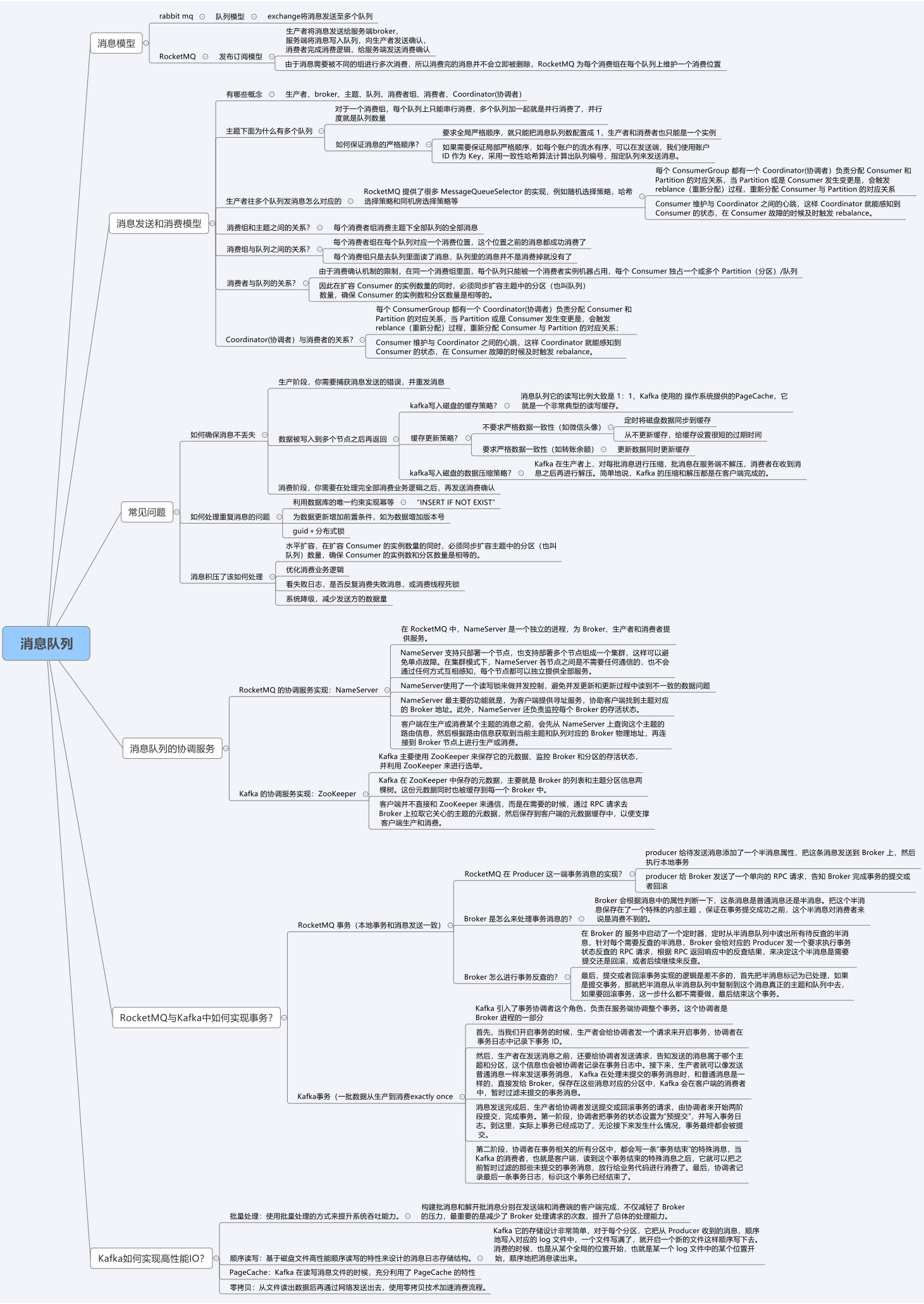
消息队列
消息模型
rabbit mq
队列模型
exchange将消息发送至多个队列
RocketMQ
发布订阅模型
生产者将消息发送给服务端broker，
服务端将消息写入队列，向生产者发送确认，
消费者完成消费逻辑，给服务端发送消费确认
由于消息需要被不同的组进行多次消费，所以消费完的消息并不会立即被删除，RocketMQ 为每个消费组在每个队列上维护一个消费位置
消息发送和消费模型
有哪些概念
生产者、broker、主题、队列、消费者组、消费者、Coordinator(协调者)
主题下面为什么有多个队列
对于一个消费组，每个队列上只能串行消费，多个队列加一起就是并行消费了，并行度就是队列数量
如何保证消息的严格顺序？
要求全局严格顺序，就只能把消息队列数配置成 1，生产者和消费者也只能是一个实例
如果需要保证局部严格顺序，如每个账户的流水有序，可以在发送端，我们使用账户 ID 作为 Key，采用一致性哈希算法计算出队列编号，指定队列来发送消息。
生产者往多个队列发消息怎么对应的
RocketMQ 提供了很多 MessageQueueSelector 的实现，例如随机选择策略，哈希选择策略和同机房选择策略等
每个 ConsumerGroup 都有一个 Coordinator(协调者）负责分配 Consumer 和 Partition 的对应关系，当 Partition 或是 Consumer 发生变更是，会触发 reblance（重新分配）过程，重新分配 Consumer 与 Partition 的对应关系
Consumer 维护与 Coordinator 之间的心跳，这样 Coordinator 就能感知到 Consumer 的状态，在 Consumer 故障的时候及时触发 rebalance。
消费组和主题之间的关系？
每个消费者组消费主题下全部队列的全部消息
消费组与队列之间的关系？
每个消费者组在每个队列对应一个消费位置，这个位置之前的消息都成功消费了
每个消费组只是去队列里面读了消息，队列里的消息并不是消费掉就没有了
消费者与队列的关系？
由于消费确认机制的限制，在同一个消费组里面，每个队列只能被一个消费者实例机器占用，每个 Consumer 独占一个或多个 Partition（分区）/队列
因此在扩容 Consumer 的实例数量的同时，必须同步扩容主题中的分区（也叫队列）数量，确保 Consumer 的实例数和分区数量是相等的。
Coordinator(协调者）与消费者的关系？
每个 ConsumerGroup 都有一个 Coordinator(协调者）负责分配 Consumer 和 Partition 的对应关系，当 Partition 或是 Consumer 发生变更是，会触发 reblance（重新分配）过程，重新分配 Consumer 与 Partition 的对应关系；
Consumer 维护与 Coordinator 之间的心跳，这样 Coordinator 就能感知到 Consumer 的状态，在 Consumer 故障的时候及时触发 rebalance。
常见问题
如何确保消息不丢失
生产阶段，你需要捕获消息发送的错误，并重发消息
数据被写入到多个节点之后再返回
kafka写入磁盘的缓存策略？
消息队列它的读写比例大致是 1：1，Kafka 使用的 操作系统提供的PageCache，它就是一个非常典型的读写缓存。
缓存更新策略？
不要求严格数据一致性（如微信头像）
定时将磁盘数据同步到缓存
从不更新缓存，给缓存设置很短的过期时间
要求严格数据一致性（如转账余额）
更新数据同时更新缓存
kafka写入磁盘的数据压缩策略？
Kafka 在生产者上，对每批消息进行压缩，批消息在服务端不解压，消费者在收到消息之后再进行解压。简单地说，Kafka 的压缩和解压都是在客户端完成的。
消费阶段，你需要在处理完全部消费业务逻辑之后，再发送消费确认
如何处理重复消息的问题
利用数据库的唯一约束实现幂等
“INSERT IF NOT EXIST”
为数据更新增加前置条件，如为数据增加版本号
guid + 分布式锁
消息积压了该如何处理
水平扩容，在扩容 Consumer 的实例数量的同时，必须同步扩容主题中的分区（也叫队列）数量，确保 Consumer 的实例数和分区数量是相等的。
优化消费业务逻辑
看失败日志，是否反复消费失败消息，或消费线程死锁
系统降级，减少发送方的数据量
消息队列的协调服务
RocketMQ 的协调服务实现：NameServer
在 RocketMQ 中，NameServer 是一个独立的进程，为 Broker、生产者和消费者提供服务。
NameServer 支持只部署一个节点，也支持部署多个节点组成一个集群，这样可以避免单点故障。在集群模式下，NameServer 各节点之间是不需要任何通信的，也不会通过任何方式互相感知，每个节点都可以独立提供全部服务。
NameServer使用了一个读写锁来做并发控制，避免并发更新和更新过程中读到不一致的数据问题
NameServer 最主要的功能就是，为客户端提供寻址服务，协助客户端找到主题对应的 Broker 地址。此外，NameServer 还负责监控每个 Broker 的存活状态。
客户端在生产或消费某个主题的消息之前，会先从 NameServer 上查询这个主题的路由信息，然后根据路由信息获取到当前主题和队列对应的 Broker 物理地址，再连接到 Broker 节点上进行生产或消费。
Kafka 的协调服务实现：ZooKeeper
Kafka 主要使用 ZooKeeper 来保存它的元数据、监控 Broker 和分区的存活状态，并利用 ZooKeeper 来进行选举。
Kafka 在 ZooKeeper 中保存的元数据，主要就是 Broker 的列表和主题分区信息两棵树。这份元数据同时也被缓存到每一个 Broker 中。
客户端并不直接和 ZooKeeper 来通信，而是在需要的时候，通过 RPC 请求去 Broker 上拉取它关心的主题的元数据，然后保存到客户端的元数据缓存中，以便支撑客户端生产和消费。
RocketMQ与Kafka中如何实现事务？
RocketMQ 事务（本地事务和消息发送一致）
RocketMQ 在 Producer 这一端事务消息的实现？
producer 给待发送消息添加了一个半消息属性，把这条消息发送到 Broker 上，然后执行本地事务
producer 给 Broker 发送了一个单向的 RPC 请求，告知 Broker 完成事务的提交或者回滚
Broker 是怎么来处理事务消息的？
Broker 会根据消息中的属性判断一下，这条消息是普通消息还是半消息。把这个半消息保存在了一个特殊的内部主题，保证在事务提交成功之前，这个半消息对消费者来说是消费不到的。
Broker 怎么进行事务反查的？
在 Broker 的 服务中启动了一个定时器，定时从半消息队列中读出所有待反查的半消息，针对每个需要反查的半消息，Broker 会给对应的 Producer 发一个要求执行事务状态反查的 RPC 请求，根据 RPC 返回响应中的反查结果，来决定这个半消息是需要提交还是回滚，或者后续继续来反查。
最后，提交或者回滚事务实现的逻辑是差不多的，首先把半消息标记为已处理，如果是提交事务，那就把半消息从半消息队列中复制到这个消息真正的主题和队列中去，如果要回滚事务，这一步什么都不需要做，最后结束这个事务。
Kafka事务（一批数据从生产到消费exactly once
Kafka 引入了事务协调者这个角色，负责在服务端协调整个事务。这个协调者是 Broker 进程的一部分
首先，当我们开启事务的时候，生产者会给协调者发一个请求来开启事务，协调者在事务日志中记录下事务 ID。
然后，生产者在发送消息之前，还要给协调者发送请求，告知发送的消息属于哪个主题和分区，这个信息也会被协调者记录在事务日志中。接下来，生产者就可以像发送普通消息一样来发送事务消息，Kafka 在处理未提交的事务消息时，和普通消息是一样的，直接发给 Broker，保存在这些消息对应的分区中，Kafka 会在客户端的消费者中，暂时过滤未提交的事务消息。
消息发送完成后，生产者给协调者发送提交或回滚事务的请求，由协调者来开始两阶段提交，完成事务。第一阶段，协调者把事务的状态设置为“预提交”，并写入事务日志。到这里，实际上事务已经成功了，无论接下来发生什么情况，事务最终都会被提交。
第二阶段，协调者在事务相关的所有分区中，都会写一条“事务结束”的特殊消息，当 Kafka 的消费者，也就是客户端，读到这个事务结束的特殊消息之后，它就可以把之前暂时过滤的那些未提交的事务消息，放行给业务代码进行消费了。最后，协调者记录最后一条事务日志，标识这个事务已经结束了。
Kafka如何实现高性能IO?
批量处理：使用批量处理的方式来提升系统吞吐能力。
构建批消息和解开批消息分别在发送端和消费端的客户端完成，不仅减轻了 Broker 的压力，最重要的是减少了 Broker 处理请求的次数，提升了总体的处理能力。
顺序读写：基于磁盘文件高性能顺序读写的特性来设计的消息日志存储结构。
Kafka 它的存储设计非常简单，对于每个分区，它把从 Producer 收到的消息，顺序地写入对应的 log 文件中，一个文件写满了，就开启一个新的文件这样顺序写下去。消费的时候，也是从某个全局的位置开始，也就是某一个 log 文件中的某个位置开始，顺序地把消息读出来。
PageCache：Kafka 在读写消息文件的时候，充分利用了 PageCache 的特性
零拷贝：从文件读出数据后再通过网络发送出去，使用零拷贝技术加速消费流程。