消息队列

- 应用场景
  - 异步任务
  - 削峰填谷
  - 应用解耦
- 开源产品
  - RabbitMQ
  - RocketMQ：低延迟、稳定
  - Kafka：大数据、流计算
  - Pulsar：新兴产品，多租户
- 选择标准
  - 开源
  - 流行
  - 编程语言
  - 可靠、集群、性能
  - 业务场景
- 消息模型
  - 队列模型
    - 定义 — 消息只能被一个消费者消费，不允许重复消费
    - 组成
      - 生产者（Producer）
      - 消费者（Consumer）
      - 队列（Queue）
      - 交换器（Exchange）（RabbitMQ）
  - 发布订阅模型
    - 概念 — 消息可以被多个消费者消费，即允许重复消费
    - 组成
      - 发布者（Publisher）
      - 订阅者（Subscriber）
      - 主题（Topic）
      - 分区（Partition）（RocketMQ、Kafka）
      - 消费者组（Consumer Group）（RocketMQ、Kafka）
- 事务消息
  - Kafka 事务 — 确保在"读数据-计算-保存结果"过程中数据不重复不丢失
  - RocketMQ 事务 — 确保执行本地事务和发消息或者都成功，或者都失败
  - 分布式事务
    - 1.开启事务
    - 2.发送半消息
    - 3.执行本地事务
    - 4.提交或回滚
    - 5.投递消息
    - 4-5.事务反查（RocketMQ）
- 消息积压处理
  - 优化性能 — 保障消费端的消费性能高于生产端的发送性能
    - 优化消息消费逻辑
    - 增加消费者实例数量（同步扩容分区数）
  - 服务降级 — 关闭不重要的业务，以减少发送方发送的数据量
  - 异常监控 — 检查是否有大量的消费错误，或触发了死锁，又或卡在等待某些资源上
- 消息顺序消费
  - RabbitMQ
    - 手动确认消息接收
    - 一个 Queue，一个 Consumer，Consumer 内部单线程消费
    - 基于统一规则（对唯一标识进行 Hash），将单个 Queue 拆分成多个 Queue
  - Kafka
    - 手动提交消费者位移
    - 一个 Topic，一个 Partion，一个 Consumer，Consumer 内部单线程消费
    - 通过指定消息的 key，从而将 key 值相同的消息都路由到同一个分区
- 重复消息处理
  - 消息交可靠性保障
    - At Most Once：消息最多会被送达一次
    - At Least Once：消息至少会被送达一次
    - Exactly Once：消息只会被送达一次，不允许丢失也不允许重复
  - 幂等操作
    - 利用数据库的唯一约束实现幂等
    - 利用 Redis SETNX 命令实现幂等
    - 为更新的数据设置前提条件（乐观锁）
    - 记录并检查操作（Token 机制或 GUID 机制）
  - Kafka 特殊处理
    - 手动提交位移，以避免重平衡时重复消费
    - 合理设置 max.poll.interval.ms 与 max.poll.records，以避免两次 poll 时间间隔超时导致重平衡
- 无消息丢失
  - 检测方法
    - 原理 — 利用消息队列的有序性
    - 实现 — 在 Producer 端给消息附加递增序号（由拦截器生成），在 Consumer 端检查序号的连续性
    - 分区检测 — 对于 Kafka 和 RocketMQ，需在发消息时指定分区，在分区上检测序号的有序性
  - 防范措施
    - RabbitMQ
      - 持久化（默认）
        - 消息持久化
        - 队列持久化
        - 交换器持久化
      - 消息发送确认
        - 是否到达交换器：ConfirmCallback
        - 是否到达队列：ReturnCallback
      - 消息接收确认
        - 手动确认
          - 肯定确认：basicAck
          - 否定确认
            - 否定消息：basicNack
            - 拒绝消息：basicReject（死信队列）
    - Kafka
      - 生产阶段
        - 同步发送：捕捉异常
异步发送：使用带有回调通知的发送 API
        - 自动重试消息发送（retries > 0）
        - 所有副本都接收到消息才算"已提交"（acks = all）
      - 存储阶段
        - 每个主题保存多个副本，副本位于集群中不同的 Broker 上（replication.factor >= 3）
        - 控制消息至少要被写入到多少个副本才算是"已提交"，Kafka 只对"已提交"的消息做有限度的持久化保证（min.insync.replicas > 1, < replication.factor）
        - 避免某个 Broker 落后 原先的 Leader 太多时被选为新的 Leader（unclean.leader.election.enable=false）
        - 单节点 Broker，配置同步刷盘（效率很低）
收到消息后，将消息写入磁盘后再给 Producer 返回确认响应
      - 消费阶段
        - 关闭自动提交位移，改为手动提交位移（enable-auto-commit=false）