目录

[一．Kafka体系架构 1](#_Toc528608888)

[二．kafka生产者——向kafka写入数据 2](#_Toc528608889)

[1.创建kafka生产者 2](#_Toc528608890)

[2.发送消息主要有以下3种方式 2](#_Toc528608891)

[3.生产者的配置 3](#_Toc528608892)

[4.序列化器 5](#_Toc528608893)

[5.分区 5](#_Toc528608894)

[三．kafka消费者——从kafka读取数据 6](#_Toc528608895)

[1.消费者和消费者群组 6](#_Toc528608896)

[2.创建kafka消费者 7](#_Toc528608897)

[3.订阅主题 8](#_Toc528608898)

[4.轮询 8](#_Toc528608899)

[5.消费者的配置 9](#_Toc528608900)

[6.提交和偏移量 10](#_Toc528608901)

[1）自动提交 10](#_Toc528608902)

[2）提交当前偏移量 11](#_Toc528608903)

[3）异步提交 11](#_Toc528608904)

[4)同步和异步组合提交 12](#_Toc528608905)

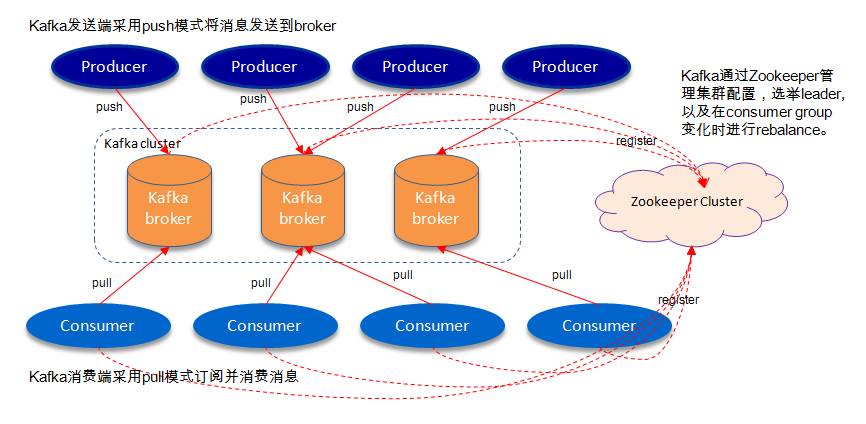
[7.再均衡监听器 12](#_Toc528608906)

[8.如何退出 13](#_Toc528608907)

[9.反序列化器 14](#_Toc528608908)

[四．Kafka数据高可靠性深度解读 14](#_Toc528608909)

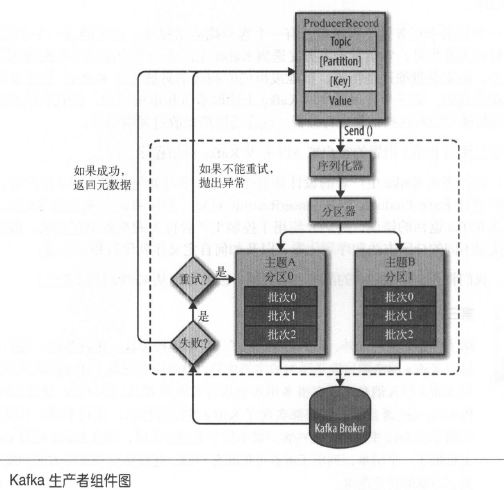
# 一．Kafka体系架构



如上图所示，一个典型的Kafka体系架构包括若干Producer（可以是服务器日志，业务数据，页面前端产生的page view等等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干Consumer (Group)，以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在consumer group发生变化时进行rebalance。Producer使用push(推)模式将消息发布到broker，Consumer使用pull(拉)模式从broker订阅并消费消息



# 二．kafka生产者——向kafka写入数据



## 1.创建kafka生产者

Kafka生产者有3个必选的属性

1）bootstrap.servers：该属性指定broker的地址清单，地址的格式为host:port。

2）key.serializer：

必须被设置为一个实现了org.apache.kafka.common.serialization.Serializer接口的类，生产者会使用这个类把键对象序列化成字节数组。

3）value.serializer：与key.serializer一样，value.serializer指定的类会将值序列化。

## 2.发送消息主要有以下3种方式

1）发送并忘记：我们把消息发送给服务器，但井不关心它是否正常到达。

ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {

producer.send(record);

} catch (Exception e) {

e.printStackTrace();

}

2）同步发送：我们使用send()方法发送消息，它会返回一个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。producer.send(record).get();

3）异步发送：我们调用send() 方怯，并指定一个回调函数， 服务器在返回响应时调用该函数。

private class DemoProducerCallback implements Callback {

@Override

public void onCompletion(RecordMetadata recordMetadata, Exception e) {

if ( e!=null) {

e.printStackTrace();

}

}

}

ProducerRecord<String,String> record = new ProducerRecord<>("CustomerCountry","Biomedical Materials","USA");

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

为了使用回调，需要一个实现了org.apache. kafka . clients. producer. Callback接口的类，这个接口只有一个onCollpletion 方法。

## 3.生产者的配置

生产者还有很多可配置的参数，它们大部分都有合理的默认值， 所以没有必要去修改它们。不过有几个参数在内存使用、性能和可靠性方面对生产者影响比较大，接下来我们会一一说明。

**1.acks**

acks 参数指定了必须要有多少个分区副本收到消息，生产者才会认为消息写入是成功的。

如果acks=0，生产者在成功写入悄息之前不会等待任何来自服务器的响应。

如果acks=1，只要集群的首领节点收到消息，生产者就会收到一个来自服务器的成功响应。

如果acks=all，只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。

**2.buffer.memory**

该参数用来设置生产者内存缓冲区的大小，生产者用它缓冲要发送到服务器的消息。

**3.compression.type**

默认情况下，消息发送时不会被压缩。该参数可以设置为snappy 、gzip 或lz4 ，它指定了消息被发送给broker之前使用哪一种压缩算也进行压缩。

**4.reties**

生产者从服务器收到的错误有可能是临时性的错误（比如分区找不到lead）。在这种情况下，retries 参数的值决定了生产者可以重发消息的次数，如果达到这个次数，生产者会放弃重试并返回错误。

**5.batch.size**

当有多个消息需要被发送到同一个分区时，生产者会把它们放在同一个批次里。该参数指定了一个批次可以使用的内存大小，按照字节数计算（而不是消息个数）。当批次被填满，批次里的所有消息会被发送出去。

**6.linger.ms**

该参数指定了生产者在发送批次之前等待更多消息加入批次的时间。KafkaProducer会在批次填满或linger. ms达到上限时把批次发送出去。默认情况下，只要有可用的线程，生产者就会把消息发送出去，就算批次里只有一个消息。

**7.client.id**

该参数可以是任意的字符串，服务器会用它来识别消息的来橱，还可以用在日志和配额指标里。

**8.max.in.flight.requests.per.connection**

该参数指定了生产者在收到服务器晌应之前可以发送多少个消息。它的值越高，就会占用越多的内存，不过也会提升吞吐量。把它设为1 可以保证消息是按照发送的顺序写入服务器的，即使发生了重试。

**9.time.out\request.timeout.ms和metadata.fetch.timeout.ms**

request.timeout.ms指定了生产者在发送数据时等待服务器返回响应的时间，meta data .fetch . timeout.ms 指定了生产者在获取元数据（比如目标分区的首领是谁）时等待服务器返回响应的时间。如果等待响应超时，那么生产者要么重试发送数据，要么返回一个错误（抛出异常或执行回调）。timeout.ms 指定了broker等待同步副本返回消息确认的时间，与asks 的配置相匹配一一如果在指定时间内没有收到同步副本的确认，那么broker 就会返回一个错误。

**10.max.block.ms**

该参数指定了在调用send() 方法或使用partitionsFor() 方法获取元数据时生产者的阻塞时间。当生产者的发送缓冲区已满，或者没有可用的元数据时，这些方法就会阻塞。在阻塞时间达到max.block.ms时，生产者会抛出超时异常

**11.max.request.size**

该参数用于控制生产者发送的请求大小。它可以指能发送的单个消息的最大值，也可以指单个请求里所有消息总的大小。另外，broker对可接收的消息最大值也有自己的限制message. max.Bytes)，所以两边的配置最好可以匹配，避免生产者发送的消息被broker 拒绝。

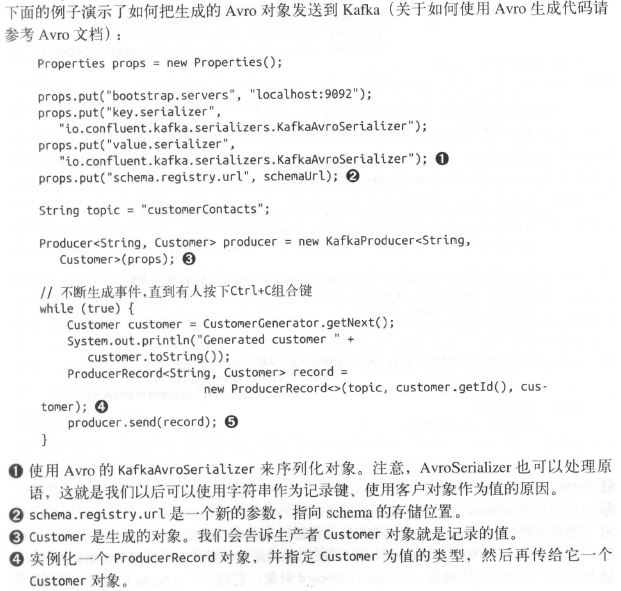
**12.receive.buffer.bytes和send.buffer.bytes**

这两个参数分别指定了TCP socket 接收和发送数据包的缓冲区大小。如果它们被设为-1 ，就使用操作系统的默认值。

## 4.序列化器

如果发送到Kafka 的对象不是简单的字符串或整型，那么可以使用序列化框架来创建消息记录，如Avro 、Thrift 或Protobuf ，或者使用自定义序列化器。我们强烈建议使用通用的序列化框架

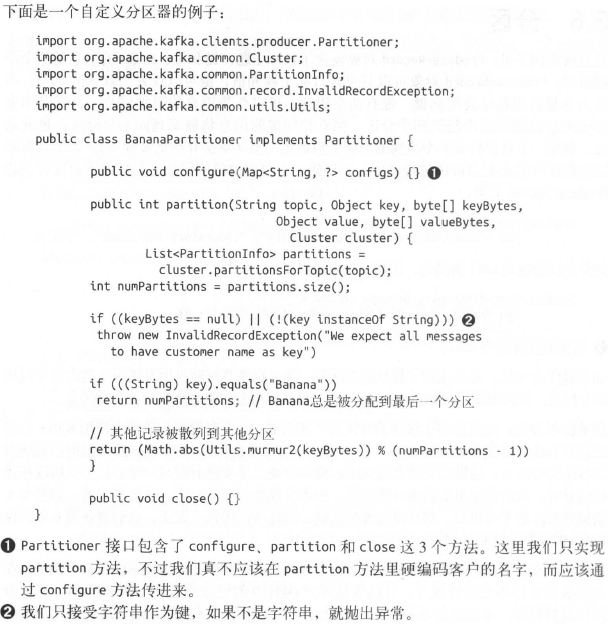
以使用Avro为例：



## 5.分区

在之前的例子里，ProducerRecord 对象包含了目标主题、键和值。键有两个用途：可以作为消息的附加信息，也可以用来决定消息该被写到主题的哪个分区。拥有相同键的消息将被写到同一个分区。如果键值为null，井且使用了默认的分区器，那么记录将被随机地发送到主题内各个可用的分区上。分区器使用轮询（ Round Robin ）算法将消息均衡地分布到各个分区上。

实现自定义分区策略：



# 三．kafka消费者——从kafka读取数据

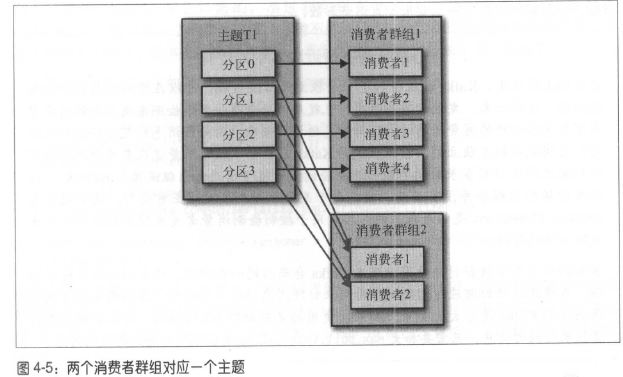
## 1.消费者和消费者群组

Kafka消费者从属于消费者群组。一个群组里的消费者订阅的是同一个主题，每个消费者接收主题一部分分区的消息。

往群组里增加消费者是横向伸缩消费能力的主要方式。Kafka 消费者经常会做一些高延迟的操作，比如把数据写到数据库或HDFS ，或者使用数据进行比较耗时的计算。在这些情况下，单个消费者无法跟上数据生成的速度，所以可以增加更多的消费者，让它们分担负载，每个消费者只处理部分分区的消息，这就是横向伸缩的主要手段。不过要注意，不要让消费者的数量超过主题分区的数量，多余的消费者只会被闲置。

还经常出现多个应用程序从同一个主题读取数据的情况。实际上， Kafka 设计的主要目标之一，就是要让Kafka 主题里的数据能够满足企业各种应用场景的需求。在这些场景里，每个应用程序可以获取到所有的消息，而不只是其中的一部分。只要保证每个应用程序有自己的消费者群组，就可以让它们获取到主题所有的消息。不同于传统的消息系统，横向伸缩Kafka 消费者和消费者群组并不会

对性能造成负面影响。



## 2.创建kafka消费者

创建KafkaConsumer对象与创建KafkaProducer对象非常相似一一把想要传给消费者的属性放在 Properties对象里。3 个必要的属性： bootstrap.server、key . deserialize和value. deserializer。

bootstrap.server指定了 Kafka 集群的连接字符串。它的用途与在KafkaProducer中的用途是一样的。key. deserialize和value. deserializer与生产者的 serializer定义也很类似，不过它们是使用指定的类把字节数组转成 Java 对象。

group.id不是必需的，不过我们现在姑且认为它是必需 的。它指定KafkaConsumer属于哪一个消费者群组。创建不属于任何一个群组的消费者也是可以的，只是这样做不太常见。

下面的代码片段演示了如何创建一个 KafkaConsumer

Properties props = new Properties();

props.put("bootstrap.servers","broker1:9092,broker2:9092");

props.put("group.id","CountryCounter");

props.put("key.deserializer","org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer","org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String,String> consumer = new KafkaConsumer<String,String>(props);

## 3.订阅主题

创建好消费者之后，下一步可以开始订阅主题了subscribe()方法接受一个主题列表作为参数，使用起来很简单：

consumer.subscribe(Collections.singletonList("customerCountries"))

为了简单起见，我们创建了一个只包含单个元素的列表，主题的名字叫作”customerCountries”

也可以在调用subscribe()方法时传入一个正则表达式。正则表达式可以匹配多个主题。类似于：consumer. subscribe(“test.\*”)；

## 4.轮询

消息轮询是消费者API的核心，通过一个简单的轮询向服务器请求数据。一旦消费者订阅了主题，轮询就会处理所有的细节，包括群组协调、分区再均衡、发送心跳和获取数据，开发者只需要使用一组简单的 API 来处理从分区返回的数据。消费者代码的主要部分如下:

try {

while(true) {

ConsumerRecords<String, String> records = consumer.poll(100);

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

log.debug("topic=%s, partition=%s, offset=%d, customer=%s,country=%s\n", record.topic(),record.patition(),record.offerset(),record.key(),record.value());

int updatedCount=1;

if(custCountryMap.countainsValue(record.value())) {

updatedCount = custCountryMap.get(record.value())+1;

}

custCountryMap.put(record.value(),updatedCount);

JSONObject json=new JSONObject(custCountryMap);

System.out.println(json.toString(4))

}

}

} finally {

consumer.close();

}

1.这是一个无限循环。消费者实际上是一个长期运行的应用程序，它通过持续轮询向Kafka 请求数据。

2.消费者必须持续对Kafka进行轮询，否则会被认为己经死亡，它的分区会被移交给群组里的其他消费者。传给poll()方法的参数是一个超时时间，用于控制poll()方法的阻塞时间（在消费者的缓冲区里没有可用数据时会发生阻塞）。如果该参数被设为 0, poll()会立即返回，否则它会在指定的毫秒数内一直等待broker 返回数据。

3.poll()方法返回一个记录列表。每条记录都包含了记录所属主题的信息、记录所在分区的信息、记录在分区里的偏移量，以及记录的键值对。我们一般会遍历这个列表 ，逐条处理这些记录。 Poll()方法有一个超时参数，它指定了方法在多久之后可以返回，不管有没有可用的数据都要返回。超时时间的设置取决于应用程序对响应速度的要求，比如要在多长时间内把控制权归还给执行轮询的线程。

4. 把结果保存起来或者对已有的记录进行更新，处理过程也随之结束。 在这里，我们的目的是统计来自各个地方的客户数量， 所以使用了一个散列表来保存结果，并JSON 的格式打印结果。在真实场景里，结果一般会被保存到数据存储系统里。

5. 在退出应用程序之前使用 close()方法关闭消费者。网络连接和 socket 也会随之关闭，并立即触发一次再均衡，而不是等待群组协调器发现它不再发送心跳并认定它已死亡，因为那样需要更长的时间，导致整个群组在一段时间内无法读取消息 。

轮询不只是获取数据那么简单。在第一次调用新消费者的 poll()方法时，它会负责查找GroupCoordinator ， 然后加入群组，接受分配的分区。 如果发生了再均衡，整个过程也是在轮询期间进行的。当然，心跳也是从轮询里发迭出去的。所以，我们要确保在轮询期间所做的任何处理工作都应该尽快完成。

## 5.消费者的配置

大部分参数都有合理的默认值，一般不需要修改它们，不过有一些参数与消费者的性能和可用性有很大关系。接下来介绍这些重要的属性。

**1.fetch.min.bytes**

该属性指定了消费者从服务器获取记录的最小字节数。broker 在收到消费者的数据请求时，如果可用的数据量小于fetch . min. bytes指定的大小，那么它会等到有足够的可用数据时才把它返回给消费者。

**2.fetch.max.wait.ms**

用于指定broker的等待时间，默认是500ms。如果 fetch.max.wait.ms 被设为 100ms，并且fetch. min.bytes被设为lMB，那么Kafka 在收到消费者的请求后，要么返回IMB 数据，要么在100ms后返回所有可用的数据，就看哪个条件先得到满足。

**3.max.partition.fectch.bytes**

该属性指定了服务器从每个分区里返回给消费者的最大字节数。它的默认值是 lMB，也就是说，KafkaConsumer.poll()方法从每个分区里返回的记录最多不超过max.partition.fectch.bytes 指定的字节. max.partition.fectch.bytes的值必须比 broker 能够接收的最大消息的字节数（通过max.message.size属性配置）大， 否则消费者可能无法读取这些消息，导致消费者一直挂起重试。在设置该属性时，另一个需要考虑的因素是消费者处理数据的时间。消费者需要频繁调用 poll() 方法来避免会话过期和发生分区再均衡，如果单次调用 poll() 返回的数据太多，消费者需要更多的时间来处理，可能无法及时进行下一个轮询来避免会话过期。如果出现这种情况， 可以把max.partition.fectch.bytes 值改小，或者延长会话过期时间。

**4.session.timeout.ms**

该属性指定了消费者在被认为死亡之前可以与服务器断开连接的时间，默认是 3s。该属性与heartbeat.interval.ms 紧密相关。heartbeat.interval.ms指定了 poll()方法向协调器发送心跳的频率，session.timeout.ms 则指定了消费者可以多久不发送心跳。所以， 一般需要同时修改这两个属性，heartbeat.interval.ms 必须比session.timeout.ms 小，一般是 session.timeout.ms的三分之一。

**5.auto.offset.reset**

该属性指定了消费者在读取一个没有偏移量的分区或者偏移量无效的情况下（因消费者长时间失效，包含偏移量的记录已经过时井被删除）该作何处理。

**6.enable.auto.commit**

该属性指定了消费者是否自动提交偏移量，默认值是 true。为了尽量避免出现重复数据和数据丢失，可以把它设为false，由自己控制何时提交偏移量。如果把它设为true，还可以通过配置auto. commit. interval.ms属性来控制提交的频率。

**7.client.id**

属性可以是任意字符串，broker 用它来标识从客户端发送过来的消息，通常被用在日志、度量指标和配额里。

**8.max.poll.records**

该属性用于控制单次调用call()方法能够返回的记录数量，可以帮你控制在轮询里需要处理的数据量。

**9.receive.buffer.bytes和send.buffer.bytes**

设置socket 在读写数据时用到的TCP缓冲区大小。

## 6.提交和偏移量

每次调用 poll()方法，它总是返回由生产者写入 Kafka 但还没有被消费者读取过的记录，我们因此可以追踪到哪些记录是被群组里的哪个消费者读取的。Kafka不会像其他 JMS 队列那样需要得到消费者的确认，这是 Kafka 的一个独特之处。相反，消费者可以使用 Kafka 来追踪消息在分区里的位置（偏移量）。把更新分区当前位置的操作叫作提交。

### 1）自动提交

最简单的提交方式是让悄费者自动提交偏移量。如果enable.auto.commit被设为true ，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，默认值是 5s。与消费者里的其他东西一样，自动提交也是在轮询里进行的。消费者每次在进行轮询时会检查是否该提交偏移量了，如果是，那么就会提交从上一次轮询返回的偏移量。

### 2）提交当前偏移量

把 auto.commit.offset 设为 false，让应用程序决定何时提交偏移量。使用committSync()提交偏移量最简单也最可靠。这个 API 会提交由 poll()方法返回的最新偏移量，提交成功后马上返回，如果提交失败就抛出异常。

要记住，commitSync()将会提交由poll()返回的最新偏移量，所以在处理完所有记录后要确保调用了commitSync()，否则还是会有丢失消息的风险。如果发生了再均衡，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。

下面是我们在处理完最近一批消息后使用commitSync()方法提交偏移量的例子。

while(true) {

ConsumerRecord<String, String> records=consumer.poll(100);

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

System.out.printf("topic=%s, partition=%s, offset=%d, customer=%s,country=%s\n", record.topic(),record.patition(),record.offerset(),record.key(),record.value());

}

try {

consumer.commitSync();

} catch (CommitFailedException e) {

log.error("commit failed", e)

}

}

处理完当前批次的消息，在轮询更多的消息之前，调用commitSync()方法提交当前批次最新的偏移量。

### 3）异步提交

手动提交有一个不足之处，在 broker 对提交请求作出回应之前，应用程序会一直阻塞，这样会限制应用程序的吞吐量。我们可以通过降低提交频率来提升吞吐量，但如果发生了均衡，会增加重复消息的数量。这个时候可以使用异步提交 API。我们只管发送提交请求，无需等待 broker 的响应。

consumer.commitAsync();

在成功提交或碰到无怯恢复的错误之前，commitSync()会一直重试，但是commitAsync ()不会，这也是 commitAsync()不好的一个地方

commitAsync()也支持回调，在broker作出响应时会执行回调。回调经常被用于记录提交错误或生成度量指标。

consumer.commitAsync(new OffsetCommitCallback() {

public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception e) {

if (e != null) log.error(“commit failed for offset {}”, offsets, e);

}

});

发送提交请求然后继续做其他事情，如果提交失败，错误信息和偏移量会被记录下来。

### 4)同步和异步组合提交

一般情况下，针对偶尔出现的提交失败，不进行重试不会有太大问题，因为如果提交失败是因为临时问题导致的，那么后续的提交总会有成功的。但如果这是发生在关闭消费者或再均衡前的最后一次提交，就要确保能够提交成功。因此，在消费者关闭前一般会组合使用 commitAsync()和commitSync()。

try {

while(true) {

ConsumerRecord<String, String> records=consumer.poll(100);

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

System.out.printf("topic=%s, partition=%s, offset=%d, customer=%s,country=%s\n", record.topic(),record.patition(),record.offerset(),record.key(),record.value());

}

consumer.commitAsync();

}

} catch (Exception e) {

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

} finally {

try {

consumer.commitSync();

} finally {

consumer.close();

}

}

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。

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。

## 7.再均衡监听器

在为消费者分配新分区或移除旧分区时，可以通过消费者API执行一些应用程序代码，在调用subscribe()方法时传进去一个ConsumerRebalanceListener实例就可以。CounsumerResbalanceListener有两个需要实现的方法。

(1)public void onPartitionsRevoked(Collection<TopicPartition> partitions)方法会在再均衡开始之前和消费者停止读取消息之后被调用。如果在这里提交偏移量，下一个接管分区的消费者就知道该从哪里开始读取了。

(2)public void onPartitionAssigned(Collection<TopicPartition> partitions)方法会在重新分配分区之后和消费者开始读取之前被调用。

下面的例子将演示如何在失去分区所有权之前通过onPartitionsRevoked()方法来提交偏移量。

private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();

private class HandleRebalance implements ConsumerRebalanceListener {

public void onPartitionsAssigned(Collection<TopicPartition> partitions){…………}

public void onPartitionsRevoked(Collection<TopicPartition> partitions) {

System.out.println("lost partitions in rebalance. Committiong current offset:"+currentOffsets);

consumer.commitSync(currentOffsets);

}

}

try {

consumer.subscribe(topic, new HandleRebalance());

while(true) {

ConsumerRecord<String, String> records=consumer.poll(100);

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

System.out.printf("topic=%s, partition=%s, offset=%d, customer=%s,country=%s\n", record.topic(),record.patition(),record.offerset(),record.key(),record.value());

currentOffsets.put( new TopicPartition(record.topic(),record.partition()), new OffsetAndMetadata(record.offerset()+1,"no metadata"));

}

consumer.commitAsync(currentOffsets,null);

}

} catch(WakeupException e) {

//忽略异常，正在关闭消费者

} catch (Exception e) {

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

} finally {

try {

consumer.commitSync(currentOffsets);

} finally {

consumer.close();

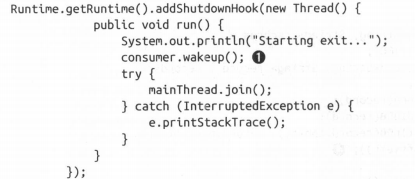
System.out.println("closed consumer and we are done")

}

}

## 8.如何退出

如果确定要退出循环，需要通过另一个线程调用consumer.wakeup()方法。如果循环运行在主线程里，可以在ShutdownHook 里调用该方法。要记住，consumer.wakeup()是消费者唯一一个可以从其他线程里安全调用的方法。调用 consumer.wakeup()可以退出 poll() ,并抛出 WakeupException 异常，或者如果调用consumer.wakeup()时线程没有等待轮询， 那么异常将在下一轮调用 poll()时抛出。我们不需要处理 WakeupException，因为它只是用于跳出循环的一种方式。不过，在退出线程之前调用 consumer.close()是很有必要的，它会提交任何还没有提交的东西，并向群组协调器发送消息，告知自己要离开群组，接下来就会触发再均衡，而不需要等待会话超时。



## 9.反序列化器

生产者需要用序列化器把对象转换成字节数组再发送给 Kafka。类似地，消费者需要用反序列化器把从Kafka接收到的字节数组转换成Java对象。

# 四．Kafka数据高可靠性深度解读

参考博文：

http://www.infoq.com/cn/articles/depth-interpretation-of-kafka-data-reliability?utm\_source=articles\_about\_Kafka&utm\_medium=link&utm\_campaign=Kafka