

运行启动Kafka

java8+

kafka可以使用zookeeper或kraft启动，但只能使用其中一种方式，不能同时使用；

kraft是kafka的内置共识机制，用于取代zookeeper

下载kafka完成后：

- 使用zookeeper启动kafka：

进入kafka的bin目录中，

```
zookeeper-server-start.sh ../config/zookeeper.properties &: 启动zookeeper (因为kafka  
安装包自带zookeeper的jar包)，可以在不安装zookeeper的情况下运行zookeeper。&表示后台运  
行。
```

```
kafka-server-start.sh ../config/server.properties &: 启动kafka
```

```
kafka-server-stop.sh ../config/server.properties: 关闭kafka
```

```
zookeeper-server-start.sh ../config/zookeeper.properties: 关闭zookeeper
```

- 使用kraft启动kafka

进入kafka的bin目录中，

1. 生成 Cluster UUID (集群UUID) : kafka-storage.sh random-uuid
2. kafka-storage.sh format -t 生成的uuid -c ../config/kraft/server.properties
3. 启动kafka: kafka-server.start.sh ../config/kraft/server.properties &
4. 关闭kafka: kafka-server.stop.sh ../config/kraft/server.properties

- docker:

1. 拉取kafka的镜像: docker pull apache/kafka:3.7.0
2. 启动kafka的docker容器: docker run -p 9092:9092 apache/kafka:3.7.0

以下均基于kraft启动

外部连接kafka

在idea中下载插件kafka，启动kafka后测试连接发现连接不上。这是由于配置的问题。

Docker容器的kafka有三种配置启动方式：

- 默认配置：使用kafka容器的默认配置，外部是连接不上的
- 文件输入：提供一个本地kafka属性配置文件，覆盖docker容器中的默认配置文件

- 环境变量：通过env变量定义kafka属性，覆盖默认配置中对应该属性的值

文件覆盖：

进入kafka：

```
docker exec -it 容器id /bin/bash
```

配置文件 `server.properties` 就在 `/etc/kafka/docker` 目录下

将容器中的配置文件复制一份到宿主机上：

```
docker cp 容器id:/etc/kafka/docker/server.properties 目的地目录
```

将复制的配置文件进行修改：

```
listeners=PLAINTEXT://0.0.0.0:9092
```

```
advertised.listeners=PLAINTEXT://<宿主机IP>:9092
```

(其实这两者如果采用本笔记启动-(1)方法启动，自动就会设置这两个配置)

配置文件映射：

某些 Kafka 镜像（例如官方镜像或其他流行的社区镜像）会在启动脚本中检查特定路径（如 `/mnt/shared/config`）是否存在配置文件。

```
docker run -v /home/mutesniper/code/docker:/mnt/shared/config -p 9092:9092  
apache/kafka:3.7.0
```

再进行远程连接就成功了。

Springboot集成kafka

在springboot项目创建好后，需要进行相关配置。

- 服务器连接: `spring.kafka.bootstrap-servers:宿主机IP: 端口`
- 生产者（配置项参考`KafkaProperties.java`）:后面根据需要进行配置
- 消费者（配置项参考`KafkaProperties.java`）:后面根据需要进行配置

生产者

*Kaka几个概念

主题Topic

- **定义**：主题是 Kafka 中消息的分类或名称。生产者将消息发布到特定的主题，而消费者订阅这些主题以接收消息。
- 特点
 - 每个消息都属于一个特定的主题。

- 主题可以有多个分区 (Partition)，这有助于并行处理和扩展性。

分区Partition

- **定义：**每个主题可以分为多个分区，这是 Kafka 实现高吞吐量和水平扩展的关键机制。
- **特点**
 - 分区内的消息是有序的，但跨分区的消息顺序不保证。
 - 每个分区可以被复制 (Replication) 以提高容错能力。
 - 消费者组中的消费者实例会分配到不同的分区上进行消费，实现负载均衡。

偏移量Offset

- **定义：**偏移量是每个分区中每条消息的唯一标识符，表示消息在分区中的位置。
- **用途**
 - 消费者通过记录偏移量来追踪已经消费的消息，以便于从上次停止的地方继续消费。
 - 偏移量由消费者提交给 Kafka 或外部存储系统（如 Zookeeper）。

生产者Producer

- **定义：**生产者是向 Kafka 主题发送消息的应用程序。
- **功能**
 - 生产者可以选择将消息发送到特定的分区，也可以让 Kafka 根据某种策略（如哈希算法）自动选择分区。
 - 生产者还可以配置消息的确认机制（如是否等待副本同步完成）。

消费者Consumer

- **定义：**消费者是从 Kafka 主题读取消息的应用程序。
- **功能**
 - 消费者通常以消费者组的形式工作，同一组内的消费者不会重复消费相同的消息。
 - 消费者需要管理自己的偏移量，并决定何时提交偏移量。

消费者组Consumer Group

- **定义：**消费者组是一组共同消费某个或某些主题的消费者的集合。
- **特点**
 - 同一消费者组内的消费者之间实现负载均衡，即每个分区只能被同一个消费者组内的一个消费者消费。
 - 不同消费者组之间的消费者可以独立消费相同的消息。

Broker

- **定义：**Broker 是 Kafka 集群中的一个节点，负责存储数据、处理客户端请求（生产和消费）以及集群间的数据复制。
- **特点**
 - Kafka 集群由多个 Broker 组成，提供高可用性和扩展性。
 - 每个 Broker 可以持有多个主题的不同分区。

日志Log

- **定义：**在 Kafka 中，主题的每个分区实际上是一个分布式的提交日志。
- **特性**
 - 日志是一种持久化、有序且不可变的序列，新的消息不断追加到日志末尾。
 - 日志支持长时间保存，具体时长取决于配置的保留策略。

Zookeeper

- **定义：**尽管较新版本的 Kafka 已经开始减少对 Zookeeper 的依赖，传统上，Kafka 使用 Zookeeper 来管理和协调 Kafka 集群的状态。
- **功能**
 - 维护 Kafka Broker 的元数据信息（如主题、分区等）。
 - 协助选举领导者分区（Leader Partition）。
 - 管理消费者组的偏移量。

Event生产

```
@Component
public class EventProducer {
    //加入spring-kafka依赖，并且我们配置好了yml,
    // springboot自动配置好了kafka，自动装配好了kafkaTemplate这个Bean
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendEvent() {
        kafkaTemplate.send("hello-topic", "hello kafka!");
    }
}
```

测试：

```
@Autowired
private EventProducer eventProducer;

@Test
void test01() {
    eventProducer.sendEvent();
}
```

发送成功

Event消费

```
@Component
public class EventConsumer {
    //采用监听的方式接收事件（消息）
    //指定订阅的topic和消费者所在消费者组
    @KafkaListener(topics="hello-topic",groupId = "hello-group")
    public void onEvent(String message) {
        System.out.println("读取到的消息："+message);
    }
}
```

通过这种方式，在springboot项目启动后，此consumer就能自动监听hello-topic主题发送的消息。

但是这种方式仅仅只能监听项目启动后发送的消息，那么要怎么获取之前已经发送的消息呢。

也就是：

默认情况下，当启动一个新的消费者组时，它会从每个分区的最新偏移量（即该分区中最后一条消息的下一个位置）开始消费。如果希望从第一条消息开始消费，需要将消费者的 `auto.offset.reset` 设置为 `earliest`

```
spring:  
  kafka:  
    consumer:  
      auto-offset-reset: earliest
```

但是，我们在启动主程序后发现并没有读到，这是因为

如果之前已经用相同的消费者组id消费过该Topic，并且kafka已经保存了该消费者组的偏移量，即使设置了`auto..`配置，该设置也不会产生实际效果，因为kafka只会在找不到偏移量时使用这个配置。在这种情况下，需要手动重置偏移量或使用一个新的消费者组id。

先使用第二种方法：

```
groupid="hello-group-02"，发现可以从头读取
```

观察一下kafka插件显示的信息：

hello-topic:

分区 ID	消息计数	起始偏移量	结束偏移量	领导者	副本
0	3	0	3	1	1

主题	分区	滞后	偏移量
hello-topic	0	1	2

主题	分区	滞后	偏移量
hello-topic	0	0	3

再使用第一种方法

- 防火墙开启9092端口
- 进入docker容器
- 使用命令 `/opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group 组名` 查看消费者组情况。
- `/opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \ --group 组名 \ --reset-offsets \`

```
--to-earliest \
```

```
--execute \
```

```
--topic topic名
```

可以将偏移量重置到开头。 (还可以重置到最新、按偏移量重置、按时间重置)

消费时偏移量策略的配置

```
spring:  
  kafka:  
    consumer:  
      auto-offset-reset:
```

取值：

- earliest：当各分区没有初始偏移量或当前偏移量无效时，自动将偏移量重置为最早的偏移量
- latest：如果没有找到初始偏移量或当前偏移量无效，自动将偏移量重置为最新的偏移量
- none：如果没有找到初始偏移量或当前偏移量无效，则抛出异常。

发送Message对象消息

```
public void sendEvent2() {  
    //通过构建器模式创建Message对象(springframework包下的)  
    Message message = MessageBuilder.withPayload("hello kafka!")  
        //在header中放置topic的名字  
        .setHeader(KafkaHeaders.TOPIC, "hello-topic-02")  
        .build();  
    kafkaTemplate.send(message);  
}
```

```
@Test  
void test02() {  
    eventProducer.sendEvent2();  
}
```

发送ProducerRecord对象消息

```
public void sendEvent3() {  
    //消息头，允许你附加一些元数据信息到消息上(key-value)  
    //消费者接收到该消息后，可以拿到消息头信息  
    Headers headers=new RecordHeaders();  
    headers.add("phone", "11111111".getBytes(StandardCharsets.UTF_8));  
    //第一个泛型是key，第二个泛型是value  
    //这里采用最全的构造函数对参数进行示范  
    //还有其他更简单的构造函数  
    ProducerRecord<String, String> record = new ProducerRecord<>(  
        "hello-topic-03", //topic
```

```
    0, //partition
    System.currentTimeMillis(), //时间戳
    "key1", //key
    "hello kafka!", //value
    headers); //消息头
    kafkaTemplate.send(record);
}
```

发送指定分区的消息

```
public void sendEvent4() {
    kafkaTemplate.send("hello-topic-02",
0, System.currentTimeMillis(), "key2", "hello kafka!");
}
```

send方法有6个重载的方法，上面仅讲了一部分

- (m) `send(String, K, V?)`
- (m) `send(String, Integer, K, V?)`
- (m) `send(Message<?>)`
- (m) `send(String, Integer, Long, K, V?)`
- (m) `send(ProducerRecord<K, V>)`
- (m) `send(String, V?)`

发送默认topic消息

```
public void sendEvent5() {
    kafkaTemplate.sendDefault(0, System.currentTimeMillis(), "key3", "hello
kafka!");
}
```

这是参数最全的方法，但是sendDefault方法的所有重载方法都没有指定topic，所以要在配置文件中进行配置

```
spring:
  kafka:
    template:
      default-topic: default-topic
```

那么就会发送到default-topic

```
(m) sendDefault(Integer, Long, K, V?)  
(m) sendDefault(K, V?)  
(m) sendDefault(V?)  
(m) sendDefault(Integer, K, V?)
```

send和sendDefault方法的区别

- 主要区别是发送消息到Kafka时是否每次都需要指定主题topic;
 - 1、`kafkaTemplate.send(...)` 该方法需要明确地指定要发送消息的目标主题topic;
 - 2、`kafkaTemplate.sendDefault()` 该方法不需要指定要发送消息的目标主题topic,
-
- `kafkaTemplate.send(...)` 方法适用于需要根据业务逻辑或外部输入动态确定消息目标topic的场景;
 - `kafkaTemplate.sendDefault()` 方法适用于总是需要将消息发送到特定默认topic的场景;
 - `kafkaTemplate.sendDefault()` 是一个便捷方法，它使用配置中指定的默认topic来发送消息，如果应用中所有消息都发送到同一个主题时采用该方法非常方便，可以减少代码的重复或满足特定的业务需求；

获取生产者消息发送结果

- `.send()`方法和`.sendDefault()`方法都返回`CompletableFuture<SendResult<K, V>>`;
- `CompletableFuture` 是Java 8中引入的一个类，用于异步编程，它表示一个异步计算的结果，这个特性使得调用者不必等待操作完成就能继续执行其他任务，从而提高了应用程序的响应速度和吞吐量；

为什么使用异步编程：

- 因为调用 `kafkaTemplate.send()` 方法发送消息时，Kafka可能需要一些时间来处理该消息（例如：网络延迟、消息序列化、Kafka集群的负载等），如果 `send()` 方法是同步的，那么发送消息可能会阻塞调用线程，直到消息发送成功或发生错误，这会导致应用程序的性能下降，尤其是在高并发场景下；
- 使用 `CompletableFuture`, `.send()` 方法可以立即返回一个表示异步操作结果的未来对象，而不是等待操作完成，这样，调用线程可以继续执行其他任务，而不必等待消息发送完成。当消息发送完成时（无论是成功还是失败），`CompletableFuture`会相应地更新其状态，并允许我们通过回调、阻塞等方式来获取操作结果；

阻塞式

使用`CompletableFuture`的`get()`方法，同步阻塞等待发送结果

```

public void sendEvent6() {
    CompletableFuture<SendResult<String, String>> completableFuture
        = kafkaTemplate.sendDefault(0, System.currentTimeMillis(), "key3",
"hello kafka!");

    //通过CompletableFuture这个类拿结果，这个类里面有很多方法
    //1.阻塞等待的方法拿结果
    try {
        SendResult<String, String> sendResult = completableFuture.get();
        if(sendResult.getRecordMetadata()!=null) {
            //如果以上判断成立（元数据不为空）Kafka服务器接收到了消息
            //以拿topic信息为例
            System.out.println("消息发送成功: "+sendResult.getRecordMetadata().topic());
        }
        System.out.println("producerRecord: "+sendResult.getProducerRecord());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

非阻塞式

使用thenAccept(), thenApply(), thenRun()等方法来注册回调函数，回调函数将在CompletableFuture完成时被执行

```

public void sendEvent7() {
    CompletableFuture<SendResult<String, String>> completableFuture
        = kafkaTemplate.sendDefault(0, System.currentTimeMillis(), "key3",
"hello kafka!");

    //通过CompletableFuture这个类拿结果，这个类里面有很多方法
    //2.非阻塞的方法拿结果
    try {

        completableFuture.thenAccept(sendResult->{
            if(sendResult.getRecordMetadata()!=null) {
                //如果以上判断成立（元数据不为空）Kafka服务器接收到了消息
                //以拿topic信息为例
                System.out.println("消息发送成功: "+sendResult.getRecordMetadata().topic());
            }
        }

        System.out.println("producerRecord: "+sendResult.getProducerRecord());
    })
        .exceptionally(t->{
            t.printStackTrace();
            //做消息发送失败处理
            return null;
        });
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

发送对象消息(序列化器)

在前面的测试中，发的消息都是字符串，下面演示发送对象的操作方法

```
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@Data  
public class User {  
    private int id;  
    private String phone;  
    private Date birthday;  
}
```

```
public void sendEvent8() {  
    User user= User.builder().id(1208).phone("111111111").birthday(new  
Date()).build();  
    //分区为null，让kafka自己决定把消息发到哪个分区  
    kafkaTemplate2.sendDefault(null, System.currentTimeMillis(), "key4", user);  
}
```

但是发现报序列化异常错误，所以加上配置：

```
spring:  
kafka:  
producer:  
#默认是StringSerializer.class序列化  
value-serializer:  
org.springframework.kafka.support.serializer.JsonSerializer
```

*Replica副本

在 Apache Kafka 中，“replica”（副本）是指分区（Partition）的一个精确复制。Kafka 使用副本机制来提高消息的可靠性和系统的容错能力。每个主题的分区可以有多个副本，其中一个作为领导者（Leader），其余的作为追随者（Follower）。以下是关于 Kafka 副本的一些关键点：

副本的作用

1. **数据冗余**：通过在多个 broker 上存储相同的数据副本，防止数据丢失。
2. **高可用性**：如果某个 broker 宕机，其上的分区副本可以在其他 broker 上找到，并且其中的一个副本会被选举为新的领导者，从而保证服务不中断。
3. **负载均衡**：读请求可以被分发到不同的副本上，减轻领导者的负担。

相关概念

- **Leader 副本**：每个分区都有一个领导者副本，它负责处理所有的读写请求。
- **Follower 副本**：除了 Leader 之外的所有副本都是 Follower。它们从 Leader 同步数据，并准备在需要时成为新的 Leader。
- **ISR (In-Sync Replicas)**：表示与 Leader 保持同步的副本集合。只有 ISR 集合中的副本才有资格被选举为新的 Leader。

注意事项

- 副本的数量不应超过 broker 的数量：因为副本必须分布在不同的 broker 上，所以 replication.factor 不能大于集群中 broker 的总数。
- ISR 与数据一致性：为了确保数据的一致性，生产者通常会等待 ISR 中的所有副本都确认收到了消息。这可以通过调整生产者的 acks 配置来实现。

总结

副本是 Kafka 实现高可用性和数据持久性的核心机制之一。通过合理配置副本因子和了解 ISR 的工作机制，可以有效地提高 Kafka 集群的可靠性和稳定性。对于大多数生产环境来说，推荐至少使用 replication.factor=3 来保证足够的容错能力。

创建topic并指定分区和副本

命令行

```
kafka-topics.sh --create --topic my-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 3
```

replica不能超过broker数量，也不能为0（最少为1--自己）

springboot

在执行代码时指定分区和副本

直接使用send方法发送消息时，如果topic不存在，kafka会帮我们自动完成topic的创建工作，但这种情况下创建的topic默认只有一个分区，分区有一个副本。

我们可以在项目中新建一个配置类专门用来初始化topic

```
@Configuration
public class KafkaConfig {

    @Bean
    public NewTopic newTopic() {
        //名字、partition、副本
        return new NewTopic("He-topic", 5, (short) 1);
    }

}
```

如果topic存在(topic名相同)，不会创建（所以重启主程序，不会丢失"He-topic"中的信息）

但是在修改时，虽然也不会弄丢数据，但是分区只能增多不能减少。

生产者发送消息的分区策略

在发送时，如果不指定放到哪个分区，那么消息放置在哪个分区采用什么策略呢？

据源码分析可知，未指定分区的消息是根据其key使用murmur2哈希算法计算出的分区进行发送的。没有key则采用另外的方法。

RoundRobinPartitioner

这种分区策略虽然名为轮询，但是并不是严格遵照轮询策略把消息进行分区的。

下面用代码实现使用这种分区策略

在配置类中写入：

```
@Value("${spring.kafka.bootstrap-servers}")
private String bootstrapServers;
@Value("${spring.kafka.producer.value-serializer}")
private String valueSerializer;
@Value("${spring.kafka.producer.key-serializer}")
private String keySerializer;

//生产者相关配置信息
public Map<String, Object> producerConfigs(){
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, keySerializer);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, valueSerializer);
    props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
    RoundRobinPartitioner.class);
    return props;
}

//生产者工厂
public ProducerFactory<String, ?> producerFactory() {
    //这里的泛型也是消息的键和值
    return new DefaultKafkaProducerFactory<String, Object>(producerConfigs());
}

//kafkaTemplate 覆盖默认配置类中的kafkaTemplate
@Bean
public KafkaTemplate<String, ?> kafkaTemplate() {
    return new KafkaTemplate<>(producerFactory());
}
```

用我们自定义的kafkaTemplate覆盖了容器中原有的kafkaTemplate，让它使用RoundRobinPartitioner。

(问题待解决)自定义分配策略

自定义XxxPartitioner类继承Partitioner

```
public class CustomPartitioner implements Partitioner {

    private AtomicInteger nextPartition=new AtomicInteger(0);
    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object
    value, byte[] valueBytes, Cluster cluster) {
        //通过kafka集群信息获取此topic的partitions
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
```

```
if(key==null) {
    //使用轮询方法选择分区
    int next = nextPartition.getAndIncrement();
    if (next >= numPartitions) {
        nextPartition.compareAndSet(next, 0);
    }
    System.out.println("分区值: " + next);
    return next;
} else {
    //如果key不为null, 则使用默认的分区策略
    return Utils.toPositive(Utils.murmur2(keyBytes))%numPartitions;
}
}

@Override
public void close() {

}

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

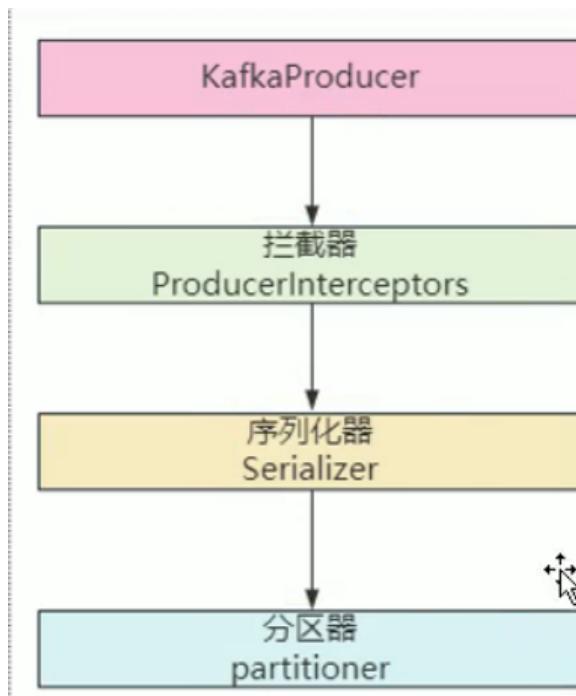
}
}
```

然后再将上一章采用RoundRobinPartitioner的分区策略改成运用我们自定义的分区策略:

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, CustomPartitioner.class);
```

但是partition方法（计算出要放入的分区的方法）会调用两次，可能达不到我们想要的效果。（待解决）

生产者发送消息的流程



其中分区器的方法会执行两次，也就是上一章未解决的问题

简略源码阅读：https://www.bilibili.com/video/BV14J4m187jz?p=86&vd_source=c054be8430afebb3d00e5f2d0b77f9fc

*自定义消息发送的拦截器

```

public class CustomProducerInterceptor implements
ProducerInterceptor<String, Object> {
    //发送消息时会先调用该方法，对消息进行拦截
    //可以在拦截中对消息进行一些处理，比如记录日志等操作
    @Override
    public ProducerRecord<String, Object> onSend(ProducerRecord producerRecord) {
        System.out.println("拦截消息：" + producerRecord.toString());
        return producerRecord;
    }

    //服务器收到消息后的确认
    @Override
    public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {
        if(recordMetadata != null){
            System.out.println("服务器收到该消息：" + recordMetadata.offset());
        } else {
            System.out.println("消息发送失败：" + e.getMessage());
        }
    }

    @Override
    public void close() {

    }

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

    }
}

```

给自定义的kafkaTemplate添加拦截器：

```
props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, CustomProducerInterceptor.class.getName());
```

RecordMetadata

`RecordMetadata` 是 Kafka 生产者拦截器中一个重要的对象，它记录了消息发送成功后的元数据信息。

RecordMetadata 的主要字段

(1) `topic()`

- **含义**: 消息被发送到的主题名称。
- **用途**: 用于确认消息被发送到哪个主题。

(2) `partition()`

- **含义**: 消息被分配到的分区编号。
- **用途**: 用于确认消息被发送到哪个分区。

(3) `offset()`

- **含义**: 消息在分区中的偏移量 (Offset)。
- **用途**
 - 偏移量是 Kafka 中消息的唯一标识符，表示该消息在分区中的位置。
 - 可以用于追踪消息的位置，或者在消费者端定位消息。

(4) `timestamp()`

- **含义**: 消息的时间戳。
- **用途**
 - 如果 Kafka 主题启用了时间戳功能，时间戳可以表示消息的创建时间或日志追加时间。
 - 时间戳对于基于时间的处理（如窗口操作、延迟消息等）非常有用。

(5) `serializedKeysize()` 和 `serializedvaluesize()`

- **含义**
 - `serializedKeysize()`: 消息键的序列化大小（以字节为单位）。
 - `serializedvaluesize()`: 消息值的序列化大小（以字节为单位）。
- **用途**
 - 用于监控消息的大小，帮助优化性能或排查问题。

RecordMetadata 的作用

`RecordMetadata` 提供了关于消息发送结果的详细信息，主要用于以下场景：

(1) 确认消息发送成功

- 当生产者成功将消息发送到 Kafka 集群后，Kafka 会返回 `RecordMetadata` 对象，表示消息已经成功写入某个分区。
- 这是生产者确认消息发送成功的标志。

(2) 消息追踪

- `RecordMetadata` 中的 `topic`、`partition` 和 `offset` 信息可以用来唯一标识一条消息。
- 在分布式系统中，这些信息可以帮助你追踪消息的流向和状态。

(3) 性能监控

- `serializedkeysize` 和 `serializedvaluesize` 可以用来监控消息的大小，帮助识别潜在的性能瓶颈（例如消息过大导致网络传输变慢）。

(4) 自定义逻辑

- 在拦截器中，你可以基于 `RecordMetadata` 实现自定义逻辑。例如：
 - 记录消息发送的日志。
 - 统计每个主题或分区的消息发送量。
 - 根据偏移量实现幂等性检查。

如果消息发送失败,metadata为null

消费者

在上面已经简单使用了KafkaListener进行消息的消费

```
@Component
public class EventConsumer {
    //采用监听的方式接收事件（消息）
    //指定订阅的topic和消费者所在消费者组
    @KafkaListener(topics = "helloTopic", groupId = "helloGroup")
    public void onEvent(@Payload String message) {
        System.out.println("读取到的消息：" + message);
    }
}
```

采用kafkaListener的消费者，其监听的topic如果不存在，在启动项目时会自动创建

`@Payload` 注解可以用来标记一个参数作为从 Kafka 接收到的消息体（即有效负载）的映射目标。这样做的好处是提高了代码的可读性和清晰度，明确指出哪些参数是用来处理消息内容的。

虽然也可以标记在ConsumerRecord参数上，但是最好就标注在消息体对应的参数上。

接收消息

接收消息头内容

```

@Component
public class EventConsumer {
    //采用监听的方式接收事件（消息）
    //指定订阅的topic和消费者所在消费者组
    @KafkaListener(topics="helloTopic",groupId = "helloGroup")
    public void onEvent(@Payload String message,
                        @Header(value= KafkaHeaders.RECEIVED_TOPIC) String
topic,
                        @Header(value= KafkaHeaders.RECEIVED_KEY) String key,
                        @Header(value= KafkaHeaders.RECEIVED_PARTITION) String
partition) {
        System.out.println("读取到的消息：" + message+",topic:"+topic+",key:"+key+",partition:"+partition);
    }
}

```

通过@Header注解获取消息头信息

接收消息所有内容(ConsumerRecord)

```

@KafkaListener(topics="helloTopic",groupId = "helloGroup")
public void onEvent(ConsumerRecord<Object, Object> record) {
    System.out.println(record.toString());
    System.out.println(record.key() + " " + record.value() + " "
record.partition() + " " + record.offset());
}

```

*接收对象消息

直接在参数中写User类的参数，无法获得发送的User对象

```

producer:
  value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

consumer:
  value-serializer:
    org.springframework.kafka.support.serializer.JsonDeserializer

```

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
</dependency>

```

即便这样，还是报错，原因是user不受信任，在序列化、反序列化时出错。

那我们只能采用转化为String再发送的策略

上面两个序列化配置就可以删掉了,但还是要引入依赖

```
public class JSONutils {  
  
    private static final ObjectMapper mapper = new ObjectMapper();  
  
    public static String toJSON(Object obj) {  
        try {  
            return mapper.writeValueAsString(obj);  
        } catch (JsonProcessingException e) {  
            throw new RuntimeException(e);  
        }  
    }  
  
    public static <T> T toBean(String json, Class<T> clazz) {  
        try {  
            return mapper.readValue(json, clazz);  
        } catch (JsonProcessingException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

```
public void sendEvent2() {  
    User user= new User();  
    user.setBirthday(new Date());  
    user.setId(1208);  
    user.setPhone("1111111111");  
    String userJSON= JSONutils.toJSON(user);  
    kafkaTemplate2.send("helloTopic",userJSON);  
}
```

```
@KafkaListener(topics="helloTopic",groupId = "helloGroup")  
public void onEvent(String userJSON) {  
    User user= JSONutils.toBean(userJSON, User.class);  
    System.out.println(user.toString());  
}
```

监听器参数引用配置文件的配置

```
spring:  
kafka:  
topic:  
    name: helloTopic  
group:  
    name: helloGroup
```

```
@KafkaListener(topics="${spring.kafka.topic.name}", groupId =
"${spring.kafka.group.name}")
public void onEvent4(String userJSON) {
    User user= JSONUtils.toBean(userJSON, User.class);
    System.out.println(user.toString());
}
```

监听器手动确认消息

```
#开启消息监听的手动确认模式
listener:
  ack-mode: manual
```

```
@KafkaListener(topics="${spring.kafka.topic.name}", groupId =
"${spring.kafka.group.name}")
public void onEvent4(String userJSON,
                     Acknowledgment ack) {
    //收到消息后，处理业务
    User user= JSONUtils.toBean(userJSON, User.class);
    System.out.println(user.toString());
    //业务处理完成，给kafka服务器手动确认
    //手动确认消息，就是告诉kafka服务器，消息已经收到，默认情况下kafka是自动确认
    ack.acknowledge();
}
```

如果开启手动确认模式后，没有进行手动确认，那么就会出现 offset 未更新的情况，可能会导致消息重复消费。

利用这种性质，我们可以将接收到消息之后的业务处理和确认消息放在try中，如果在消息处理的时候出现异常，将不会确认消息。这样下次还能接收消息并进行业务处理。

```
@KafkaListener(topics="${spring.kafka.topic.name}", groupId =
"${spring.kafka.group.name}")
public void onEvent4(String userJSON,
                     Acknowledgment ack) {
    try {
        //收到消息后，处理业务
        User user= JSONUtils.toBean(userJSON, User.class);
        System.out.println(user.toString());
        //业务处理完成，给kafka服务器手动确认
        //手动确认消息，就是告诉kafka服务器，消息已经收到，默认情况下kafka是自动确认
        ack.acknowledge();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

指定topic,partition,offset

```

    @KafkaListener(groupId = "${spring.kafka.group.name}",
        topicPartitions = {
            @TopicPartition(
                topic="${spring.kafka.topic.name}",
                partitions = {"0", "1", "2"},
                partitionOffsets = {
                    @PartitionOffset(partition = "3",initialOffset =
"3"),
                    @PartitionOffset(partition = "4",initialOffset =
"3")
                }
            )
        }
    public void onEvent5(String userJSON,
        Acknowledgment ack) {
        try {
            //收到消息后，处理业务
            User user= JSONUtils.toBean(userJSON, User.class);
            System.out.println(user.toString());
            //业务处理完成，给kafka服务器手动确认
            //手动确认消息，就是告诉kafka服务器，消息已经收到，默认情况下kafka是自动确认
            ack.acknowledge();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

这个注解指定了消费者的组名，并监听这个topic中的0,1,2分区的所有消息，监听3,4分区从offset3开始的消息。

如果先开启监听，再发送消息，012分区能被消费所有的消息，34分区从3开始消费。

如果先发送消息，再开启监听，此时如果没设置auto-offset-reset=earliest，那么012不会消费消息，因为默认从最新开始消费。而34分区能消费3offset后的消息。

而这五个分区仍然遵守消息确认就移动该消费者组的offset，offset前的消息该消费者组中的消费者就不再消费了。

批量消费信息

- 设置配置文件开启批量消费

- 设置批量消费

```
spring.kafka.listener.type=batch
```

- 批量消费每次最多消费多少条消息

```
spring.kafka.consumer.max-poll-records=100
```

- 接收消息时用List来接收

```

@Component
public class EventConsumer {

    @KafkaListener(topics = "batchTopic", groupId = "batchGroup")
    public void onEvent(List<ConsumerRecord<String, String>> records) {
        System.out.println("批量消费, records="+records+"size="+records.size());
    }

}

```

*消息拦截器

在消息消费之前，我们可以通过配置拦截器对消息进行拦截，在消息被实际处理之前对其进行一些操作，例如日志记录、修改消息内容或安全检查。

1.实现kafka的ConsumerInterceptor拦截器接口

```

public class CustomConsumerInterceptor implements ConsumerInterceptor<String,
String> {

    //在拿到消息后、消费者拿到消息前执行
    //主要用于对即将返回给消费者的记录进行预处理或修改
    @Override
    public ConsumerRecords<String, String> onConsume(ConsumerRecords<String,
String> consumerRecords) {
        System.out.println("onConsume执行"+consumerRecords);
        return consumerRecords;
    }

    //在提交偏移量后调用该方法
    //onCommit 方法的作用是用来处理提交后的逻辑，例如记录日志、更新监控指标等。
    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> map) {
        System.out.println("onCommit执行"+map);
    }

    @Override
    public void close() {

    }

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

    }
}

```

2.在kafka消费者的ConsumerFactory配置中注册这个拦截器:

```

props.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,CustomConsumerInterceptor.class
.getName())

```

因为spring容器中由默认的consumerFactory和ListenerFactory

这就需要配置一个ConsumerFactory覆盖容器中默认的ConsumerFactory。并向其添加拦截器。

再配置一个自定义的ListenerFactory（需要用到ConsumerFactory），但我们却发现容器中就会出现两个ListenerFactory。解决方法是在@KafkaListener注解中指定ListenerFactory

```
spring:  
  kafka:  
    consumer:  
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer  
      value-deserializer:  
        org.apache.kafka.common.serialization.StringDeserializer
```

自定义ConsumerFactory和ListenerFactory：

```
@Configuration  
public class KafkaConfig {  
    @Value("${spring.kafka.bootstrap-servers}")  
    private String bootstrapServers;  
    @Value("${spring.kafka.consumer.value-deserializer}")  
    private String valueSerializer;  
    @Value("${spring.kafka.consumer.key-deserializer}")  
    private String keySerializer;  
  
    //消费者相关配置信息  
    public Map<String, Object> consumerConfigs(){  
        Map<String, Object> props = new HashMap<>();  
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);  
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, keySerializer);  
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,  
        valueSerializer);  
        props.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,  
        CustomConsumerInterceptor.class.getName());  
        return props;  
    }  
  
    //消费者工厂  
    @Bean  
    public ConsumerFactory<String, String> ourConsumerFactory() {  
        //这里的泛型也是消息的键和值  
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());  
    }  
    @Bean  
    public KafkaListenerContainerFactory<?>  
    ourKafkaListenerContainerFactory(ConsumerFactory<String, String>  
    ourConsumerFactory) {  
        ConcurrentKafkaListenerContainerFactory<String, String>  
        listenerContainerFactory = new ConcurrentKafkaListenerContainerFactory<>();  
        listenerContainerFactory.setConsumerFactory(ourConsumerFactory);  
        return listenerContainerFactory;  
    }  
}
```

消费者使用自定义ListenerFactory

```

@Component
public class EventConsumer {
    @KafkaListener(topics = "interceptorTopic", groupId =
"interceptorGroup", containerFactory = "ourKafkaListenerContainerFactory")
    public void onEvent(ConsumerRecord<String, String> record) {
        System.out.println("消息消费" + record);
    }
}

```

测试:

生产者:

```

@Component
public class EventProducer {
    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate2;

    public void sendEvent3() {
        User user = new User();
        user.setBirthday(new Date());
        user.setId(1208);
        user.setPhone("11111111");
        String userJSON = JSONUtils.toJSONString(user);
        kafkaTemplate2.send("interceptorTopic", "key" + i, userJSON);
    }
}

```

测试方法:

```

@SpringBootTest
class SpringBoot03KafkaBaseApplicationTests {
    @Autowired
    private EventProducer eventProducer;
    @Test
    void test01() {
        eventProducer.sendEvent3();
    }
}

```

消息转发

消息转发就是应用A从TopicA收到消息，经过处理后转发到TopicB，再由应用B监听接收该消息，即一个应用处理完成后将该消息转发至其他应用处理，这在实际开发中，是可能存在的

使用@SendTo注解实现

```

@Component
public class EventConsumer {
    @KafkaListener(topics = "topicA", groupId = "aGroup")
    @SendTo(value="topicB")
    public String onEventA(ConsumerRecord<String, String> record) {

```

```

        System.out.println("A消息消费"+record);
        return record.value()+"--forward message";
    }

    @KafkaListener(topics = "topicB",groupId = "bGroup")
    public void onEventB(ConsumerRecord<String, String> record) {
        System.out.println("B消息消费"+record);
    }
}

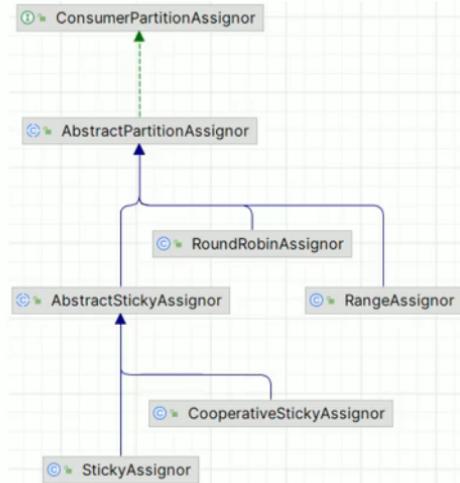
```

消息消费时的分区策略

指的是Topic中哪些分区由哪些消费者来消费

➤ 消息消息的分区策略

- Kafka有多种分区分配策略，默认的分区分配策略是 [RangeAssignor](#)，除了 [RangeAssignor](#) 策略外，
Kafka还有其他分区分配策略：
- [RoundRobinAssignor](#)、
➤ [StickyAssignor](#)
➤ [CooperativeStickyAssignor](#)，
- 这些策略各有特点，可以根据实际的应用场景和需求
来选择适合的分区分配策略；



默认分区策略

➤ 消息消息的分区策略

- Kafka默认的消费分区分配策略：[RangeAssignor](#)；假设如下：
 - 一个主题myTopic有10个分区：(p0 - p9)
 - 一个消费者组内有3个消费者：consumer1、consumer2、consumer3；
- [RangeAssignor](#)消费分区策略：
 - 1、计算每个消费者应得的分区数：分区总数 (10) / 消费者数量 (3) = 3 ... 余1；
 ➤ 每个消费者理论上应该得到3个分区，但由于有余数1，所以前1个消费者会多得到一个分区；
 ➤ consumer1（作为第一个消费者）将得到 $3 + 1 = 4$ 个分区；
 ➤ consumer2 和 consumer3 将各得到 3 个分区；
 - 2、具体分配：分区编号从0到9，按照编号顺序为消费者分配分区：
 ➤ consumer1 将分配得到分区 0、1、2、3；
 ➤ consumer2 将分配得到分区 4、5、6；
 ➤ consumer3 将分配得到分区 7、8、9；

[RangeAssignor](#)策略是根据消费者组内的消费者数量和主题的分区数量，来均匀地为每个消费者分配分区。

不同的消费者组之间是相互独立的，它们会各自独立地消费 Topic 中的所有分区消息。也就是说，一个消费者组内的消费者只会与该组内的其他消费者共享分区，而不会与其他消费者组发生冲突或竞争。

代码演示：

```

@Configuration
public class KafkaConfig {
    @Bean
    public NewTopic newTopic() {
        return new NewTopic("myTopic", 10, (short) 1);
    }
}

```

```

@Component
public class EventConsumer {
    //concurrency参数代表消费者组中消费者个数,
    @KafkaListener(topics = "myTopic", groupId = "myGroup3", concurrency = "3")
    public void onEvent(ConsumerRecord<String, String> record) {
        //这里可以把每个消费者当作一个线程
        System.out.println(Thread.currentThread().getId()+"-----消费消息-----" +
record);
    }
}

```

```

@Component
public class EventProducer {
    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate2;
    public void sendEvent3() {
        for (int i = 0; i < 100; i++) {
            User user= new User();
            user.setBirthday(new Date());
            user.setId(1208);
            user.setPhone("111111111");
            String userJSON= JSONUtils.toJSONString(user);
            kafkaTemplate2.send("myTopic", "key"+i ,userJSON);
            //注意这里key要不同, 否则会发到一个分区
        }
    }
}

```

经测试上面图片中的方法是正确的。

RoundRobinAssignor

```

@Configuration
public class KafkaConfig {
    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServers;
    @Value("${spring.kafka.consumer.value-deserializer}")
    private String valueSerializer;
    @Value("${spring.kafka.consumer.key-deserializer}")
    private String keySerializer;
    @Value("${spring.kafka.consumer.auto-offset-reset}")
    private String autoOffsetReset;

    //消费者相关配置信息
    public Map<String, Object> consumerConfigs(){

```

```

        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, keySerializer);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        valueSerializer);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, autoOffsetReset);
        //指定使用轮询分区器
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
RoundRobinAssignor.class.getName());
        return props;
    }
    //消费者工厂
    @Bean
    public ConsumerFactory<String, String> ourConsumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    public KafkaListenerContainerFactory<?>
ourKafkaListenerContainerFactory(ConsumerFactory<String, String>
ourConsumerFactory) {
        ConcurrentKafkaListenerContainerFactory<String, String>
listenerContainerFactory = new ConcurrentKafkaListenerContainerFactory<>();
        listenerContainerFactory.setConsumerFactory(ourConsumerFactory);
        return listenerContainerFactory;
    }

    @Bean
    public NewTopic newTopic() {
        return new NewTopic("myTopic", 10, (short) 1);
    }
}

```

```

@Component
public class EventConsumer {
    //concurrency参数代表消费者组中消费者个数，并指定自定义的ListenerFactory
    @KafkaListener(topics = "myTopic", groupId = "myGroup3", concurrency =
"3", containerFactory = "ourKafkaListenerContainerFactory")
    public void onEvent(ConsumerRecord<String, String> record) {
        //这里可以把每个消费者当作一个线程
        System.out.println(Thread.currentThread().getId()+"-----消费消息-----" +
record);
    }
}

```

经测试，这是完全符合轮询策略的。

两个sticky分区策略

➤ **StickyAssignor消费分区策略:**

- 尽可能保持消费者与分区之间的分配关系不变，即使消费组的消费者成员发生变化，减少不必要的分区重分配；
- 尽量保持现有的分区分配不变，仅对新加入的消费者或离开的消费者进行分区调整。这样，大多数消费者可以继续消费它们之前消费的分区，只有少数消费者需要处理额外的分区；所以叫“**粘性**”分配；

➤ **CooperativeStickyAssignor消费分区策略:**

- 与 StickyAssignor 类似，但增加了对协作式重新平衡的支持，即消费者可以在它离开消费者组之前通知协调器，以便协调器可以预先计划分区迁移，而不是在消费者突然离开时立即进行分区重分配；

建议

建议使用两个粘性分区策略中的一种

消息的存储

- kafka的所有事件(消息、数据)都存储在 /tmp/kafka-logs 目录中，可通过 log.dirs=/tmp/kafka-logs 配置；
- Kafka的所有事件(消息、数据)都是以日志文件的方式来保存；
- Kafka一般都是海量的消息数据，为了避免日志文件过大，日志文件被存放在多个日志目录下，日志目录的命名规则为：<topic_name>-<partition_id>；
- 比如创建一个名为 firstTopic 的 topic，其中有 3 个 partition，那么在 kafka 的数据目录 (/tmp/kafka-log) 中就有 3 个目录，firstTopic-0、firstTopic-1、firstTopic-2；
 - 00000000000000000000.index 消息索引文件
 - 00000000000000000000.log 消息数据文件
 - 00000000000000000000.timeindex 消息的时间戳索引文件
 - 00000000000000000006.snapshot 快照文件，生产者发生故障或重启时能够恢复并继续之前的操作
 - leader-epoch-checkpoint 记录每个分区当前领导者的epoch以及领导者开始写入消息时的起始偏移量
 - partition.metadata 存储关于特定分区的元数据 (metadata) 信息

这里我用的是docker启动的，存储的目录需要进入docker容器后，进入 /tmp/kraft-combined-logs 文件夹

__consumer_offsets内部主题

刚才的目录中，除了有存储消息相关数据的文件夹，还有 __consumer_offsets 文件夹。它是自动创建的内部主题 __consumer_offsets 对应的文件夹。用来存放各消费者组的 offset 信息。

这个文件夹默认有50个。

每次消费一个消息并提交后，会保存当前消费消息的下一个offset。

消费者提交的offset信息会写入该消费者组对应的 __consumer_offsets 目录中。

消费者组的offset信息存放在哪个 __consumer_offsets 文件夹的计算公式：

```
Math.abs("groupid".hashCode())%groupMetadataTopicPartitionCount
```

在idea的kafka插件中打开 显示内部主题 选项即可显示此内部主题。但是要真正查看文件内容还是要用命令行

offset

生产者offset

生产者发送一条消息到kafka的broker的某个topic的某个partition中

kafka内部会为每条消息分配一个唯一的offset，该offset就是该消息在partition中的位置

生产者的offset不用进行设置，由服务器端自动管理

消费者组offset

每个消费者组会独立的维护自己的offset，当消费者从某个partition读取消息时，它会记录当前读取到的offset。这样即使崩溃或重启，也可以从上次读取的位置继续读取，而不会重复读取或遗漏消息。

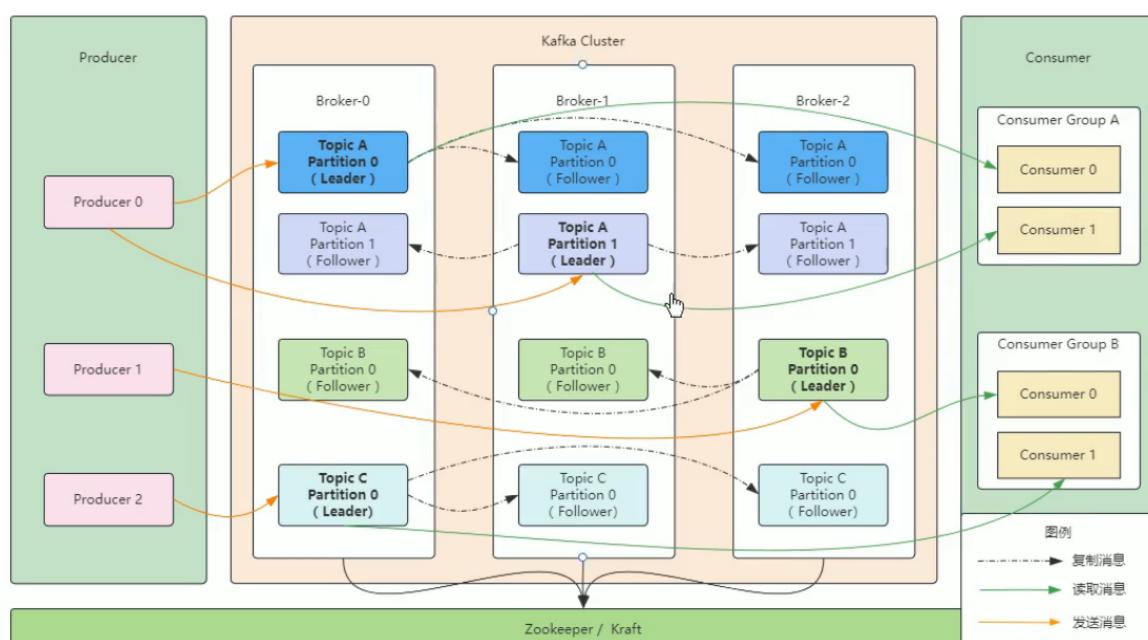
消费者消费消息后，如果不提交确认（ack），则offset不更新。

可以通过 `/opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group 组名` 命令查看消费者组的offset。

###

Kafka集群

Kafka集群架构



副本个数不能为0，也不能大于broker个数

主副本和从副本不能在同一个broker上，主副本加从副本=replica个数

主副本在哪个broker上，是由kafka内部机制决定的

概念

- broker: kafka服务器
- Topic: 主题
- event: 消息 (message、事件、数据)
- 生产者: producer
- 消费者: consumer
- 消费者组: consumer group
- 分区: partition
- 偏移量: offset
- replica: 副本 (leader replica 和follower replica)
- ISR副本 (In-Sync Replicas) :
- LEO:
- HW:

ISR副本

In-Sync Replicas 在同步中的副本，包含了Leader副本和所有与Leader副本保持同步的Follower副本。

- 写请求首先由 Leader 副本处理，之后 Follower 副本会从 Leader 上拉取写入的消息，这个过程会有一定的延迟，导致 Follower 副本中保存的消息略少于 Leader 副本，但是只要没有超出阈值都可以容忍，但是如果一个 Follower 副本出现异常，比如宕机、网络断开等原因长时间没有同步到消息，那这个时候，Leader就会把它踢出去，Kafka 通过ISR集合来维护一个 “可用且消息量与Leader相差不多的副本集合，它是整个副本集合的一个子集”；
- 在Kafka中，一个副本要成为ISR (In-Sync Replicas) 副本，需要满足一定条件：
 - 1、Leader副本本身就是一个ISR副本；
 - 2、Follower副本最后一条消息的offset与Leader副本的最后一条消息的offset之间的差值不能超过指定的阈值，超过阈值则该Follower副本将会从ISR列表中剔除；
 -) ➤ replica.lag.time.max.ms: 默认是30秒；如果该follower在此时间间隔内一直没有追上过Leader副本的所有消息，则该Follower副本就会被剔除ISR列表；
 - replica.lag.messages: 落后了多少条消息时，该Follower副本就会被剔除ISR列表，该配置参数现在在新版本的Kafka已经过时了；

LEO

Log End Offset 日志末端偏移量，指的是某个副本（无论是主副本还是跟随副本）当前日志末尾的消息偏移量。换句话说，它是该副本上最新写入的消息的下一个位置。LEO 标识了每个副本当前已写入的最大偏移量。也就是说，如果LEO=10，那么该副本保存了[0,9]的10条消息。

HW

High Watermark 高水位值，HW 是所有 **ISR 成员副本中最小的 LEO**。这意味着 HW 表示的是消费者可以安全读取的最大偏移量。只有当 ISR 中的所有副本都确认接收到某条消息时，这条消息的偏移量才会被包含在 HW 中。

基于Kraft搭建kafka集群

controller节点：

Controller节点在Kafka集群中扮演着管理和协调的角色，管理整个集群中所有分区和副本的状态，当某个分区的leader副本出现故障时，Controller负责为该分区选举新的leader副本；

Broker节点在Kafka集群中主要承担消息存储和转发等任务；

- **Broker 角色**: 处理客户端请求（生产者和消费者的请求），负责存储和管理分区数据。
- **Controller 角色**: 负责管理集群的元数据（如主题、分区、副本分配等），协调分区的领导者选举和故障恢复。

zookeeper搭建集群的controller选举方式：

集群有三个节点都是Broker角色，其中一个Broker（褐色）是Controller控制器节点，控制器节点将集群元数据信息（比如主题分类、消费进度等）保存到zookeeper，用于集群各节点之间分布式交互；

kraft搭建集群的controller选举方式：

一个集群有四个Broker节点，人为指定其中三个作为Controller角色（蓝色），从三个Controller中选举出一个Controller作为主控制器（褐色），其它2个备用，Zookeeper不再被需要，相关的元数据信息以kafka日志的形式存在（即：以消息队列消息的形式存在）；

kraft建议选举三个controller节点

docker-compose.yml文件

kafka-kraft-cluster 目录下创建data目录，创建docker-compose.yml文件

```
services:  
  kafka1:  
    image: bitnami/kafka:3.7  
    container_name: kafka1  
    hostname: kafka1  
    ports:  
      - "19092:9092"  
    environment:  
      - KAFKA_KRAFT_CLUSTER_ID=kraft-cluster-id  
      - KAFKA_CFG_NODE_ID=1  
      - KAFKA_CFG_PROCESS_ROLES=broker,controller  
      -  
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka1:9093,2@kafka2:9093,3@kafka3:9093  
      - KAFKA_CFG_LISTENERS=INTERNAL://:9092,CONTROLLER://:9093
```

```

- KAFKA_CFG_ADVERTISED_LISTENERS=INTERNAL://192.168.14.128:19092
-
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=INTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT
- KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
- KAFKA_CFG_INTER_BROKER_LISTENER_NAME=INTERNAL
volumes:
- ./data/kafka1:/bitnami/kafka
networks:
- kafka-net

kafka2:
image: bitnami/kafka:3.7
container_name: kafka2
hostname: kafka2
ports:
- "19093:9092"
environment:
- KAFKA_KRAFT_CLUSTER_ID=kraft-cluster-id
- KAFKA_CFG_NODE_ID=2
- KAFKA_CFG_PROCESS_ROLES=broker,controller
-
KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka1:9093,2@kafka2:9093,3@kafka3:9093
- KAFKA_CFG_LISTENERS=INTERNAL://:9092,CONTROLLER://:9093
- KAFKA_CFG_ADVERTISED_LISTENERS=INTERNAL://192.168.14.128:19093
-
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=INTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT
- KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
- KAFKA_CFG_INTER_BROKER_LISTENER_NAME=INTERNAL
volumes:
- ./data/kafka2:/bitnami/kafka
networks:
- kafka-net

kafka3:
image: bitnami/kafka:3.7
container_name: kafka3
hostname: kafka3
ports:
- "19094:9092"
environment:
- KAFKA_KRAFT_CLUSTER_ID=kraft-cluster-id
- KAFKA_CFG_NODE_ID=3
- KAFKA_CFG_PROCESS_ROLES=broker,controller
-
KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka1:9093,2@kafka2:9093,3@kafka3:9093
- KAFKA_CFG_LISTENERS=INTERNAL://:9092,CONTROLLER://:9093
- KAFKA_CFG_ADVERTISED_LISTENERS=INTERNAL://192.168.14.128:19094
-
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=INTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT
- KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
- KAFKA_CFG_INTER_BROKER_LISTENER_NAME=INTERNAL
volumes:
- ./data/kafka3:/bitnami/kafka
networks:
- kafka-net

networks:
kafka-net:

```

```
driver: bridge
```

记得要让防护墙开放端口。

远程就可以通过 192.168.14.128:19094/192.168.14.128:19093/192.168.14.128:19092 连接上 kafka 集群。