1.概念

1.1 消费者与消费群组

kafka消费者从属于消费者群组。一个群组订阅一个主题，每个消费者对应该主题一部分分区

a)往群组增加消费者是横向伸缩消费能力的主要方式

b)存在多个应用程序从同一个主题读取数据

1.2 消费者群组和分区再平衡

再均衡：分区的使用权从一个消费者转移到另外一个消费者，再均衡期间消费者无法读取消息

群组协调器：消费者通过向被指派为群组协调器的broker发送心跳来维持它们和群组的从属关系以及对分区的所有权关系

分配分区过程：消费者加入群组，向群组协调器发送一个JoinGroup请求，第一个加入群组的消费者就是群主。群主从协调器获得群组的成员列表，并且给每一个消费者分配分区。它使用一个实现了PartitionAssignor接口的类来决定分区分配（有2种分配策略，后面谈）。群主把分配情况列表给协调器，协调器在把这些信息发送给所有消费者。每个消费者只能看到自己的分配信息，只有群主知道所有消费者的分配信息。每次再均衡都会发生一次这个过程。

2kafka消费者编程

a)创建KafkaConsumer，设置

bootstrap.server,key.deserializer,value.deserializer,group.id（指定该消费者属于哪个群组）

b)订阅主题

c)轮询

代码细节请查看github/kafka-study

3.消费者配置

fetch.min.bytes:消费者获取记录的最小字节

fetech.max.wait.ms：最长等待时间。和fetch.min.bytes二者满足一个就行

max.partition.fetch.bytes：从每个分区返回给消费者的最大字节数

session.timeout.ms:comsuer被认为dead之前可以和服务器断开连接的时间。

auto.offset.reset 消费者在读取一个没偏移量/偏移量无效的分区的情况

下该如何处理。默认是latest，还有一个是earliest
enable.auto.commit:消费者是否自动提交偏移量
partition.assignment.strategy：分区分配策略，包括
a)Range
该策略把连续的分区分配给消费者，
partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor
b)RoundRobin
该策略把所有分区逐个分配给消费
者,partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor
c)自定义
这时候，partition.assignment.strategy属性就是自定义类的名字
max.poll.records：单次调用call()方法能返回的记录数量
receive.buffer.bytes和send.buffer.bytes socket在读写数据时用到的TCP缓存区大小

4.提交和偏移量
提交：更新分区当前位置的操作叫作***提交***。
偏移量：消费者往一个 叫作 _consumer_offset 的特殊主题发送 消息，消息里包含每个分区的偏移量，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。不过，如果悄费者发生崩溃或者有新 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。为了能够继续 之前的工作，消费者需要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 存储在_consumer_offset 中的并且被消费者重新读取的偏移量 小于kafka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，如图 4-6所示。

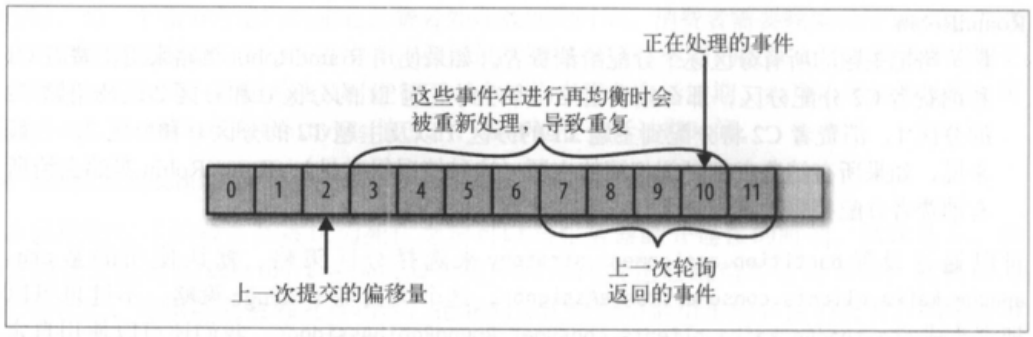

图 4-6：提交的偏移量小于客户端处理的最后一个消息的偏移量

如果 存储在_consumer_offset 中的并且被消费者重新读取的偏移量 小于kafka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，如图 4-7 所示。

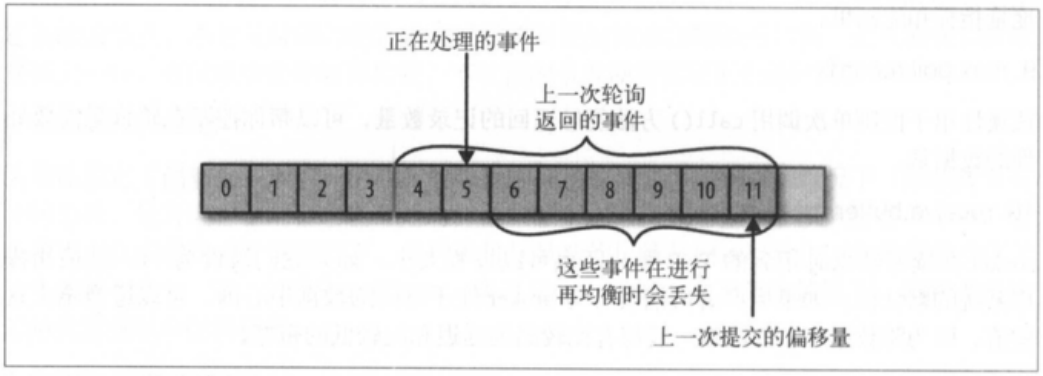

图 4-7：提交的偏移量大于客户端处理的最后一个消息的偏移量

偏移量提交方式：

a) 自动方式

`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

问题在于：假设我们仍然使用默认的 5s提交时间间隔，在最近一次提交之后的 3s发生了再均衡，再 均衡之后，消费者从最后一次提交的偏移量位置开始读取消息。这个时候偏移量已经落后 了 3s，所以在这 3s 内到达的消息会被重复处理。

b)提交当前偏移量

auto.commit.offset 设为 false，使用 commitSync() 提交偏移量最简单也最可靠

```
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()); ❶
    }
    try {
      consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
       log.error("commit failed", e) ❸
    }
}
```

❶ 我们假设把记录内容打印出来就算处理完毕，这个是由应用程序根据具体的使用场景来决定的。

❷ 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。

❸ 只要没有发生不可恢复的错误，commitSync() 方法会一直尝试直至提交成功。如果提交失败，我们也只能把异常记录到错误日志里。

c)异步提交 commitAsync

同步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。

```
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());
    }
    consumer.commitAsync(); ❶
}
```

❶ 提交最后一个偏移量，然后继续做其他事情。

成功提交或碰到无怯恢复的错误之前， commitSync() 会一直重试(应用程序也一直阻塞)，但是 commitAsync() 不会，这也是 commitAsync() 不好的 一个地方。它之所以不进行重试，是因为在它收到 服务器响应的时候，可能有一个更大的偏移量已经提交成功。假设我们发出一个请求用于提交偏移量 2000，这个时候发生了短暂的通信问题 ，服务器收不到请求，自然也不会 作出任何响应。与此同时，我们处理了另外一批消息，并成功提交了偏移量 3000。如果 commitAsync() 重新尝试提交偏移量 2000，它有可能在偏移量 3000之后提交成功。这个时 候如果发生再均衡，就会出现重复消息。

我们之所以提到这个问题的复杂性和提交顺序的重要性，是因为 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， 不 过如果你要用它来进行重试， 一定要注意提交的顺序。

```
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());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
      }); ❶
}
```

❶ 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。

重试异步提交

我们可以使用一个单调递增的序列号来维护异步提交的顺序。在每次提交偏 移量之后或在回调里提交偏移量时递增序列号。在进行重试前，先检查回调 的序列号和即将提交的偏移量是否相等，如果相等，说明没有新的提交，那么可以安全地进行重试。如果序列号比较大，说明有一个新的提交已经发送出去了，应该停止重试。

d)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()。它们的工作原理如下(后面讲到再均衡监听器时，我们会讨论如何在发生再均衡前提交偏移量):

```
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(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

❶ 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。

❷ 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。

e)提交特定的偏移量

提交偏移量的频率与处理消息批次的频率是一样的。但如果想要更频繁地提交出怎么办？如果 poll() 方法返回一大批数据，为了避免因再均衡引起的重复处理整批消息，想要在批次中间提交偏移量该怎么办？这种情况无法通过调用 commitSync()或 commitAsync() 来实现，因为它们只会提交最后一个偏移量，而此时该批次里的消息还没有处理完。

幸运的是，消费者 API 允许在调用 commitSync()和 commitAsync()方法时传进去希望提交 的分区和偏移量的 map。假设你处理了半个批次的消息， 最后一个来自主题“customers”分区 3 的消息的偏移量是 5000， 你可以调用 commitSync() 方法来提交它。不过，因为消费者可能不只读取一个分区， 你需要跟踪所有分区的偏移量，所以在这个层面上控制偏移 量 的提交会让代码变复杂。

下面是提交特定偏移量的例子：

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

...

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()); ❷
        currentOffsets.put(new TopicPartition(record.topic(),
        record.partition()), new
        OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0) ❹
            consumer.commitAsync(currentOffsets,null); ❺
        count++;
    }
}
```

❶ 用于跟踪偏移量的 map。
❷ 记住，printf 只是处理消息的临时方案。
❸ 在读取每条记录之后，使用期望处理的下一个消息的偏移量更新 map 里的偏移量。下一次就从这里开始读取消息。
❹ 我们决定每处理 1000 条记录就提交一次偏移量。在实际应用中，你可以根据时间或记录的内容进行提交。
❺ 这里调用的是 commitAsync()，不过调用 commitSync() 也是完全可以的。当然，在提交特定偏移量时，仍然要处理可能发生的错误。

5.再均衡监听器（未完成）

6.从特定偏移量处开始处理记录（未完成）

7.如何退出（未完成）

consumer.wakeup()

8.反序列化器（未完成）

9.独立消费者

一般情况下，分区被自动分配给群组里的消费者，在群组里新增或移出消费者的时候进行再分配。

但是如果只需要一个消费者从一个主题的所有分区或者某个特定的分区读取数据，就不需要群组和再均衡了。，只需要把主题或者分区分配给消费者，如何开始读取消息并提交偏移量。

如果是这样。那么就不需要订阅主题而是为自己分配分区。

```
package com.bonc.rdpe.kafka110.consumer;

import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.TopicPartition;

/**
 * @author
 * @date 2018-07-17 12:44:50
 * @description: 独立消费者
 */
public class AloneConsumer {

    public static void main(String[] args) {

        Properties props = new Properties();
        props.put("bootstrap.servers",
"rdpecore4:9092,rdpecore5:9092,rdpecore6:9092");
        // 独立消费者不需要设置消费组
        // props.put("group.id", "dev3-yangyunhe-topic001-
group003");
        props.put("auto.offset.reset", "earliest");
        props.put("auto.commit.offset", false);
        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<>(props);

        /*
```

```
         * consumer.partitionsFor(topic)用于获取topic的分区信息
         * 当有新的分区加入或者原有的分区被改变后，这个方法是不能动态感知的
         * 所以要么周期性的执行这个方法，要么当分区数改变的时候，你需要重新执行这个程序
         */
        List<PartitionInfo> partitionInfos = consumer.partitionsFor("dev3-yangyunhe-topic001");
        List<TopicPartition> partitions = new ArrayList<>();

        if(partitionInfos != null && partitionInfos.size() != 0) {
            for (PartitionInfo partition : partitionInfos) {
                partitions.add(new TopicPartition(partition.topic(), partition.partition()));
            }

            consumer.assign(partitions);

            try {
                while (true) {
                    ConsumerRecords<String, String> records = consumer.poll(1000);
                    for (ConsumerRecord<String, String> record : records) {
                        System.out.println("partition = " + record.partition() + ", offset = " + record.offset());
                    }
                    consumer.commitAsync();
                }
            } finally {
                consumer.commitSync();
                consumer.close();
            }
        }
    }
}
```