Kafka 串讲

MLE 算法指北

2025 年 3 月 11 日

1 太长不看版–kafka 常见面试题

1. Kafka 为什么使用 Zookeeper？

- 维护 Kafka 元数据，如 **Broker 列表、Partition 领导者信息 **。
- 负责 **Leader 选举 **，确保分区主副本的可用性。
- 管理 **Consumer Group 的 Rebalance**，当 Consumer 组成员变更时，触发分区重新分配。

2. Kafka 如何进行 Leader 选举？

- 每个 Partition 具有多个副本（Replica），包括一个 **Leader** 和若干 **Follower**。
- **Zookeeper 负责选举 Leader**，如果当前 Leader 崩溃，会从 ISR（In-Sync Replicas）中选出新的 Leader。
- 如果 ISR 为空，则可能会从 OSR（Out-Sync Replicas）中选 Leader，但会导致数据丢失。

3. Kafka 如何存储数据？

- Kafka 使用 ** 日志文件（Log Segments）** 存储消息，每个 Partition 由多个日志段组成。
- Kafka 采用 **PageCache 进行文件缓存 **，减少磁盘 I/O，提高读取效率。
- 旧数据由 **Log Retention（日志保留策略）** 控制，可基于时间（log.retention.hours）或大小（log.retention.bytes）。

4. Kafka 如何保证数据一致性？

- Kafka 采用 ** 副本同步机制（ISR）** 确保数据一致性，所有 ISR 副本确认写入的数据才会更新 High Watermark（HW）。
- Consumer 只能消费 **HW 之前的消息 **，保证所有 Consumer 读取到的数据一致。
- 采用 **Leader-Follower 机制 **，Follower 只从 Leader 读取数据，确保数据的一致性。

5. Kafka 如何处理分区副本同步？

- Follower ** 异步拉取 Leader 的数据 **，以保持同步。
- 如果 Follower ** 落后时间过长 **（超过 ‘replica.lag.time.max.ms‘），Kafka 会将其从 ISR 移除。
- Follower 重新同步时，会 ** 截断高于 HW 的日志 **，然后从 HW 开始同步，保证数据一致性。

6. Kafka 事务（Transaction）如何实现？

- Kafka ** 引入 Producer ID（PID）**，确保事务范围内的所有消息要么全部成功提交，要么全部回滚。
- Kafka 事务支持 ** 多个分区的原子性提交 **，但不支持跨 Topic 或外部系统的事务。

7. Kafka 如何避免消息重复消费？

- ** 幂等 Producer**:
  - 通过 ‘enable.idempotence=true‘，确保相同的消息不会重复写入。
- **Exactly Once 语义（EOS）**:
  - 通过 Kafka 事务 API 结合 ‘acks=all‘ 和 ‘isolation.level=readcommitted‘，避免重复消费。

- **Consumer 手动提交 Offset**:
  - 采用 ‘enable.auto.commit=false‘，确保消息处理完成后再提交 Offset，防止重复消费。

8. 如何监控 Kafka 运行状态？

- Kafka 提供 JMX（Java Management Extensions）指标:
  - ** 消息堆积情况 **: ‘kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions‘
  - **Consumer Lag**: 监控 Consumer 消费的 Offset 与 Log End Offset（LEO）之间的差距
  - **Broker 运行状态 **: ‘kafka.server:type=KafkaServer,name=BrokerState‘
- 使用 **Prometheus + Grafana** 进行可视化监控。
- 使用 **Burrow** 监控 Consumer 偏移量，防止 Consumer Lag 过大。

9. Kafka Consumer Lag 过大的原因及优化方案？

- **原因**:
  - Consumer 消费速度慢，处理不过来。
  - Consumer 线程数量太少，导致吞吐量不足。
  - 发生 Rebalance，导致 Consumer 重新分配 Partition，消费滞后。
- **优化方案**:
  - 增加 Consumer 线程数，提高消费能力。
  - 调整 ‘max.poll.records‘，增大单次拉取的消息量，提高吞吐量。
  - 使用 ** 手动提交 Offset**，确保消息消费完成后再提交。

10. Kafka 发生 Rebalance 的常见原因？如何优化？

- **Rebalance 发生的原因**:
  - Consumer 组成员增加或减少。
  - 订阅的 Topic 发生变化。
  - Consumer 长时间未发送心跳，Kafka 认为其失联，触发 Rebalance。
- **优化方案**:
  - ** 使用 Sticky Assignor** 使 Kafka 在 Rebalance 时尽量保持原有分配，减少数据迁移。
  - ** 设置 ‘session.timeout.ms‘ 和 ‘heartbeat.interval.ms‘**，调整 Consumer 失联判定时间。
  - ** 采用 Incremental Cooperative Rebalance**，逐步 Rebalance，避免影响整个 Consumer 组。

2 Kafka 概述

Kafka 最初由 LinkedIn 开发，是一个 **分布式、分区的、多副本的、基于订阅模式**的消息中间件，主要用于高吞吐的日志收集、消息队列和流式处理。

Kafka 主要应用于以下场景:

- **日志收集系统**:
  - 大型网站会产生大量日志数据，Kafka 可作为日志管道，将日志存储到 Hadoop 或其他存储系统。
- **实时数据流处理**:
  - 结合 Flink、Spark Streaming 等流式计算框架，实现实时 ETL 和数据分析。
- **消息队列（MQ）**:
  - 替代传统 MQ（如 RabbitMQ、ActiveMQ），用于异步处理和解耦服务。
- **事件驱动架构**:
  - 适用于微服务架构，Kafka 用于事件驱动通信，支持服务间松耦合。

Kafka 主要设计目标

- **高吞吐**: 支持 ** 单机每秒 100K+ 消息传输 **。
- **持久化存储**: 基于磁盘顺序写，**O(1) 复杂度 ** 确保高效存储。
- **分布式扩展**: 支持多 Broker，** 水平扩展 ** 容易。
- **数据顺序性**: 保证 ** 同一分区（Partition）内的消息严格有序 **。
- **支持批量处理**和 ** 流式计算 **。

2.1 Kafka 架构概述

Kafka 是一个高吞吐、可扩展的分布式消息系统，主要用于日志收集、流式计算和事件驱动架构。Kafka 主要由以下核心组件构成：

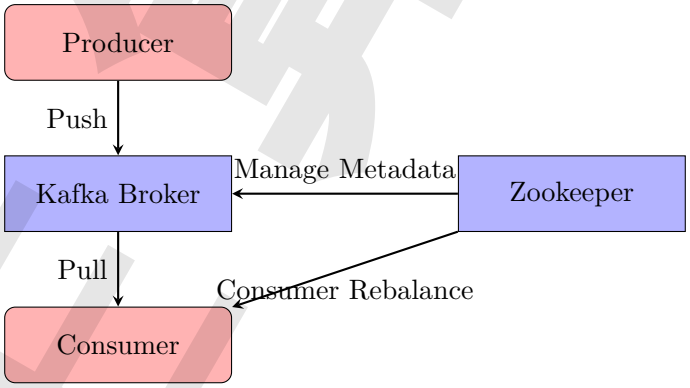
- **Producer（生产者）**：负责向 Kafka 发送消息。
- **Broker（代理）**：Kafka 服务器实例，存储和转发消息。
- **Topic（主题）**：组织消息的方式，Producer 向特定 Topic 发送消息。
- **Partition（分区）**：Topic 可拆分成多个 Partition，提高并行处理能力。
- **Consumer（消费者）**：从 Kafka 订阅 Topic 并消费数据。
- **Consumer Group（消费者组）**：多个消费者组成的组，共同消费一个 Topic。
- **Zookeeper**：用于管理 Kafka 集群，负责 Leader 选举、元数据存储和 Consumer 组协调。

2.2 Kafka 消息流动过程

Kafka 的消息流动大致分为以下几个步骤：

1. **Producer 推送消息**：
   - 生产者（Producer）使用 **push** 模式将消息发送到 Kafka 集群。
   - 每条消息被发送到特定的 **Topic**，并存储在该 Topic 的某个 **Partition** 中。
2. **Broker 存储与管理消息**：
   - Broker 负责接收 Producer 发送的消息，并存储到合适的 Partition。
   - Kafka 通过 **Zookeeper** 维护 Broker 列表，并负责选举 Partition 的 Leader。
3. **Consumer 拉取消息**：
   - 消费者（Consumer）使用 **pull** 模式从 Broker 获取消息。
   - 同一个 Consumer Group 内的多个 Consumer 共同消费不同 Partition。
4. **Zookeeper 负责协调集群**：
   - 维护 Kafka 集群元数据，如 Broker 信息和 Partition 位置。
   - 选举新的 Partition Leader，在 Consumer 组发生变化时触发 Rebalance。

2.3 Kafka 架构示意图



2.4 Kafka 关键特性

Kafka 具有以下关键特性：

- **高吞吐**：Kafka 采用 **顺序写**和 **分区并行处理**，大幅提升吞吐量。
- **可扩展性**：支持 **水平扩展**，可增加 Broker 提高存储和处理能力。
- **容错性**：Kafka 通过 **多副本机制（Replication）**确保数据不丢失，即使某个 Broker 发生故障，Follower 也可成为新的 Leader。
- **持久化存储**：Kafka 允许 **持久化数据**，支持 **回溯消费**。
- **可靠的消费者模型**：
  - Consumer Group 机制确保多个消费者可以并行消费，提高吞吐量。
  - Offset 机制允许消费者随时回溯消息。

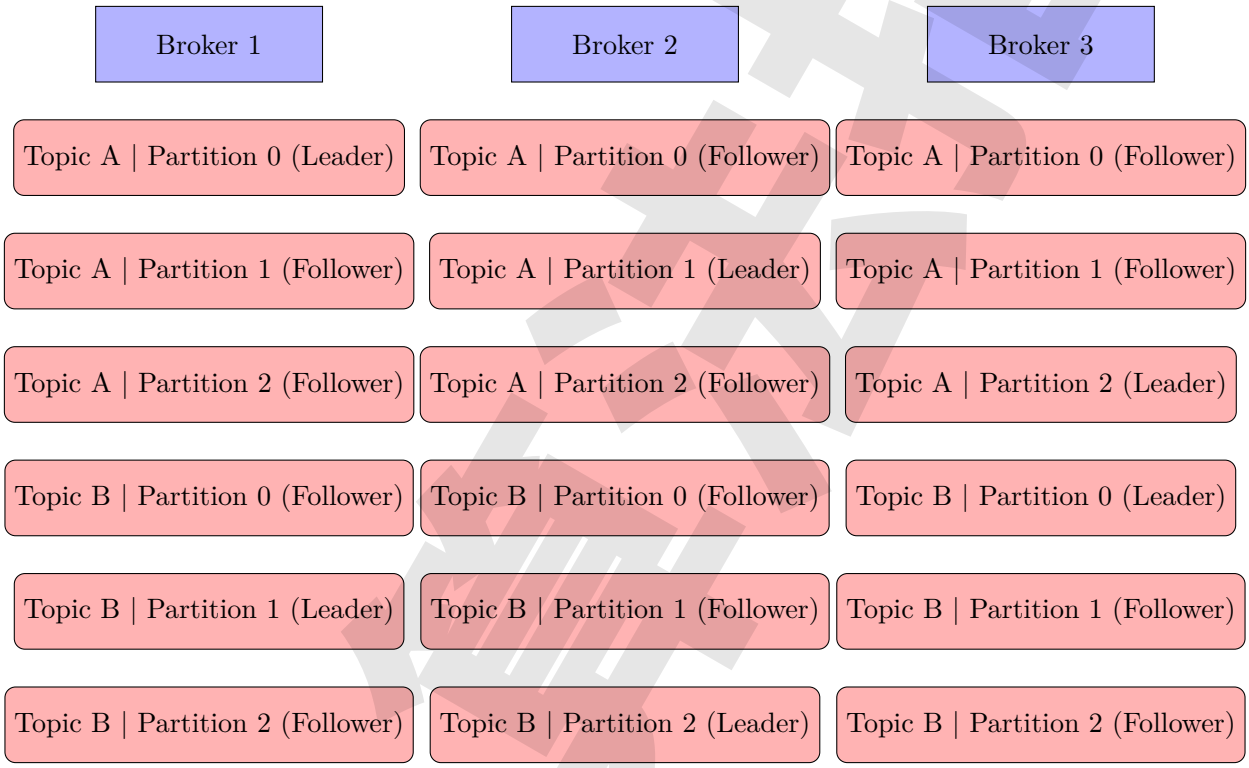
3 Kafka 文件管理

Kafka 以 **Topic + 分区（Partition）** 进行存储管理，每个 **Topic** 由多个 **Partition** 组成，每个分区对应一个单独的日志文件。同时，Kafka 采用 ** 多副本机制（Replication）**，确保数据的高可用性和容错能力。每个分区都会有一个 **Leader** 和若干个 **Follower**，Kafka 内部机制保证：

- **Leader 负责处理所有的读写请求 **，确保高效的数据访问。
- **Follower 负责同步 Leader 数据 **，以保证数据冗余，提高可靠性。
- **Kafka 通过 Broker 平衡 Leader 角色的分配 **，确保分区的 Leader 在 Broker 之间均匀分布。

3.1 Kafka 文件布局结构

Kafka 的存储结构主要由 **Broker** 负责管理，每个 **Broker** 维护多个 **Topic** 的多个 **Partition**，其组织方式如下：



3.2 Kafka 分区管理

Kafka 的数据组织方式基于 **Partition**，每个 **Topic** 由多个 **Partition** 组成：

- 每个 **Partition** 存储在不同的 **Broker** 上，以提高数据访问速度和分布式存储能力。
- **Partition 数据是不可变的追加日志 **，新消息只会追加到日志末尾。
- Kafka 通过 **Offset** 机制保证消息的顺序性和可追溯性。

每个 **Partition** 具体存储如下：

- **索引文件（.index）**：记录消息的偏移量（Offset）。
- **日志文件（.log）**：存储具体的消息数据。
- **时间索引文件（.timeindex）**：用于高效的时间范围查询。

3.3 Kafka 分区策略

Kafka 提供三种常见的分区策略：

- **轮询策略（Round-Robin）**：适用于无 Key 的情况，Producer 轮流发送到不同分区。
- **Key 哈希策略**：计算 hash(key) mod partition_count，确保相同 Key 的消息进入同一分区。
- **自定义分区策略**：用户可以实现 `Partitioner` 接口，自定义分区逻辑。

3.4 Kafka 负载均衡

Kafka 通过 **Broker 之间均衡 Leader 角色 ** 来实现负载均衡：

- Kafka ** 自动分配 Partition Leader**，确保 Broker 负载均衡。
- 如果某个 Broker 宕机，Kafka 会 ** 自动将 Leader 角色转移 ** 到其他 Broker。
- 通过 **Rebalance 机制 ** 确保新的 Leader 选举，避免单点故障。

4 Kafka 消息发送机制

Kafka 在消息发送客户端引入了 ** 批处理思想 **，以提高消息传输效率和吞吐量。其核心机制如下：

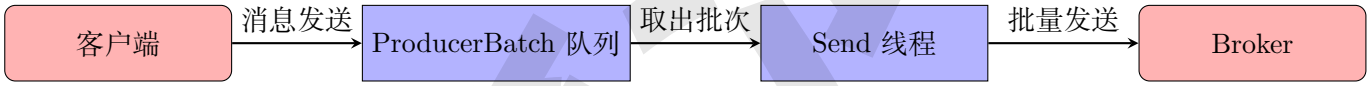
4.1 批量发送机制

Kafka 生产者（Producer）在发送消息时，消息不会直接发送到 **Broker**，而是 ** 先存入本地的一个双端队列 **，然后再进行批量发送。该过程如下：

- 每个 **ProducerBatch** 表示一次批量发送，存储在双端队列中。
- **batch.size** 参数控制每批最大消息大小（默认 16KB）。
- **Send 线程 ** 负责从队列中取出批次，并将其发送到 Kafka Broker。
- **linger.ms** 参数控制 ** 发送延迟 **，用于等待更多消息加入批次，以提高吞吐量。

4.2 Kafka 消息批量发送流程

Kafka 批量发送消息的工作流程如下：



4.3 batch.size 与 linger.ms 参数调优

Kafka 允许用户通过 **batch.size** 和 **linger.ms** 参数调整批量发送策略：

- **batch.size**：设置 ProducerBatch 的最大大小（默认 16KB）。
- **linger.ms**：控制消息在缓冲区的等待时间，提高批量效果。

优化策略：

1. 增加 **batch.size** 可以减少 Producer 发送请求的次数，提升吞吐量。
2. 设置较小的 **linger.ms** 可以减少延迟，但可能降低吞吐量。
3. 如果消息发送频率较低，适当提高 **linger.ms**，可以提高批量发送的效率。

4.4 Kafka 批量发送的优势

Kafka 采用批量发送机制，相较于单条消息发送，具备以下优势：

- ** 提高吞吐量 **：减少单个消息发送的开销，提高 Broker 端的处理能力。
- ** 降低网络开销 **：批量传输减少了网络请求次数，提高传输效率。
- ** 优化 CPU 资源 **：批量数据处理能更好地利用 Broker 端的 CPU 资源，减少频繁的 I/O 操作。

4.5 Kafka 与 RocketMQ 批量发送的对比

Kafka 的批量发送与 RocketMQ 相比，具有不同的特点：

- Kafka 的 **ProducerBatch** 采用 ** 网络协议级 ** 的数据压缩，提高传输效率。
- RocketMQ 允许更细粒度的批量控制，但 Kafka 的批处理机制 ** 更适合高吞吐场景 **。
- Kafka 通过 **linger.ms** 控制批量发送时机，在吞吐和延迟之间提供更好的权衡。

5 Kafka 副本机制

Kafka 通过 ** 副本机制（Replication）** 确保数据的高可用性和容错能力。每个 **Partition** 由多个副本（Replica）组成，副本机制主要包括以下三个核心概念：

- **Leader**：负责处理该 Partition 的所有读写请求。
- **Follower**：从 Leader 复制数据，确保副本同步，并在 Leader 失效时参与选举。
- **ISR（In-Sync Replicas）**：同步副本集合，包含 Leader 和所有与 Leader 保持同步的 Follower。

5.1 副本角色分类

Kafka 副本机制涉及以下几种角色：

1. **AR（Assigned Replicas）**：一个 Partition 的所有副本集合（不区分 Leader 或 Follower）。
2. **ISR（In-Sync Replicas）**：能和 Leader 保持同步的 Follower 集合 + Leader 本身。
3. **OSR（Out-Sync Replicas）**：不能与 Leader 同步的 Follower 集合（即滞后的副本）。
4. **AR = ISR + OSR**：即 Assigned Replicas 由同步副本和滞后副本组成。

5.2 ISR 机制

Kafka 采用 **ISR 机制 ** 保证数据一致性：

- Leader 维护一个 ** 动态的 ISR 集合 **，其中包含所有同步副本的 Follower。
- 当 Follower ** 长时间未同步 ** 数据时，会被移出 ISR，防止其影响 Leader 确认消息的能力。
- Leader 发生故障时，Kafka ** 从 ISR 中选举新的 Leader**，确保系统快速恢复。
- 被移出 ISR 的 Follower 需要同步到 Partition 最新的 High Watermark（HW）位置，才能重新加入 ISR。

5.3 ack 机制与副本同步

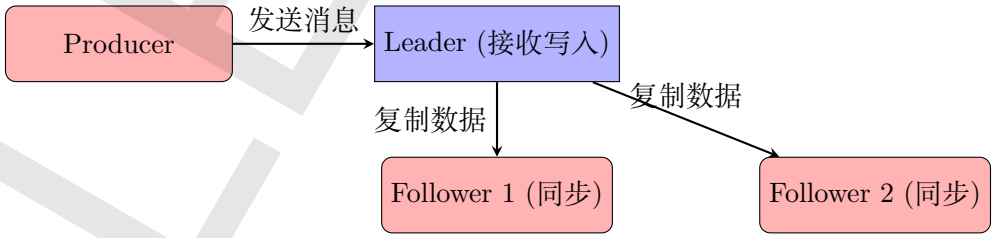
Kafka 提供三种 **ack 配置 **，允许在 ** 可靠性 ** 和 ** 性能 ** 之间进行权衡：

- **acks=0**：Producer 不等待 Broker 确认 ack，即”fire-and-forget” 模式，可能丢失数据，但性能最佳。
- **acks=1**：Producer 仅等待 Leader 写入成功后返回 ack，Follower 可能不同步，存在 Leader 故障时的数据丢失风险。
- **acks=-1 (all)**：Producer 需要等待 ISR 集合中的所有 Follower 复制完成后才返回 ack，确保最高可靠性，但增加延迟。

5.4 副本同步过程

Kafka 副本同步采用 ** 拉取模式（Pull-based）**：

1. Producer 向 **Leader** 发送消息，Leader 写入日志后返回 ack。
2. ISR 中的 **Follower** 定期从 Leader 拉取消息，并写入本地日志。
3. 只有 ** 所有 ISR 副本同步成功后 **，Kafka 才会更新该 Partition 的 **High Watermark (HW)**，表示消息已安全持久化。



5.5 Leader 失效处理

当 Leader 失效时：

1. Kafka 从 **ISR 副本中 ** 选举新的 Leader（保证数据一致性）。
2. 若 ISR 为空，则从 **OSR 副本 ** 中选举 Leader，可能会丢失部分数据。
3. 选出的 Leader 开始对外提供读写服务，Follower 重新同步数据。

5.6 Kafka 副本机制的优势

Kafka 采用的 **ISR 机制 + 多副本架构 ** 具有以下优势：

- ** 高可用性 **：Leader 失效时，可以快速从 ISR 选出新 Leader，保证服务不中断。
- ** 数据可靠性 **：通过 ‘acks=-1‘ 机制，确保所有 ISR 副本同步成功，避免数据丢失。
- ** 负载均衡 **：Kafka 自动分配 **Partition Leader**，均衡 Broker 负载，提升吞吐量。

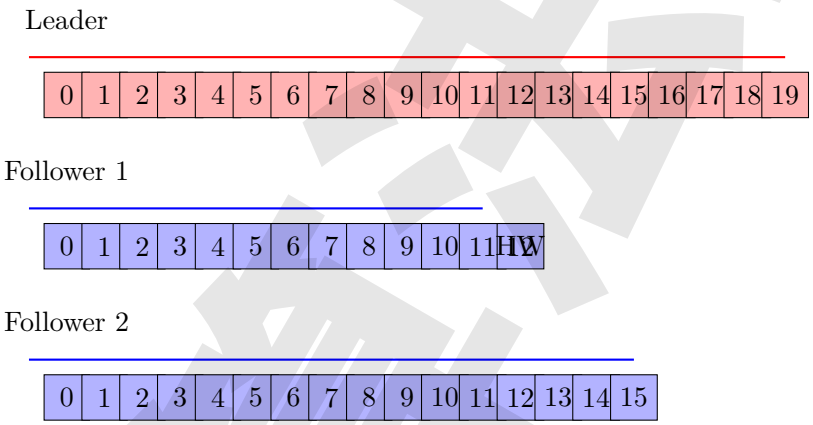
6 Kafka 故障处理机制

Kafka 采用多副本机制来保证高可用性和数据一致性。当 **Leader 或 Follower 发生故障 ** 时，Kafka 需要执行相应的恢复操作，以确保系统的稳定性。

6.1 LEO 和 HW 概念

Kafka 维护两个重要的日志偏移量：

- **LEO（Log End Offset）**：每个副本的日志的最后一个 offset，表示该副本当前日志的最大偏移量。
- **HW（High Watermark）**：ISR 副本集合中最小的 LEO，表示所有副本都同步的最大偏移量，Consumer 只能读取 HW 之前的数据。



说明：

- Leader 的 **LEO（19）** 高于 **HW（12）**。
- Follower 1 同步较慢，LEO = 12，因此 HW = 12。
- Follower 2 LEO = 15，高于 HW，但 Kafka 仅允许 Consumer 读取 HW 之前的数据，以保证一致性。

6.2 Follower 故障处理

当 **Follower 发生故障 **（如网络异常、磁盘故障）时：

1. 该 Follower 会 ** 临时从 ISR（In-Sync Replicas）中移除 **，避免影响 Kafka 的消息确认机制。
2. Follower 恢复后，首先会 ** 读取本地磁盘 ** 记录的上次 HW，并 ** 截断 ** 高于 HW 的日志数据，以确保数据一致性。
3. 然后，Follower 从 HW 位置开始重新向 Leader ** 同步日志 **，直到 LEO 追上 Leader，才能重新加入 ISR。

6.3 Leader 故障处理

当 **Leader 发生故障 **（如 Broker 宕机、网络断开）时：

1. Kafka 从 **ISR 副本集合 ** 中选举新的 Leader，保证数据一致性。
2. 被选中的新 Leader 会 ** 截断高于 HW 的数据 **，避免不一致。
3. 其他 Follower 开始从新 Leader 进行日志同步，确保系统正常运行。

6.4 故障处理机制的优势

Kafka 采用 **LEO + HW 机制 ** 确保数据一致性：

- ** 防止数据丢失 **：Leader 仅确认 HW 之前的数据，保证所有已提交的消息不会丢失。
- ** 保证数据一致性 **：Follower 恢复后，必须截断超出 HW 的日志，避免出现不一致数据。
- ** 高可用性 **：Kafka 通过 ISR 选举新 Leader，确保服务不中断。

7 Kafka 支持 Exactly Once

Kafka 默认支持 ** 最少一次（At-Least-Once）** 语义，即消息可能会被重复发送。在 Kafka 0.11.0 版本后，引入了 ** 幂等性（Idempotence）** 机制，使得 Producer 可以保证 **Exactly Once** 语义，即每条消息只会被处理一次。

7.1 幂等性（Idempotence）机制

Kafka 允许 Producer 通过以下方式启用幂等性：

```
props.put("enable.idempotence", true)
```

当幂等性启用时：

- 生产者的 **acks** 机制会自动设为 **"all"**，确保所有 ISR 副本都同步后才确认消息。
- 生产者每次初始化时，Kafka ** 分配一个 Producer ID（PID）** 和 **Sequence Number**，用于标识唯一的 Producer 实例。
- 对于相同的 **PID**，每个 **Topic-Partition** 维护一个单调递增的 Sequence Number。

7.2 Broker 端序列号校验

Kafka **Broker 端 ** 也会维护 '<PID, Topic, Partition>' 的 ** 序号信息 **, 并对每条消息的 **Sequence Number** 进行检查：

- 如果 **Producer 发送的序号 == Broker 维护的最新序号 + 1**，则消息按顺序写入 Kafka。
- 如果 **Producer 发送的序号 > Broker 维护的序号 + 1**，说明中间存在数据丢失，Broker ** 拒绝该消息 **，Producer 触发 'InvalidSequenceNumber' 异常。
- 如果 **Producer 发送的序号 < Broker 维护的序号 **，说明是 ** 重复消息 **，Broker 直接 ** 丢弃该消息 **，Producer 触发 'DuplicateSequenceNumber' 异常。

7.3 Exactly Once 语义的适用场景

Exactly Once 语义适用于以下场景：

- ** 需要保证 Producer 仅做单个分区的幂等性 **，即单分区内不会出现重复消息，但 ** 多个分区无法保证顺序性 **。
- ** 跨会话的事务处理 **: 默认情况下, Kafka ** 无法保证跨会话的 Exactly Once**, 即如果 Producer 宕机重启，则无法保证事务的连续性。
- ** 流式计算和事务提交 **：需要 Kafka Streams 或事务 API 支持，以确保消息在多个系统之间一致传输。

7.4 Exactly Once 语义的实现限制

尽管 Kafka 提供了 **Exactly Once** 机制，但仍然有以下局限：

- ** 仅支持 Kafka 内部的 Exactly Once**，如果数据需要同步到外部存储（如数据库、HDFS），则需要事务配合。
- ** 在多分区情况下，无法保证跨分区的全局有序性 **。
- ** 依赖事务机制（Kafka Transaction API）**，如果 Producer 发生故障，事务可能会导致消息回滚，影响吞吐量。

8 Kafka、RocketMQ 和 RabbitMQ 对比分析

8.1 高可用性

- **RocketMQ**:
  - 支持多个 **nameserver**，无状态存储，Broker 需要主动注册。
  - 4.5 版本之前采用 **master/slave** 结构，固定主从角色。
  - 4.5 版本后支持 **Dledger 组件 **，可实现 Raft 选举，增强可用性。
- **Kafka**:
  - 依赖 **Zookeeper** 记录元数据。
  - 采用 **broker 1 主多从 **，主从切换依赖 **Zookeeper 选举 **，支持从节点自动切换。
- **RabbitMQ**:
  - 采用 ** 主从方式 **，可配置 ** 镜像队列 ** 以提高可靠性。

8.2 架构与存储方式

特性	RocketMQ	Kafka	RabbitMQ
存储方式	文件存储	文件存储	内存 + 持久化可选
消息读取模式	集群模式	集群模式	直接存储队列
副本机制	同步/异步	同步/异步	镜像存储

表 1: RocketMQ、Kafka 和 RabbitMQ 存储方式对比

8.3 性能对比

特性	RocketMQ	Kafka	RabbitMQ
QPS	十万级	数千万级	万级
延迟	毫秒级	毫秒级	微秒级
多 Topic 影响	影响不明显	影响明显	影响不明显

表 2: RocketMQ、Kafka 和 RabbitMQ 性能对比

8.4 扩展性

- **RocketMQ**:
  - ** 支持 Broker 直接扩展 **，动态注册到 nameserver。
  - 负载均衡机制采用 **DLedger 或自定义策略 **。
- **Kafka**:
  - ** 支持水平扩展 **，Zookeeper 负责动态管理 Broker 信息。
  - ** 分区模式 **，可均衡分配 Leader 角色。
- **RabbitMQ**:
  - 依赖 ** 外部负载均衡（如 HAProxy, LVS）**，水平扩展能力较弱。

8.5 消息投递可靠性

投递语义	RocketMQ	Kafka	RabbitMQ
At least once	支持	支持	支持
At most once	支持	支持	支持
Exactly once	不支持	支持	不支持

表 3: RocketMQ、Kafka 和 RabbitMQ 消息投递语义对比