本文针对java api 1.1.0 翻译理解得来的，就是下面这个地址。

<http://kafka.apache.org/11/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html>

如果不清楚的地方还得，参考英文原版。

**兼容性**

可以以前的broker，不如前面的版本没有对应的功能，会报 UnsupportedVersionException。有的功能都能正常访问。

**offsets和consumer position**

kafka会在partition中为每个record maintain 一个offset。在partition内部，offset的唯一标志。也可用来表示consumer在partition中的position。

consumer的position是下一次获取message的offset

**consumer group 和 主题订阅**

each partition is assigned to exactly one consumer in the group

any number of consumer groups for a given topic without duplicating data 多个consumer group 可以对应一个topic

一个topic的消息被分配到多个partition上，一个partition对应一个consumer。

一个topic对应一个consumer group 他们是多对多的关系。

也就是一个consumer可能消费多个topic的消息， 一个consumer对应多个topic下的一个partition。

存在不同的topic共享partition么？

consumer端怎么不同topic的数据分开？

**检查消费端失败**

不停地发心跳

the consumer sends periodic heartbeats to the server. If the consumer crashes or is unable to send heartbeats for a duration of session.timeout.ms, then the consumer will be considered dead and its partitions will be reassigned.

livelock 客户端有问题，但是 send heartbeats 为了防止客户端无限hold on partition

liveness detction mechanism max.poll.interval.ms

Basically if you don't call poll at least as frequently as the configured max interval, then the client will proactively leave the group so that another consumer can take over its partitions.

不活跃的consumer是不能 commit offset的

When this happens, you may see an offset commit failure (as indicated by a CommitFailedException thrown from a call to commitSync()).

This is a safety mechanism which guarantees that only active members of the group are able to commit offsets. So to stay in the group, you must continue to call poll.

max.poll.interval.ms 增加值的话， 可以增加consumer处理批量消息的时间，缺点是延迟组再平衡，因为consumer要在调用poll方法的时候才会rebalance，

max.poll.records 限制调用一次poll方法返回record数量的最大值， This can make it easier to predict the maximum that must be handled within each poll interval.

By tuning this value, you may be able to reduce the poll interval, which will reduce the impact of group rebalancing

For use cases where message processing time varies unpredictably, neither of these options may be sufficient. The recommended way to handle these cases is to move message processing to another thread,

which allows the consumer to continue calling poll while the processor is still working. Some care must be taken to ensure that committed offsets do not get ahead of the actual position.

Typically, you must disable automatic commits and manually commit processed offsets for records only after the thread has finished handling them (depending on the delivery semantics you need).

Note also that you will need to pause the partition so that no new records are received from poll until after thread has finished handling those previously returned.

**自动提交offset**

**手动提交offset**

手动提交有2中方法：

commitSync and commitAsync

可以将kafka的offset 在消费端应用自己保存

kafka消费者应用可以不必使用kafka服务端内置的offset存储

when it is it will make the consumption fully atomic and give "exactly once" semantics that are stronger than the default "at-least once" semantics you get with Kafka's offset commit functionality.

批量操作的时候，不能自动提交，因为有可能在处理一批数据的时候，出现了错误，会导致这批数据丢失。

props.put("enable.auto.commit", "false");关掉自动提交，当处理完成后才提交offset，consumer.commitSync();

也可能数据处理成功了，然后提交offsets的时候出错了。这会导致重复消费最后一批数据

这就是kafka提供的 "at-least-once" 递送保证。每条消息至少递送一次，出错的情况下，会重复的递送。

Using automatic offset commits can also give you "at-least-once" delivery, but the requirement is that you must consume all data returned from each call to poll(long) before any subsequent calls,

or before closing the consumer. If you fail to do either of these, it is possible for the committed offset to get ahead of the consumed position, which results in missing records.

The advantage of using manual offset control is that you have direct control over when a record is considered "consumed."

commitSync to mark all received records as committed. commitSync 是提交所有的offsets

如果想提交指定的offset，可以分片partition 控制提交offsets

|  |
| --- |
| try {  while(running) {  ConsumerRecords<String, String> records = consumer.poll(Long.MAX\_VALUE);  for (TopicPartition partition : records.partitions()) {  List<ConsumerRecord<String, String>> partitionRecords = records.records(partition);  for (ConsumerRecord<String, String> record : partitionRecords) {  System.out.println(record.offset() + ": " + record.value());  }  long lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();  consumer.commitSync(Collections.singletonMap(partition, new OffsetAndMetadata(lastOffset + 1)));  }  }  } finally {  consumer.close();  } |

The committed offset should always be the offset of the next message that your application will read. 提交的offset应该是下次消费的消息的offset

**手动分配分片**

当用这种方式consumer.subscribe(Arrays.asList(topic)); kafka会动态的将分片partition均衡的分配给group下面的活跃的consumer

String topic = "foo";

TopicPartition partition0 = new TopicPartition(topic, 0);

TopicPartition partition1 = new TopicPartition(topic, 1);

consumer.assign(Arrays.asList(partition0, partition1)

手动分配分片没有使用group 协调，所以一个消费者失败了不会引起partitions重新分配。每个consumer独立运行，尽管他们在一个groupId下面。为了避免offset提交冲突，应该确保一个groupId只有一个consumer实例

手动分配分片和自动分配分片不可能一起使用，在一个主题下。

在kafka外面存储offsets——就是存在应用里面

consumer不必使用内置的offset存储，offsets和results存在一起，在某种程度上可以保证原子性。

1、比如消费的结果存入关系型数据库，offset也存入数据库。可以保证事务性。要么事务成功：消息消费成功且offset也更新；要么事务失败：结果不会被存储，offset也不会更新。

2、result和offset会被存在一起，在某种程度上保证原子性。即使一个crash导致未同步的data丢失，offset同样没有存储。相当于回滚到了消费之前的状态

Configure enable.auto.commit=false

Use the offset provided with each ConsumerRecord to save your position.

On restart restore the position of the consumer using seek(TopicPartition, long).

分片partition是手动分配的，用法就很简单。

如果分片是自动分配，需要特别注意处理分片改变的情况。可以通过ConsumerRebalanceListener来实现，

调用下面两个方法就好了。

subscribe(Collection, ConsumerRebalanceListener)

subscribe(Pattern, ConsumerRebalanceListener)

提交offset用 ConsumerRebalanceListener.onPartitionsRevoked(Collection)

搜索offset来初始化consumer用 ConsumerRebalanceListener.onPartitionsAssigned(Collection)

ConsumerRebalanceListener.onPartitionsRevoked(Collection)

If the partition assignment is done automatically special care is needed to handle the case where partition assignments change.

when partitions are taken from a consumer ，the consumer will want to commit its offset for those partitions by implementing ConsumerRebalanceListener.onPartitionsRevoked(Collection)

When partitions are assigned to a consumer, the consumer will want to look up the offset for those new partitions and correctly initialize the consumer to that position

by implementing ConsumerRebalanceListener.onPartitionsAssigned(Collection).

**控制consumer的位置**

seek(TopicPartition, long) 指定offset

seekToBeginning(Collection) and seekToEnd(Collection) 查找服务器上保留的最早的或者最晚的offset

**消费流控制**

一个消费者同时分配了多个分片，消费者想先全速消费部分分片，然后才消费其他数据量少或者没有数据的分片。

kafka 支持动态控制流量

pause(Collection) 暂停分配的部分分片消费

resume(Collection) 回复指定的暂停状态的分片消费

然后再调用 poll(long) 来开始消费。

**事务的读取消息**

kafka 0.11.0 引入了trasanctions

isolation.level=read\_committed 配置kafka消费者读取已提交的数据。这个有点像关系型数据库的隔离级别

在read\_committed模式下，consumer会读取事务成功提交的消息。非事务消息还是像以前一样读取。 在read\_committed模式下，没有客户端buffer。

LSO —— Last Stable Offset 对于read\_committed的consumer来说，分片的最后一个offset就是open transaction中的第一条消息的offset

有事务消息的分片，会记录事务提交和abort的标志。标志不会返回给客户端，但是会占用一个offset保存在log中

消费者从有事务消息的topics读取消息，可以看到gap在消费的offset中，这些缺失的offset可能就是事务标志

gap 可能是aborted transaction

**多线程处理**

kafka客户端不是线程安全的。保证多线程适当的synchronized是用户的责任。不同步会导致ConcurrentModificationException

wakeup() 是唯异常，他可以在外部线程中安全的调用来打断活跃的操作。被打断操作的线程块会跑出 WakeupException 。这个可以用于在另一个线程中 shutdown consumer

public class KafkaConsumerRunner implements Runnable {

private final AtomicBoolean closed = new AtomicBoolean(false);

private final KafkaConsumer consumer;

public void run() {

try {

consumer.subscribe(Arrays.asList("topic"));

while (!closed.get()) {

ConsumerRecords records = consumer.poll(10000);

// Handle new records

}

} catch (WakeupException e) {

// Ignore exception if closing

if (!closed.get()) throw e;

} finally {

consumer.close();

}

}

// Shutdown hook which can be called from a separate thread

public void shutdown() {

closed.set(true);

consumer.wakeup();

}

}

不建议使用thread interrupt 来放弃一个阻塞的操作 cause a clean shutdown of the consumer to be aborted，就是shutdown不会成功。

每个线程一个consumer

优点：

1、很容易实现

2、速度最快，因为没有内部线程协调

3、使顺序处理每个分片上的数据非常容易实现

缺点：

1、更多的consumer，就有更多的tcp连接。不过kafka处理连接非常的高效，所以通常来说消耗还是很小的。

2、多个consumer意味着更多的请求发往kafka server，稍微小一点的数据批可能会引起 i/o 卡顿。

3、处理程序线程的总数会受到partition的限制。

解耦线程和consumer

优点：

1、允许处理程序和consumer独立扩展，可能一个consummer对应多个处理线程，这样可以避免partition的限制。（一个consumer和partition一一对应）

缺点

1、保证处理程序的顺序需要特别的注意，因为线程可能会后处理先获取到的数据，这取决于线程执行的时机。不过对于没有顺序要求的是没有关系的。

2、手动提交postition（offset）变得更困难，因为需要所有的线程协调来保证一个partition所有处理都完成