# 生产者

## 一．发送消息的3种方式

1. 发送并忘记（fire-and-forget）:只发送不关心是否正常到达。（第二个参数设置为null，即不设置回调函数）

代码实现：producer.send(record)

1. 同步发送: 调用send()方法，它返回一个Future对象，调用get()方法进行等待，就可以知道消息是否发送成功。

代码实现:producer.send(record).get()

1. 异步发送: 调用send()方法，指定一个回调函数，服务器在返回响应时调用该函数

代码实现: producer.send(record,new DemoProducerCallBack())

其中DemoProducerCallBack类实现Callback接口

注： ”发送并忘记“和“异步发送”本质上都是异步发送，区别只在于是否传入Callback的实现类对象。

## 关于三种发送方式的部分应用

说明：在这里我们对个别参数做出了说明，因此在下述情况中使用时就不进行赘述了。

①关于max.in.flight.requests.per.connection参数，它决定了生产者在收到服务器响应之前能发送多少批次。当其设置为1时，相当于同步发送。当它设为大于1的数值时，对于同步发送不生效。

②关于acks参数，它指定了必须有多少个分区副本接受到消息，生产者才会认为消息写入是成功的。当其设置为0的情况下，无法保证服务器已收到记录，同时retries配置将不会生效（因为客户端通常不会知道任何故障）。

③关于retries参数，决定了生产者可以重发消息的次数。我们可以通过设置这个参数来有效的避免消息的丢失。当重发次数达到这个参数设置的次数，生产者会放弃重试并返回错误。

注：在一些要求满足无序且不丢失的场景下，我们可以将acks参数设为非0，将retries参数设为非0，同时将max.in.flight.requests.per.connection参数设置为大于1的数。这样的话，当第一个批次数据写入失败，而第二次批次数据写入成功，broker会重试写入第一批次的数据。如果第一批次的数据写入成功，那么两个批次的数据顺序就反了过来。

### 一．同步发送

1. 同步发送有序且消息不丢失。

①保证有序：由于每发送一条消息，便会等待服务器响应，因此同步发送情况 下肯定有序。

②尽量避免消息不丢失：设置acks>0,且retries>0

(2) 同步发送有序且消息可以丢失

①保证有序：同步发送保证有序

②不介意消息丢失：设置retries参数为0(或acks=0)

(3) 同步发送无序的情况不会发生。

由于同步发送一批消息之后，需要调用get方法，等待服务器响应正确消息或 者错误，既然这样，就一定保证有序。

### 二．异步发送

1. 异步发送不保证顺序且消息不丢失。

要求：

①设置acks非0，最好为all，

②设置retries参数非0，当产生异常时，做同步发送或者异步发送

③设置max.in.flight.requests.per.connection参数为多次

案例演示：

private class DemoProducerCallback implements Callback {

private String sendMsg = “”;

public DemoProducerCallback(String sendMsg ) {

this.sendMsg = sendMsg;

}

public DemoProducerCallback() {}

@Override

public void onCompletion(RecordMetadata recordMetadata,Exception e){

if(e != null) {

try {

producer.send(record).get();//同步发送

} catch (Exception e) {

e.printStackTrance();

}

}

}

}

ProducerRecord<String,String> record

= new ProducerRecord<>(“CustomerCountry”,”Biomedical Materials”,”USA”);

producer.send(record,new DemoProducerCallback());

1. 异步发送有序且消息不丢失

要求：

①设置acks非0，最好为all

②设置retries参数非0，并且多次，当产生异常时，做同步发送或者异步发送

③设置max.in.flight.requests.per.connection参数为1次

案例代码请参照标题（1），区别只在于max.in.flight.requests.per.connection设置不同。

1. 异步发送有序且可以允许消息丢失

要求：

①设置retries为0,消息没有发送成功也不进行重试。

②设置max.in.flight.requests.per.connection参数为1次。

案例演示：

private class DemoProducerCallback implements Callback {

private String sendMsg = “”;

public DemoProducerCallback(String sendMsg ) {

this.sendMsg = sendMsg;

}

public DemoProducerCallback() {}

@Override

public void onCompletion(RecordMetadata recordMetadata,Exception e){

e.printStackTrance();

}

}

ProducerRecord<String,String> record

= new ProducerRecord<>(“CustomerCountry”,”Biomedical Materials”,”USA”);

producer.send(record,new DemoProducerCallback());

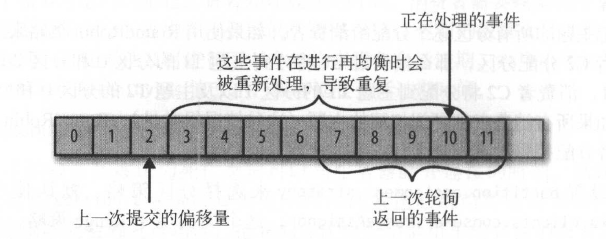
# 消费者

## 提交偏移量

### 一．提交偏移量可能会导致的后果

#### 1.消息的重复处理

如果提交的偏移量小于客户端处理的最后一个消息的偏移量，那么处于两个偏移量之间的消息就会被重复处理。



比如说：当消费者自动提交偏移量时，即enable.auto.commit设置为true，那么每过5秒,消费者会自动把poll()方法接收到的最大偏移量提交上去。提交时间间隔由auto.commit.interval.ms控制，默认值为5s。如果最近一次提交之后的3s发生了再均衡，再均衡之后，消费者就从最后一次提交的偏移量位置开始读取消息，那么这3s内到达的消息会被重复处理。

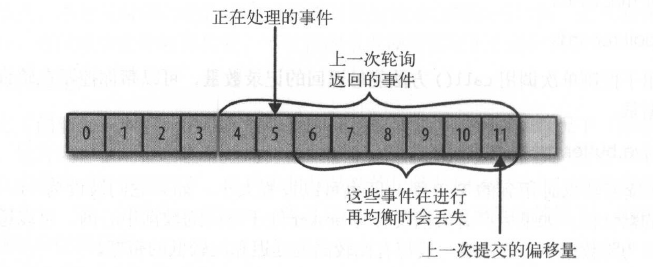
重复处理不光是在自动提交的时候产生，同步提交和异步提交的时候都会发生。当我们设置一次轮询同步提交或异步提交一次消息时，只有最后一批次的消息会被重复处理。而自动提交是基于时间间隔的，因此重复处理的消息批次是不定的。

解决方案：

为了避免再均衡时消息的重复处理，我们可以采用手动提交偏移量的方式，这样可以降低重复操作出现的概率，并且设置再均衡监听器。

#### 2.消息的丢失

如果提交的偏移量大于客户端处理的最后一个消息的偏移量，那么处于两个偏移量之间的消息将会丢失。



消费者出现丢失操作的情况：

①消费者手动提交特定偏移量(过大)后再执行消息处理

②当消费者所在服务器在执行消息阶段由于故障、电源等意外问题产生而导致宕机

...

解决方案：

先执行消息处理，再提交偏移量。

### 二．关于提交偏移量的方式

1. 自动提交。若enable.auto.commit为true时，那么每过5s，消费者会自动把从poll()方法接收到的最大偏移量提交上去。提交时间间隔由auto.commit.interval.ms控制，默认为5s。可能会造成重复消费和消费丢失。
2. 手动提交。手动提交包括同步提交和异步提交。需要把enable.auto.commit设为false。

#### 同步提交

特点：只要没有发生不可恢复的错误， commiSync()方法会一直尝试直至提交成功

1. 优点：简单可靠，提交poll()方法返回的最新偏移量，若成功则马上返回，提交失败就抛出异常。
2. 缺点：如果发生再均衡，从最近一批消息到发生再均衡之间的所有消息都将被重复处理。

案例演示：

//消费者从消费者产生的内容中进行消费

while(true) {

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

for (ConsumerRecord<String,String> record : records) {

//将消费到的内容数据到控制台，以便大家可以看到消费详情

System.out.printf("offset = %d, value = %s%n", record.offset(), record.value());

}

try {

//在每次处理完poll过来的数据后进行偏移量的同步提交

consumer.commitSync();

}catch (CommitFailedException e){

log.error("commit failed",e);

}

}

#### 异步提交

特点：commitAsync()只管发送提交请求，无需等待broker的响应。

1. 优点: 提升了吞吐量，效率高
2. 缺点：再均衡时会增加重复消息的数量

案例演示：

//消费者从消费者产生的内容中进行消费

while(true) {

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

for (ConsumerRecord<String,String> record : records) {

//将消费到的内容数据到控制台，以便大家可以看到消费详情

System.out.printf("offset = %d, value = %s%n", record.offset(), record.value());

}

try {

//在每次处理完poll过来的数据后进行偏移量的异步提交

consumer.commitSync();

}catch (CommitFailedException e){

log.error("commit failed",e);

}

}

#### 同步和异步组合提交

使用场景：在消费者关闭前一般会组合使用commitSync()和commitAsync()。

1. 优点：速度比同步更快，比异步更加可靠
2. 缺点：再均衡时会增加重复消息的数量

案例演示：

//消费者从消费者产生的内容中进行消费

try{

while(true) {

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

for (ConsumerRecord<String,String> record : records) {

//将消费到的内容数据到控制台，以便大家可以看到消费详情

System.out.printf("offset = %d, value = %s%n", record.offset(), record.value());

}

try {

//如果一切正常，我们使用commitAsync()方法来提交。这样速度更快，而且即使这次提交失败，下一次提交很可能会成功。

consumer.commitAsync();

}catch (CommitFailedException e){

log.error("commit failed",e);

}

}

}catch (Exception e){

log.error("Unexpected error",e);

}finally{

try{

//如果直接关闭消费者，就没有所谓的“下一次提交”了。使用commitSync()方法会一直重试，直到提交成功或发生无法恢复的错误。

consumer.commitSync();

}finally {

consumer.close();

}

}

#### 设置再均衡监听器

消费者在退出和进行分区再均衡之前，可以做一些清理工作，可以通过实现ComsumerRebalanceListener的onPartitionRevoked（再均衡开始之前和消费者停止读取消息之后被调用）和onPartitionAssigned（在重新分配分区之前和消费者开始读取消息之前调用）方法实现。可以帮助我们在失去分区所有权时提交偏移量（在onPartitionRevoked方法中实现）。

#### 利用数据库原子性保证不出现重复消费

应用程序从Kafka读取事件，对它们进行处理，然后把结果保存在数据库、NoSQL存储引擎或者Hadoop。可以保证不丢失数据，也不在数据库保存重复数据。