Kafka

Kafka 知识点

- kafka发送消息的两种模式
- kafka如何保证消息的可靠性（生产者通过ack应答机制，消费者通过提交offset确认消费成功，生产者消费者broker三个方面考虑）
- kafka如何保证消息的有序性（分区内有序，可以考虑设置同一类型的消息的key相同，将相同key的消息写入同一分区）
- kafka如何保证消息不重复（通过幂等性和kafka的事务型producer）

开源分布式事件流平台，用于高性能数据数据管道，流分析，数据集成和关键任务应用

- 消息队列
- 临时存储
- 分布式流平台（发布订阅流数据流、以容错的持久化方式存储数据流、处理流数据（kafka stream））
- 低延迟

消息队列的应用场景

- 异步处理，将耗时的操作放入到其他系统中，通过消息队列将需要进行处理的消息进行存储，（如短信验证码的发送）实现快速响应
- 系统解耦，避免两个系统之间的直接调用（迪米特法则），一个微服务将消息放入到消息队列中，另一个微服务可以从队列中把消息取出来进行处理，实现解除耦合
- 流量削峰，避免过高的并发导致的系统瘫痪
- 日志处理（大数据领域常见，将消息队列作为临时存储或通信管道）

kafka的特性： 可靠性：分布式的、可分区的、数据可备份的、高容错的 可扩展性：在无需停机的情况下实现轻松扩展 消息持久性：kafka支持将消息持久化道本地磁盘 高性能：发布订阅具有很高的吞吐量，存储TB级别的消息数据依然能保持稳定的性能 数据以log日志的方式顺序存储在磁盘上

生产者消费者模型

producer负责将消息放入队列 consumer负责将消息取出进行处理

kafka broker使用scala实现 producer和comsumer使用java实现

两种模式

点对点模式：

- 每个消息只有一个接收者，一旦被消费，消息就会从消息队列中移除
- 生产者消费者没有依赖性
- 接收者在成功接收消息之后需向队列应答成功，以便队列删除当前消息 发布订阅模式：
- 一条消息可以被对多个订阅者接受，支持消息广播
- 发布者和订阅者之间有时间上的依赖性，针对某个主题的订阅者，它必须创建一个订阅者之后，才能消费发布订阅者
- 为了消费消息，订阅者需要提前订阅该角色主题，保持在线运行

- 提供对消费者进行分组的功能，同一组的消费者并行消费，消费者消费不同的分区
- 对消息数据进行分区，并提供多个分区备份，一个分区只能由一个消费者进行消费 **实际应用中多用的是发布订阅模式**

kafka集群部署

集群配置 这里采取本地部署三个kafka实例的方式 server.properties配置

```
# 节点broker-id
broker.id=0
#手动指定端口号
port=9092
# 日志数据目录
log.dirs=/Users/lijie3/Documents/data/kafka/kafka-data
# zookeeper连接地址，方便kafka数据的清理
zookeeper.connect=localhost:2181,localhost:2182,localhost:2083/kafka
```

connect-distrubted.properties

```
#配置服务地址
bootstrap.servers=localhost:9092
```

producer.properties

```
bootstrap.servers=localhost:9092
```

consumer.properties

```
bootstrap.servers=localhost:9092
```

connect-standalone.propeties

```
bootstrap.servers=localhost:9092
```

zookeeper.properties

```
#zookeeper数据存储配置地址
dataDir=/Users/lijie3/Documents/data/kafka/zookeeper-data
#zookeeper端口号
clientPort=2181
```

启动kafka

先启动zk

```
./bin/zookeeper-server-start.sh config/zookeeper.properties
```

然后再启动kafka进程

```
./bin/kafka-server-start.sh config/server.properties
```

停止kafka时先关闭kafka进程，再关闭zookeeper

kafka基准测试工具

kafka内部提供了性能测试工具 生产者：测试生产者每秒传输的数据量 消费者：测试消费每秒拉取的数据量

基本命令

主题

```
./bin/kafka-topics.sh
#查询所有topic
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
#新增test-topic
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic
test-topic
#新增test-topic 指定3分区3副本
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic
test-topic --partitions 3 --replication-factor 3
#查看分区详情
/bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic test-topic
--describe
Topic: test-topic   TopicId: GlEcZgJkTaWjBeqBiwev2Q PartitionCount: 3
ReplicationFactor: 1    Configs:
    Topic: test-topic   Partition: 0    Leader: 0   Replicas(AR): 0 Isr: 0
    Topic: test-topic   Partition: 1    Leader: 0   Replicas: 0 Isr: 0
    Topic: test-topic   Partition: 2    Leader: 0   Replicas: 0 Isr: 0
#修改，指定3个分区，分区只能增加不能减少，不能通过命令行修改副本
/bin/kafka-topics.sh --bootstrap-server localhost:9092  --topic test-topic
--alter --partitions 3
```

生产者

```
#发送数据
./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic
```

```
test-topic
> hello world
```

消费者

```
#从上次消费之后的偏移量开始消费
./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
test-topic
#从队列头部开始消费
./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
test-topic --from-beginning
hello world
```

javaAPI

生产者

```
//不带回调的异步模式
 Properties prop = new Properties();
        prop.put("bootstrap.servers","localhost:9092");
        //应答种类
        prop.put("acks","all");

prop.put("key.serializer","org.apache.kafka.common.serialization.StringSer
ializer");

prop.put("value.serializer","org.apache.kafka.common.serialization.StringS
erializer");
        KafkaProducer<String,String> kafkaProducer = new
KafkaProducer<String, String>(prop);
        for (int i = 0; i < 10; i++) {
             Future<RecordMetadata> future = kafkaProducer.send(new
ProducerRecord<>("quickstart-events", null, i + ""));
            future.get();
            log.info("第"+i+"条消息");
        }
```

```
//异步带回调
 public static void main(String[] args) throws ExecutionException,
InterruptedException {
        Properties prop = new Properties();
        prop.put("bootstrap.servers","localhost:9092");
        //应答种类
        prop.put("acks","all");

prop.put("key.serializer","org.apache.kafka.common.serialization.StringSer
ializer");
```

```
prop.put("value.serializer","org.apache.kafka.common.serialization.StringS
erializer");
        KafkaProducer<String,String> kafkaProducer = new
KafkaProducer<String, String>(prop);
        for (int i = 0; i < 10; i++) {
            Future<RecordMetadata> future = kafkaProducer.send(new
ProducerRecord<>("quickstart-events", null, i + ""),((recordMetadata, e) -
> {
                    //判断发送是否成功
                if(e == null){
                    String topic = recordMetadata.topic();
                    int partition = recordMetadata.partition();
                    long offset = recordMetadata.offset();
                    log.info("topic：{},partition:{},offset:
{}",topic,partition,offset);
                }else{
                    e.printStackTrace();
                }
            }));
            future.get();
            log.info("第"+i+"条消息");
        }
    }
   //同步发送,直接使用阻塞方法get拿到发送的结果
      kafkaProducer.send(
                                new ProducerRecord<>("test-topic", null,
"hello world,"+ i)).get();
```

kafka概念

- zookeeper集群：保存kafka相关元数据，管理协调kafka集群
- broker：由多个broker组成，无状态，通过zk来维护集群状态
- producer：生产者
- consumer：消费者
- consumer group: 可扩展且具有容错性的消费者机制，一个消费者组具有唯一的id group.id
- partition： 一个topic对应多个分区，将分区分布在不同的服务器上,分区提供负载均衡的能力，实现系统的高伸缩性
- replicas：分区的副本，分区副本也是放在不同的服务器上，用来容错
- topic： 主题是一个逻辑概念，用于生产者发布数据，消费者拉取数据，一个主题中的消息是有结构的，一般一个主题包含一类消息，一旦一个消息发送到主题中，这些消息不能被更新
- offset：偏移量，记录下一跳将要发送给comsumer的消息的序号，默认kafka将offset存储在zk中，在一个分区中，消息是有顺序存储着，每个分区的消费都是有一个递增的id，这就是偏移量offset，偏移量在分区中有意义，在分区之间没有意义

消费者组

- 一个消费者组中能包含多个消费者，共同消费topic中的数据
- 一个topic中如果只有一个分区，那么这个分区只能被消费者组中的一个消费者消费
- 有多少个分区，那么可以被同一个组内的多少个消费者消费

kafka幂等性（解决生产者消息重复性问题）

> http请求中，一次请求或多次请求拿到的响应是一致的 执行多次操作与执行一次操作的影响时一样的
> 如果kafka生产者在提交数据到broker后数据写入分区中，而broker响应给producer的ack应答失败，这时，producer会再次尝试发送相同的消息到broker，直到收到正常的ACK应答，而broker能保证producer retry的多条数据只有一条写入分区中

开启kafka的幂等性： 发送消息时，会连着pid(生产者唯一编号)和squence number一起发送，kafka接受到消息，会将消息和pid、sequence number一起保存下来，当生产者发送过来的sequence number小于等于partition消息中的sequence number，kafka会忽略这条消息

```
prop.put("enable.idempotence",true);
```

kafka生产者

发送数据流程 producer --> send（producerRecord）--> 拦截器（interceptor ，如flume等）--> 序列化器（serializer）-->分区器（partitioner,默认大小32M，每一批次大小16K，只有当发送的数据达到16k，才能发送这一批数据）-->sender（读取数据发送到broker） 分区器：双端队列，内存池 配置： batch.size 16K 分批发送大小 linger.ms 500 （ms），如果批次数据没有达到16K，而时间到了500ms（默认0ms，表示发送没有延迟），这还是会发送这一批数据 sender（读取数据发送到broker）：最多缓存5个允许5个请求同时发送，允许未收到应答继续发送，最多容忍5次未收到ACK 同步处理：所有发送任务必须全部完成之后才进行返回 异步发送：允许用户把消息放入消息队列，并不立即处理它，然后再需要的时候再去处理（实际上是在分区器中进行的异步，消息还未到达broker） 消息压缩：为提高消息传输效率，尽量节省网络带宽，可以在Producer端指定compression.type开启压缩，例如指定为gzip可以使用gzip压缩 消息解压缩发生在consumer端或者broker端

分区器

- 合理的使用存储资源，在每个Partition的broker上存储，把数据分布在broker，实现负载均衡
- 提高并行度，以分区为单位发送数据，消费者可以以分区为单位消费数据

生产者分区写入策略

默认分区策略：

- 如果指定的分区，则直接使用该分区进行发送
- 如果没有指定分区，指定了key，则按key进行hash运算，得到分区进行发送
- 在老版本中，如果没有指定key，则会选择粘性分区器，会随机选择一个分区，并尽可能一直使用该分区，待该分区的batch（16k一批次）已经满了，kafka再随机选择一个分区并与上次分区不同
- 在新版本中，没有指定key，默认会使用轮询策略进行消息写入 如果没有指定分区，则会根据key进行进行hash计算如果指定了分区，则直接发送到指定分区

可以通过

```
//指定key发送到相同分区
 kafkaProducer.send(new ProducerRecord<>("test-topic", "f", "hell world
async" +i ),((recordMetadata, e) -> {
```

```
                    //判断发送是否成功
                if(e == null){
                    String topic = recordMetadata.topic();
                    int partition = recordMetadata.partition();
                    long offset = recordMetadata.offset();
                    log.info("topic: {},partition:{},offset:
{}",topic,partition,offset);
                }else{
                    e.printStackTrace();
                }
            }));
```

自定义分区器

```
public class MyPartitioner implements Partitioner {
    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object
value, byte[] valueBytes, Cluster cluster) {
        int partition = 0;
        //获取数据
        String msgVal = value.toString();
        //根据内容选择分区
        if(msgVal.contains("0")){
            partition = 0;
        }else{
            partition = 1;
        }
        return partition;
    }

    @Override
    public void close() {

    }

    @Override
    public void configure(Map<String, ?> map) {

    }
}
//producer端指定自定义分区器
//指定自定义的partitioner

prop.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,MyPartitioner.class.getNa
me());
        KafkaProducer<String,String> kafkaProducer = new
KafkaProducer<String, String>(prop);
        for (int i = 0; i < 100; i++) {
//            Future<RecordMetadata> future =

                    kafkaProducer.send(new ProducerRecord<>("test-topic",
"hell world" +i ),((recordMetadata, e) -> {
```

```
                    //判断发送是否成功
                if(e == null){
                    String topic = recordMetadata.topic();
                    int partition = recordMetadata.partition();
                    long offset = recordMetadata.offset();
                    log.info("topic: {},partition:{},offset:
{}",topic,partition,offset);
                }else{
                    e.printStackTrace();
                }
            }));
//            future.get();
            log.info("第"+i+"条消息");
        }
        kafkaProducer.close();
```

分区策略

1. 轮询分区策略（默认策略）
    - 最大限度保证消息平均分配到一个分区
    - 如果在生产key为null的数据时，使用轮询算法均衡的分配分区
    - 使用轮询算法没有办法保证消息的有序性（只能保证在分区中的局部有序）
2. 随机分区策略（目前没有使用）
3. 按key分区分配策略
    - 当key不为null时，按key的hash来进行分区，可能出现数据倾斜，某个key中可能包含大量的数据。
    - 按key存储可以实现一定程度上的有序存储（局部有效存储），实际生产环境中需要结合实际情况来做取舍
4. 自定义分区策略
    - 实现partitoner接口自定义分区

生产者如何提高吞吐量

- batch.size 默认16k修改批次大小， 36k
- linger.ms 默认0ms，修改发送等待时间 5-100ms
- compresson.type 压缩，使用snappy
- RecordAccumaulator :设置缓存区大小，修改为64m

```
 public static void main(String[] args) {
        Properties prop = new Properties();

prop.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
        //应答种类
        prop.put(ProducerConfig.ACKS_CONFIG,"all");
```

```
        //序列化
        prop.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

prop.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,StringSerializer.cla
ss.getName());

        prop.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,true);
        //设置缓冲区大小,以字节为单位
        prop.put(ProducerConfig.BUFFER_MEMORY_CONFIG,33554432);
        //设置批次大小，以字节为单位
        prop.put(ProducerConfig.BATCH_SIZE_CONFIG,16386);
        //设置linger.ms，以ms为单位
        prop.put(ProducerConfig.LINGER_MS_CONFIG,5);
        //设置压缩
        prop.put(ProducerConfig.COMPRESSION_TYPE_CONFIG,"snappy");
        KafkaProducer<String,String> producer = new KafkaProducer<>(prop);
        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<>("test-topic","hello world
custom send properites "+ i));
        }
        producer.close();
    }
```

发送数据的可靠性

kafka只能对已提交的消息做有限度的持久化保证 可靠性最佳实践 需要考虑producer、broker、comsumer三方的协调

1. 在producer端一般是使用异步的send，必须使用带回调函数的send来确认消息发送是否成功
2. producer端设置acks=all(表示所有副本broker都已经接收到消息，最高等级的消息提交要求) 使用ack应答机制来保证数据可靠性

> -1/all 所有的副本都写入成功才算写入成功，性能最差 0 不等待broker确认，直接发送下一条数据，性能最高，可能存在数据丢失，不能在生产中使用 1 等待leader副本确认接受后，才会发送下一条数据，性能中等 当ack为all时，考虑这种情况： leader收到数据后，还未同步数据给follower，有一个follower出现了宕机无法与leader通信，此时这条数据是不是一定就发送失败了？ leader通过维护动态的in-sync replica set （ISR），指和leader保持同步的follower和leader的集合（ 示例： leader： 0，isr：0，1，2） 如果follower 长时间未向leader发送通信请求或同步数据，则该follower将被踢出ISR，时间阈值配置：

如果分区副本设置为1，或者ISR中应答的最小副本数量（min.insyc.replica默认为1）设置为1，和ack=1的效果是一样的，仍会有丢失数据的风险 4. 设置producer端的retries为一个较大值，出现网络抖动导致的消息发送失败可以通过producer端的自动重试实现 5. 设置unclean.leader.election.enable = false，控制哪些broker有资格竞选分区的leader，broker落后leader太多时不允许有选举资格 6. 设置replication.factor >=3 保证有足够的副本冗余 7. 设置min.insync.replicas > 1 控制消息至少被写入多少个副本才算已提交 8. 保证reolication.factor > min.insync.replicas,保证集群可用性 9. consumer端设置enable.auto.commmit为false，采用手动提交位移的方式

数据完全可靠的条件

- ACK的级别设置为-1/all
- 分区副本大于等于2
- ISR里应答的最小副本数量大于等于2

生产环境中，如果数据丢失可接受，可以使用ack =1，如果不允许数据丢失，需要设置ack = -1

```
//应答种类,默认就是all
prop.put(ProducerConfig.ACKS_CONFIG,"all");
prop.put(ProducerConfig.ACKS_CONFIG,"0");
//设置producer重试次数，默认为Integer的最大值
prop.put(ProducerConfig.RETRIES_CONFIG,10);
```

发送数据的重复问题

kafka通过幂等性和事务来保证消息精确一次发送

- 幂等性保证： 在producer端通过设置enable.idempotence参数为true来保证消息的完全 幂等性只能保证单分区上的幂等，无法实现多分区的幂等，只能单会话幂等，不能实现跨会话的幂等
- 事务： 为producer端设置transsactional.id来开启事务

```
producer.initTransactions();
try {
            producer.beginTransaction();
            producer.send(record1);
            producer.send(record2);
            producer.commitTransaction();
} catch (KafkaException e) {
            producer.abortTransaction();
}
```

consumer设置 参数isolation.level, 有两个选择： read_uncommitted: 默认值，读取未提交消息
read_committed: 读取成功提交事务写入的消息

事务能保证跨分区、跨会话减的幂等性，另外，事务型的producer性能会差一点

ack = -1 情况下，可能出现数据重复问题：

> leader接收到数据，且已经同步给follower，在应答时leader宕机，这时集群中发生重新选举，producer没有接受到应答，尝试重新发送数据，新的leader会再次同步发送的数据给follower，出现数据重复的问题

数据传递语义：

- 至少一次 at least once： ACK的级别设置为-1/all，分区副本大于等于2，ISR里应答的最小副本数量大于等于2 ---存在重复发送的问题
- 最多一次 at most once： ack级别设置为0 ---可能丢数据
- 精确一次 exactly once： 幂等性 + 至少一次

重复判断标准：具有<PID,Partition,SeqNumber>相同主键的消息提交时，broker只会持久化一条数据，其中PID是kafka producer每次重启时分配的pid，Partition是分区号，SeqNumber 是单调自增的标志 幂等性只能保证一个会话单分区内不重复 幂等性的开启

```
//幂等性，默认开启
prop.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,true);
```

gpt对生产者消息重复的回答： Kafka生产者消息重复问题通常是由于多种原因引起的，下面是一些可能的解决方法和建议： 消息幂等性：确保你的生产者代码实现了消息的幂等性。这意味着无论消息发送多少次，最终的效果都是一样的。你可以为每个消息分配一个唯一的消息ID，并在消费者端维护一个状态，以确保不会处理重复消息。 配置重试机制：Kafka生产者在发送消息时具有重试机制。你可以配置生产者以在发送失败时自动重试。确保你正确配置了重试次数和重试间隔，以防止过多的重复消息。 使用事务：Kafka提供了事务支持。如果你的应用程序需要更严格的消息处理保证，可以使用事务来确保消息在发送和确认之间不会丢失或重复。 确认消息：确保消费者正确确认了消息的处理，以避免在消息处理失败时再次处理相同的消息。消费者可以使用Kafka的auto.commit属性来自动提交确认，或者手动提交确认。 消息重复检测：在消费者端，你可以实现消息重复检测机制，以检测和丢弃重复的消息。你可以使用数据库或缓存来存储已处理消息的信息，以便进行比对。 消息去重：如果消息重复问题仍然存在，你可以考虑实现消息去重机制，以便在消费者端消除重复消息。这可以通过缓存已处理消息的ID或内容来实现。 监控和日志：设置监控和日志记录以便及时发现消息重复问题，并快速采取纠正措施。监控可以帮助你识别生产者和消费者的性能问题。 升级Kafka版本：某些版本的Kafka可能会存在已知的问题，可能导致消息重复。升级到最新版本可以帮助解决这些问题。 网络稳定性：确保Kafka集群和生产者、消费者之间的网络连接是稳定的。网络问题可能导致消息重复。

> 当出现多会话（多个producer时），需要使用kakfa的事务，开启事务必须开启幂等性 producer在使用事务之前，必须先自定义一个唯一的transaction_id，这样即使客户端挂掉，它重启后也能继续处理未完成的事务 事务流程：

- producer 请求producer id（幂等性需要），向broker的事务协调器（transaction coordinator）
- 事务处理器发送producer_id
- producer发送消息到broker
- producer发送commit请求
- broker持久化commit请求
- 事务协调器发送commit请求到topic的leader partition
- leader 返回成功，持久化事物成功信息到特殊的分区主题

事务的使用

```
   public static void main(String[] args) throws ExecutionException,
InterruptedException {
        Properties prop = new Properties();

prop.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
        //应答种类
        prop.put(ProducerConfig.ACKS_CONFIG,"all");
        prop.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

```
prop.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,StringSerializer.cla
ss.getName());
        prop.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,true);
        //设置全局事务唯一id

prop.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,"transaction_id_test_01");
        KafkaProducer<String,String> kafkaProducer = new
KafkaProducer<String, String>(prop);
        //开启事务
        kafkaProducer.initTransactions();
        //启动事务
        kafkaProducer.beginTransaction();
        try {

            for (int i = 0; i < 10; i++) {
//              Future<RecordMetadata> future =
                kafkaProducer.send(
                        new ProducerRecord<>("test-topic", null, "hello
world,"+ i));
//              future.get();
                log.info("第"+i+"条消息");
            }
            //提交事务
            kafkaProducer.commitTransaction();
        }catch (Exception e){
            //回滚事务
            kafkaProducer.abortTransaction();
        }finally {
            kafkaProducer.close();
        }
    }
```

发送消息的有序性

由于topic分区，无法保证整个topic范围内的数据有序，只能保证单分区内有序（有条件的有序）

如何保证多分区有序： 由consumer端取到所有的分区数据，在consumer端进行排序

生产者默认每个broker最多缓存5个请求，由于可能存在数据1，2，4发送成功，而数据3发送失败需要进行重试，会导致数据乱序 kafka 1.x版本之前，将max.in.flight.requests.per.connection=1，保证缓存请求最多是一个，不会出现乱序 之后的版本，如果开启了幂等性，max.in.flight.requests.per.connection需要设置为小于等于5，可以保证不会乱序 当出现上面的数据发送重试时，在kafka集群中会在持久化数据前根据幂等性条件<PID,Partition,SeqNumber>判断前面是否有数据未接收，如果有，则等待缺失数据到达再进行排序后持久化

kafka broker 服务器

zookeeper中的存储的kafka数据

在/broker/topic 节点中存储的时节点的topic、分区、state（leader、ISR信息） 在/broker/ids存储的是哪些服务器在集群中 在/consumer 中存储的消费者的信息（最重要的是offset信息） 在/controller存储辅助选举主节

点的信息

总体工作流程

- kafka启动后向zookeeper进行注册
- 开始选举，broker中的conroller通过抢先向zookeeper的controller中注册自己，最先注册的controller会成为controller leader
- controller leader会首先监听zookeeper中的/broker/ids节点信息
- controller leader按这种规则选举partition leader：在ISR中存活为前提，按照AR（kafka分区中副本的统称）中排在前面的优先，controller leader按AR的顺序进行轮询，排在第一位的就是partition leader
- 将选举结果写入到zookeeper的树节点下，通知其他节点，其他节点副本从leader同步数据
- 生产者发送数据，任意broker接受到数据后，首先会和partition leader同步数据，在持久化方面，使用segment（.log，默认1g）存储数据，使用.index文件加快从.log文件查询数据 的速度
- 当partition leader的broker宕机后，会重新按照前面的分区leader选举规则进行再次选举，从而更新ISR、AR信息

kafka节点的服役和退役

- 服役 设置broker.id 配置zookeeper服务地址（不需要再单独启动zookeeper服务） 查询zookeeper上是否已经存在了响应的broker信息 历史数据迁移（进行负载均衡）： 创建topic-to-move.json

```
{
    "topic":[
        {"topic":"test-topic"}
    ]
}
```

执行计划迁移命令

```
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
topic-to-move-json-file topic-to-move.json --broker-list "0,1,2,3" --
generate
```

将计划信息拷贝放入json中

```
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
reassignment-json-file  increase-replication-factor.json --broker-list
"0,1,2,3" --generate
```

- 退役(首先考虑数据的安全性)

> 将数据导入其他节点上（利用上面的kafka-reassign-partitions.sh脚本） 切走要退役节点的流量 执行要退役节点的kafka-server-stop.sh

```
#将broker ID为3的节点排除
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
topic-to-move-json-file topic-to-move.json --broker-list "0,1,2," --
generate
```

执行计划

```
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
reassignment-json-file  increase-replication-factor.json --broker-list
"0,1,2,3" --generate
```

kafka的副本

提高数据可靠性 默认副本数量是1个，生产环境一般配置2个，保证数据可靠性 kafka中的副本分为leader和follower，生产者将数据发送给leader，然后follower找到leader进行同步数据 kafka中生产者消费者读写数据的对象只能是leader，follower只做数据备份 kafka中分区中所有的副本统称为AR（assigned replicas） AR = ISR（in-sync-replica 和leader保持同步的follower集合）+OSR（out-sync-replica 标识follower与leader副本同步是，延迟过多的副本）

leader选举

集群中每启动一个broker就会向zookeeper中注册/brokers/ids, leader controller监听/brokers/ids节点变化 leader controller 选举分区leader，在ISR中存活为前提，从AR中按照顺序选择排在前面的副本作为leader 当某个broker挂掉后，controller leader监控到节点宕机信息，ISR、AR 更新，会更新/brokers/topics/first/parittion，再次按前面的规则进行选举

follower故障处理

LEO （log end offset）：每个副本的最后一个offset，LEO是最新的offset+1（数组的下标从0开始） HW（high watermark）： 所有副本最小的leo，消费者能看到的offset是hw 当follower故障：

> follower会被从ISR中临时踢出 这期间leader和follower继续接收数据 当故障follower恢复后，follower会读取本地磁盘记录的上次的hw，并将log文件高于hw的部分截取掉，从hw开始向leader进行同步 当该follower的leo大雨等于该partition 的hw，即follower追上leader之后，就可以冲刺你加入ISR

leader故障处理

> leader发生故障后，会从ISR中选举出一个新的leader 为保证多个副本之间的一致性，其余follower会先将各自的log文件高于hw的部分截取掉，然后从新的leader处同步数据 只能保证副本之间的数据一致性，不能保证数据丢失或不重复

分区副本配置

考虑负载均衡和数据的安全性，保证分区leader尽量均匀分布在集群中的不同节点上 **手动调整分区副本** 考虑4分区两个副本的情况，需要按照服务器新能分配数据存储的位置 使用命令来实现分区副本的调整

```

```

```
#执行命令
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
reassignment-json-file  increase-replication-factor.json  --execute
#验证上面的execute是否执行完毕
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
reassignment-json-file  increase-replication-factor.json  --verify
```

leader partition负载均衡

保证partition均匀分布在各个机器上 可能由于集群中的节点故障导致partition分布不均匀 可以使用leader.imbalance.per.broker.percentage=10%,当broker中的leader不均匀比例超过10%时会出发leader的均衡机制 均衡机制检测时间配置leader.imbalance.check.interval.second=300,默认300s **一般情况下不建议开启再均衡机制**

增加副本因子

topic在创建以后无法通过--alter来修改topic分区副本数量 需要手动进行副本的数量修改

```
./bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --
reassignment-json-file  increase-replication-factor.json  --execute
```

数据的文件存储

topic是逻辑上的概念，而分区partition是物理概念 每个partition对应一个log文件（虚拟概念），log中的文件是以segment来存储（默认1g），一个partition进行分片，分为多个segment 每个segment包含.log文件和.index文件（用来索引）和.timeindex(判断日志是否需要进行删除) producer的数据会不断追加到segment的末尾，不会进行修改 .index文件 **稀疏索引**，每4KB文件才会记录一条索引，存储相对offset，即segment的offset，避免索引数据过大 可以使用kafka 的命令查看具体的.log文件信息

```
./bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files
../../data/kafka/kafka-data1/test-topic-0/00000000000000000000.log
```

数据文件清除策略

相关参数配置： log.retention.hours 最低优先级小时 默认7天 log.retention.minutes 分钟 log.retention.ms 最高优先级ms log.retention.check.interval.ms 负责周期检查，默认5分钟检查一次

数据清除策略 log.cleanup.policy = delete/compact 默认删除 基于时间的删除：默认打开，以segment中所有记录中的最大时间戳作为该文件的计算删除的时间戳 基于大小的删除：log.retention.bytes=-1,默认关闭，超过设置的所有日志的总大小，删除最早的segment

compact：对与相同的key不同的value只保留最后一个版本，适合用户信息的迭代更新等场景

高效读写数据

> kafka自身是分布式集群，采用分区技术，并行度高 读取数据采用稀疏索引，可以快读定位到要消费的数据 顺序读写磁盘，producer写入过程是追加数据到log文件末端，顺序写能达到600M/s,省去磁盘寻址的空间 采用页缓存和零拷贝技术（借用linux系统内核的页缓存）

kafka消费者

kafka的消费者采用的主动拉取pull数据的模式， 由于采用推送push数据的方式由broker来决定消息的发送速率，使用主动pull数据避免由于未知消费者消费数据的能力而导致推送数据速率和消费速率不匹配导致的问题

消费者工作流程

消费者组：每个分区中的数据只能由消费者组中的一个消费者消费，避免重复消费 消费记录：使用offset记录，老板本的消费者的offset数据存储在zookeeper上，新版本的维护在系统的__consumer_offset topic上 由于zookeeper的性能问题，如果所有的消费者offset数据全部存储在zookeeper上，broker需要从zk上取offset，会产生大量的向zk的请求

消费者组 由多个comsumer组成，组内所有的消费者的groupid相同

- 消费者组内的每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费
- 消费者组之间互不影响，所有的消费者都属于某个消费者组，即消费者是逻辑上的一个订阅者
- 当组内消费者大于分区数量时，会出现闲置消费者的情况，闲置消费者不会消费任何消息

消费者组初始化流程

coordinator： 辅助实现消费者组的初始化和分区配置，每个broker节点上都有一个用来处理消费者offset的数据 coordinator 节点选择 = groupid的hash值%50（__consumer_offsets的分区数量）

> 消费者组中的consumer都发送joinGroup请求 coordinator 收到请求之后，选择一个consumer作为leader 作为leader的consumer制定消费计划，哪个消费者消费哪些分区 消费者定期向coordinator发送心跳确认消费者存活，45s内（session.timeout.ms）未收到消费者心跳，则认为消费者已下线，会触发消费者消费消息再均衡 当消费者处理消息时间过程大于max.poll.interval.mx（默认5分钟），也会触发再均衡

消费者消费流程

> 消费者创建消费者网络连接客户端（ConsumerNetworkClient）,发送消费请求 根据每批次抓起字节数（fetch.min.bytes），默认抓去一个字节，来抓去数据 当字节数未达到上面指定的值，而抓取时间已经到达fetch.max.wait.ms（默认500ms），则也会进行数据的拉去 fetch.max.bytes 每批次最大抓取大小，默认50M max.poll.records指定拉取后返回给客户端的批次条数，默认500条 拉取到数据后，会对数据进行反序列化 然后进过拦截器，监控消费数据 最后消费者处理数据

消费者代码

```
//消费者开发,订阅主题
    public static void main(String[] args) {
        Properties prop = new Properties();

prop.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
        //设置消费者组
        prop.put(ConsumerConfig.GROUP_ID_CONFIG,"test-topic-group");
```

```
        //自动提交offeset
        prop.put("enable.auto.commit","true");
        //自动提交时间间隔
        prop.put("auto.commit.interval.ms","1000");

prop.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.co
mmon.serialization.StringDeserializer");

prop.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.
common.serialization.StringDeserializer");
        KafkaConsumer<String,String> consumer = new KafkaConsumer<String,
String>(prop);
        final ArrayList<String> topic = new ArrayList<>();
        topic.add("test-topic");
        consumer.subscribe(topic);
        //开始消费
        while (true){
            ConsumerRecords<String, String> poll =
consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String,String> record : poll) {
                log.info("offset:{},topic:{},partition:{},key:{},value:
{}",

record.offset(),record.topic(),record.partition(),record.key(),record.valu
e());
            }
        }
    }
```

订阅topic并指定分区

```
public static void main(String[] args) {
        Properties prop = new Properties();

prop.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
        //设置消费者组，组名一样的消费者消费的消息是一样的
        prop.put(ConsumerConfig.GROUP_ID_CONFIG,"test-topic-group");
        //自动提交offeset
        prop.put("enable.auto.commit","true");
        //自动提交时间间隔
        prop.put("auto.commit.interval.ms","1000");
        //反序列化

prop.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.co
mmon.serialization.StringDeserializer");

prop.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.
common.serialization.StringDeserializer");
        KafkaConsumer<String,String> consumer = new KafkaConsumer<String,
String>(prop);
        //  订阅topic并指定分区
        final ArrayList<TopicPartition> topicPartitinos = new ArrayList<>
```

```
();
        topicPartitinos.add(new TopicPartition("first",0));
        consumer.assign(topicPartitinos);
        //开始消费
        while (true){

            ConsumerRecords<String, String> poll =
consumer.poll(Duration.ofSeconds(1));


            for (ConsumerRecord<String,String> record : poll) {
                log.info("offset:{},topic:{},partition:{},key:{},value:
{}",

record.offset(),record.topic(),record.partition(),record.key(),record.valu
e());
            }
        }
    }
```

每个分区到底应该由哪个消费者消费?

消费者的分区分配策略及再均衡

- range范围分配策略，确保每个消费者消费的分区数量是均衡的。range范围分配是针对每个topic的。按消费者字母顺序对消费者排序，按分区号对分区进行排序 使用以下计算公司进行分区分配 公式： n = 分区数量/消费者数量 m = 分区数量%消费者数量 前m个消费者分别消费n+1个分区， 剩余的消费者分别消费n个 range方式存在的问题：

> 当消费组同时订阅多个topic时，可能会导致consumer0消费的数据很大，出现数据倾斜 当某个消费者退出时，它所消费的分区在这45s内接收到的数据会被全部分配给某一个消费者，45s后触发再均衡，然后再重新按上面公式进行分区分配

- RoundRobin轮训策略 将消费组内所有的消费者以及消费者所订阅的所有topic的partition按照字典顺序排序（topic和分区的hashcode进行排序） 通过轮询方式逐个将分区以这种分配分给每个消费者

> 当某个消费者退出时，45s后它所消费的分区会按照hash+轮询的方式发送给其他消费者

- stricky黏性分配 分区分配尽可能均匀 分配的结果带有粘性，在执行新的分配之前，考虑上一次分配结果，尽量少的调整分配的变动，节省开销

> 当某个消费者退出时，触发再均衡，会将退出消费者的分区尽量均匀的分配给其他消费者
> 在发生rebalance时，走一遍轮询策略，分区的分配尽可能与上一次保持相同，仅将出现变化的分区进行重新分配， 没有发生rebalance时，与轮询分配策略保持一致

- cooperativeSticky 合作者粘性

默认使用range+cooperativesticky两种策略

```
//修改分区分配策略
//指定分区分配策略,策略RoundRobin
```

```
        prop.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
RoundRobinAssignor.class.getName());
        // Sticky
      //  prop.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
StickyAssignor.class.getName());
        //Range
        //prop.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
RangeAssignor.class.getName());
        //CooperativeSticky
        //prop.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
CooperativeStickyAssignor.class.getName());
```

自动offset提交机制

两个参数

- enable.auto.commit: 是否开启自动提交offset功能，默认true
- auto.commit.interval.ms： 自动提交offset的时间间隔，默认5s

```
            //自动提交offeset
//        prop.put("enable.auto.commit","true");
        prop.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,"true");
        //自动提交时间间隔
    //    prop.put("auto.commit.interval.ms","1000");
        prop.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG,"1000");
```

存储位置
0.9版本之后offset存储在__consumer_offset的系统topic中，其中的key为group.id+topic+分区号 由于存储在zk中会受限于zk的性能，导致集群性能受到限制 修改consumer.properties配置允许查看系统主题 exclude.internal.topics = false (默认为true，表示不允许查看系统主题数据)

手动提交

同步提交：必须等待offset提交完毕之后才去消费下一批数据

```
   Properties prop = new Properties();

prop.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
        //设置消费者组，组名一样的消费者消费的消息是一样的
        prop.put(ConsumerConfig.GROUP_ID_CONFIG,"test-topic-group");
        //关闭自动提交offeset
        prop.put("enable.auto.commit","false");

        //自动提交时间间隔
//        prop.put("auto.commit.interval.ms","1000");
        //反序列化

prop.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.co
mmon.serialization.StringDeserializer");
```

```
prop.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.
common.serialization.StringDeserializer");
        KafkaConsumer<String,String> consumer = new KafkaConsumer<String,
String>(prop);
        //订阅topic
        final ArrayList<String> topic = new ArrayList<>();
        topic.add("test-topic");
        consumer.subscribe(topic);
        //开始消费
        while (true){

            ConsumerRecords<String, String> poll =
consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String,String> record : poll) {
                log.info("offset:{},topic:{},partition:{},key:{},value:
{}",

record.offset(),record.topic(),record.partition(),record.key(),record.valu
e());
            }
            //手动同步提交
            consumer.commitSync();
        }
    }
```

异步提交：发送完offset请求后就开始消费下一批数据

```
            //异步提交
            consumer.commitAsync();
```

指定offset开始消费 seek消费

消费者参数 auto.offset.reset = earliest|latest|none earliest:从头开始消费，即命令行中的 --from-beginning latest(默认值):自动将偏移量重置为最新偏移量 none：如果未找到消费者组的先前偏移量，则抛出异常

```
  public static void main(String[] args) {
        Properties prop = new Properties();

prop.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
        //设置消费者组，组名一样的消费者消费的消息是一样的
        prop.put(ConsumerConfig.GROUP_ID_CONFIG,"test-topic-group");
        //关闭自动提交offeset
        prop.put("enable.auto.commit","false");

        //自动提交时间间隔
//        prop.put("auto.commit.interval.ms","1000");
        //反序列化
```

```
prop.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.co
mmon.serialization.StringDeserializer");

prop.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.
common.serialization.StringDeserializer");
        KafkaConsumer<String,String> consumer = new KafkaConsumer<String,
String>(prop);
        //订阅topic
        final ArrayList<String> topic = new ArrayList<>();
        topic.add("test-topic");
        consumer.subscribe(topic);
        //指定位置进行消费
        final Set<TopicPartition> assignment = consumer.assignment();
        //防止分区方案可能还未进行分配情况，先进行数据拉去分配分区，保证分区分配方案已经
指定完毕
        while (assignment.size() == 0){
            consumer.poll(Duration.ofSeconds(1));
            assignment =consumer.assignment();
        }
        assignment.forEach(ass ->{
            //指定每个分区都从从offset的位置开始消费
            consumer.seek(ass,10);
        });
        //开始消费
        while (true){
            ConsumerRecords<String, String> poll =
consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String,String> record : poll) {
                log.info("offset:{},topic:{},partition:{},key:{},value:
{}",

record.offset(),record.topic(),record.partition(),record.key(),record.valu
e());
            }
            //手动同步提交
//            consumer.commitSync();
            //异步提交
            consumer.commitAsync();
        }
    }
```

按指定时间进行消费

思想：将时间点转化为offset，然后指定该offset进行消费

```
  public static void main(String[] args) {
        Properties prop = new Properties();
        prop.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        //设置消费者组，组名一样的消费者消费的消息是一样的
        prop.put(ConsumerConfig.GROUP_ID_CONFIG, "test-topic-group1");
```

```
        //关闭自动提交offeset
        prop.put("enable.auto.commit", "false");

        //自动提交时间间隔
//        prop.put("auto.commit.interval.ms","1000");
        //反序列化
        prop.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        prop.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(prop);
        //订阅topic
        final ArrayList<String> topic = new ArrayList<>();
        topic.add("test-topic");
        consumer.subscribe(topic);
        //指定位置进行消费
        Set<TopicPartition> assignment = consumer.assignment();
        //防止分区方案可能还未进行分配情况，先进行数据拉去分配分区，保证分区分配方案已经
指定完毕
        while (assignment.size() == 0) {
            consumer.poll(Duration.ofSeconds(1));
            assignment = consumer.assignment();
        }
        //取一天之内的消息数据
        HashMap<TopicPartition, Long> topicPartitionLongHashMap = new
HashMap<>();
        for (TopicPartition partition : assignment) {
            //取一天前的offset位置

topicPartitionLongHashMap.put(partition,System.currentTimeMillis()-24*3600
*1000);
        }
        //将时间转为offset
        Map<TopicPartition, OffsetAndTimestamp>
topicPartitionOffsetAndTimestampMap =
consumer.offsetsForTimes(topicPartitionLongHashMap);
        assignment.forEach(partition -> {
            //指定从指定的每个分区从offset的位置开始消费
            consumer.seek(partition,
topicPartitionOffsetAndTimestampMap.get(partition).offset());
        });
        //开始消费
        while (true) {
            ConsumerRecords<String, String> poll =
consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> record : poll) {
                log.info("offset:{},topic:{},partition:{},key:{},value:
{}",
                        record.offset(), record.topic(),
record.partition(), record.key(), record.value());
            }
            //手动同步提交
//            consumer.commitSync();
```

```
            //异步提交
            consumer.commitAsync();
        }
    }
```

漏消费与重复消费

重复消费

1. 由于自动提交，可能是多个消息消费之后做一次整体的提交，提交的offset是连续的已消费数据的最后一个offset，可能出现由于0，1，2，3，4数据成功收到，但是此时consumer挂掉，可能这一批数据的offset还未来得及提交导致下次重启还是会重新消费0，1，2，3，4

漏消费： 设置offset手动提交，数据已经被consumer取到，这时consumer已经提交了offset，但是数据还未做相应的逻辑处理，此时consumer挂掉，下次重启时会从offset处开始消费，导致数据漏消费

解决方案：使用事务，将kafka 的消费过程和提交offset过程做原子保定 可以使用mysql 的事务

数据积压

如何提高消费者吞吐量

1. 如果是消费者消费能力不足可以考虑增加分区，增加消费者的数量和CPU核心数（消费者数 = 分区数）
2. 如果是下游数据处理不及时，提高每批次拉取的数据量，可以修改一次拉取数据条数（默认500条）和最大的数量（默认50M）

kafka监控

使用kafka eagle实现监控

- 安装mysql
- 修改kakfa默认内存，从1G修改成2G，从kafka-server-start.sh中修改 -Xmx2G -Xms2G

```
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx2G -Xms2G"
fi
```

- 修改kafka eagle 配置文件system-config.properties

```
efak.zk.cluster.alias=cluster1
cluster1.zk.list=localhost:2181/kafka
cluster1.efak.offset.storage=kafka
```

- 配置kafka eagle环境变量

```

```

```
export KE_HOME=/Users/lijie3/Documents/tool-package/kafka-eagle-bin-
3.0.1/efak-web-3.0.1
export PATH=$PATH:$KE_HOME/bin
```

- 配置mysql连接信息

kafka kraft模式

> 2.8.0之后kafka可以不依赖zookeeper部署 配置文件在conf/kraft目录下 启动时需要进行初始化 ,每个节点都要执行

```
./bin/kafka-storage.sh random-uuid fsafsadf -c
config/kraft/server.properties
```

> 接着启动集群

```
./bin/kafka-server-start.sh config/kraft/server.properties
```

外部系统集成

集成flume

集成架构 flume作为生产者 日志文件 --》 flume（client ->memorychannel-> sink） --》 kafka

flume作为消费者 生产者 --》kafka --》 kafkasource--》memorychannel --》logger--》下游 flume :cloudera 开发的实时日志收集系统 一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。支持在日志系统中定制各类数据发送方，用于收集数据;同时，Flume提供对数据进行简单处理，并写到各种数据接受方(比如文本、HDFS、Hbase等)的能力 。flume的数据流由事件(Event)贯穿始终。事件是Flume的基本数据单位，它携带日志数据(字节数组形式)并且携带有头信息，这些Event由Agent外部的Source生成，当Source捕获事件后会进行特定的格式化，然后Source会把事件推入(单个或多个)Channel中。你可以把Channel看作是一个缓冲区，它将保存事件直到Sink处理完该事件。Sink负责持久化日志或者把事件推向另一个Source。 flume的概念

- Client：Client生产数据，运行在一个独立的线程。
- Event： 一个数据单元，消息头和消息体组成。（Events可以是日志记录、 avro 对象等。）
- Flow： Event从源点到达目的点的迁移的抽象。
- Agent： 一个独立的Flume进程，包含组件Source、 Channel、 Sink。（Agent使用JVM 运行Flume。每台机器运行一个agent，但是可以在一个agent中包含多个sources和sinks。）
- Source： 数据收集组件。（source从Client收集数据，传递给Channel）
- Channel： 中转Event的一个临时存储，保存由Source组件传递过来的Event。（Channel连接 sources 和 sinks ，这个有点像一个队列。）
- Sink： 从Channel中读取并移除Event， 将Event传递到FlowPipeline中的下一个Agent（如果有的话）（Sink从Channel收集数据，运行在一个独立线程。）

集成flink

集成架构 flink作为生产者

flink作为消费者

集成springboot

使用Spring 提供的KafkaTemplate<String,String>操作kafka

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    producer:
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.apache.kafka.common.serialization.StringSerializer
      acks: all
    consumer:
      group-id: test-topic-group
      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      enable-auto-commit: true
```

```
//生产者
   @Autowired
    KafkaTemplate<String,String> kafkaTemplate;

    @PostMapping("/send")
    public String send(String msg){
        kafkaTemplate.send("test-topic",msg);
        return "OK";
    }

//消费者
@Configuration
public class ConsumerController {

    @KafkaListener(topics = "test-topic")
    public void consumerTopic(String msg){

        System.out.println("receive msg :"+ msg);
    }
}
```

集成spark

spark作为生产者

spark作为消费者

kafka调优

总体：

1. 提高生产吞吐量
2. 增加分区
3. 提高消费者吞吐量
4. 数据精准一次（生产者角度 事务+幂等性 + broker角度（分区副本数大于2，ISR副本数大于2）+ 消费者角度（事务+手动提交，消费者输出的目的地必须支持事务）
5. 合理设置分区数 分区数一般设置为3-10个
6. 对与单条日志大于1M， kafka broker默认处理单条数据（message.max.bytes = 1M）可以改成更大10M 生产者发到broker每个请求消息的最大值 ，默认max.request.size = 1M 副本同步数据，每个批次消息最大值 默认reolica.fetch.max.bytes = 1M 消费者获取一批消息最大的字节数，fetch.max.bytes = 50M ，大于50M依然可以拉取回来，broker端的一批次大小受message.max.bytes或max.message.bytes的影响
7. 服务器挂了的排查（重启、查看内存情况、cpu情况、网络带宽）

硬件配置选择

100万日活，每人每天100条日志=1亿条日志 处理日志的速度： 1亿/24/3600s = 1150条/s 1条日志（0.5k-2k）,按1k计算 1150 * 1k/s = 1M/s 高峰值： 中午小高峰，晚上8-12 ，按20倍算 20M/s-40M/s

1. 购买多少服务器： 服务器台数 = 2* （生产者峰值生产速率 * 副本数量/100） + 1 2 *（202*/100）+1 一般选择3台
2. 磁盘选择 kafka 按照顺序读写，选择机械硬盘和固态硬盘的读写速度差不多 3台服务器大于1t硬盘
3. 内存选择 kakfa 内存 = 堆内存（kafka内部配置） + 页缓存（服务器内存） = 10G~15G + 1G = 15G
4. cpu选择 负责写磁盘的线程数（num.io.threads=8） 负责写磁盘线程数 副本拉取线程数据 num.replica.fetcher = 1 数据传输线程数 num.network.threads = 3 建议32个cpu核心
5. 网络 选择千兆网卡，选择百兆带宽明月12.5M/s

生产者调优

主要是对生产者的配置参数进行配置 batch.size 只有数据累计到指定大小才会发送，默认16k buffer.memory 缓冲区大小默认32M 如果要保证所有分区的有序性，建议在topic只做一个分区，或是在consumer进行排序

消费者调优

broker调优

对broker的各种参数进行调优 禁止自动创建主题(修改配置参数auto.create.topic.enable = false)

压力测试

压力测试工具： kafka-producer-perf-test.sh

```
./bin/kafka-producer-perf-test.sh --topic test-topic --record-size 1024 --
num-records 1000000  --throughput 1000 --producer-props
bootstrap.servers=localhost:9092 batch.size=16384  linger.ms=0
compression.type=snappy buffer.memory=67108864

--record-size 1024  (一条数据的大小1M )
--num-records 1000000  (测试数据量)
--throughput 1000  (吞吐量)
batch.size=16384 (一批数据量16k)
linger.ms=0 数据发送不延迟
compression.type=snappy/zstd/gzip/lz4 压缩方式
buffer.memory=67108864 缓冲区大小，64M
```

kafka-consumer-perf-test.sh

```
./bin/kafka-producer-perf-test.sh --bootstrap.server=localhost:9092 --
topic test-topic --messages 100000 --consumer.config
config/consumer.properties
max.poll.records = 500 --默认拉取最大条数默认500
fetch.max.bytes = 50M --每批次抓取数据大小默认 50M
```

producer的ACKs参数

```
 prop.put("acks","all");
```

ACKs对副本的影响较大

根据具体的业务场景选择不同的应答机制

为确保小粉着消费的数据是一致的，只能从分区leader读写消息，follower分区只负责同步数据，做热备份

高级API（Higher API）和低级API（lower API）

> 以上用的代码都是高级api

- 不需要去执行offset，直接通过zk管理，不需要管理分区，副本，由kafka统一管理
- 消费者会自动根据上一次在zk中保存的offset去接着获取数据
- 在zk中，不同的消费者组同一个topic记录着不同的offset，这样不同的程序读取同一个topic不回收到offset的影响 高级api不能控制offset，无法从指定位置读取 低级api会在各种框架中进行使用，有编写的程序自己控制逻辑，自己管理offset，将offset存储在zk、mysql、redis、hbase、flink的状态存储

手动指定分区消费，无法进行rebalance

```
 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(prop);
        TopicPartition partition0 = new TopicPartition("quickstart-
events",0);
```

```
        TopicPartition partition1 = new TopicPartition("quickstart-
events",0);
        consumer.assign(Arrays.asList(partition0,partition1));
```

kafka eagle 监控工具

实现原理

leader、follower

> leader follower是针对分区,不是针对broker 每个分区必须有一个leader，有0个或多个follower 创建topic是kafka会尽量均匀的将topic的分区分配到broker上，尽量让leader分配均匀 当leader发生故障时，其他follower会被重新选举为leader follower只会从leader同步数据，不对客户端提供读写功能 从配置文件指定的路径`log.dirs`能看到具体的kafka数据信息，可以看到索引、topic、分区等信息

AR\ISR\OSR --follower的状态

leader出现故障后，通过投票进行选举，根据follower的状态进行选举

- AR （Assigned Replicas） 已分配副本，分区的所有副本称为AR
- ISR (in sync replicas) 所有与leader副本保持一定程度同步的副本，在同步中的副本
- OSR （out sync replicas） 由于follower副本同步滞后过多的副本组成OSR

正常情况下，所有副本都处于同步状态，即AR = ISR，OSR为空 可以使用kafka eagle 查看所有ISR副本

Controller与leader选举

kafka如何确定某个partition为leader，哪个时follower?

> 在开始启动时时随机选择partition为leader 某个leader崩溃了，如何快速确定另一个leader?

controller kafka在启动时会在所有的broker中选择controller,contoller是针对controller 创建topic、添加分区、修改副本数量都是由controller来实现的 controller选举

> 集群启动时，每个broker都会尝试去zk上注册成为controller 只要有一个注册成功，则其他的broker会注册该节点的监视器 一旦该节点发生变化，就可以进行相应的处理 controller是高可用的，一旦broker崩溃，其他的broker就会重新注册为controller controller是通过zk来进行选举的 **controller选举leader** 所有的partition的leader选举都是由controller来决定的 controller会将leader的改变直接通过rpc方式通知需为此做出相应的broker controller 读取到当前分区的ISR，只要有一个replica还幸存，就选择其中一个作为leader，否则任意选择一个replica作为leader 如果该partition的所有replica都已经宕机，则新的leader为-1（标识当前分区挂掉）

为什么不用zk的方式选取leader？ kafka集群如果业务很多的情况下，会有很多个partition 假设某个broker宕机，就会出现很多partition需要重新选举leader 如果使用zk选举leader，回给zk带来很大的压力，所以leader选举不能用zk来实现

leader 的负载均衡 preferred replica

在ISR列表中第一个replica就是preferred replica 第一个分区存放的borker，就肯定时preferred replica

> 如果某个broker宕机滞后，就可能导致partition的leader分布不均匀，broker上存在一个topic下不同的partition的leader leader不是均匀分布在某台broker上（一个broker上有多个leader），则这台broker就不是preferred replica 通过执行`bin/kafka-leader-election.sh`命令对leader进行重新选举，确保leader是均匀分配的

kafka的读写数据流程

写入

> 从zk上的'/brokers/topics/主题名/partitions/分区名/state'节点找到partition的leader 生产者在zk中找到Id对应的broker broker进程上的leader将消息写入本地的log中（顺序写，速度快） follower从leader上拉取消息，写入本地log，并leader发送ACK leader接受到所有的ISR中的replica的ACK后，并向生产者返回ACK

读取（消费）

> kafka采取的是拉模式 由消费者自己记录消费状态，每个消费者互相独立的顺序拉取每个分区的消息 消费者可以按照人一的顺序消费消息，比如可以重置偏移量 每个consumer都可以根据分配策略（默认range），获得要消费的分区，通过zk找到分区对应的leader（leader负责读） 获取到consumer对应的offset，默认从zk中获取上一次消费的offset 找到改分区的leader，拉取数据 消费者提交offset

kafka的数据存储

> 一个topic有多个分区组成 一个分区由多个segment（段）组层（默认IG进行滚动） 一个segment（段由多个文件组成（log、index、timeindex））

存储日志 写日志

> .log 日志数据文件 .index 索引文件（使用稀疏索引，避免索引数据量过大） .timindex 时间索引文件 leader-epoch-chekpoint 持久化每个partition leader对应的leo(log end offset)，leo日志文件中下一条待写入消息的offset 文件名是起始偏移量

读取数据

> 消费者使用offset（针对partition全局的offset）找到对应的segment 将offfset转化为segment段文件的offset 在根据segment段文件的局部offset查找segment段中的数据 为了提高查询效率，每个文件都会维护好对应的范围内存，找到的时候使用简单的二分法查找

删除消息 在kafka中，消息是会被定期清理，一次删除一个segment段的日志文件 kafka的日志管理器会根据kafka的配置，来决定哪些文件可以被删除

如何保证消息不丢失

> broker数据不丢失 生产者通过分区的leader写入数据后，所有在ISR中follower都糊从leader复制数据，这样可以确保即使leader崩溃了，其他的follower数据仍然不会丢失 生产者数据不丢失 生产者连接leader写入数据时，可以通过ACK机制来确保数据已经成功写入，ACK机制有三个选项可以配置

- -1/all 所有follower节点都要收到数据才发送ack
- 1 leader收到数据后响应
- 0 生产者只负责发送数据，不关心数据是否丢失 生产者采用同步和异步两种方式发送数据
- 同步：发送一批数据给kafka，等待kafka返回结果

- 异步：发送一批数据给kafka，只提供一个回调 生产者使用带有回调方法的send，保证发送消息丢失时能正常的进行重试处理 如果broker不给ack，而buffer又满了，开发者可以设置是否直接清空buffer中的数据

> 消费者数据不丢失

- 消费者在消费数据是，只要每个消费者记录好offset值即可，就能保证数据不丢失
- 消费者在拿到消息，要保证消息的正常处理完成之后，才将offset写回zk

> 消息重复消费

- 消费者从zk拿到offset从分区leader中消费消息
- 消费者处理完消息再将offset存储到zk
- offset写回zk的过程中失败导致offset没有更新，可能出现重复消费的情况

消息传递语义

at most once 最多一次消费 at least once 最少一次消费（可能出现重复消费） exactly once 仅有一次消费（事务性保证消息仅被处理一次） 通过lowerlevel API自己维护offset，将offset写入mysql（使用mysql事务保证整个操作的原子性，不能使用kafka的事务保证），不使用zk维护

数据积压

kafka消费者数据时由于外部IO、或者产生网络拥堵，就会造成kafka中的数据积压，如果数据一致积压，会导致数据的实时性受到影响 解决方案： 正常情况下kafka的消费者消费数据的速度是很快的，产生数据积压往往是因为消费者下游程序处理逻辑太慢导致的数据积压（不考虑网络的问题），处理好下游程序的逻辑提升效率就能解决数据积压的问题

日志清理

kafka消息存储在磁盘中，为了控制磁盘空间，Kafka每个分区都有很多日志文件，方便了清理 日志删除： 按照自定的策略直接删除不符合条件的日志 日志压缩：按照消息的key进行整合，相同的key具有不同的value值 设置保留策略

> 基于时间的保留策略(默认七天)

- log.retention.hours
- log.retention.minutes
- log.retention.ms

> 设置topic多少秒删除一次

- retention.ms

如何保证消息的顺序

kafka stream

kafka集群的重要配置

broker端参数（静态参数，需要重启kafka才能生效） 存储信息配置 log.dirs : kafka数据存储路径 log.dir

zookeeper配置 zookeeper.connect： zk连接信息

Broker 连接