一、Kafka的Java客户端-生产者

1.引入依赖

```
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.4.1</version>
    </dependency>
```

2.生产者发送消息的基本实现

```
//消息的发送方
public class MyProducer {

  private final static String TOPIC_NAME = "my-replicated-topic";

  public static void main(String[] args) throws ExecutionException,
InterruptedException {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"10.31.167.10:9092,10.31.167.10:9093,10.31.167.10:9094");

    //把发送的key从字符串序列化为字节数组
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
    //把发送消息value从字符串序列化为字节数组
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

    Producer<String, String> producer = new KafkaProducer<String,
String>(props);

    Order order = new Order((long) i, i);

    ProducerRecord<String, String> producerRecord = new
ProducerRecord<String, String>(TOPIC_NAME
        , order.getOrderId().toString(), JSON.toJSONString(order));
```

```
    //等待消息发送成功的同步阻塞方法
    RecordMetadata metadata = producer.send(producerRecord).get();
    //=====阻塞=======
    System.out.println("同步方式发送消息结果：" + "topic-" +
metadata.topic() + "|partition-"
                    + metadata.partition() + "|offset-" +
metadata.offset());

  }
}
```

3.发送消息到指定分区上

```
 ProducerRecord<String, String> producerRecord = new
ProducerRecord<String, String>(TOPIC_NAME
   , 0, order.getOrderId().toString(), JSON.toJSONString(order));
```

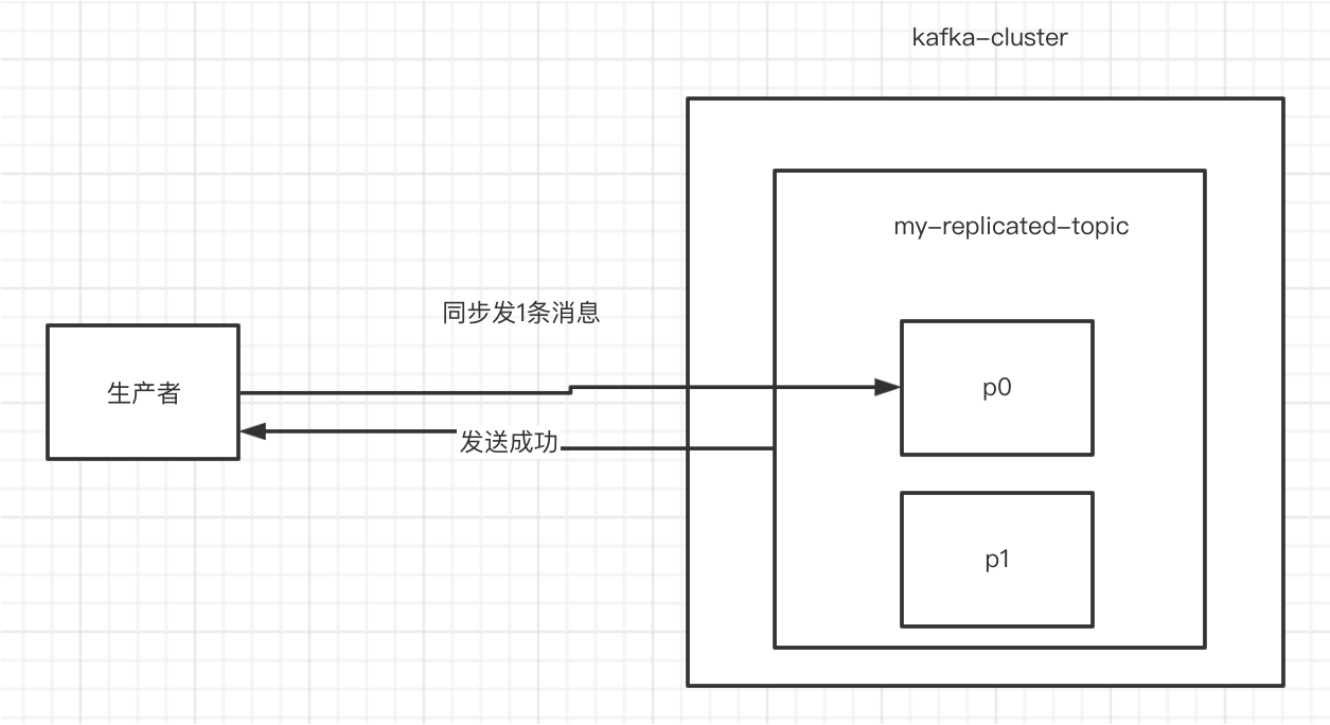
4.未指定分区，则会通过业务key的hash运算，算出消息往哪个分区上发

```
//未指定发送分区，具体发送的分区计算公式：hash(key)%partitionNum
ProducerRecord<String, String> producerRecord = new
ProducerRecord<String, String>(TOPIC_NAME
  , order.getOrderId().toString(), JSON.toJSONString(order));
```

5.同步发送

生产者同步发消息，在收到kafka的ack告知发送成功之前一直处于阻塞状态

```
//等待消息发送成功的同步阻塞方法
 RecordMetadata metadata = producer.send(producerRecord).get();
            System.out.println("同步方式发送消息结果：" + "topic-" +
metadata.topic() + "|partition-"
                    + metadata.partition() + "|offset-" +
metadata.offset());
```



6.异步发消息

生产者发消息，发送完后不用等待broker给回复，直接执行下面的业务逻辑。可以提供callback，让broker异步的调用callback，告知生产者，消息发送的结果

```
//要发送5条消息
      Order order = new Order((long) i, i);
      //指定发送分区
            ProducerRecord<String, String> producerRecord = new
ProducerRecord<String, String>(TOPIC_NAME
                  , 0, order.getOrderId().toString(),
JSON.toJSONString(order));
       //异步回调方式发送消息
      producer.send(producerRecord, new Callback() {
        public void onCompletion(RecordMetadata metadata, Exception
exception) {
          if (exception != null) {
            System.err.println("发送消息失败: " +
exception.getStackTrace());

          }
          if (metadata != null) {
            System.out.println("异步方式发送消息结果: " + "topic-" +
metadata.topic() + "|partition-"
```

```
                + metadata.partition() + "|offset-" + metadata.offset());
            }
        }
    });

```

7.关于生产者的ack参数配置

在同步发消息的场景下：生产者发动broker上后，ack会有3种不同的选择：

```
（1）acks=0： 表示producer不需要等待任何broker确认收到消息的回复，就可以继续发送下一条消息。性能最高，但是最容易丢消息。
（2）acks=1： 至少要等待leader已经成功将数据写入本地log，但是不需要等待所有follower是否成功写入。就可以继续发送下一
              条消息。这种情况下，如果follower没有成功备份数据，而此时leader又挂掉，则消息会丢失。
（3）acks=-1或all： 需要等待 min.insync.replicas(默认为1，推荐配置大于等于2) 这个参数配置的副本个数都成功写入日志，这种策略会保证只要有一个备份存活就不会丢失数据。这是最强的数据保证。一般除非是金融级别，或跟钱打交道的场景才会使用这种配置。
```

code:

```
props.put(ProducerConfig.ACKS_CONFIG, "1");
```

8.其他一些细节

- 发送会默认会重试3次，每次间隔100ms
- 发送的消息会先进入到本地缓冲区（32mb），kakfa会跑一个线程，该线程去缓冲区中取16k的数据，发送到kafka，如果到10毫秒数据没取满16k，也会发送一次。

二、消费者

1.消费者消费消息的基本实现

```
public class MyConsumer {

  private final static String TOPIC_NAME = "my-replicated-topic";
  private final static String CONSUMER_GROUP_NAME = "testGroup";

  public static void main(String[] args) {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"10.31.167.10:9092,10.31.167.10:9093,10.31.167.10:9094");
    // 消费分组名
    props.put(ConsumerConfig.GROUP_ID_CONFIG, CONSUMER_GROUP_NAME);
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    //创建一个消费者的客户端
    KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(props);
    // 消费者订阅主题列表
    consumer.subscribe(Arrays.asList(TOPIC_NAME));

    while (true) {
      /*
       * poll() API 是拉取消息的长轮询
       */
      ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));
      for (ConsumerRecord<String, String> record : records) {
        System.out.printf("收到消息：partition = %d,offset = %d, key =
%s, value = %s%n", record.partition(),
          record.offset(), record.key(), record.value());
      }
    }
  }
}

```

2.自动提交offset

设置自动提交参数 - 默认

```
// 是否自动提交offset，默认就是true
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
// 自动提交offset的间隔时间
props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
```

消费者poll到消息后默认情况下，会自动向broker的_consumer_offsets主题提交当前主题-分区消费的偏移量。

自动提交会丢消息：因为如果消费者还没消费完poll下来的消息就自动提交了偏移量，那么此时消费者挂了，于是下一个消费者会从已提交的offset的下一个位置开始消费消息。之前未被消费的消息就丢失掉了。

3.手动提交offset

- 设置手动提交参数

```
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
```

- 在消费完消息后进行手动提交
  - 手动同步提交

```
if (records.count() > 0) {
    // 手动同步提交offset，当前线程会阻塞直到offset提交成功
    // 一般使用同步提交，因为提交之后一般也没有什么逻辑代码了
    consumer.commitSync();
}
```

  - 手动异步提交

```
if (records.count() > 0) {
   // 手动异步提交offset，当前线程提交offset不会阻塞，可以继续处理后面的程序逻辑
   consumer.commitAsync(new OffsetCommitCallback() {
      @Override
      public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {
            if (exception != null) {
                System.err.println("Commit failed for " + offsets);
                System.err.println("Commit failed exception: " + exception.getStackTrace());
            }
      }
   });
}
```

4.消费者poll消息的过程

- 消费者建立了与broker之间的长连接，开始poll消息。
- 默认一次poll500条消息

```
props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
```

可以根据消费速度的快慢来设置，因为如果两次poll的时间如果超出了30s的时间间隔，kafka会认为其消费能力过弱，将其踢出消费组。将分区分配给其他消费者。

可以通过这个值进行设置：

```
props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 30 * 1000);
```

- 如果每隔1s内没有poll到任何消息，则继续去poll消息，循环往复，直到poll到消息。如果超出了1s，则此次长轮询结束。

```
ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));
```

- 消费者发送心跳的时间间隔

```
props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 1000);
```

- kafka如果超过10秒没有收到消费者的心跳，则会把消费者踢出消费组，进行rebalance，把分区分配给其他消费者。

```
props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 10 * 1000);
```

5.指定分区消费

```
consumer.assign(Arrays.asList(new TopicPartition(TOPIC_NAME, 0)));
```

6.消息回溯消费

```
consumer.assign(Arrays.asList(new TopicPartition(TOPIC_NAME, 0)));
consumer.seekToBeginning(Arrays.asList(new TopicPartition(TOPIC_NAME, 0)));
```

7.指定offset消费

```
consumer.assign(Arrays.asList(new TopicPartition(TOPIC_NAME, 0)));
consumer.seek(new TopicPartition(TOPIC_NAME, 0), 10);
```

8.从指定时间点消费

```
        List<PartitionInfo> topicPartitions = consumer.partitionsFor(TOPIC_NAME);
        //从1小时前开始消费
        long fetchDataTime = new Date().getTime() - 1000 * 60 * 60;
        Map<TopicPartition, Long> map = new HashMap<>();
        for (PartitionInfo par : topicPartitions) {
            map.put(new TopicPartition(TOPIC_NAME, par.partition()), fetchDataTime);
        }
```

```
        Map<TopicPartition, OffsetAndTimestamp> parMap =
consumer.offsetsForTimes(map);
        for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry :
parMap.entrySet()) {
            TopicPartition key = entry.getKey();
            OffsetAndTimestamp value = entry.getValue();
            if (key == null || value == null) continue;
            Long offset = value.offset();
            System.out.println("partition-" + key.partition() +
"|offset-" + offset);
            System.out.println();
            //根据消费里的timestamp确定offset
            if (value != null) {
                consumer.assign(Arrays.asList(key));
                consumer.seek(key, offset);
            }
        }
```

9.新消费组的消费偏移量

当消费主题的是一个新的消费组，或者指定offset的消费方式，offset不存在，那么应该如何消费?

- latest(默认)：只消费自己启动之后发送到主题的消息
- earliest：第一次从头开始消费，以后按照消费offset记录继续消费，这个需要区别于consumer.seekToBeginning(每次都从头开始消费)

```
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

三、Springboot中使用Kafka

1.引入依赖

```xml
    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka</artifactId>
    </dependency>
```

2.配置文件

```yaml
server:
  port: 8080

spring:
  kafka:
    bootstrap-servers: 172.16.253.21:9093
    producer: # 生产者
      retries: 3 # 设置大于0的值，则客户端会将发送失败的记录重新发送
      batch-size: 16384
      buffer-memory: 33554432
      acks: 1
      # 指定消息key和消息体的编解码方式
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.apache.kafka.common.serialization.StringSerializer
    consumer:
      group-id: default-group
      enable-auto-commit: false
      auto-offset-reset: earliest
      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      max-poll-records: 500
    listener:
      # 当每一条记录被消费者监听器（ListenerConsumer）处理之后提交
      # RECORD
      # 当每一批poll()的数据被消费者监听器（ListenerConsumer）处理之后提交
      # BATCH
      # 当每一批poll()的数据被消费者监听器（ListenerConsumer）处理之后，距离上
次提交时间大于TIME时提交
      # TIME
```

```
      # 当每一批poll()的数据被消费者监听器（ListenerConsumer）处理之后，被处理record数量大于等于COUNT时提交
      # COUNT
      # TIME | COUNT 有一个条件满足时提交
      # COUNT_TIME
      # 当每一批poll()的数据被消费者监听器（ListenerConsumer）处理之后，手动调用Acknowledgment.acknowledge()后提交
      # MANUAL
      # 手动调用Acknowledgment.acknowledge()后立即提交，一般使用这种
      # MANUAL_IMMEDIATE
      ack-mode: MANUAL_IMMEDIATE
  redis:
    host: 172.16.253.21
```

3.消息生产者

发送消息到指定topic

```
@RestController
public class KafkaController {
  private final static String TOPIC_NAME = "my-replicated-topic";

  @Autowired
  private KafkaTemplate<String, String> kafkaTemplate;

  @RequestMapping("/send")
  public void send() {
    kafkaTemplate.send(TOPIC_NAME, 0, "key", "this is a msg");
  }
}
```

4.消息消费者

- 设置消费组，消费指定topic

```
@KafkaListener(topics = "my-replicated-topic",groupId = "MyGroup1")
  public void listenGroup(ConsumerRecord<String, String> record,
Acknowledgment ack) {
    String value = record.value();
    System.out.println(value);
    System.out.println(record);
    //手动提交offset
    ack.acknowledge();
  }
```

- 设置消费组、多topic、指定分区、指定偏移量消费及设置消费者个数。

```
  @KafkaListener(groupId = "testGroup", topicPartitions = {
      @TopicPartition(topic = "topic1", partitions = {"0", "1"}),
      @TopicPartition(topic = "topic2", partitions = "0",
       partitionOffsets = @PartitionOffset(partition = "1",
initialOffset = "100"))
    },concurrency = "3")//concurrency就是同组下的消费者个数，就是并发消费数，
建议小于等于分区总数
  public void listenGroup(ConsumerRecord<String, String> record,
Acknowledgment ack) {
    String value = record.value();
    System.out.println(value);
    System.out.println(record);
    //手动提交offset
    ack.acknowledge();
  }
```

四、Kafka集群Controller、Rebalance和HW

1.Controller

Kafka集群中的broker在zk中创建临时序号节点，序号最小的节点（最先创建的节点）将作为集群的controller，负责管理整个集群中的所有分区和副本的状态：

- 当某个分区的leader副本出现故障时，由控制器负责为该分区选举新的leader副本。
- 当检测到某个分区的ISR集合发生变化时，由控制器负责通知所有broker更新其元数据信息。

- 当使用kafka-topics.sh脚本为某个topic增加分区数量时，同样还是由控制器负责让新分区被其他节点感知到。

2.Rebalance机制

前提是：消费者没有指明分区消费。当消费组里消费者和分区的关系发生变化，那么就会触发rebalance机制。

这个机制会重新调整消费者消费哪个分区。

在触发rebalance机制之前，消费者消费哪个分区有三种策略：

- range：通过公示来计算某个消费者消费哪个分区
- 轮询：大家轮着消费
- sticky：在触发了rebalance后，在消费者消费的原分区不变的基础上进行调整。

3.HW和LEO

HW俗称高水位，HighWatermark的缩写，取一个partition对应的ISR中最小的LEO(log-end-offset)作为HW，consumer最多只能消费到HW所在的位置。另外每个replica都有HW,leader和follower各自负责更新自己的HW的状态。对于leader新写入的消息，consumer不能立刻消费，leader会等待该消息被所有ISR中的replicas同步后更新HW，此时消息才能被consumer消费。这样就保证了如果leader所在的broker失效，该消息仍然可以从新选举的leader中获取。

五、Kafka线上问题优化

1.如何防止消息丢失

- 发送方： ack是1 或者-1/all 可以防止消息丢失，如果要做到99.9999%，ack设成all，把min.insync.replicas配置成分区备份数
- 消费方：把自动提交改为手动提交。

2.如何防止消息的重复消费

一条消息被消费者消费多次。如果为了消息的不重复消费，而把生产端的重试机制关闭、消费端的手动提交改成自动提交，这样反而会出现消息丢失，那么可以直接在防治消息丢失的手段上再加上消费消息时的幂等性保证，就能解决消息的重复消费问题。

幂等性如何保证：

- mysql 插入业务id作为主键，主键是唯一的，所以一次只能插入一条
- 使用redis或zk的分布式锁（主流的方案）

3.如何做到顺序消费RocketMQ

- 发送方：在发送时将ack不能设置0，关闭重试，使用同步发送，等到发送成功再发送下一条。确保消息是顺序发送的。
- 接收方：消息是发送到一个分区中，只能有一个消费组的消费者来接收消息。

  因此，kafka的顺序消费会牺牲掉性能。

4.解决消息积压问题

消息积压会导致很多问题，比如磁盘被打满、生产端发消息导致kafka性能过慢，就容易出现服务雪崩，就需要有相应的手段：

- 方案一：在一个消费者中启动多个线程，让多个线程同时消费。——提升一个消费者的消费能力。
- 方案二：如果方案一还不够的话，这个时候可以启动多个消费者，多个消费者部署在不同的服务器上。其实多个消费者部署在同一服务器上也可以提高消费能力——充分利用服务器的cpu资源。
- 方案三：让一个消费者去把收到的消息往另外一个topic上发，另一个topic设置多个分区和多个消费者 ，进行具体的业务消费。

5.延迟队列

延迟队列的应用场景：在订单创建成功后如果超过30分钟没有付款，则需要取消订单，此时可用延时队列来实现

- 创建多个topic，每个topic表示延时的间隔

    - topic_5s: 延时5s执行的队列
    - topic_1m: 延时1分钟执行的队列
    - topic_30m: 延时30分钟执行的队列
- 消息发送者发送消息到相应的topic，并带上消息的发送时间
- 消费者订阅相应的topic，消费时轮询消费整个topic中的消息
    - 如果消息的发送时间，和消费的当前时间超过预设的值，比如30分钟
    - 如果消息的发送时间，和消费的当前时间没有超过预设的值，则不消费当前的offset及之后的offset的所有消息都消费
    - 下次继续消费该offset处的消息，判断时间是否已满足预设值

六、Kafka-eagle监控平台

安装Kafka-eagle

- 官网下载压缩包

http://www.kafka-eagle.org/

- 安装jdk
- 解压缩后修改配置文件 system-config.properties

```
# 配置zk
cluster1.zk.list=172.16.253.35:2181
# 配置mysql
kafka.eagle.driver=com.mysql.cj.jdbc.Driver
kafka.eagle.url=jdbc:mysql://172.16.253.22:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
kafka.eagle.username=root
kafka.eagle.password=123456
```

- 进入到bin目录，为ke.sh增加可执行的权限

```
chmod +x ke.sh
```

- 启动kafka-eagle

```
./ke.sh start
```

千锋教育Java教研院 关注公众号：Java架构栈 获取更多资料