18 | Kafka中位移提交那些事儿

2019-07-13 胡夕

你好，我是胡夕。今天我们来聊聊Kafka中位移提交的那些事儿。

之前我们说过，Consumer端有个位移的概念，它和消息在分区中的位移不是一回事儿，虽然它们的英文都是Offset。今天我们要聊的位移是Consumer的消费位移，它记录了Consumer要消费的下一条消息的位移。这可能和你以前了解的有些出入，不过切记是下一条消息的位移，而不是目前最新消费消息的位移。

我来举个例子说明一下。假设一个分区中有10条消息，位移分别是0到9。某个Consumer应用已消费了5条消息，这就说明该Consumer消费了位移为0到4的5条消息，此时Consumer的位移是5，指向了下一条消息的位移。

Consumer需要向Kafka汇报自己的位移数据，这个汇报过程被称为提交位移（Committing Offsets）。因为Consumer能够同时消费多个分区的数据，所以位移的提交实际上是在分区粒度上进行的，即**Consumer需要为分配给它的每个分区提交各自的位移数据**。

提交位移主要是为了表征Consumer的消费进度，这样当Consumer发生故障重启之后，就能够从Kafka中读取之前提交的位移值，然后从相应的位移处继续消费，从而避免整个消费过程重来一遍。换句话说，位移提交是Kafka提供给你的一个工具或语义保障，你负责维持这个语义保障，即如果你提交了位移X，那么Kafka会认为所有位移值小于X的消息你都已经成功消费了。

这一点特别关键。因为位移提交非常灵活，你完全可以提交任何位移值，但由此产生的后果你也

要一并承担。假设你的Consumer消费了10条消息，你提交的位移值却是20，那么从理论上讲，位移介于11～19之间的消息是有可能丢失的；相反地，如果你提交的位移值是5，那么位移介于5～9之间的消息就有可能被重复消费。所以，我想再强调一下，**位移提交的语义保障是由你来负责的，Kafka只会“无脑”地接受你提交的位移**。你对位移提交的管理直接影响了你的Consumer所能提供的消息语义保障。

鉴于位移提交甚至是位移管理对Consumer端的巨大影响，Kafka，特别是KafkaConsumer API，提供了多种提交位移的方法。**从用户的角度来说，位移提交分为自动提交和手动提交；从Consumer端的角度来说，位移提交分为同步提交和异步提交**。

我们先来说说自动提交和手动提交。所谓自动提交，就是指Kafka Consumer在后台默默地为你提交位移，作为用户的你完全不必操心这些事；而手动提交，则是指你要自己提交位移，Kafka Consumer压根不管。

开启自动提交位移的方法很简单。Consumer端有个参数enable.auto.commit，把它设置为true或者压根不设置它就可以了。因为它的默认值就是true，即Java Consumer默认就是自动提交位移的。如果启用了自动提交，Consumer端还有个参数就派上用场了：auto.commit.interval.ms。它的默认值是5秒，表明Kafka每5秒会为你自动提交一次位移。

为了把这个问题说清楚，我给出了完整的Java代码。这段代码展示了设置自动提交位移的方法。有了这段代码做基础，今天后面的讲解我就不再展示完整的代码了。

```
Properties props = new Properties();
     props.put("bootstrap.servers", "localhost:9092");
     props.put("group.id", "test");
     props.put("enable.auto.commit", "true");
     props.put("auto.commit.interval.ms", "2000");
     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.subscribe(Arrays.asList("foo", "bar"));
     while (true) {
         ConsumerRecords<String, String> records = consumer.poll(100);
         for (ConsumerRecord<String, String> record : records)
             System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());
     }
```

上面的橙色粗体部分，就是开启自动提交位移的方法。总体来说，还是很简单的吧。

和自动提交相反的，就是手动提交了。开启手动提交位移的方法就是设置enable.auto.commit为false。但是，仅仅设置它为false还不够，因为你只是告诉Kafka Consumer不要自动提交位移而已，你还需要调用相应的API手动提交位移。

最简单的API就是KafkaConsumer#commitSync()。该方法会提交KafkaConsumer#poll()返回的最新位移。从名字上来看，它是一个同步操作，即该方法会一直等待，直到位移被成功提交才会返回。如果提交过程中出现异常，该方法会将异常信息抛出。下面这段代码展示了commitSync()的使用方法：

```
while (true) {
        ConsumerRecords<String, String> records =
                consumer.poll(Duration.ofSeconds(1));
        process(records); // 处理消息
        try {
                consumer.commitSync();
        } catch (CommitFailedException e) {
                handle(e); // 处理提交失败异常
        }
}
```

可见，调用consumer.commitSync()方法的时机，是在你处理完了poll()方法返回的所有消息之后。如果你莽撞地过早提交了位移，就可能会出现消费数据丢失的情况。那么你可能会问，自动提交位移就不会出现消费数据丢失的情况了吗？它能恰到好处地把握时机进行位移提交吗？为了搞清楚这个问题，我们必须要深入地了解一下自动提交位移的顺序。

一旦设置了enable.auto.commit为true，Kafka会保证在开始调用poll方法时，提交上次poll返回的所有消息。从顺序上来说，poll方法的逻辑是先提交上一批消息的位移，再处理下一批消息，因此它能保证不出现消费丢失的情况。但自动提交位移的一个问题在于，**它可能会出现重复消费**。

在默认情况下，Consumer每5秒自动提交一次位移。现在，我们假设提交位移之后的3秒发生了Rebalance操作。在Rebalance之后，所有Consumer从上一次提交的位移处继续消费，但该位移已经是3秒前的位移数据了，故在Rebalance发生前3秒消费的所有数据都要重新再消费一次。虽然你能够通过减少auto.commit.interval.ms的值来提高提交频率，但这么做只能缩小重复消费的时间窗口，不可能完全消除它。这是自动提交机制的一个缺陷。

反观手动提交位移，它的好处就在于更加灵活，你完全能够把控位移提交的时机和频率。但是，它也有一个缺陷，就是在调用commitSync()时，Consumer程序会处于阻塞状态，直到远端的

Broker返回提交结果，这个状态才会结束。在任何系统中，因为程序而非资源限制而导致的阻塞都可能是系统的瓶颈，会影响整个应用程序的TPS。当然，你可以选择拉长提交间隔，但这样做的后果是Consumer的提交频率下降，在下次Consumer重启回来后，会有更多的消息被重新消费。

鉴于这个问题，Kafka社区为手动提交位移提供了另一个API方法：KafkaConsumer#commitAsync()。从名字上来看它就不是同步的，而是一个异步操作。调用commitAsync()之后，它会立即返回，不会阻塞，因此不会影响Consumer应用的TPS。由于它是异步的，Kafka提供了回调函数（callback），供你实现提交之后的逻辑，比如记录日志或处理异常等。下面这段代码展示了调用commitAsync()的方法：

```
while (true) {
            ConsumerRecords<String, String> records =
 consumer.poll(Duration.ofSeconds(1));
            process(records); // 处理消息
            consumer.commitAsync((offsets, exception) -> {
 if (exception != null)
 handle(exception);
 });
}
```

commitAsync是否能够替代commitSync呢？答案是不能。commitAsync的问题在于，出现问题时它不会自动重试。因为它是异步操作，倘若提交失败后自动重试，那么它重试时提交的位移值可能早已经“过期”或不是最新值了。因此，异步提交的重试其实没有意义，所以commitAsync是不会重试的。

显然，如果是手动提交，我们需要将commitSync和commitAsync组合使用才能到达最理想的效果，原因有两个：

1. 我们可以利用commitSync的自动重试来规避那些瞬时错误，比如网络的瞬时抖动，Broker端GC等。因为这些问题都是短暂的，自动重试通常都会成功，因此，我们不想自己重试，而是希望Kafka Consumer帮我们做这件事。
2. 我们不希望程序总处于阻塞状态，影响TPS。

我们来看一下下面这段代码，它展示的是如何将两个API方法结合使用进行手动提交。

```
   try {
           while (true) {
                   ConsumerRecords<String, String> records =
                           consumer.poll(Duration.ofSeconds(1));
                   process(records); // 处理消息
                   commitAysnc(); // 使用异步提交规避阻塞
           }
} catch (Exception e) {
           handle(e); // 处理异常
} finally {
           try {
                   consumer.commitSync(); // 最后一次提交使用同步阻塞式提交
 } finally {
      consumer.close();
}
}
```

这段代码同时使用了commitSync()和commitAsync()。对于常规性、阶段性的手动提交，我们调用commitAsync()避免程序阻塞，而在Consumer要关闭前，我们调用commitSync()方法执行同步阻塞式的位移提交，以确保Consumer关闭前能够保存正确的位移数据。将两者结合后，我们既实现了异步无阻塞式的位移管理，也确保了Consumer位移的正确性，所以，如果你需要自行编写代码开发一套Kafka Consumer应用，那么我推荐你使用上面的代码范例来实现手动的位移提交。

我们说了自动提交和手动提交，也说了同步提交和异步提交，这些就是Kafka位移提交的全部了吗？其实，我们还差一部分。

实际上，Kafka Consumer API还提供了一组更为方便的方法，可以帮助你实现更精细化的位移管理功能。刚刚我们聊到的所有位移提交，都是提交poll方法返回的所有消息的位移，比如poll方法一次返回了500条消息，当你处理完这500条消息之后，前面我们提到的各种方法会一次性地将这500条消息的位移一并处理。简单来说，就是**直接提交最新一条消息的位移**。但如果我想更加细粒度化地提交位移，该怎么办呢？

设想这样一个场景：你的poll方法返回的不是500条消息，而是5000条。那么，你肯定不想把这5000条消息都处理完之后再提交位移，因为一旦中间出现差错，之前处理的全部都要重来一遍。这类似于我们数据库中的事务处理。很多时候，我们希望将一个大事务分割成若干个小事务分别提交，这能够有效减少错误恢复的时间。

在Kafka中也是相同的道理。对于一次要处理很多消息的Consumer而言，它会关心社区有没有方法允许它在消费的中间进行位移提交。比如前面这个5000条消息的例子，你可能希望每处理完100条消息就提交一次位移，这样能够避免大批量的消息重新消费。

庆幸的是，Kafka Consumer API为手动提交提供了这样的方法：commitSync(Map<TopicPartition, OffsetAndMetadata>)和commitAsync(Map<TopicPartition, OffsetAndMetadata>)。它们的参数是一个Map对象，键就是TopicPartition，即消费的分区，而值是一个OffsetAndMetadata对象，保存的主要是位移数据。

就拿刚刚提过的那个例子来说，如何每处理100条消息就提交一次位移呢？在这里，我以commitAsync为例，展示一段代码，实际上，commitSync的调用方法和它是一模一样的。

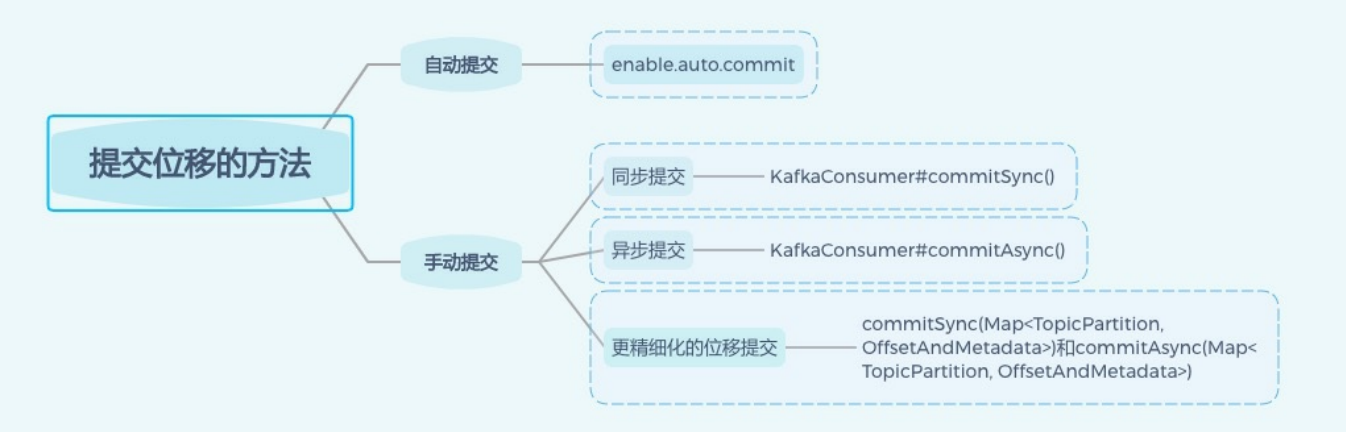
```
private Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
int count = 0;
……
while (true) {
            ConsumerRecords<String, String> records =
 consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> record: records) {
                        process(record);  // 处理消息
                        offsets.put(new TopicPartition(record.topic(), record.partition()),
                                    new OffsetAndMetadata(record.offset() + 1);
                        if（count % 100 == 0）
                                    consumer.commitAsync(offsets, null); // 回调处理逻辑是null
                        count++;
 }
}
```

简单解释一下这段代码。程序先是创建了一个Map对象，用于保存Consumer消费处理过程中要提交的分区位移，之后开始逐条处理消息，并构造要提交的位移值。还记得之前我说过要提交下一条消息的位移吗？这就是这里构造OffsetAndMetadata对象时，使用当前消息位移加1的原因。代码的最后部分是做位移的提交。我在这里设置了一个计数器，每累计100条消息就统一提交一次位移。与调用无参的commitAsync不同，这里调用了带Map对象参数的commitAsync进行细粒度的位移提交。这样，这段代码就能够实现每处理100条消息就提交一次位移，不用再受poll方法返回的消息总数的限制了。

小结

好了，我们来总结一下今天的内容。Kafka Consumer的位移提交，是实现Consumer端语义保障

的重要手段。位移提交分为自动提交和手动提交，而手动提交又分为同步提交和异步提交。在实际使用过程中，推荐你使用手动提交机制，因为它更加可控，也更加灵活。另外，建议你同时采用同步提交和异步提交两种方式，这样既不影响TPS，又支持自动重试，改善Consumer应用的高可用性。总之，Kafka Consumer API提供了多种灵活的提交方法，方便你根据自己的业务场景定制你的提交策略。



开放讨论

实际上，手动提交也不能避免消息重复消费。假设Consumer在处理完消息和提交位移前出现故障，下次重启后依然会出现消息重复消费的情况。请你思考一下，如何实现你的业务场景中的去重逻辑呢？

欢迎写下你的思考和答案，我们一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。

精选留言

nightmare 9

老师手动提交的设计很优美，先用异步提交不影响程序的性能，再用consumer关闭时同步提交来确保位移一定提交成功。这里我有个疑问，比如我程序运行期间有多次异步提交没有成功，比如101的offset和201的offset没有提交成功，程序关闭的时候501的offset提交成功了，是不是就代表前面500条我还是消费成功了，只要最新的位移提交成功，就代表之前的消息都提交成功了？第二点 就是批量提交哪里，如果一个消费者晓得多个分区的消息，封装在一个Map对象里面消费者也能正确的对多个分区的位移都保证正确的提交吗？

2019-07-13

Tony Du 2

老师，您好～ 看了今天的教程我有两个问题想请教下，希望老师能赐教。

1. 从文中的代码看上去，使用commitAsync提供offset，不需要等待异步执行结果再次poll就能拿到下一批消息，是那么这个offset的最新值是不是理解为其实是在consumer client的内存中管

理的（因为实际commitAsync如果失败的话，offset不会写入broker中）？如果是这样的话，如果在执行到commitSync之前，consumer client进程重启了，就有可能会消费到因commitAsync失败产生的重复消息。
2. 教程中手动提交100条消息的代码是一个同步处理的代码，我在实际工程中遇到的问题是，为了提高消息处理效率，coumser poll到一批消息后会提交到一个thread pool中处理，这种情况下，请教下怎样做到控制offset分批提交？
谢谢

2019-07-13

A_F

1

先说一下，课后思考，解决的办法应该就是，将消息处理和位移提交放在一个事务里面，要么都成功，要么都失败。

老师文章里面的举的一个例子没有很明白，能不能再解释一下。就是那个位移提交后Rebalance的例子。

2019-07-13

lmtoo

1

对于手动同步和异步提交结合的场景，如果poll出来的消息是500条，而业务处理200条的时候，业务抛异常了，后续消息根本就没有被遍历过，finally里手动同步提交的是201还是000，还是501？

2019-07-13

作者回复

如果调用没有参数的commit，那么提交的是500

2019-07-15

曾轼麟

0

不过老师有一个问题，poll下来的数据是有序的吗？同一个partition中各个消息的相对顺序，当然不同partition应该是不一定的

2019-07-15

作者回复

是的

2019-07-15

曾轼麟

0

我们目前的做法是kafka消费前都有一个消息接口表，可以使用Redis或者MySQL(Redis只存最近100个消息)，然后会设置consumer拉取消息的大小极限，保证消息数量不超过100(这个阈值可以自行调整)，其中我们会保证kafka消息的key是全局唯一的，比如使用雪花算法，在进行消费的时候可以通过前置表进行幂等性去重

2019-07-15

Liam

0

所以自动提交有2个时机吗？

1 固定频率提及，例如5s提及一次
2 poll新数据之前提交前面消费的数据

2019-07-15

作者回复

它们实际上是一个时机

2019-07-15

ikimiy 0

老师 consumer offset在spark程序中如何控制手动提交的 有没有sample code可以参考的 thks

2019-07-15

leaning_人生 0

避免重复消费：
1.（不考虑rebalance）producer在生成消息体是，里面加上唯一标识符比如：唯一Id，即保证消息的幂等性，consumer在处理消息的过程中，将消费后的消息Id存储到数据库中或者redis，等消息处理完毕后在手动提交offset
2.（考虑rebalance）监听consumer的rebalance，rebalance发生前将topic-partion-offset存入数据库，rebalance后根据获取到的分区信息到数据库中查找上次消费到的位置seek到上次消费位置，在处理消息中，利用数据库事务管理处理消息
2.

2019-07-15

z.l 0

消费者端实现消费幂等。具体做法：创建本地消息表，以messageId作为主键。消费消息的同时也插入消息表，把消费逻辑的sql语句和消息表的insert语句放在同一个数据库事务中。

2019-07-14

双叶 0

我理解中的 enable.auto.commit 跟文章中说的不太一样，我理解设置为 true 的时候提供的是至多一次的语义，而不是至少一次的语义。

我不用 java，看的是 librdkafka 的文档。enable.auto.commit 的文档说明是：Automatically and periodically commit offsets in the background. 也就是说他会定期提交 offset，但是这里没有明说提交的 offset 是什么时候记录的，我的理解是记录是由 enable.auto.offset.store 决定的。

enable.auto.offset.store 文档说明是：Automatically store offset of last message provided to application. The offset store is an in-memory store of the next offset to (auto-)commit for each partition. 也就是说如果设置成 true（默认值），他会自动把上个提交给应用程序的offset 记录到内存中。

也就是说，如果应用拿到一个 offset 了，librdkafka 就会把这个 offset 记录到内存中，然后默认情况下至多 5s 之后，就会提交给 broker。这时候如果应用还没有完成这个 offset 的处理时，发生了崩溃，这个 offset 就丢掉了，所以是一个至多一次的语义。

我理解中提供至少一次语义需要关掉 enable.auto.commit 自己控制提交才行。

2019-07-14

kursk.ye 0

我现在有点糊涂了，kafka的offset是以broker发消息给consumer时，broker的offset为准；还是以consumer 的commit offset为准？比如，一个partition现在的offset是99，执行poll(10)方法时，broker给consumer发送了10条记录，在broker中offset变为109；假如 enable.auto.commit 为false，为手动提交consumer offset,但是cosumer在执行consumer.commitSync()或consumer.commitAsync()时进程失败，整个consumer进程都崩溃了；于是一个新的consumer接替原consumer继续消费，那么他是从99开始消费，还是从109开始消费？

2019-07-14

作者回复

首先，poll(10)不是获取10条消息的意思。

其次，consumer获取的位移是它之前最新一次提交的位移，因此是99

2019-07-15

电光火石 0

老师好，consumer的api在读取的时候能指定从某个partition的某个offset开始读取吗？看参数只能用latest,oldest进行指定，然后用kafka记录的offset进行读取，我们能自己控制起始的offset吗，这样可以做更精准的exact once的语义

2019-07-14

作者回复

可以控制，使用KafkaConsumer.seek可以精确控制你要开始消费的位移

2019-07-15

ban 0

老师，有个疑问。commitSync 和 commitAsync 组合这个代码，捕获异常在while循环外面，如果发生异常不就不走了吗，程序也就停止。是不是放在while循环比较好，既可以处理异常，还能提交偏移量，还能继续消费处理消息？

2019-07-13

作者回复

try不是在while外面吗？

2019-07-15

ban 0

老师，你好。有个场景不太明白。我做个假设，比如说我的模式是自动提交，自动提交间隔是20秒一次，那我消费了10个消息，很快一秒内就结束。但是这时候我自动提交时间还没到（那是不是意味着不会提交offer），然后这时候我又去poll获取消息，会不会导致一直获取上一批的消息？

还是说如果consumer消费完了，自动提交时间还没到，如果你去poll，这时候会自动提交，就不会出现重复消费的情况。

2019-07-13

作者回复

不会的。consumer内部维护了一个指针，能够探测到下一条要消费的数据

2019-07-15

明翼 0

有同学问offset是否在内存控制等问题，可能是没用过kafka，kafka的消费者启动时候可以设置参数从什么位置开始读，有的是从最新的开始读，有的是从最老的开始读，从最新位置读就是从上次提交的位移读，所以提交的offset是用作下一次程序启动或重新平衡后读取的位置的。同样像老师这种先异步再同步提交数据的场景如果一次拉500条数据，消费到200条之后异常了，同步提交是提交500条的，我觉得是不是可以类似下面分批提交的方法提交不知道此方法有同步的吗？有的话应该会比较完美解决。

对于老师的问题，想不重复只有自己在程序中保留offsetid.如果后端系统有数据库类似数据库主建机制，可以用这个方法判断，插入报约束冲突就忽视...

2019-07-13

蛋炒番茄 0

自动提交就一定能够保证不丢消息吗？

2019-07-13

作者回复

不能绝对保证

2019-07-15

科莫湖畔的球童 0

Consumer自己记录一下最近一次已消费的offset

2019-07-13

海贼王 0

自动提交也可能出现消息丢失的情况，如果拉取消息和处理消息是两个线程去处理的就可能发生拉取线程拉取了两次，处理线程第一次的消息没处理完，崩溃恢复后再消费导致可能丢失某些消息。不过我觉得这不能怪kafka了，这种丢失超出kafka的责任边界了

2019-07-13

lmtoo 0

关于业务去重的逻辑，可以考虑在业务字段里加一个txid，用consumer的offset值表示事务id，如果有这个id表示被处理过了，如果没有，则表示还没处理过，这样可以利用mysql或者MongoDB来实现避免重复消费

2019-07-13