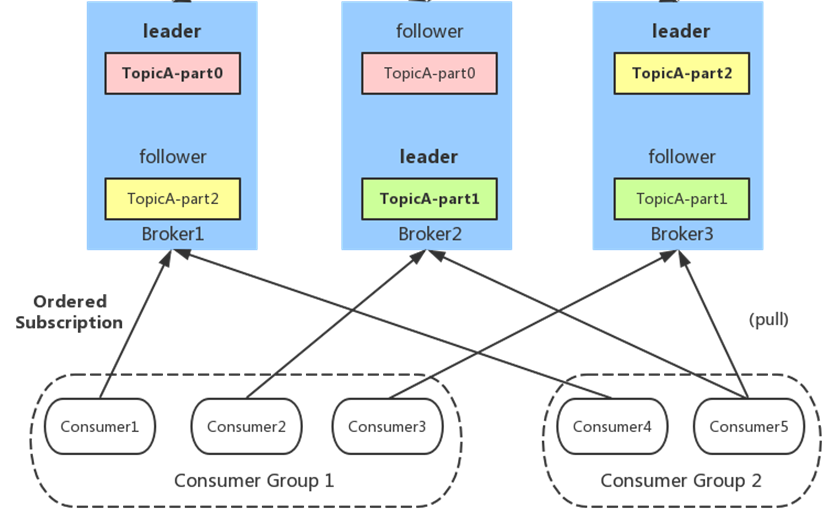
Kafka Consumer

Kakfa Consumer端向broker发送fetch请求，并告知其获得消息的offset，此后消费者会获得一定条数的消息，Consumer端也可以重置offset来重置消费消息，如下图：



在Kafka中，采用pull方式即Consumer在和broker建立连接之后，主动去pull消息，这种模式下Consumer端可以根据自己的消费能力适时的fetch消息并处理，且可以控制消息消费的进度（offset），此外消费者可以良好的控制消息消费的数量。

Consumer Group是Kafka提供的可扩展且具有容错性的消费者机制，在组内可以有多个消费者或消费者实例，共享一个ID(group ID)。组内所有的消费者协调在一起消费订阅主题的所有分区。当然每个分区只能由同一个消费组内的一个Consumer来消费。

消费者从broker中获取消息的流程图如下所示：

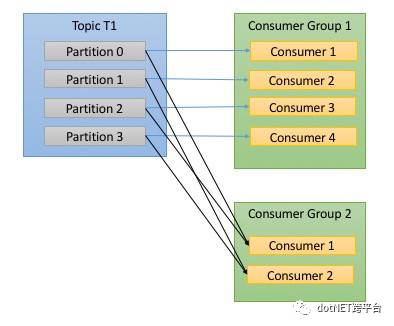


涉及到两个主要线程：

* 主线程，负责将从Broker中获取到数据经过反序列化后封装成ConsumerRecord，然后经过拦截器过滤后将其信息返回KafkaConsumer。
* Fetcher线程，负责将fetch请求发送给Broker，取出数据后写入到CompletedFetches

# 消费者群组

Consumer Group是Kafka提供的可扩展且具有容错性的消费者机制，组内可以有多个消费者或者消费者实例，共享一个公共的ID(group ID)，组内所有消费者起义消费订阅主题的所有分区，每个分区只能由同一个消费组内的一个Consumer来消费。



Consumer Group基本特性：

* 消费者群组下可以有一个或者多个Consumer Instance，可以是进程或者线程
* group.id是一个字符串，唯一标识consumer group
* consumer group下订阅的topic下每个分区只能分配给某个group下的一个Consumer(分区还可以分配给其他group)

向群组中增加消费者是横向伸缩消费能力的主要方式，Kafka消费者经常会作出一些高延迟的操作，例如把数据写入HDFS，或者使用数据进行比较耗时的计算，这些情况下可以增加消费者，让其分担负载，每个消费者只处理部分分区的消息。

# 创建消费者

在读取消息之前，先创建一个KafkaConsumer对象，示例如下：

*Properties props = new Properties();*

*props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG,*

*KafkaProperties.KAFKA\_SERVER\_URL + ":" + KafkaProperties.KAFKA\_SERVER\_PORT);*

*props.put(ConsumerConfig.GROUP\_ID\_CONFIG, "DemoConsumer");*

*props.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "true");*

*props.put(ConsumerConfig.AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG, "1000");*

*props.put(ConsumerConfig.SESSION\_TIMEOUT\_MS\_CONFIG, "30000");*

*props.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class.getName());*

*props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, CompanyDeserializer.class.getName());*

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

把消费者的属性放在Properties对象中，必要的属性：

* bootstrap.servers，该属性指定broker的地址清单，地址格式host:port，清单中不需要包含所有的broker地址，生产者会从给定的broker中查找到其他的broker信息，建议提供至少两个broker信息，防止配置的broker宕机
* key.deserializer，配置指定的类把字节数组转换成Java对象
* value.deserializer，与key.deserializer一样，将value值反序列化

其他重要的属性：

* fetch.min.bytes，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，那么会等到有租后的可用数据时才返回给消费者，降低消费者和broker的工作负载
* fetch.max.wait.ms，用于指定broker的等待时间，默认500ms
* max.partition.fetch.bytes，该属性指定从每个分区中返回给消费者最大字节数，默认值1MB，从每个分区中返回的最大记录最多不超过其指定的字节。
* session.timeout.ms，该属性指定消费者在被认为死亡之前可以与服务器断开链接的时间，默认3s，如果消费者没有在session.timeout.ms指定的时间内发送心跳给群组协调者，就会被认为已经死亡，协调器会触发再均衡，把它的分区发送给群组中其他消费者。该属于与heartbeat.interval.ms紧密相关，指定poll方法向协调者发送心跳的频率。
* auto.offset.reset，指定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并删除）该如何处理。默认值为latest，在偏移量无效的情况下，消费者将从最新的记录开始读取数据；earliest，在偏移量无效的情况下，消费者将从起始位置读取分区的记录
* enable.auto.commit，该属性指定消费者是否自动提交偏移量，默认值true。为了尽量避免出现重复数据和数据丢失，可以设置为flase，由自己控制何时提交偏移量
* partition.assignment.stragey，分区会被分配给群组中的消费者，PartitionAssignor根据给定消费者和主题，决定哪些分区应该被分配给哪个消费者。默认分配策略有两个：

1. Range，把主题的若干个连续分区分配给消费者
2. RoundBobin，把主题所有分区逐个分配给消费者

* client.id，任意字符串，标识从客户端发送过来的消息
* max.poll.records，控制单次调用call方法能够返回的记录数量
* receive.buffer.bytes和send.buffer.bytes，socket在读写数据时用到的TCP缓冲区，如果生产者和消费者处于不同的数据中心上，可以适当增大这些值

创建好消费者之后，就可以开始订阅主题了，subscribe方法接收一个主题类表作为参数，示例如下：

*consumer.subscribe(Collections.singletonList(this.topic));*

可以在调用subscribe方法时传入一个正则表达式，其匹配多个主题。

# 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所示：

*while (isRunning)*

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

*}*

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

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，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送出去的。

Poll方法返回一个记录列表CustomerRecords<K,V>，包含所属主题的信息、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：

*private final String topic;*

*private final int partition;*

*private final long offset;*

*private final int serializedKeySize;*

*private final int serializedValueSize;*

*private final K key;*

*private final V value;*

# 提交和偏移量

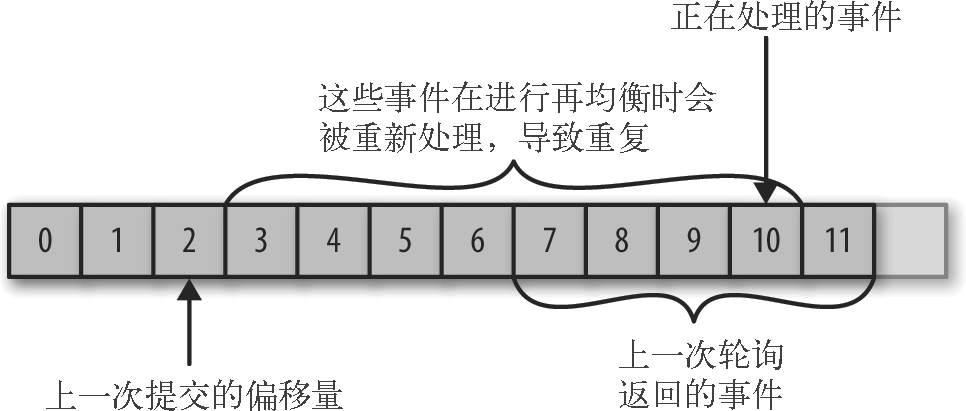
每次调用poll方法，总是返回由生产者写入Kafka但还没有被消费者读取过的记录，因此需要追踪哪些记录是被群组里的哪个消费者读取的，即消费者可以使用Kafka来追踪消息在分区里的位置（偏移量）。

更新分区当前位置的操作叫做提交，消费者往一个叫做\_\_consumer\_offset的特殊标题发送消息，消息里包含每个分区的偏移量，如下所示：

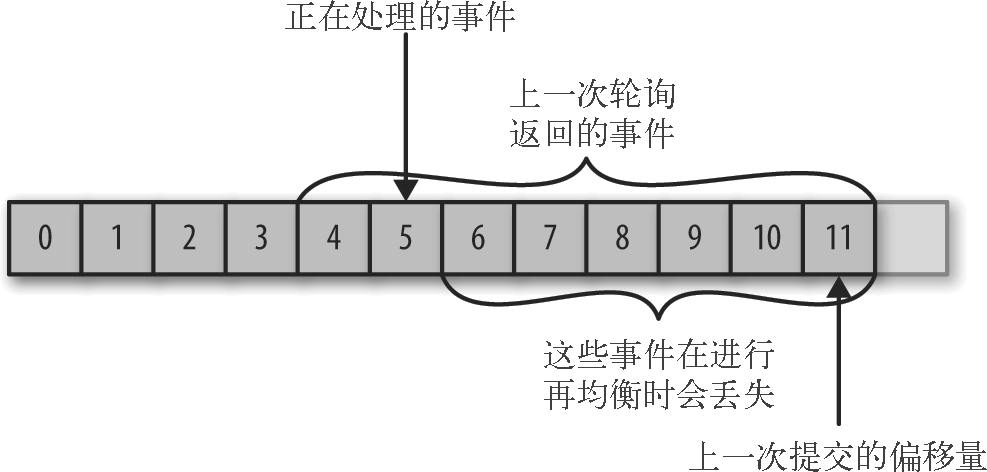
*/brokers/topics/\_\_consumer\_offsets*

如果消费者一直处于运行状态，那么偏移量就没有什么用处，当消费者发生崩溃或者新的消费者加入群组，就会触发再均衡，完成再均衡之后，每个消费者可能分配到新的分区，而不是之前处理的那个，消费者读取每个分区最后一次提交的偏移量，然后从偏移量指定的地方继续处理。

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，如下图所示：



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，如下图所示：



KafkaConsumer API提供很多方式提交偏移量，默认为enable.auto.commit为true

1. 自动提交，如果设置enable.auto.commit为true，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5s，自动提交也是在轮询中进行，消费者每次在进行轮询时会检查是否该提交偏移量了，那么是那么就会提交上次轮询返回的偏移量。
2. 同步提交，提交当前偏移量，如果设置enable.auto.commit为false时，让应用程序决定何时提交最新的偏移量，代码示例：

*while (true) {*

*ConsumerRecords<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.partition(),*

*record.offset(), record.key(), record.value());*

*}*

*try {*

*consumer.commitSync();*

*} catch (CommitFailedException e) {*

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

*}*

*}*

使用commitSync提交poll返回的最新偏移量，在处理完所有记录后要确保调用commitSync，否则还是会有丢失消息的风险。

1. 异步提交，同步提交不足之处在于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，可以使用异步提交API，只管发送提交请求，无需等待broker的响应，调用接口为consumr.commitAsync，其也支持回调，回调经常被用于记录提交错误或生产度量指标

*while (true) {*

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

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

*.....*

*}*

*consumer.commitAsync(new OffsetCommitCallback() {*

*public void onComplete(Map<TopicPartition,*

*OffsetAndMetadata> offsets, Exception e) {*

*if (e != null)*

*log.error("Commit failed for offsets {}", offsets, e);*

*}*

*});*

*}*

1. 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，因为失败是临时问题导致的，后续的提交总会有成功的。但是如果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，因此考虑使用混合提交的方法：

*try {*

*while (true) {*

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

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

*System.out.println("topic = %s, partition = %s, offset = %d,*

*customer = %s, country = %s\n",*

*record.topic(), record.partition(),*

*record.offset(), record.key(), record.value());*

*}*

*consumer.commitAsync();*

*}*

*} catch (Exception e) {*

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

*} finally {*

*try {*

*consumer.commitSync();*

*} finally {*

*consumer.close();*

*}*

*}*

在程序正常运行过程中，使用commitAsync方法进行提交，如果直接关闭消费者就没有所谓的下一次提交，不会再调用poll方法，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。

1. 提交特定的偏移量

# 从特定偏移量出开始处理记录

有时候需要从特定偏移量处开始读取消息，例如像从分区起始位置开始读取消息，或者直接跳到分区的末尾开始读取消息，可以使用seekToBeginning(Collection<TopicPartition> tp)和seekToEnd(Collection<TopicPartition> tp)这两个方法。

Kafka也提供了用于查找特定偏移量的API，比如向后回退几个消息或者向前跳过几个消息，当然也可以使用Kafka之外的系统来存储偏移量。