消费者多线程实现

KafkaProducer 是线程安全的，然而 KafkaConsumer 却是非线程安全的。KafkaConsumer 中定义了一个 acquire() 方法，用来检测当前是否只有一个线程在操作，若有其他线程正在操作则会抛出 ConcurrentModifcationException 异常：

```
java.util.ConcurrentModificationException:
KafkaConsumer is not safe for multi-threaded
access.
```

KafkaConsumer 中的每个公用方法在执行所要执行的动作之前都会调用这个 acquire() 方法，只有 wakeup() 方法是个例外，具体用法可以参考第11节。acquire () 方法的具体定义如下：

```
private final AtomicLong currentThread
    = new AtomicLong(NO_CURRENT_THREAD);
//KafkaConsumer中的成员变量

private void acquire() {
    long threadId =
Thread.currentThread().getId();
    if (threadId != currentThread.get() &&

!currentThread.compareAndSet(NO_CURRENT_THREAD,
threadId))
        throw new ConcurrentModificationException
                ("KafkaConsumer is not safe for
multi-threaded access");
    refcount.incrementAndGet();
}
```

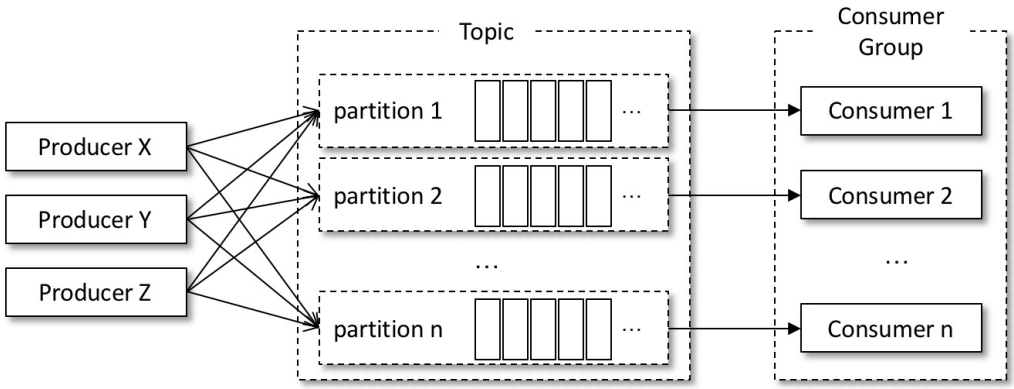
acquire() 方法和我们通常所说的锁（synchronized、Lock 等）不同，它不会造成阻塞等待，我们可以将其看作一个轻量级锁，它仅通过线程操作计数标记的方式来检测线程是否发生了并发操作，以此保证只有一个线程在操作。acquire() 方法和 release() 方法成对出现，表示相应的加锁和解锁操作。release() 方法也很简单，具体定义如下：

```
private void release() {
    if (refcount.decrementAndGet() == 0)
        currentThread.set(NO_CURRENT_THREAD);
}
```

acquire() 方法和 release() 方法都是私有方法，因此在实际应用中不需要我们显式地调用，但了解其内部的机理之后可以促使我们正确、有效地编写相应的程序逻辑。

KafkaConsumer 非线程安全并不意味着我们在消费消息的时候只能以单线程的方式执行。如果生产者发送消息的速度大于消费者处理消息的速度，那么就会有越来越多的消息得不到及时的消费，造成了一定的延迟。除此之外，由于 Kafka 中消息保留机制的作用，有些消息有可能在被消费之前就被清理了，从而造成消息的丢失。

我们可以通过多线程的方式来实现消息消费，多线程的目的就是为了提高整体的消费能力。多线程的实现方式有多种，第一种也是最常见的方式：线程封闭，即为每个线程实例化一个 KafkaConsumer 对象，如下图所示。



一个线程对应一个 KafkaConsumer 实例，我们可以称之为消费线程。一个消费线程可以消费一个或多个分区中的消息，所有的消费线程都隶属于同一个消费组。这种实现方式的并发度受限于分区的实际个数，根据第7节中介绍的消费者与分区数的关系，当消费线程的个数大于分区数时，就有部分消费线程一直处于空闲的状态。

与此对应的第二种方式是多个消费线程同时消费同一个分区，这个通过 assign()、seek() 等方法实现，这样可以打破原有的消费线程的个数不能超过分区数的限制，进一步提高了消费的能力。不过这种实现方式对于位移提交和顺序控制的处理就会变得非常复杂，实际应用中使用得极少，笔者也并不推荐。一般而言，分区是消费线程的最小划分单位。下面我们通过实际编码来演示第一种多线程消费实现的方式，详细示例参考如代码清单14-1所示。

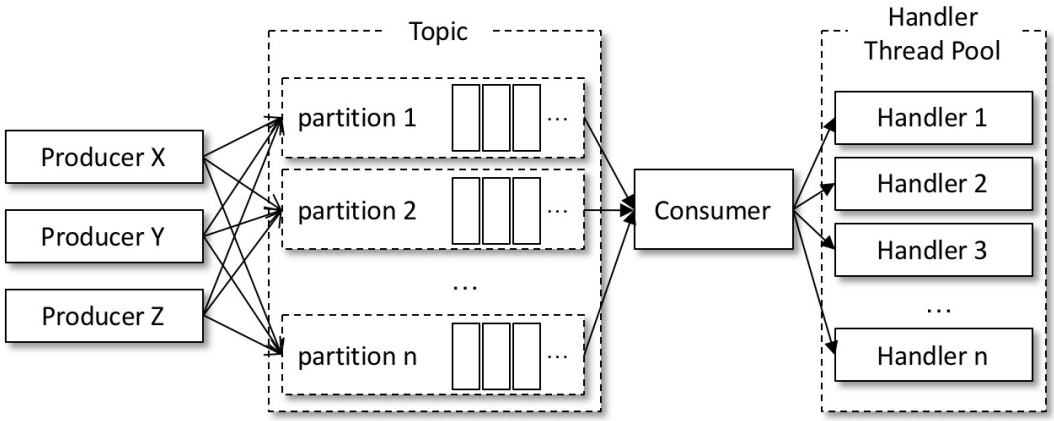
```
//代码清单14-1 第一种多线程消费实现方式
public class FirstMultiConsumerThreadDemo {
    public static final String brokerList =
"localhost:9092";
    public static final String topic = "topic-
demo";
    public static final String groupId =
"group.demo";

    public static Properties initConfig(){
        Properties props = new Properties();
```

```
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_C
ONFIG,

StringDeserializer.class.getName());

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS
_CONFIG,

StringDeserializer.class.getName());

props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG
, brokerList);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
groupId);

props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFI
G, true);
        return props;
    }

    public static void main(String[] args) {
        Properties props = initConfig();
        int consumerThreadNum = 4;
        for(int i=0;i<consumerThreadNum;i++) {
            new
KafkaConsumerThread(props,topic).start();
        }
    }

    public static class KafkaConsumerThread
extends Thread{
        private KafkaConsumer<String, String>
```

```
kafkaConsumer;

        public KafkaConsumerThread(Properties
props, String topic) {
            this.kafkaConsumer = new
KafkaConsumer<>(props);

this.kafkaConsumer.subscribe(Arrays.asList(topic)
);
        }

        @Override
        public void run(){
            try {
                while (true) {
                    ConsumerRecords<String,
String> records =

kafkaConsumer.poll(Duration.ofMillis(100));
                    for (ConsumerRecord<String,
String> record : records) {
                        //处理消息模块     ①
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                kafkaConsumer.close();
            }
        }
    }
}
```

内部类 KafkaConsumerThread 代表消费线程，其内部包裹着一个独立的 KafkaConsumer 实例。通过外部类的 main() 方法来启动多个消费线程，消费线程的数量由 consumerThreadNum 变量指定。一般一个主题的分区数事先可以知晓，可以将 consumerThreadNum 设置成不大于分区数的值，如果不知道主题的分区数，那么也可以通过 KafkaConsumer 类的 partitionsFor() 方法来间接获取，进而再设置合理的 consumerThreadNum 值。

上面这种多线程的实现方式和开启多个消费进程的方式没有本质上的区别，它的优点是每个线程可以按顺序消费各个分区中的消息。缺点也很明显，每个消费线程都要维护一个独立的TCP连接，如果分区数和 consumerThreadNum 的值都很大，那么会造成不小的系统开销。



参考代码清单14-1中的第①行，如果这里对消息的处理非常迅速，那么 poll() 拉取的频次也会更高，进而整体消费的性能也会提升；相反，如果在这里对消息的处理缓慢，比如进行一个事务性操作，或者等待一个RPC的同步响应，那么 poll() 拉取的频次也会随之下降，进而造成整体消费性能的下降。一般而言，poll() 拉取消息的速度是相当快的，而整体消费的瓶颈也正是在处理消息这一块，如果我们通过一定的方式来改进这一部分，那么我们就能带动整体消费性能的提升。

参考上图，考虑第三种实现方式，将处理消息模块改成多线程的实现方式，具体实现如代码清单14-2所示。

```
//代码清单14-2 第三种多线程消费实现方式
public class ThirdMultiConsumerThreadDemo {
    public static final String brokerList =
"localhost:9092";
    public static final String topic = "topic-
demo";
    public static final String groupId =
"group.demo";

    //省略initConfig()方法，具体请参考代码清单14-1
    public static void main(String[] args) {
        Properties props = initConfig();
        KafkaConsumerThread consumerThread =
                new KafkaConsumerThread(props,
topic,

Runtime.getRuntime().availableProcessors());
        consumerThread.start();
    }

    public static class KafkaConsumerThread
extends Thread {
        private KafkaConsumer<String, String>
kafkaConsumer;
        private ExecutorService executorService;
        private int threadNumber;

        public KafkaConsumerThread(Properties
props,
                String topic, int threadNumber) {
```

```
            kafkaConsumer = new KafkaConsumer<>
(props);

kafkaConsumer.subscribe(Collections.singletonList
(topic));
            this.threadNumber = threadNumber;
            executorService = new
ThreadPoolExecutor(threadNumber, threadNumber,
                    0L, TimeUnit.MILLISECONDS,
new ArrayBlockingQueue<>(1000),
                    new
ThreadPoolExecutor.CallerRunsPolicy());
        }

        @Override
        public void run() {
            try {
                while (true) {
                    ConsumerRecords<String,
String> records =

kafkaConsumer.poll(Duration.ofMillis(100));
                    if (!records.isEmpty()) {

executorService.submit(new
RecordsHandler(records));
                    }
①
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                kafkaConsumer.close();
```

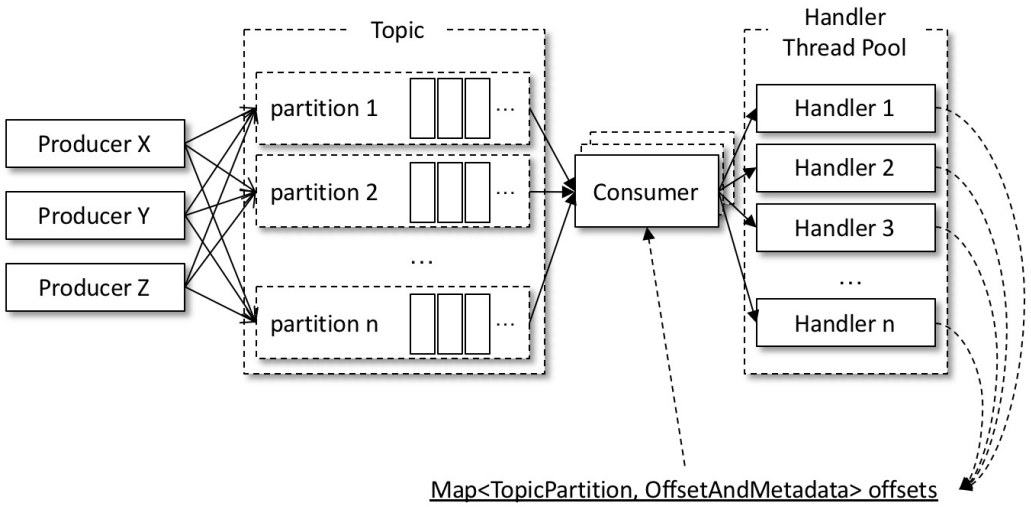
```
            }
        }

    }

    public static class RecordsHandler extends
Thread{
        public final ConsumerRecords<String,
String> records;

        public
RecordsHandler(ConsumerRecords<String, String>
records) {
            this.records = records;
        }

        @Override
        public void run(){
            //处理records.
        }
    }
}
```

代码清单14-2中的 RecordHandler 类是用来处理消息的，而 KafkaConsumerThread 类对应的是一个消费线程，里面通过线程池的方式来调用 RecordHandler 处理一批批的消息。注意 KafkaConsumerThread 类中 ThreadPoolExecutor 里的最后一个参数设置的是 CallerRunsPolicy()，这样可以防止线程池的总体消费能力跟不上 poll() 拉取的能力，从而导致异常现象的发生。第三种实现方式还可以横向扩展，通过开启多个 KafkaConsumerThread 实例来进一步提升整体的消费能力。

第三种实现方式相比第一种实现方式而言，除了横向扩展的能力，还可以减少TCP连接对系统资源的消耗，不过缺点就是对于消息的顺序处理就比较困难了。在代码清单14-1中的 initConfig() 方法里笔者特意加了一个配置：

```
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
```

这样旨在说明在具体实现的时候并没有考虑位移提交的情况。对于第一种实现方式而言，如果要做具体的位移提交，它的具体实现和第11节讲述的位移提交没有什么区别，直接在 KafkaConsumerThread 中的 run() 方法里实现即可。而对于第三种实现方式，这里引入一个共享变量 offsets 来参与提交，如下图所示。



每一个处理消息的 RecordHandler 类在处理完消息之后都将对应的消费位移保存到共享变量 offsets 中，KafkaConsumerThread 在每一次 poll() 方法之后都读取 offsets 中的内容并对其进行位移提交。注意在实现的过程中对 offsets 读写需要加锁处理，防止出现并发问题。并且在写入 offsets 的时候需要注意位移覆盖的问题，针对这个问题，可以将 RecordHandler 类中的 run() 方法实现改为如下内容（参考代码清单11-3）：

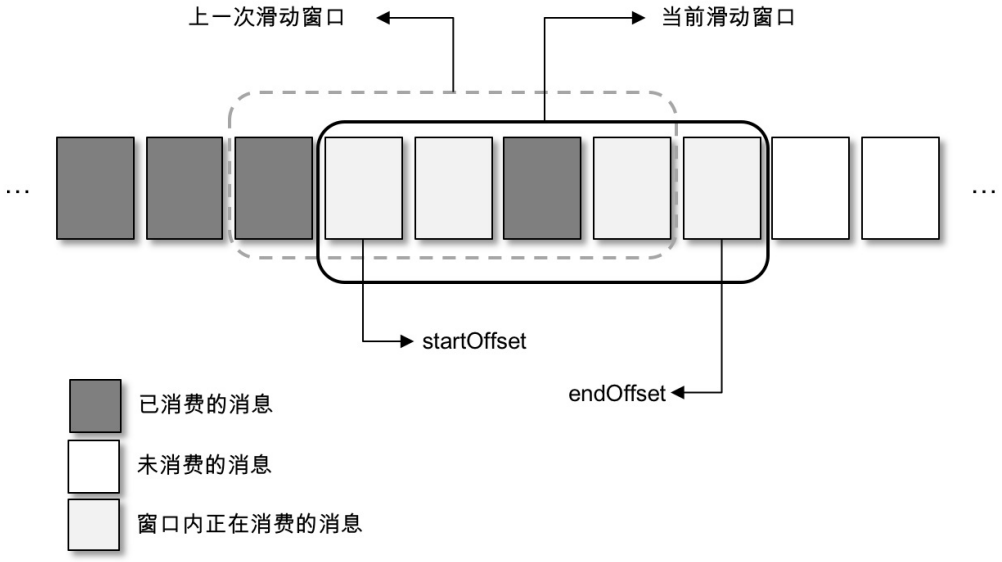
```
for (TopicPartition tp : records.partitions()) {
    List<ConsumerRecord<String, String>>
tpRecords = records.records(tp);
    //处理tpRecords.
    long lastConsumedOffset =
tpRecords.get(tpRecords.size() - 1).offset();
    synchronized (offsets) {
        if (!offsets.containsKey(tp)) {
            offsets.put(tp, new
OffsetAndMetadata(lastConsumedOffset + 1));
        }else {
            long position =
offsets.get(tp).offset();
            if (position < lastConsumedOffset +
1) {
                offsets.put(tp, new
OffsetAndMetadata(lastConsumedOffset + 1));
            }
        }
    }
}
```

对应的位移提交实现可以添加在代码清单14-2中KafkaConsumerThread 类的第①行代码下方，具体实现参考如下：

```
synchronized (offsets) {
    if (!offsets.isEmpty()) {
        kafkaConsumer.commitSync(offsets);
        offsets.clear();
    }
}
```

读者可以细想一下这样实现是否万无一失？其实这种位移提交的方式会有数据丢失的风险。对于同一个分区中的消息，假设一个处理线程 RecordHandler1 正在处理 offset 为0～99的消息，而另一个处理线程 RecordHandler2 已经处理完了 offset 为100～199的消息并进行了位移提交，此时如果 RecordHandler1 发生异常，则之后的消费只能从200开始而无法再次消费0～99的消息，从而造成了消息丢失的现象。这里虽然针对位移覆盖做了一定的处理，但还没有解决异常情况下的位移覆盖问题。

对此就要引入更加复杂的处理机制，这里再提供一种解决思路，参考下图，总体结构上是基于滑动窗口实现的。对于第三种实现方式而言，它所呈现的结构是通过消费者拉取分批次的消息，然后提交给多线程进行处理，而这里的滑动窗口式的实现方式是将拉取到的消息暂存起来，多个消费线程可以拉取暂存的消息，这个用于暂存消息的缓存大小即为滑动窗口的大小，总体上而言没有太多的变化，不同的是对于消费位移的把控。



如上图所示，每一个方格代表一个批次的消息，一个滑动窗口包含若干方格，startOffset 标注的是当前滑动窗口的起始位置，endOffset 标注的是末尾位置。每当 startOffset 指向的方格中的消息被消费完成，就可以提交这部分的位移，与此同时，窗口向前滑动

一格，删除原来 startOffset 所指方格中对应的消息，并且拉取新的消息进入窗口。滑动窗口的大小固定，所对应的用来暂存消息的缓存大小也就固定了，这部分内存开销可控。

方格大小和滑动窗口的大小同时决定了消费线程的并发数：一个方格对应一个消费线程，对于窗口大小固定的情况，方格越小并行度越高；对于方格大小固定的情况，窗口越大并行度越高。不过，若窗口设置得过大，不仅会增大内存的开销，而且在发生异常（比如 Crash）的情况下也会引起大量的重复消费，同时还考虑线程切换的开销，建议根据实际情况设置一个合理的值，不管是对于方格还是窗口而言，过大或过小都不合适。

如果一个方格内的消息无法被标记为消费完成，那么就会造成 startOffset 的悬停。为了使窗口能够继续向前滑动，那么就需要设定一个阈值，当 startOffset 悬停一定的时间后就对这部分消息进行本地重试消费，如果重试失败就转入重试队列，如果还不奏效就转入死信队列。真实应用中无法消费的情况极少，一般是由业务代码的处理逻辑引起的，比如消息中的内容格式与业务处理的内容格式不符，无法对这条消息进行决断，这种情况可以通过优化代码逻辑或采取丢弃策略来避免。如果需要消息高度可靠，也可以将无法进行业务逻辑的消息（这类消息可以称为死信）存入磁盘、数据库或 Kafka，然后继续消费下一条消息以保证整体消费进度合理推进，之后可以通过一个额外的处理任务来分析死信进而找出异常的原因。