消息队列

1. 节点控制器（broker controller）
2. 分区副本选主（partition leader）
3. 消费组协调器选主（consumer group LeaderCoordinator）

1. 技术选型

1）ActiveMQ

- 优点：老牌的消息中间件，过去很多国内的公司运用广泛，功能强大；
- 缺点：没法确认 ActiveMQ 可以支撑互联网公司的高并发、高负载以及高吞吐的复杂场景，国内互联网公司落地较少。而且多是传统企业在使用，ActiveMQ 一般作为异步调用和系统解耦的中间件。

2）RocketMQ

- 优点：阿里开源的 MQ 框架，基于 Java 语言写的。经历了阿里生产环境的高并发、高吞吐考验，性能卓越，还支持分布式事务等场景；而且 Java 源码也容易看懂，后面进行二次开发和改造可能比较容易。
- 缺点：虽然 RocketMQ 已经在前几年就捐给了 Apache，但 GitHub 上的活跃度其实不算高（已经很高了），而且还可能考虑到社区突然黄掉的风险。所以，除非对自己公司的技术实力非常自信，否则还是推荐 RabbitMQ。

3）RabbitMQ

- 优点：RabbitMQ 不仅可以支撑高并发、高吞吐和消息高可靠的业务场景，而且还有着非常完善便捷的后台管理页面。另外，它还支持集群化、高可用的部署架构，功能较为强大。最主要的是，RabbitMQ 的开源社区非常活跃，较高频率的迭代版本，来修复发现的 bug 和各种优化。因此，综合考虑，选用 RabbitMQ。
- 缺点：自身是 erlang 语言开发，源码比较难分析，需要扎实的 erlang 语言功底。

4）Kafka

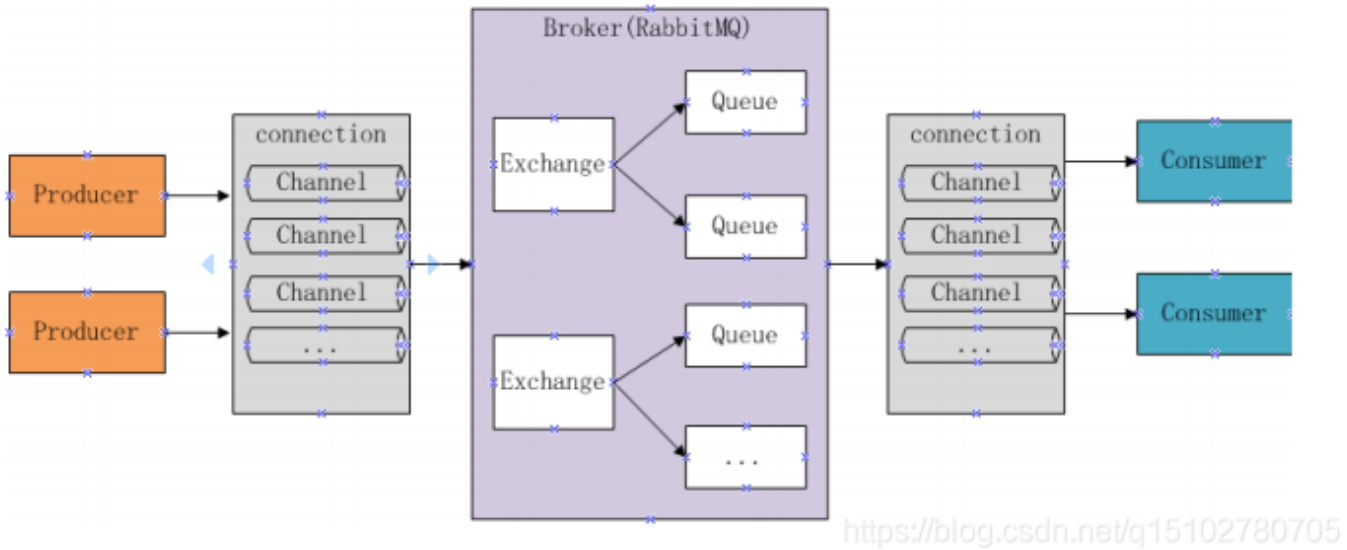
- 优点：kafka 采用拉取（Pull）的方式消费消息、吞吐量相对更高，适用于海量数据收集和传递场景，例如日志采集和集中分析。它的优势在于专为超高吞吐量的实时日志采集、数据同步和数据计算等场景设计；

- 缺点：消息中间件的功能相比其它 MQ 明显少一些，像常用的顺序消费、消息可靠性方面。

kafka 和 RabbitMQ 对比：

- 性能：消息中间件的性能主要衡量吞吐量，Kafka 单机 QPS 能达到百万级别，RabbitMQ 单机 QPS 万级别，kafka 更高；
- 数据可靠性：kafka 和 rabbitMQ 都具备多副本机制，数据可靠性都比较高；
- 消费模式：Kafka 由客户端主动拉取，RabbitMQ 支持主动拉取和服务器推送两种模式。所以 RabbitMQ 的消息实时性更高，且对于消费者来说更简单；而 kafka 可以由消费者根据自身情况去拉取消息，吞吐量更高；
- 幂等性：kafka 支持单个生产者，单分区单会话的幂等性，而 RabbitMQ 不支持；
- 其它特性：RabbitMQ 支持优先级队列，延迟队列，死信队列（**存储无法被消费的消息队列**）等等。

2.RabbitMQ



1）组件说明

- Broker：消息队列服务进程，包含 Exchange 和 Queue；
- Exchange：消息队列交换机，按一定规则将消息路由转发到某个队列，对消息进行过滤；
- Queue：消息队列，**存储消息**，消息到达队列会转发给某个消费者；
- Channel：信道，**消息推送**使用的通道；
- Producer：生产者，生产方客户端将消息发送到 MQ；

- Consumer：消费者，接收 MQ 转发的消息。

2）消息发送

- 生产者和 Broker 建立 TCP 连接（需要账号密码进行认证），建立 channel 通道；
- 生产者通过通道，把消息发送给 Broker；
- 由 Exchange 将消息根据 routeKey 进行转发，给对应的 Queue 队列；

3）接收消息

- 消费者和 Broker 建立 TCP 连接，建立 channel 通道；
- 消费者监听指定的 Queue 队列；
- 当有消息到达 Queue 时，Broker 默认将消息推送给消费者（如果是订阅模式，将推送给多个消费者）；
- 消费消息。

4）RabbitMQ顺序消费

针对消息有序性的问题，解决方法就是保证生产者入队的顺序是有序的，出队后的顺序消费交给消费者去保证。具体流程为：

拆分 queue，使得一个 queue 只对应一个消费者。由于 RabbitMQ 可以保证内部队列是先进先出的，所以让需要保持有序性的消息分配到同一个消息队列中，然后只用一个消费者去消费该队列，这样就可以保证消费消息的顺序性了。

5）重复消费问题（幂等性）

1. 产生原因

正常情况下，消费者消费消息以后，会发送一个确认消息给 MQ，待 MQ 知道消息被消费以后，就会将消息从队列中删除。

但由于网络传输故障等原因，确认消息没有传送到消息队列，或者消费者手动 ACK 的时候没有返回确认消息，导致 MQ 重复推送消息。

2. 解决方案

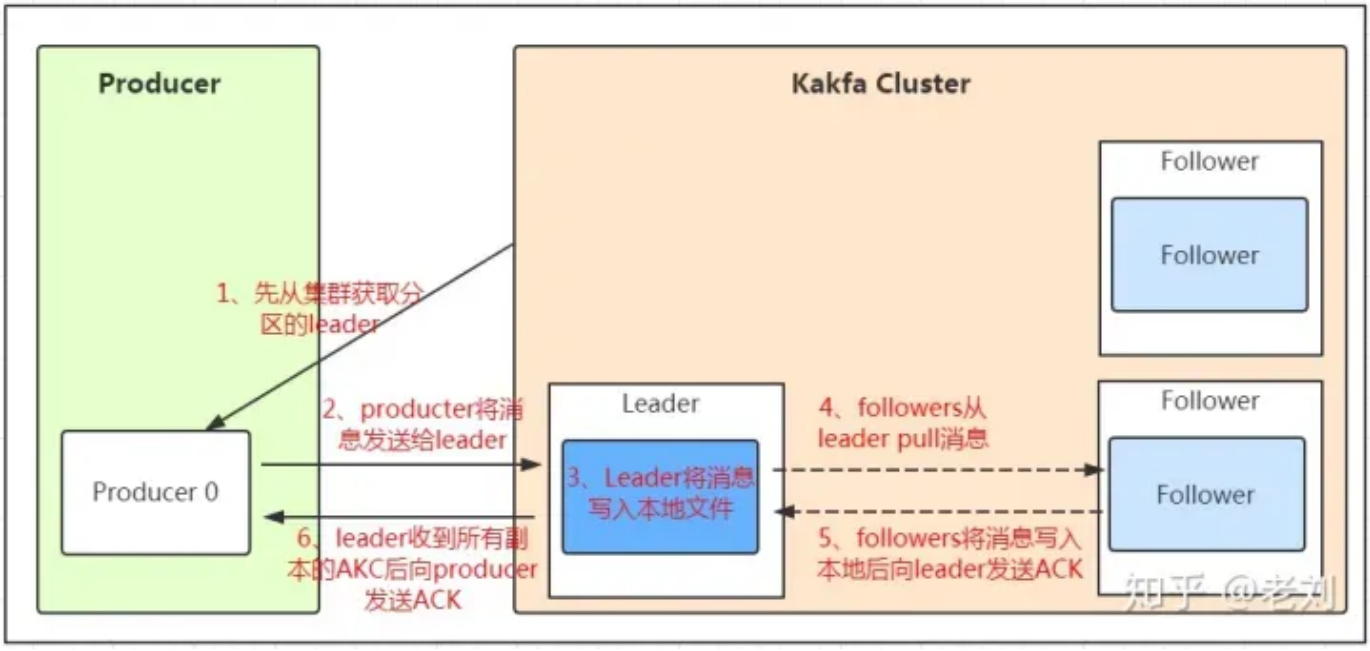
所以，针对以上异常导致的重复消费，消费者需要保证消息的幂等性，即不能让多次消费给服务带来问题。通常可以采用分布式锁来解决：将消息的唯一标识加一个过期缓存，消息消费前先判断一下是否消费过了。如果消费过就丢弃，并返回 ACK；否则就消费该消息，并缓存唯一标识。

3.kafka

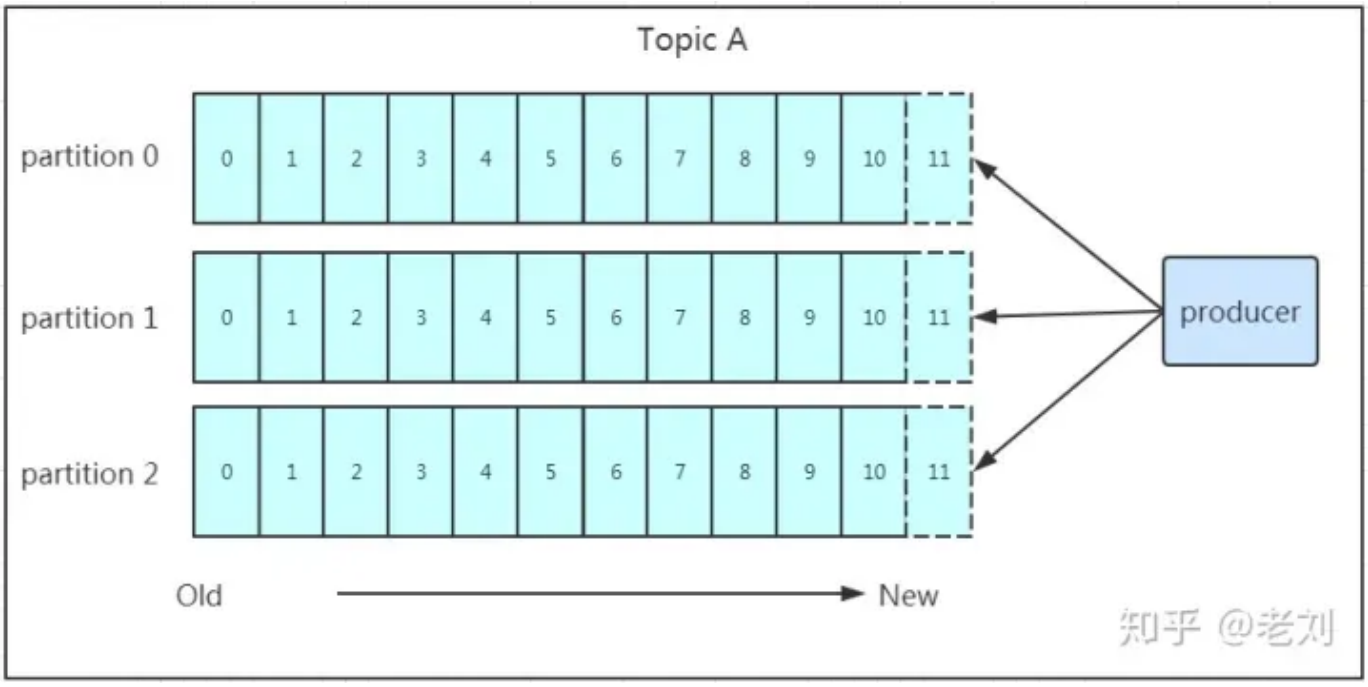
1）概念和组件

名称	解释
Broker	消息中间件处理节点，一个 Kafka 节点就是一个 broker，broker.id 不能重复
Topic	Kafka 根据 topic 对消息进行归类，发布到 Kafka 集群的每条消息都需要指定一个 topic
Producer	消息生产者，向 broker 发送消息的客户端
Consumer	消费者，从 broker 读取消息的客户端
Partition	分区，将一个 topic 的消息存放到不同分区，好处是方便扩展和提升并发
Replication	副本，分区的多个备份，备份分别存放在集群不同的 broker 中，有一个 leader，多个 follower。

2）kafka的工作流程



Producer 采用 push 模式将数据发布到 broker，每条消息会追加到分区中，然后分区顺序写入磁盘。所以，同一分区内的数据是有序的，写入示意图如下：



Partition 分区的作用主要有两个：

- 方便扩展：当一个 topic 上的数据日益增长，partition 可以解决一个分区日志存储文件过大的问题；
- 提高并发：多个消费者可以同时在多个分区上消费数据，提升了消息处理效率。

3）如何保证顺序

生产者：通过 ACK 应答机制，使用同步发送。当 Producer 向分区 Leader 写入数据时，可以通过设置参数来决定 Kafka 是否确认数据的安全性，这个参数可设置为 0、1、-1。此处需要同步发送，设置为1【0不返回确认，-1只需要 Leader 收到就返回确认】。

消费者：主题只能设置一个 Partition 分区，消费组中只能有一个消费者。

Kafka 顺序消费会严重牺牲性能，所以使用场景不多。

4）如何保证高可用

Kafka 主要通过集群的几种机制来保证高可用性：

- Partition 分区、副本和 ISR
- Broker 中的 controller
- 消费组和分区的 Rebalance 机制
- 分区副本的 HW（high-weight）高水位机制

1. Partition 分区、副本和 ISR

当新建主题时，需要指定分区和副本数。

分区相当于负载均衡，当 topic 上的消息越来越多时，消息的存储和读取可能就变得越来越慢。一个主题上的多个分区可以让数据分开存储，每一个分区上的数据都不一样，可以解决一个分区存储的日志文件过大问题；同时，也让消费者可以同时在分区上进行读取，提高了并发能力。

副本就是分区创建的多个备份，多个备份会分布在 Kafka 集群中的多个 broker 节点上，以保证高可用。分区副本有一个 Leader 和多个 Follower，Leader 负责读写，Follower 负责主动从 Leader 上 pull 消息，保存后发送一个 ACK。

Kafka 在全同步模式下，需要 Leader 等待所有 Follower 同步完成后，才发生 ACK。这时如果某个 Follower 出现故障，就会严重影响同步的整体流程。于是，Leader 会维护一个与其保持同步的副本集合，这个集合就是 ISR。如果同步过程中某个 Follower 超过配置的时间阈值还未向 Leader 发送同步信息，该 Follower 就会被踢出 ISR 集合。

当 Leader 挂了之后，多个 Follower 通过 ISR 集合中的顺序，选出第一个正常的节点升级为 Leader 提供读写服务。

2. Broker 中的 controller

每个 broker 在启动时会向 zookeeper 创建一个临时节点【比如 1,2,3】，最小序号的 broker 节点会作为集群中的 controller，负责以下两件事：

- 集群中有一个副本 Leader 挂掉以后，需要在 Leader 对应的 Follower 中选举出一个新的 Leader。选举规则是 ISR 集合第一个正常可用的 Follower，将其升级为 Leader；
- 当集群中的 broker/partition 新增或减少时，controller 会同步给其它 broker；

3. 消费组和分区的 Rebalance 机制

前提：消费组中的消费者没有指定分区消费

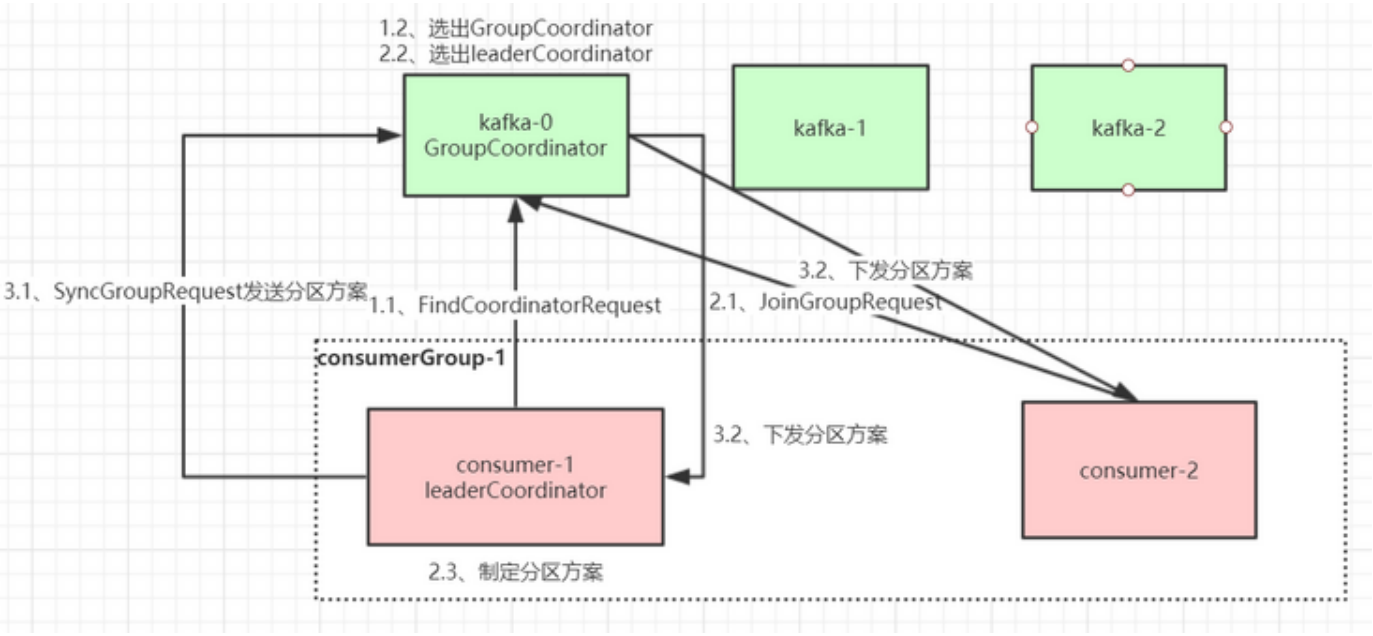
触发条件：当消费组中的消费者和分区的数量发生变化时

Rebalance 策略：

- range：按照分区序号分配，比如（9个分区，3个消费者的情况）：partition0–3 分配给 consumer1，p4–6–>c2，p7–9–>c3；
- round–robin 轮询：比如 partion0,3,6 –>c1，p1,4,7 –>c2；
- sticky 粘合：当发生 rebalance 时，会在之前已分配的关系上进行调整，不会改变之前的分配情况。比如在 range 策略中，如果第三个 consumer 挂了，c1会新增一个 partition7，c2新增8、9.

Rebalance 过程：

当有消费者加入消费组时，消费者、消费组即组协调器之间会经历以下几个阶段：（新增 consumer–2）



第一阶段：选择组协调器（GroupCoordinator）

每个消费组（consumer group）都会选择一个 broker 节点作为自己的组协调器，来负责监控消费组里所有消费者的心跳，判断是否宕机，然后开启消费者 rebalance。

消费者在启动时，会向 kafka 集群中的某个节点发送请求来查找对应的组协调器，并根其建立网络连接。组协调器的选择方式是，通过公式【hash(cousumer group id)%主题的分区数】选出当前消费组对应的分区，这个分区的 Leader 副本对应的 broker 节点就是当前消费组的 coordinator 组协调器。

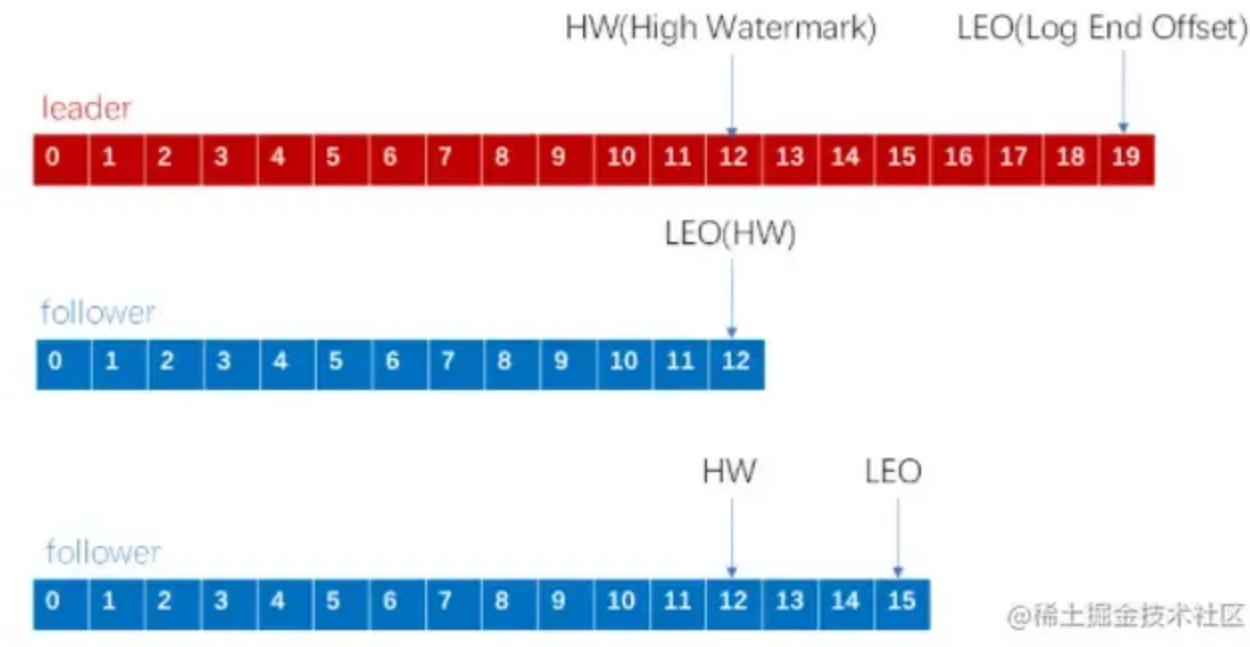
第二阶段：加入消费组（Join Group）

新来的消费者向 GroupCoordinator 发送加入请求，然后 GroupCoordinator 从这个消费组中选择第一个加入 group 的 consumer 作为 LeaderCoordinator 组里面的领头协调器，把消费组的情况发送给这个 Leader，由组间 Leader 负责制定分区方案。

第三阶段：同步消费组（Sync Group）

LeaderCoordinator 制定好分区方案以后发送给 GroupCoordinator，然后组间协调器就把分区方案下发给各个消费者，消费者后续的消费都会根据该分区方案来执行。

4. 分区副本的 HW 和 LEO



HW（high-water mark）是指副本中已完成 Leader-Follower 同步的位置，一个消息需要被所有 Follower 同步成功返回 ACK，且 HW 更新后，这个消息才会被消费者读到，以此来保证消费数据的一致性和副本数据的一致性。

LEO（log-end-offset）是指每个副本中最新消息的位置，最小的 LEO 就是 HW。

如果没有 HW，当 Leader 和 Follower 数据不一致时，比如 Leader 的 offset 已经到 15，后续消费者应该消费 16. 此时 Leader 挂掉，选择某个 Follower 升级为 Leader，当消费者找新的 Leader 消费，发现新 Leader 没有 offset 为 16 的数据，就会报错。

Follower故障

Follower 发生故障以后会被踢出 ISR 集合（动态变化），待 Follower 恢复以后，会读取本地磁盘记录的上次的 HW，并将该 log 文件高于 HW 的部分截取掉。从 HW 开始向 Leader 拉取数据同步，直到该 Follower 的 LEO 大于等于该 Partition 分区的 HW 时，说明 Follower 追上 Leader，就可以重新加入 ISR。

Leader故障

Leader 发送故障以后，会从 ISR 集合中选出一个新的 Leader。这时为了保证多个副本的数据一致性，其余 Follower 会先将各自 log 文件中高于 HW 的部分截掉（新 Leader 自己不会截掉），然后 Follower 们开始从新的 Leader 拉取数据进行同步。

5）Kafka中的优化问题

1. 防止消息丢失

- 生产者：使用同步发送，将 ack 设为 1 全同步（0为异步进行数据复制，–1只保证 Leader 接收到消息）；
- 消费者：自动提交改为手动提交，当消费完成后再进行 ack 应答。

2. 防止重复消费

为了防止消息丢失，当 Producer 发送完消息后，会根据有无收到 ack 应答去决定是否重新发送消息。当消费者已经收到消息，返回 ack 时由于网络抖动或者其它原因，Producer 没有收到应答重发消息后，消费者就会收到多条相同的消息。解决方案：

- 生产者关闭重试机制，可能造成消息丢失（不推荐）；
- 消费者幂等性保证：分布式锁。

3. 消息积压问题

当消费者的消费速度，远远赶不上生产消息的速度一段时间后，kafka 会堆积大量未消费的消息。导致消费者寻址的速度越来越慢，kafka 对外提供服务的性能也越来越差，从而可能会造成整个服务链变慢，导致服务雪崩。解决方案如下：

- 消费者使用多线程消费，充分利用机器的性能；

- 在同一个消费组中创建多个消费者，部署到其它机器上，一起消费；

4. 延时队列实现

应用场景：订单创建后，超过 30 分钟没有支付就自动取消。

具体实现：

- kafka 中创建相应的主题，并创建消费者消费该主题的消息，消息中带有创建的时间戳；
- 消费消息时判断，未支付订单消息的创建时间是否已经超过 30 分钟：1）如果是，就修改订单状态为超时取消；2）否则，记录当前消息的 offset，并等待 1 分钟后，再次向 kafka 中拉取该 offset 的消息进行判断，直到支付订单或超时取消。

6）kafka 的选主问题

kafka 中的选主问题，涉及到三处：

- 节点控制器选主（broker controller）
- 分区副本选主（partition leader）
- 消费组协调器选主（consumer group LeaderCoordinator）

1. 节点控制器（broker controller）

每个 broker 在启动时会向 zookeeper 创建一个临时节点【比如 1,2,3】，最小序号的 broker 节点会作为集群中的 controller，负责以下两件事：

- 集群中有一个副本 Leader 挂掉以后，需要在 Leader 对应的 Follower 中选举出一个新的 Leader。选举规则是 ISR 集合第一个正常可用的 Follower，将其升级为 Leader；
- 当集群中的 broker/partition 新增或减少时，controller 会同步给其它 broker。

Broker 选举的场景

第一，集群中第一个启动的 broker 节点会在 zk 中创建一个临时节点(/controller)来让自己成为控制器，其它节点启动后发现控制器已存在，就会在 zk 中创建 watch 对象。

第二，如果 controller 节点由于宕机或网络原因与 zk 断开，其它 broker 就会通过 watch 收到控制器变更的通知，于是纷纷竞选节点。竞选的方式也是在 zk 上创建一个 /controller 节点，且只能有一个 broker 成功，其余 broker 重新创建 watch 对象。

Broker 控制器脑裂问题

如果控制器所在的 broker 节点挂掉了或者 Full GC 停顿时间太长，超过了 zk 的会话超时时间，就会出现假死。这时，kafka 集群会选举出一个新的控制器，但如果之前“假死”的控制器恢复正常以后，它仍然是控制器身份，这样集群就会出现两个控制器，即控制器“脑裂”问题。

解决方式：

为了解决脑裂问题，zk 中还有一个和控制器相关的持久节点 /controller_epoch，存放的是一个数值 epoch number（纪元编号，可以认为是版本号）。集群每选举一次控制器，就会新增这个纪元编号，如果 broker 收到的数据中纪元编号小于该值，则忽略此消息。

2. 分区副本选主（partition leader）

分区副本的选举机制由 broker 控制器（controller）执行：

- 从 zk 中读取当前分区的所有 ISR（in-sync replicas，同步副本） 集合；
- 调用配置的分区选择算法，选择分区的 leader。

当分区的 Leader 挂了之后，控制器会根据 ISR 集合中的顺序，选出第一个正常的节点升级为 Leader 提供读写服务。

Unclean leader 选举

ISR 是动态变化的，当 ISR 为空时说明 leader 副本全挂掉了，此时需判断一下 kafka broker 的参数 unclean.leader.election.enable 是否打开。若开启，则 kafka 会在非同步副本中选出一个作为 leader，这个过程叫做 **unclean leader 选举**。

建议关闭 unclean leader 选举，因为一般业务下数据的一致性要比可用性重要。

3. 消费组协调器选主（consumer group LeaderCoordinator）

Kafka 消费端，消费组协调器需要为组内的消费者选举出一个组间 leader。涉及两种场景：

- 如果消费组内还没有 leader，那么第一个加入消费组的消费者将成为组内 leader；
- 如果某一时刻 leader 退出了消费组，那么会重新选举 leader，即组协调器中保存的消费者 hashmap 的第一个键值对 key（等同于随机）。