# RocketMQ

## MQ的作用

### 应用解耦

系统耦合性越高，容错性就越低。如果一个系统耦合调用另一个系统，被调用系统出了故障或因为升级等原因暂时不可用，都会造成调用操作异常，影响用户使用。

使用消息队列解耦合，系统的耦合性就会提高了。一个系统调用另一个系统只需要将调用信息发送到MQ中后便可直接返回结果，被调用系统可异步的处理消息队列中的调用请求。

### 流量削峰

应用系统如果遇到系统请求流量的瞬间猛增，有可能会将系统压垮，有了消息队列可以将大量请求缓存起来，分散到很长的一段时间处理，这样可以大大提高系统的稳定性和用户体验。

### 数据分发

通过消息队列可以让数据在多个系统之间进行流通，数据的产生方不需要关心谁来使用数据，只需要将数据发送到消息队列，数据使用方直接在消息队列中直接获取数据即可。

## MQ的优点和缺点

### 优点

#### 解耦、削峰、数据分发

### 缺点

#### 系统可用性降低

系统引入的外部依赖越多，系统稳定性越差，一旦MQ宕机，就会对业务造成影响。

如何保证MQ的高可用？

#### 系统复杂度提高

MQ的加入大大增加了系统的复杂度，以前系统间是同步远程调用，现在是通过MQ进行异步调用。

如何保证消息没有被重复消费？怎么处理消息丢失情况？怎么保证消息传递的顺序性？

#### 一致性问题

A系统处理完业务，通过MQ给B、C、D三个系统发消息数据，如果B系统、C系统处理成功，D系统处理失败。

如何保证消息数据处理的一致性？

## MQ的比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特性 | ActiveMQ | RabbitMQ | RocketMQ | Kafka |
| 开发语言 | Java | Erlang | Java | Scala |
| 单机吞吐量 | 万级 | 万级 | 10万级 | 10万级 |
| 时效 | ms级 | us级 | ms级 | ms级以内 |
| 可用性 | 高（主从架构） | 高（主从架构） | 非常高（分布式架构） | 非常高（分布式架构） |
| 功能特性 | 成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持比较好 | 基于erlang开发，所以性能并发能力很强，性能极其好，延时很低；管理界面较丰富。 | MQ功能比较完备，扩展性佳 | 只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广 |

## RockerMQ快速入门

### 安装RocketMQ

#### 环境要求

* Linux64位系统
* Jdk1.8（64位）
* 源码安装需要安装Maven

#### 安装步骤

以二进制包方式安装

1. 解压安装包

unzip rocketmq-all-x.x.x-bin-release.zip

2、进入安装目录

cd rocketmq-all-x.x.x-bin-release

#### 目录介绍

* benchmark：提供一些测试功能
* bin：启动脚本，包括shell脚本和CMD脚本
* conf：实例配置文件，包括broker配置文件、logback配置文件等
* lib：依赖jar包，包括Netty、Commons-lang、FastJson等

### 启动RocketMQ

#### 启动NameServer

1. 启动NameServer

nohup sh bin/mqnamesrv &

1. 查看启动日志

tail -f ~/logs/rocketmqlogs/namesrv.log

#### 启动Broker

1. 启动Broker

nohup sh bin/mqbroker -n localhost:9876 &

1. 查看启动日志

tail -f ~/logs/rocketmqlogs/broker.log

#### 问题描述：

RocketMQ默认的虚拟机内存较大，启动Broker如果因为内存不足失败，需要编辑如下两个配置文件，修改JVM内存大小。

# 编辑runbroker.sh和runserver.sh 修改默认JVM内存大小

vi runbroker.sh

vi runserver.sh

**参考设置**

JAVA\_OPT=”${JAVA\_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m”

### 测试RocketMQ

#### 发送消息

1. 设置环境变量

export NAMESRV\_ADDR=localhost:9876

1. 使用安装包的Demo发送消息

sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer

#### 接收消息

1. 设置环境变量

export NAMESRV\_ADDR=localhost:9876

1. 接收消息

sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer

### 关闭RocketMQ

#### 关闭NameServer

sh bin/mqshutdown namesrv

#### 关闭Broker

sh bin/mqshutdown broker

## RocketMQ角色介绍

* Producer：消息的发送者
* Consumer：消息的接收者
* Broker：暂存和传输消息
* NameServer：管理Broker
* Topic：区别消息的种类；一个发送者可以发送消息给一个或多个Topic；一个消息的接收者可以订阅一个或多个Topic消息；
* Message Queue：相当于是Topic的分区；用于并行发送和接收消息。

## 集群搭建

暂不管； // TODO

## mqadmin集群管理工具 // TODO

### 使用方式

进入RocketMQ安装位置，在bin目录下执行 ./mqadmin {command} {args}

### 命令介绍 // TODO

Topic相关

## rocketmq-console集群监控 // TODO

## 消息发送样例

### 导入依赖

<!-- https://mvnrepository.com/artifact/org.apache.rocketmq/rocketmq-client -->

<dependency>

<groupId>org.apache.rocketmq</groupId>

<artifactId>rocketmq-client</artifactId>

<version>4.4.0</version>

</dependency>

### 消息发送者步骤分析

1. 创建消息生产者Producer，并制定生产者组名
2. 指定NameServer地址
3. 启动Broker
4. 创建消息对象，指定主题Topic、Tag和消息体
5. 发送消息
6. 关闭生产者Producer

### 消息接收者步骤分析

1. 创建消息消费者Consumer，制定消费者组名
2. 指定NameServer地址
3. 订阅主题Topic和Tag
4. 设置回调函数，处理消息
5. 启动消费者Consumer

### 基本样例

#### 发送消息

##### 发送同步消息

发送完消息时线程会阻塞，直到mq回传一个接收的结果。这种可靠性同步地发送方式使用的比较广泛，比如：重要的消息通知、短信通知等

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producerSync = new DefaultMQProducer("rocketMQ\_group");

//2. 指定NameServer地址

producerSync.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producerSync.start();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

Message message = new Message("rocketMQ\_Topic","Tag\_sync",("hello " + i).getBytes());

//5. 发送消息

SendResult send = producerSync.send(message);

System.out.println("发送结果：" + send);

Thread.sleep(500);

}

//6. 关闭生产者Producer

producerSync.shutdown();

##### 发送异步消息

发送完消息线程不会阻塞，可以开启一个回调函数来输出发送结果。通常用在对响应时间敏感的业务场景，即发送端不能容忍长时间地等待Broker的响应。

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producerAsync = new DefaultMQProducer("rocketMQ\_group");

//2. 指定NameServer地址

producerAsync.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producerAsync.start();

// producer\_sync.setAsyncSenderExecutor(); // 可以设置一个线程池，给其异步发送消息

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

Message message = new Message("rocketMQ\_Topic","Tag\_async",("hello " + i).getBytes());

//5. 异步发送消息

producerAsync.send(message, new **SendCallback**() {

// 回调函数来显示发送结果 --> 即异步发送消息

@Override

public void onSuccess(SendResult sendResult) {

System.out.println("OK sendResult = [" + sendResult + "]");

}

@Override

public void onException(Throwable throwable) {

System.out.println("NOT OK = [" + throwable + "]");

}

});

}

// 这里别关太快，不然异步发送还没发完，Producer就被关了

Thread.sleep(5000);

//6. 关闭生产者Producer

producerAsync.shutdown();

##### 单向发送消息

这种方式主要用于**生产者不关心发送结果**的场景，例如日志的发送。

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producerOneWay = new DefaultMQProducer("rocketMQ\_group");

//2. 指定NameServer地址

producerOneWay.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producerOneWay.start();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

Message message = new Message("rocketMQ\_Topic","Tag\_one\_way",("hello one way " + i).getBytes());

//5. 发送单向消息 sendOneway

producerOneWay.**sendOneway**(message);

}

//6. 关闭生产者Producer

producerOneWay.shutdown();

#### 消费消息

##### 基本消费消息

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumerSync = new DefaultMQPushConsumer("rocketMQ\_group");

//2. 指定NameServer地址

consumerSync.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和Tag

consumerSync.subscribe("rocketMQ\_Topic","Tag\_sync");

//4. 设置回调函数，处理消息

consumerSync.registerMessageListener((List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) -> {

list.forEach(x -> System.out.println("msg = " + new String(x.getBody())));

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

});

//5. 启动消费者Consumer 一直监听

consumerSync.start();

##### 消费者负载均衡模式（默认）

多个消费者共同消费（轮询或其他方式）同一组消息

// 设置消费方式 负载均衡模式（默认）

consumer.setMessageModel(MessageModel.CLUSTERING);

##### 消费者广播模式

每个消费者都消费同样的消息

// 设置消费方式 广播模式

consumer.setMessageModel(MessageModel.BROADCASTING);

### 顺序消息

#### 概念

消费者消费的消息的顺序需和生产者生产的消息顺序一致

注：消息通过Broker传递，Broker是队列结构，虽队列结构已满足先进先出的条件，但是Broker不止一个队列，而**Broker在接收消息时是采用轮询（队列）方式接收生产者的消息**，**消费者则是采用多线程的方式去消费Broker中队列中的消息**，则还是不能保证消费顺序和生产顺序一致。

#### 消息顺序

##### 全局消息顺序

没必要

##### 局部消息顺序

将一个局部消息只发送到Broker的一个队列中，而消费者也是只开一个线程去消费该队列的消息。

###### 发送顺序消息

// 1、创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producer = new DefaultMQProducer("OrderGroup");

// 2、制定NameServer地址

producer.setNamesrvAddr("192.168.0.141:9876");

// 3、启动Producer

producer.start();

// 4、发送多组消息,以组id选择队列顺序发送

// 消息组id

int[] m = new int[]{1,2,1,3,2,1,3,3,2};

for(int i = 0; i < 9; i ++) {

Message msg = new Message("OrderTopic","OrderTag",(m[i] + "的消息 第 " + i + "条").getBytes());

**/\*\***

**\* send方法**

**\* 参数一：消息对象**

**\* 参数二：消息队列选择器**

**\* 参数三：选择队列的业务标识（用来选择队列的标识）**

**\*/**

SendResult send = producer.send(msg, (List<MessageQueue> list, Message message, Object o) -> {

/\*\*

\* **消息队列选择器** **MessageQueueSelector**

\* 参数一：可选择的消息队列

\* 参数二：发送的消息

\* 参数三：传入的业务标识

\*/

int id = (int) o;

// 使用对id取模的方式，将同一id的消息按先后顺序发送到同一队列中

// （可能不连在一起，但保证了消息的先后顺序）

int index = list.size() % id;

return list.get(index);

}, m[i]);

System.out.println("send = " + send);

}

// 5、关闭生产者Producer

producer.shutdown();

###### 接收顺序消息

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("OrderGroup");

//2. 指定NameServer地址

consumer.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和Tag

consumer.subscribe("OrderTopic","\*");

//4. 设置回调函数，处理消息 **MessageListenerOrderly** = 一个队列开一个线程

consumer.registerMessageListener(**(List<MessageExt> list, ConsumeOrderlyContext consumeOrderlyContext) ->** {

list.forEach(x -> System.out.println(Thread.currentThread().getName() + " msg = " + new String(x.getBody())));

return ConsumeOrderlyStatus.SUCCESS;

});

//5. 启动消费者Consumer 一直监听

consumer.start();

### 延时消息

#### 概念

消息发送到队列后，消费者不能立即消费消息，需等待一定的时间才能消费。

#### 使用限制

1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h

传入Level，Level表示延时1s 以此类推，从1s到2h分别对应着等级1到18。

#### 发送样例

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producerSync = new DefaultMQProducer("DelayGroup");

//2. 指定NameServer地址

producerSync.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producerSync.start();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

// 参数一：Topic 参数二：Tag 参数三：消息

Message message = new Message("DelayTopic","TagDelay",("hello delay msg " + i).getBytes());

// {==延时消息设置==} 参数为level 1=1s 2=5s 3=10s ... 18=2h

**message.setDelayTimeLevel(2);**

//5. 发送消息

SendResult send = producerSync.send(message);

System.out.println("发送结果：" + send);

Thread.sleep(500);

}

//6. 关闭生产者Producer

producerSync.shutdown();

接收样例

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumerSync = new DefaultMQPushConsumer("DelayGroup");

//2. 指定NameServer地址

consumerSync.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和Tag

consumerSync.subscribe("DelayTopic","TagDelay");

//4. 设置回调函数，处理消息 MessageListenerConcurrently = 多线程获取各个队列消息

consumerSync.registerMessageListener((List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) -> {

list.forEach(x -> System.out.println("msg = " + new String(x.getBody()) + "延时时间 = " + (System.currentTimeMillis() - x.getStoreTimestamp())));

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

});

//5. 启动消费者Consumer 一直监听

consumerSync.start();

### 批量发送消息

一次发送多条消息

#### 发送样例

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producer = new DefaultMQProducer("BatchGroup");

//2. 指定NameServer地址

producer.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producer.start();

// 消息集合

List<Message> messages = new ArrayList<>();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

// 参数一：Topic 参数二：Tag 参数三：消息

Message message = new Message("BatchTopic","BatchTag",("hello batch msg " + i).getBytes());

messages.add(message);

}

//5. {==发送批量消息==} 注：**批量消息单次总大小不能大于4MB**

SendResult send = producer.send(messages);

System.out.println("发送结果：" + send);

//6. 关闭生产者Producer

producer.shutdown();

#### 接收样例

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumerSync = new DefaultMQPushConsumer("BatchGroup");

//2. 指定NameServer地址

consumerSync.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和Tag

consumerSync.subscribe("BatchTopic"," BatchTag ");

//4. 设置回调函数，处理消息 MessageListenerConcurrently = 多线程获取各个队列消息

consumerSync.registerMessageListener((List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) -> {

list.forEach(x -> System.out.println("msg = " + new String(x.getBody())));

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

});

//5. 启动消费者Consumer 一直监听

consumerSync.start();

### 过滤消息

#### 1、根据Tag过滤消息

consumerSync.subscribe("xxx","Tag || Tag1 || TagA");

##### 发送样例

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producer = new DefaultMQProducer("filterGroup");

//2. 指定NameServer地址

producer.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producer.start();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

// **{==发送Tag0,1,2的消息==} 发送多个Tag的消息**

Message message = new Message("filterTopic",**"filterTag" + (i % 3)**,("hello tag filter " + i).getBytes());

//5. 发送消息

SendResult send = producer.send(message);

System.out.println("发送结果：" + send);

}

//6. 关闭生产者Producer

producer.shutdown();

##### 接收样例

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("filterGroup");

//2. 指定NameServer地址

consumer.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和Tag **{==根据Tag过滤，只接收Tag0和Tag2的消息==}**

consumer.subscribe("filterTopic","**filterTag0 || filterTag2**");

//4. 设置回调函数，处理消息 MessageListenerConcurrently = 多线程获取各个队列消息

consumer.registerMessageListener((List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) -> {

list.forEach(x -> System.out.println("msg = " + new String(x.getBody()) + " || Tag = " + x.getTags()));

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

});

//5. 启动消费者Consumer 一直监听

consumer.start();

#### 2、根据sql语法过滤消息

##### Sql基本语法

* 数值比较，比如 > , >= , < , <= , BETWEEN , =
* 字符比较，比如 = , <> , IN
* IS NULL 或者 IS NOT NULL
* 逻辑符号 AND , OR , NOT

常量支持类型为：

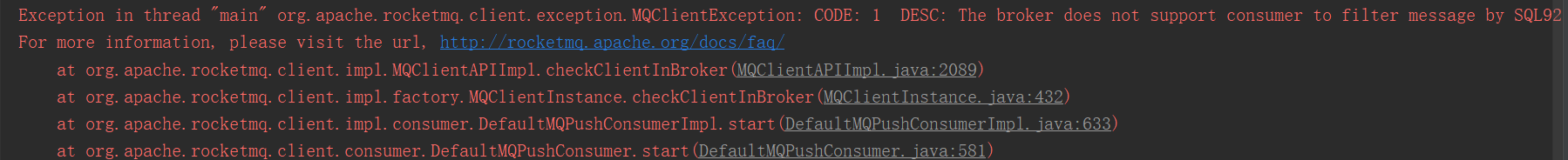
* 数值，比如：123 , 3.14
* 字符，比如：‘abc’ （必须用单引号包裹起来）
* NULL，特殊常量
* 布尔值，TRUE或FALSE

只有在使用push模式的消费者才能使用SQL92标准的sql语句，接口如下

public void subscribe(final String topic, final MessageSelector messageSelector);

##### 出现错误

在使用sql过滤时会出现“mq不支持sql过滤错误”



##### 解决方案

**TIPS** •该方式只支持RoketMQ，不支持Kafka/RabbitMQ•**用了sql，就不要用Tag**

RocketMQ支持使用SQL语法过滤消息。官方文档：http://rocketmq.apache.org/rocketmq/filter-messages-by-sql92-in-rocketmq/

**开启SQL 92支持**

默认情况下，RocketMQ的**SQL过滤支持是关闭的**，要想使用SQL 92过滤消息，需要：

1 在 conf/broker.conf 添加

enablePropertyFilter = true

2 启动RocketMQ 在后边带上配置文件

**nohup sh bin/mqbroker -n localhost:9876 -c ./conf/broker.conf &**

##### 发送样例

//1. 创建消息生产者Producer，并制定生产者组名

DefaultMQProducer producer = new DefaultMQProducer("filterGroup");

//2. 指定NameServer地址

producer.setNamesrvAddr("192.168.0.141:9876");

//3. 启动Broker （启动Producer实例）

producer.start();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 10; i < len; i ++) {

Message message = new Message("filterSqlTopic","filterTag",("hello sql filter " + i).getBytes());

// **{==指定用户属性，用于sql过滤条件==}**

**message.putUserProperty("i",String.valueOf(i));**

//5. 发送消息

SendResult send = producer.send(message);

System.out.println("发送结果：" + send);

}

//6. 关闭生产者Producer

producer.shutdown();

##### 接收样例

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("filterGroup");

//2. 指定NameServer地址

consumer.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和 用户消息属性条件 **{==根据sql过滤，只接收用户属性i >5的消息==}**

consumer.subscribe("filterSqlTopic",**MessageSelector.bySql("i>5")**);

//4. 设置回调函数，处理消息 MessageListenerConcurrently = 多线程获取各个队列消息

consumer.registerMessageListener((List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) -> {

list.forEach(x -> System.out.println("msg = " + new String(x.getBody())));

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

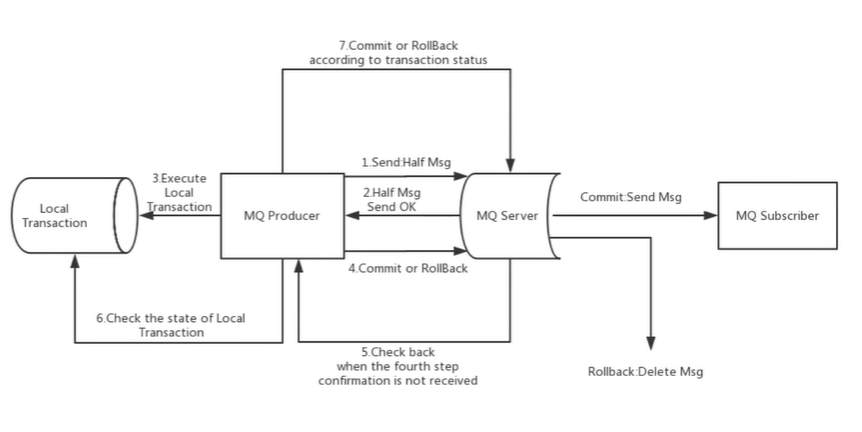
});

//5. 启动消费者Consumer 一直监听

consumer.start();

### 事务消息

#### 流程分析



基本分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程（没有commit和rollback的情况）。

##### 事务消息发送及提交

1. 发送消息（half消息）
2. 服务端响应消息写入结果
3. 根据发送结果执行本地事务（如果写入失败，此时half消息对业务不可见，本地逻辑不执行）
4. 根据本地事务状态执行Commit或者Rollback（Commit操作生成消息索引，消息对消费者可见）

##### 事务补偿

1. 对没有Commit\Rollback的事务消息（pending状态的消息），从服务端发起一次“回查”
2. Producer收到回查消息，检查回查消息对应的本地事务的状态
3. 根据本地事务状态，重新Commit或者Rollback

其中，补偿阶段用于解决消息Commit或Rollback发生超时或者失败的情况

#### 事务消息的状态

事务消息共有三种状态：提交状态、回滚状态、中间状态

* TransactionStatus.CommitTransaction：提交事务，它允许消费者消费此消息
* TransactionStatus.RollbackTransaction：回滚事务，它表示消息将被删除，不允许被消费
* TransactionStatus.Unknown：中间状态，它表示需要检查消息队列来确定状态

#### 发送样例

//1. 创建消息生产者Producer，并制定生产者组名 {==使用事务Producer==}

TransactionMQProducer producer = new **TransactionMQProducer**("TransactionGroup");

//2. 指定NameServer地址

producer.setNamesrvAddr("192.168.0.141:9876");

//  **{==设置事务监听器，用于服务端给Producer回查事务==}**

**producer.setTransactionListener(new TransactionListener()** {

/\*\*

\* 用于回查事务状态

\* @param message

\* @param o

\* @return

\*/

@Override

public LocalTransactionState executeLocalTransaction(Message message, Object o) {

if(StringUtils.equals("TAGA",message.getTags())) {

// 提交

return LocalTransactionState.COMMIT\_MESSAGE;

}else if(StringUtils.equals("TAGB",message.getTags())) {

// 回滚

return LocalTransactionState.ROLLBACK\_MESSAGE;

}else if(StringUtils.equals("TAGC",message.getTags())) {

// 回查 -> 事务补偿 注：server默认 UNKNOW 去事务补偿处理的等待时间较长

return LocalTransactionState.UNKNOW;

}

return LocalTransactionState.UNKNOW;

}

/\*\*

\* 用于事务补偿

\* @param messageExt

\* @return

\*/

@Override

public LocalTransactionState checkLocalTransaction(MessageExt messageExt) {

// 提交

return LocalTransactionState.COMMIT\_MESSAGE;

}

});

String[] tags = new String[]{"TAGA","TAGB","TAGC"};

//3. 启动Broker （启动Producer实例）

producer.start();

//4. 创建消息对象，指定主题Topic、Tag和消息体

for(int i = 0, len = 3; i < len; i ++) {

// 参数一：Topic 参数二：Tag 参数三：消息

Message message = new Message("TransactionTopic",tags[i],("hello transaction " + i).getBytes());

//5. 发送消息 **{==参数二为单条消息事务处理方式，这里统一使用事务监听器的处理方式，故传null==}**

SendResult send = producer.**sendMessageInTransaction**(message,null);

System.out.println("发送结果：" + send);

}

// **不关闭Producer，需等待事务监听工作**

// producer.shutdown();

#### 接收样例

//1. 创建消息消费者Consumer，制定消费者组名

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("TransactionGroup");

//2. 指定NameServer地址

consumer.setNamesrvAddr("192.168.0.141:9876");

//3. 订阅主题Topic和Tag 若接收到TAGA 和TAGC 的消息，则表示事物消息成功

consumer.subscribe("TransactionTopic","\*");

//4. 设置回调函数，处理消息 MessageListenerConcurrently = 多线程获取各个队列消息

consumer.registerMessageListener((List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) -> {

list.forEach(x -> System.out.println("msg = " + new String(x.getBody()) + " || TAG = " + x.getTags()));

return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;

});

//5. 启动消费者Consumer 一直监听

consumer.start();