Kafka

分布式的，基于发布订阅模式的消息队列，主要用于大数据实时处理

---场景---
异步、并行、排队

---好处---
解耦、可恢复性、缓冲、灵活性&峰值处理、异步通信

---MQ 模式---
1. 点对点（Queue），一对一，消费者主动拉取数据，消息收到后消息清除；支持多个消费者，但消息只可消费一次
2. 发布/订阅（Topic），一对多，消费者消费数据后不会清除消息；支持多个消费者（订阅者），Topic 的消息会被所有订阅者消费

- **集群部署**
  1.
  - 参数配置

    ---修改项---
    # broker 的全局唯一编号，不能重复
    broker.id=1
    delete.topic.enable=true
    # kafka 运行日志存放的路径
    log.dirs=/home/aurelius/software/kafka_2.13-2.7.0/logs
    # 连接 zookeeper 集群的地址
    zookeeper.connect=node01:2181,node02:2181,node03:2181
    # ISR 时间阈值
    # replica.lag.time.max.ms=

  - 命令操作
    - kafka-server-start.sh

      # 启动
      bin/kafka-server-start.sh -daemon config/server.properties
    - kafka-server-stop.sh

      # 停止
      bin/kafka-server-stop.sh stop
    - kafka-topics.sh

      # 查看所有 topic
      bin/kafka-topics.sh --zookeeper node01:2181 --list

      # 创建 topic
      bin/kafka-topics.sh --zookeeper node01:2181 --create --replication-factor 3 --partitions 1 --topic first

      # 删除 topic
      bin/kafka-topics.sh --zookeeper node01:2181 --delete --topic first

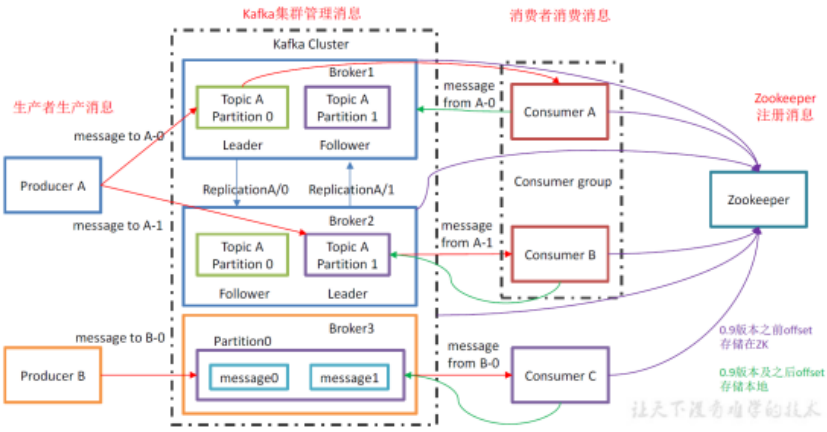
      # 查看 topic 详情
      bin/kafka-topics.sh --zookeeper node01:2181 --describe --topic first

      # 修改分区数
      bin/kafka-topics.sh --zookeeper node01:2181 --alter --topic first --partitions 6
    - kakfa-console-producer.sh

      # 发送消息
      bin/kafka-console-producer.sh --broker-list node01:9092 --topic first
    - kafka-console-consumer.sh

      # 消费消息
      # 通过 zookeeper
      bin/kafka-console-consumer.sh --zookeeper node01:2181 --topic first
      # 从订阅开始消费消息
      bin/kafka-console-consumer.sh --bootstrap-server node01:9092 --topic first
      # 从主题启动开始消费消息
      bin/kafka-console-consumer.sh --bootstrap-server node01:9092 --from-begining --topic first
- **架构**
  2.

- 基础架构



  - Producer

    生产者，向 broker 发送消息的客户端
  - Consumer

    消费者，向 broker 取消息的客户端
  - Consumer Group

    消费者组，逻辑上的订阅者（对 Topic），由多个 consumer 组成。
    每个消费者消费不同分区的数据，一个分区只能由组内一个消费者消费，组之间互不影响
  - Broker

    一个 Kafka 服务器就是一个 broker，一个 broker 可以容纳多个 topic
  - Topic

    主题，一个抽象消息队列，Producer 和 Consumer 操作的对象
  - Partition

    为实现扩展性，一个 Topic 分为多个 Partition 并分布到多个 Broker 上，每个 Partition 是一个有序队列
  - Replica

    副本，保障节点故障时，Partition 数据不丢，且 Kafka 集群可继续工作，每个 Topic 的每个 Partition 都有若干副本，一个 Leader 和多个 Follower
  - Leader

    每个 Partition 多个副本的主，Producer 和 Consumer 的写读对象都是 Leader
  - Follower

    每个 Partition 的多个 Replica 的从，实时从 Leader 同步数据，保持和 Leader 数据的同步，Leader 故障时，Follower 变成新的 Leader
- 工作原理
  - Topic

    Kafka 消息以 Topic 分类，Producer 和 Consumer 面对的都是 Topic
    Topic 是逻辑上的概念
    - Partition

      Partition 是物理上的概念
      - log

        每个 Partition 对应一个 log 文件
        Producer 生产的数据会追加到 log 文件末端，每条数据有自己的 offset
        Consumer Group 中每个 Consumer 会记录自己消费到的 offset，以便故障恢复
        - Segment

          分片
          防止 log 文件过大导致数据定位效率低下

          路径命名规则：Topic - Partition，如：first-0
          .index/.log 文件命名规则：当前 segment 第一条消息的 offset 命名，如：00000000000000.index, 00000000000000.log
          - .index

            存储索引

            存储两列：offset, log_offset(log 文件对应 message 的物理便宜地址-行偏移量)
          - .log

            存储 message
  - Producer
    - 分区策略

      向一个 Topic 发布的消息会被合理分数到 Topic 的可用 Partition 中

- 好处

  1. 方便集群扩展，调整 Partition 可以适应所在机器，且 Topic 可以有多个 Partition，从而集群可以适应任意大小的数据
  2. 提高并发，以 Partition 为单位读写

- 原则

  1. 直接指明 ProducerRecord 的 Partition
  2. 没有指明 Partition，但有 Key，对 Key 的 hash 与 Topic 的可用 Partition 数取模，即是所属 Partition
  3. 没有指明 Partition，也没有 Key，第一次调用随机生成一个整数 n，对 n 与 Topic 的可用 Partition 数取模，即是所属 Partition （Round-Robin 算法）

- 数据可靠性

  每个 Partition 收到 Producer 发送的数据后，都需要向 Producer 发送 ack，Producer 收到 ack 才会进行下一轮发送，否则重新发送

  - 副本同步策略

    1. 半数以上完成同步才 ack - 延迟低，但容忍 n 个节点故障，需要 2n+1 的副本数，数据大量冗余
    2. 全部同步完成才 ack - 延迟高，但容忍 n 个节点故障，只需要 n+1 的副本数

    Kafka 选择 第2种 方案，网络延迟对 Kafka 影响较小

  - ISR

    in-sync replica set
    Leader 维护的一个动态集合，持有所有与 Leader 保持同步的 Follower
    ISR 中的 Follower 完成数据同步后，Leader 会想 Follower 发送 ack，如果 Follower 长期未应答 Leader，该 Follower 会被提出 ISR，超时时间阈值由 replica.lag.time.max.ms 参数设定
    Leader 故障时，新的 Leader 从 ISR 选举

  - ack 应答机制

    ---acks---
    0: Producer 不等待 Broker 的 ack，Broker 接收到消息没有写入磁盘就返回，最低延迟，但 Broker 故障可能丢失数据
    1: Producer 等待 Broker 的 ack，Partition 的 Leader 落盘成功后返回 ack，Follower 同步成功前 Leader 故障，可能丢失数据
    -1: (all) Producer 等待 Broker ack，Partition 的全部 Replica 落盘成功后返回 ack，Follower 同步完成后，Broker 发送 ack 前如果 Leader 故障，可能导致数据重复

  - 故障处理

    ---Follower 故障---
    1. Follower 故障后会被临时踢出 ISR
    2. Follower 恢复后，Follower 先读取本地之前记录的 HW，并删除 log 中高于该 HW 的部分
    3. Follower 从该 HW 开始同步 Leader 数据，待该 Follower 的 LEO 大于等于该 Partition 当前 HW，表示该 Follower 已追上 Leader
    4. 回归 ISR

    ---Leader 故障---
    1. 从 ISR 选出新的 Leader
    2. 其余 Follower 截掉各自 log 中高于 Partition HW 的部分，以保证 Replica 之间的数据一致性（不保证整体数据不丢失或不重复）
    3. 其余 Follower 从新 Leader 同步数据

    - LEO

      Log End Offset
      每个副本的最后一个 offset

    - HW

      High Watermark
      所有副本的最小 LEO，HW 之前的数据才是 Consumer 可见的

- Exactly Once

  要求数据既不重复也不丢失
  At Least Once + 幂等性 => Exactly Once
  0.11 版本以前，只能保证数据不丢失，需在下游对数据全局去重
  0.11 引入幂等性

  - At Least Once

    acks=-1 可保证数据不丢失

  - At Most Once

    acks=0 可保证 Producer 每条消息只会发送一次

  - 幂等性

    Producer 无论向 Server 发送多少次重复数据，Server 端只会持久化一条
    enable.idompotence=true # 启用幂等性

    1. 开启幂等性的 Producer 初始化时会分配 PID
    2. 发往 Partition 的消息会附带 Sequence Number
    3. Broker 端对 <PID,Partition,SeqNumber> 做缓存去重，最终只会持久化一条

    PID 重启会变化，不同 Partition 也不能保障幂等性

- Consumer

---pull 模式---
从 Broker 读取数据，缺点是无数据时 Consumer 会陷入请求循环，可以通过设置 timeout 等待一段时间

---push 模式---
消息发送速率由 Broker 决定，很难适应消费速率不同的消费者，可以尽快传递消息，但可能造成 Consumer 来不及处理消息，如拒绝服务，网络拥塞

- 分区分配策略

  一个 Consumer Group 有多个 Customer，一个 Topic 有多个 Partition，确认哪个 Customer 消费哪个 Partition 有两种策略

  - RoundFobin

    将 Consumer Group 订阅的所有 Topic 的 Partition 当做一个整体，对 TopicAndPartition 编号，然后轮询分配到 Consumer
    好处: 分配均衡，最多只相差一个 Partition
    坏处: Group 中的 Customer 消费不同 Topic 时会存在业务逻辑异常

  - Range

    默认分配策略，以 Topic 为单位，将 Partition 分范围分配给 Consumer
    好处: 不存在业务逻辑异常
    坏处: 分布不均，订阅的 Topic 越多越不均衡

  - Rebalance

    分区重新分配
    当 Customer Group 中 Customer 个数变更，或订阅 Topic 的 Partition 变更时，会触发分区的重写分配

- offset 维护

  Consumer 实时记录自己消费到的 offset，以便故障恢复，原地继续消费

  - zookeeper

    0.9 之前 offset 默认存放在 Zookeeper

  - __consumer_offsets

    0.9 开始 offset 默认放在 Kafka 内置的 Topic __consumer_offsets 中

    ```
    # consumer.properties，关闭内置 topic 中的 offset
    exclude.internal.topics=false

    ---读取 offset---
    # 0.11 之前
    bin/kafka-console-consumer.sh --topic __consumer_offsets --zookeeper node01:2181 --formatter
    "Kafka.coordinator.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config
    config/consumer.properties --from-beginning
    # 0.11 开始
    bin/kafka-console-consumer.sh --topic __consumer_offsets --zookeeper node01:2181 --formatter
    "Kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config
    config/consumer.properties --from-beginning
    ```

- Consumer Group

  一个消息只会被 Consumer Group 中一个 Consumer 消费

  - 配置

    ```
    # 在 node01, node02 节点修改 consumer.properties
    group.id=aurelius
    ```

  - 演示

    ```
    # 在 node01, node02 节点启动消费者
    bin/kafka-console-consumer.sh --bootstrap-server node01:9092 --topic first --consumer.config
    config/consumer.properties

    # 在 node03 节点启动生产者
    bin/kafka-console-producer.sh --broker-list node01:9092 --topic first
    > hello kafka

    # 只会在 node01 或 node02 其一看到 hello kafka
    ```

- 高效读写
  - 顺序写磁盘

    写过程一直追加到文件末端，顺序写 600M/s，而随机写只有 100K/s（受限磁盘的机械结构，磁头寻址需要大量时间）

  - 零复制技术

    常规拷贝：SourceFile -> KernelSpace(Page Cache) -> UserSpace(Application Cache) -> KernelSpace(Socket Cache) -> NIC
    零拷贝：SourceFile -> KernelSpace(Page Cache) -> NIC
    用户程序通过内核指令完成拷贝，数据不经过用户空间
    NIC: network interface controller

- Zookeeper

  协助 Controller

- Controller

  集群的一个 Broker 会被选举为 Controller，负责管理集群 Broker 上下线，Topic 的分区副本分配，Leader 选举等
  1. 所有 Broker 注册在 ZK 的 /brokers/ids
  2. ZK 中 /brokers/topics/first/partition/0/state "leader":1,"isr":[1,2] 记录 Topic 的所有 Partition 状态，如 Leader, ISR
  3. Controller 监听 ZK 上 /brokers/ids
  4. 当 Leader(Broker) 下线时，Controller 获取 /brokers/topics/... 中 ISR，从中选举新的 Leader
  5. Controller 更新 /brokers/topics/... 中的 Leader 和 ISR

- 事务

  事务保证在 Exactly Once 语义基础上，生产和消费可以跨区和跨会话全部成功或全部失败

  - Producer 事务

    引入全局唯一 Transaction ID，与 Producer 的 PID 绑定
    引入 Transaction Coordinator，管理 Transaction 任务状态，并负责将事务写入 Kafka 内部 Topic，保障事务的持久性

  - Consumer 事务

    事务保障相对较弱，Consumer 可以通过 offset 访问任意消息，且不同 Segment File 的生命周期不同，同一事务的消息可能重启后部分已被删除

API

3

- Producer API

  Producer 发送消息采用异步发送方式
  1. Producer 的 main 线程通过 Interceptors -> Serializer -> Partitioner 等处理 ProducerRecord 后，以 RecordBatch 将消息发送给线程共享变量 RecordAccumulator
  2. Proudcer 的 Sender 线程不断从 RecordAccumulator 拉取消息，发送到 Broker

  ---参数---
  batch.size: sender 发送数据的阈值
  linger.ms: 如果迟迟未达 batch.size，sender 等待该时间阈值后也会发送数据

  - 异步

    KafkaProducer: Producer 对象，用于发送消息
    ProducerConfig: Producer 的一系列配置参数
    ProducerRecord: 待发送的消息对象

    - 回调

      onCompletion(RecordMetadata, Exception) // Producer 收到 ack 时调用

      消息发送失败会自动重试，不需要在回调中手动重试

  - 同步

    同步会阻塞当前线程，直到返回 ack
    Producer.send 返回的是一个 Future 对象，调用 Future 对象的 get 方法即可实现同步发送效果

- Consumer API

  数据在 Kafka 中是持久化的，Consumer 一般不用担心数据丢失
  为保障 Consumer 故障恢复后原地继续消费，Consumer 自己消费的 offset 需实时记录
  offset 维护是 Consumer 消费数据的关键问题

  先提交 offset 后消费 -> 漏消费
  先消费后提交 offset -> 重复消费

  - 自动提交 offset

    KafkaConsumer: Consumer 对象，用于消费数据
    ConsumerConfig: Consumer 的一系列配置参数
    ConsumerRecord: 消费的消息对象

    ---offset 相关参数---
    enable.auto.commit: 是否开启自动提交 offset
    auto.commit.interval.ms: 自动提交 offset 的时间间隔

    自动提交 offset 基于时间间隔，无法把握 offset 提交时机

  - 手动提交 offset

    将本次 poll 的一批数据最高的 offset 提交

    - 同步

      consumer.commitSync();
      提交 offset 会阻塞当前线程，失败自动重试，一直到成功提交，大大影响吞吐量

    - 异步

      consumer.commitAsync(new OffsetCommitCallback(){
      onComplete(Map<TopicPartition, OffsetAndMetadata>, Exception);
      });

      提交 offset 不会阻塞当前线程，失败不会重试，可能提交失败，但吞吐量大，是更好的选择

- 自定义存储 offset

  0.9 前 offset 在 ZK
  0.9 起 offset 在 __consumer_offsets
  此外 Kafka 可以选择自定义存储 offset

  ---Rebalance---
  1. Rebalance 后，每个 Consumer 消费的 Partition 会发生变化
  2. Consumer 先找到自己被重写分配的 Partition，并定位到 Partition 最近提交的 offset 位置，继续消费

  ConsumerRebalanceListener
  - onPartitionsRevoked, Rebalance 之前调用，一般调用 commitOffset(current) 用于保存现场 offset
  - onPartitionsAssigned, Rebalance 之后调用，一般调用 consumer.seek(partition, offset) 用于复原 offset

- Interceptor

  用于实现 client 端定制化逻辑，0.10 引入

  - 原理

    ProducerInterceptor
    - configure(configs), 获取配置信息和初始化数据
    - onSend(ProducerRecord), 运行在 main 线程中，在消息被序列化和计算分区之前被调用，不建议在此修改所属 Topic 和 Partition
    - onAcknowledgement(RecordMetadata, Exception), 在 Sender 线程成功发送 RecordAccumulator 的消息，或发送过程中失败时调用，通常在 Producer 回调逻辑前触发，不建议放入很重的逻辑，否则会拖慢 Producer 的发送效率（Sender IO 线程）
    - close, 关闭 interceptor，在 producer.close() 时触发，用于执行一些资源清理工作

    多个 interceptor 的线程安全需要自行保证，可自定顺序调用
    interceptor 的异常会被内部消化，不会向上传递?

  - 案例

    1. onSend 中在消息头部加入时间戳
    2. 在 onAcknowledgement 中统计成功数和失败数，在 close 中答应统计结果

Kafka-Eagle

4

监控 & 管理

- 配置

  ```
  ---修改(kafka)kafka-server-start.sh---
  export KAFKA_HEAP_OPTS="-server -Xms2G -Xmx2G -XX:PermSize=128m -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=8 -XX:ConcGCThreads=5 -XX:InitiatingHeapOccupancyPercent=70"
  export JMX_PORT="9999"
  # export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"

  ---修改(eagle)system-config.properties---
  cluster1.zk.list=node01:2181,node02:2181,node03:2181
  cluster1.kafka.eagle.offset.storage=kafka
  kafka.eagle.metrics.charts=true
  kafka.eagle.driver=com.mysql.jdbc.Driver
  kafka.eagle.url=jdbc:mysql://node01:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
  kafka.eagle.username=aurelius
  kafka.eagle.password=999999
  ```

- 演示

  ```
  # 启动，需先启动 ZK & Kafka
  bin/ke.sh start

  # 登录 http://node01:8084/ke
  ```

Flume-Kafka

5

- 配置

```
# define
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F -c +0 /home/aurelius/data/flume.log
a1.sources.r1.shell = /bin/bash -c

# sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers = node01:9092,node02:9092,node03:9092
a1.sinks.k1.kafka.topic = first
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1

# channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# bind
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

- 演示

```
1. 启动 KafkaConsumer
2. 启动 flume
bin/flume-ng agent -c conf/ -n al -f jobs/flume-kafka.conf
3. 向 /home/aurelius/data/flume.log 追加数据，查看 KafkaConsumer 消费情况
echo hello kafka >> /home/aurelius/data/flume.log
```

- **Questions**

6

1. Kafka 的 follower 长期下线,会重新分配 follower 吗?
2. ISR(InSyncRepli), OSR(OutSyncRepli), AR(AllRepli) 分别是?
3. HW, LEO 分表是?
4. Kafka 怎么体现消息顺序性?
5. 分区器，序列化器，拦截器是什么？处理顺序是什么?
6. Producer 整体结构是什么？使用了几个线程？分别是?
7. Consumer Group 中 Consumer 个数超过 Topic 分区数，Consumer 就会消费不到数据，这是否正确?
8. offset 提交的是当前 offset 还是 offset+1？（后者）
9. 重复消费的场景?
10. 漏消费的场景?
11. kafka-topics.sh 增删一个 topic 的背后逻辑?
1. 在 ZK 的 /brokers/topics 创建新的节点，如：/brokers/topics/first
2. 触发 Controller 的监听
3. Controller 创建 topic，并更新 metadata cache
12. Topic 的 Partition 是否可以增加，怎么增加，或为何不可以?
13. Topic 的 Partition 是否可以减少，怎么减少，或为何不可以?
14. Kafka 内部 Topic 有哪些，有什么用?
15. kafka 分区分配策略?
16. Kafka 日志目录结构?
17. 怎么通过指定 offset 找到对应消息?
18. Kafka Controller 的作用？（Broker 上下线，Partition 分区分配，Leader 选举）
19. 哪些地方需要选举？各自的选举策略?
20. 失效副本是指？应对策略?
21. 哪些设计提高了 Kafka 的性能?