RocketMQ笔记

基础概念

1. RocketMQ是什么，有哪些功能?
2. RocketMQ有哪些组件，集群架构是怎么样的

系统架构

- NameServer：broker的注册中心，管理broker集群元数据
- broker集群：接收消息，存储消息，转发消息；基于主从架构实现数据多副本和高可用
- 生产者：消息发送方
- 消费者：消息接收方

NameServer

- NameServer集群化部署，保证高可用，管理broker集群元数据
- 每个broker启动都会向**所有**的NameServer进行注册
- 多个NameServer之间无通信，挂了一台，其他可用继续提供服务
- 生产者，消费者主动从NameServer拉取broker元数据更新

broker挂了，NameServer如何感知

> 心跳机制

- Broker每隔30s会给所有的NameServer发送心跳，告诉NameServer自己还活着，NameServer收到一个Broker心跳，就会更新一下它最近一次心跳的时间
- NameServer会每隔10s运行一个任务，检查一下各个Broker最近一次的心跳时间，如果某个Broker超过120s都没发送心跳了，就认为这个broker挂了

【核心】

NameServer集群化部署

Broker会注册到所有的NameServer上去

30s心跳机制和120s故障检测机制

生产者消费者客户端容错机制

Broker

主从架构

- 主从数据同步方式：slave broker通过pull模式，不断的从master broker中拉取数据
  - 如果是master向slave推，那么master必须维护所有broker的节点，以及所有broker当前的同步进度
    - 什么时候推又成了一个问题，写一条消息推一次，还是批量推一次，同步推送还是异步推送?
  - slave向master拉取，只需在每个slave broker中维护它是master是谁，当前的同步进度是多少就可以了
- 消费者从哪个broker拉取数据？有可能从master，有可能从slave
  - 在master返回数据给消费者时，会根据自己的负载情况，以及slave的同步情况，向消费者建议下一次拉取数据的时候从master拉取还是slave拉取
  - 如果master负载太大，slave数据与master同步的差不多，会建议从slave中去拉取
  - 如果slave数据落后master太多，没法从slave中获取最新的消息，则仍然会去master中拉取
- broker挂了怎么办?
  - 如果是slave挂了，影响不大，消息的写入和读取仍旧可以通过master，整体影响不大，只是master会承担更多的读请求
  - 如果master挂了
    - 在4.5版本之前，slave无法自动切换为master，需要手动修改配置，重启，导致一段时间不可用
    - 4.5之后，通过Dledger实现自动切换（基于Raft协议），一旦master宕机，可以在多个slave中选举出一个新的master，对外提供服务

【重点】数据存储

> 决定了生产者消息写入的吞吐量，决定了消息不能丢失，决定了消费者获取消息的吞吐量

CommitLog

Broker收到消息，将消息按顺序写入CommitLog文件，追加在尾部。

CommitLog是多个磁盘文件，每个文件大小最多为1G，如果一个文件写满了，就会创建一个新的CommitLog文件

MessageQueue

在Broker中，对于Topic下每个MessageQueue都会有一系列的**ConsumeQueue**文件。

在磁盘上，会有以下格式的一系列文件：

```
$HOME/store/consumequeue/{topic}/{queueId}/{filename}
# 例如：/store/consumequeue/order-topic/1/00000000000000000000
```

在ConsumeQueue文件中存储的是一条消息对应在CommitLog文件中的**offset偏移量**

在ConsumeQueue中存储的每条消息不只是CommitLog中的offset偏移量，还包含了消息的长度，tag hashcode，一条数据是20个字节，每个ConsumeQueue文件保存30w条数据，大概每个文件是5.72MB

近似内存的写入性能

基于OS操作系统的**PageCache**和**顺序写入**机制，来提升CommitLog的写入性能

磁盘文件顺序写 + OS PageCache写入 + OS异步刷盘策略

刷盘策略

- 同步刷盘
  生产者把消息发送给Broker，Broker必须强制将消息刷入底层的物理磁盘文件中，才会返回ack给producer
  优点：保证数据不丢失
  缺点：消息写入性能下降，写入吞吐量下降
- 异步刷盘
  生产者把消息发送给Broker，Broker将消息写入PageCache中，就直接返回ACK给producer
  优点：消息写入吞吐量非常高
  缺点：有数据丢失风险

Dledger自动选举

- Dledger它自己有一个CommitLog机制，给它数据，它会写入CommitLog磁盘文件。
- 使用Dledger来管理CommitLog，基于Dledger替换各个Broker上的CommitLog管理组件
- 基于Raft协议来进行Leader Broker选举

选举流程

- 每个broker投票给自己，然后发送自己的投票给别的broker
- 每个broker进入一个随机时间的休眠，例如，broker01休眠3秒，broker02休眠5秒，broker03休眠6秒，此时broker01先苏醒，直接会继续尝试给自己投票，并且发送自己的选票给别；接着broker02苏醒过来，发现broker01已经发送来了一个选票投给broker01自己，broker02此时还没投票，会尊重别人选择，就直接把票投给broker01了，同时把自己的选票发送给别人；
- 当有超过一半的投票投给某个broker，那么就会选举他当Leader
- 依靠随机休眠的机制，经过几轮投票后，一般都是可以快速选举出来一个Leader

数据同步

分为两个阶段，一个uncommited阶段，一个commited阶段

- Leader Broker上的Dledger收到一条数据后，会标记为uncommited状态，然后通过自己的DledgerServer组件把这个uncommited数据发送给Follow Broker的DledgerServer
- Follow Broker的DledgerServer收到uncommited消息之后，必须返回一个ack给Leader Broker的DledgerServer。
- 如果Leader Broker收到超过半数的Follow Broker返回的ack之后，就会将消息标记为commited状态
- 然后Leader Broker上的DledgerServer就会发送commited消息给Follower Broker的DledgerServer，让他们也把消息标记为commited状态。

这就是基于Raft协议实现的两阶段完成的数据同步机制。

【核心】

- 主从同步原理
- 故障手动切换缺点
- Dledger自动选举

生产者

- 怎么将消息写入MessageQueue
  每个Broker中有多个Topic
  每个Topic中由多个MessageQueue组成
  一个Topic可以配多个MessageQueue，同一个topic在多个broker上有分片，通过这种分片机制使得数据有多副本
  一个Topic有多个MessageQueue，通过均匀写入的策略将数据写入各个Broker上的MessageQueue
- Broker故障，自动容错机制
  如果有Broker发生异常，在Producer中可以配置`sendLatencyFaultEnable`开关，开启自动容错机制。如果某次访问某个Broker发现网络延迟有500ms，然后还无法访问，那么就会自动回避这个Broker一段时间。比如接下来3000ms内，就不会访问这个Broker

1. 如何拉取broker元数据
2. 怎么发送消息
3. 消息发送异常怎么处理
4. 怎么处理粘包拆包
5. 怎么接受数据，响应处理
6. 同步发送，异步发送

消息发送模式

- 同步发送：可靠性最高，吞吐量低，同步获取返回结果
  - 使用场景：对可靠性要求靠，保证消息不能丢失；消息量小
- 异步发送：性能最佳，消息存在丢失风险，通过回调函数，异步获取返回结果
  - 使用场景：消息量大，对吞吐量要求高，允许消息丢失
- 单向发送：性能佳，没有返回结果
  - 使用场景：不关注发送结果，适合发送日志消息

消费者

消费者组

```
1 // 设置消费者组
2 DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("simple
```

```
-consume-group");
```

- 作用：不同的消费者组，都会拉取到消息。

集群模式

同一消费者组，只有一台机器会获取到消息

广播模式

```
// 设置广播模式
consumer.setMessageModel(MessageModel.BROADCASTING);
```

同一消费者组，每台机器都会获取到消息

MessageQueue与消费者关系

一个Topic的多个MessageQueue会均匀分摊给消费者组内的多个机器去消费

一个MessageQueue只能被一个机器处理，但是一个机器可以处理多个MessageQueue的消息

消费模式

两者模式的本质都是消费者机器主动去broker机器拉取一批消息

pull模式

push模式

底层也是基于消费者主动拉取的模式来实现，只不过它的名字是push，broker会尽可能实时的把新消息交给消费者机器处理，他的消息时效性会更好

- 实现思路：当消费者发送请求到Broker去拉取消息，如果有新的消息可以消费那么就会立即返回一批消息到消费者机器去处理，处理完之后接着立刻发送请求到broker去拉取下一批消息
- 请求挂起和长轮询：当请求发送到broker，结果broker没有消息要发送，就会将请求线程挂起15秒；期间会有后台线程每隔一会就去检查一下是否有新的消息给你，另外如果在这个挂起过程中，如果有新的消息到达了会主动唤醒挂起的线程，然后把消息返回给你

broker如何读取消息

根据消费的MessageQueue以及开始消费的位置，去找对应ConsumeQueue读取里面对应的消息在CommitLog中的物理offset偏移量，然后到CommitLog中根据offset偏移量读取数据

如何响应消息

注册回调函数

```
consumer.registerMessageListener(new MessageListenerConcurrently()
{
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageEx
t> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) {
        System.out.printf("%s Receive New Messages: %s %n", Thread
.currentThread().getName(), list);
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
```

消息处理成功，broker会记录消费成功的位置

消费者宕机或扩容

会进入rebalance环节，重新给各个消费机器分配他们要处理的MessageQueue

从Master或Slave拉取消息的策略

- broker收到一条消息后，会先写入os cache，再写入CommitLog文件，os后台线程异步将os cache数据刷入磁盘

- 每次消费者读取数据时，先从ConsumeQueue中读取offset，再去CommitLog中读取消息
- ConsumeQueue主要存放CommitLog的offset偏移量，每个文件都很小，30W条消息的offset只有5.72MB，会被缓存在os cache，可以抗大量的读
- 从CommitLog中读消息，是从os cache中读还是从磁盘中读?
  - 两者都有。
  - 当读取的消息是刚刚写入CommitLog，那么大概率他们还停留在os cache中，此时可以从os cache中读到数据，性能很高
  - 当读取的数据是比较早之前写入CommitLog，那么数据被早早的刷入磁盘了，已经不在os cache里了，那么此时就只能从磁盘上读取文件了

- broker会对比当前你没有拉取到消息的数量和大小，以及最多可以存放在os cache内存里消息的大小来判断到底从哪去取消息。
  - 如果没拉取的消息超过了最大能使用的内存量，那么说明后续会频繁的从磁盘加载数据，此时就让你从slave去加载数据了

消费者跟多少broker建立连接

- 消费者订阅的Topic分布在几个Broker上，只跟这几个Broker建立连接

网络通信框架

> 高性能、高并发

- Reactor主线程在端口上监听Producer建立连接的请求，建立长连接
- Reactor线程池并发的监听多个连接的请求是否到达
- Worker线程池并发的对多个请求进行预处理
- 业务线程池并发的对多个请求进行读写磁盘的业务操作

mmap内存映射

> 高性能读写

概念：将磁盘文件的一些地址和用户进程私有空间的一些虚拟内存地址进行了一个映射
基于 JDK NIO包下 `MappedByteBuffer.map()` 实现mmap技术，调用map函数会执行 madvise系统调用

优点：相比传统IO，减少了一次数据拷贝

预映射机制

Broker会对磁盘上的各种CommitLog、ConsumeQueue文件预先分配好MappedFile，也就是提前对一些接下来要读写的磁盘文件，使用MappedByteBuffer执行map函数，完成映射，这样后续写文件的时

候，可以直接执行了

文件预热

在提前对一些文件完成映射之后，因为映射不会直接将数据加载到内存里来，那么后续在读取尤其是CommitLog、ConsumeQueue的时候，其他有可能会频繁的从磁盘中加载数据到内存中去。

在执行完map函数后，会执行madvise系统调用，就是提前尽可能多的把磁盘文件加载到内存里去。

通过上述优化，才真正实现一个效果，写磁盘文件的时候都是进入PageCache，保证写入高性能；
同时尽可能多的通过map + madvise的映射后预热机制，把磁盘文件里的数据尽可能多的加载到PageCache里来，后续对ConsumeQueue、CommitLog进行读取的时候，才能尽可能从内存里读取数据。

事务消息

消息发送流程

- producer发送半事务消息到broker
- 半事务消息发送成功，producer执行本地事务，根据本地事务执行结果，发送commit或rollback消息到broker
- 若发送commit消息，broker就会将此消息投递给消费者；若发送rollback消息，broker就会删除待发送的消息
- 若producer发送commit或rollback消息超时了，broker会定时回调producer，查询producer事务执行是否成功，producer重新发送 commit 或 rollback

实现原理

- 将半事务消息写入 `RMQ_SYS_TRANS_HALF_TOPIC` 这个topic对应的一个ConsumeQueue中，并不是自己指定的Topic
- 若producer由于网络故障等原因，没有收到半事务消息；RocketMQ有个定时任务，会定时扫描`RMQ_SYS_TRANS_HALF_TOPIC`中的半事务消息，如果超过一定时间还是半事务消息，那么就会回调producer接口，判断这个半事务消息是要 commit 还是 rollback
- 最多回调15次，15次回调都没有反馈，则自动将消息标记为rollback
- 怎么rollback？用一个OP操作来标记半事务消息的状态，RocketMQ内部有一个OP_TOPIC，此时可以写一个rollback OP记录到这个Topic里，标记某个半事务消息是rollback的
- 怎么commit？往OP_TOPIC里写入一条记录，标记半事务消息已经是commit状态了，接着把`RMQ_SYS_TRANS_HALF_TOPIC` 中的半事务消息写入到指定Topic的ConsumeQueue中，Consumer就可以消费了

消息零丢失

什么时候会丢消息

- 生产者丢消息：producer向broker发送消息，可能由于网络问题或broker异常，导致发送失败；如果producer对此不处理，会造成消息丢失
- broker丢消息：broker收到消息，先写入os cache，再异步的将消息刷入磁盘，在消息刷入磁盘之前，broker如果宕机，就会导致os cache的消息丢失；broker数据还未同步给从，就宕机了，导致选举出来的从缺失对应的消息
- 消费者丢消息：消费者处理消息，业务逻辑还未处理完成，就对broker进行响应，就会导致消息丢失

丢消息的解决方案

- 生产者丢失消息：可以通过事务消息的方式，保证消息不丢失；或者producer通过失败重试的机制
- broker丢失消息：broker同步刷盘，与从 broker 之间进行同步复制；
- 消费者丢消息：消费者处理消息，业务逻辑处理成功之后，响应成功ACK给broker

零丢失方案的优缺点

- 优点：可以保证从producer----》broker-----》consumer 各个环节消息不丢失
- 缺点：整个MQ的吞吐量急剧下降（写os cache 与写磁盘的性能差距大，数据还需同步复制给slave，这个过程耗时长）

零丢失方案适合场景

- 涉及金钱，交易，订单等核心数据的消息

消息重复

什么时候会重复

- 生产者端：producer向broker发送消息，broker收到了消息，但是响应超时，producer进行重发
- broker端：消费者消费消息，响应超时，broker端重发
- 消费者端：consumer消费消息，但是还没提交ack响应，就重启了；等重启之后broker就会进行重发

怎么避免重复

- 基于Redis缓存做幂等
  - 一般情况下可以，但在极端情况下，consumer消费了，但是写入redis失败了，此时broker重发，仍然会导致重复消费
- 基于业务逻辑判断

- 根据数据存储中的记录来判断这个消息是否处理过，如果处理过了，就别在处理了

Redis缓存（setnx保证数据唯一，lua释放锁） + 数据库唯一键（兜底方案）

死信队列

- 对于consumer处理异常，响应`RECONSUME_LATER`状态给broker
- broker会有一个针对ConsumeGroup的重试队列，如果consumer响应`RECONSUME_LATER`，那么就会将消息写入消费组的重试队列中`%RETRY%VoucherConsumerGroup`
- 过一段时间，重试队列会再次将消息投递给consumer，consumer可能会再次处理异常，响应`RECONSUME_LATER`，往复几次，阶梯重试，最多重试16次。
- 重试间隔：messageDelayLevel=1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
- 重试16次仍失败后，会将消息写入死信队列 `%DLQ%VoucherConsumerGroup`
- 业务侧可以订阅死信队列进行单独处理

有序消息

消息乱序问题

每个Topic指定多个MessageQueue，当我们写入消息的时候，是将消息均匀分发给不同的MessageQueue

部署多台Consumer组成一个消费者组，对于Consumer的每台机器都会负责消费一部分MessageQueue

当我们向broker按顺序发送2条消息，一条A，一条B，可能分别位于不同的MessageQueue中，Consumer端可能会先消费B，再消费A

解决方案

写入顺序：基于业务标识，比如订单id，将相同订单id的消息按顺序写入同一个MessageQueue（取模计算）

消费顺序：一个MessageQueue只能交给1个Consumer处理

重试处理：消费失败，不能响应异常给broker，只能Consumer自己重试继续处理这批消息

实现方式

- 生产者

```
1 SendResult sendResult = producer.send(message, new MessageQueueSe
  lector(){
2                                       public MessageQueue selec
  t(List<MessageQueue> mqs, Message msg, Object arg) {
```

```
                                                    // 根据订单id选择发送mes
sagequeue
                                                    Long orderId = (Long)
arg;
                                                    // 对队列数进行取模
                                                    long index = orderId
% mqs.size();
                                                    // 返回指定队列
                                                    return mqs.get((int)
index);
                                                }
                                            }, orderId);
```

- 消费者

```
consumer.registerMessageListener(new MessageListenerOrderly(){
    public ConsumeOrderlyStatus consumeMessage(List<MessageExt> m
sgs, ConsumeOrderlyContext context) {
        context.setAutoCommit(true);
        try{
            for (MessageExt msg : msgs) {
                // 对有序消息处理
            }
            return ConsumeOrderlyStatus.SUCCESS;
        } catch(Exception e) {
            // 如果消息处理有问题，返回一个状态，让它暂停一会再继续处理这批消
息
            return SUSPEND_CURRENT_QUEUE_A_MOMENT;
        }
    }
});
```

消息过滤

发送消息时，给消息设置tag和属性

```
Message msg = new Message("TopicDbData", "TableA", ("binlog").getB
ytes(RemotingHelper.DEFAULT.CHARSET));
```

```
msg.putUserProperty("a", 10);
msg.putUserProperty("b", "abc");
```

消费消息时，根据tag和属性过滤

```
// 指定tag属性，只消费这类消息
consumer.subscribe("TopicDbData", "TableA || TableB");

// 指定过滤条件
consumer.subscribe("TopicDbData", MessageSelector.bySql("a > 5 AND b = 'abc'"));;
```

延迟消息

- 生产者

```
DefaultMQProducer producer = new DefaultMQProducer("OrderSystemProducerGroup");
producer.start();

Message message = new Message("Topic", msg.getBytes());
// 设置延迟级别
message.setDelayTimeLevel(3);

producer.send(message);
```

通过`message.setDelayTimeLevel(3)`设置延迟级别，
默认支持：1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
级别3就对应10s，意思是消息发出去会过10s被消费者获取到

实战总结

- 灵活的运用tags来过滤消息
- 基于消息key来定位消息是否丢失
  - 订单场景：设置一个消息的key为订单id，消息到broker上，会基于key构建哈希索引，这个哈希索引就存放在IndexFile索引文件里
- 消息零丢失配置

    - 生产端失败重试，或通过事务消息发送
    - broker端配置数据同步刷盘，同步复制
    - 消费端配置成功消费后提交ack
    - broker集群宕机，将消息写入本地磁盘或数据库中暂存起来
- 提高消费者吞吐量
    - 部署多台消费者
    - 开启批量消费功能
- 历史消息是否要消费
    - consumer支持从哪个位置开始消费消息
        - 从Topic第一条数据开始消费（CONSUME_FROM_LAST_OFFSET）
        - 从最后一次消费过的消息之后消费（CONSUME_FROM_FIRST_OFFSET）

访问权限

消息轨迹

- 在broker的配置文件配置 `traceTopicEnable=true` ，开启消息轨迹功能
    - 开启配置后，broker自动创建一个内部Topic， `RMQ_SYS_TRACE_TOPIC` 存储所有消息追踪的数据
- producer配置消息开启轨迹追踪
  第二个参数：enableMsgTrace配置为true

```
1 DefaultMQProducer producer = new DefaultMQProducer("topic", true);
```

- consumer配置消费消息的时候开启轨迹追踪
  第二个参数：enableMsgTrace配置为true

```
1 DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("topic"
  , true);
```

三端都配置轨迹追踪功能后，消息在从 producer----->broker----->consumer 都会上报消息到RMQ_SYS_TRACE_TOPIC里去
producer上报的内容：producer的信息，发送消息的时间，消息是否发送成功，发送消息的耗时
broker上报的内容：消息存储的TOPIC，消息存储的位置，消息的key，消息的tags
consumer端上报的内容：consumer的信息，接收消息的世界，这是第几轮投递消息，消息消费是否成功，消费这条消息的耗时

线上案例实战

百万消息积压

解决方案：

- 先定位消息者问题，快速恢复
- 怎么去解决堆积消费的问题
    - 消息可丢失：直接部署消费者丢弃消息
    - 消息不可丢：
        - 消息topic对应的messagequeue较多，部署多台consumer，快速消费
        - messagequeue较少，新建一个topic调大messagequeue，将消息消费快速写入新topic中，再部署多台consumer消费新topic