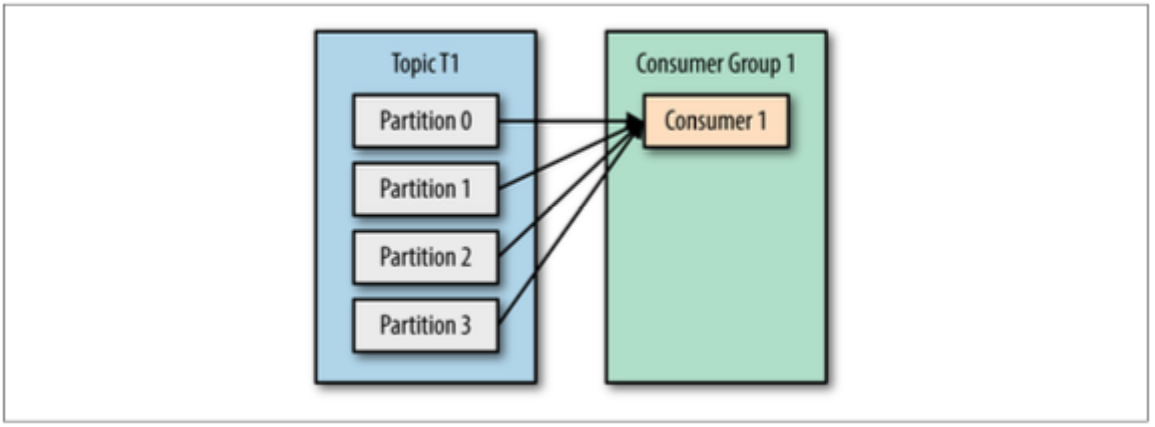
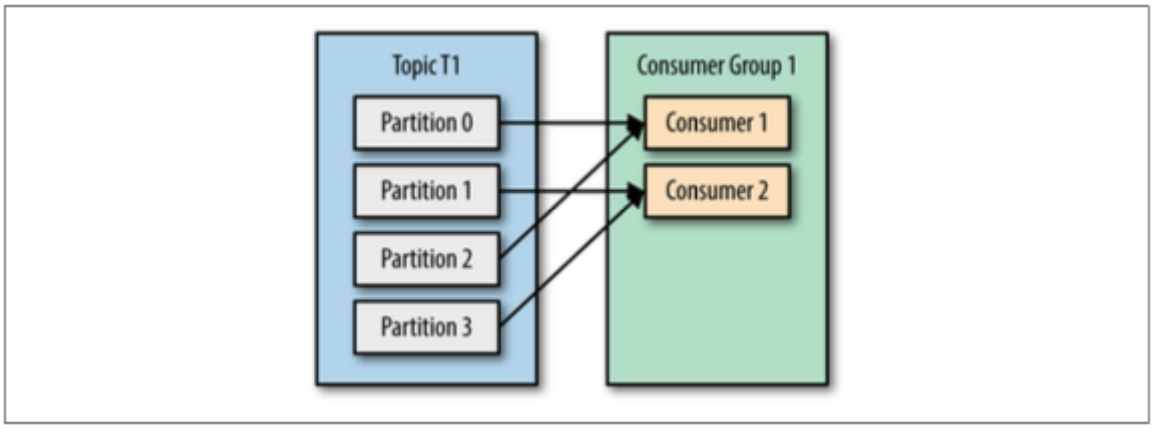
消费者与消费组

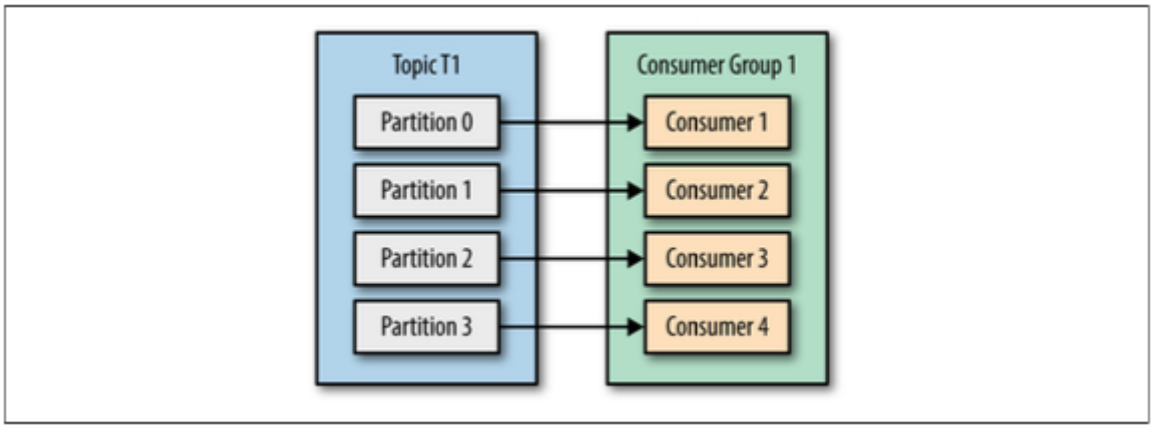
Kafka消费者是消费组的一部分，当多个消费者形成一个消费组来消费主题时，每个消费者会收到不同分区的消息。假设有一个T1主题，该主题有4个分区；同时我们有一个消费组G1，这个消费组只有一个消费者C1。那么消费者C1将会收到这4个分区的消息，如下所示：



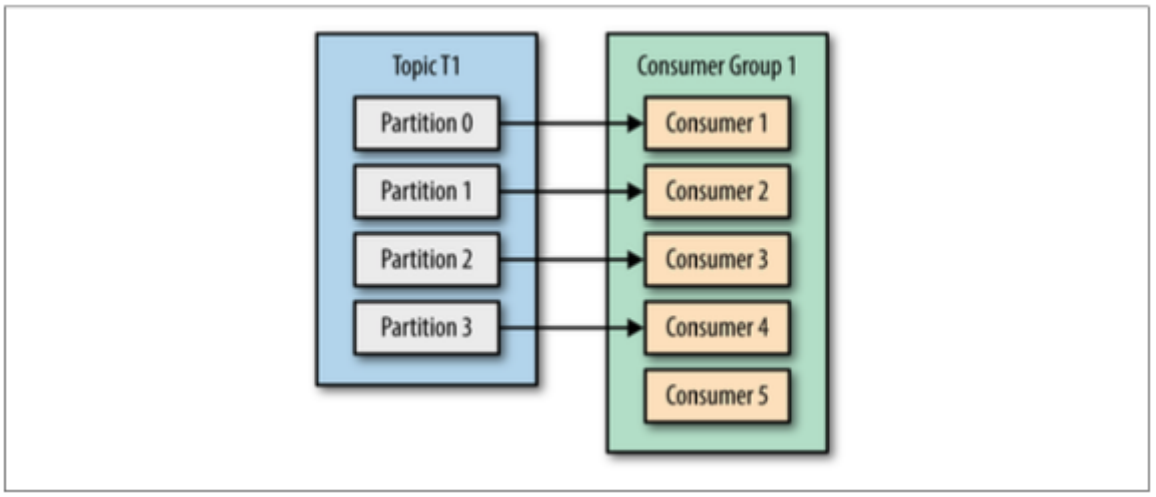
如果我们增加新的消费者C2到消费组G1，那么每个消费者将会分别收到两个分区的消息，如下所示：



如果增加到4个消费者，那么每个消费者将会分别收到一个分区的消息，如下所示：

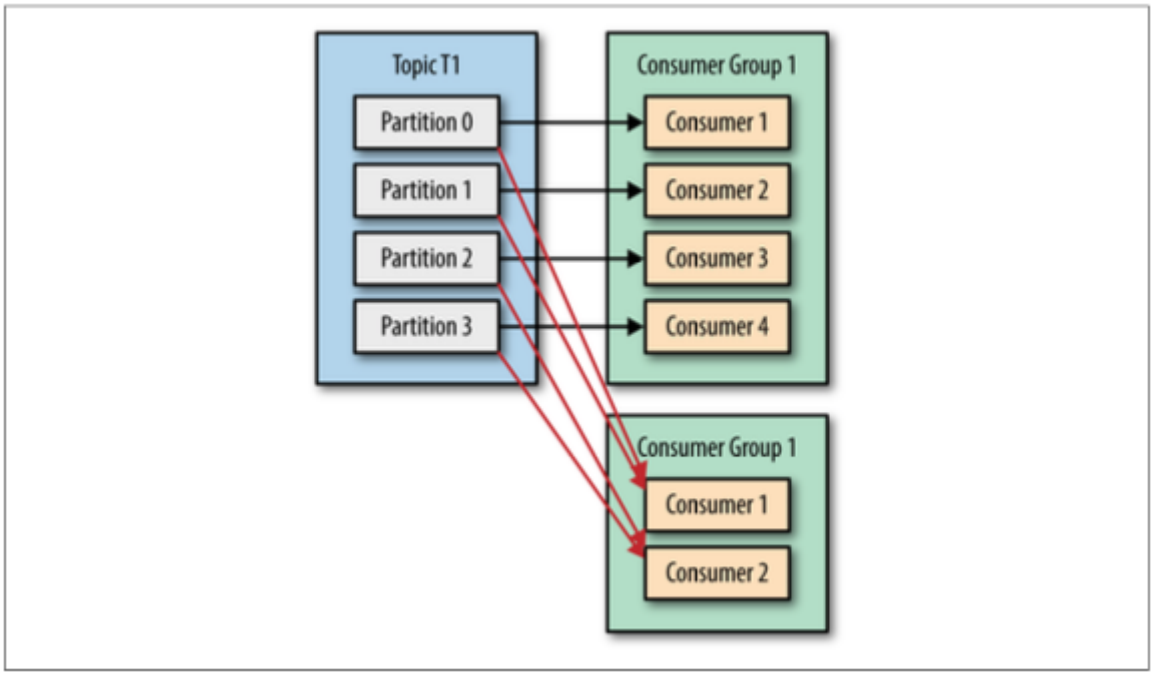


但如果我们继续增加消费者到这个消费组，剩余的消费者将会空闲，不会收到任何消息：



总而言之，我们可以通过增加消费组的消费者来进行水平扩展提升消费能力。这也是为什么建议创建主题时使用比较多的分区数，这样可以在消费负载高的情况下增加消费者来提升性能。另外，消费者的数量不应该比分区数多，因为多出来的消费者是空闲的，没有任何帮助。

Kafka一个很重要的特性就是，只需写入一次消息，可以支持任意多的应用读取这个消息。换句话说，每个应用都可以读到全量的消息。为了使得每个应用都能读到全量消息，应用需要有不同的消费组。对于上面的例子，假如我们新增了一个新的消费组G2，而这个消费组有两个消费者，那么会是这样的：



在这个场景中，消费组G1和消费组G2都能收到T1主题的全量消息，在逻辑意义上来说它们属于不同的应用。

最后，总结起来就是：如果应用需要读取全量消息，那么请为该应用设置一个消费组；如果该应用消费能力不足，那么可以考虑在这个消费组里增加消费者。

总结：每个组都能够获取kafka的所有数据，如果要获取所有数据只需设置不同的group.id即可

消费者与分区重平衡

可以看到，当新的消费者加入消费组，它会消费一个或多个分区，而这些分区之前是由其他消费者负责的；另外，当消费者离开消费组（比如重启、宕机等）时，它所消费的分区会分配给其他分区。这种现象称为重平衡（rebalance）。重平衡是Kafka一个很重要的性质，这个性质保证了高可用和水平扩展。不过也需要注意到，在重平衡期间，所有消费者都不能消费消息，因此会造成整个消费组短暂的不可用。而且，将分区进行重平衡也会导致原来的消费者状态过期，从而导致消费者需要重新更新状态，这段期间也会降低消费性能。后面我们会讨论如何安全的进行重平衡以及如何尽可能避免。

消费者通过定期发送心跳（hearbeat）到一个作为组协调者（group coordinator）的broker来保持在消费组内存活。这个broker不是固定的，每个消费组都可能不同。当消费者拉取消息或者提交时，便会发送心跳。

如果消费者超过一定时间没有发送心跳，那么它的会话（session）就会过期，组协调者会认为该消费者已经宕机，然后触发重平衡。可以看到，从消费者宕机到会话过期是有一定时间的，这段时间内该消费者的分区都不能进行消息消费；通常情况下，我们可以进行优雅关闭，这样消费者会发送离开的消息到组协调者，这样组协调者可以立即进行重平衡而不需要等待会话过期。

在0.10.1版本，Kafka对心跳机制进行了修改，将发送心跳与拉取消息进行分离，这样使得发送心跳的频率不受拉取的频率影响。另外更高版本的Kafka支持配置一个消费者多长时间不拉取消息但仍然保持存活，这个配置可以避免活锁（livelock）。活锁，是指应用没有故障但是由于某些原因不能进一步消费。

从kafka消费代码中只需要以下几步

创建kafka消费者

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
```

```
5 props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
6 KafkaConsumer<String, String> consumer = new KafkaConsumer<String,String>(props);
```

订阅主题

订阅一个主题

```
1 consumer.subscribe(Collections.singletonList("customerCountries"));
```

订阅一些主题

```
1 Collection<String> collection = new ArrayList<>();
2 collection.add("topic1");
3 collection.add("topic2");
4 ...
5 consumer.subscribe(collection);
```

订阅所有test主题

```
1 consumer.subscribe("test.*");
```

轮询消费

```
1 try{
2  while(true){
3  //Consumer.poll(long long)方法已被弃用,该方法如果没有获取到数据就会一直阻塞直到获取到数据为止
4  //ConsumerRecords<String, String> records = consumer.poll(100);
5  ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
6  for (ConsumerRecord<String, String> record : records){
7  log.debug("topic = %s, partition = %s, offset = %d,customer = %s, country = %s\n",
8  record.topic(), record.partition(), record.offset(),record.key(), record.value());
9  //获取该条记录
10  String value = record.value()
11  System.out.println(value);
12  }
13  }
14 } finally {
15  //记得最后最后关掉Consumer
16  consumer.close();
17 }
```

其中，代码中标注了几点，说明如下：

- 1）这个例子使用无限循环消费并处理数据，这也是使用Kafka最多的一个场景，后面我们会讨论如何更好的退出循环并关闭。

- 2）这是上面代码中最核心的一行代码。我们不断调用poll拉取数据，如果停止拉取，那么Kafka会认为此消费者已经死亡并进行重平衡。参数值是一个超时时间，指明线程如果没有数据时等待多长时间，0表示不等待立即返回。
- 3）poll()方法返回记录的列表，每条记录包含key/value以及主题、分区、位移信息。
- 4）主动关闭可以使得Kafka立即进行重平衡而不需要等待会话过期。

消费者配置Properties

上面只说了bootstrap.servers，group.id，key.deserializer和value.deserializer，很多情况下只是使用默认设置就行，但了解一些比较重要的参数还是很有帮助的。

fetch.min.bytes

这个参数允许消费者指定从broker读取消息时最小的数据量。当消费者从broker读取消息时，如果数据量小于这个阈值，broker会等待直到有足够的数据，然后才返回给消费者。对于写入量不高的主题来说，这个参数可以减少broker和消费者的压力，因为减少了往返的时间。而对于有大量消费者的主题来说，则可以明显减轻broker压力。

fetch.max.wait.ms

上面的fetch.min.bytes参数指定了消费者读取的最小数据量，而这个参数则指定了消费者读取时最长等待时间，从而避免长时间阻塞。这个参数默认为500ms。

max.partition.fetch.bytes

这个参数指定了每个分区返回的最多字节数，默认为1M。也就是说，KafkaConsumer.poll()返回记录列表时，每个分区的记录字节数最多为1M。如果一个主题有20个分区，同时有5个消费者，那么每个消费者需要4M的空间来处理消息。实际情况中，我们需要设置更多的空间，这样当存在消费者宕机时，其他消费者可以承担更多的分区。

需要注意的是，max.partition.fetch.bytes必须要比broker能够接收的最大的消息（由max.message.size设置）大，否则会导致消费者消费不了消息。另外，在上面的样例可以看到，我们通常循环调用poll方法来读取消息，如果max.partition.fetch.bytes设置过大，那么消费者需要更长的时间来处理，可能会导致没有及时poll而会话过期。对于这种情况，要么减小max.partition.fetch.bytes，要么加长会话时间。

session.timeout.ms

这个参数设置消费者会话过期时间，默认为3秒。也就是说，如果消费者在这段时间内没有发送心跳，那么broker将会认为会话过期而进行分区重平衡。这个参数与

heartbeat.interval.ms有关，heartbeat.interval.ms控制KafkaConsumer的poll()方法多长时间发送一次心跳，这个值需要比session.timeout.ms小，一般为1/3，也就是1秒。更小的session.timeout.ms可以让Kafka快速发现故障进行重平衡，但也加大了误判的概率（比如消费者可能只是处理消息慢了而不是宕机）。

auto.offset.reset

这个参数指定了当消费者第一次读取分区或者上一次的位置太老（比如消费者下线时间太久）时的行为，可以取值为latest（从最新的消息开始消费）或者earliest（从最老的消息开始消费）。

enable.auto.commit

这个参数指定了消费者是否自动提交消费位移，默认为true。如果需要减少重复消费或者数据丢失，你可以设置为false。如果为true，你可能需要关注自动提交的时间间隔，该间隔由auto.commit.interval.ms设置。

partition.assignment.strategy

我们已经知道当消费组存在多个消费者时，主题的分区需要按照一定策略分配给消费者。这个策略由PartitionAssignor类决定，默认有两种策略：

- 范围（Range）：对于每个主题，每个消费者负责一定的连续范围分区。假如消费者C1和消费者C2订阅了两个主题，这两个主题都有3个分区，那么使用这个策略会导致消费者C1负责每个主题的分区0和分区1（下标基于0开始），消费者C2负责分区2。可以看到，如果消费者数量不能整除分区数，那么第一个消费者会多出几个分区（由主题数决定）。
- 轮询（RoundRobin）：对于所有订阅的主题分区，按顺序一一的分配给消费者。用上面的例子来说，消费者C1负责第一个主题的分区0、分区2，以及第二个主题的分区1；其他分区则由消费者C2负责。可以看到，这种策略更加均衡，所有消费者之间的分区数的差值最多为1。

partition.assignment.strategy设置了分配策略，默认为org.apache.kafka.clients.consumer.RangeAssignor（使用范围策略），你可以设置为org.apache.kafka.clients.consumer.RoundRobinAssignor（使用轮询策略），或者自己实现一个分配策略然后将partition.assignment.strategy指向该实现类。

client.id

这个参数可以为任意值，用来指明消息从哪个客户端发出，一般会在打印日志、衡量指标、分配配额时使用。

max.poll.records

这个参数控制一个poll()调用返回的记录数，这个可以用来控制应用在拉取循环中的处理数据量。

receive.buffer.bytes、send.buffer.bytes

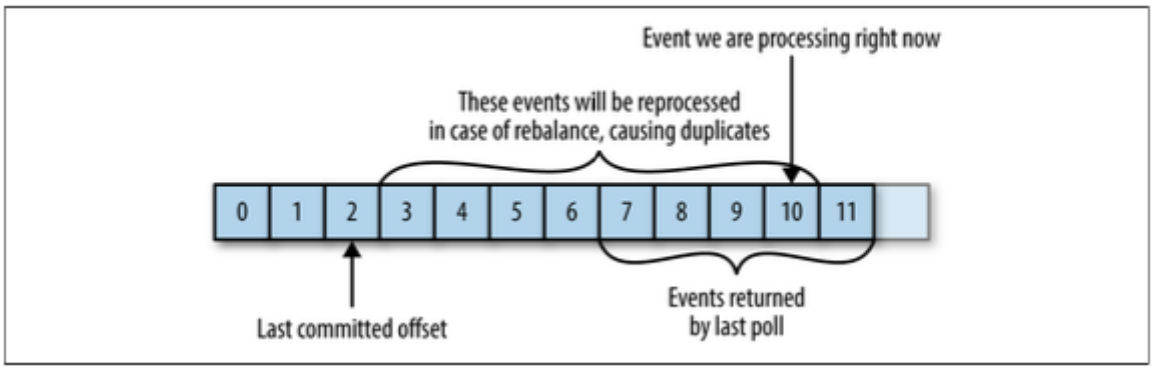
这两个参数控制读写数据时的TCP缓冲区，设置为-1则使用系统的默认值。如果消费者与broker在不同的数据中心，可以一定程度加大缓冲区，因为数据中心间一般的延迟都比较大。

提交（commit）与位移（offset）

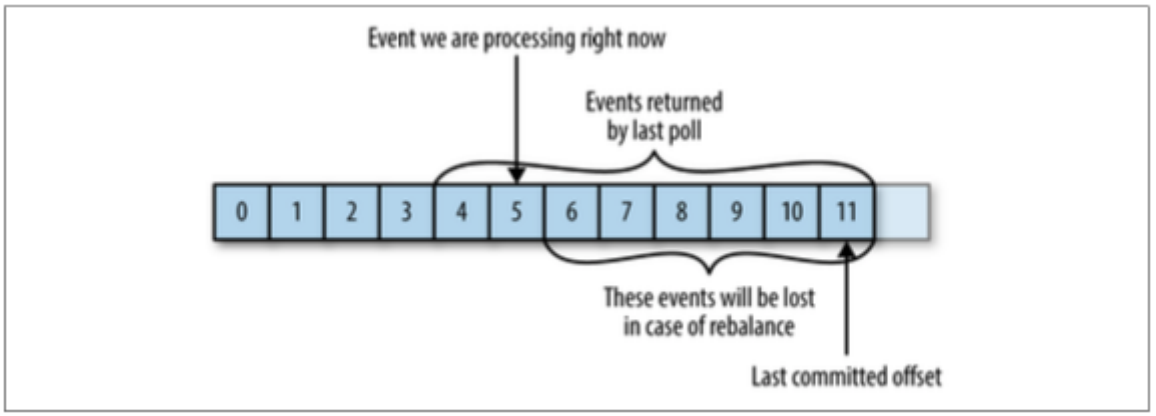
当我们调用poll()时，该方法会返回我们没有消费的消息。当消息从broker返回消费者时，broker并不跟踪这些消息是否被消费者接收到；Kafka让消费者自身来管理消费的位移，并向消费者提供更新位移的接口，这种更新位移方式称为提交（commit）。

在正常情况下，消费者会发送分区的提交信息到Kafka，Kafka进行记录。当消费者宕机或者新消费者加入时，Kafka会进行重平衡，这会导致消费者负责之前并不属于它的分区。重平衡完成后，消费者会重新获取分区的位移，下面来看下两种有意思的情况。

假如一个消费者在重平衡前后都负责某个分区，如果提交位移比之前实际处理的消息位移要小，那么会导致消息重复消费，如下所示：



假如在重平衡前某个消费者拉取分区消息，在进行消息处理前提交了位移，但还没完成处理宕机了，然后Kafka进行重平衡，新的消费者负责此分区并读取提交位移，此时会“丢失”消息，如下所示：



因此，提交位移的方式会对应用有比较大的影响，下面来看下不同的提交方式。

自动提交

这种方式让消费者来管理位移，应用本身不需要显式操作。当我们将enable.auto.commit设置为true，那么消费者会在poll方法调用后每隔5秒（由auto.commit.interval.ms指定）提交一次位移。和很多其他操作一样，自动提交也是由poll()方法来驱动的；在调用poll()时，消费者判断是否到达提交时间，如果是则提交上一次poll返回的最大位移。

需要注意到，这种方式可能会导致消息重复消费。假如，某个消费者poll消息后，应用正在处理消息，在3秒后Kafka进行了重平衡，那么由于没有更新位移导致重平衡后这部分消息重复消费。

提交当前位移

为了减少消息重复消费或者避免消息丢失，很多应用选择自己主动提交位移。设置auto.commit.offset为false，那么应用需要自己通过调用commitSync()来主动提交位移，该方法会提交poll返回的最后位移。

为了避免消息丢失，我们应当在完成业务逻辑后才提交位移。而如果在处理消息时发生了重平衡，那么只有当前poll的消息会重复消费。下面是一个自动提交的代码样例：

```
1  while(true){
2   ConsumerRecords<String, String> records = consumer.poll(Duration.ofMilli
s(100));
3   for (ConsumerRecord<String, String> record : records){
4   //todo 处理消息
5   }
6   try{
7   consumer.commitSync();
8   } catch (CommitFailedException e){
9   //todo 处理提交失败事件
10   }
11  }
```

上面代码poll消息，循环里面进行处理，最后完成处理后进行了位移提交。

异步提交

手动提交有一个缺点，那就是当发起提交调用时应用会阻塞。当然我们可以减少手动提交的频率，但这个会增加消息重复的概率（和自动提交一样）。另外一个解决办法是，使用异步提交的API。以下为使用异步提交的方式，应用发了一个提交请求然后立即返回：

```
while(true){
 ConsumerRecords<String, String> records = consumer.poll(Duration.ofMilli
s(100));
 for (ConsumerRecord<String, String> record : records){
 //todo 处理消息
 }
 consumer.commitAsync();
}
```

但是异步提交也有个缺点，那就是如果服务器返回提交失败，异步提交不会进行重试。相比较起来，同步提交会进行重试直到成功或者最后抛出异常给应用。异步提交没有实现重试是因为，如果同时存在多个异步提交，进行重试可能会导致位移覆盖。举个例子，假如我们发起了一个异步提交commitA，此时的提交位移为2000，随后又发起了一个异步提交commitB且位移为3000；commitA提交失败但commitB提交成功，此时commitA进行重试并成功的话，会将实际上将已经提交的位移从3000回滚到2000，导致消息重复消费。

因此，基于这种性质，一般情况下对于异步提交，我们可能会通过回调的方式记录提交结果：

```
while(true){
 ConsumerRecords<String, String> records = consumer.poll(Duration.ofMilli
s(100));
 for (ConsumerRecord<String, String> record : records){
 //todo 处理消息
 }
 consumer.commitAsync(new OffsetCommitCallback() {
 public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, E
xception exception) {
 if (e != null){
 log.error("Commit failed for offsets {}", offsets, e);
  }
  }
  });
}
```

而如果想进行重试同时又保证提交顺序的话，一种简单的办法是使用单调递增的序号。每次发起异步提交时增加此序号，并且将此时的序号作为参数传给回调方法；当消息提交失败回调时，检查参数中的序号值与全局的序号值，如果相等那么可以进行重试提交，否则放弃（因为已经有更新的位移提交了）。

混合同步提交与异步提交

正常情况下，偶然的提交失败并不是什么大问题，因为后续的提交成功就可以了。但是在某些情况下（例如程序退出、重平衡），我们希望最后的提交成功，因此一种非常普遍的方式是混合异步提交和同步提交，如下所示：

```
try{
 while(true){
 ConsumerRecords<String, String> records = consumer.poll(Duration.ofMilli
s(100));
 for (ConsumerRecord<String, String> record : records){
 //todo 处理消息
 }
 consumer.commitAsync();
 } catch (Exception e) {
 log.error("Unexpected error", e);
  } finally {
  try{
  consumer.commitSync();
  } finally {
  consumer.close();
  }
  }
}
```

在正常处理流程中，我们使用异步提交来提高性能，但最后使用同步提交来保证位移提交成功。

提交特定位移

commitSync()和commitAsync()会提交上一次poll()的最大位移，但如果poll()返回了批量消息，而且消息数量非常多，我们可能会希望在处理这些批量消息过程中提交位移，以免重平衡导致从头开始消费和处理。幸运的是，commitSync()和commitAsync()允许我们指定特定的位移参数，参数为一个分区与位移的map。由于一个消费者可能会消费多个分区，所以这种方式会增加一定的代码复杂度，如下所示：

```
//全局变量
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
...

//方法体
int count = 0;
while (true) {
 ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
 for (ConsumerRecord<String, String> record : records) {
 //todo 处理事件
 //当前offset
 currentOffsets.put(new TopicPartition(record.topic(), record.partition()), new OffsetAndMetadata(record.offset() + 1, "no metadata"));
 if (count % 1000 == 0)
 consumer.commitAsync(currentOffsets, null);
 count++;
 }
}
```

代码中在处理poll()消息的过程中，不断保存分区与位移的关系，每处理1000条消息就会异步提交（也可以使用同步提交）。

重平衡监听器Rebalance Listener

在分区重平衡前，如果消费者知道它即将不再负责某个分区，那么它可能需要将已经处理过的消息位移进行提交。Kafka的API允许我们在消费者新增分区或者失去分区时进行处理，我们只需要在调用subscribe()方法时传入ConsumerRebalanceListener对象，该对象有两个方法：

- public void onPartitionRevoked(Collection partitions)：此方法会在消费者停止消费消费后，在重平衡开始前调用。
- public void onPartitionAssigned(Collection partitions)：此方法在分区分配给消费者后，在消费者开始读取消息前调用。

下面来看一个(onPartitionRevoked)的例子，该例子在消费者失去某个分区时提交位移（以便其他消费者可以接着消费消息并处理）：

```
public interface HandleRebalance extends ConsumerRebalanceListener {
}
```

```
//全局变量
```

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
...

//订阅时传入HandleRebalance
consumer.subscribe(Collections.singletonList(topic), new HandleRebalance() {
 /**
 * 此方法会在消费者停止消费消费后，在重平衡开始前调用。
 * @param collection
 */
 @Override
 public void onPartitionsRevoked(Collection<TopicPartition> collection) {
 //todo 提交处理结果,保存分区和位移

 }

 /**
 * 此方法在分区分配给消费者后，在消费者开始读取消息前调用。
 * @param collection
 */
 @Override
 public void onPartitionsAssigned(Collection<TopicPartition> collection) {
 //在开始消费前，从数据库中获取分区的位移，并使用seek()来指定开始消费的位移
 }
});

...
//轮询
try{
 while (true) {
 ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
 for (ConsumerRecord<String, String> record : records) {
 //todo 处理事件
 //当前offset
 currentOffsets.put(new TopicPartition(record.topic(), record.partition()), new OffsetAndMetadata(record.offset() + 1, "no metadata"));
 }
```

```
37   consumer.commitAsync(currentOffsets, null);
38   }
39  } finally {
40   try {
41   consumer.commitSync(currentOffsets);
42   } finally{
43   consumer.close();
44   }
45  }
```

代码中实现了onPartitionsRevoked()方法，当消费者失去某个分区时，会提交已经处理的消息位移（而不是poll()的最大位移）。上面代码会提交所有的分区位移，而不仅仅是失去分区的位移，但这种做法没什么坏处。

从指定位移开始消费

在此之前，我们使用poll()来从最后的提交位移开始消费，但我们也可以从一个指定的位移开始消费。

如果想从分区开始端重新开始消费，那么可以使用seekToBeginning(TopicPartition tp)；如果想从分区的最末端消费最新的消息，那么可以使用seekToEnd(TopicPartition tp)。而且，Kafka还支持我们从指定位移开始消费。从指定位移开始消费的应用场景有很多，其中最典型的一个是：位移存在其他系统（例如数据库）中，并且以其他系统的位移为准。

考虑这么个场景：我们从Kafka中读取消费，然后进行处理，最后把结果写入数据库；我们既不想丢失消息，也不想数据库中存在重复的消息数据。对于这样的场景，我们可能会按如下逻辑处理：

```
1  while (true) {
2   ConsumerRecords<String, String> records = consumer.poll(100);
3   for (ConsumerRecord<String, String> record : records) {
4   currentOffsets.put(new TopicPartition(record.topic(),
record.partition()), record.offset());
5   //todo save DB
6   consumer.commitAsync(currentOffsets);
7   }
8  }
```

这个逻辑似乎没什么问题，但是要注意到这么个事实，在持久化到数据库成功后，提交位移到Kafka可能会失败，那么这可能会导致消息会重复处理。对于这种情况，我们可以优化方案，将持久化到数据库与提交位移实现为原子性操作，也就是要么同时成功，要么同时失败。但这个是不可能的，因此我们可以在保存记录到数据库的同时，也保存位移，然后在消

费者开始消费时使用数据库的位移开始消费。这个方案是可行的，我们只需要通过seek()来指定分区位移开始消费即可。下面是一个改进的样例代码：

```
public interface HandleRebalance extends ConsumerRebalanceListener {
}
```

```
//全局变量
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
...

//订阅传入重平衡监听器
consumer.subscribe(Collections.singletonList(topic), new HandleRebalance() {
 /**
 * 此方法会在消费者停止消费消费后，在重平衡开始前调用。
 * @param collection
 */
 @Override
 public void onPartitionsRevoked(Collection<TopicPartition> collection) {
 //在消费者负责的分区被回收前提交数据库事务，保存消费的记录和位移
 for (TopicPartition partition : collection) {
 currentOffsets.put(partition, new OffsetAndMetadata(currentOffsets.get(partition).offset() + 1, "no metadata"));
 }
 }

 /**
 * 此方法在分区分配给消费者后，在消费者开始读取消息前调用。
 * @param collection
 */
 @Override
 public void onPartitionsAssigned(Collection<TopicPartition> collection) {
 //在开始消费前，从数据库中获取分区的位移，并使用seek()来指定开始消费的位移
 for (TopicPartition partition : collection) {
 consumer.seek(partition, currentOffsets.get(partition).offset());
 }
 }
});
```

```
//在订阅完了立马进行一次poll,获取分区位移，使用seek()指定开始消费的位移
consumer.poll(Duration.ofMillis(0));
for (Object partition : consumer.assignment()){
 TopicPartition partitionObject = (TopicPartition) partition;
 consumer.seek(partitionObject,currentOffsets.get(partitionObject).offset());
}
...
try{
 while(true){
 ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
 records.forEach(record -> {
 //todo 处理业务
 //当前offset
 currentOffsets.put(new TopicPartition(record.topic(), record.partition()), new OffsetAndMetadata(record.offset() + 1, "no metadata"));
 });
 }
}
```

seek()只是指定了poll()拉取的开始位移，这并不影响在Kafka中保存的提交位移（当然我们可以在seek和poll之后提交位移覆盖）。

优雅退出

在一般情况下，我们会在一个主线程中循环poll消息并进行处理。当需要退出poll循环时，我们可以使用另一个线程调用consumer.wakeup()，调用此方法会使得poll()抛出WakeupException。如果调用wakup时，主线程正在处理消息，那么在下一次主线程调用poll时会抛出异常。主线程在抛出WakeUpException后，需要调用consumer.close()，此方法会提交位移，同时发送一个退出消费组的消息到Kafka的组协调者。组协调者收到消息后会立即进行重平衡（而无需等待此消费者会话过期）。

优雅退出样例：

```
//注册JVM关闭时的回调钩子，当JVM关闭时调用此钩子。
Runtime.getRuntime().addShutdownHook(new Thread() {
 public void run() {
 System.out.println("Starting exit...");
 //调用消费者的wakeup方法通知主线程退出
 consumer.wakeup();
 try {
```

```
8   //等待主线程退出
9   mainThread.join();
10   } catch (InterruptedException e) {
11   e.printStackTrace();
12   }
13   }
14  });
15
16  ...
17
18  try {
19   while (true) {
20   ConsumerRecords<String, String> records = consumer.poll(1000);
21   for (ConsumerRecord<String, String> record : records) {
22   System.out.printf("offset = %d, key = %s, value =
%s\n",record.offset(), record.key(), record.value());
23   }
24   for (TopicPartition tp: consumer.assignment())
25   System.out.println("Committing offset at position:" +
consumer.position(tp));
26   consumer.commitSync();
27   }
28   } catch (WakeupException e) {
29   //todo 通知异常处理
30   } finally {
31   consumer.close();
32   System.out.println("Closed consumer and we are done");
33  }
```

单个消费者

一般情况下我们都是使用消费组（即便只有一个消费者）来消费消息的，因为这样可以在增加或减少消费者时自动进行分区重平衡。这种方式是推荐的方式。在知道主题和分区的情况下，我们也可以使用单个消费者来进行消费。对于这种情况，我们需要自己给消费者分配消费分区，而不是让消费者订阅（成为消费组）主题。

单个消费者指定分区进行消费的代码样例：

```
1  List<PartitionInfo> partitionInfos = null;
2  //获取主题下所有的分区。如果你知道所指定的分区，可以跳过这一步
```

```
partitionInfos = consumer.partitionsFor(topic);
if (partitionInfos != null) {
 for (PartitionInfo partition : partitionInfos){
 partitions.add(new TopicPartition(partition.topic(),
partition.partition()));
 //为消费者指定分区
 consumer.assign(partitions);
 }
}
...

while (true) {
 ConsumerRecords<String, String> records = consumer.poll(1000);
 for (ConsumerRecord<String, String> record: records) {
 //todo
 }
 consumer.commitSync();
}
```

除了需要主动获取分区以及没有分区重平衡，其他的处理逻辑都是一样的。需要注意的是，如果添加了新的分区，这个消费者是感知不到的，需要通过consumer.partitionsFor()来重新获取分区。