# RocketMQ

# 简介

* RocketMQ 是一款分布式、队列模型的消息中间件，具有以下特点：
* 能够保证严格的消息顺序
* 提供丰富的消息拉取模式
* 高效的订阅者水平扩展能力
* 实时的消息订阅机制
* 亿级消息堆积能力
* 选用理由：
*  强调集群无单点，可扩展，任意一点高可用，水平可扩展。
*  海量消息堆积能力，消息堆积后，写入低延迟。
*  支持上万个队列
*  消息失败重试机制
*  消息可查询
*  开源社区活跃
*  成熟度（经过双十一考验)

# 基本概念

# 功能

# 发布订阅

# 消息优先级（支持多个队列，每个队列有着不同的优先级)

# 顺序消息（Producer单线程顺序发送，且发送到同一队列)

* 缺点：

# 消息过滤（Broker端过滤)

# 消息持久化（文件)

# 消息可靠性（同步双写保证数据不丢，异步复制保证最多丢失少量数据)

# 低延迟消息投递（长轮询)

# 消费确认（消费者消费后为消息进行确认)

# 消息重复（无法保证消息不重复，业务需要保证幂等)

* 消息有一个Key属性，用于消息去重，但是要手动保证。（比如存到数据库)

# 回溯消费

# 消息堆积

# 定时消息

# 消费重试

# 部署

# 物理结构

# 逻辑结构

# 配置方式

* 推荐使用双Master模式
* 如果要严格保证实时，那么需要使用多主多从的方式（异步复制or同步双写)。
* RT是响应时间的意思

# Producer使用

# 普通模式

* send发送消息

# 顺序模式

* 负载均衡与顺序消费不冲突，如果采用顺序消费，那么消息会经由一个broker中转，而非多个。实现是通过单个队列实现的。Producer把消息发送到同一个队列（添加一个队列selector，保证该类消息都发送到某个队列中)，Consumer从同一个队列中取消息进行消费。
* SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
* @Override

public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {

* Integer id = (Integer) arg;
* int index = id % mqs.size();
* return mqs.get(index);
* }
* }, orderId);

# 事务模式

* 流程图
* MQ可以避免分布式事务
* 事务（修改数据库)提交之前，先发送消息。
* 数据库事务提交后，才发送消息。
* 这样可以避免出现事务回滚而消息已发送的情况。
* 这就是一个再次确认的过程。
* 关键
* RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改消息的状态。
* 为解决确认消息发送失败的问题（消息回查)，RocketMQ会定期扫描消息集群中的事务消息，如果发现了Prepared消息，它会向消息发送端(生产者)确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。
* sendMessageInTransaction方法的源码，总共分为3个阶段：发送Prepared消息、执行本地事务、发送确认消息。

public TransactionSendResult sendMessageInTransaction(.....) {

* // 逻辑代码，非实际代码
* // 1.发送消息
* sendResult = this.send(msg);
* // sendResult.getSendStatus() == SEND\_OK
* // 2.如果消息发送成功，处理与消息关联的本地事务单元
* LocalTransactionState localTransactionState = tranExecuter.executeLocalTransactionBranch(msg, arg);
* // 3.结束事务
* this.endTransaction(sendResult, localTransactionState, localException);
* }
* endTransaction方法会将请求发往broker(mq server)去更新事务消息的最终状态：
  1. 根据sendResult找到Prepared消息

1. 根据localTransaction更新消息的最终状态

* 如果endTransaction方法执行失败，导致数据没有发送到broker，broker会有回查线程定时（默认1分钟)扫描每个存储事务状态的表格文件，如果是已经提交或者回滚的消息直接跳过，如果是prepared状态则会向Producer发起CheckTransaction请求，Producer会调用DefaultMQProducerImpl.checkTransactionState()方法来处理broker的定时回调请求，而checkTransactionState会调用我们的事务设置的决断方法，最后调用endTransactionOneway让broker来更新消息的最终状态。
* 在3.2.6以及之后版本中移除了消息回查的实现，所以此版本不支持事务消息。
* Producer存在的问题（确认消息发送失败时的消息回查和消息发送失败)
* 其实事务消息开始是prepare状态，然后RMQ会将其持久化到MySQL当中，然后如果收到确认消息，就删除掉这条prepare消息，如果迟迟收不到确认消息，那么RMQ会定时的扫描prepare消息，发送给produce group进行回查确认！
* 如果endTransaction方法执行失败，数据没有发送到broker，导致事务消息的状态更新失败，broker会有回查线程定时（默认1分钟)扫描每个存储事务状态的表格文件，如果是已经提交或者回滚的消息直接跳过，如果是prepared状态则会向Producer发起CheckTransaction请求，Producer会调用
* DefaultMQProducerImpl.checkTransactionState()方法来处理broker的定时回调请求，而checkTransactionState会调用我们的事务设置的决断方法来决定是回滚事务还是继续执行，最后调用endTransactionOneway让broker来更新消息的最终状态。
* Producer的send方法本身支持内部重试，重试逻辑如下：
  1. 至多重试3次。
  2. 如果发送失败，则轮转到下一个Broker。
  3. 这个方法的总耗时时间不超过sendMsgTimeout设置的值，默认10s。
* 所以，如果本身向broker发送消息产生超时异常，就不会再做重试。
* 以上策略仍然不能保证消息一定发送成功，为保证消息一定成功，建议应用这样做：
* 如果调用send同步方法发送失败，则尝试将消息存储到db，由后台线程定时重试，保证消息一定到达Broker。
* Consumer存在的问题（消费失败和消费超时)
* 会出现消费失败和消费超时两个问题，解决超时问题的思路就是一直重试，直到消费端消费消息成功，整个过程中有可能会出现消息重复的问题，按照前面的思路解决即可（去重表)。
* 这样基本上可以解决消费端超时问题，但是如果消费失败怎么办？阿里提供给我们的解决方法是：人工解决。
* 阿里提供的示例代码（基于闭源RocketMQ)
* Producer 执行事务

public class TransactionProducerClient {

private final static Logger log = ClientLogger.getLog(); // 用户需要设置自己的 log, 记录日志便于排查问题

public static void main(String[] args) throws InterruptedException {

* final BusinessService businessService = new BusinessService(); // 本地业务 Service
* Properties properties = new Properties();
* // 您在控制台创建的 Producer ID。注意：事务消息的 Producer ID 不能与其他类型消息的 Producer ID 共用
* properties.put(PropertyKeyConst.ProducerId, "");
* // 阿里云身份验证，在阿里云服务器管理控制台创建
* properties.put(PropertyKeyConst.AccessKey, "");
* // 阿里云身份验证，在阿里云服务器管理控制台创建
* properties.put(PropertyKeyConst.SecretKey, "");
* // 设置 TCP 接入域名（此处以公共云生产环境为例)
* properties.put(PropertyKeyConst.ONSAddr,
* "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
* TransactionProducer producer = ONSFactory.createTransactionProducer(properties,
* new LocalTransactionCheckerImpl());
* producer.start();
* Message msg = new Message("Topic", "TagA", "Hello MQ transaction===".getBytes());
* try {
* SendResult sendResult = producer.send(msg, new LocalTransactionExecuter() {
* @Override

public TransactionStatus execute(Message msg, Object arg) {

* // 消息 ID(有可能消息体一样，但消息 ID 不一样, 当前消息 ID 在控制台无法查询)
* String msgId = msg.getMsgID();
* // 消息体内容进行 crc32, 也可以使用其它的如 MD5
* long crc32Id = HashUtil.crc32Code(msg.getBody());
* // 消息 ID 和 crc32id 主要是用来防止消息重复
* // 如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
* // 如果要求消息绝对不重复, 推荐做法是对消息体 body 使用 crc32或 md5来防止重复消息
* Object businessServiceArgs = new Object();
* TransactionStatus transactionStatus = TransactionStatus.Unknow;
* try {
* boolean isCommit =
* // 执行本地事务
* businessService.execbusinessService(businessServiceArgs);
* if (isCommit) {
* // 本地事务成功、提交消息
* transactionStatus = TransactionStatus.CommitTransaction;
* } else {
* // 本地事务失败、回滚消息
* transactionStatus = TransactionStatus.RollbackTransaction;
* }
* } catch (Exception e) {
* log.error("Message Id:{}", msgId, e);
* }
* System.out.println(msg.getMsgID());
* log.warn("Message Id:{}transactionStatus:{}", msgId, transactionStatus.name());
* return transactionStatus;
* }
* }, null);
* }
* 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:" + msg.getTopic());
* e.printStackTrace();
* }
* // demo example 防止进程退出(实际使用不需要这样)
* TimeUnit.MILLISECONDS.sleep(Integer.MAX\_VALUE);
* }
* }
* Producer 消息回查

public class LocalTransactionCheckerImpl implements LocalTransactionChecker {

private final static Logger log = ClientLogger.getLog();

* final BusinessService businessService = new BusinessService();
* @Override

public TransactionStatus check(Message msg) {

* //消息 ID(有可能消息体一样，但消息 ID 不一样, 当前消息属于 Half 消息，所以消息 ID 在控制台无法查询)
* String msgId = msg.getMsgID();
* //消息体内容进行 crc32, 也可以使用其它的方法如 MD5
* long crc32Id = HashUtil.crc32Code(msg.getBody());
* //消息 ID、消息本 crc32Id 主要是用来防止消息重复
* //如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
* //如果要求消息绝对不重复, 推荐做法是对消息体使用 crc32或 md5来防止重复消息.
* //业务自己的参数对象, 这里只是一个示例, 实际需要用户根据情况来处理
* Object businessServiceArgs = new Object();
* TransactionStatus transactionStatus = TransactionStatus.Unknow;
* try {
* boolean isCommit = businessService.checkbusinessService(businessServiceArgs);
* if (isCommit) {
* //本地事务已成功、提交消息
* transactionStatus = TransactionStatus.CommitTransaction;
* } else {
* //本地事务已失败、回滚消息
* transactionStatus = TransactionStatus.RollbackTransaction;
* }
* } catch (Exception e) {
* log.error("Message Id:{}", msgId, e);
* }
* log.warn("Message Id:{}transactionStatus:{}", msgId, transactionStatus.name());
* return transactionStatus;
* }
* }
* Consumer 远程事务

public class ConsumerTest {

public static void main(String[] args) {

* Properties properties = new Properties();
* // 您在控制台创建的 Consumer ID
* properties.put(PropertyKeyConst.ConsumerId, "XXX");
* // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
* properties.put(PropertyKeyConst.AccessKey, "XXX");
* // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
* properties.put(PropertyKeyConst.SecretKey, "XXX");
* // 设置 TCP 接入域名（此处以公共云生产环境为例)
* properties.put(PropertyKeyConst.ONSAddr,
* "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
* // 集群订阅方式 (默认)
* // properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
* // 广播订阅方式
* // properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
* Consumer consumer = ONSFactory.createConsumer(properties);
* consumer.subscribe("TopicTestMQ", "TagA||TagB", new MessageListener() { //订阅多个 Tag

public Action consume(Message message, ConsumeContext context) {

* // 执行本地事务
* System.out.println("Receive: " + message);
* return Action.CommitMessage;
* }
* });
* //订阅另外一个 Topic
* consumer.subscribe("TopicTestMQ-Other", "\*", new MessageListener() { //订阅全部 Tag

public Action consume(Message message, ConsumeContext context) {

* System.out.println("Receive: " + message);
* return Action.CommitMessage;
* }
* });
* consumer.start();
* System.out.println("Consumer Started");
* }
* }
* 外围实现消息回查+消费端消息去重
* A持有所有本地事务执行成功的消息，B持有远程事务执行成功、已被消费的消息，两者的差集为确认消息发送失败的消息。
* 同时
* A也知道超过重试发送次数的消息
* B也知道超过重试消费次数的消息
* 这两份消息都需要人工介入处理（重试或放弃，不能直接放弃，需要人工回滚A的本地事务)。
* 当重试时，为了能更新A的数据，需要把消息状态设置为未被消费，并将sendTimes重置为0。
* 当回滚时，需要把消息状态设置为已被回滚。
* 注意A和B的数据不一定是最终一致的，比如A中某消息回滚后不会通知B（否则这又构成一个分布式事务)，但起码A的数据是准确的。
* A
  + 1) db  
    producer*msg(msgId,body,message*status,create*time,update*time,send\_times,topic) msgId这里为orderId
  + 2) mq
* 作为producer时，注册Topic :account：当执行本地事务时同时插入producer*msg，默认status都是未被消费。如果本地事务执行失败，那么直接回滚，不插入。当消息发送失败时，我们已经在producer*msg插入了记录，可以进行回查。
  + 3) scheduler
* A需要同步B的数据库，使得两个数据库数据一致，不同的即为确认信息发送失败的。
* 消息状态有未被消费、已被消费、消费失败、超过消费失败重试次数、超过确认消息发送失败的重试次数和已被回滚。
* A和B数据库同步维护所有消息，只是A数据库保存内容更多，比如会保存消息的body。
* 如果消息已经是超过重试次数或已被消费，那么A不会再去考虑它。
* A的Scheduler会遍历A数据库，找出未被消费和消费失败的id且创建时间距离当前时间超过1min，发送给B（作为一个消息，topic可以共用之前的topic，在消费端用keys去区分是事务消息还是回查消息；因为如果使用RPC的话会造成producer与consumer存在耦合，使用消息可以解耦)。
* B会遍历这些id
* for(id in ids){
* 如果 id 不存在，说明确认消息发送失败，
* 如果 id 存在，则将该id对应的status一并返回，map.put(id,status)
* }
* 返回时也是发送一条消息，重新选取一个topic，比如是之前那个topic+”-check”
* A 设置一个consumer，接收到topic+”check”的消息后，接收到map后，keySet取得所有id，拿发送过去的id减去这些id（差集)，就是确认消息发送失败的消息，进行重新发送；遍历map，将本地数据库同步为B数据库。
* 这个方法可能会出现消息重复，因为A刚发送消息，B还没有处理，A的Scheduler就去查询了，当然消息都没有被消费，因为A会重发刚才的消息，但是B有做消息去重，所以不会影响。
* B
  + 1) db
* consumer*msg(msgId,create*time. message\_status,topic) msgId这里是orderId
  + 2) mq
* 作为consumer，注册Topic: account：
* 当接收到消息后，查询是否被执行过，如果没有被消费过（id未找到)或者消费失败了，则执行远程事务后插入/更新consumer\_ msg（status为已被消费)，已被消费则跳过。
* 远程事务执行失败时，插入/更新consumer\_ msg（status为消费失败)
* 超过重试消费次数的消息也更新consumer\_ msg，status为超过消费的重试次数。
* B这里就维护它所接收的消息的状态。
* 消息管理系统
* 就Producer而言：
* 消息管理子系统就是在网页上查看数据库中存储的消息（包括所有发送过的消息)。
* 按消息id搜索消息，按消息状态搜索消息，按topic搜索
* 而且还可以处理超过（确认消息)重试次数的消息，比如(批量)重新发送，或者回滚A事务。
* 另外还可以处理超过（消费)重试次数的消息，比如重新发送消息，或者回滚A事务。
* 而RocketMQ等客户端只能看到成功发送的消息，不能查看未被确认发送的消息和重发次数。

# 最佳实践

## 发送消息

## 发送消息失败

## oneway

# Consumer使用

* 分类
* 一般使用PushConsumer。
* PushConsumer是监听消息，而PullConsumer是主动拉取。

@Slf4j  
public class PullConsumer {  
  
 private static final Map<MessageQueue, Long> offseTable = new HashMap<MessageQueue, Long>();  
  
 public static void main(String[] args) throws MQClientException {  
  
 DefaultMQPullConsumer consumer = new DefaultMQPullConsumer("please\_rename\_unique\_group\_name\_5");  
 consumer.setNamesrvAddr("192.168.0.113:9876;192.168.0.114:9876");  
 consumer.start();  
 Set<MessageQueue> mqs = consumer.fetchSubscribeMessageQueues("TopicTest");  
 for (MessageQueue mq : mqs) {  
 System.out.println("Consume from the queue: " + mq);  
 SINGLE\_MQ:  
 while (true) {  
 try {  
 PullResult pullResult =  
 consumer.pullBlockIfNotFound(mq, null, getMessageQueueOffset(mq), 32);  
 log.info("{}",pullResult);  
 putMessageQueueOffset(mq, pullResult.getNextBeginOffset());  
 switch (pullResult.getPullStatus()) {  
 case FOUND:  
 List<MessageExt> list = pullResult.getMsgFoundList();  
 for (MessageExt msg : list) {  
 log.info("{}",new String(msg.getBody()));  
 }  
 break;  
 case NO\_MATCHED\_MSG:  
 break;  
 case NO\_NEW\_MSG:  
 break SINGLE\_MQ;  
 case OFFSET\_ILLEGAL:  
 break;  
 default:  
 break;  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 consumer.shutdown();  
 }  
  
 private static void putMessageQueueOffset(MessageQueue mq, long offset) {  
 offseTable.put(mq, offset);  
 }  
  
 private static long getMessageQueueOffset(MessageQueue mq) {  
 Long offset = offseTable.get(mq);  
 if (offset != null)  
 return offset;  
 return 0;  
 }  
}

* 配置

# 最佳实践

## 消费端去重

## 消息消费并行度

## 批量消费

# 原理

# 架构

# 存储

## 零拷贝（mmap或sendfile)

## 文件系统

## 数据存储结构

* consume queue是消息的逻辑队列，相当于字典的目录，用来指定消息在物理文件commit log上的位置。
* 我们可以在配置中指定consumequeue与commitlog存储的目录  
  每个topic下的每个queue都有一个对应的consumequeue文件，比如：
* ${rocketmq.home}/store/consumequeue/${topicName}/${queueId}/${fileName}

# 负载均衡

## Producer负载均衡

## Consumer负载均衡

# 单队列并行消费

# 消息过滤

## Filter

* 总结：