位移提交

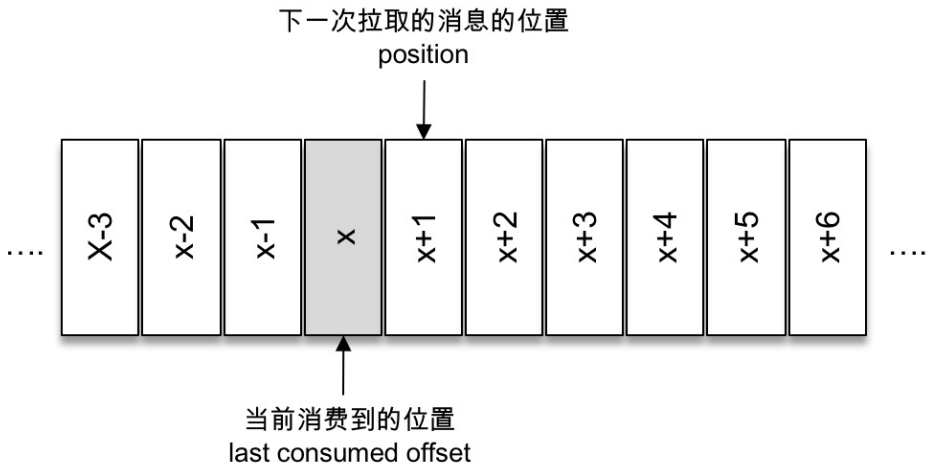
对于 Kafka 中的分区而言，它的每条消息都有唯一的 offset，用来表示消息在分区中对应的位置。对于消费者而言，它也有一个 offset 的概念，消费者使用 offset 来表示消费到分区中某个消息所在的位置。

单词“offset”可以翻译为“偏移量”，也可以翻译为“位移”，读者可能并没有过多地在意这一点：在很多中文资料中都会交叉使用“偏移量”和“位移”这两个词，并没有很严谨地进行区分。笔者对 offset 做了一些区分：对于消息在分区中的位置，我们将 offset 称为“偏移量”；对于消费者消费到的位置，将 offset 称为“位移”，有时候也会更明确地称之为“消费位移”。

做这一区分的目的是让读者在遇到 offset 的时候可以很容易甄别出是在讲分区存储层面的内容，还是在讲消费层面的内容，如此也可以使“偏移量”和“位移”这两个中文词汇具备更加丰富的意义。当然，对于一条消息而言，它的偏移量和消费者消费它时的消费位移是相等的，在某些不需要具体划分的场景下也可以用“消息位置”或直接用“offset”这个单词来进行表述。

在每次调用 poll() 方法时，它返回的是还没有被消费过的消息集（当然这个前提是消息已经存储在 Kafka 中了，并且暂不考虑异常情况的发生），要做到这一点，就需要记录上一次消费时的消费位移。并且这个消费位移必须做持久化保存，而不是单单保存在内存中，否则消费者重启之后就无法知晓之前的消费位移。再考虑一种情况，当有新的消费者加入时，那么必然会有再均衡的动作，对于同一分区而言，它可能在再均衡动作之后分配给新的消费者，如果不持久化保存消费位移，那么这个新的消费者也无法知晓之前的消费位移。

在旧消费者客户端中，消费位移是存储在 ZooKeeper 中的。而在新消费者客户端中，消费位移存储在 Kafka 内部的主题 __consumer_offsets 中。这里把将消费位移存储起来（持久化）的动作称为“提交”，消费者在消费完消息之后需要执行消费位移的提交。



参考上图中的消费位移，x表示某一次拉取操作中此分区消息的最大偏移量，假设当前消费者已经消费了x位置的消息，那么我们就可以说消费者的消费位移为x，图中也用了 lastConsumedOffset 这个单词来标识它。

不过需要非常明确的是，当前消费者需要提交的消费位移并不是x，而是x+1，对应于上图中的 position，它表示下一条需要拉取的消息的位置。读者可能看过一些相关资料，里面所讲述的内容可能是提交的消费位移就是当前所消费到的消费位移，即提交的是x，这明显是错误的。类似的错误还体现在对 LEO（Log End Offset） 的解读上。在消费者中还有一个 committed offset 的概念，它表示已经提交过的消费位移。

KafkaConsumer 类提供了 position(TopicPartition) 和 committed(TopicPartition) 两个方法来分别获取上面所说的 position 和 committed offset 的值。这两个方法的定义如下所示。

```
public long position(TopicPartition partition)
public OffsetAndMetadata committed(TopicPartition
partition)
```

为了论证 lastConsumedOffset、committed offset 和 position 之间的关系，我们使用上面的这两个方法来做相关演示。我们向某个主题中分区编号为0的分区发送若干消息，之后再创建一个消费者去消费其中的消息，等待消费完这些消息之后就同步提交消费位移（调用 commitSync() 方法，这个方法的细节在下面详细介绍），最后我们观察一下 lastConsumedOffset、committed offset 和 position 的值。示例代码如代码清单11-1所示。

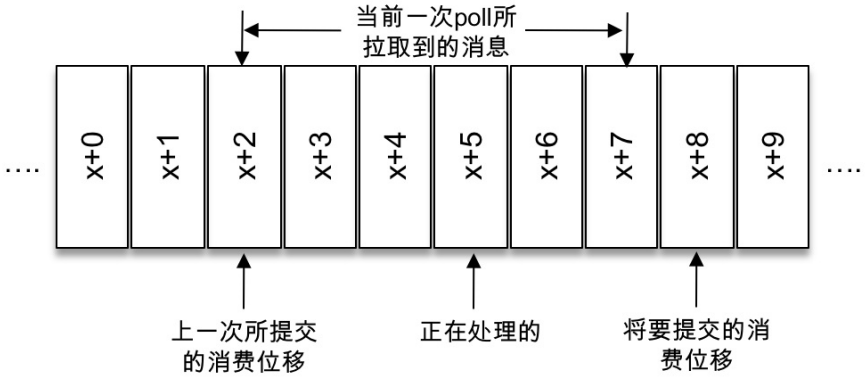
```
//代码清单11-1 消费位移的演示
TopicPartition tp = new TopicPartition(topic, 0);
consumer.assign(Arrays.asList(tp));
long lastConsumedOffset = -1;//当前消费到的位移
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(1000);
    if (records.isEmpty()) {
        break;
    }
    List<ConsumerRecord<String, String>>
partitionRecords
            = records.records(tp);
    lastConsumedOffset = partitionRecords
            .get(partitionRecords.size() -
1).offset();
    consumer.commitSync();//同步提交消费位移
}
System.out.println("comsumed offset is " +
lastConsumedOffset);
OffsetAndMetadata offsetAndMetadata =
consumer.committed(tp);
System.out.println("commited offset is " +
offsetAndMetadata.offset());
long posititon = consumer.position(tp);
System.out.println("the offset of the next record
is " + posititon);
```

示例中先通过 assign() 方法订阅了编号为0的分区，然后消费分区中的消息。示例中还通过调用 ConsumerRecords.isEmpty() 方法来判断是否已经消费完分区中的消息，以此来退出 while(true) 的循环，当然这段逻辑并不严谨，这里只是用来演示，读者切勿在实际开发中效仿。

最终的输出结果如下：

```
comsumed offset is 377
commited offset is 378
the offset of the next record is 378
```

可以看出，消费者消费到此分区消息的最大偏移量为377，对应的消费位移 lastConsumedOffset 也就是377。在消费完之后就执行同步提交，但是最终结果显示所提交的位移 committed offset 为378，并且下一次所要拉取的消息的起始偏移量 position 也为378。在本示例中，position = committed offset = lastConsumedOffset + 1，当然 position 和 committed offset 并不会一直相同，这一点会在下面的示例中有所体现。



对于位移提交的具体时机的把握也很有讲究，有可能会造成重复消费和消息丢失的现象。参考上图，当前一次 poll() 操作所拉取的消息集为 [x+2, x+7]，x+2 代表上一次提交的消费位移，说明已经完成了 x+1 之前（包括 x+1 在内）的所有消息的消费，x+5 表示当前正在处理的位置。如果拉取到消息之后就进行了位移提交，即提交了 x+8，那么当前消费 x+5 的时候遇到了异常，在故障恢复之后，我们重新拉取的消息是从 x+8 开始的。也就是说，x+5 至 x+7 之间的消息并未能被消费，如此便发生了消息丢失的现象。

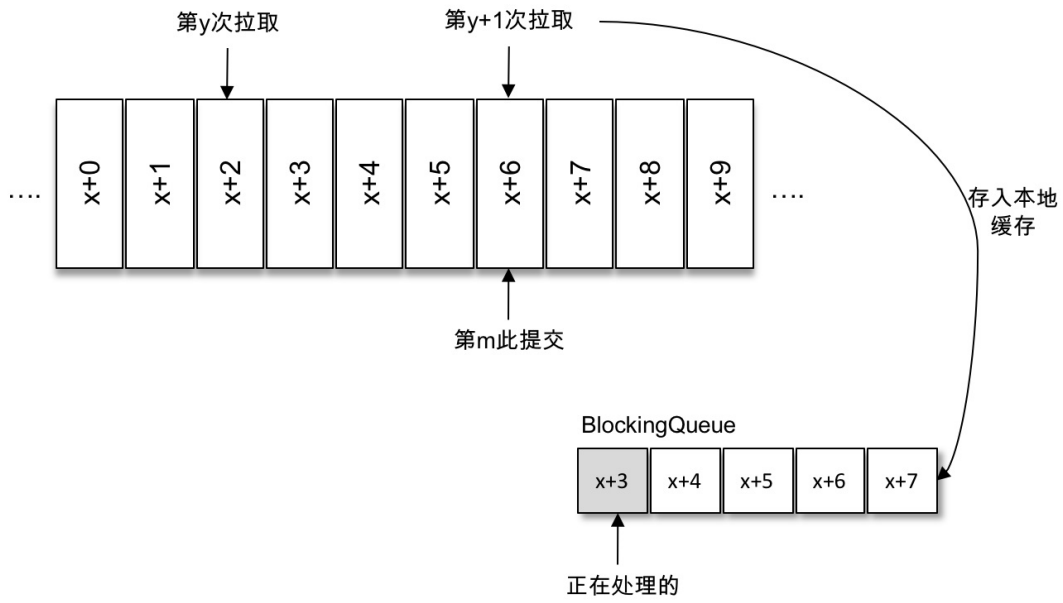
再考虑另外一种情形，位移提交的动作是在消费完所有拉取到的消息之后才执行的，那么当消费 x+5 的时候遇到了异常，在故障恢复之后，我们重新拉取的消息是从 x+2 开始的。也就是说，x+2 至 x+4 之间的消息又重新消费了一遍，故而又发生了重复消费的现象。

而实际情况还会有比这两种更加复杂的情形，比如第一次的位移提交的位置为 x+8，而下一次的位移提交的位置为 x+4，后面会做进一步的分析。

在 Kafka 中默认的消费位移的提交方式是自动提交，这个由消费者客户端参数 enable.auto.commit 配置，默认值为 true。当然这个默认的自动提交不是每消费一条消息就提交一次，而是定期提交，这个定期的周期时间由客户端参数 auto.commit.interval.ms 配置，默认值为5秒，此参数生效的前提是 enable.auto.commit 参数为 true。在代码清单8-1中并没有展示出这两个参数，说明使用的正是默认值。

在默认的方式下，消费者每隔5秒会将拉取到的每个分区中最大的消息位移进行提交。自动位移提交的动作是在 poll() 方法的逻辑里完成的，在每次真正向服务端发起拉取请求之前会检查是否可以进行位移提交，如果可以，那么就会提交上一次轮询的位移。

在 Kafka 消费的编程逻辑中位移提交是一大难点，自动提交消费位移的方式非常简便，它免去了复杂的位移提交逻辑，让编码更简洁。但随之而来的是重复消费和消息丢失的问题。假设刚刚提交完一次消费位移，然后拉取一批消息进行消费，在下一次自动提交消费位移之前，消费者崩溃了，那么又得从上一次位移提交的地方重新开始消费，这样便发生了重复消费的现象（对于再均衡的情况同样适用）。我们可以通过减小位移提交的时间间隔来减小重复消息的窗口大小，但这样并不能避免重复消费的发送，而且也会使位移提交更加频繁。



按照一般思维逻辑而言，自动提交是延时提交，重复消费可以理解，那么消息丢失又是在什么情形下会发生的呢？我们来看一下上图中的情形。拉取线程A不断地拉取消息并存入本地缓存，比如在 BlockingQueue 中，另一个处理线程B从缓存中读取消息并进行相应的逻辑处理。假设目前进行到了第 y+1 次拉取，以及第m次位移提交的时候，也就是 x+6 之前的位移已经确认提交了，处理线程B却还正在消费 x+3 的消息。此时如果处理线程B发生了异常，待其恢复之后会从第m此位移提交处，也就是 x+6 的位置开始拉取消息，那么 x+3 至 x+6 之间的消息就没有得到相应的处理，这样便发生消息丢失的现象。

自动位移提交的方式在正常情况下不会发生消息丢失或重复消费的现象，但是在编程的世界里异常无可避免，与此同时，自动位移提交也无法做到精确的位移管理。在 Kafka 中还提供了手动位移提交的方式，这样可以使得开发人员对消费位移的管理控制更加灵活。很多时候并不是说拉取到消息就算消费完成，而是需要将消息写入数据库、写入本地缓存，或者是更加复杂的业务处理。在这些场景下，所有的业务处理完成才能认为消息被成功消费，手动的提交方式可以让开发人员根据程序的逻辑在合适的地方进行位移提交。开启手动提交功能的前提是消费者客户端参数 enable.auto.commit 配置为 false，示例如下：

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

手动提交可以细分为同步提交和异步提交，对应于 KafkaConsumer 中的 commitSync() 和 commitAsync() 两种类型的方法。我们这里先讲述同步提交的方式，commitSync() 方法的定义如下：

```
public void commitSync()
```

这个方法很简单，下面使用它演示同步提交的简单用法：

```
while (isRunning.get()) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        //do some logical processing.
    }
    consumer.commitSync();
}
```

可以看到示例中先对拉取到的每一条消息做相应的逻辑处理，然后对整个消息集做同步提交。参考 KafkaConsumer 源码中提供的示例，针对上面的示例还可以修改为批量处理+批量提交的方式，关键代码如下：

```
final int minBatchSize = 200;
List<ConsumerRecord> buffer = new ArrayList<>();
while (isRunning.get()) {
    ConsumerRecords<String, String> records =
consumer.poll(1000);
    for (ConsumerRecord<String, String> record :
records) {
        buffer.add(record);
    }
    if (buffer.size() >= minBatchSize) {
        //do some logical processing with buffer.
        consumer.commitSync();
        buffer.clear();
    }
}
```

上面的示例中将拉取到的消息存入缓存 buffer，等到积累到足够多的时候，也就是示例中大于等于200个的时候，再做相应的批量处理，之后再做批量提交。这两个示例都有重复消费的问题，如果在业务逻辑处理完之后，并且在同步位移提交前，程序出现了崩溃，那么待恢复之后又只能从上一次位移提交的地方拉取消息，由此在两次位移提交的窗口中出现了重复消费的现象。

commitSync() 方法会根据 poll() 方法拉取的最新位移来进行提交（注意提交的值对应于本节第1张图中 position 的位置），只要没有发生不可恢复的错误（Unrecoverable Error），它就会阻塞消费者线程直至位移提交完成。对于不可恢复的错误，比如 CommitFailedException、WakeupException、InterruptException、AuthenticationException、AuthorizationException 等，我们可以将其捕获并做针对性的处理。

对于采用 commitSync() 的无参方法而言，它提交消费位移的频率和拉取批次消息、处理批次消息的频率是一样的，如果想寻求更细粒度的、更精准的提交，那么就需要使用 commitSync() 的另一个含参方法，具体定义如下：

```
public void commitSync(final Map<TopicPartition,
OffsetAndMetadata> offsets)
```

该方法提供了一个 offsets 参数，用来提交指定分区的位移。无参的 commitSync() 方法只能提交当前批次对应的 position 值。如果需要提交一个中间值，比如业务每消费一条消息就提交一次位移，那么就可以使用这种方式，我们来看一下代码示例，如代码清单11-2所示。

```
//代码清单11-2 带参数的同步位移提交
while (isRunning.get()) {
    ConsumerRecords<String, String> records =
consumer.poll(1000);
    for (ConsumerRecord<String, String> record :
records) {
        //do some logical processing.
        long offset = record.offset();
        TopicPartition partition =
                new
TopicPartition(record.topic(),
record.partition());
        consumer.commitSync(Collections
                .singletonMap(partition, new
OffsetAndMetadata(offset + 1)));
    }
}
```

在实际应用中，很少会有这种每消费一条消息就提交一次消费位移的必要场景。commitSync() 方法本身是同步执行的，会耗费一定的性

能，而示例中的这种提交方式会将性能拉到一个相当低的点。更多时候是按照分区的粒度划分提交位移的界限，这里我们就要用到了第10节中提及的 ConsumerRecords 类的 partitions() 方法和 records(TopicPartition) 方法，关键示例代码如代码清单11-3所示（修改自 KafkaConsumer 源码中的示例）。

```
//代码清单11-3 按分区粒度同步提交消费位移
try {
    while (isRunning.get()) {
        ConsumerRecords<String, String> records =
consumer.poll(1000);
        for (TopicPartition partition :
records.partitions()) {
            List<ConsumerRecord<String, String>>
partitionRecords =
                    records.records(partition);
            for (ConsumerRecord<String, String>
record : partitionRecords) {
                //do some logical processing.
            }
            long lastConsumedOffset =
partitionRecords
                    .get(partitionRecords.size()
- 1).offset();

consumer.commitSync(Collections.singletonMap(part
ition,
                    new
OffsetAndMetadata(lastConsumedOffset + 1)));
        }
    }
} finally {
    consumer.close();
}
```

与 commitSync() 方法相反，异步提交的方式（commitAsync()）在执行的时候消费者线程不会被阻塞，可能在提交消费位移的结果还未返回之前就开始了新一次的拉取操作。异步提交可以使消费者的性

能得到一定的增强。commitAsync 方法有三个不同的重载方法，具体定义如下：

```
public void commitAsync()
public void commitAsync(OffsetCommitCallback
callback)
public void commitAsync(final Map<TopicPartition,
OffsetAndMetadata> offsets,
                        OffsetCommitCallback callback)
```

第一个无参的方法和第三个方法中的 offsets 都很好理解，对照 commitSync() 方法即可。关键的是这里的第二个方法和第三个方法中的 callback 参数，它提供了一个异步提交的回调方法，当位移提交完成后会回调 OffsetCommitCallback 中的 onComplete() 方法。这里采用第二个方法来演示回调函数的用法，关键代码如下：

```
while (isRunning.get()) {
    ConsumerRecords<String, String> records =
consumer.poll(1000);
    for (ConsumerRecord<String, String> record :
records) {
        //do some logical processing.
    }
    consumer.commitAsync(new
OffsetCommitCallback() {
        @Override
        public void
onComplete(Map<TopicPartition, OffsetAndMetadata>
offsets,
                                   Exception
exception) {
            if (exception == null) {
                System.out.println(offsets);
            }else {
                log.error("fail to commit offsets
{}", offsets, exception);
            }
        }
    });
}
```

commitAsync() 提交的时候同样会有失败的情况发生，那么我们应该怎么处理呢？读者有可能想到的是重试，问题的关键也就在这里了。如果某一次异步提交的消费位移为x，但是提交失败了，然后下一次又异步提交了消费位移为x+y，这次成功了。如果这里引入了重试机制，前一次的异步提交的消费位移在重试的时候提交成功了，那么此时的消费位移又变为了x。如果此时发生异常（或者再均衡），那么恢复之后的消费者（或者新的消费者）就会从x处开始消费消息，这样就发生了重复消费的问题。

为此我们可以设置一个递增的序号来维护异步提交的顺序，每次位移提交之后就增加序号相对应的值。在遇到位移提交失败需要重试的时候，可以检查所提交的位移和序号的值的大小，如果前者小于后者，则说明有更大的位移已经提交了，不需要再进行本次重试；如果两者相同，则说明可以进行重试提交。除非程序编码错误，否则不会出现前者大于后者的情况。

如果位移提交失败的情况经常发生，那么说明系统肯定出现了故障，在一般情况下，位移提交失败的情况很少发生，不重试也没有关系，后面的提交也会有成功的。重试会增加代码逻辑的复杂度，不重试会增加重复消费的概率。如果消费者异常退出，那么这个重复消费的问题就很难避免，因为这种情况下无法及时提交消费位移；如果消费者正常退出或发生再均衡的情况，那么可以在退出或再均衡执行之前使用同步提交的方式做最后的把关。

```
try {
    while (isRunning.get()) {
        //poll records and do some logical
processing.
        consumer.commitAsync();
    }
} finally {
    try {
        consumer.commitSync();
    }finally {
        consumer.close();
    }
}
```

示例代码中加粗的部分是在消费者正常退出时为位移提交“把关”添加的。发生再均衡情况的“把关”会在第13节中做详细介绍。

控制或关闭消费

KafkaConsumer 提供了对消费速度进行控制的方法，在有些应用场景下我们可能需要暂停某些分区的消费而先消费其他分区，当达到一定条件时再恢复这些分区的消费。KafkaConsumer 中使用 pause() 和 resume() 方法来分别实现暂停某些分区在拉取操作时返回数据给客户端和恢复某些分区向客户端返回数据的操作。这两个方法的具体定义如下：

```
public void pause(Collection<TopicPartition>
partitions)
public void resume(Collection<TopicPartition>
partitions)
```

KafkaConsumer 还提供了一个无参的 paused() 方法来返回被暂停的分区集合，此方法的具体定义如下：

```
public Set<TopicPartition> paused()
```

之前的示例展示的都是使用一个 while 循环来包裹住 poll() 方法及相应的消费逻辑，如何优雅地退出这个循环也很有考究。细心的读者可能注意到有些示例代码并不是以 while(true) 的形式做简单的包裹，而是使用 while(isRunning.get()) 的方式，这样可以通过在其他地方设定 isRunning.set(false) 来退出 while 循环。还有一种方式是调用 KafkaConsumer 的 wakeup() 方法，wakeup() 方法是 KafkaConsumer 中唯一可以从其他线程里安全调用的方法（KafkaConsumer 是非线程安全的，可以通过14节了解更多细节），调用 wakeup() 方法后可以退出 poll() 的逻辑，并抛出 WakeupException 的异常，我们也不需要处理 WakeupException 的异常，它只是一种跳出循环的方式。

跳出循环以后一定要显式地执行关闭动作以释放运行过程中占用的各种系统资源，包括内存资源、Socket 连接等。KafkaConsumer 提供了 close() 方法来实现关闭，close() 方法有三种重载方法，分别如下：

```
public void close()
public void close(Duration timeout)
@Deprecated
public void close(long timeout, TimeUnit
timeUnit)
```

第二种方法是通过 timeout 参数来设定关闭方法的最长执行时间，有些内部的关闭逻辑会耗费一定的时间，比如设置了自动提交消费位移，这里还会做一次位移提交的动作；而第一种方法没有 timeout 参数，这并不意味着会无限制地等待，它内部设定了最长等待时间（30秒）；第三种方法已被标记为 @Deprecated，可以不考虑。一个相对完整的消费程序的逻辑可以参考下面的伪代码：

```
consumer.subscribe(Arrays.asList(topic));
try {
    while (running.get()) {
        //consumer.poll(***)
        //process the record.
        //commit offset.
    }
} catch (WakeupException e) {
    // ingore the error
} catch (Exception e){
    // do some logic process.
} finally {
    // maybe commit offset.
    consumer.close();
}
```

当关闭这个消费逻辑的时候，可以调用 consumer.wakeup()，也可以调用 isRunning.set(false)。