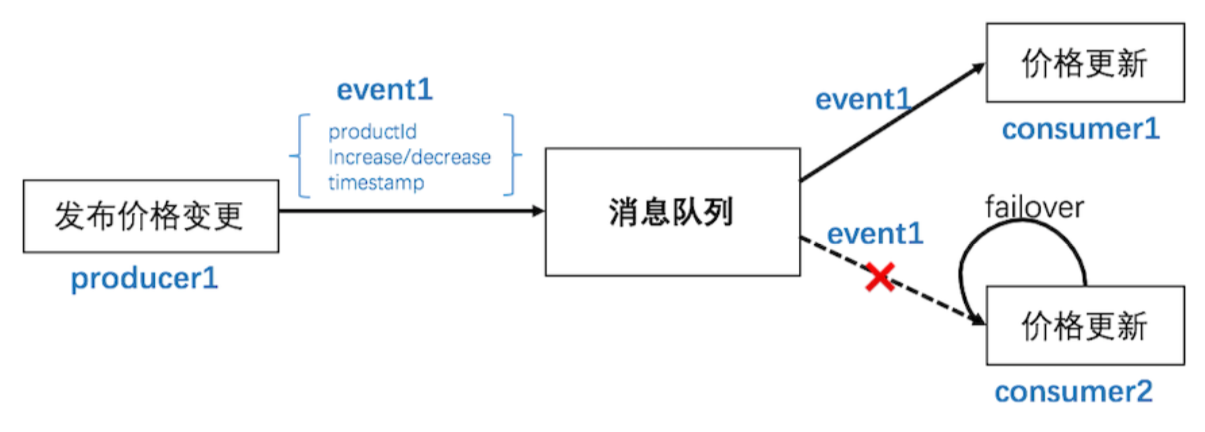
Exactly-Once 是指发送到消息系统的消息只能被消费端处理且仅处理一次，即使生产端重试消息发送导致某消息重复投递，该消息在消费端也只被消费一次。

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

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

如果您要实现一条消息的消费结果只能在业务系统中生效一次，您需要解决的只是如何保证同一条消息的消费幂等问题。消息队列 RocketMQ 版的 Exactly-Once 语义就是解决业务中最常见的一条消息的消费结果（消息在消费端计算处理的结果）在数据库系统中有且仅生效一次的问题。

## 典型使用场景

在电商系统中，上游实时计算模块发布商品价格变更的信息，异步通知到下游商品管理模块进行价格变更。此时，需要保证每一条信息的消费幂等，即重复的价格变更信息只会生效一次，这样便不会发生价格多次重复修改的情况，确保实现了消息消费的幂等。[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/1629885751/p69179.png)

## 更多信息

如何使用 Exactly-Once 投递语义的具体步骤，请参见[使用 Exactly-Once 投递语义收发消息](https://help.aliyun.com/document_detail/102777.html#multiTask17923)。

# 集群消费和广播消费

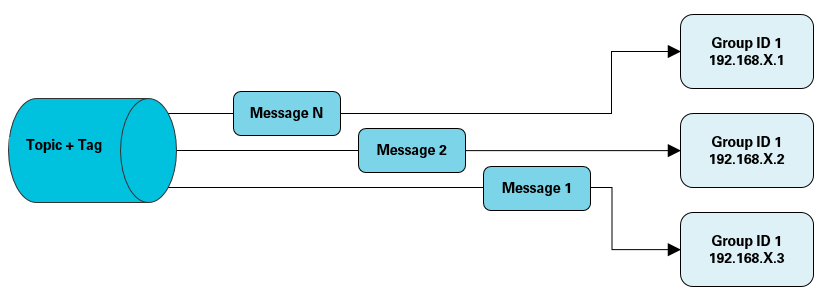
本文介绍消息队列 RocketMQ 版的集群消费和广播消费的基本概念、适用场景以及注意事项。

消息队列 RocketMQ 版是基于发布/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 RocketMQ 版约定以下概念：

* 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](https://help.aliyun.com/document_detail/43523.html#concept-2047148)。
* 集群消费：当使用集群消费模式时，消息队列 RocketMQ 版认为任意一条消息只需要被集群内的任意一个消费者处理即可。
* 广播消费：当使用广播消费模式时，消息队列 RocketMQ 版会将每条消息推送给集群内所有注册过的消费者，保证消息至少被每个消费者消费一次。

## 集群消费模式

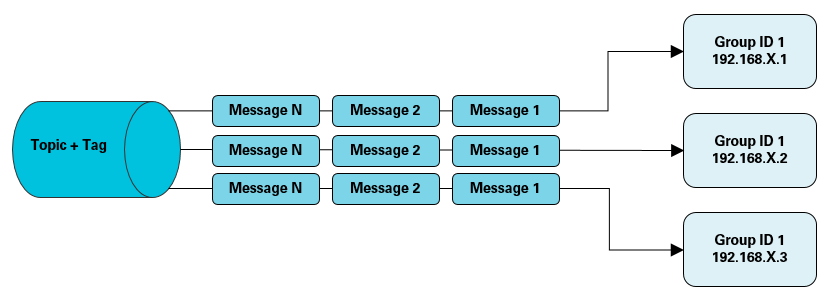
* 适用场景

适用于消费端集群化部署，每条消息只需要被处理一次的场景。此外，由于消费进度在服务端维护，可靠性更高。具体消费示例如下图所示。[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/7469771951/p69189.png)

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

## 广播消费模式

* 适用场景

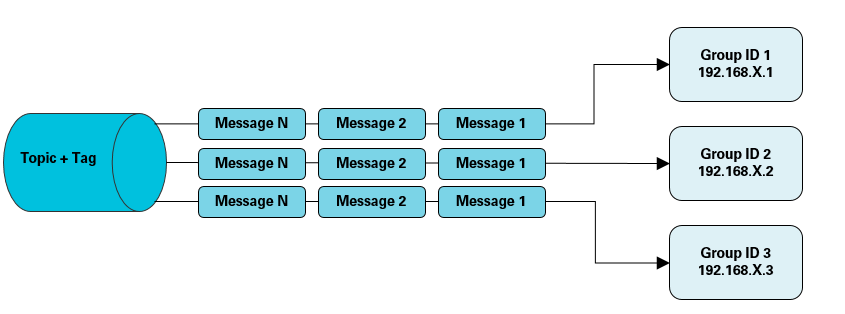
适用于消费端集群化部署，每条消息需要被集群下的每个消费者处理的场景。具体消费示例如下图所示。[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/7469771951/p69191.png)

* 注意事项
  + 广播消费模式下不支持顺序消息。
  + 广播消费模式下不支持重置消费位点。
  + 每条消息都需要被相同订阅逻辑的多台机器处理。
  + 消费进度在客户端维护，出现重复消费的概率稍大于集群模式。
  + 广播模式下，消息队列 RocketMQ 版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 RocketMQ 版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。

## 多个 Group ID 通过集群订阅方式实现广播消费模式

* 适用场景

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

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/7469771951/p69192.png)

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

* 注意事项
  + 消费进度在服务端维护，可靠性高于广播模式。
  + 对于一个 Group ID 来说，可以部署一个消费者实例，也可以部署多个消费者实例。当部署多个消费者实例时，实例之间又组成了集群模式（共同分担消费消息）。假设 Group ID 1 部署了三个消费者实例 C1、C2、C3，那么这三个实例将共同分担服务器发送给 Group ID 1 的消息。同时，实例之间订阅关系必须保持一致。详见[订阅关系一致](https://help.aliyun.com/document_detail/43523.html#concept-2047148)。

# 适用场景

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

例如，针对一家互联网电商企业，其业务涉及广泛，如注册、订单、库存、物流等；同时，也会涉及许多业务峰值时刻，如秒杀活动、周年庆、定期特惠等。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 RocketMQ 版作为分布式系统中的重要组件，可用于应对这些挑战，例如解决应用的异步解耦。

下文先以用户注册为场景说明消息队列 RocketMQ 版如何实现以下功能：

* 异步解耦
* 分布式事务的数据一致性
* 消息的顺序收发

最后，再以电商的秒杀场景和价格同步场景分别说明消息队列 RocketMQ 版所实现的削峰填谷和大规模机器的缓存同步。

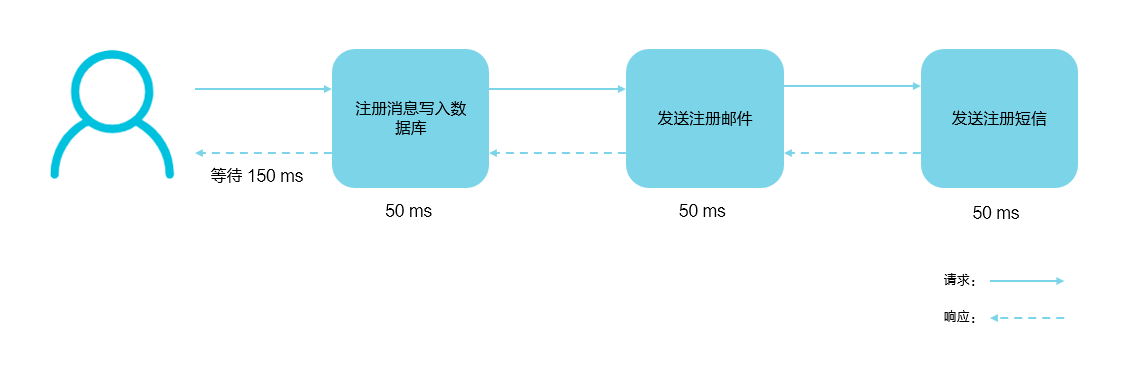
## 异步解耦

**传统处理**

最常见的一个场景是用户注册后，需要发送注册邮件和短信通知，以告知用户注册成功。传统的做法有以下两种：

* 串行方式

串行方式下的注册流程如下图所示。

[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/5219985751/p68911.png)数据流动如下所述：

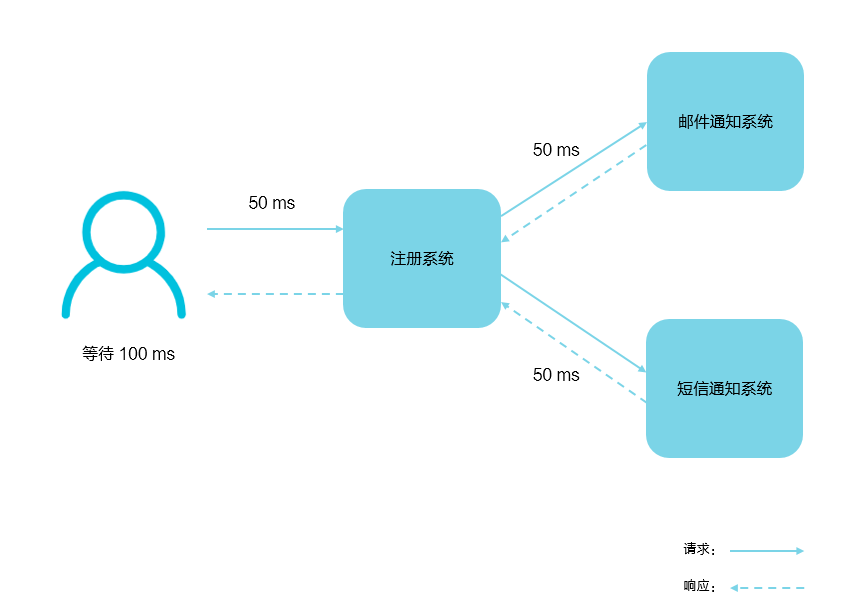
* 1. 用户在注册页面填写账号和密码并提交注册信息，这些注册信息首先会被写入注册系统成功。
  2. 注册信息写入注册系统成功后，再发送请求至邮件通知系统。邮件通知系统收到请求后向用户发送邮件通知。
  3. 邮件通知系统接收注册系统请求后再向下游的短信通知系统发送请求。短信通知系统收到请求后向用户发送短信通知。

以上三个任务全部完成后，才返回注册结果到客户端，用户才能使用账号登录。

假设每个任务耗时分别为50ms，则用户需要在注册页面等待总共需要150ms 才能登录。

* 并行方式

并行方式下的注册流程如下图所示。

[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6219985751/p68912.png)

数据流动如下所述：

* 1. 用户在注册页面填写账号和密码并提交注册信息，这些注册信息首先会被写入注册系统成功。
  2. 注册信息写入注册系统成功后，再同时发送请求至邮件和短信通知系统。邮件和短信通知系统收到请求后分别向用户发送邮件和短信通知。

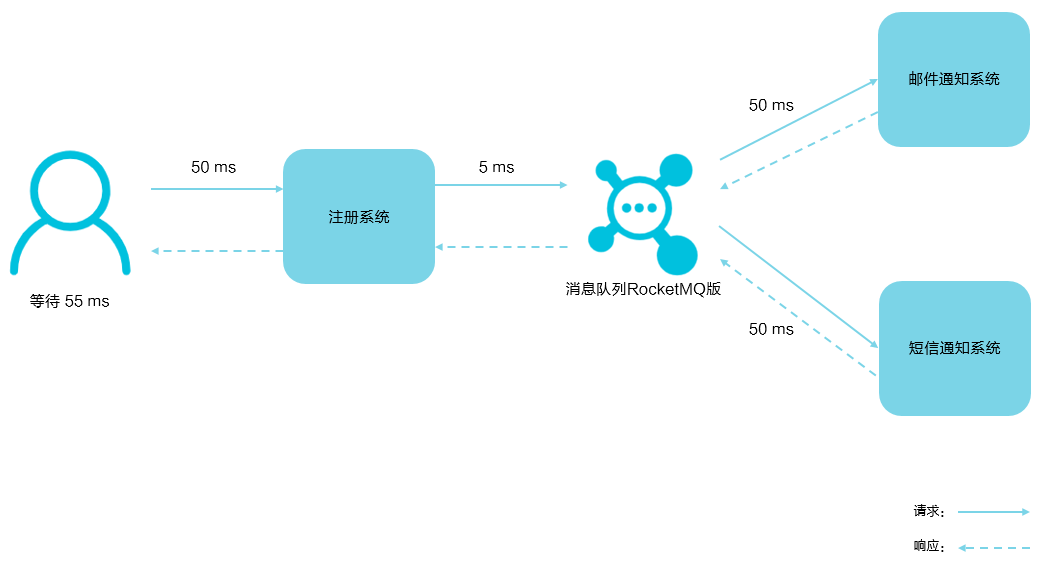
以上三个任务全部完成后，才返回注册结果到客户端，用户才能使用账号登录。

假设每个任务耗时分别为 50 ms，其中，邮件和短信通知并行完成，则用户需要在注册页面等待总共需要 100 ms 才能登录。

以下就注册场景中使用了消息队列 RocketMQ 版的效果进行说明。

**异步解耦**

对于用户来说，注册功能实际只需要注册系统存储用户的账户信息后，该用户便可以登录，后续的注册短信和邮件不是即时需要关注的步骤。

对于注册系统而言，发送注册成功的短信和邮件通知并不一定要绑定在一起同步完成，所以实际当数据写入注册系统后，注册系统就可以把其他的操作放入对应的消息队列 RocketMQ 版中然后马上返回用户结果，由消息队列 RocketMQ 版异步地进行这些操作。[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6219985751/p68913.png)

数据流动如下所述：

1. 用户在注册页面填写账号和密码并提交注册信息，这些注册信息首先会被写入注册系统成功。
2. 注册信息写入注册系统成功后，再发送消息至消息队列 RocketMQ 版。消息队列 RocketMQ 版会马上返回响应给注册系统，注册完成。用户可立即登录。
3. 下游的邮件和短信通知系统订阅消息队列 RocketMQ 版的此类注册请求消息，即可向用户发送邮件和短信通知，完成所有的注册流程。

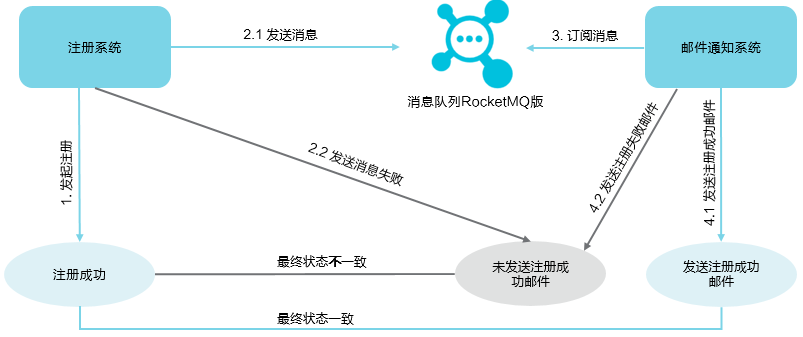
用户只需在注册页面等待注册数据写入注册系统和消息队列 RocketMQ 版的时间，即等待 55 ms 即可登录。

异步解耦是消息队列 RocketMQ 版的主要特点，主要目的是减少请求响应时间和解耦。主要的适用场景就是将比较耗时而且不需要即时（同步）返回结果的操作作为消息放入消息队列。同时，由于使用了消息队列 RocketMQ 版，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

注册系统注册的流程中，用户入口在网页注册系统，通知系统在邮件系统，两个系统之间的数据需要保持最终一致。

**普通消息处理**

如上所述，注册系统和邮件通知系统之间通过消息队列进行异步处理。注册系统将注册信息写入注册系统之后，发送一条注册成功的消息到消息队列 RocketMQ 版，邮件通知系统订阅消息队列 RocketMQ 版的注册消息，做相应的业务处理，发送注册成功或者失败的邮件。[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6219985751/p68914.png)

流程说明如下：

1. 注册系统发起注册。
2. 注册系统向消息队列 RocketMQ 版发送注册消息成功与否的消息。

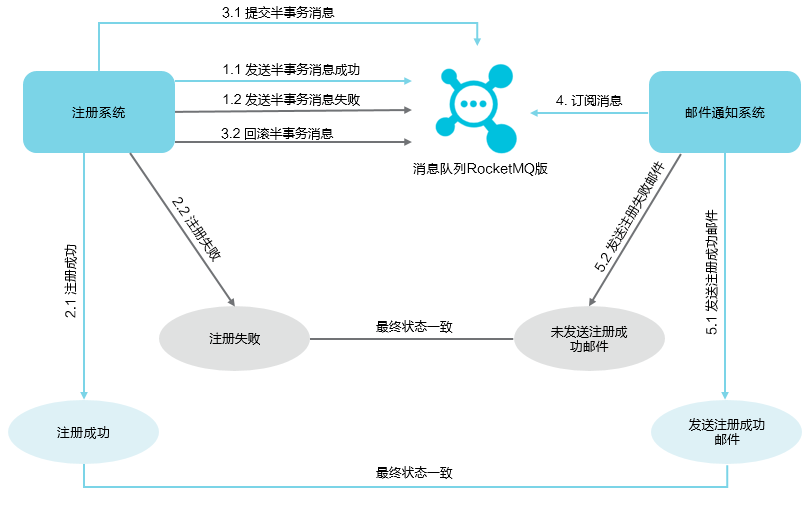
2.1 消息发送成功，进入 3。

2.2 消息发送失败，导致邮件通知系统未收到消息队列 RocketMQ 版发送的注册成功与否的消息，而无法发送邮件，最终邮件通知系统和注册系统之间的状态数据不一致。

1. 邮件通知系统收到消息队列 RocketMQ 版的注册成功消息。
2. 邮件通知系统发送注册成功邮件给用户。

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

**事务消息处理**

此时，需要有利用消息队列 RocketMQ 版所提供的事务消息来实现系统间的状态数据一致性。[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6219985751/p68919.png)

流程说明如下：

1. 注册系统向消息队列 RocketMQ 版发送半事务消息。

1.1 半事务消息发送成功，进入 2。

1.2 半事务消息发送失败，注册系统不进行注册，流程结束。（最终注册系统与邮件通知系统数据一致）

1. 注册系统开始注册。

2.1 注册成功，进入 3.1。

2.2 注册失败，进行 3.2。

1. 注册系统向消息队列 RocketMQ 版发送半消息状态。

3.1 提交半事务消息，产生注册成功消息，进入 4。

3.2 回滚半事务消息，未产生注册成功消息，流程结束。（最终注册系统与邮件通知系统数据一致）

1. 邮件通知系统接收消息队列 RocketMQ 版的注册成功消息。
2. 邮件通知系统发送注册成功邮件。（最终注册系统与邮件通知系统数据一致）

分布式事务消息的更多详细内容请参见[事务消息](https://help.aliyun.com/document_detail/43348.html#concept-2047067)。

## 消息的顺序收发

消息队列 RocketMQ 版顺序消息分为两种情况：

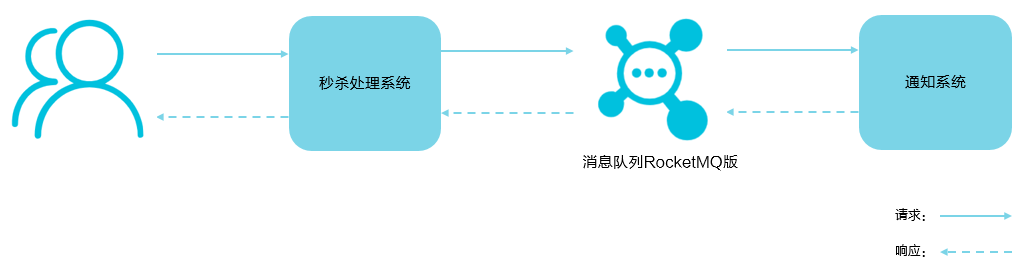
* 全局顺序：对于指定的一个 Topic，所有消息将按照严格的先入先出（FIFO）的顺序，进行顺序发布和顺序消费；
* 分区顺序：对于指定的一个 Topic，所有消息根据 Sharding Key 进行区块分区，同一个分区内的消息将按照严格的 FIFO 的顺序，进行顺发布和顺序消费，可以保证一个消息被一个进程消费。

在注册场景中，可使用用户 ID 作为 Sharding Key 来进行分区，同一个分区下的新建、更新或删除注册信息的消息必须按照 FIFO 的顺序发布和消费。

顺序消息的详细内容请参见[顺序消息](https://help.aliyun.com/document_detail/49319.html#concept-2047066)。

## 削峰填谷

流量削峰也是消息队列 RocketMQ 版的常用场景，一般在秒杀或团队抢购活动中使用广泛。

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，秒杀的应用在处理如此大量的访问流量后，下游的通知系统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。为解决这些问题，可在应用和下游通知系统之间加入消息队列 RocketMQ 版。[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/6219985751/p68920.png)

秒杀处理流程如下所述：

1. 用户发起海量秒杀请求到秒杀业务处理系统。
2. 秒杀处理系统按照秒杀处理逻辑将满足秒杀条件的请求发送至消息队列 RocketMQ 版。
3. 下游的通知系统订阅消息队列 RocketMQ 版的秒杀相关消息，再将秒杀成功的消息发送到相应用户。
4. 用户收到秒杀成功的通知。

## 大规模机器的缓存同步

双十一大促时，各个分会场会有玲琅满目的商品，每件商品的价格都会实时变化。使用缓存技术也无法满足对商品价格的访问需求，缓存服务器网卡满载。访问较多次商品价格查询影响会场页面的打开速度。

此时需要提供一种广播机制，一条消息本来只可以被集群的一台机器消费，如果使用消息队列 RocketMQ 版的广播消费模式，那么这条消息会被所有节点消费一次，相当于把价格信息同步到需要的每台机器上，取代缓存的作用。

广播消费的详细内容请参见[集群消费和广播消费](https://help.aliyun.com/document_detail/43163.html#concept-2047071)。

# 产品架构

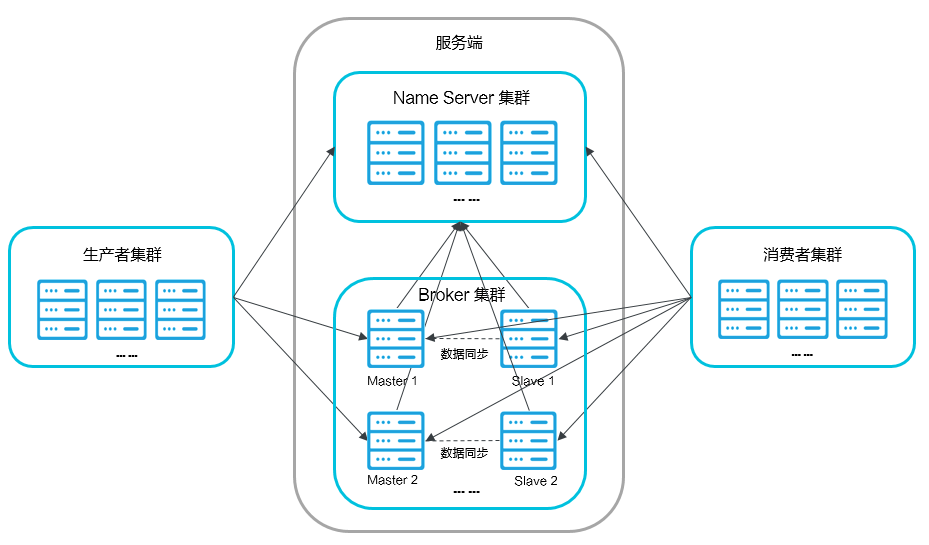
本文介绍消息队列 RocketMQ 版的系统部署架构，方便您更好地理解消息队列 RocketMQ 版的高可用性。

消息队列 RocketMQ 版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 RocketMQ 版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。

**注意** 文中所提及的消息队列 RocketMQ 版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。

[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/4229885751/p68921.png)

图中所涉及到的概念如下所述：

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

# 权益分发

在电商和新零售领域，权益分发适用于对特定的用户进行指定运营活动的场景。技术层面上，此种场景的 DB 和缓存的数据强一致性较难保证。针对该问题，阿里云消息队列 RocketMQ 版推出了权益分发解决方案。本文将以电商场景为例说明权益分发解决方案的背景信息、方案架构、以及方案优势。

## 背景信息

在电商平台运营决定对特定用户进行营销活动时，会针对不同的用户群设置不同的营销策略，可能会涉及但不局限于以下内容：

* 用户规则：针对什么样的用户下发权限，如新用户、会员等，结合业务需求设置具体的判断条件。
* 权限类型：红包、积分或是优惠券等不同类型的权益。
* 领取成本：需要付出成本才能获取相应权益，如消耗会员积分才能领取折扣券。
* 时间控制：发放权益的时间点，如早上 10:00 针对新用户下发 100 张优惠券。
* 库存：权益的个数是否充足、与配置的用户个数是否匹配等。

这些运营数据写入 DB 后，转换成技术策略，写入缓存，再转换成用户发放链路数据（变成一条一条的规则）。只有符合这些规则的用户才能领取对应的权益。

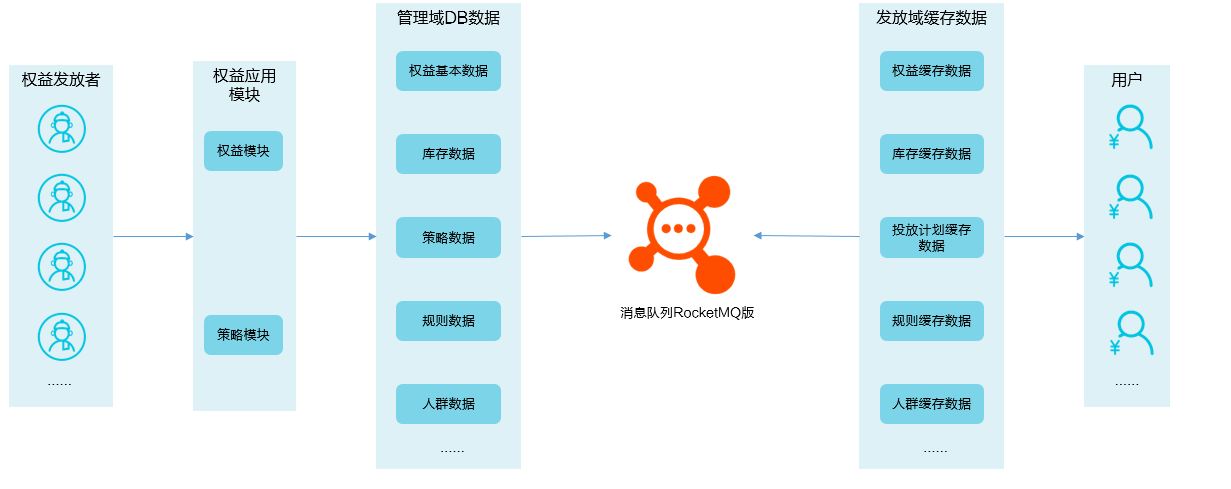
## 痛点

让用户通过访问缓存数据来领取对应的权益，这样既保证了用户高效的访问，也减轻了 DB 的压力。但是，这就会引发新的问题：

* 当规则特别多时，DB 写入缓存的数据量也会特别大，下游的缓存压力较大（此类情况较少）。
* DB 写入数据至缓存时只写一次，可能会因为网络抖动等原因导致数据写入失败，进而造成数据更新不及时的结果。例如，库存原定需设置 100,000，但运营策略变成 10 个，如果数据没有及时同步，那么就无法通知到下游，造成资损。

## 方案架构

因为消息队列 RocketMQ 版具有重试功能且能保证消息不丢失，所以推出此方案来确保 DB 和缓存数据的强一致性，即在 DB 和缓存间使用消息队列 RocketMQ 版，架构如下图所示。

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/7848093851/p88027.png)

## 方案优势

* 消息队列 RocketMQ 版拥有巨大的数据吞吐量。
* 消息队列 RocketMQ 版支持海量消息堆积，为下游应用减轻流量洪峰的冲击。
* 消息队列 RocketMQ 版可以帮助简化 DB 和缓存间的实现，大大减少代码开发量。如果没有消息队列 RocketMQ 版，实现方法则十分复杂。
* 消息队列 RocketMQ 版的实时消息收发以及重试功能可确保消息不丢，从而确保消息强一致性。

# 使用限制

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

具体的限制项和限制值请参见下表。

| **限制项** | **限制值** | **说明** |
| --- | --- | --- |
| 单个地域（Region）内的消息队列 RocketMQ 版实例数 | * 标准版：8 个 * 企业铂金版：数量不限 | 无 |
| Topic 名称长度 | 64 个字符 | Topic 名称长度不得超过该限制，否则会导致无法发送或者订阅。 |
| 消息大小 | * 普通和顺序消息：4 MB * 事务和定时/延时消息：64 KB   **说明** 其中，所有消息类型的消息属性大小均不能超过 16 KB。 | 消息大小不得超过其类型所对应的限制，否则消息会发送失败。 |
| 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。 |
| 单消息队列 RocketMQ 版实例的消息收发 TPS | * 标准版：5000 条/秒；如需更高规格，请[提交工单](https://selfservice.console.aliyun.com/ticket) * 企业铂金版：参考所购买的规格 | 无 |
| 定时/延时消息的延时时长 | 40 天 | msg.setStartDeliverTime 参数（单位：毫秒）可设置 40 天内的任何时刻，超过 40 天消息发送将失败。  **说明** 定时消息的精度有 1s ~ 2s 的延迟误差。 |
| 批量发送消息 | 不支持 | 消息队列 RocketMQ 版不支持批量发送消息。 |

# 名词解释

本文主要对消息队列 RocketMQ 版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 RocketMQ 版。

***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 最佳实践](https://help.aliyun.com/document_detail/95837.html#concept-2047146)。

***消息（Message）***

消息队列中信息传递的载体。

***Message ID***

消息的全局唯一标识，由消息队列 RocketMQ 版系统自动生成，唯一标识某条消息。

***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。

***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 最佳实践](https://help.aliyun.com/document_detail/95837.html#concept-2047146)。

***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 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。

***Consumer***

消息消费者，也称为消息订阅者，负责接收并消费消息。可分为两类：

* Push Consumer：消息由消息队列 RocketMQ 版推送至 Consumer。
* Pull Consumer：该类 Consumer 主动从消息队列 RocketMQ 版拉取消息。目前仅 TCP Java SDK 支持该类 Consumer。

**注意** 如需使用 Pull Consumer，请确保您的消息队列 RocketMQ 版实例为企业铂金版。

详情请参见[接口和参数说明](https://help.aliyun.com/document_detail/52591.html#concept1481)和[订阅消息](https://help.aliyun.com/document_detail/29551.html#concept-2047092)。

***分区***

即 Topic Partition，物理上的概念。每个 Topic 包含一个或多个分区。

***消费位点***

每个 Topic 会有多个分区，每个分区会统计当前消息的总条数，这个称为最大位点 MaxOffset；分区的起始位置对应的位置叫做起始位点 MinOffset。

消息队列 RocketMQ 版的 Pull Consumer 会按顺序依次消费分区内的每条消息，记录已经消费了的消息条数，称为消费位点 ConsumerOffset。剩余的未消费的条数（也称为消息堆积量）= 最大位点 MaxOffset - 消费位点 ConsumerOffset。

***Consumer 实例***

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***

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 的标识。

***队列***

每个 Topic 下会由一到多个队列来存储消息。每个 Topic 对应队列数与消息类型以及实例所处地域（Region）相关，具体的队列数可[提交工单](https://selfservice.console.aliyun.com/ticket/category/ons/today)咨询。

**注意** 标准版实例不支持变更队列数，铂金版实例支持变更队列数。

***Exactly-Once 投递语义***

Exactly-Once 投递语义是指发送到消息系统的消息只能被 Consumer 处理且仅处理一次，即使 Producer 重试消息发送导致某消息重复投递，该消息在 Consumer 也只被消费一次。详情请参见 [Exactly-Once 投递语义](https://help.aliyun.com/document_detail/113681.html#concept-2047070)。

***集群消费***

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](https://help.aliyun.com/document_detail/43163.html#concept-2047071)。

***广播消费***

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](https://help.aliyun.com/document_detail/43163.html#concept-2047071)。

***定时消息***

Producer 将消息发送到消息队列 RocketMQ 版服务端，但并不期望这条消息立马投递，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](https://help.aliyun.com/document_detail/43349.html#concept-2047065)。

***延时消息***

Producer 将消息发送到消息队列 RocketMQ 版服务端，但并不期望这条消息立马投递，而是延迟一定时间后才投递到 Consumer 进行消费，该消息即延时消息。详情请参见[定时和延时消息](https://help.aliyun.com/document_detail/43349.html#concept-2047065)。

***事务消息***

消息队列 RocketMQ 版提供类似 X/Open XA 的分布事务功能，通过消息队列 RocketMQ 版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](https://help.aliyun.com/document_detail/43348.html#concept-2047067)。

***顺序消息***

消息队列 RocketMQ 版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。详情请参见[顺序消息](https://help.aliyun.com/document_detail/49319.html#concept-2047066)。

***全局顺序消息***

对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。详情请参见[顺序消息](https://help.aliyun.com/document_detail/49319.html#concept-2047066)。

***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 Message Key 是完全不同的概念。详情请参见[顺序消息](https://help.aliyun.com/document_detail/49319.html#concept-2047066)。

***消息堆积***

Producer 已经将消息发送到消息队列 RocketMQ 版的服务端，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 RocketMQ 版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***

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 RocketMQ 版的服务端完成。详情请参见[消息过滤](https://help.aliyun.com/document_detail/29543.html#concept-2047069)。

***消息轨迹***

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 RocketMQ 版服务端，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](https://help.aliyun.com/document_detail/43357.html#concept-2335151)。

***重置消费位点***

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 RocketMQ 版服务端的消息。详情请参见[重置消费位点](https://help.aliyun.com/document_detail/63390.html#task-2047153)。

***死信队列***

死信队列用于处理无法被正常消费的消息。当一条消息初次消费失败，消息队列 RocketMQ 版会自动进行[消息重试](https://help.aliyun.com/document_detail/43490.html#concept-2047068)；达到最大重试次数后，若消费依然失败，则表明 Consumer 在正常情况下无法正确地消费该消息。此时，消息队列 RocketMQ 版不会立刻将消息丢弃，而是将这条消息发送到该 Consumer 对应的特殊队列中。

消息队列 RocketMQ 版将这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），将存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。

详情请参见[死信队列](https://help.aliyun.com/document_detail/87277.html#concept-2047154)。

***消息路由***

消息路由常用于不同地域之间的消息同步，保证地域之间的数据一致性。消息队列 RocketMQ 版的全球消息路由功能依托阿里云优质基础设施实现的高速通道专线，可以高效地实现不同地域之间的消息同步复制。详情请参见[全球消息路由](https://help.aliyun.com/document_detail/90483.html#concept-2047155)。

# Topic 与 Tag 最佳实践

在消息队列 RocketMQ 版中，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：

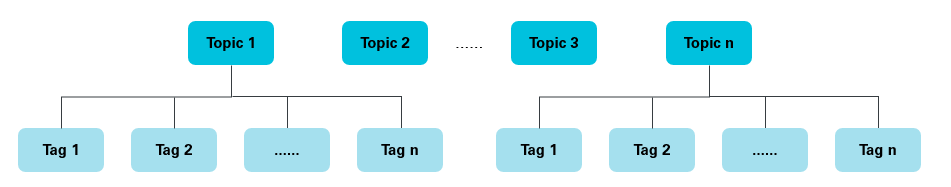
***Topic***

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。

[](http://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/9639885751/p69112.png)

## 适用场景

您可能会有这样的疑问：到底什么时候该用 Topic，什么时候该用 Tag？

建议您从以下几个方面进行判断：

* 消息类型是否一致：如普通消息、事务消息、定时（延时）消息、顺序消息，不同的消息类型使用不同的 Topic，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，如淘宝交易消息，京东物流消息使用不同的 Topic 进行区分；而同样是天猫交易消息，电器类订单、女装类订单、化妆品类订单的消息可以用 Tag 进行区分。
* 消息优先级是否一致：如同样是物流消息，盒马必须小时内送达，天猫超市 24 小时内送达，淘宝物流则相对会慢一些，不同优先级的消息用不同的 Topic 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。

## 场景示例

以天猫交易平台为例，订单消息和支付消息属于不同业务类型的消息，分别创建 **Topic\_Order** 和 **Topic\_Pay**，其中订单消息根据商品品类以不同的 Tag 再进行细分，列如电器类、男装类、女装类、化妆品类等被各个不同的系统所接收。

通过合理的使用 Topic 和 Tag，可以让业务结构清晰，更可以提高效率。

如何通过 Tag 实现消息过滤，请参见[消息过滤](https://help.aliyun.com/document_detail/29543.html#concept-2047069)。

# 消费幂等

为了防止消息重复消费导致业务处理异常，消息队列 RocketMQ 版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，那么整个过程就可实现消息幂等。

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 RocketMQ 版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 RocketMQ 版的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

* 负载均衡时消息重复（包括但不限于网络抖动、Broker 重启以及消费者应用重启）

当消息队列 RocketMQ 版的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息。

## 处理方法

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

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

Message message = **new** Message();

message.setKey("ORDERID\_100");

SendResult sendResult = producer.send(message);

消费者收到消息时可以根据消息的 Key，即订单号来实现消息幂等：

consumer.subscribe("ons\_test", "\*", **new** MessageListener() {

**public** Action consume(Message message, ConsumeContext context) {

**String** key = message.**getKey**()

// 根据业务唯一标识的 Key 做幂等处理

}

});

# 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 Consumer 实例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。本文提供订阅关系一致的正确示例以及订阅关系不一致的错误示例代码，帮助您顺畅地订阅消息。

## 背景信息

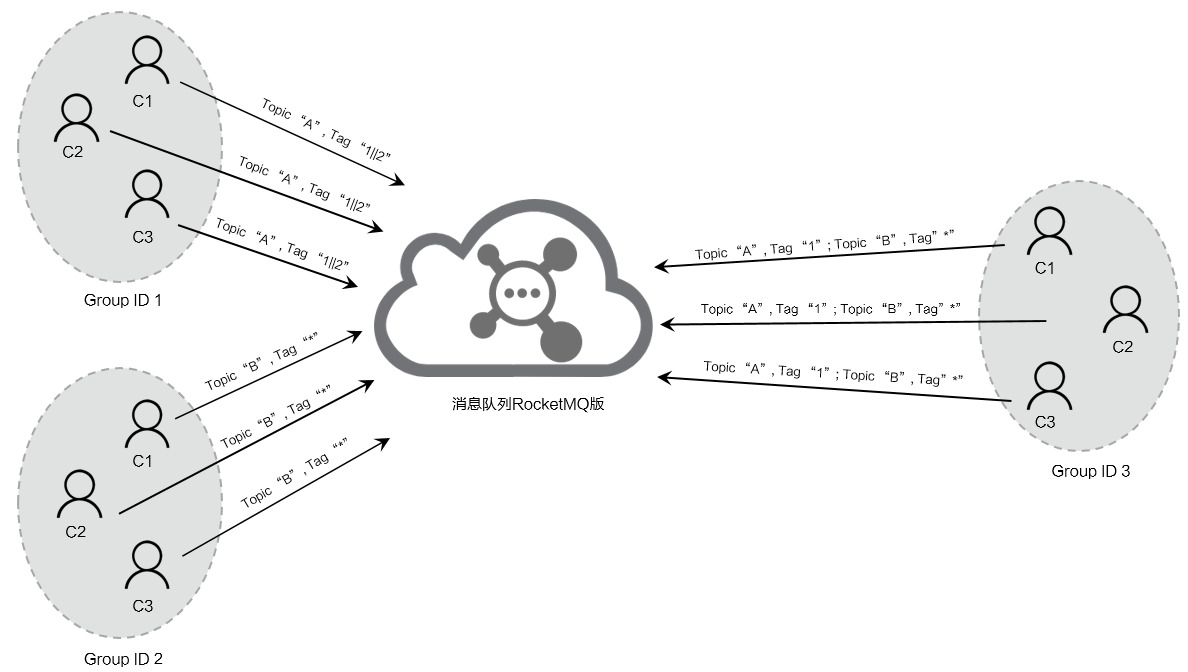
消息队列 RocketMQ 版里的一个消费者 Group ID 代表一个 Consumer 实例群组。对于大多数分布式应用来说，一个消费者 Group ID 下通常会挂载多个 Consumer 实例。

由于消息队列 RocketMQ 版的订阅关系主要由 Topic + Tag 共同组成，因此，保持订阅关系一致意味着同一个消费者 Group ID 下所有的实例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

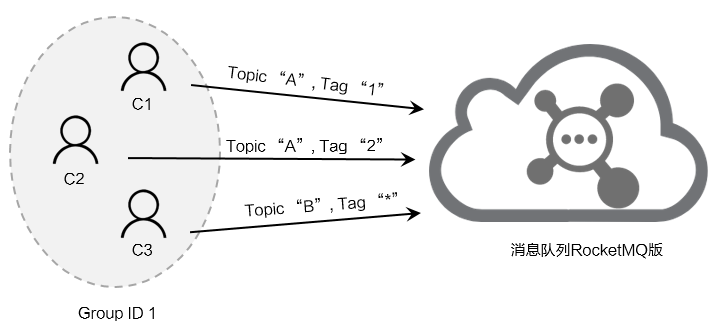
## 正确订阅关系图片示例

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

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/1769601951/p69113.png)

## 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 里的多个消费者实例的订阅关系并没有保持一致。

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/1769601951/p69117.png)

## 错误订阅关系代码示例一

以下例子中，同一个 Group ID 下的两个实例订阅的 Topic 不一致。

* Consumer 实例 1-1：
* Properties properties = **new** Properties();
* properties.put(PropertyKeyConst.GROUP\_ID, "GID\_jodie\_test\_1");
* Consumer consumer = ONSFactory.createConsumer(properties);
* consumer.subscribe("jodie\_test\_A", "\*", **new** MessageListener() {
* **public** Action **consume**(Message message, ConsumeContext context) {
* System.out.println(message.getMsgID());
* **return** Action.CommitMessage;
* }

});

* Consumer 实例 1-2：
* Properties properties = **new** Properties();
* properties.put(PropertyKeyConst.GROUP\_ID, "GID\_jodie\_test\_1");
* Consumer consumer = ONSFactory.createConsumer(properties);
* consumer.subscribe("jodie\_test\_B ", "\*", **new** MessageListener() {
* **public** Action **consume**(Message message, ConsumeContext context) {
* System.out.println(message.getMsgID());
* **return** Action.CommitMessage;
* }

});

## 错误订阅关系代码示例二

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

* Consumer 实例 2-1：
* Properties properties = **new** Properties();
* properties.put(PropertyKeyConst.GROUP\_ID, "GID\_jodie\_test\_2");
* Consumer consumer = ONSFactory.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 实例 2-2：
* Properties properties = **new** Properties();
* properties.**put**(PropertyKeyConst.GROUP\_ID, "GID\_jodie\_test\_2");
* Consumer consumer = ONSFactory.createConsumer(properties);
* consumer.subscribe("jodie\_test\_A", "\*", **new** MessageListener() {
* **public** Action consume(Message message, ConsumeContext context) {
* System.out.**println**(message.getMsgID());
* **return** Action.CommitMessage;
* }

});

## 错误订阅关系代码示例三

此例中，同一个 Group ID 下订阅 Topic 个数不一致，且订阅的 Topic 的 Tag 不一致。

* Consumer 实例 3-1：
* Properties properties = **new** Properties();
* properties.put(PropertyKeyConst.GROUP\_ID, "GID\_jodie\_test\_3");
* Consumer consumer = ONSFactory.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;
* }

});

* Consumer 实例 3-2：
* Properties properties = **new** Properties();
* properties.put(PropertyKeyConst.GROUP\_ID, "GID\_jodie\_test\_3");
* Consumer consumer = ONSFactory.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;
* }

});