再均衡

再均衡是指分区的所属权从一个消费者转移到另一消费者的行为，它为消费组具备高可用性和伸缩性提供保障，使我们可以既方便又安全地删除消费组内的消费者或往消费组内添加消费者。不过在再均衡发生期间，消费组内的消费者是无法读取消息的。也就是说，在再均衡发生期间的这一小段时间内，消费组会变得不可用。

另外，当一个分区被重新分配给另一个消费者时，消费者当前的状态也会丢失。比如消费者消费完某个分区中的一部分消息时还没有来得及提交消费位移就发生了再均衡操作，之后这个分区又被分配给了消费组内的另一个消费者，原来被消费完的那部分消息又被重新消费一遍，也就是发生了重复消费。一般情况下，应尽量避免不必要的再均衡的发生。

第8节中在讲述 subscribe() 方法时提及再均衡监听器 ConsumerRebalanceListener，在 subscribe(Collection<String> topics, ConsumerRebalanceListener listener)和 subscribe(Pattern pattern, ConsumerRebalanceListener listener)方法中都有它的身影。再均衡监听器用来设定发生再均衡动作前后的一些准备或收尾的动作。ConsumerRebalanceListener 是一个接口，包含2个方法，具体的释义如下：

1. void onPartitionsRevoked(Collection partitions)
   这个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。可以通过这个回调方法来处理消费位移的提交，以此来避免一些不必要的重复消费现象的发生。参数 partitions 表示再均衡前所分配到的分区。
2. void onPartitionsAssigned(Collection partitions)
   这个方法会在重新分配分区之后和消费者开始读取消费之前被

调用。参数 partitions 表示再均衡后所分配到的分区。

下面我们通过一个例子来演示 ConsumerRebalanceListener 的用法，具体内容如代码清单13-1所示。

```
//代码清单13-1 再均衡监听器的用法
Map<TopicPartition, OffsetAndMetadata>
currentOffsets = new HashMap<>();
consumer.subscribe(Arrays.asList(topic), new
ConsumerRebalanceListener() {
    @Override
    public void
onPartitionsRevoked(Collection<TopicPartition>
partitions) {
        consumer.commitSync(currentOffsets);
            currentOffsets.clear();
    }
    @Override
    public void
onPartitionsAssigned(Collection<TopicPartition>
partitions) {
        //do nothing.
    }
});

try {
    while (isRunning.get()) {
        ConsumerRecords<String, String> records =

consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String>
record : records) {
            //process the record.
```

```
                currentOffsets.put(
                        new
TopicPartition(record.topic(),
record.partition()),
                        new
OffsetAndMetadata(record.offset() + 1));
            }
            consumer.commitAsync(currentOffsets,
null);
        }
    } finally {
        consumer.close();
    }
```

代码清单13-1中将消费位移暂存到一个局部变量 currentOffsets 中，这样在正常消费的时候可以通过 commitAsync() 方法来异步提交消费位移，在发生再均衡动作之前可以通过再均衡监听器的 onPartitionsRevoked() 回调执行 commitSync() 方法同步提交消费位移，以尽量避免一些不必要的重复消费。

再均衡监听器还可以配合外部存储使用。在代码清单12-4中，我们将消费位移保存在数据库中，这里可以通过再均衡监听器查找分配到的分区的消费位移，并且配合 seek() 方法来进一步优化代码逻辑，将代码清单12-4中的第一行代码修改为如下内容：

```
consumer.subscribe(Arrays.asList(topic), new
ConsumerRebalanceListener() {
    @Override
    public void
onPartitionsRevoked(Collection<TopicPartition>
partitions) {
        //store offset in DB  (storeOffsetToDB)
    }
    @Override
    public void
onPartitionsAssigned(Collection<TopicPartition>
partitions) {
        for(TopicPartition tp: partitions){
            consumer.seek(tp,
getOffsetFromDB(tp));//从DB中读取消费位移
        }
    }
});
```

本节只是简单演示了再均衡监听器的用法，再均衡期间消费者客户端与 Kafka 服务端之间的交互逻辑及相关原理并不简单，更多的细节可以参考[《图解Kafka之核心原理》(https://juejin.im/book/5c7d270ff265da2d89634e9e)](https://juejin.im/book/5c7d270ff265da2d89634e9e)中的相关的内容。

消费者拦截器

第4节中讲述了生产者拦截器的使用，对应的消费者也有相应的拦截器的概念。消费者拦截器主要在消费到消息或在提交消费位移时进行一些定制化的操作。

与生产者拦截器对应的，消费者拦截器需要自定义实现 org.apache.kafka.clients.consumer. ConsumerInterceptor 接口。ConsumerInterceptor 接口包含3个方法：

- public ConsumerRecords<K, V> onConsume(ConsumerRecords<K, V> records)；
- public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets)；
- public void close()。

KafkaConsumer 会在 poll() 方法返回之前调用拦截器的 onConsume() 方法来对消息进行相应的定制化操作，比如修改返回的消息内容、按照某种规则过滤消息（可能会减少 poll() 方法返回的消息的个数）。如果 onConsume() 方法中抛出异常，那么会被捕获并记录到日志中，但是异常不会再向上传递。

KafkaConsumer 会在提交完消费位移之后调用拦截器的 onCommit() 方法，可以使用这个方法来记录跟踪所提交的位移信息，比如当消费者使用 commitSync 的无参方法时，我们不知道提交的消费位移的具体细节，而使用拦截器的 onCommit() 方法却可以做到这一点。

close() 方法和 ConsumerInterceptor 的父接口中的 configure() 方法与生产者的 ProducerInterceptor 接口中的用途一样，这里就不赘述了。

在某些业务场景中会对消息设置一个有效期的属性，如果某条消息在既定的时间窗口内无法到达，那么就会被视为无效，它也就不需要再被继续处理了。下面使用消费者拦截器来实现一个简单的消息 TTL（Time to Live，即过期时间）的功能。在代码清单13-1中，自定义的消费者拦截器 ConsumerInterceptorTTL 使用消息的 timestamp 字段来判定是否过期，如果消息的时间戳与当前的时间戳相差超过10秒则判定为过期，那么这条消息也就被过滤而不投递给具体的消费者。

```
//代码清单13-1 自定义的消费者拦截器
public class ConsumerInterceptorTTL implements
        ConsumerInterceptor<String, String> {
    private static final long EXPIRE_INTERVAL =
10 * 1000;

    @Override
    public ConsumerRecords<String, String>
onConsume(
            ConsumerRecords<String, String>
records) {
        long now = System.currentTimeMillis();
        Map<TopicPartition,
List<ConsumerRecord<String, String>>> newRecords
                = new HashMap<>();
        for (TopicPartition tp :
records.partitions()) {
            List<ConsumerRecord<String, String>>
tpRecords =
            records.records(tp);
            List<ConsumerRecord<String, String>>
newTpRecords = new ArrayList<>();
            for (ConsumerRecord<String, String>
record : tpRecords) {
                if (now - record.timestamp() <
EXPIRE_INTERVAL) {
                    newTpRecords.add(record);
                }
            }
            if (!newTpRecords.isEmpty()) {
                newRecords.put(tp, newTpRecords);
            }
        }
```

```
        return new ConsumerRecords<>(newRecords);
    }

    @Override
    public void onCommit(Map<TopicPartition,
OffsetAndMetadata> offsets) {
        offsets.forEach((tp, offset) ->
                System.out.println(tp + ":" +
offset.offset()));
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> configs)
{}
}
```

实现自定义的 ConsumerInterceptorTTL 之后，需要在 KafkaConsumer 中配置指定这个拦截器，这个指定的配置和 KafkaProducer 中的一样，也是通过 interceptor.classes 参数实现的，此参数的默认值为“”。示例如下：

```
props.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONF
IG,
        ConsumerInterceptorTTL.class.getName());
```

我们在发送消息的时候修改 ProducerRecord 中的 timestamp 的值来使其变得超时，具体可以参考下面的示例：

```
ProducerRecord<String, String> record1 = new
ProducerRecord<>(topic, 0, System
        .currentTimeMillis()-EXPIRE_INTERVAL,
null, "first-expire-data");
producer.send(record1).get();

ProducerRecord<String, String> record2 = new
ProducerRecord<>(topic, 0, System
        .currentTimeMillis(), null, "normal-
data");
producer.send(record2).get();

ProducerRecord<String, String> record3 = new
ProducerRecord<>(topic, 0, System
        .currentTimeMillis()-EXPIRE_INTERVAL,
null, "last-expire-data");
producer.send(record3).get();
```

示例代码中一共发送了三条消息：“first-expire-data”“normal-data”和“last-expire-data”，其中第一条和第三条消息都被修改成超时了，那么此时消费者通过 poll() 方法只能拉取到“normal-data”这一条消息，另外两条就被过滤了。

不过使用这种功能时需要注意的是：在使用类似代码清单11-2中这种带参数的位移提交的方式时，有可能提交了错误的位移信息。在一次消息拉取的批次中，可能含有最大偏移量的消息会被消费者拦截器过滤。

在消费者中也有拦截链的概念，和生产者的拦截链一样，也是按照 interceptor.classes 参数配置的拦截器的顺序来一一执行的（配置的时候，各个拦截器之间使用逗号隔开）。同样也要提防“副作用”的发生。如果在拦截链中某个拦截器执行失败，那么下一个拦截器会接着从上一个执行成功的拦截器继续执行。