消息队列 MQ

概念

概念

名词解释

本文主要对 MQ 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 MQ。

Message Queue

消息队列，阿里云商用的专业消息中间件，是企业级互联网架构的核心产品，提供基于高可用分布式集群技术搭建的消息发布订阅、轨迹查询、资源统计、定时（延时）、监控报警等一系列消息云服务。

Message

消息，消息队列中信息传递的载体。

Message ID

消息的全局唯一标识，由 MQ 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。

Topic

消息主题，一级消息类型，通过 Topic 对消息进行分类。

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。

Producer

消息生产者，也称为消息发布者，负责生产并发送消息。

Producer ID

一类 Producer 的标识，这类 Producer 通常生产并发送一类消息，且发送逻辑一致。

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。

Consumer ID

一类 Consumer 的标识，这类 Consumer 通常接收并消费一类消息，且消费逻辑一致。

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。

集群消费

一个 Consumer 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 Consumer 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。

广播消费

一个 Consumer 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 Consumer 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。

定时消息

Producer 将消息发送到 MQ 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 MQ 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。

事务消息

MQ 提供类似 X/Open XA 的分布事务功能，通过 MQ 事务消息能达到分布式事务的最终一致。

顺序消息

MQ 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。

顺序发布

对于指定的一个 Topic，客户端将按照一定的先后顺序进行发送消息。

顺序消费

对于指定的一个 Topic，按照一定的先后顺序进行接收消息，即先发送的消息一定会先被客户端接收到。

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。

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。

消息堆积

Producer 已经将消息发送到 MQ 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 MQ 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

订阅者可以根据消息标签（Tag）对消息进行过滤，确保订阅者最终只接收被过滤后的消息类型。消息过滤在 MQ 服务端完成。

消息轨迹

在一条消息从发布者发出到订阅者消费处理过程中，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，用户能清晰定位消息从发布者发出，经由 MQ 服务端，投递给消息订阅者的完整链路，方便定位排查问题。

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），重新设置消息订阅者对其订阅 Topic 的消费进度，设置完成后订阅者将接收设定时间点之后由消息发布者发送到 MQ 服务端的消息。

消息类型

普通消息

普通消息是指 MQ 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**。

收发普通消息的示例代码

Java

- 发送消息（三种方式）
- 发送消息（多线程）
- 订阅消息

C/C++

- 收发普通消息

.NET

- 收发普通消息

定时消息和延时消息

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

概念介绍

- 定时消息：Producer 将消息发送到 MQ 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 MQ 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。

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 MQ 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：比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 MQ 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，比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。
- 除 Java 语言支持延时消息外，其他语言都不支持延时消息。

示例代码

关于收发定时消息和延时消息的示例代码，请参考以下文档：

Java

- 收发定时消息
- 收发延时消息

C++

- 收发定时消息

.NET

- 收发定时消息

顺序消息

本文主要介绍 MQ 顺序消息的概念、适用场景以及使用过程中的注意事项。

概念介绍

顺序消息（FIFO 消息）是 MQ 提供的一种严格按照顺序进行发布和消费的消息类型。 顺序消息指消息发布和消息消费都按顺序进行。

顺序发布：对于指定的一个 Topic，客户端将按照一定的先后顺序发送消息。

顺序消费：对于指定的一个 Topic，按照一定的先后顺序接收消息，即先发送的消息一定会先被客户端接收到。

全局顺序

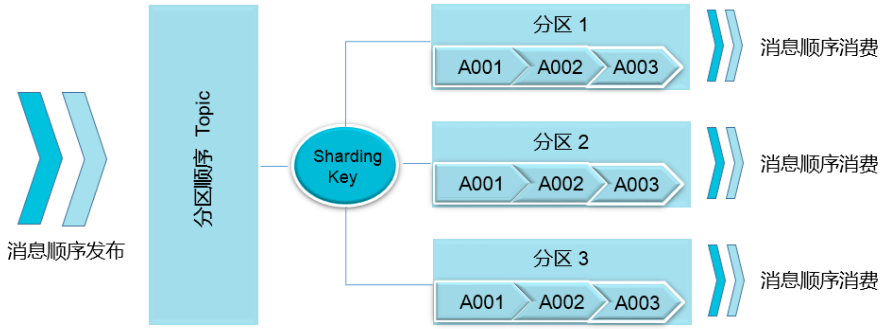
对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。

适用场景

性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景。

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。



适用场景

性能要求高，以 sharding key 作为分区字段，在同一个区块中严格的按照 FIFO 原则进行消息发布和消费的场景。

示例

【例一】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先后顺序来发布和订阅。

【例二】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先后顺序来发布和订阅。

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。

消息类型对比

Topic 的消息类型	支持事务消息	支持定时消息	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

发送方式对比

消息类型	支持可靠同步发送	支持可靠异步发送	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

- 顺序消息暂不支持广播模式。
- 同一个 Producer ID 或者 Consumer ID 只能对应一种类型的 Topic，即不能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。
- 对于全局顺序消息，建议创建实例个数 >=2。 同时运行多个实例的作用是为了防止工作实例意外退出时，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，实际同时工作的只会有一个实例。

SDK 支持和示例代码

请使用 Java SDK 1.2.7 及以上版本。

示例代码请参考以下文档：

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

事务消息

本文主要介绍 MQ 事务的概念、适用场景以及使用过程中的注意事项。

概念介绍

- 事务消息：MQ 提供类似 X/Open XA 的分布事务功能，通过 MQ 事务消息能达到分布式事务的最终一致。
- 半消息：暂不能投递的消息，发送方已经将消息成功发送到了 MQ 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，MQ 服务端通过扫描发现某条消息长期处于“半消息”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），该过程即消息回查。

适用场景

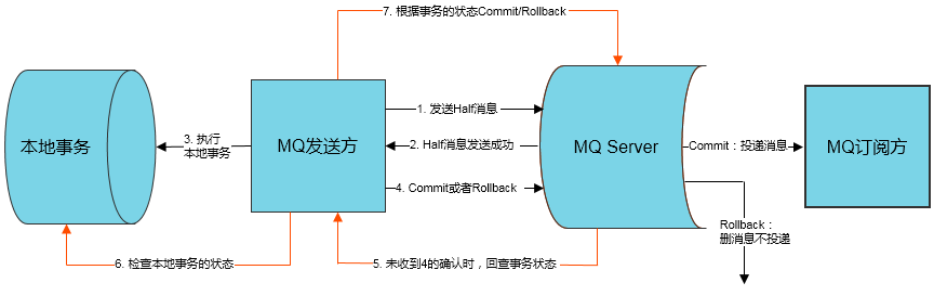
MQ 事务消息适用于以下场景：

- 帮助用户实现类似 X/Open XA 的分布事务功能，通过 MQ 事务消息能达到分布式事务的最终一致。

使用方式

交互流程

MQ 事务消息交互流程如下所示：



其中：

1. 发送方向 MQ 服务端发送消息。
2. MQ Server 将消息持久化成功之后，向发送方 ACK 确认消息已经发送成功，此时消息为半消息。
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 MQ Server 提交二次确认（Commit 或是 Rollback），MQ Server 收到 Commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 Rollback 状态则删除半消息，订阅方将不会接受该消息。
5. 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 MQ Server，经过固定时间后 MQ Server 将对该消息发起消息回查。
6. 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。
7. 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，MQ Server 仍按照步骤4对半消息进行操作。

说明：事务消息发送对应步骤 1、2、3、4，事务消息回查对应步骤 5、6、7。

注意事项

事务消息的 Producer ID 不能与其他类型消息的 Producer 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MQ Server会根据Producer ID去查询客户端。

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。

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，期待固定时间以后 MQ Server 向发送方进行消息回查。

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 ~ 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0~5秒；如第一次回查后事务仍未提交，后续每隔5秒回查一次。
```

示例代码

关于收发事务消息的示例代码，请参考以下文档：

- Java 收发事务消息
- C/C++ 收发事务消息
- .NET 收发事务消息

集群消费和广播消费

本文主要介绍 MQ 集群消费和广播消费的基本概念，适用场景以及注意事项。

基本概念

MQ 是基于发布订阅模型的消息系统。 消息的订阅方订阅关注的 Topic，以获取并消费消息。 由于订阅方应用一般是分布式系统，以集群方式部署有多台机器。 因此 MQ 约定以下概念。

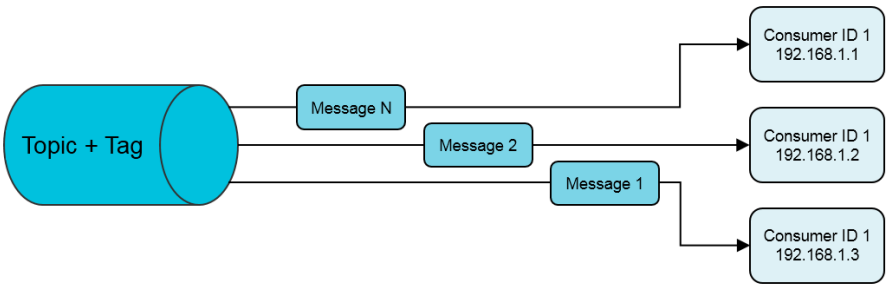
集群：MQ 约定使用相同 Consumer ID 的订阅者属于同一个集群。同一个集群下的订阅者消费逻辑必须完全一致（包括 Tag 的使用），这些订阅者在逻辑上可以认为是一个消费节点。

集群消费：当使用集群消费模式时，MQ 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，MQ 会将每条消息推送给集群内所有注册过的客户端，保证消息至少被每台机器消费一次。

场景对比

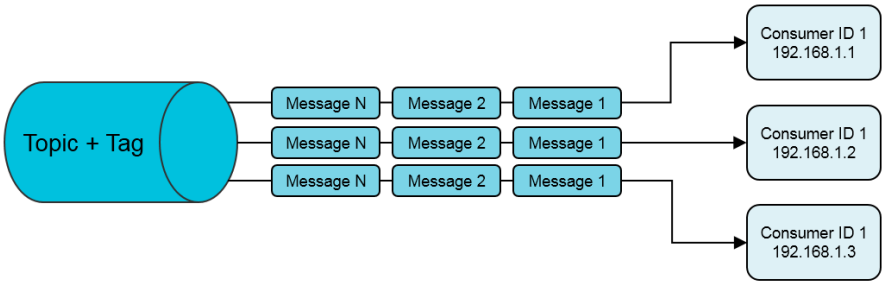
集群消费模式：



适用场景&注意事项

- 消费端集群化部署，每条消息只需要被处理一次。
- 由于消费进度在服务端维护，可靠性更高。
- 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，因此处理消息时不应该做任何确定性假设。

广播消费模式：

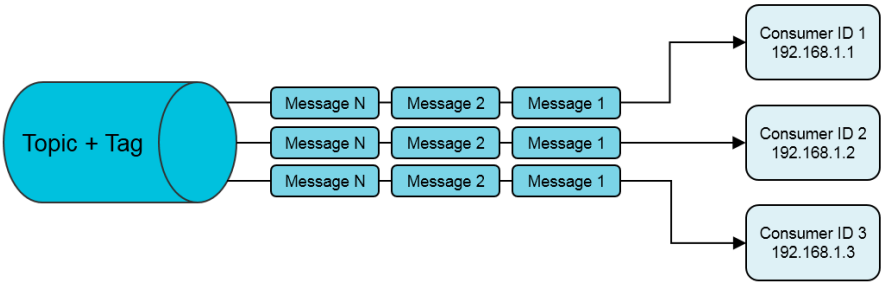


适用场景&注意事项

- 广播消费模式下不支持顺序消息。
- 每条消息都需要被相同逻辑的多台机器处理。
- 消费进度在客户端维护，出现重复的概率稍大于集群模式。
- 广播模式下，MQ 保证每条消息至少被每台客户端消费一次，但是并不会对消费失败的消息进行失败重投，因此业务方需要关注消费失败的情况。
- 广播模式下，客户端第一次启动时默认从最新消息消费。客户端的消费进度是被持久化在客户端本地的隐藏文件中，因此不建议删除该隐藏文件，否则会丢失部分消息。
- 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。
- 目前仅 Java 客户端支持广播模式。
- 广播模式下服务端不维护消费进度，所以 MQ 控制台不支持消息堆积查询和堆积报警功能。

使用集群模式模拟广播：

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



适用场景&注意事项

- 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。
- 消费进度在服务端维护，可靠性高于广播模式。
- 对于一个 Consumer ID 来说，可以部署一个消费端实例，也可以部署多个消费端实例。 当部署多个消费端实例时，实例之间又组成了集群模式（共同分担消费消息）。 假设 Consumer ID1 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 Consumer ID1 的消息。 同时，实例之间订阅关系必须保持一致。

订阅关系一致

MQ 里的一个 Consumer ID 代表一个 Consumer 实例群组。 对于大多数分布式应用来说，一个 Consumer ID 下通常会挂载多个 Consumer 实例。 订阅关系一致指的是同一个 Consumer 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。

由于 MQ 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 Consumer ID 下所有的实例需在以下两方面均保持一致：

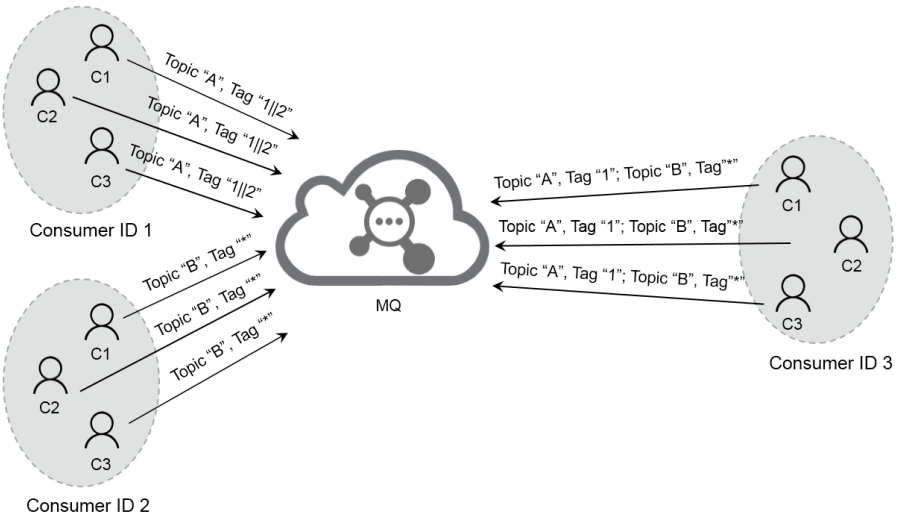
订阅的 Topic 必须一致；

订阅的 Topic 中的 Tag 必须一致。

订阅关系图片示例

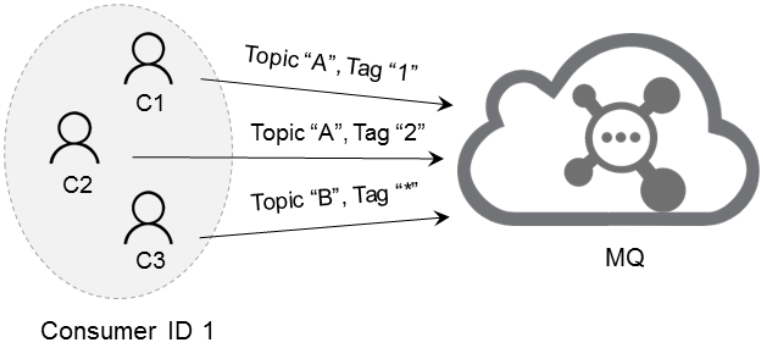
正确订阅关系图片示例

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



错误订阅关系图片示例

在下图中，单个 Consumer ID 订阅了多个 Topic，但是该 Consumer ID 里的多个消费者实例的订阅关系并没有保持一致。



订阅关系代码示例

错误订阅关系代码示例

【例一】

以下例子中，同一个 Consumer 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.ConsumerId, "CID_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.ConsumerId, " CID_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;
}
});
```

【例二】

以下例子中，同一个 Consumer 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.ConsumerId, "CID_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.ConsumerId, " CID_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;
}
});
```

【例三】

此例中，错误的原因有俩个：

1. 同一个 Consumer ID 下订阅 Topic 个数不一致。
2. 同一个 Consumer ID 下订阅 Topic 的 Tag 不一致。

Consumer 实例3-1：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, "CID_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.ConsumerId, " CID_jodie_test_3");
```

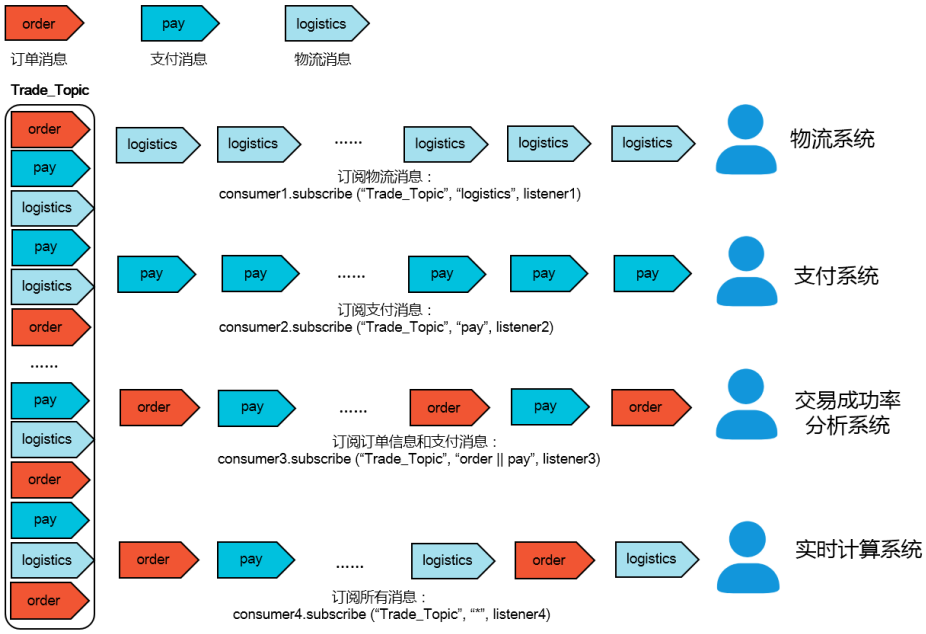
```
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A ", "TagB", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

消息过滤

本文描述 MQ 消费者如何根据 Tag 在 MQ 服务端完成消息过滤，关于 Topic 和 Tag 的介绍，请参考Topic & Tag 最佳实践。

Tag，即消息标签、消息类型，用于对某个 Topic 下的消息进行分类。 MQ 允许消费者按照 Tag 对消息进行过滤，确保消费者最终只消费到他关心的消息类型。

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，比如订单创建消息（order）、支付消息（pay）、物流消息（logistics）。 这些消息会发送到 Topic 为 Trade_Topic 的队列中，被各个不同的系统所接收，比如支付系统、物流系统、交易成功率分析系统、实时计算系统等。 其中，物流系统只需接收物流类型的消息（logistics），而实时计算系统需要接收所有和交易相关（order、pay、logistics）的消息。



说明：针对消息归类，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，比如全集和子集的关系，流程先后的关系。

参考示例

发送消息

发送消息时，每条消息必须指明 Tag：

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

消费方式-1

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;
}
});
```

消费方式-2

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;
}
});
```

消费方式-3

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;
}
});
```

消费方式-4（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 只能接收到 MQ_TOPIC 下 TagB 的消息，而不能接收 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;
}
});
```