

kafka

zqh

Published
with GitBook



目錄

Introduction	0
kafka-Intro	1
kafka-Unix	2
kafka-Producer	3
kafka-Producer-Scala	4
kafka-SocketServer	5
kafka-LogAppend	6
kafka-ISR	7
kafka-Consumer-init-Scala	8

Kafka源码分析

```
val version = 0.9
val topic = "kafka-code"

def readKafkaCode() {
    while(true) {
        println("keep reading...")
    }
}
```

Background

MQ的通讯模式

Mode	Feature
点对点通讯	点对点方式是最为传统和常见的通讯方式，它支持一对一、一对多、多对多、多对一等多种配置方式，支持树状、网状等多种拓扑结构。
多点广播	MQ 适用于不同类型的应用。其中重要的，也是正在发展中的是"多点广播"应用，即能够将消息发送到多个目标站点 (Destination List)。可以使用一条 MQ 指令将单一消息发送到多个目标站点，并确保为每一站点可靠地提供信息。MQ 不仅提供了多点广播的功能，而且还拥有智能消息分发功能，在将一条消息发送到同一系统上的多个用户时，MQ 将消息的一个复制版本和该系统上接收者的名单发送到目标 MQ 系统。目标 MQ 系统在本地复制这些消息，并将它们发送到名单上的队列，从而尽可能减少网络的传输量。
发布/订阅 (Publish/Subscribe) 模式	发布/订阅功能使消息的分发可以突破目的队列地理指向的限制，使消息按照特定的主题甚至内容进行分发，用户或应用程序可以根据主题或内容接收到所需要的消息。发布/订阅功能使得发送者和接收者之间的耦合关系变得更为松散，发送者不必关心接收者的地址，而接收者也不必关心消息的发送地址，而只是根据消息的主题进行消息的收发。
群集 (Cluster)	为了简化点对点通讯模式中的系统配置，MQ 提供 Cluster(群集) 的解决方案。群集类似于一个域 (Domain)，群集内部的队列管理器之间通讯时，不需要两两之间建立消息通道，而是采用群集 (Cluster) 通道与其它成员通讯，从而大大简化了系统配置。此外，群集中的队列管理器之间能够自动进行负载均衡，当某一队列管理器出现故障时，其它队列管理器可以接管它的工作，从而大大提高系统的高可靠性。

关键概念

Concepts	Function
Topic	用于划分Message的逻辑概念，一个Topic可以分布在多个Broker上。
Partition	是Kafka中横向扩展和一切并行化的基础，每个Topic都至少被切分为1个Partition。
Offset	消息在Partition中的编号，编号顺序不跨Partition(在Partition内有序)。
Consumer	用于从Broker中取出/消费 Message。
Producer	用于往Broker中发送/生产 Message。
Replication	Kafka支持以Partition为单位对Message进行冗余备份，每个Partition都可以配置至少1个Replication(当仅1个Replication时即仅该Partition本身)。
Leader	每个Replication集合中的Partition都会选出一个唯一的Leader，所有的读写请求都由Leader处理。其他Replicas从Leader处把数据更新同步到本地。
Broker	Kafka中使用Broker来接受Producer和Consumer的请求，并把Message持久化到本地磁盘。每个Cluster当中会选举出一个Broker来担任Controller，负责处理Partition的Leader选举，协调Partition迁移等工作。
ISR	In-Sync Replica,是Replicas的一个子集，表示目前Alive且与Leader能够“Catch-up”的Replicas集合。由于读写都是首先落到Leader上，所以一般来说通过同步机制从Leader上拉取数据的Replica都会和Leader有一些延迟(包括了延迟时间和延迟条数两个维度)，任意一个超过阈值都会把该Replica踢出ISR。每个Leader Partition都有它自己独立的ISR。

设计思想

Concepts	Function
Consumergroup	各个consumer可以组成一个组，每个消息只能被组中的一个consumer消费，如果一个消息可以被多个consumer消费的话，那么这些consumer必须在不同的组。
消息状态	在Kafka中，消息的状态被保存在consumer中，broker不会关心哪个消息被消费了被谁消费了，只记录一个offset值（指向partition中下一个要被消费的消息位置），这意味着如果consumer处理不好的话，broker上的一个消息可能会被消费多次。
消息持久化	Kafka中会把消息持久化到本地文件系统中，并且保持极高的效率。
消息有效期	Kafka会长久保留其中的消息，以便consumer可以多次消费，当然其中很多细节是可配置的。
批量发送	Kafka支持以消息集合为单位进行批量发送，以提高push效率。
push-and-pull	Kafka中的Producer和consumer采用的是push-and-pull模式，即Producer只管向broker push消息，consumer只管从broker pull消息，两者对消息的生产和消费是异步的。
Broker之间的关系	不是主从关系，各个broker在集群中地位一样，我们可以随意的增加或删除任何一个broker节点。
负载均衡	Kafka提供了一个 metadata API来管理broker之间的负载（对Kafka0.8.x而言，对于0.7.x主要靠zookeeper来实现负载均衡）。
同步异步	Producer采用异步push方式，极大提高Kafka系统的吞吐率（可以通过参数控制是采用同步还是异步方式）。
分区机制 partition	Kafka的broker端支持消息分区，Producer可以决定把消息发到哪个分区，在一个分区中消息的顺序就是Producer发送消息的顺序，一个主题中可以有多个分区，具体分区的数量是可配置的。分区的意义很重大，后面的内容会逐渐体现。

Kafka Replication

The leader maintains a set of in-sync replicas (ISR): the set of replicas that have fully caught up with the leader.

For each partition, we store in Zookeeper the current leader and the current ISR.

Leader维护了ISR(能完全赶上Leader的副本集).每个Partition当前的Leader和ISR信息会记录在ZooKeeper中.

问题:为什么是由Leader来维护ISR?

背景:Leader会跟踪与其保持同步的Replica列表,该列表称为ISR。如果一个Follower宕机,或者落后太多,Leader将把它从ISR中移除.

答案:只有Leader才能知道哪些Replica能够及时完全赶上.所有Follower都会和Leader通信获取最新的消息.

但是Follower之间并不互相知道彼此的信息.所以由Leader来管理ISR最合适了.Leader还可以决定移除落后太多的Replicas.

Each replica stores messages in a local log and maintains a few important offset positions in the log.

The log end offset (LEO) represents the tail of the log.

The high watermark (HW) is the offset of the last committed message.

每个Replica都在自己的local log中存储消息,并在日志中维护了重要的offset位置信息.

LEO代表了日志的最新的偏移量,HW是最近提交消息的偏移量(HW也是每个Replica都有的吗?).

Each log is periodically synced to disks. Data before the flushed offset is guaranteed to be persisted on disks.

As we will see, the flush offset can be before or after HW.

每个日志都会定时地同步到磁盘.在flushed offset之前的数据一定能保存成功持久化到磁盘上.

flush offset可以在HW之前或者之后(因为follower只是先写到内存中然后返回ack给leader,hw增加时,

follower在内存中的消息不一定什么时候写到磁盘上,即可能在hw增加前就写到磁盘,或者等hw增加后才写到磁盘).

Writes

To publish a message to a partition, the client first finds the leader of the partition from

Zookeeper and sends the message to the leader .The leader writes the message to its local log .

Each follower constantly pulls new messages from the leader using a single

socket channel.

That way, the follower receives all messages in the same order as written in the leader.

为了将消息发布给一个Partition,客户端会从ZK中先找到这个Partition的Leader,然后把消息发送给Leader.

Leader会将消息写到自己的本地日志文件中,(Partition的)每个follower会从Leader一直拉取数据.

通过这种方式,follower接收的所有消息的顺序一定和写到leader的消息是同样的顺序.

The follower writes each received message to its own log and sends an acknowledgment back to the leader.

Once the leader receives the acknowledgment from all replicas in ISR , the message is committed .

The leader advances the HW and sends an acknowledgment to the client.

follower收到的每条消息都会写到自己的日志中,并且发送ack给leader.

一旦leader接收到在ISR中所有副本的ack,这条消息就会被提交.然后Leader会增加HW,并发送ack给客户端.

For better performance, each follower sends an acknowledgment after the message is written to memory .

So, for each committed message, we guarantee that the message is stored in multiple replicas in memory.

However, there is no guarantee that any replica has persisted the commit message to disks though.

Given that correlated failures are relatively rare, this approach gives us a good balance between response time and durability.

In the future, we may consider adding options that provide even stronger guarantees.

为了性能考虑,每个follower当消息被写到内存时就发送ack(而不是要完全地刷写到磁盘上才ack).

所以对于每条提交的消息,我们能够保证的是这条消息一定是存储在多个副本(所有ISR)的内存中.

但是并不保证任何副本已经把这条提交的消息持久化到磁盘中.这是基于响应时间和持久性两者平衡的.

The leader also periodically broadcasts the HW to all followers.

The broadcasting can be piggybacked(背负) on the return value of the fetch requests from the followers.

From time to time, each replica checkpoints its HW to its disk .

Leader也会定时地将HW广播给所有的followers. 广播消息可以附加在从follower过来的fetch请求的结果中.

同时,每个副本(不管是leader还是follower)也会定时地将HW持久化到自己的磁盘上.

当follower向leader提交fetch请求时,leader也会告诉所有的follower说,我现在的hw是多少了.这是一种保护机制.

假设只有leader一个人保护了hw这个重要的信息,一旦leader不幸挂掉了,就没有人知道hw现在到底是多少了.

所以只要一有follower过来获取消息时,leader就不厌其烦地像个老太婆不断地唠叨说我这一次的hw更新到了哪里.

每个follower也就都会知道leader的最新hw.这样即使leader挂掉了,hw仍然在其他follower上都备份有这个重要信息.

几个follower在一阵商量后,选举出了新的leader,这些人都知道上一个leader最新的hw,因此hw这个香火会继续传承下去.

Reads

For simplicity, reads are always served from the leader. Only messages up to the HW are exposed to the reader.

为了简单起见,只有leader可以提供读消息的服务.并且最多只到hw位置的消息才会暴露给客户端.

Producer在发布消息到某个Partition时, 先通过Zookeeper找到该Partition的Leader,

然后无论该Topic的Replication Factor为多少 (也即该Partition有多少个Replica) , Producer只将该消息发送到该Partition的Leader。 Leader会将该消息写入其本地Log。 每个Follower都从Leader pull数据。

这种方式上, Follower存储的数据顺序与Leader保持一致。 Follower在收到该消息并写入其Log后, 向Leader发送ACK。

一旦Leader收到了ISR中的所有Replica的ACK, 该消息就被认为已经commit了,

Leader将增加HW并且向Producer发送ACK。

为了提高性能，每个Follower在接收到数据后就立马向Leader发送ACK，而非等到数据写入Log中。

因此，对于已经commit的消息，Kafka只能保证它被存于多个Replica的内存中，而不能保证它们被持久化到磁盘中，

也就不能完全保证异常发生后该条消息一定能被Consumer消费。但考虑到这种场景非常少见，

可以认为这种方式在性能和数据持久化上做了一个比较好的平衡。在将来的版本中，Kafka会考虑提供更高的持久性。

Consumer读消息也是从Leader读取，只有被commit过的消息（offset低于HW的消息）才会暴露给Consumer。

Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制。事实上，同步复制要求所有能工作的Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率。

而异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，

这种情况下如果Follower都复制完都落后于Leader，而如果Leader突然宕机，则会丢失数据。

而Kafka的这种使用ISR的方式则很好的均衡了确保数据不丢失以及吞吐率。

Follower可以批量的从Leader复制数据，这样极大的提高复制性能（批量写磁盘），极大减少了Follower与Leader的差距

Checkpoint用在Follower failure是怎么解决HW的同步问题：

After a configured timeout period, the leader will drop the failed follower from its ISR

and writes will continue on the remaining replicas in ISR.

如果Follower失败了，在超过一定时间后，Leader会将这个失败的follower从ISR中移除(follower没有发送fetch请求)

由于ISR保存的是所有全部赶得上Leader的follower replicas，失败的follower肯定是赶不上了。

虽然ISR现在少了一个，但是并不会引起的数据的丢失，ISR中剩余的replicas会继续同步数据(只要ISR中有一个follower，就不会丢失数据)

(注意：这里讨论的是一个Partition的follower副本，而不是节点，如果是一个节点，它不止存储一个Partition，而且不都是follower)

If the failed follower comes back, it first truncates its log to the last checkpointed HW.

It then starts to catch up all messages after its HW from the leader.

When the follower fully catches up, the leader will add it back to the current ISR.

如果失败的follower恢复过来,它首先将自己的日志截断到上次checkpointed时刻的HW.

因为checkpoint记录的是所有Partition的hw offset. 当follower失败时,checkpoint中关于这个Partition的HW就不会再更新了.

而这个时候存储的HW信息和follower partition replica的offset并不一定是一致的. 比如这个follower获取消息比较快,

但是ISR中有其他follower复制消息比较慢,这样Leader并不会很快地更新HW,这个快的follower的hw也不会更新(leader广播hw给follower)

这种情况下,这个follower日志的offset是比hw要大的.所以在它恢复之后,要将比hw多的部分截掉,然后继续从leader拉取消息(跟平时一样).

实际上,ISR中的每个follower日志的offset一定是比hw大的.因为只有ISR中所有follower都复制完消息,leader才会增加hw.

也就是说有可能有些follower复制完了,而有些follower还没有复制完,那么hw是不会增加的,复制完的follower的offset就比hw要大.

KafkaProducer and KafkaConsumer

KafkaProducer

KafkaConsumer

Consumer API

High Level Consumer

high-level方式的消费不用关心offset,它会自动的读ZK中这个Consumer Group的last offset(最近读取过的消息).

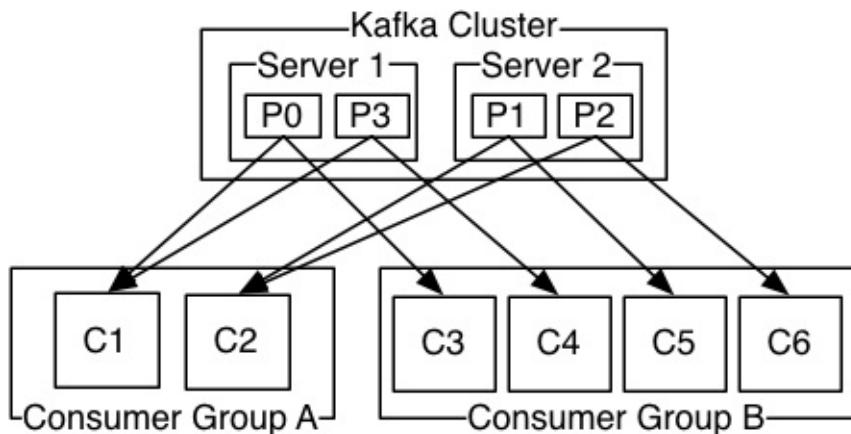
因为消费者在读取消息后,会将最近读取Partition的offset写到ZK中.写到ZK中的offset名称是以ConsumerGroup.

一个消费者组可以有多个消费者,Kafka中的一个Partition只会被消费者组中的一个消费者消费,但可以被多个消费组同时消费.

下图Server1的Partition0(P0),会被消费组A和消费组B同时消费,但是消费组A中只能有一个消费者读取比如C1(消费组B也只有C3读取).

通过这种方式,可以将一个Broker上的多个Partition负载到不同的消费者(同一个消费组的多个消费者).

比如Broker1上有两个Partition(P0,P3), 对于消费组B, P0分给C3,P3分给C4. 但是对于消费组A,都分给了C1.



对于多个partition和多个consumer有以下这样的限制条件:

- 如果consumer比partition多, 是浪费, 因为kafka的设计是在一个partition上是不允许并发的, 所以consumer数不要大于partition数
- 如果consumer比partition少, 一个consumer会对应于多个partitions, 这里主要合理分配consumer数和partition数, 否则会导致partition里面的数据被取的不均匀. 最好partiton数目是consumer数目的整数倍, 所以partition数目很重要, 比如取24, 就很容易设定consumer数目
- 如果consumer从多个partition读到数据, 不保证数据间的顺序性, kafka只保证在一个partition上数据是有序的, 但多个partition, 根据你读的顺序会有不同
- 增减consumer, broker, partition会导致rebalance, 所以rebalance后 consumer对应的partition会发生变化
- High-level接口中获取不到数据的时候是会block住消费者线程的

```

String topic = "page_visits";
String group = "pv";

Properties props = new Properties();
props.put("group.id", group);
props.put("auto.offset.reset", "smallest"); // 必须要加, 如果要读最新的消息
props.put("zookeeper.connect", "localhost:2181");
props.put("zookeeper.session.timeout.ms", "400");
props.put("zookeeper.sync.time.ms", "200");
props.put("auto.commit.interval.ms", "1000");

// 创建消费者到Kafka集群的连接(主要是通过ZooKeeper, 因为ZK中不仅记录了topic和线程的对应关系)
ConsumerConfig conf = new ConsumerConfig(props);
ConsumerConnector consumer = kafka.consumer.Consumer.createJavaConsumerConnector(conf);

// topic和线程的对应关系, 这里放了一个topic, 并且用一个线程来消费
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put(topic, new Integer(1));

// 消费者指定了要消费的topics, 返回所有topics的消息流. 返回的key是topic
Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.listTopics();

// 获取这个topic的Kafka消息流, 一个topic是有多个消息流的, 因为会从多个partition中读取
List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);
KafkaStream<byte[], byte[]> stream = streams.get(0);

// 读取消息, 使用KafkaStream, 获得流的迭代器, 包含了消息的key和message
ConsumerIterator<byte[], byte[]> it = stream.iterator();
while (it.hasNext()){
    System.out.println("message: " + new String(it.next().message));
}
// 其实执行不到, 因为对于high level, 没有数据的话, 上面的hasNext会block
if (consumer != null) consumer.shutdown();

```

Simple Consumer

New Consumer

<http://kafka.apache.org/090/javadoc/>

```
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(p  
consumer.subscribe(Collections.singletonList(this.topic));  
ConsumerRecords<Integer, String> records = consumer.poll(1000);  
for (ConsumerRecord<Integer, String> record : records) {  
    System.out.println("Received message: (" + record.key() + "  
}
```



Ref

- <http://www.ibm.com/developerworks/cn/opensource/os-cn-kafka/index.html>
- <http://bbs.umeng.com/thread-12086-1-1.html>
- <http://www.aboutyun.com/forum.php?mod=viewthread&tid=15812>

原文:<http://logallthethings.com/2015/09/15/kafka-by-example-kafka-as-unix-pipes/>

什么是Kafka

想象下有这样的对话.

你: 什么是Apache Kafka?

我: Apache Kafka是发布-订阅消息系统,分布式的提交日志

你: ...什么?

我: 是的,它是一个分布式的,分区的,复制的提交日志服务

你: 你到底在说什么?

上面的描述(我)是正确的. 你只需要知道这些术语是什么意思, 但是如果你不知道这些术语,就会感到很困惑.

那就让我们以另外一种方式来解释吧. 我喜欢通过例子来学习, 并且在学习的时候通过和我已经知道的东西互相比较,

我发现这种学习方式非常有帮助. 那么我们就以这种举例子,并且比较的方式来描述什么是Kafka吧.

Kafka就像Unix的管道

我会给一些例子来说明Kafka能干什么, 比较的对象是很多人都熟悉的: 命令行的 Unix管道

看一个简单的例子:

```
$ cat *.txt | tr A-Z a-z | grep hello
```

这段脚本找出以.txt结尾的文件中所有包含单词"hello"的行.它包含了三个步骤/阶段:

1. 从所有文件中输出所有行
2. 将所有文本转换为小写
3. 找出含有"hello"单词的行

所有这些步骤的每一个都写到标准输出流,后面的阶段会从标准的输入流中读取.

最简单的来看, Kafka就像一个Unix的管道: 你将数据写到它的一端, 然后数据从另一端出来.

(严格来说, 你写的数据会通过网络传输, 你读取的数据也是通过网络, 不过现在我们暂时忽略这些.)

如果这就是Kafka所能做的, 那有什么了不起的, 对吧? 实际上Kafka还有一些额外的特征, 带来新的能力.

结构化数据

Unix的管道在文本数据行之间流动, 通常是以新的一行为结束(这条管道). 这些行可以很长, 但是工作单元仍然是一行文本.

如果你处理的不是ASCII数据, 或者你处理的数据不能以一一行来表示就会有点麻烦. 而Kafka支持任意的格式和任意大小.

这就允许你可以存储任何数据到Kafka中: 文本, CSV, 二进制数据, 自定义编码数据等等. 对于Kafka而言, 它只是一系列的

消息, 其中每条消息都是一系列的字节. 比如可以(模拟)写一个Kafka的"命令行":

```
$ TwitterFeed | filter_tweets From @apache kafka
```

这里的filter_tweets命令可能不是一个简单的基于字符串的grep, 而是一种能够理解从TwitterFeed输出的数据格式.

比如TwitterFeed可能输出JSON, 则filter_tweets需要做些JSON的处理. TwitterFeed如果返回的是二进制数据,

则filter_tweets需要知道二进制的格式/协议. 这种灵活性可以让Kafka成为一种发送任何数据类型的Unix管道.

数据持久化

我们可能有一个复杂的会花费一些时间才能跑完的命令. 如果只运行一次, 你可能不关心. 但是如果你要多次迭代运行,

你可能会将输出结果先写到一个文件中, 这样之后的阶段可以更快地迭代, 而不需要重新多次运行很慢的那部分命令.

```
$ find . -type f | grep \.java > javafiles.txt
$ cat javafiles.txt | xargs grep ClassName
```

这个模式工作的很好,但是这意味着你需要提前计划去做(先写文件). 如果管道自身能够做这件时间就方便多了.

Kafka会持久化你发送的所有数据到磁盘上. 持久化非常方便,不仅节省了你的一些时间,它还允许你能做之前不能做的一些事情.

就像上面的命令行一样,每个阶段的输出都被保存下来. 由于第一个阶段的输出被保存了,第二个阶段甚至不要求正在运行.

这种方式, Kafka作为生产者数据和消费者数据之间的缓冲区. 它保持了数据,允许消费者可用并且准备好的时候才读取数据.

Kafka是高性能的,它甚至可以运行在多台机器上,并且可以复制统一的数据到多台机器防止数据丢失造成的风险.

三个Kafka节点组成的集群能够处理每秒钟两百万的写入, 并能使网卡饱和.

由于数据被持久化到了Kafka中,并没有要求消费者要多快去读取数据. 消费者可以想多快就多快,想多慢就多慢地读取数据.

因此它允许一个高性能的生产者, 并不会因为一个很慢的消费者而拖慢生产者的性能. 看一个很慢的消费者的例子.

```
$ produceNumbers > numbers.txt
$ cat numbers.txt | xargs findPrimeFactorization
```

从密码学我们知道,将一个数字因式分解成质数是很慢的. 假设我们分解了100万个数字,程序挂掉了.

当下次重启程序的时候如果能够从上次离开位置的那个点继续处理,而不是重复很多工作,那就很友好了.

以这个例子中,我期望的是从numbers.txt中的第一百万零一行开始继续处理.

Kafka有类似的概念叫做"offset". Kafka中的每条记录都被分配了有序的offset,消费者可以选择在指定的offset重新开始.

数据持久化和offsets这两个特性,允许你构建一个消费者数据和生产者数据分开的系统.

数据持久化--非常快的数据持久化--意味着它能很快地吸收大批量的数据.

它允许消费者按照它能够读取的任何速度读取数据. 它允许持久化数据, 即使消费者挂掉了.

offsets允许消费者继续执行, 无论它上次在什么地方退出, 而不会重复工作.

在某种情况下, 这是很有意义的: 你并不想在一次汇款中从银行账号中扣了两次钱.

另一方面, 这是出于效率方面考虑的: 你并不想重新对已经处理的数字重新进行因式分解.

无论哪种情况, 这两个特性都允许你做传统的Unix管道所不能做的事情.

流数据

再看下第一个例子:

```
$ cat *.txt | tr A-Z a-z | grep hello
```

在这里例子中, 第一个阶段(cat)输出所有的行然后就结束了. 整个管道会找到所有包含单词"hello"的行最后命令结束. 和下面的命令进行比较:

```
$ tail -F *.txt | tr A-Z a-z | grep hello
```

这个命令不会结束, 第一个阶段(tail)输出一些行, 但是仍然保持着监听更多的数据. 如果你在之后往其中的一个添加了一行, tail命令会输出这个新行, 然后接下来的命令会处理它.

Kafka支持相同的概念. 数据写到到Kafka并且被消费者读取可以看做一个流.

如果消费者到达数据的末尾, 它会继续等待即将到来的更多的数据. 当新的数据写入到Kafka, 它会很快地被发送到消费者.

我在之前说过数据流进Kafka是很快的, 实际上数据从Kafka流出也是很快的.

一条记录被添加到Kafka后, 能够在20ms之内发送给一个正在等待的消费者.

现在我们知道Kafka除了支持数据持久化, 也支持流数据. 我们复习下之前的例子

```
$ produceNumbers > numbers.txt
$ cat numbers.txt | xargs findPrimeFactorization
```

上面的命令看起来向上一种批处理模式, 因为produceNumbers最终会结束的. 但是数字是无限的, 它永远不会结束, 所以实际上看起来应该是这样的:

```
$ produceNumbers |* findPrimeFactorization
```

这里我自己造了一个语法: `|*` 表示这是一个Kafka管道. 它能够归档所有东西到磁盘, 并且发送流式的更新.

streaming updates流式更新, 数据是流式传入的, 下游的方法基于最新的流数据做更新操作. 即对流数据更新操作

这种流式的数据允许你创建一个实时的管道, 这里有个例子:

```
$ tail -F /var/log/httpd/access_log |* grep index.html |* get_load_
```

这个管道会查询你的web服务器日志. 它会提取主页的所有pageload, 获取出页面加载的时间, 创建一个可视化的图, 并及时更新.

太棒了, 你刚刚创建了一台服务器的监控面板. 如果页面加载时间抖动, 你可以在几秒内从图中观察到.

所有的这些Kafka管道(每个 `|*`)都会持久化和缓冲数据. 管道中的任何一个阶段都可能出错, 并在任何时候重启,

并且可以在它们上次离开的地方继续. 它们可以处理的很慢, 或者一直紧紧跟着(上一个阶段).

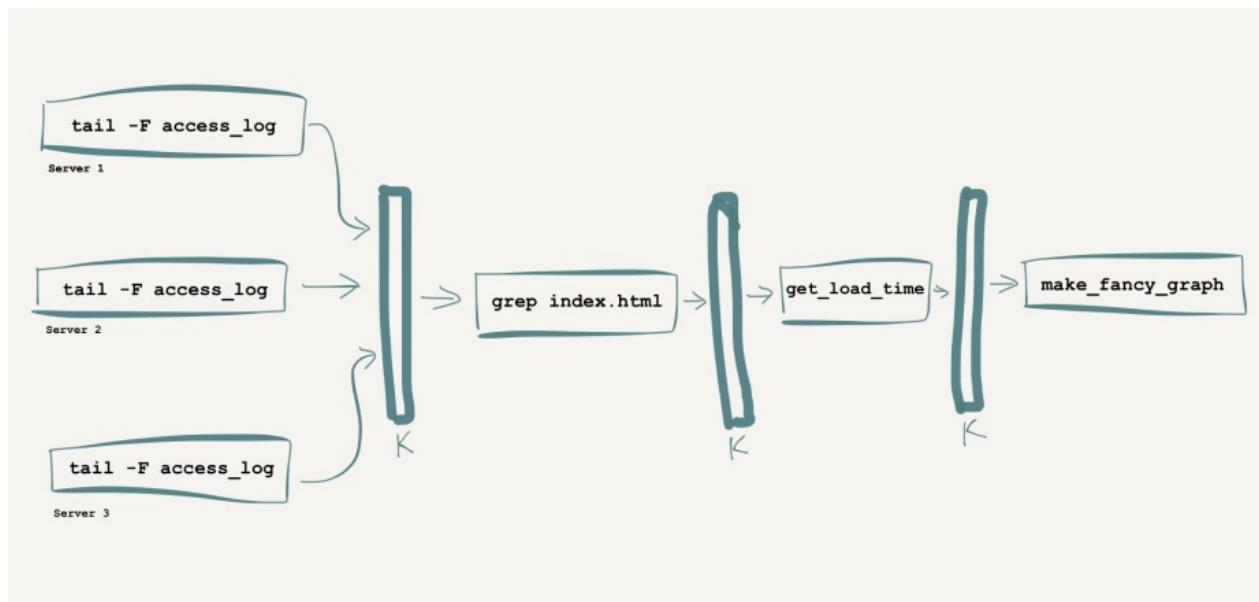
或者如果它们落后的太多, 可以被停止, 并移到新拥有更快CPU的服务器上, 也能够从上一次作业离开的地方继续.

你还可以创建一些其他类型的实时管道:

- 在黑色星期五这天实时更新你的店铺的销量. 你不仅能够实时获知哪些物品的销量, 还能实时地响应: 对畅销品订阅更多的库存.
- 实时收集登陆次数, 并注入到指令监测系统用来检查正在进行的攻击, 并且能够屏蔽欺诈的IP地址
- 实时更新交通速度传感器, 你能够分析交通模式, 并控制交通灯的时间

Fan in

Kafka同时也支持多个生产者往相同的地方写数据. 想象下前面的场景, 但现在从多个服务器上收集web服务器日志.



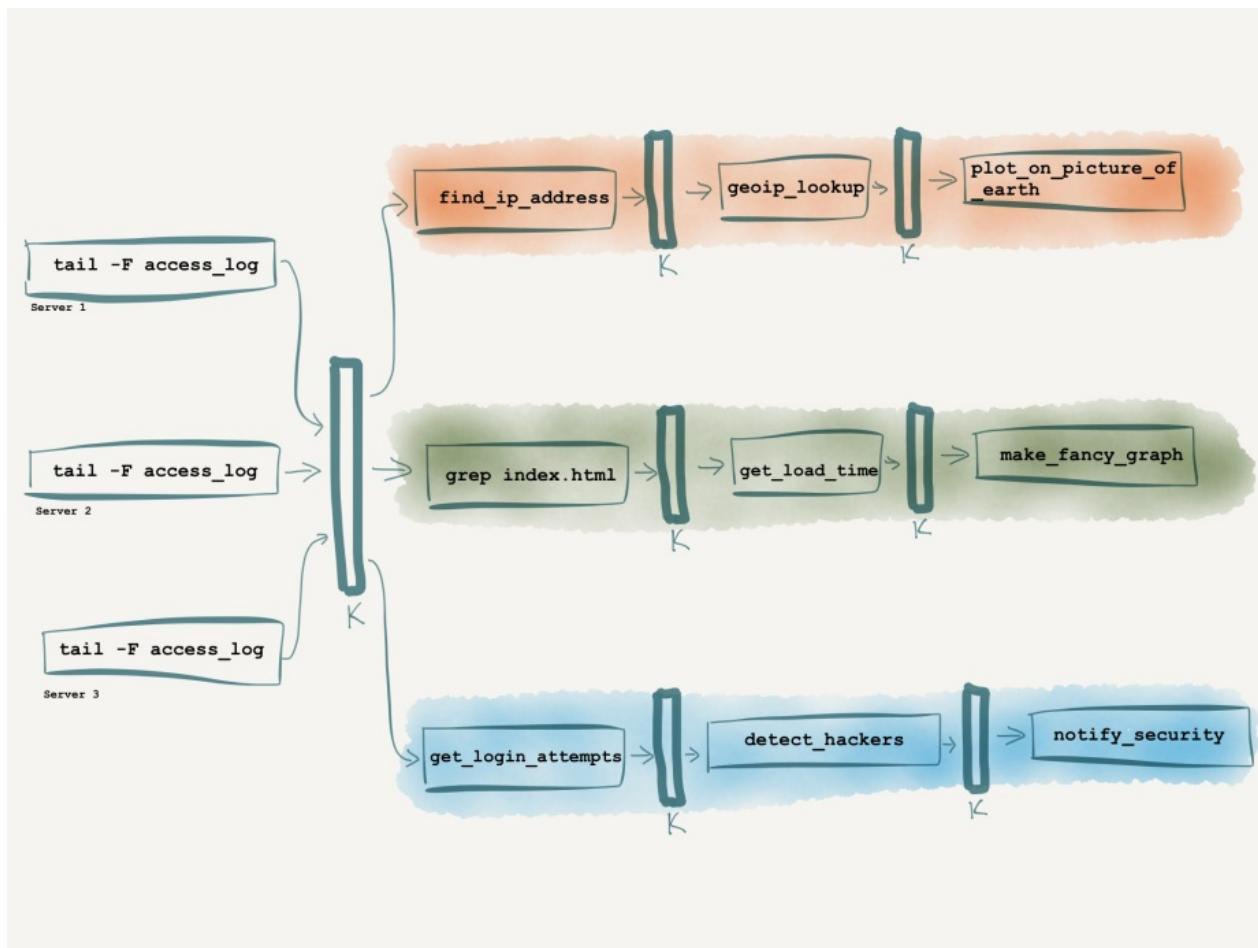
所有的服务器以漏斗形式的数据流入到Kafka管道. 你只有一个grep的进程在运行, 获取加载时间的进程在运行, 只有一个绘图的进程在运行. 但是它们是基于所有web服务器的输出日志的聚合. 恭喜你, 现在创建了一个数据中心的监控面板.

这里的好处是你可以从很多的地方收集数据, 但是只在一个中央的地方存储并处理所有这些收集到的数据.

Kafka可以成为你的公司中所有数据的中心收集节点. 将分散在各个服务器上的数据都收集到统一一个节点.

Fan out

Kafka不仅支持多个生产者写到同一个地方, 也支持多个消费者从相同的地方读取数据.



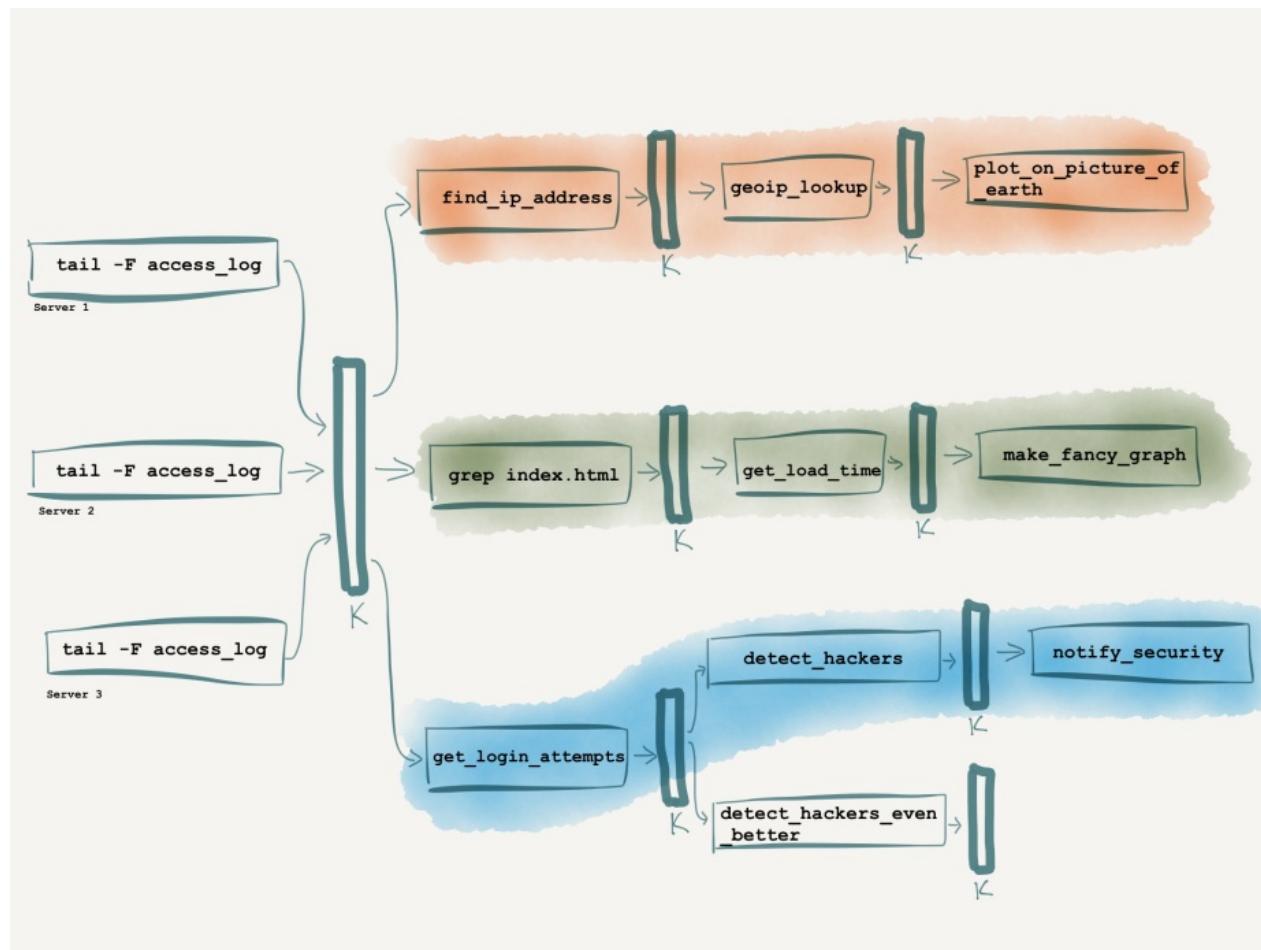
再强调一次,Kafka在多个阶段之间能够缓冲数据. 上面的三个管道: `find_ip_address` , `grep index.html` , `get_login_attempts` --都能够按照自己的步伐(消费速度)从`access_log`这个 Kafka管道中读取数据.

前面两个看起来会相当快,但是第三个可能会慢点.但是没关系,Kafka会保持这些数据(不会因为其他消费者消费了就删除数据)

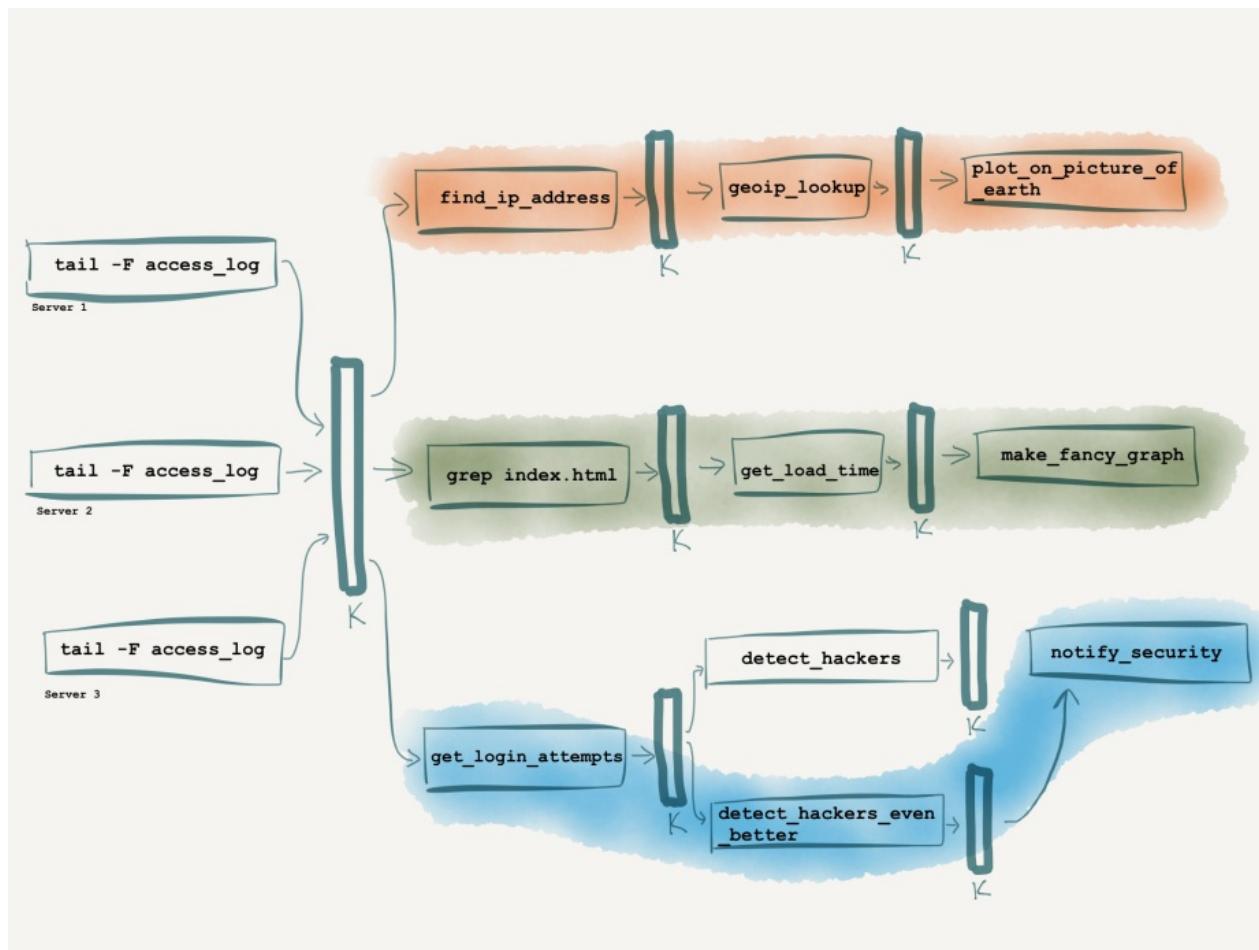
这样的好处是一个单一的数据源可能用不同的方式处理,每种使用方式都和其他方式都是独立的,并且不会相互影响.

假设我们找到了一种检测黑客的方式. 我们可以将 `detect_hackers` 实例部署在已有的实例旁(共存),然后一起测试.

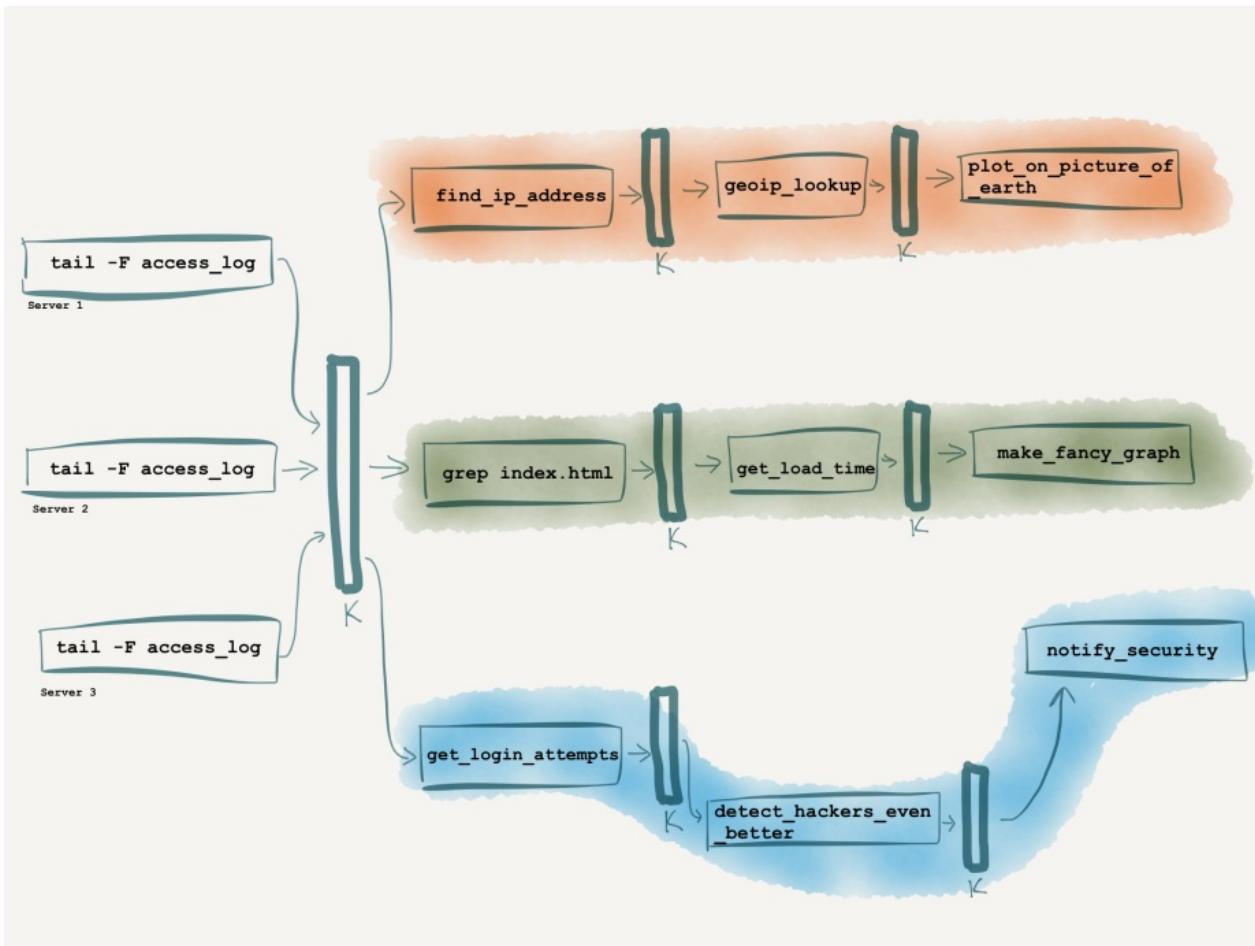
对于相同的输入,看看他们都有什么不同的表现(验证我们的新的检测方式是否达到了预期的效果).



一旦我们决定选择使用新的方式会更好点,我们会通知下游的 `notify_security` 作业监听更好的检测方式.



并且新的方式真的很稳定了,我们可以将老的检测方式移除掉.



看看我们都做了什么？

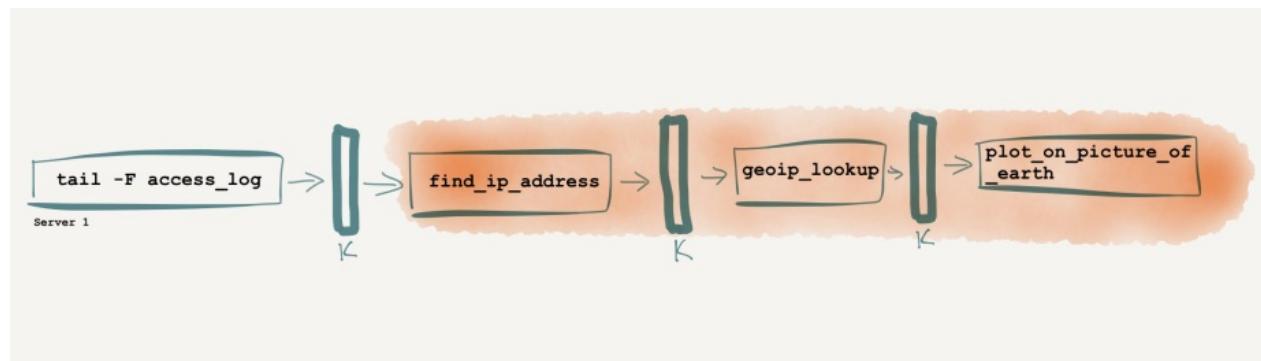
- 1.我们在生产环境的数据上直接运用新的算法,并做了真实的测试
- 2.对相同的数据,将新的算法和旧的算法一起测试
- 3.仅仅使用了一个开关就更改了notify_security作业的输入
- 4.保持旧的算法继续运行,以防需要切回去(上面的场景实际是将旧的算法删除了)

这个特性使得Kafka带给我们的威力非常大.通过将同一份数据分散到多个地方,我们可以从数据中获得多个分组的能力.

每个管道的工作都是独立的并且都是以自己的消费速度进行的. 并且让我们在开发新的功能时能够重用已经存在的数据.

并行

让我们专注于上面多个管道中的其中一个.

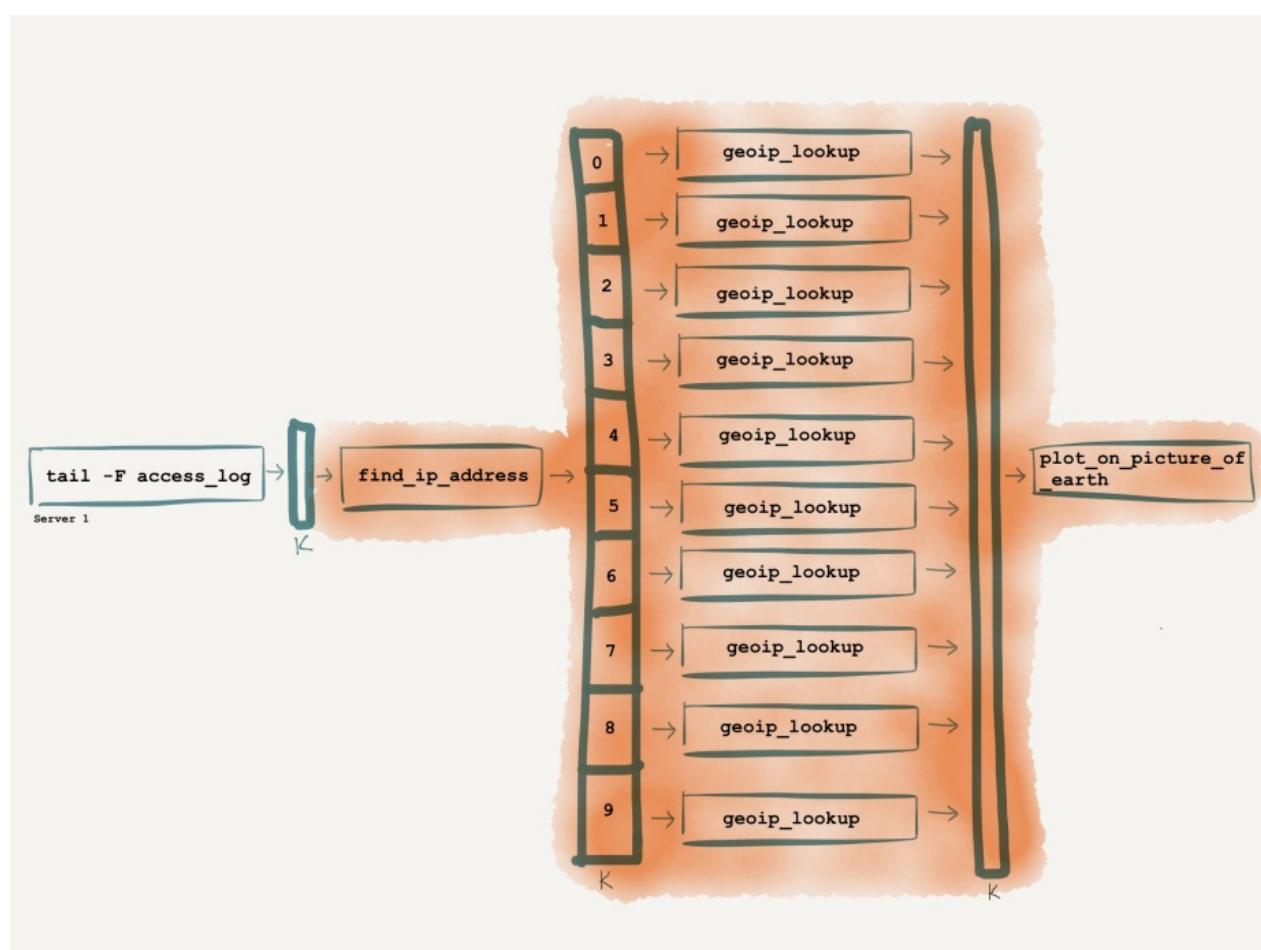


假设geoip(地理位置)数据库是非常慢的. Kafka会在这个阶段之前缓冲所有的数据, 所以即使很慢, 也不会丢失任何数据.

但是查询geoip会拖慢整个管道的速度. 所以你会部署一个很快速的geoip数据库. 但是这并不能帮你太多, 因为你每次

都是从find_ip_address的输出结果中一条接着一条地查询. 你真正需要的是并行!

Kafka支持在你的Kafka管道中添加子管道(sub-pipes). 你可以将所有以1结束的IP地址发送到第一个子管道, 将所有以2结束的IP地址发送到第二个管道, 等等. 现在你的请求能够通过round-robin的方式发送到数据库中. 看起来是这样的:



Kafka管道中的数字0到9表示所有以这个数字结束的IP地址,会被放到相同的管道中(图中每个geoip_lookup都是一个子管道)

每个geoip_lookup作业都只会从find_ip_address管道中读取一部分数据,可以允许你以并行的方式查询:一次10个线程.

这种方式应该能满足你在geoip阶段快速地在地球图形上绘点,这下你满意了吧!

Kafka称所有的这些是 partitions . 它允许你将数据以逻辑的分组方式分到多个通道中,但是每个函数都是独立的.

一批数据会分散到多个节点,每个节点之间都做同样的工作.但是它们之间不会相互影响的.

Kafka和Unix哲学

仔细看看上面的例子,你会发现Kafka的管道这个角色是很小的.Kafka管道并不会做过滤IP地址的工作,不会做查询IP地址的工作,

也不会对很大的数字做因式分解.这都取决于你. Kafka做的事情是将你的所有工具都联系在一起.这样看来它就像胶水/粘合物.

但是它这个粘合物能够让你构建出很多有趣的东西. Kafka负责很多平凡的事情,而这些都是作为事情的解决者的你并不愿意去做的.

它能够帮你保存数据,能在任何一个点开始读取数据,可以从多个数据源聚合数据,并将数据同时发送给多个目标.

Kafka这种能力让你重新思考解决问题的方式. 将一个问题分解成多个阶段,每个阶段可以单独开发实现,并独立地测试.

这一切都是基于Kafka能将所有的组件都粘合在一起. 而且Kafka可以在网络之间完成这些事情,所以你甚至可以将你的计算组件

分布在多个节点,也就有了水平扩展,分布式处理,高可用性等特点.

这种将一个大问题分解成多个小问题的思想和Unix的哲学是一致的. 实际上Unix管道的发明人Doug McIlroy这么说过:

This is the Unix philosophy: Write programs that do one thing and do it well.
Write programs to work together. Write programs to handle text streams,
because that is a universal interface.

Kafka允许你将Unix哲学运用到工程师急待解决的大数据量,低延迟,网络之间的问题.

声明

在这篇文章中,我简化了一些事情,现在我们解释下之前遗留的东西.

1. Kafka是一个软件,你能够通过网络和它对话. 它有自己定制的网络协议,但有客户端库帮你做这些事情了.
2. 有方便的命令行kafka-console-producer.sh读取标准输入流写到Kafka中. kafka-console-consumer.sh可以从Kafka中读取输出,输出到标准输出流. 你可以使用他们实现上面的命令.
3. Kafka客户端使得你能够从Kafka中读写数据构建自己的应用程序
4. Kafka的管道实际上是叫做"topics"
5. Kafka的topic都有名称. 每个topic的数据集和其他topic都是分开的.

EOF.

Kafka的Producer新客户端API实现(JAVA)

KafkaProducer

Producer

生产者线程:异步发送消息,提供一个Callback;同步发送消息,则调用Future.get()会Block住直到结果返回.

```

public class Producer extends Thread {
    private final KafkaProducer<Integer, String> producer;
    private final String topic;
    private final Boolean isAsync;

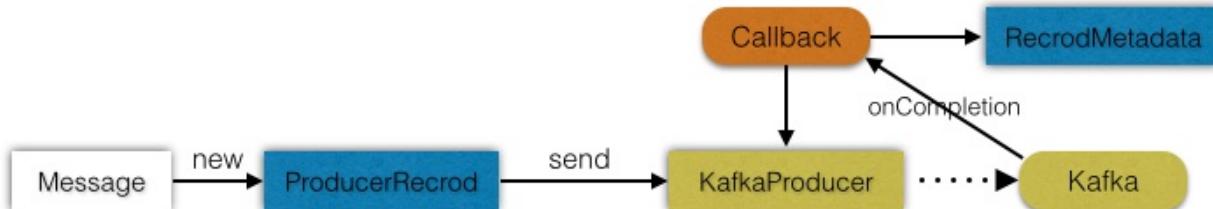
    public Producer(String topic, Boolean isAsync) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("client.id", "DemoProducer");
        props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        producer = new KafkaProducer<Integer, String>(props);
        this.topic = topic;
        this.isAsync = isAsync;
    }

    public void run() {
        int messageNo = 1;
        while (true) {
            String messageStr = "Message_" + messageNo;
            if (isAsync) { // Send asynchronously
                producer.send(new ProducerRecord<Integer, String>(topic, messageStr));
                new Callback() {
                    public void onCompletion(RecordMetadata metadata, Exception e) {
                        System.out.println("The offset of the record is " + metadata.offset());
                    }
                };
            } else { // Send synchronously
                producer.send(new ProducerRecord<Integer, String>(topic, messageStr));
            }
            ++messageNo;
        }
    }
}

```

- KafkaProducer需要指定消息Key,Value的类型. ProducerRecord还需要指定topic.

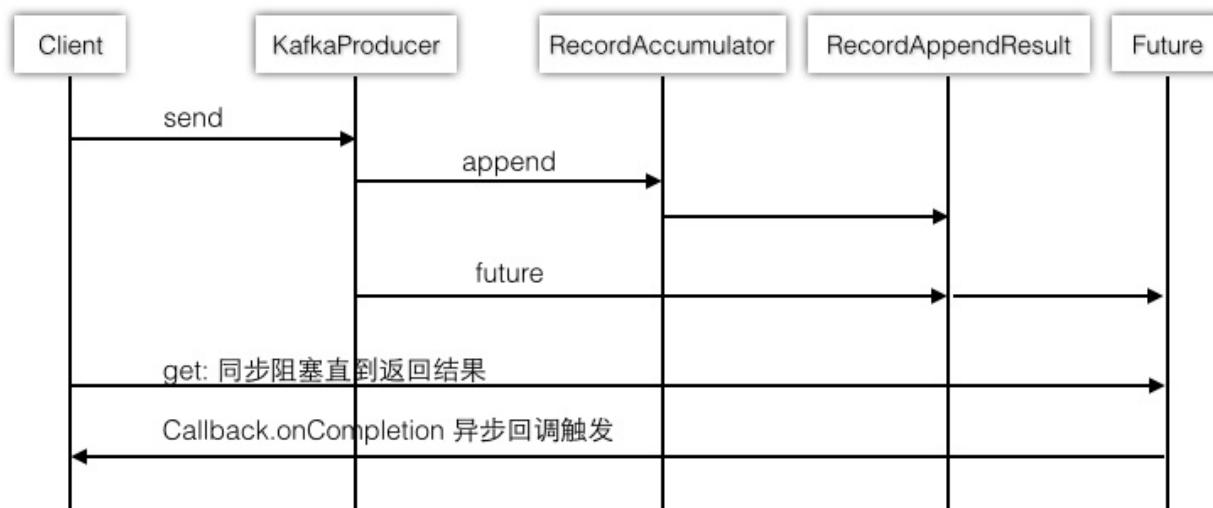
- 根据配置文件创建KafkaProducer, 指定了Broker地址, Key,Value的序列化方式, 消息必须要指定topic
- 发送消息的返回结果RecordMetadata记录元数据包括了消息的offset(在哪个partition的哪里offset)



blocking vs non-blocking

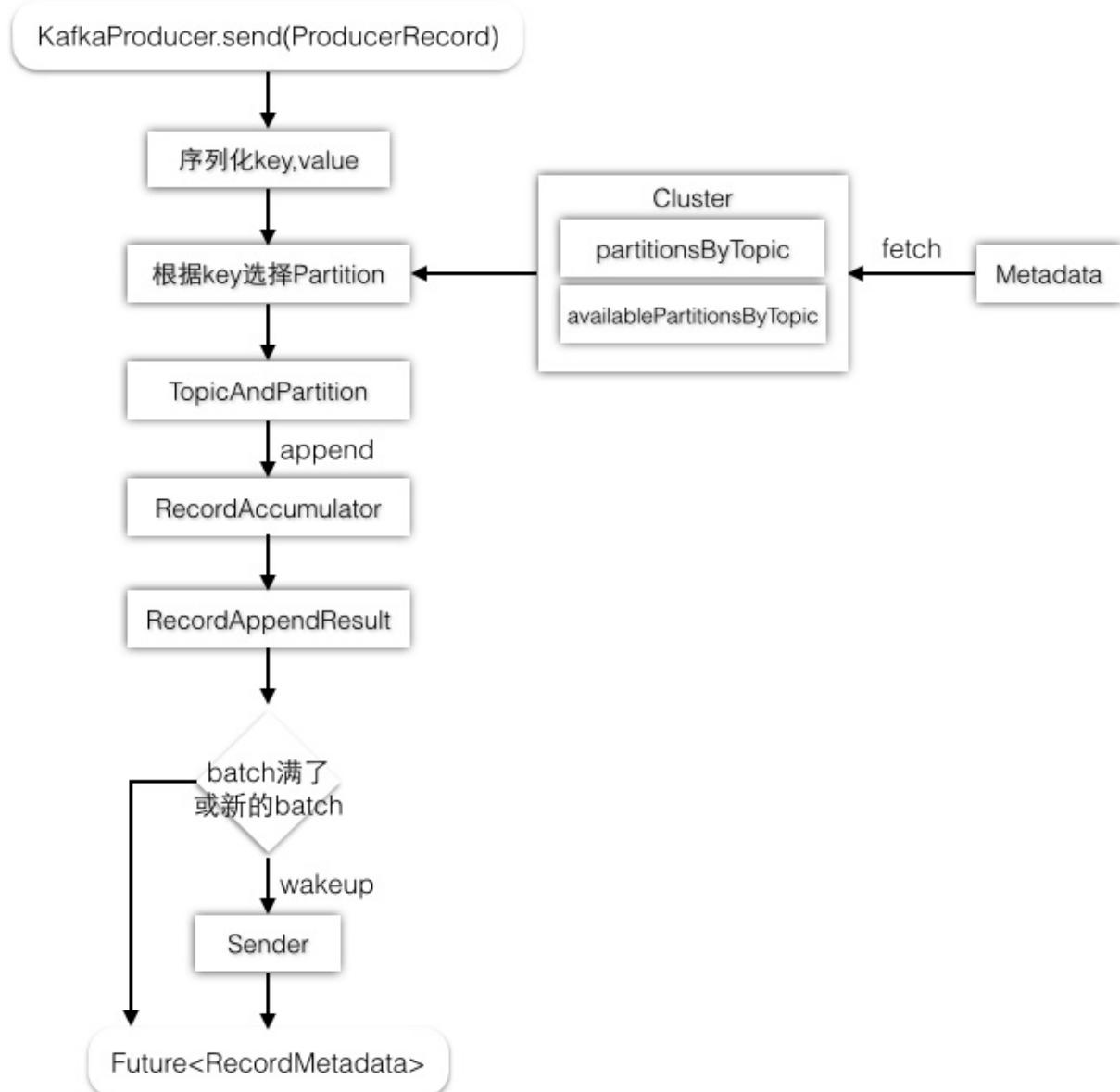
KafkaProducer.send方法返回的是一个Future,那么它如何同时实现blocking方式和non-blocking方式.

- blocking: 在调用send返回Future时, 立即调用get, 因为Future.get在没有返回结果时会一直阻塞
- non-block: 提供一个callback, 调用send后, 可以继续发送消息而不用等待.当有结果返回时,callback会被自动通知执行



```
public Future<RecordMetadata> send(ProducerRecord<K, V> record,  
    // first make sure the metadata for the topic is available  
    long waitedOnMetadataMs = waitOnMetadata(record.topic(), t)  
    long remainingWaitMs = Math.max(0, this.maxBlockTimeMs - wa  
  
    // 序列化key和value  
    byte[] serializedKey= keySerializer.serialize(record.topic()  
    byte[] serializedValue = valueSerializer.serialize(record.t  
  
    // 选择这条消息的Partition  
    int partition = partition(record, serializedKey, serializedV  
TopicPartition tp = new TopicPartition(record.topic(), partitio  
RecordAccumulator.RecordAppendResult result = accumulator.a  
  
    // 在每次追加一条消息到收集器之后,都要判断是否满了.如果满了,就执行一次  
    if (result.batchIsFull || result.newBatchCreated) this.send  
    return result.future;  
}
```





在发送消息前,消息所属的topic必须已经建好,并且也指定这个topic的partition数量(没有指定则默认是server.properties的 num.partitions).

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replicat:
```

所以 Topic , Partition , Key , Value 组合起来就能表示 消息 发送到哪个 topic 的哪个 partition 上.

partition

一个Partition的主要组成部分是topic名称,partition编号,所在的Leader,所有的副本,isr列表.表示这个Partition的分布情况.

```

public class PartitionInfo {
    private final String topic;
    private final int partition;
    private final Node leader;
    private final Node[] replicas;
    private final Node[] inSyncReplicas;
}

```

下图是kafka-manager中某个topic的PartitionInfo信息(副本数=4,Broker数量刚好也是4,导致每个Partition都分布在所有Broker上).

Partition	Leader	Replicas	In Sync Replicas
0	5	(2,4,5,0)	(0,5,2,4)
1	2	(4,5,0,2)	(2,5,4,0)
2	0	(5,0,2,4)	(0,5,4,2)
3	0	(0,2,4,5)	(0,5,2,4)
4	5	(2,5,0,4)	(0,5,2,4)
5	2	(4,0,2,5)	(2,0,4,5)

在Cluster的构造函数中, 会根据所有节点和所有partitions构建集群状态信息.availablePartitions只保存有Leader的Partition.

正常来说每个Partition都是有Leader Partition的. 如果Partition没有Leader的话, 说明这个Partition就是有问题的.

```

List<PartitionInfo> availablePartitions = new ArrayList<
for (PartitionInfo part : partitionList) {
    if (part.leader() != null)
        availablePartitions.add(part);
}
this.availablePartitionsByTopic.put(topic, Collections.

```

要选择消息所属的partition,首先需要知道topic一共有多少个partition(numPartitions),

所以metadata.fetch获得的Cluster信息中有topic->partitions的映射关系(partitionsByTopic).

消息有key的话,对key进行hash,然后和partitions数量取模,类似于round-robin的方式来确定key所在的partition达到负载均衡.

如果消息没有key,会根据递增的counter的值确定partition, count不断递增,确保消息不会都发到同一个partition里.

问题: 写入消息时是写到Leader Partition的话,下面的代码如何体现Leader?

答案: 实际上为消息选择Partition,只是为了负载均衡,跟Leader没有多大关系.

因为一个PartitionInfo一定能确定一个唯一的Leader(一个Partition只有一个Leader)

如果一个topic只有一个Partition的话,在集群环境下就不能水平扩展:这个topic的消息只能写到一个节点.

而为一个topic设置多个Partition,可以同时往多个节点的多个Partition写数据.

注意: 多个Partition都是同一个topic的,每个Partition的逻辑意义都是相同的,只是物理位置不同而已.

```
public int partition(String topic, Object key, byte[] keyBytes,
    // 这个topic所有的partitions. 用来负载均衡, 即Leader Partition
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size();
    if (keyBytes == null) {
        int nextValue = counter.getAndIncrement();
        // 这个topic可以使用的partitions: availablePartitionsByTopic
        List<PartitionInfo> availablePartitions = cluster.availablePartitions(
            topic);
        if (availablePartitions.size() > 0) {
            int part = DefaultPartitioner.toPositive(nextValue) % numPartitions;
            return availablePartitions.get(part).partition();
        } else {
            // no partitions are available, give a non-available partition
            return DefaultPartitioner.toPositive(nextValue) % numPartitions;
        }
    } else {
        // hash the keyBytes to choose a partition
        return DefaultPartitioner.toPositive(Utils.murmur2(keyBytes));
    }
}
```

下图是partition的分布算法.topic1有4个partition. 则总共有4个对应的PartitionInfo对象.

每个PartitionInfo(比如topic1-part1)都有唯一的Partition编号(1),replicas(1,2,3).

注意:replicas并不是一个PartitionInfo对象,它们仅仅是某个Partition编号对应的PartitionInfo的replicas信息.

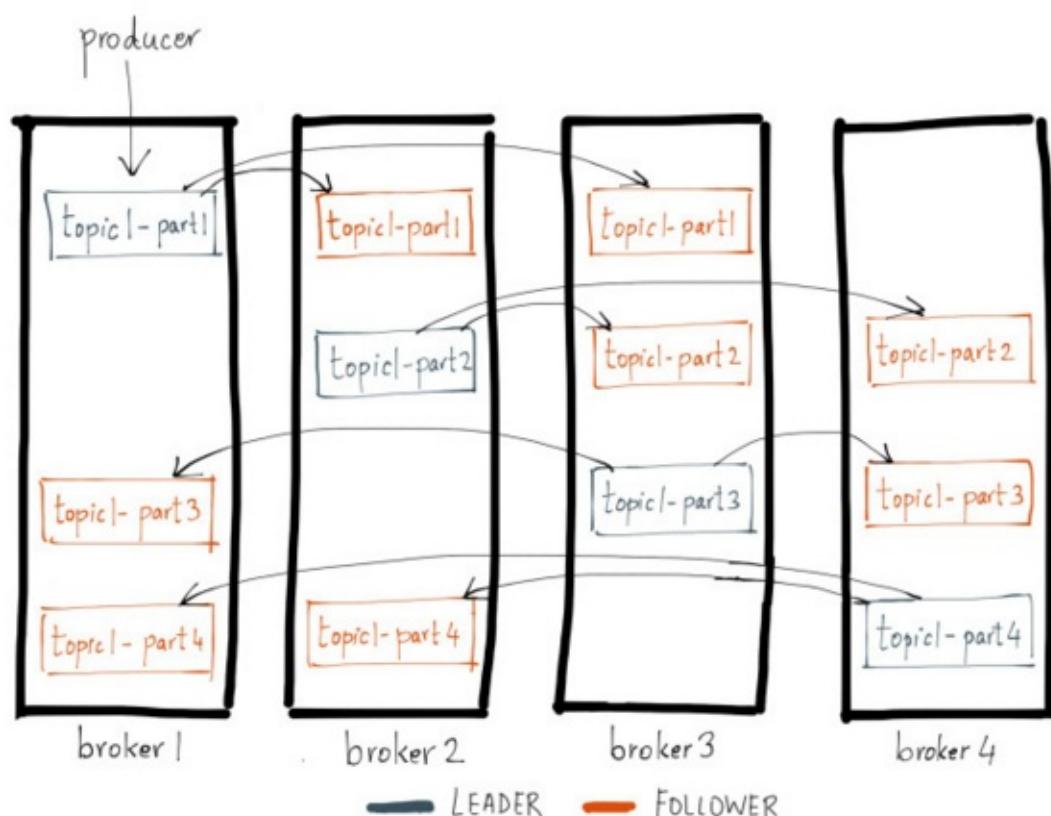
即partitionsForTopic和availablePartitionsForTopic里面其实是没有follower replicas的.

因为如果Replicas都算作PartitionInfo的话,则Partition编号就不好表示了(4个Partition,每个Partition由3个副本).

实际上在选择Partition的时候,根本就先不要考虑replicas的存在.就只有Partition编号.

每个Partition是分布在不同的节点上的(可以把这个Partition就认为是Leader Partition).

然后在写消息的时候采用round-robin方式将消息平均负载到每一个Partition上.假设第一条消息写到了topic1-part1,则下一条消息就写到topic1-part2,以此类推.



Kafka中的副本是以Partition为粒度的. Follower仅仅顺序地拷贝Leader的日志

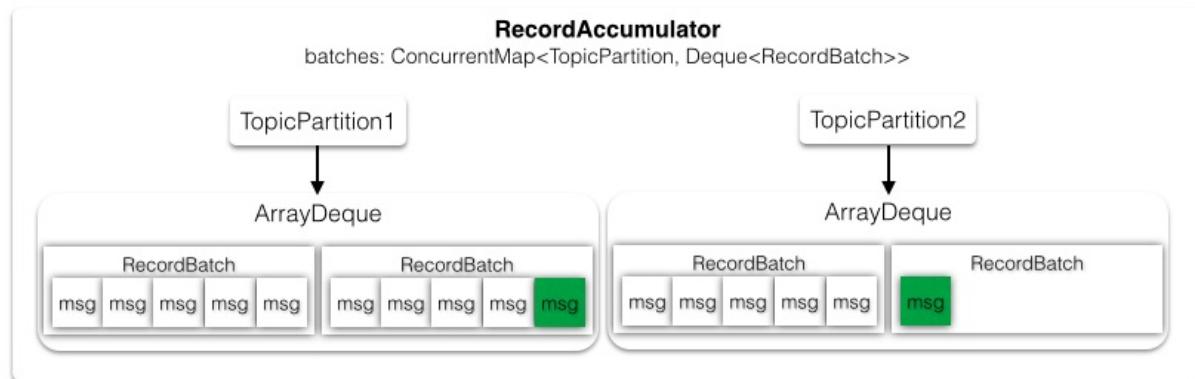
RecordAccumulator

由于生产者发送消息是异步地,所以可以将多条消息缓存起来,等到一定时机批量地写入到Kafka集群中,RecordAccumulator就扮演了缓冲者的角色.

生产者每生产一条消息,就向accumulator中追加一条消息,并且要返回本次追加是否

导致batch满了,如果batch满了,则开始发送这一批数据.

最开始以为 `Deque<RecordBatch>` 就是一个消息队列,实际上一批消息会首先放在 `RecordBatch` 中,然后Batch又放在 双端队列 中.

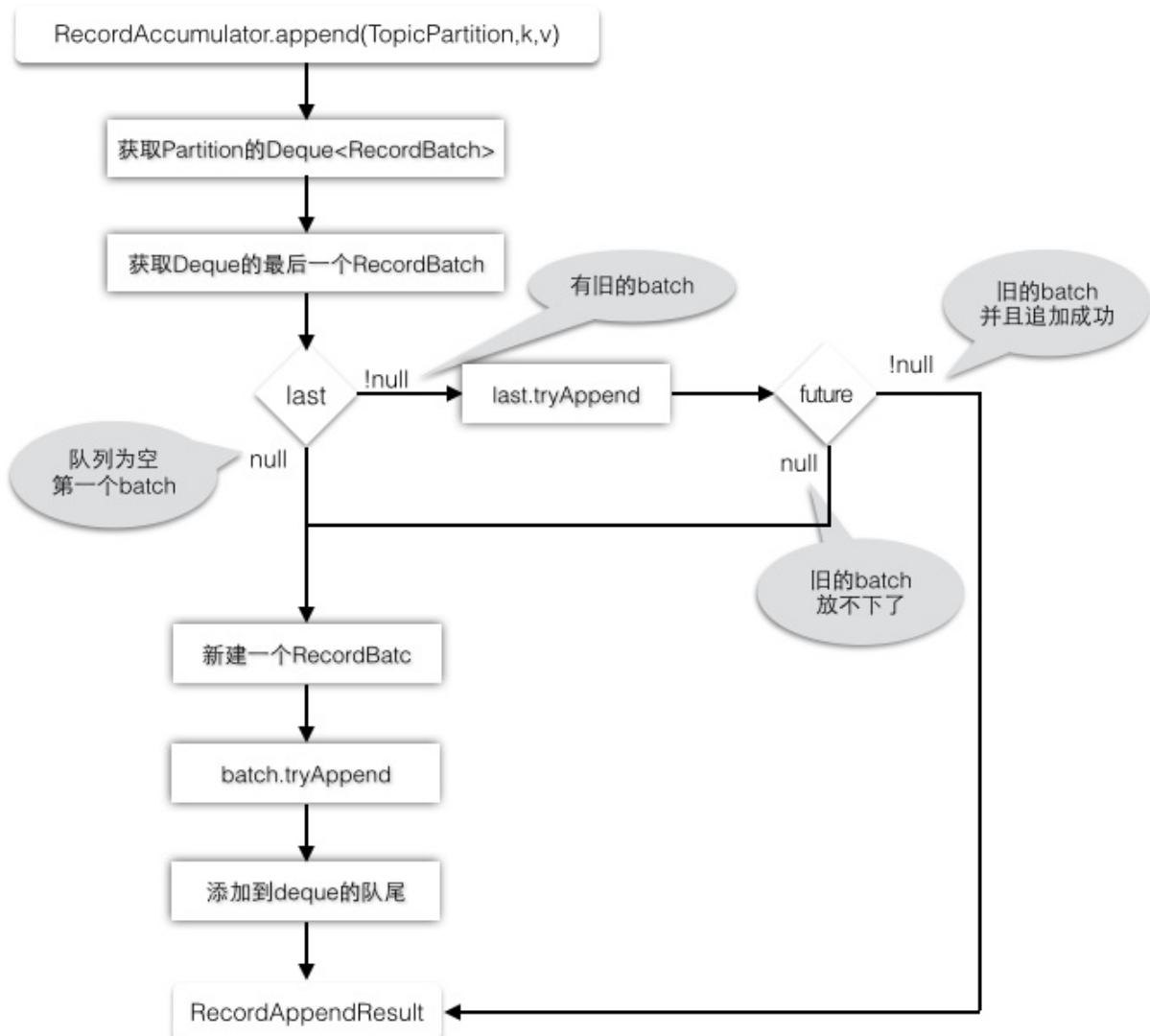


```

public RecordAppendResult append(TopicPartition tp, byte[] key,
        Deque<RecordBatch> dq = dequeFor(tp);
        synchronized (dq) {
            RecordBatch last = dq.peekLast();
            if (last != null) {
                FutureRecordMetadata future = last.tryAppend(key, );
                // 有旧的batch, 并且能往这个batch继续追加消息
                if (future != null) return new RecordAppendResult(1)
            }
            // 队列为空(没有一个RecordBatch, last=null), 或者新的RecordBatch
            int size = Math.max(this.batchSize, Records.LOG_OVERHEAD +
                    ByteBuffer buffer = free.allocate(size, maxTimeToBlock);
            synchronized (dq) {
                // 内存的ByteBuffer, 追加新消息时,会最终写到这个ByteBuffer中
                MemoryRecords records = MemoryRecords.emptyRecords(buffer);
                RecordBatch batch = new RecordBatch(tp, records, time.now());
                FutureRecordMetadata future = Utils.notNull(batch.tryAppend(key));
                dq.addLast(batch);
                incomplete.add(batch);
                return new RecordAppendResult(future, dq.size() > 1 || incomplete.size() > 1);
            }
        }
    }
}
    
```

batches是一个并发安全的,但是每个TopicPartition里的ArrayDeque并不是线程安全的,所以在修改Dequeue时都需要同步块操作.

队列中只要有一个以上的batch(dq.size),或者追加了这条消息后,当前Batch中的记录满了(batch.records),就可以发送消息了.



RecordBatch的tryAppend判断MemoryRecords是否能容纳下新的消息,如果可以就追加,如果没有空间返回null,让调用者自己新建一个Batch.

所以一个RecordBatch只对应了一个MemoryRecords. 而一个MemoryRecords可以存放至多maxRecordSize大小的消息.

注意: 客户端传递的Callback是在这里和消息一起被加入的. 但是因为生产者是批量地写数据,所以回调函数是在一批数据完成后才被调用

```
public FutureRecordMetadata tryAppend(byte[] key, byte[] value,  
    boolean roomEnough = this.records.hasRoomFor(key, value)  
    if(!roomEnough) return null;  
    this.records.append(0L, key, value);  
    this.maxRecordSize = Math.max(this.maxRecordSize, Record.re  
FutureRecordMetadata future = new FutureRecordMetadata(this.  
    if (callback != null) thunks.add(new Thunk(callback, future));  
    this.recordCount++;  
    return future;  
}
```

思考:为什么追加数据的offset固定是0? 实际上由于消息之间都是独立的,一条消息自己是无法确定自己的offset的. 那么offset是怎么管理的?

在Sender线程开始运行之前,首先要找到 每个PartitionInfo的Leader 节点 ,由 RecordAccumulator统一收集已经准备好的节点.

```

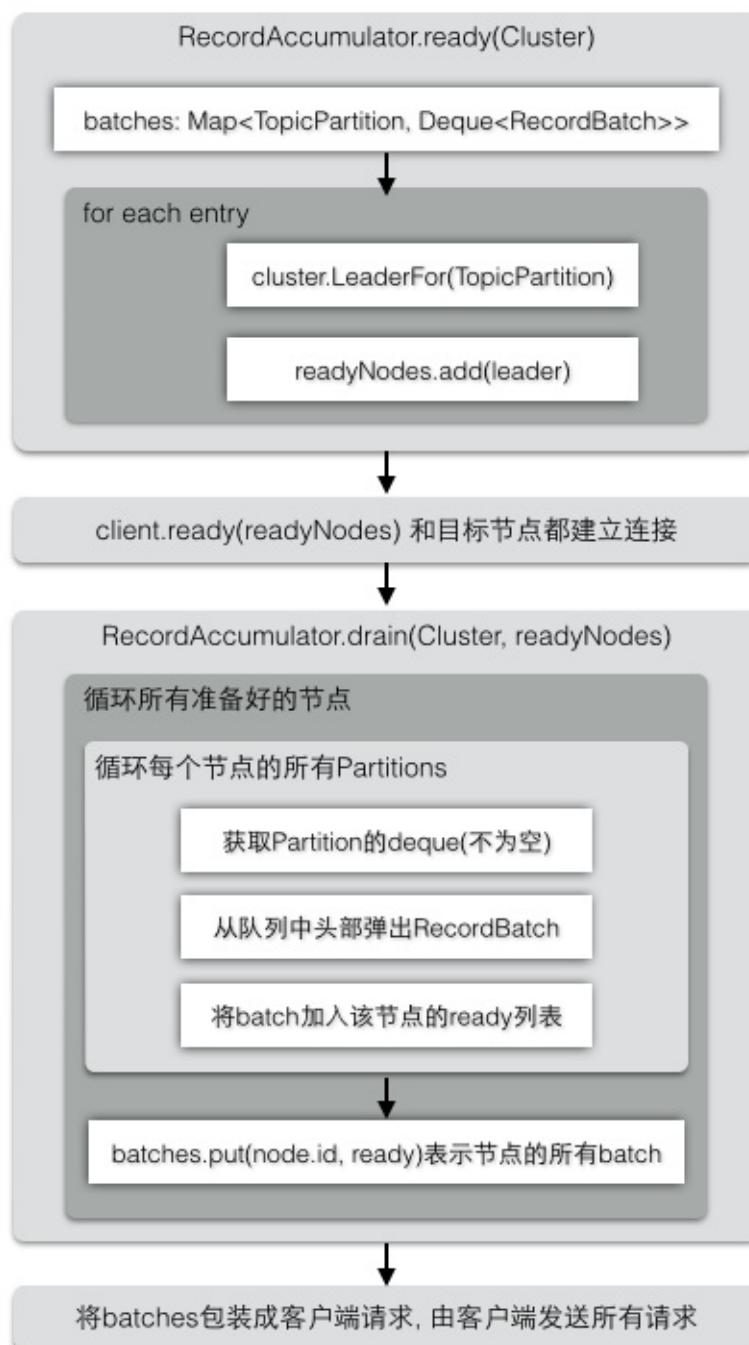
public ReadyCheckResult ready(Cluster cluster, long nowMs) {
    Set<Node> readyNodes = new HashSet<Node>();
    // batches: 每个TopicPartition都对应了一个双端队列
    for (Map.Entry<TopicPartition, Deque<RecordBatch>> entry :
        TopicPartition part = entry.getKey();
        Deque<RecordBatch> deque = entry.getValue();
        // 找出这个TopicPartition的Leader节点, 在正式开始发送消息时,
        Node leader = cluster.leaderFor(part);
        if (leader == null) {
            unknownLeadersExist = true;
        } else if (!readyNodes.contains(leader)) {
            synchronized (deque) {
                RecordBatch batch = deque.peekFirst();
                if (batch != null) {
                    boolean sendable = full || expired || exhausted;
                    if (sendable && !backingOff) {
                        // 加入到等待连接的节点中.
                        readyNodes.add(leader);
                    } else {
                        nextReadyCheckDelayMs = Math.min(nextReadyCheckDelayMs, timeLeftForSendableBatch);
                    }
                }
            }
        }
    }
    return new ReadyCheckResult(readyNodes, nextReadyCheckDelayMs);
}

```

数据有生产就会有被消费的地方, 对应Deque队列的话, 将RecordBatch加入, 就有对应的pollFirst获取并删除第一个batch.

由于在生产数据的时候, 每个TopicPartition都有自己的队列, 并且都统一被收集到了RecordAccumulator的batches中.

在消费数据的时候, 最好对batches中的每个TopicPartition重新整理成以Node节点为级别, 对后面的发送流程是有很大帮助的.



```

public Map<Integer, List<RecordBatch>> drain(Cluster cluster, Set<Integer> nodes) {
    Map<Integer, List<RecordBatch>> batches = new HashMap<Integer, List<RecordBatch>>();
    for (Node node : nodes) {
        int size = 0;
        List<PartitionInfo> parts = cluster.partitionsForNode(node);
        List<RecordBatch> ready = new ArrayList<RecordBatch>();
        int start = drainIndex = drainIndex % parts.size();
        do {
            PartitionInfo part = parts.get(drainIndex);
            Deque<RecordBatch> deque = dequeFor(new TopicPartition(part.topic(), part.partition()));
            if (deque != null) {
                synchronized (deque) {
                    RecordBatch first = deque.peekFirst();
                    if (first != null) {
                        RecordBatch batch = deque.pollFirst();
                        batch.records.close();
                        ready.add(batch);
                    }
                }
            }
            this.drainIndex = (this.drainIndex + 1) % parts.size();
        } while (start != drainIndex);

        batches.put(node.id(), ready);
    }
    return batches;
}

```

Sender

RecordAccumulator.RecordAppendResult的batch满了,唤醒Sender线程.Sender线程的启动在创建KafkaProducer时.

Sender再唤醒NetworkClient(不是线程,相当于通知客户端开始服务了),client也唤醒Selector,最终唤醒NIO的Selector.

为什么需要有wakeup动作:因为可能有线程在select等待事件被阻塞了(没有事件),通过wakeup唤醒那个线程开始工作(有事件进来了)

Sender不仅承载了RecordAccumulator记录的收集器,也要通知客户端服务:把Accumulator收集的批记录通过客户端发送出去.

Sender作为一个线程,是在后台不断运行的,如果线程被停止,可能RecordAccumulator中还有数据没有发送出去,所以要优雅地停止.

```
public void run() {  
    while (running) {  
        run(time.milliseconds());  
    }  
    while (!forceClose && (this.accumulator.hasUnsent() || this.  
        run(time.milliseconds()));  
    }  
    this.client.close();  
}
```



发送消息的工作统一由Sender来控制.之前的wakeup只是一个通知,实际的工作还是由线程的run方法来控制的.

同样调用client.send也只是把请求先放到队列中, client.poll才是会将读写真正发送到socket链路上.

```

public void run(long now) {
    Cluster cluster = metadata.fetch();
    // ① get the list of partitions with data ready to send
    RecordAccumulator.ReadyCheckResult result = this.accumulator
    // ② remove any nodes we aren't ready to send to 建立到Lea
    Iterator<Node> iter = result.readyNodes.iterator();
    long notReadyTimeout = Long.MAX_VALUE;
    while (iter.hasNext()) {
        Node node = iter.next();
        if (!this.client.ready(node, now)) {
            iter.remove();
            notReadyTimeout = Math.min(notReadyTimeout, this.c
        }
    }

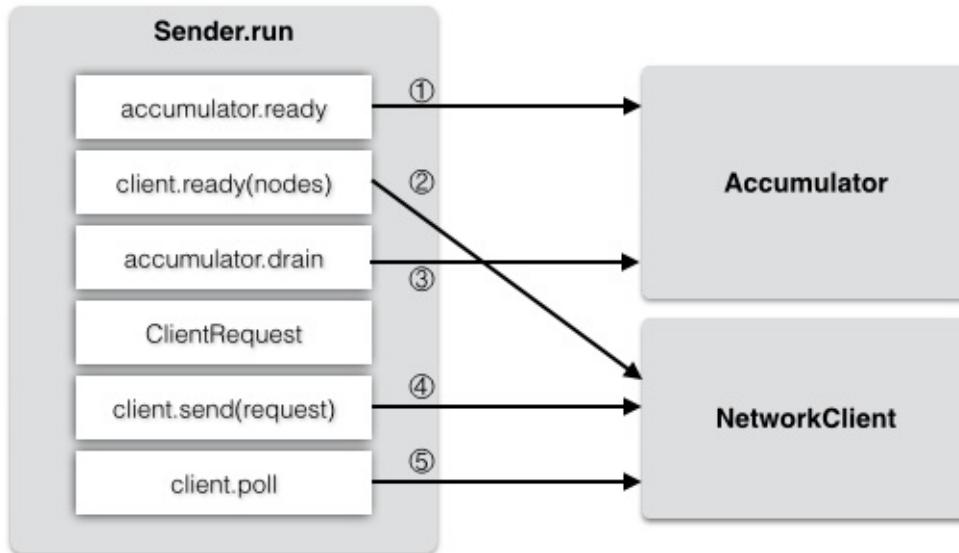
    // ③ create produce requests 之前加入了accumulator收集器中,
    Map<Integer, List<RecordBatch>> batches = this.accumulator
    // ④ Transfer the record batches into a list of produce re
    List<ClientRequest> requests = createProduceRequests(batches

    long pollTimeout = Math.min(result.nextReadyCheckDelayMs, r
    // ⑤ Queue up the given request for sending. Requests can
    for (ClientRequest request : requests) client.send(request,
    // ⑥ Do actual reads and writes to sockets. 这里才是真正的读写
    this.client.poll(pollTimeout, now);
}

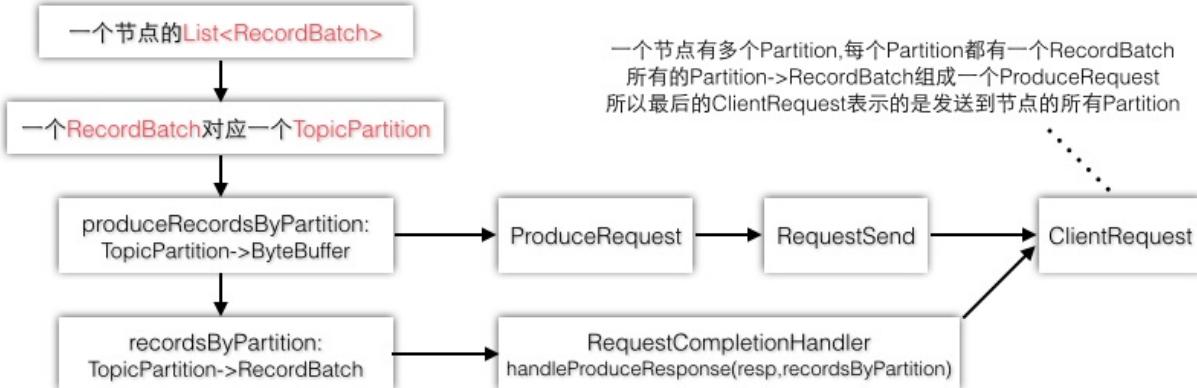
```



accumulator在之前一直append数据,到真正要发送一批数据时,先①准备(ready)需要发送的partitions到哪些Nodes上,②并建立到节点的连接
然后③构造每个Node需要的RecordBatch列表(一个节点同时可以接受多批数据),④并转换为客户端的请求ClientRequest.



由于batches已经是按照节点划分好的了,所以创建的客户端请求也是按照节点划分好了.不过虽然produceRequest方法中的batches是某个节点所有的batches,但是客户端请求面向的还是Partition级别! 所以要对batches重新按照Partition的粒度整理.
不过注意的是一个节点只有一个ClientRequest,它本身并不关心包含了多少个Partition,你只要需要发送的对象包装成RequestSend即可.



问题: batches中会不会有相同的Partition? 不会的! 如果那样的话,两个Map的key因为都是Partition,会导致value被覆盖.但是怎么保证?

```

private ClientRequest produceRequest(long now, int destination,
    Map<TopicPartition, ByteBuffer> produceRecordsByPartition =
        final Map<TopicPartition, RecordBatch> recordsByPartition =
            for (RecordBatch batch : batches) {
                TopicPartition tp = batch.topicPartition;
                produceRecordsByPartition.put(tp, batch.records.buffer());
                recordsByPartition.put(tp, batch);
            }
        // 构造生产者的请求(每个Partition都有生产记录), 并指定目标节点, 请求
        ProduceRequest request = new ProduceRequest(acks, timeout,
            RequestSend send = new RequestSend(Integer.toString(destination),
                callback);
        // 回调函数会作为客户端请求的一个成员变量, 当客户端请求完成后, 会触发
        RequestCompletionHandler callback = new RequestCompletionHandler() {
            public void onComplete(ClientResponse response) {
                handleProduceResponse(response, recordsByPartition,
                    destination);
            }
        };
        return new ClientRequest(now, acks != 0, send, callback);
    }

```

ProduceRequest是Producer的生产请求,需要acks和timeout这两个参数,在后面的DelayedOperation中会用到.

ClientRequest & ClientResponse & Callback

ClientRequest是客户端的请求,这个请求会被发送(Send)到服务器上,所以它包装的是RequestSend

ClientResponse是客户端的响应,也需要ClientRequest是因为请求有返回值时响应要和请求对的上.

由于Callback是附加在Request里的,为了让Response能够触发Callback回调,将Request设置到Response.

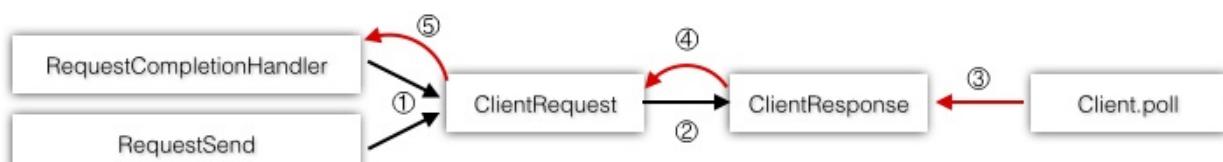
```
// A request being sent to the server. This holds both the network
public final class ClientRequest {
    private final long createdTimeMs;
    private final boolean expectResponse;
    private final RequestSend request;
    private final RequestCompletionHandler callback;
    private final boolean isInitiatedByNetworkClient;
    private long sendTimeMs;
}

// A response from the server. Contains both the body of the response
public class ClientResponse {
    private final long receivedTimeMs;
    private final boolean disconnected;
    private final ClientRequest request;
    private final Struct responseBody;
}
```

回调函数传给了ClientRequest客户端请求,当客户端真正发生读写后(poll),会产生ClientResponse对象,触发回调函数的执行.

因为回调对象RequestCompletionHandler的回调方法onComplete的参数是ClientResponse.

NetworkClient.poll是真正发生读写的地方,所以它也会负责生成客户端的响应信息.



```
public List<ClientResponse> poll(long timeout, long now) {  
    // .....真正的读写操作，会生成responses  
  
    // invoke callbacks  
    for (ClientResponse response : responses) {  
        if (response.request().hasCallback()) {  
            response.request().callback().onComplete(response);  
        }  
    }  
    return responses;  
}
```

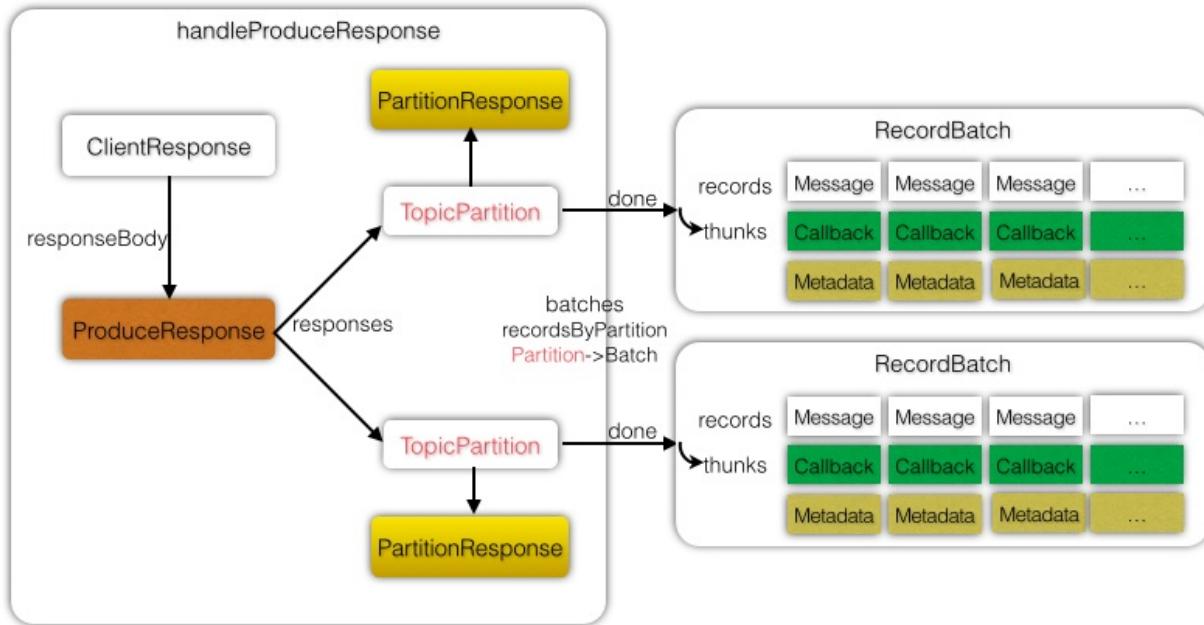
到poll出来的responses是一个列表(客户端请求也是一个列表).每个ClientRequest和ClientResponse都是针对一个节点的.

再次强调下,发生在Client级别的动作都是针对每个节点而言,至于底层每个节点分成多个Partition则需要自己把内容封装进去.

```
for (ClientRequest request : requests)  
    client.send(request, now);
```

handleProduceResponse

每个ClientResponse代表的是一个节点的响应,要从中解析出ProduceResponse中所有Partition的PartitionResponse.



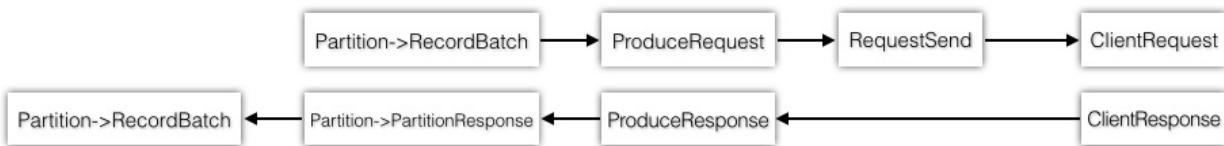
```

private void handleProduceResponse(ClientResponse response, Map<TopicPartition, ProduceResponse> batches) {
    if (response.hasResponse()) { // if we have a response
        ProduceResponse produceResponse = new ProduceResponse(response);
        for (Map.Entry<TopicPartition, ProduceResponse.PartitionResponse> entry : batches.entrySet()) {
            TopicPartition tp = entry.getKey(); // 每一个分区
            ProduceResponse.PartitionResponse partResp = entry.getValue();
            Errors error = Errors.forCode(partResp.errorCode);
            RecordBatch batch = batches.get(tp); // 因为是单线程，所以直接从batches中拿
            completeBatch(batch, error, partResp.baseOffset, partResp.offset);
        }
    } else { // this is the acks = 0 case, just complete all batches
        for (RecordBatch batch : batches.values()) completeBatch(batch, Errors.NONE);
    }
}

private void completeBatch(RecordBatch batch, Errors error, long baseOffset) {
    batch.done(baseOffset, error.exception()); // tell the user
    this.accumulator.deallocate(batch); // release resources
}

```

这里的ClientRequest和ClientResponse分别由ProduceRequest和ProduceResponse组成,两者有一定的共同点.



客户端组织生成的每一批Batch记录都属于一个Partition,所以每个Batch都要complete(调用RecordBatch.done).

每次Batch中,如果客户端不需要响应,则baseOffset=-1,否则从response中解析出baseOffset用来表示消息的offset.

由于RecordBatch记录的是每一条消息,每条消息都有Callback的话,一个Batch里就有和消息数量相等的thunks(callback).

```

public void done(long baseOffset, RuntimeException exception) {
    // execute callbacks
    for (int i = 0; i < this.thunks.size(); i++) {
        Thunk thunk = this.thunks.get(i);
        if (exception == null) {
            RecordMetadata metadata = new RecordMetadata(this.t
                thunk.callback.onCompletion(metadata, null);
        } else {
            thunk.callback.onCompletion(null, exception);
        }
    }
    this.produceFuture.done(topicPartition, baseOffset, excepti
}
    
```

客户端追加消息时附属的Callback终于在这里出现了.我们看到了很熟悉的RecordMetadata对象作为onCompletion的回调参数

```

new Callback() {
    public void onCompletion(RecordMetadata metadata, Exception
        System.out.println("The offset of the record we just se
    }
}
    
```

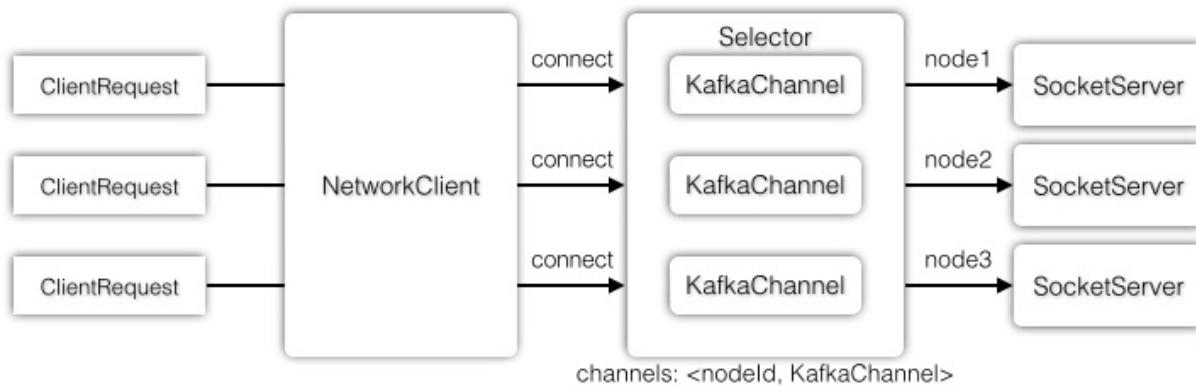
Selector

NetworkClient的请求一般都是交给Selector去完成的. Selector使用NIO异步非阻塞模式管理连接,读写请求.

Selector用一个单独的线程就可以管理多个网络连接的channel,并能够知晓通道是否为读写事件做好准备.

connect连接

客户端在和节点连接的时候,会创建和服务端的SocketChannel连接通道.Selector维护了每个目标节点对应的KafkaChannel.



```

public void connect(String id, InetSocketAddress address, int s)
    SocketChannel socketChannel = SocketChannel.open();
    socketChannel.configureBlocking(false);
    Socket socket = socketChannel.socket();           // 这是客户端, 所以
    socketChannel.connect(address);                  // 连接服务端, 注意
    SelectionKey key = socketChannel.register(nioSelector, Selec
    KafkaChannel channel = channelBuilder.buildChannel(id, key,
    key.attach(channel);                           // 将KafkaChann
    this.channels.put(id, channel);                // Selector维护
}
    
```

因为是非阻塞模式,此时调用connect()可能在连接建立之前就返回了.为了确定连接是否建立,需要再调用finishConnect()确认完全连接上了.

KafkaChannel finishConnect

finishConnect会作为key.isConnectable的处理方法.在确认连接后可以取消Connect事件,并添加READ事件.

为什么在成功连接之后就注册了READ,首先只有成功连接,才可以进行读写操作.对于客户端的读一般都是读取响应结果.

而什么时候响应结果返回给客户端是不确定的,所以不能在发送请求的时候注册读,因为有些发送请求并不需要读取结果的.

```
public void finishConnect() throws IOException {
    socketChannel.finishConnect();
    key.interestOps(key.interestOps() & ~SelectionKey.OP_CONNECT);
}
```

send发送

客户端发送的每个Send请求会被用到一个KafkaChannel中.如果一个KafkaChannel上还有未发送成功的Send请求.

则后面的请求不允许发送.也就是说客户端发送请求给服务端,在一个KafkaChannel中,一次只能发送一个Send请求.

```
public void send(Send send) {                                // NetworkClient
    KafkaChannel channel = channelOrFail(send.destination());
    channel.setSend(send);                                     // 设置当前发送的
}
private KafkaChannel channelOrFail(String id) {
    return this.channels.get(id);
}
```

KafkaChannel.setSend

客户端的请求Send设置到KafkaChannel中,KafkaChannel的TransportLayer会为SelectionKey注册WRITE事件.

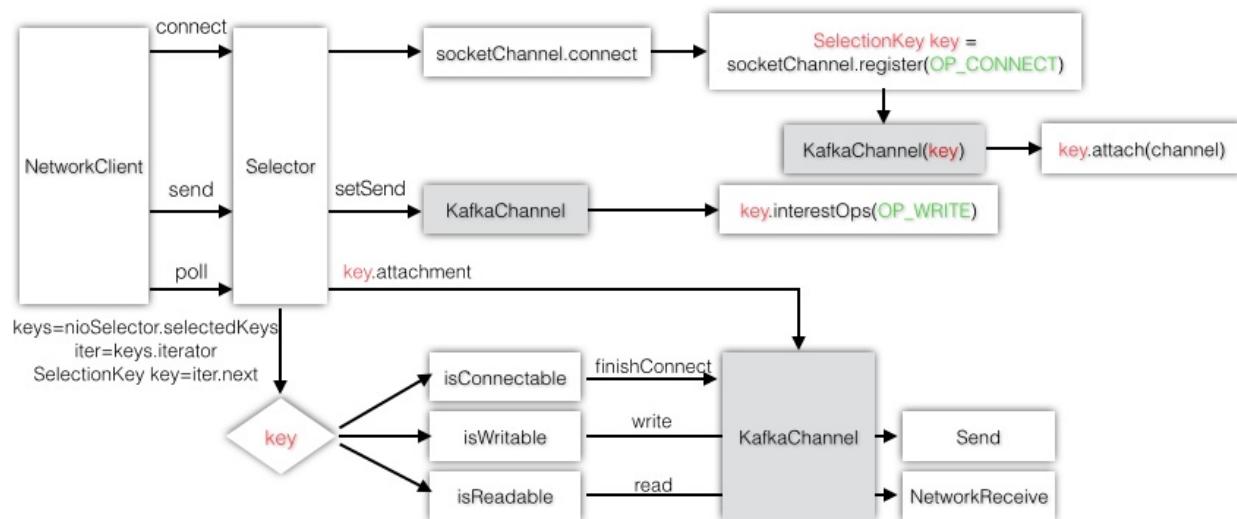
Channel的SelectionKey有了Connect和Write事件,在Selector的轮询过程中当发现这些事件到来,就开始执行真正的操作.

虽然一个KafkaChannel一次只能处理一个Send请求,每次Send时都要添加WRITE事件,当Send发送成功后,就要取消掉WRITE.下一个Send请求事件进来时,继续添加WRITE,然后在请求发送成功后,又取消WRITE.
因为KafkaChannel是由请求事件驱动的.如果没有请求就不需要监听WRITE,KafkaChannel就不需要做写操作.
基本流程就是:开始发送一个Send请求->注册OP_WRITE->发送请求...->Send请求发送完成->取消OP_WRITE

```
public void setSend(Send send) {
    // 如果存在send请求,说明之前的Send请求还没有发送完毕,新的请求不能进
    if (this.send != null) throw new IllegalStateException("At1");
    this.send = send;
    this.transportLayer.addInterestOps(SelectionKey.OP_WRITE);
}

// 这是KafkaChannel的transportLayer的方法, transportLayer的key来自
public void addInterestOps(int ops) {
    key.interestOps(key.interestOps() | ops);
}
```

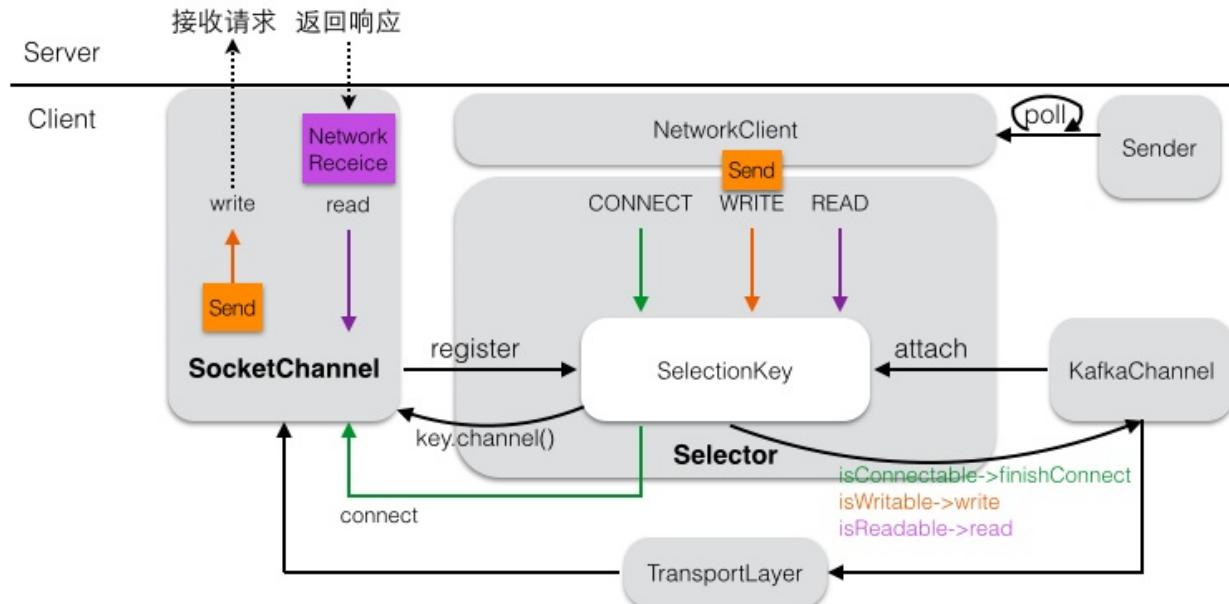
现在对于客户端而言,连接,读,写事件都有了(CONNECT,READ,WRITE).在selector的轮询中可以操作读写事件.



poll轮询

轮询的策略是如果有数据(timeout=0)直接调用nioSelector.selectNow,否则每隔一定时间触发一次select调用.

绑定到SelectionKey上的是KafkaChannel,基于Kafka的传输层 TransportLayer包含了IO层的SocketChannel.



poll轮询.一次轮询调用是不断地while循环.当然它的条件是能够选择到感兴趣的SelectionKey集合.

在得到SelectionKey后,获取其中的KafkaChannel,因为Channel上有事件发生,然后选择对应的操作.

在一开始时,我们为SocketChannel注册了某类SelectionKey,并绑定KafkaChannel到key上.

当SocketChannel上有读写事件时,SelectionKey会被触发,就能取到绑定的KafkaChannel.

```

public void poll(long timeout) throws IOException {
    clear();
    if (hasStagedReceives()) timeout = 0;
    int readyKeys = select(timeout); //选择器, 触发事件
    if (readyKeys > 0) {
        Set<SelectionKey> keys = this.nioSelector.selectedKeys();
        Iterator<SelectionKey> iter = keys.iterator();
        while (iter.hasNext()) {
            SelectionKey key = iter.next();
            iter.remove();
            KafkaChannel channel = channel(key); //获得绑定到key的channel
            /* complete any connections that have finished the handshake */
            if (key.isConnectable()) channel.finishConnect();
            /* if channel is not ready finish prepare */
            if (channel.isConnected() && !channel.ready()) channel.finishPrepare();
            /* if channel is ready read from any connections that have finished the handshake */
            if (channel.ready() && key.isReadable() && !hasStagedReceives()) {
                NetworkReceive networkReceive;
                while ((networkReceive = channel.read()) != null) {
                    addToStagedReceives(channel, networkReceive);
                }
            }
            /* if channel is ready write to any sockets that have finished the handshake */
            if (channel.ready() && key.isWritable()) {
                Send send = channel.write();
                if (send != null) this.completedSends.add(send);
            }
            /* cancel any defunct sockets */
            if (!key.isValid()) close(channel);
        }
    }
    addToCompletedReceives(); // 没有新的SelectionKey了! 说明要读取消息
    //不过Selector.poll->NetworkClient.poll->Sender.run是在这里循环的
}

```

KafkaChannel write

写操作 的事件没有使用while循环来控制,而是在完成发送时取消掉Write事件.如果Send在一次write调用时没有写完,

SelectionKey的OP_WRITE事件没有取消,下次isWritable事件会继续触发,直到整个Send请求发送完毕才取消.

所以发送一个完整的Send请求是通过最外层的while(iter.hasNext),即SelectionKey控制的.

```

public Send write() throws IOException {
    Send result = null;
    if (send != null && send(send)) { //如果send方法返回值为false
        result = send;
        send = null; //发送完毕后,设置send=null
    }
    return result;
}
private boolean send(Send send) throws IOException {
    send.writeTo(transportLayer); //transportLayer有Socket
    if (send.completed()) //只有Send请求全部写出去了
        transportLayer.removeInterestOps(SelectionKey.OP_WRITE);
    return send.completed(); //如果Send只发送了一点点,返回false
}

```

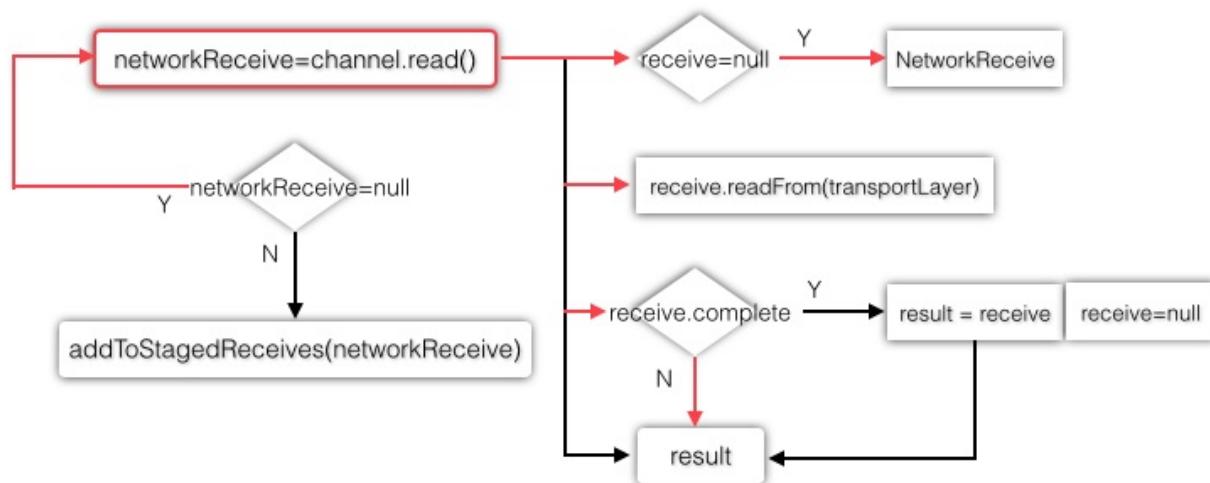


KafkaChannel read

读取操作 需要读取一个完整的NetworkReceive. 初始时receive=null,创建一个空的NetworkReceive.

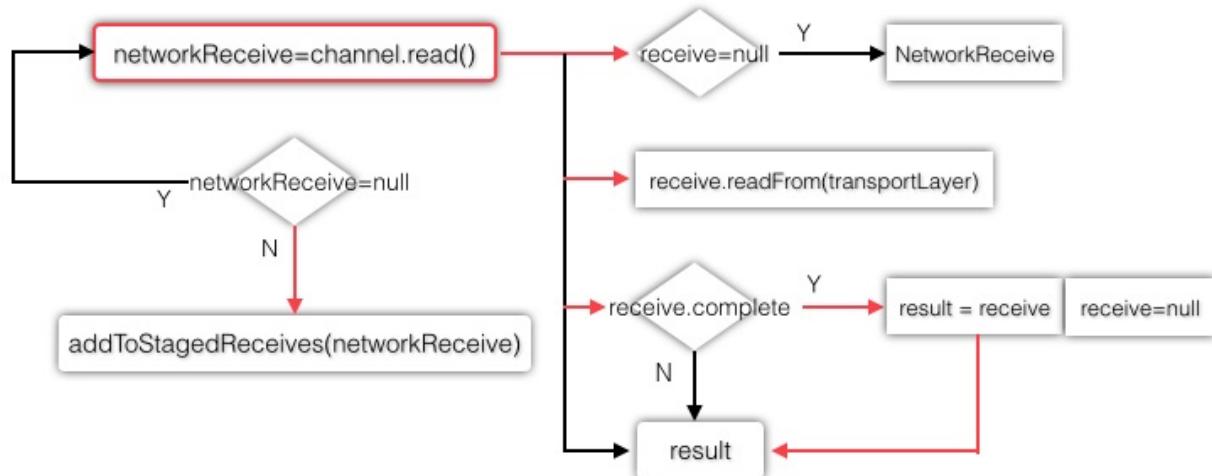
receive(receive)方法从transportLayer中读取到NetworkReceive对象中. 假设读了一次没有读完,

即receive.complete=false,read()返回的result=null,导致poll中的while循环继续调用read().



第二次读取时, $receive \neq null$, 继续从 $transportLayer$ 读取到 $receive$ 对象中, 这次成功地读完了, 设置 $result$

为读取成功的 $receive$ NetworkReceive, 这个 $result$ 不为空, while 循环结束, 调用 $addToStagedReceives$.



```

public NetworkReceive read() throws IOException {
    NetworkReceive result = null;
    if (receive == null) receive = new NetworkReceive(maxReceiveSize);
    receive(receive);
    if (receive.complete()) {
        receive.payload().rewind();
        result = receive;
        receive = null;
    }
    return result;
}
private long receive(NetworkReceive receive) throws IOException {
    return receive.readFrom(transportLayer);
}

```

比较读和写在poll中的处理方式.一旦有读操作,就要读取一个完整的NetworkReceive,如果是写,可以分多次写.

即读操作会在一次SelectionKey循环读取一个完整的接收动作,而写操作会在多次SelectionKey中完成一个完整的发送动作.

写完后(成功地发送了Send请求),会取消掉WRITE事件(本次写已完成). 而读完并没有取消掉READ事件(可能还要读新的数据).

complete sends and receives

写操作会将当前发送成功的Send加入到 completedSends . 因为每次写请求在每个通道中只会有一个.

读操作先加到stagedReceives,最后全部读取完之后才从stagedReceives复制到 completedReceives .

completedSends和completedReceives分别表示在Selector端已经发送的和接收到的请求.

它们会在NetworkClient的poll调用之后被不同的handleCompleteXXX使用.

```

// 一次读操作就会有一个NetworkReceive生成，并加入到channel对应的队列中
private void addToStagedReceives(KafkaChannel channel, NetworkReceive receive) {
    if (!stagedReceives.containsKey(channel))
        stagedReceives.put(channel, new ArrayDeque<NetworkReceive>());
    Deque<NetworkReceive> deque = stagedReceives.get(channel);
    deque.add(receive);
}

// 只有在本次轮询中没有读操作了(也没有写了)，在退出轮询时，将上一步的所有
private void addToCompletedReceives() {
    if (this.stagedReceives.size() > 0) {
        Iterator<Map.Entry<KafkaChannel, Deque<NetworkReceive>>> iter =
        KafkaChannel channel = entry.getKey();
        if (!channel.isMute()) { // 当前通道上没有OP_READ事件
            Deque<NetworkReceive> deque = entry.getValue();
            NetworkReceive networkReceive = deque.poll();
            this.completedReceives.add(networkReceive);
            if (deque.size() == 0) iter.remove();
        }
    }
}

```

问题1：最后加入到completedReceives的条件是channel.isMute()=false,即通道上没有OP_READ事件时.但是什么时候会取消READ呢?

```

public boolean isMute() {
    return key.isValid() && (key.interestOps() & SelectionKey.OP_READ) == 0;
}

```

问题2：deque.poll()是弹出一个元素,对于一个channel有多个元素,在while循环的是channel级别,如何弹出某个channel的所有元素?

NetworkClient

send动作将ClientRequest添加到队列 `inFlightRequests` 用来缓冲请求,然后触发 Selector->KafkaChannel->transportLayer添加 `OP_WRITE` 写事件通知poll动作也将实际处理交给Selector,客户端服务于读和写,即发送和接收.

Selector在每次轮询调用之后,都会触发读写请求的完成handler,并添加到responses,用于回调函数的参数.

不管是 `Send` 发送请求还是 `NetworkReceive` 接收请求,都可以被转换为 `ClientRequest` 表示客户端的请求.

```

public void send(ClientRequest request, long now) {
    String nodeId = request.request().destination();
    if (!canSendRequest(nodeId)) throw new IllegalStateException();
    doSend(request, now);
}

private void doSend(ClientRequest request, long now) {
    this.inFlightRequests.add(request); //还没开始真
    selector.send(request.request()); //标记下收到
}

public List<ClientResponse> poll(long timeout, long now) {
    // ① Selector轮询, 真正读写发生的地方. 如果客户端请求被完整地处理过
    this.selector.poll(Utils.min(timeout, metadataTimeout, requestTimeout));
    // ② process completed actions 处理已经完成的动作,如果没有收到写入
    List<ClientResponse> responses = new ArrayList<>();
    handleCompletedSends(responses, updatedNow); //完成发送的
    handleCompletedReceives(responses, updatedNow); //完成接收的
    handleDisconnections(responses, updatedNow); //断开连接的
    handleConnections(); //处理连接的
    handleTimedOutRequests(responses, updatedNow); //超时请求的
    // ③ invoke callbacks 将responses用于触发回调函数的调用
    return responses;
}

```

ready

Sender的run中,在drain produce requests前会先判断readyNodes是否已经准备好了,因为不会发送请求给没有准备好的节点.

```
Iterator<Node> iter = result.readyNodes.iterator();
long notReadyTimeout = Long.MAX_VALUE;
while (iter.hasNext()) {
    Node node = iter.next();
    if (!this.client.ready(node, now)) {
        iter.remove();
        notReadyTimeout = Math.min(notReadyTimeout, this.c...
```

如果isReady返回false,会先初始化连接initiateConnect,这里通过Selector向远程节点发起连接.

ready指的是已经建立连接,并且通道也准备好,也允许往inFlightRequests发送更多的请求.

如果是之前没有连接,现在刚刚发起连接请求,则不算准备好,因为连接动作肯定需要一定时间.

```

// Begin connecting to the given node, return true if we are already connected
public boolean ready(Node node, long now) {
    if (isReady(node, now)) return true;
    if (connectionStates.canConnect(node.idString(), now))
        initiateConnect(node, now); // if we are interested in this connection
    return false;
}

// Initiate a connection to the given node
private void initiateConnect(Node node, long now) {
    String nodeConnectionId = node.idString();
    this.connectionStates.connecting(nodeConnectionId, now);
    selector.connect(nodeConnectionId, new InetSocketAddress(node.host(), node.port()));
}

// Check if the node with the given id is ready to send more requests
public boolean isReady(Node node, long now) {
    return !metadataUpdater.isUpdateDue(now) && canSendRequest(node);
}

// Are we connected and ready and able to send more requests to this node
private boolean canSendRequest(String node) {
    return connectionStates.isConnected(node) && selector.isChosen(node);
}

```

注意可以往某个节点发送请求的最后一个条件：队列为空，或者 队列的第一个请求必须已经完成。

如果有客户端的上一次请求没有完成，说明这个节点有正在处理的请求，比较忙，则不允许发送给它。

```

public boolean canSendMore(String node) {
    Deque<ClientRequest> queue = requests.get(node);
    return queue == null || queue.isEmpty() ||
           (queue.peekFirst().request().completed() && queue.size() < maxConcurrentRequests);
}

```

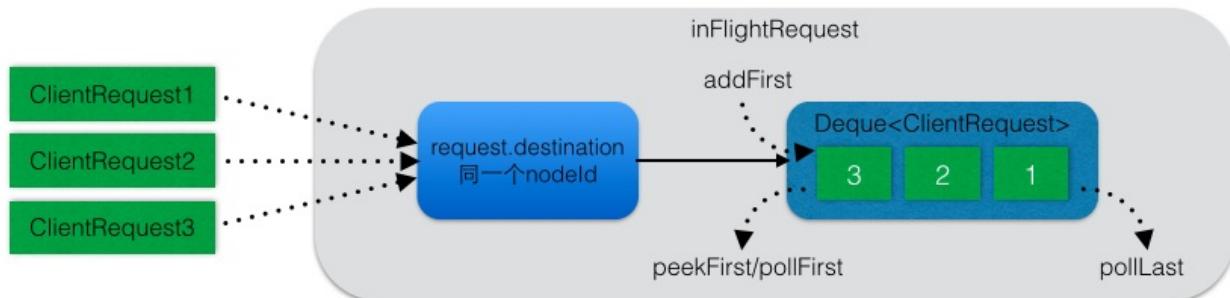
我们还要继续看下什么条件才算是completed,对于ByteBufferSend是没有要发送的数据了,也没有正在写的数据.

所以这里的请求完成指的是 上一个发送请求 已经 成功发送 到服务端了,但并 不需要 等待那个请求收到响应结果 .

```
public boolean completed() {
    return remaining <= 0 && !pending;
}
```

inFlightRequests.add

表示已经发送,或正在发送.并且还没有收到响应的(客户端)请求.请求首先加入到(目标节点对应的)队列中.



注意:上面ready的限制条件,则最开始的两个请求一定已经发送成功,否则请求3不会被添加到Deque中.

所以如果第一个请求发送到某个节点迟迟不能完成,有可能那个节点网络有问题,则后面的请求不会发送过来.

这就避免了因为网络阻塞,请求一直堆积在某个节点上.

使用队列虽然可以存储多个请求,但是 新的请求 能加进来的条件是 上一个请求必须 已经发送成功 !

```

public void add(ClientRequest request) {
    Deque<ClientRequest> reqs = this.requests.get(request.request().destination());
    if (reqs == null) {
        reqs = new ArrayDeque<>();
        this.requests.put(request.request().destination(), reqs);
    }
    reqs.addFirst(request);           // 新的请求总是加到队列的头部.
}

```

requests Map的key是request.request().destination(),表示这个请求要发送到哪个Broker节点上.

所以从这里也可以看出,现在的作用域都只是在客户端,因为只有客户端才有目标节点destination.

如果是Kafka作为服务端(目标节点),客户端连接服务端,就可以通过SocketChannel和服务端通信.

client-server mode

注意doSend不只是用于Producer发送,也可以用于Consumer消费.参数ClientRequest表示客户端的请求.

对于Kafka而言,P和C都是客户端.客户端发送请求给Kafka,从Kafka这方面来说,都要Receive读取请求.

实际上KafkaProducer和KafkaConsumer都有NetworkClient,说明客户端是嵌入在P和C里面的.

Send请求的destination, NetworkReceive的source都表示远程的Kafka节点.

因为在P和C的一亩三分地里, NetworkClient是它们和远程服务器交互的中间介质.

客户端和服务端只会有一个通道进行通信,所以客户端和服务端的交互只有两种形式:发送请求和读取响应.

Selector以及请求对象都只是NetworkClient和SocketServer之间互相通信上的一些介质.

只不过Selector提供了NIO非阻塞IO, 而请求对象(Send,NetworkReceive)是通信的数据.

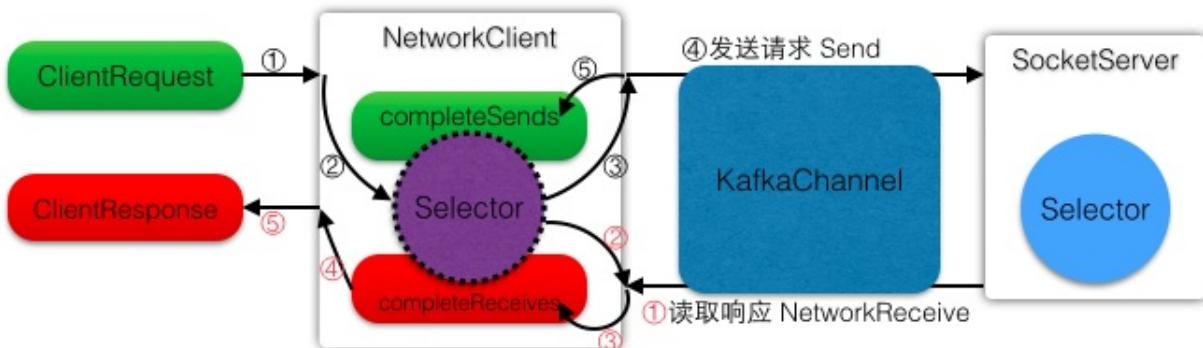
不要认为NetworkClient和Selector之间也有一层通道.通道只有跨机器(客户端和服务器)才会建立的连接.



还是以客户端发送请求为例,当NetworkClient把一个完整的请求发送到服务器,静静地等待服务端处理.

然后某个时刻服务端对这个请求的处理结果要发送给客户端, NetworkClient看到有响应消息过来了,不等静静地了,

一点一点地接收响应,直到把响应结果完整地接收下来, 发送请求是Send, 响应结果就是NetworkReceive.



不过有可能服务端发送完请求之后并不想知道这个请求的结果,那么它发送完就不管了. 和有响应结果的不同点是

ClientResponse的最后一个参数表示响应内容body,在不期望得到响应时为null,有响应时为receive.payload.

比如CS模式中的set和get请求. set更改后并不需要知道响应,而get请求需要服务端返回一批数据给客户端.

注意:NetworkClient在发送完set/get请求后,就会调用handleCompletedSends,表示请求已经发送到服务端了.

至于请求在服务端被处理,什么时候完成完全取决于服务端,当get请求收到响应时,才调用handleCompletedReceives.

complete handler

客户端发送请求后,handleCompletedSends中对于有响应的请求,并不会将ClientRequest从inFlightRequests中移除.

因为inFlightRequests表示的是还没有收到响应的客户端请求.而现在才发完请求,肯定还没收到响应,所以不会移除.

而如果是客户端请求不需要响应,则这时候是可以将ClientRequest从中删除,添加时放到头部,删除时也是从头部删除.

问:删除时也是从头部删除,如果第一个请求发送完毕,刚好来了第二个请求,这时候删除头部是删除哪个请求?

答1(x):前面一个请求发送完,如果它还没有完成,则第二个请求是不会进入到这个节点的队列中的.

只有前一个请求已经完成,下一个请求才可以进入到队列.确保下面的删除是删除已经完成的,而不是新进来刚开始的请求.

```
// Handle any completed request send. In particular if no response is expected
private void handleCompletedSends(List<ClientResponse> responses) {
    // if no response is expected then when the send is completed
    for (Send send : this.selector.completedSends()) {
        // 这里获取目标节点的队列中第一个请求, 但并没有从队列中删除, 取出
        ClientRequest request = this.inFlightRequests.lastSent();
        // 如果不需要响应, 当Send请求完成时, 就直接返回. 不过还是有ClientResponse
        if (!request.expectResponse()) {
            this.inFlightRequests.completeLastSent(send.destination());
            responses.add(new ClientResponse(request, now, false));
        }
        // 如果客户端请求需要有响应, 那么它的响应是在下面的handleCompletedReceives方法中处理
    }
}
```

收到响应是在handleCompletedReceives,这时候才可以调用completeNext删除source对应的ClientRequest,

因为我们知道inFlightRequests存的是未收到请求的ClientRequest,现在这个请求已经有响应了,就不需要再其中保存了.

问:inFlightRequests在这里的作用是什么?首先每个节点都有一个正在进行中的请求队列,可以防止请求堆积(流控?).

当请求发送成功,还没有收到响应(对于需要响应的客户端请求而言)的这段时间里,ClientRequest是处于in-flight状态的.

同时每个节点的队列有个限制条件是:上一个请求没有发送完毕,下一个请求不能进来.或者队列中的未完成的请求数很多时都会限制.

不需要响应的流程:开始发送请求->添加到inFlightRequests->发送请求...->请求发送成功->从inFlightRequests删除请求

需要响应的流程:开始发送请求->添加到inFlightRequests->发送请求...->请求发送成功->等待接收响应->接收到响应->删除请求

```
// Handle any completed receives and update the response list
private void handleCompletedReceives(List<ClientResponse> responses) {
    for (NetworkReceive receive : this.selector.completedReceives()) {
        String source = receive.source();
        // 接收到完整的响应了, 现在可以删除inFlightRequests中的ClientRequest
        ClientRequest req = inFlightRequests.completeNext(source);
        ResponseHeader header = ResponseHeader.parse(receive.payload());
        // Always expect the response version id to be the same as the request
        short apiKey = req.request().header().apiKey();
        short apiVer = req.request().header().apiVersion();
        Struct body = ProtoUtils.responseSchema(apiKey, apiVer)
            .correlate(req.request().header(), header);
        if (!metadataUpdater.maybeHandleCompletedReceive(req, header))
            responses.add(new ClientResponse(req, now, false, body));
    }
}
```

InFlightRequests complete

Deque是个双端队列,可以往头和尾方便地添加/删除/获取ClientRequest(前面Partition的RecordBatch也用过Deque).

```
//Get the oldest request (the one that will be completed first)
public ClientRequest completeNext(String node) {
    return requestQueue(node).pollLast();
}

// Get the last request we sent to the given node (but don't remove it)
public ClientRequest lastSent(String node) {
    return requestQueue(node).peekFirst();
}

// Complete the last request that was sent to a particular node
public ClientRequest completeLastSent(String node) {
    return requestQueue(node).pollFirst();
}
```

Question

I'm reading new client design of version 0.9. and I has a question of inFlightRequests in and out.

Here is the basic flow :

When Sender send a ClientRequest to NetworkClient, it add to inFlightRequests indicator in-flight requests

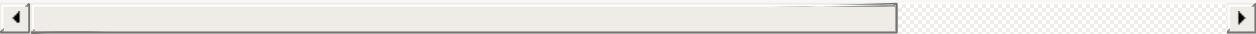
```
private void doSend(ClientRequest request, long now) {
    this.inFlightRequests.add(request);
    selector.send(request.request());
}
```

the inFlightRequests map node to deque. the new request add as first element of deque

```

public void add(ClientRequest request) {
    Deque<ClientRequest> reqs = this.requests.get(request.request().destination());
    if (reqs == null) {
        reqs = new ArrayDeque<>();
        this.requests.put(request.request().destination(), reqs);
    }
    reqs.addFirst(request);
}

```



then poll happen on client and then selector, after success send this ClientRequest, the send will add to selector's completedSends

```

private void handleCompletedSends(List<ClientResponse> responses) {
    // if no response is expected then when the send is completed
    for (Send send : this.selector.completedSends()) {
        ClientRequest request = this.inFlightRequests.lastSent();
        if (!request.expectResponse()) {
            this.inFlightRequests.completeLastSent(send.destination());
            responses.add(new ClientResponse(request, now, false));
        }
    }
}

```



if this request does't need response, the ClientRequest will remove from inFlightRequest

```

public ClientRequest completeLastSent(String node) {
    return requestQueue(node).pollFirst();
}

```

I'm curios why poll First? A scene like this: after the first ClientRequest sended out success,

and not yet execute to handleCompletedSends, another ClientRequest coming,

and the new request addFirst to deque.

then pollFirst execute, as first element of deque now become to the new request, pollFirst will delete the new one, not old one.



I has also check `NetworkClient.send->canSendRequest ->`
`inFlightRequests.canSendMore(node) ->`
`queue.peekFirst().request().completed()`
only the first element of deque finish, then new request can send to the same
node. but the condition of completed
by `ByteBufferSend` is `remaining <= 0 && !pending`. which means If Send
sended success to server, it's completed!

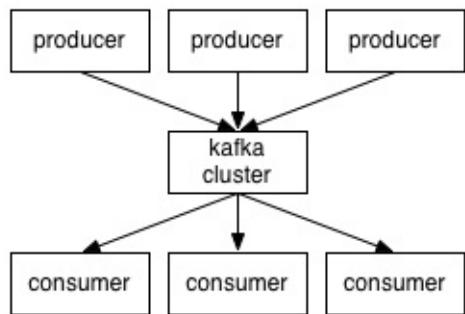
Am I missig something(are there any other limitation)? Can some one point out.
Tks.

KafkaServer

上面的Producer和Consumer都不是作为Kafka的内置服务,而是一种客户端(所以它们都在clients包).

客户端可以独立于Kafka,和Kafka服务所在的节点互相隔离. Producer和Consumer和Kafka集群进行交互.

每个Kafka节点都有一些自己内置的服务进程,比如Broker,KafkaController,GroupCoordinator,ReplicaManager



接下来我们看下客户端的请求在服务端是怎么被处理的. 客户端有发送和接收请求, 服务端同样也有接收和发送的逻辑.

因为对于I/O来说是双向的:客户端发送请求,就意味着服务端要接收请求,同样服务端也会发送响应,客户端就要接收响应.

Ref

- <http://blog.csdn.net/lizhitao/article/details/39499283>
- <http://ifeve.com/socket-channel/>

Kafka的Producer Scala客户端API实现

Scala OldProducer

scala版本的生产者发送消息示例: 面向KeyedMessage,指定了topic和消息内容.

```
Properties props = new Properties();
props.put("metadata.broker.list", "192.168.4.31:9092");
props.put("serializer.class", "kafka.serializer.StringEncoder");
Producer<Integer, String> producer = new Producer<Integer, String>(props);
KeyedMessage<Integer, String> data = new KeyedMessage<Integer, String>(1, "Hello World");
producer.send(data);
producer.close();
```

问:core下也有Producer和Consumer.scala, 和clients中的KafkaProducer,KafkaConsumer有什么区别?

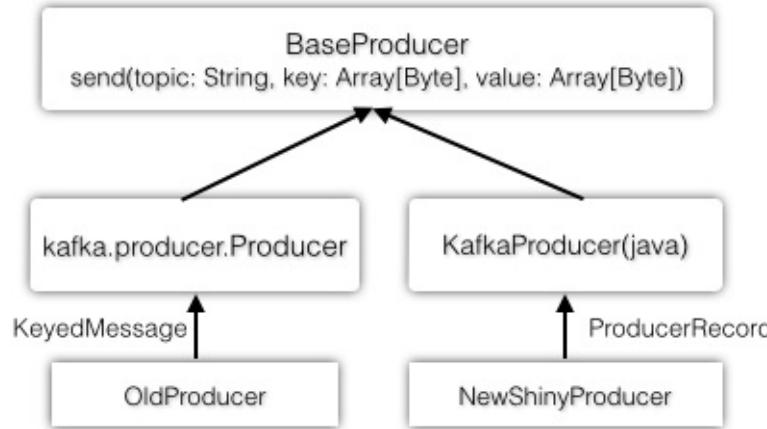
答:从ConsoleProducer看到两种不同的实现:OldProducer->scala的Producer,NewShinyProducer->KafkaProducer

```
val producer = if(config.useOldProducer) {
    new OldProducer(getOldProducerProps(config))
} else {
    new NewShinyProducer(getNewProducerProps(config))
}
```

旧的Producer消息用KeyedMessage,新的用ProducerRecord.不同的Producer实现,用trait定义共同的send接口.

因为消息最终是以字节的形式存储在日志文件中的,所以字节数组的key和value可以作为两种不同实现的共同存储结构.

```
trait BaseProducer {
    def send(topic: String, key: Array[Byte], value: Array[Byte])
    def close()
}
```



用scala实现的Producer构造方式是一样的,需要指定分区方式,Key,Value的序列化.
如果是异步还有一个发送线程.

在java版本中用了RecordAccumulator,RecordBatch,MemoryRecords等来缓存一批消息,scala版本用简单的队列来缓存.

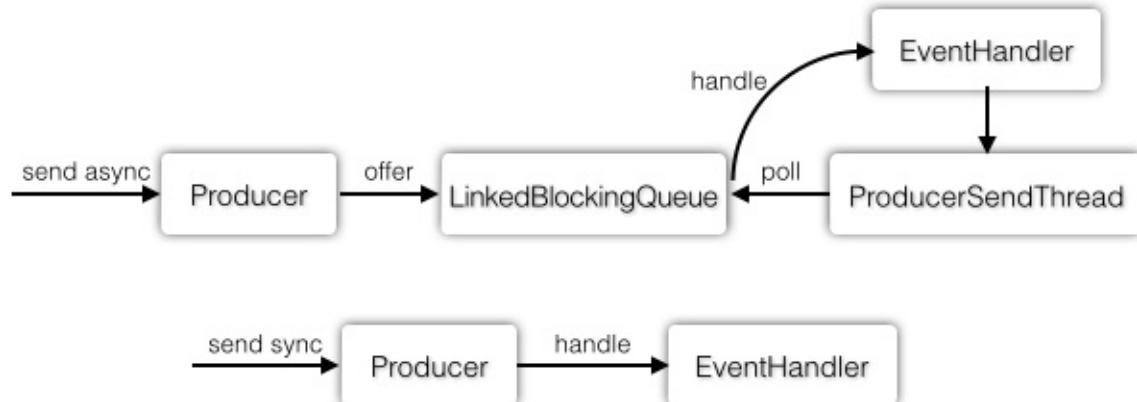
```

class Producer[K,V](val config: ProducerConfig, private val eventHandler: EventHandler[K,V])
  extends BaseProducer with ProducerLike[K,V] {
  private val queue = new LinkedBlockingQueue[KeyedMessage[K,V]](config.acks)
  private var producerSendThread: ProducerSendThread[K,V] = null
  config.producerType match {
    case "sync" =>
    case "async" =>
      sync = false
      producerSendThread = new ProducerSendThread[K,V]("ProducerSendThread", queue, eventHandler)
      producerSendThread.start()
  }

  def this(config: ProducerConfig) = this(config, new DefaultEventHandler[K,V])
    CoreUtils.createObject[Partitioner[K,V]](config.partitionStrategy)
    CoreUtils.createObject[Encoder[K,V]](config.keySerializer)
    CoreUtils.createObject[Encoder[V]](config.valueSerializer)
    new ProducerPool(config))
}
  
```

注意ProducerSendThread线程处理消息也是通过事件处理器eventHandler的,
当然少不了阻塞队列queue.

同步发送消息直接调用事件处理器, 异步发送消息则会加入到阻塞队列
BlockingQueue,
通过后台ProducerSendThread线程完成异步发送, 类似于java版本的Sender线程



```

def send(messages: KeyedMessage[K, V]*) {
    lock synchronized {
        sync match {
            case true => eventHandler.handle(messages)
            case false => asyncSend(messages)
        }
    }
}

```

上面的send方法其实就是最开始的Producer示例了,参数用 * 表示可以发送多条KeyedMessage消息.

异步发送就是将客户端代码中传入的消息messages转存到Producer的队列中.
然后ProducerSendThread在Producer中被启动了,就可以从队列中消费消息,完成消息的发送动作.

```

private def asyncSend(messages: Seq[KeyedMessage[K,V]]) {
    for (message <- messages) {
        config.queueEnqueueTimeoutMs match {
            case 0 => queue.offer(message)
            case _ => config.queueEnqueueTimeoutMs < 0 match {
                case true => queue.put(message); true
                case _ => queue.offer(message, config.queueEnqueueTimeo
            }
        }
    }
}

```

在async方式中，将产生的数据放入queue时有三种不同的放入方式：

- 当queue.enqueue.timeout.ms=0，则立即放入queue中并返回true，若queue已满，则立即返回false
- 当queue.enqueue.timeout.ms<0，则立即放入queue，若queue已满，则一直等待queue释放空间
- 当queue.enqueue.timeout.ms>0，则立即放入queue中并返回true，若queue已满，则等待queue.enqueue.timeout.ms指定的时间以让queue释放空间，若时间到queue还是没有足够空间，则立即返回false

ProducerSendThread

批处理的方式是在每次从queue中poll一条消息,先加入到一个数组events中,并在每次加入之后判断是否超过batchSize.

如果超过batchSize,则进行一次批处理,同时重置events数组和设置最后一次发送的时间.最后还需要有一次handle处理.

批处理的操作方式和java版本的RecordAccumulator类似,每次添加新消息时,都要判断加了之后,是否可以进行批处理.

```

private def processEvents() {
    var lastSend = SystemTime.milliseconds
    var events = new ArrayBuffer[KeyedMessage[K,V]]
    var full: Boolean = false

    // drain the queue until you get a shutdown command
    Iterator.continually(queue.poll(scala.math.max(0, (lastSend + c
        .takeWhile(item => if(item != null) item ne s
    currentQueueItem =>
        val expired = currentQueueItem == null
        if(currentQueueItem != null) {
            events += currentQueueItem      // 加入到临时数组中
        }
        full = events.size >= batchSize // check if the batch size
        if(full || expired) {          // 除了batch满了, 还可能是没有消息了
            tryToHandle(events)        // 开始批处理
            events = new ArrayBuffer[KeyedMessage[K,V]]
        }
    }
    tryToHandle(events)           // send the last batch of events
}

```

因为创建ProducerSendThread也指定了默认的eventHandler,所以在得到每一批消息时,可以交给handler处理了.

而对于同步的发送方式,是直接在handler上处理全部数据.而异步是将全部消息先放到队列中,再一小批一小批地处理.

```

def tryToHandle(events: Seq[KeyedMessage[K,V]]) {
    val size = events.size
    if(size > 0) handler.handle(events)
}

```

目前为止,scala的代码看起来非常简洁.主要是使用了比较简单的阻塞队列来缓存消息,而不像java中自己实现了很多类.

BrokerPartitionInfo

BrokerPartitionInfo -> topicPartitionInfo -> TopicMetadata -> PartitionMetadata

BrokerPartitionInfo 的getBrokerPartitionInfo会根据topic名称获取对应的BrokerPartition列表.

对于客户端而言只关心这个Partition的Leader副本,所以返回的是PartitionAndLeader列表.

```

class BrokerPartitionInfo(producerConfig: ProducerConfig, producerF
def getBrokerPartitionInfo(topic: String, correlationId: Int): Se
    val topicMetadata = topicPartitionInfo.get(topic) // check if
    val metadata: TopicMetadata = topicMetadata match {
        case Some(m) => m
        case None =>           // refresh the topic metadata cache
            updateInfo(Set(topic), correlationId)
            val topicMetadata = topicPartitionInfo.get(topic)
            topicMetadata match {
                case Some(m) => m
                case None => throw new KafkaException("Failed to fetch
            }
        }
    }
    // 一个TopicMetadata会有多个PartitionMetadata,每个PartitionMetad
    val partitionMetadata = metadata.partitionsMetadata
    partitionMetadata.map { m =>
        m.leader match {
            case Some(leader) =>      new PartitionAndLeader(topic, m.pai
            case None =>             new PartitionAndLeader(topic, m.pai
        }
    }.sortWith((s, t) => s.partitionId < t.partitionId) // 按照par
}
}

```

在java版本中Cluster保存了broker-topic-partitions等的关系,PartitionInfo表示一个分区(有Leader,ISR等)

每条消息都要根据Partitioner为它选择一个PartitionInfo,然后得到这个Partition的Leader Broker.根据Leader分组.

PartitionAndLeader 类似PartitionInfo,有一个可选的LeaderBrokerId,但是没有isr,replicas等信息.

```
case class PartitionAndLeader(topic: String, partitionId: Int, lead
```

updateInfo 会更新每个Topic的元数据TopicMetadata. 更新动作也相当于向Kafka发送一种Producer请求(fetch request).

客户端发送TopicMetadata的fetch请求后,会收到topicMetadata的Response响应,最后放到topicPartitionInfo map中.

```
def updateInfo(topics: Set[String], correlationId: Int) {  
    var topicsMetadata: Seq[TopicMetadata] = Nil  
    val topicMetadataResponse = ClientUtils.fetchTopicMetadata(topics)  
    topicsMetadata = topicMetadataResponse.topicsMetadata  
    topicsMetadata.foreach(tmd => if(tmd.errorCode == Errors.NONE.code) producerPool.updateProducer(topicsMetadata))  
}
```

这里更新之后的TopicMetadata还会返回给ProducerPool. 而Producer会从ProducerPool中获取可用的生产者实例服务于生产请求.

TopicMetadata 是一个Topic的元数据.一个topic有多个Partitions,所以一个TopicMetadata对应多个PartitionMetadata.

```
object TopicMetadata {  
    val NoLeaderNodeId = -1  
    def readFrom(buffer: ByteBuffer, brokers: Map[Int, BrokerEndPoint],  
                brokerEpoch: Int): TopicMetadata = {  
        val errorCode = readShortInRange(buffer, "error code", (-1, Short.MAX_VALUE))  
        val topic = readShortString(buffer)  
        val numPartitions = readIntInRange(buffer, "number of partitions", 0, Int.MaxValue)  
        val partitionsMetadata: Array[PartitionMetadata] = new Array[PartitionMetadata](numPartitions)  
        for(i <- 0 until numPartitions) {  
            val partitionMetadata = PartitionMetadata.readFrom(buffer, brokers, brokerEpoch)  
            partitionsMetadata(i) = partitionMetadata  
        }  
        new TopicMetadata(topic, partitionsMetadata, errorCode)  
    }  
    case class TopicMetadata(topic: String, partitionsMetadata: Seq[PartitionMetadata],  
                            errorCode: Short)  
}
```

PartitionMetadata 就等于PartitionAndLeader的元数据,包含 isr,replicas 等(类似于java版本的PartitionInfo).

```

object PartitionMetadata {
    def readFrom(buffer: ByteBuffer, brokers: Map[Int, BrokerEndPoint]): PartitionMetadata = {
        val errorCode = readShortInRange(buffer, "error code", (-1, Short.MaxValue))
        val partitionId = readIntInRange(buffer, "partition id", (0, Int.MaxValue))
        val leaderId = buffer.getInt
        val leader = brokers.get(leaderId) // brokers是集群所有节点, 每个分区的leader都是从这些节点中选出来的

        /* list of all replicas */
        val numReplicas = readIntInRange(buffer, "number of all replicas", (0, Int.MaxValue))
        val replicaIds = (0 until numReplicas).map(_ => buffer.getInt)
        val replicas = replicaIds.map(brokers)

        /* list of in-sync replicas */
        val numIsr = readIntInRange(buffer, "number of in-sync replicas", (0, Int.MaxValue))
        val isrIds = (0 until numIsr).map(_ => buffer.getInt)
        val isr = isrIds.map(brokers)

        new PartitionMetadata(partitionId, leader, replicas, isr, errorCode)
    }
}

case class PartitionMetadata(partitionId: Int, leader: Option[BrokerEndPoint], replicas: Seq[BrokerEndPoint], isr: Seq[BrokerEndPoint], errorCode: Short)

```

Partition

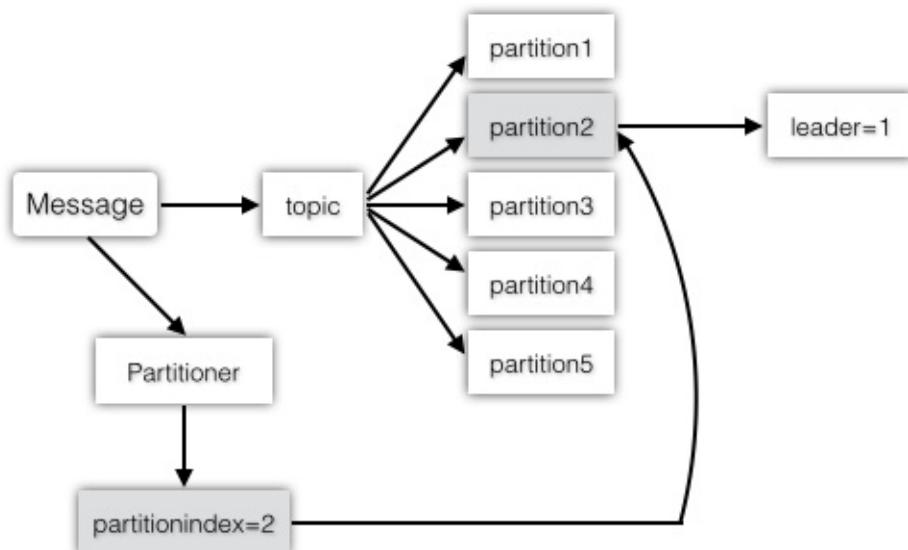
1. 为消息选择 Partition

上一步获取每条消息所属的topic对应的PartitionAndLeader列表.

PartitionAndLeader是这个topic的所有Partition.

但这些Partition并不一定都有Leader, 所以PartitionAndLeader的leaderBrokerIdOpt是可选的. 即还不确定有没有Leader.

为消息选择Partition一定要选择有Leader的Partition. 如果消息没有key则使用缓存, 相同topic的消息都分配同一个partitionId.



```

private def getPartitionListForTopic(m: KeyedMessage[K, Message]): List[Int] = {
  val topicPartitionsList = brokerPartitionInfo.getBrokerPartitionsList(topic)
  topicPartitionsList
}

private def getPartition(topic: String, key: Any, topicPartitionList: List[Int]): Int = {
  val numPartitions = topicPartitionList.size
  // 如果消息中没有key, 则从sendPartitionPerTopicCache中为这个topic的
  val partition = if(key == null) {
    val id = sendPartitionPerTopicCache.get(topic)
    id match {
      case Some(partitionId) => partitionId
      case None =>
        val availablePartitions = topicPartitionList.filter(_.isAvailable)
        val index = Utils.abs(Random.nextInt) % availablePartitions.size
        val partitionId = availablePartitions(index).partitionId
        sendPartitionPerTopicCache.put(topic, partitionId)
        partitionId
    }
  } else partitioner.partition(key, numPartitions) // 使用Partitioner
  partition
}
  
```

2.根据Broker-Partition重新组织消息集

由于生产者的消息集messages可能没有区分topic. 对每条消息选择所属的Partition, 要重新按照Broker组织数据. 通过将乱序的消息按照BrokerId进行分组, 这样可以将属于某个Broker的消息一次性发送过去.Int就是BrokerId/NodeId.

对于某个Broker的消息, 也要分成不同的TopicPartition, 最终每个Partition都会分到一批消息.

```
[Map[Int,  
      Map[TopicAndPartition,  
          Seq[KeyedMessage[K, Message]]]] ← BrokerId  
                           ← Partition  
                           ← 属于这个Partition的消息集
```

消息最终会追加到Partition的消息集里. 有多层Map, 根据key获取集合, 如果不存在, 则新建并放入Map; 如果存在则直接使用.

```

def partitionAndCollate(messages: Seq[KeyedMessage[K, Message]]):
    val ret = new HashMap[Int, collection.mutable.Map[TopicAndPar
        for (message <- messages) {
            //一个topic有多个partition
            val topicPartitionsList = getPartitionListForTopic(message)
            //一条消息只会写到一个partition, 根据Partitioner会分到一个partit
            val partitionIndex = getPartition(message.topic, message.p
            //一个partition因为有副本, 所以有多个broker, 但是写的时候只写到Leader
            val brokerPartition = topicPartitionsList(partitionIndex)
            // postpone the failure until the send operation, so that we can
            val leaderBrokerId = brokerPartition.leaderBrokerIdOpt.getOr
            // 每个Broker的数据. ret里有嵌套的Map[Int, Map[Partition, Seq]
            var dataPerBroker: HashMap[TopicAndPartition, Seq[KeyedMess
            ret.get(leaderBrokerId) match {
                case Some(element) =>           // Broker存在里层的Map, 直接使用这个
                    dataPerBroker = element.asInstanceOf[HashMap[TopicAndPar
                case None =>                  // Broker不存在里层Map, 创建一个新的
                    dataPerBroker = new HashMap[TopicAndPartition, Seq[Keye
                    ret.put(leaderBrokerId, dataPerBroker)
                }
                // 构造Topic和Partition对象
                val topicAndPartition = TopicAndPartition(message.topic, bi
                // Broker对应的消息集合, 即使是相同的Broker, Topic-Partition组合
                var dataPerTopicPartition: ArrayBuffer[KeyedMessage[K, Messa
                dataPerBroker.get(topicAndPartition) match {
                    case Some(element) =>           // Partition对应的消息集又是一个Seq
                        dataPerTopicPartition = element.asInstanceOf[ArrayBu
                    case None =>                  // Partition对应的消息集还不存在, 创建
                        dataPerTopicPartition = new ArrayBuffer[KeyedMessage[K,
                        dataPerBroker.put(topicAndPartition, dataPerTopicParti
                }
                // 到这里, 才是真正将消息添加到集合中
                dataPerTopicPartition.append(message)
            }
            Some(ret)
        }
    }

```

3.消息分组

上面返回的结构包括了每个Broker的消息集,在实际处理时,会针对每个Broker的消息进一步分组.

`groupMessagesToSet` 关于消息的输入和输出类型由 `KeyedMessage` 转为了 `ByteBufferMessageSet`

```
private def groupMessagesToSet(messagesPerTopicAndPartition: collection.Map[TopicAndPartition, collection.Seq[KeyedMessage]]) = {
    val messagesPerTopicPartition = messagesPerTopicAndPartition.mapValues(_.asScala)
    // KeyedMessage包括了Key,Value, 其中message就是value原始数据
    val rawMessages = messagesPerTopicPartition.map(_.map(_.message))
    // 输入是个Map(用case元组匹配), 返回值也是元组, 也会转成Map: key没有
    (topicAndPartition, config.compressionCodec match {
        case NoCompressionCodec => new ByteBufferMessageSet(NoCompressionCodec)
        case _ => new ByteBufferMessageSet(config.compressionCodec)
    })
}
Some(messagesPerTopicPartition)
}
```

在有了上面这些基础后,我们来看DefaultEventHandler如何处理一批不同topic的消息集.

DefaultEventHandler

事件处理器首先序列化,然后通过`dispatchSerializedData`发送消息,这里还带了重试发送功能.

```
def handle(events: Seq[KeyedMessage[K,V]]) {  
    val serializedData = serialize(events) // ① 序列化  
    var outstandingProduceRequests = serializedData // 未完成的请求  
    var remainingRetries = config.messageSendMaxRetries + 1  
    while (remainingRetries > 0 && outstandingProduceRequests.size > 0) {  
        outstandingProduceRequests = dispatchSerializedData(outstandingProduceRequests)  
        if (outstandingProduceRequests.size > 0) { // 有返回值表示失败  
            remainingRetries -= 1 // 重试次数减一  
        }  
    }  
}
```

在发送消息过程中,只要有失败的消息就加入到failedProduceRequests,这样返回的集合不为空,就会重试

```
private def dispatchSerializedData(messages: Seq[KeyedMessage[K, V]]): Unit = {
    val partitionedDataOpt = partitionAndCollate(messages)
    partitionedDataOpt match {
        case Some(partitionedData) =>
            val failedProduceRequests = new ArrayBuffer[KeyedMessage[K, V]]
            for ((brokerid, messagesPerBrokerMap) <- partitionedData) {
                val messageSetPerBrokerOpt = groupMessagesToSet(messagesPerBrokerMap)
                messageSetPerBrokerOpt match {
                    case Some(messageSetPerBroker) =>
                        val failedTopicPartitions = send(brokerid, messageSetPerBroker)
                        failedTopicPartitions.foreach(topicPartition => {
                            messagesPerBrokerMap.get(topicPartition) match {
                                case Some(data) => failedProduceRequests.appendAll(data)
                                case None => // nothing, 所有的消息都发送成功
                            }
                        })
                    case None => messagesPerBrokerMap.values.foreach(m => failedProduceRequests.appendAll(m))
                }
            }
            failedProduceRequests
        case None => messages // failed to collate messages
    }
}
```

发送消息,从生产者池中获取SyncProducer(每个Broker一个Producer),将消息集封装到ProducerRequest,调用Producer.send发送请求

java版本的发送请求是创建ProduceRequest-RequestSend-ClientRequest.并交给Sender-NetworkClient处理.

```
private def send(brokerId: Int, messagesPerTopic: collection.mutable  
    val currentCorrelationId = correlationId.getAndIncrement  
    val producerRequest = new ProducerRequest(currentCorrelationId,  
    val syncProducer = producerPool.getProducer(brokerId)  
    val response = syncProducer.send(producerRequest)  
}
```

发送消息,只需要指定brokerId,以及消息内容(TopicPartition->MessageSet),同时指定本请求是否需要ack和超时时间.

```
case class ProducerRequest(versionId: Short = ProducerRequest.CurrentVersion,  
                           requiredAcks: Short, ackTimeoutMs: Int,  
                           topicPartition: TopicPartition, messages: Seq[MessageSet])
```

ProducerPool

每个Broker都有一个SyncProducer,因为同步的Producer一次只会有一个请求发生在Broker上.

如果请求没有结束会一直阻塞的,其他请求没机会执行,所以没有必要一个Broker有多个Producer实例.

```

class ProducerPool(val config: ProducerConfig) extends Logging {
    private val syncProducers = new HashMap[Int, SyncProducer]
    private val lock = new Object()

    def updateProducer(topicMetadata: Seq[TopicMetadata]) {
        val newBrokers = new collection.mutable.HashSet[BrokerEndPoint]
        // 首先根据topicMetadata找出所有的Leader Broker.
        topicMetadata.foreach(tmd => {
            tmd.partitionsMetadata.foreach(pmd => {
                if(pmd.leader.isDefined) {
                    newBrokers += pmd.leader.get
                }
            })
        })
        // 每个Broker都创建一个同步类型的SyncProducer, 并放入缓存中, 等待getProducer
        lock synchronized {
            newBrokers.foreach(b => {
                if(syncProducers.contains(b.id)){
                    syncProducers(b.id).close()
                    syncProducers.put(b.id, ProducerPool.createSyncProducer(config))
                } else
                    syncProducers.put(b.id, ProducerPool.createSyncProducer(config))
            })
        }
    }

    def getProducer(brokerId: Int) : SyncProducer = {
        lock.synchronized {
            val producer = syncProducers.get(brokerId)
            producer match {
                case Some(p) => p
                case None => throw new UnavailableProducerException("Sync producer for broker " + brokerId + " is not available")
            }
        }
    }
}

```

创建SyncProducer,需要指定Broker的地址,因为这个Producer会负责和Broker通信,消息通过Producer发送到Broker.

```
object ProducerPool {
    // Used in ProducerPool to initiate a SyncProducer connection with
    def createSyncProducer(config: ProducerConfig, broker: BrokerEndpoint)
        val props = new Properties()
        props.put("host", broker.host)
        props.put("port", broker.port.toString)
        props.putAll(config.props.props)
        new SyncProducer(new SyncProducerConfig(props))
    }
}
```

SyncProducer

如果请求需要ack,则需要返回ProducerResponse给客户端(返回的消息内容放在response的payload字节缓冲区中).

```
private val blockingChannel = new BlockingChannel(config.host, config.port)

def send(producerRequest: ProducerRequest): ProducerResponse = {
    val readResponse = if(producerRequest.requiredAcks == 0) false
    var response: NetworkReceive = doSend(producerRequest, needAck = true)
    if(readResponse) ProducerResponse.readFrom(response.payload)
    else null // 如果生产请求需要响应(等待Leader响应, 或者除了Leader, ISR
}
private def doSend(request: RequestOrResponse, readResponse: Boolean): NetworkReceive
    verifyRequest(request)
    getOrMakeConnection() // 创建连接
    var response: NetworkReceive = null // 准备
    blockingChannel.send(request) // 向阻塞队列发送
    if(readResponse) response = blockingChannel.receive() // 从阻塞队列接收
}
}
```

这里还有一个send方法,不过发送的是TopicMetadata请求,这个请求一定是有响应的.

```
def send(request: TopicMetadataRequest): TopicMetadataResponse =  
    val response = doSend(request)  
    TopicMetadataResponse.readFrom(response.payload)  
}
```

ProducerResponse和TopicMetadataResponse的readFrom参数都是ByteBuffer,类似于反序列化.

发生在BlockingChannel的读写操作,前提是先建立连接,所以在doSend之前会getOrMakeConnection.

注意: 由于一个Broker只有一个SyncProducer,一个Producer也就只有一个BlockingChannel.

```
private def getOrMakeConnection() {  
    if(!blockingChannel.isConnected) connect()  
}  
  
private def connect(): BlockingChannel = {  
    if (!blockingChannel.isConnected && !shutdown) {  
        blockingChannel.connect()  
    }  
    blockingChannel  
}
```

BlockingChannel

实际的发送请求是交给BlockingChannel,它实现了I/O中的连接connect,发送请求send,接收响应receive

从它的名字看出这是一个阻塞类型的Channel,所以并没有用到NIO的多路选择特性,难怪这是Old的设计.

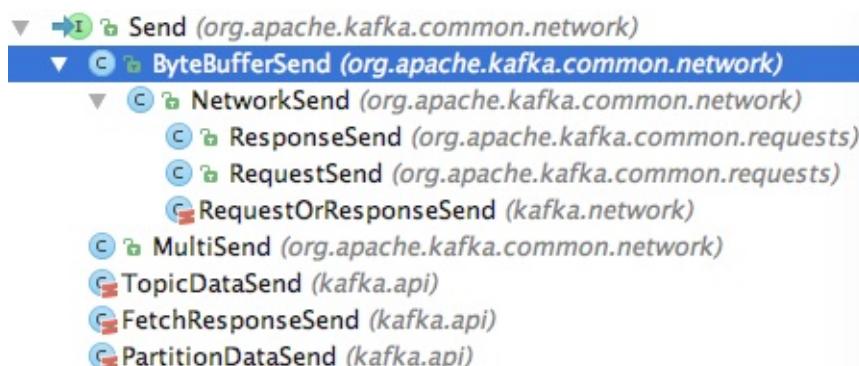
```

class BlockingChannel( val host: String,  val port: Int,  val readBufferSize: Int,  val writeBufferSize: Int) {
    private var connected = false
    private var channel: SocketChannel = null
    private var readChannel: ReadableByteChannel = null
    private var writeChannel: GatheringByteChannel = null
    private val lock = new Object()
    private val connectTimeoutMs = readTimeoutMs
    private var connectionId: String = ""

    def connect() = lock synchronized {
        if(!connected) {
            channel = SocketChannel.open()
            if(readBufferSize > 0) channel.socket.setReceiveBufferSize(readBufferSize)
            if(writeBufferSize > 0) channel.socket.setSendBufferSize(writeBufferSize)
            channel.configureBlocking(true)
            channel.socket.connect(new InetSocketAddress(host, port), connectTimeoutMs)
            writeChannel = channel
            readChannel = Channels.newChannel(channel.socket().getInputStream())
            connected = true
            connectionId = localHost + ":" + localPort + "-" + remoteHost + "-" + remotePort
        }
    }
}

```

KafkaProducer构造的请求是从ProduceRequest到RequestSend最后形成ClientRequest中.这里把ProduceRequest(是一种RequestOrResponse)封装到RequestOrResponseSend,他们都是ByteBufferSend的子类.



```
def send(request: RequestOrResponse): Long = {
    val send = new RequestOrResponseSend(connectionId, request)
    send.writeCompletely(writeChannel)
}

def receive(): NetworkReceive = {
    val response = readCompletely(readChannel)
    response.payload().rewind()          //读取到响应的ByteBuffer,回到缓冲区
    response                           //返回响应, 如果客户端需要ack, 则直接
}
private def readCompletely(channel: ReadableByteChannel): NetworkReceive =
    val response = new NetworkReceive
    while (!response.complete())
        response.readFromReadableChannel(channel)
    response
}
```

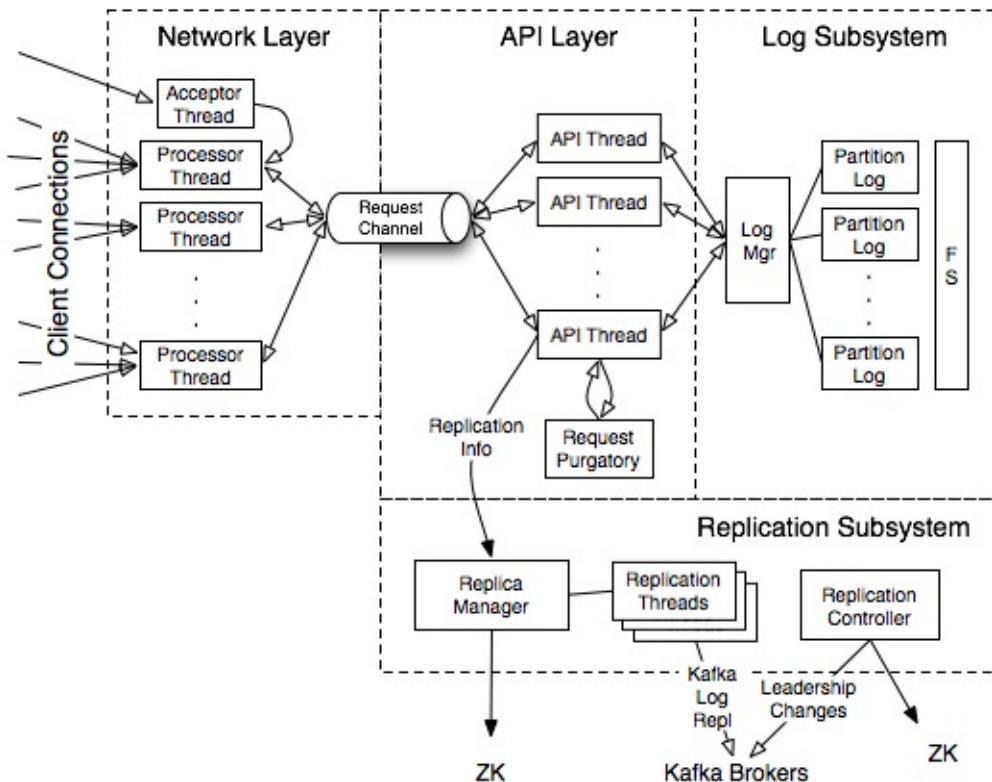
SyncProducer是阻塞类型的,所以并没有像java版本的NetworkClient使用Selector非阻塞异步模式.

Kafka的SocketServer

SocketServer

Kafka的Broker内部分成:网络层,API层,日志存储层,副本复制. 这里先介绍网络层的SocketServer

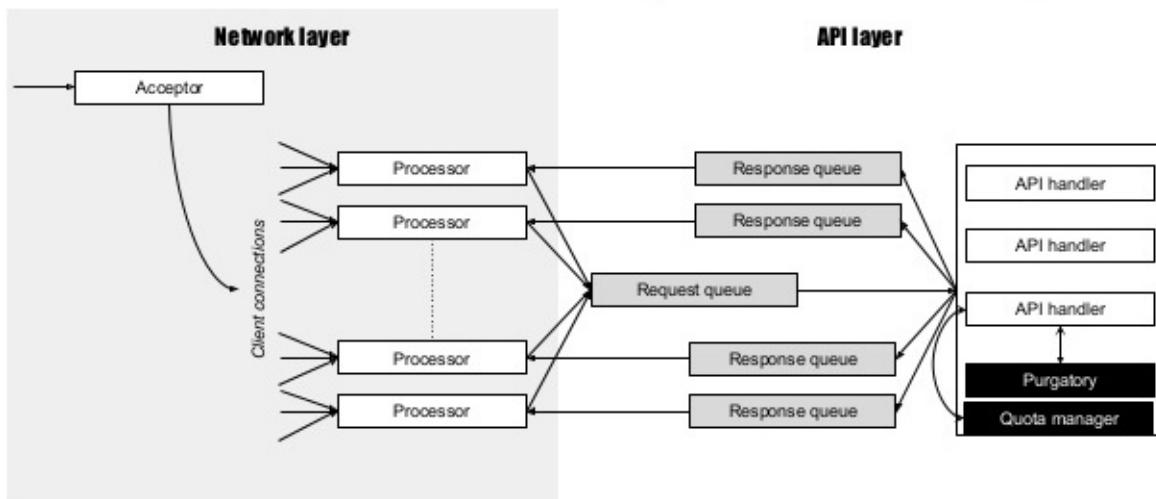
Kafka Broker Internals



SocketServer是一个NIO的服务器,它的线程模型:

- 一个Acceptor线程接受/处理所有的新连接
- N个Processor线程,每个Processor都有自己的selector,从每个连接中读取请求
- M个Handler线程处理请求,并将产生的请求返回给Processor线程用于写回客户端

Life-cycle of a Kafka request

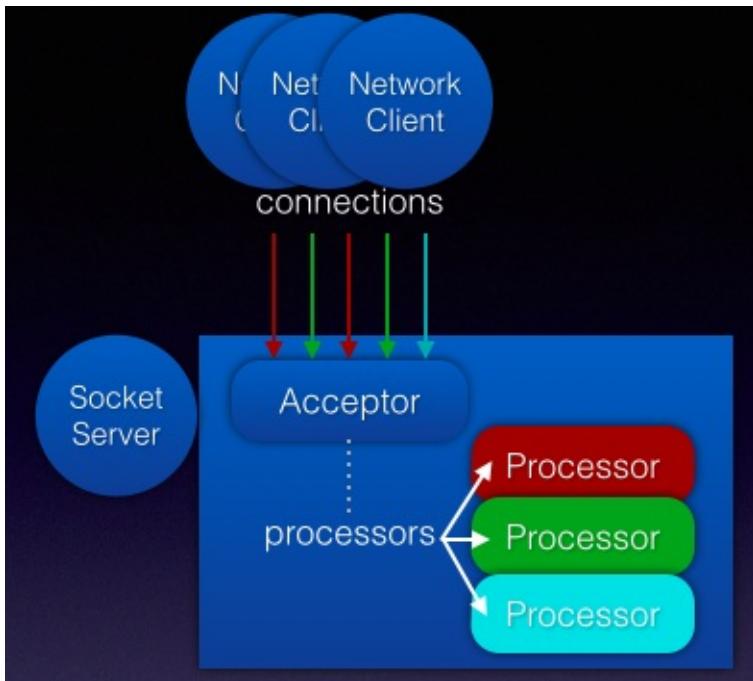


SocketServer在启动时(Kafka->KafkaServer),会启动一个Acceptor和N个Processor.

```

def startup() {
    val brokerId = config.brokerId
    var processorBeginIndex = 0
    endpoints.values.foreach { endpoint =>
        val protocol = endpoint.protocolType
        val processorEndIndex = processorBeginIndex + numProcessors
        for (i <- processorBeginIndex until processorEndIndex) {
            processors(i) = new Processor(i, time, maxRequestSize, requestTimeout)
        }
        //Processor线程是附属在Acceptor线程中,随着Acceptor的创建而启动线程
        val acceptor = new Acceptor(endpoint, sendBufferSize, recvBufferSize)
        acceptors.put(endpoint, acceptor)
        //启动Acceptor线程
        Utils.newThread("kafka-socket-acceptor-%s-%d".format(protocol, i))
            .start()
            .awaitStartUp()          //等待启动完成,通过CountDownLatch
        processorBeginIndex = processorEndIndex
    }
}

```



Acceptor

SelectionKey是表示一个Channel和Selector的注册关系。在Acceptor中的selector,

只有监听客户端连接请求的ServerSocketChannel的OP_ACCEPT事件注册在上面。

当selector的select方法返回时，则表示注册在它上面的Channel发生了对应的事件。

在Acceptor中，这个事件就是OP_ACCEPT，表示这个ServerSocketChannel的OP_ACCEPT事件发生了。

因此，Acceptor的accept方法的处理逻辑为：首先通过SelectionKey来拿到对应的ServerSocketChannel，

并调用其accept方法来建立和客户端的连接，然后拿到对应的SocketChannel并交给了processor。

然后Acceptor的任务就完成了，开始去处理下一个客户端的连接请求。

Acceptor只负责接受新的客户端的连接，并将请求转发给Processor处理,采用Round-robin的方式分给不同的Processor

```

def run() {
    serverChannel.register(nioSelector, SelectionKey.OP_ACCEPT)
    startupComplete()
    var currentProcessor = 0
    while (isRunning) {
        val ready = nioSelector.select(500)
        if (ready > 0) {
            val keys = nioSelector.selectedKeys()
            val iter = keys.iterator()
            while (iter.hasNext && isRunning) {
                val key = iter.next
                iter.remove()
                if (key.isAcceptable) accept(key, processors(currentProcessor))
                // round robin to the next processor thread
                currentProcessor = (currentProcessor + 1) % processors
            }
        }
    }
}

```

注册OP_ACCEPT时,注册到Selector上的serverChannel是一个ServerSocketChannel.

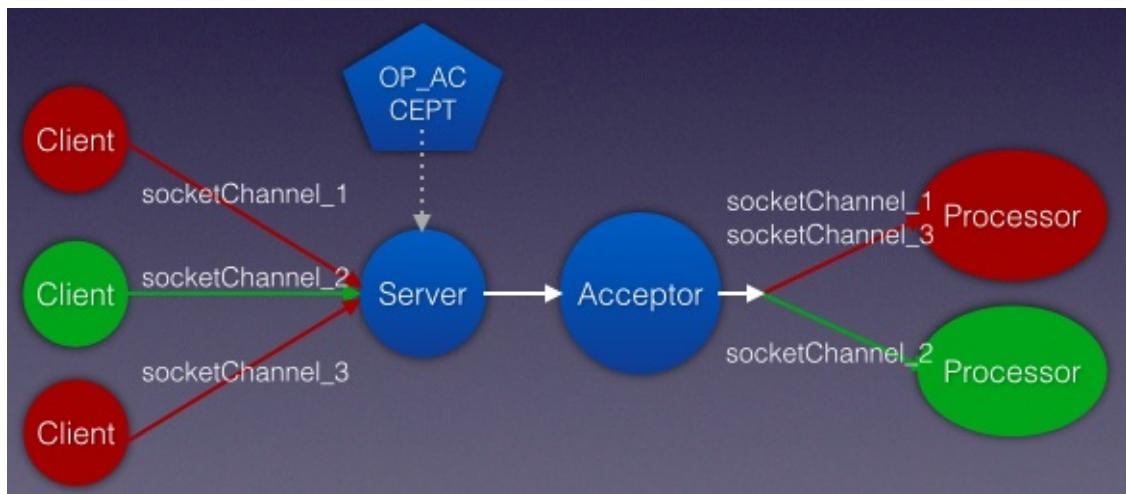
所以每个Processor都能获得Acceptor成功的连接上的SocketChannel.

```

def accept(key: SelectionKey, processor: Processor) {
    val serverSocketChannel = key.channel().asInstanceOf[ServerSocketChannel]
    val socketChannel = serverSocketChannel.accept()
    processor.accept(socketChannel)
}

```

客户端发起连接SocketChannel.connect,服务端接受这个连接
ServerSocketChannel.accept,于是双方可以互相通信了.



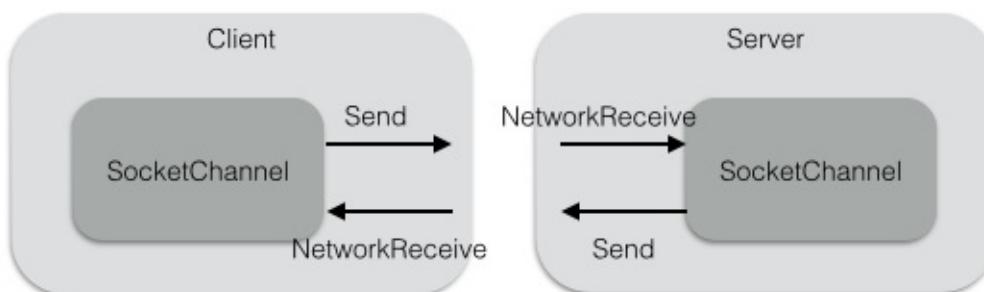
Processor

Processor的主要职责是负责从客户端读取数据和将响应返回给客户端，它本身不处理具体的业务逻辑，也就是说它并不认识它从客户端读取回来的数据。

每个Processor都有一个Selector，用来监听多个客户端，因此可以非阻塞地处理多个客户端的读写请求。

由于采用Round-Robin的方式分配连接给Processor,所以一个Processor会有多个SocketChannel,对应多个客户端连接.

每个SocketChannel都代表服务端和客户端建立的连接,Processor通过一个Selector不断轮询(并不需要每个连接对应一个Selector).



Client和Server都有自己的SocketChannel,代表和对方的连接通道
所有在这个通道上的读写操作,要么发送数据给对方,或者从对方接收数据

Processor接受一个新的SocketChannel通道连接时,先放入LinkedQueue队列中,然后唤醒Selector线程开始工作

Processor在运行时会首先从通道队列中去取SocketChannel,将客户端连接ID注册到Selector中,

便于后面Selector能够根据ConnectionID获取注册的不同的SocketChannel(比如 selector.completedReceives).

```

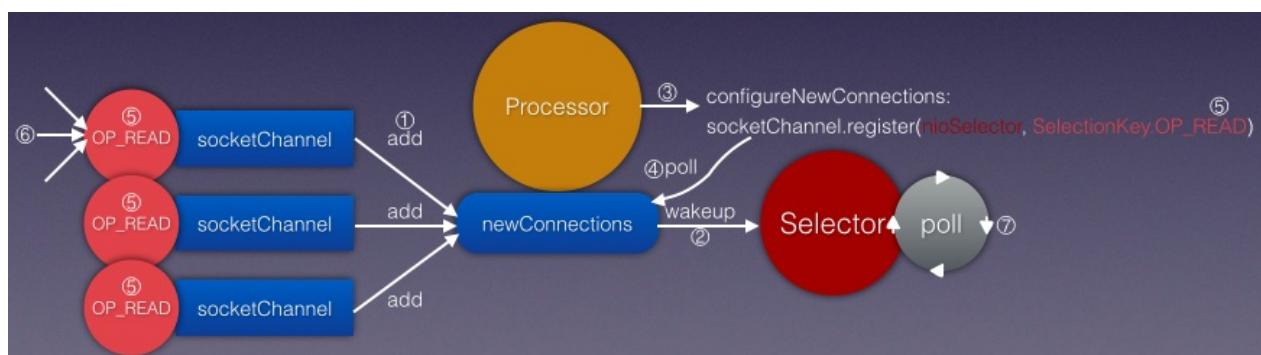
//Acceptor会把多个客户端的数据连接SocketChannel分配一个Processor, 因此每
//把一个SocketChannel放到队列中, 然后唤醒Processor的selector
def accept(socketChannel: SocketChannel) {
    newConnections.add(socketChannel)
    wakeup()
}

//如果有队列中有新的SocketChannel, 则它首先将其OP_READ事件注册到该Proces
private def configureNewConnections() {
    while(!newConnections.isEmpty) {
        val channel = newConnections.poll()
        val localHost = channel.socket().getLocalAddress.getHostAddress
        val localPort = channel.socket().getLocalPort
        val remoteHost = channel.socket().getInetAddress.getHostAddress
        val remotePort = channel.socket().getPort
        val connectionId = ConnectionId(localHost, localPort, remoteHost, remotePort)
        selector.register(connectionId, channel)
    }
}

```

回顾下客户端NetworkClient也是在finishConnect的时候注册了OP_READ事件, 用于读取服务端的响应.

而对于服务端而言, 在和客户端建立连接的时候, 注册OP_READ事件, 是为了读取客户端发送的请求.



下面的Selector也是前面分析KafkaProducer的Selector.每次轮询一次调用都需要处理返回的completedReceives, completedSends等
所以KafkaProducer/KafkaConsumer和SocketServer都采用NIO Selector方式.当然

客户端也可以是阻塞模式(比如OldProducer)

Selector模型是一种多路复用的通信模式,并不一定只在服务端才可以使用的.所以看到SocketServer使用了公用的Selector.

```

override def run() {
    startupComplete()
    while(isRunning) {
        configureNewConnections()      // setup any new connections th
        processNewResponses()         // register any new responses t
        selector.poll(300)            // poll轮询逻辑已经在KafkaProduce

        // NetworkClient.poll之后对已经完成发送和已经完成接收的都进行了han
        selector.completedReceives.asScala.foreach { receive =>
            //receive是NetworkReceive, 其中包含了源节点地址, 对应config
            val channel = selector.channel(receive.source)
            val session = RequestChannel.Session(new KafkaPrincipal(
                receive.principal,
                receive.host,
                receive.port))
            val req = RequestChannel.Request(processor = id, connection =
                session)
            requestChannel.sendRequest(req)          //Request请求, 发送给
            selector.mute(receive.source)           //移除OP_READ事件. 指
        }
        // 上面的completedReceives是服务端接收到客户端的请求, 下面的comple
        selector.completedSends.asScala.foreach { send =>
            val resp = inflightResponses.remove(send.destination).orNull
            selector.unmute(send.destination)     //添加OP_READ事件. 指
        }
    }
    shutdownComplete()
}

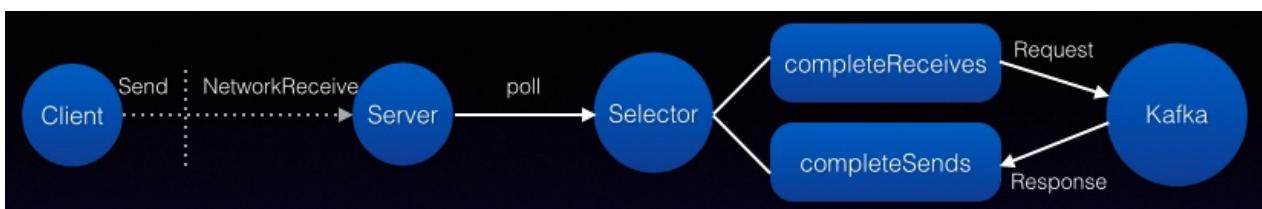
```

再和客户端的代码进行比较, 其实两者是有很多共同点的.

- 因为对于客户端和服务端而言,都有读写操作,所以也就都有Selector轮询产生的 completedSends,completedReceives.
- 客户端的发送请求(Send)对于服务端而言是读取请求(NetworkReceive),反过来服务端的Send对应客户端的NetworkReceive.
- 客户端Send的destination对应服务端NetworkReceive的source. 反过来服务端 Send的dest对应客户端NetworkReceice的source.
- 客户端请求ClientRequest在还没有收到响应时会将请求放到inFlightRequests

中,对应服务端的requestChannel.

- inFlightRequests表示正在进行中的客户端请求,在客户端开始发送请求时就加入到缓冲队列中,只有收到请求对应的响应结果,才从队列中删除.
- inflightResponses表示正在进行中的服务端响应,也是在服务端开始发送响应请求时加入到队列中,只有响应发送成功,才从队列中删除.
- 完整的客户端到服务端的流程:加到inFlightRequests,服务端处理请求,加到inflightResponses,删除inflightResponses,删除inFlightRequests
- 客户端的send表示要发送一个Send请求,但还没开始,服务端的processNewResponses也表示要发送一个Response,但也没开始,所以都需要缓冲队列.
-
- 客户端从Sender-NetworkClient-Selector. 服务端从Acceptor-Processor-Selector. Selector是真正干活的(在底层的通道上读写数据).
- 而NetworkClient和Processor会对读写请求的结果completeXXX做进一步处理(可以和客户端的handleCompletedXXX做比较).
- 客户端建立连接时注册READ:它不知道什么时候服务端会返回消息,服务端配置新连接时也注册READ:它也不知道客户端什么时候发送请求过来
- 客户端在刚开始要发送Send请求时,会注册OP_WRITE,当发送完一个完整的Send请求后,会取消OP_WRITE
- 而服务端在连接时注册了READ,接收到一个完整的NetworkReceive请求后,才会取消OP_READ.服务端在发送响应之后,还要重新注册READ.



Request->completedReceives

将客户端发送的请求和服务端的接收请求串联起来:客户端(P或C)发送请求会经过NetworkClient发送Send请求(ClientRequest),

服务端对每个客户端连接使用Processor处理(Processor和客户端建立SocketChannel进行通信).

客户端的发送和服务端的接收都采用了NIO的Selector轮询.客户端发送请求后可能并不需要响应(handleCompletedSends).

服务端接收客户端的请求处理是在selector.completedReceives中,这个时候会将收到的请求发给RequestChannel处理.

服务端注册OP_READ有两个地方,一个是在 `configureNewConnections`,一个是在 `completedSends` 完成响应发送之后.

注册OP_READ目的是让服务端能够读取客户端发送的请求,如果没有注册,即使客户端发送请求,也不会被Selector轮询出来.

在完成发送响应之后,为什么要继续注册READ?类似于配置新的连接时,就注册了READ.确保连接一旦建立如果有数据进来,就能立即读取到数据.

那么既然读取操作这么频繁,或者说不固定,何不如一直注册READ算了,不要在读取完毕后又取消了READ.显然不行,事件被触发比一直轮询要好.

```
// Use this on server-side, when a connection is accepted by a
public void register(String id, SocketChannel socketChannel) {
    SelectionKey key = socketChannel.register(nioSelector, SelectionEvent.OP_READ);
    KafkaChannel channel = channelBuilder.buildChannel(id, key);
    key.attach(channel);
    this.channels.put(id, channel);
}
```

客户的也有上面类似的代码,不过它是在Selector的connect方法里,虽然上面的register也是在Selector中,但是只有服务端调用.

实际上只会由客户端发起connect请求(SocketChannel.connect),所以对于客户端而言在建立连接时就创建KafkaChannel就很合适了.

旧版本的代码使用了BoundedByteBufferReceive,而不是NetworkReceive:

- a. 首先从SelectionKey中拿到对应的SocketChannel，并且取出attach在SelectionKey上的Receive对象，如果是第一次读取，Receive对象为null，则创建一个BoundedByteBufferReceive，由它来处理具体的读数据的逻辑。可以看到每个客户端都有一个Receive对象来读取数据(同时也有一个Send对象用来发送数据)。
- b. 如果数据从客户端读取完毕(receive.complete)，则将读取的数据封装成Request对象，并添加到requestChannel中去。如果没有读取完毕（可能是客户端还没有发送完或者网络延迟），那么就让selector继续监听这个通道的OP_READ事件。

Processor通过selector来监听它负责的那些数据通道，当通道上有数据可读时，它就是把这个事情交给BoundedByteBufferReceive。

BoundedByteBufferReceive先读一个int来确定数据量有多少，然后再读取真正的数据。那数据读取进来后又是如何被处理的呢？

Response->completedSends

Processor不仅负责从客户端读取数据，还要将Handler的处理结果返回给客户端.类似客户端的NetworkClient也负责发送和读取.

服务端发送响应给客户端:在processNewResponses[A]中,判断到是SendAction(肯定有人告诉Processor说这是一个发送命令),

于是注册了OP_WRITE事件,同时加入到inflightResponses中. 当轮询发生时,SelectionKey会监听到写事件,将写请求发送出去.

在poll轮询结束后,selector.completedSends表示已经完成发送的请求.需要做些清理工作:从inflightResponses删除相关信息[B].

Processor处理客户端的读请求,则要返回response给客户端,在poll之前就要注册任何新的response.

```

private def processNewResponses() {
    var curr = requestChannel.receiveResponse(id)           // id是Proc
    while(curr != null) {
        try {
            curr.responseAction match {
                case RequestChannel.NoOpAction =>          // 没有响应需要
                    selector.unmute(curr.request.connectionId)
                case RequestChannel.SendAction =>           // 有响应需要
                    selector.send(curr.responseSend)           // 将响应通过
                    inflightResponses += (curr.request.connectionId -> curr)
                case RequestChannel.CloseConnectionAction => // 根据返回值
                    close(selector, curr.request.connectionId)
            }
        } finally {
            curr = requestChannel.receiveResponse(id)       // while循环
        }
    }
}

```

NetworkClient发送Send请求给服务端加入到inFlightRequests,这里服务端发送Send请求给客户端加入到inflightResponses.

问题: 在selector.poll之后为什么要有completedReceives和completedSends的处理逻辑?

解释1: poll操作只是轮询,把注册在SocketChannel上的读写事件获取出来,那么得到事件后需要进行实际的处理才有用.

比如服务端注册了OP_READ,轮询时读取到客户端发送的请求,那么怎么服务端怎么处理客户端请求呢?就交给了completedReceives

可以把poll操作看做是服务端允许接收这个事件,然后还要把这个允许的事件拿出来进行真正的逻辑处理.

解释2: poll操作并不一定一次性完整地读取完客户端发送的请求,只有等到完整地读取完一个请求,才会出现在completeReceives中!

RequestChannel

RequestChannel是Processor和Handler交换数据的地方(Processor获取数据后通过RC交给Handler处理)。

它包含了一个队列requestQueue用来存放Processor加入的Request, Handler会从里面取出Request来处理；

它还为每个Processor开辟了一个respondQueue, 用来存放Handler处理了Request后给客户端的Response

RequestChannel是SocketServer全局的,一个服务端只有一个RequestChannel.

第一个参数Processor数量用于response,第二个参数队列大小用于request阻塞队列.

```
val requestChannel = new RequestChannel(totalProcessorThreads, m
// register the processor threads for notification of responses
requestChannel.addResponseListener(id => processors(id).wakeup())
```

每个Processor都有一个response队列,而Request请求则是全局的.

为什么request不需要给每个Processor都配备一个队列, 而response则需要呢?

```

class RequestChannel(val numProcessors: Int, val queueSize: Int) extends Actor {
    private var responseListeners: List[(Int) => Unit] = Nil
    private val requestQueue = new ArrayBlockingQueue[RequestChannel.Request](queueSize)
    private val responseQueues = new Array[BlockingQueue[RequestChannel.Response]](numProcessors)
    for(i <- 0 until numProcessors)
        responseQueues(i) = new LinkedBlockingQueue[RequestChannel.Response](queueSize)

    // Send a request to be handled, potentially blocking until there is a response
    // 如果requestQueue满的话，这个方法会阻塞在这里直到有Handler取走一个Request
    def sendRequest(request: RequestChannel.Request) {
        requestQueue.put(request)
    }

    // Send a response back to the socket server to be sent over the network
    def sendResponse(response: RequestChannel.Response) {
        responseQueues(response.processor).put(response)
        for(onResponse <- responseListeners) onResponse(response.processor)
    }

    // Get the next request or block until specified time has elapsed
    // Handler从requestQueue中取出Request，如果队列为空，这个方法会阻塞在这里
    def receiveRequest(timeout: Long): RequestChannel.Request = requestQueue.poll(timeout, TimeUnit.MILLISECONDS)

    // Get a response for the given processor if there is one
    def receiveResponse(processor: Int): RequestChannel.Response = responseQueues(processor).take()
}

//object RequestChannel的case class有：Session, Request, Response, ResponseProcessor

```

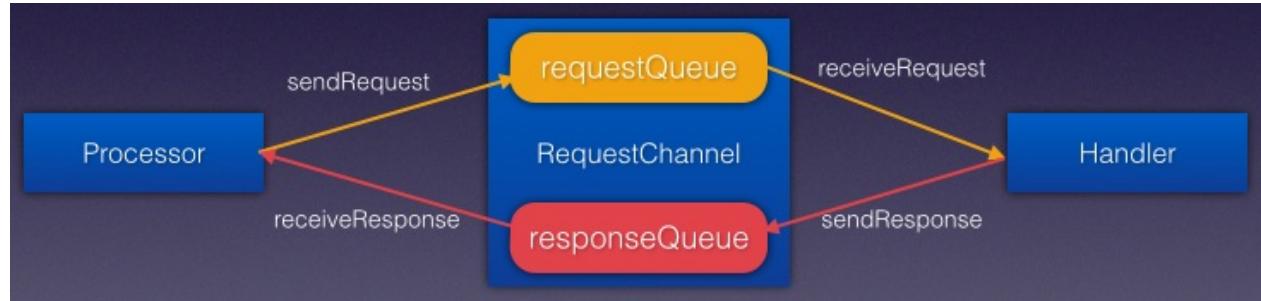
sendRequest 出现在Processor接收一个完整的客户端请求completedReceives后,加入到RequestChannel的请求队列

receiveResponse 出现在Processor中processNewResponses,它会读取RequestChannel中指定Processor的响应队列

- receiveRequest 用于KafkaRequestHandler从RequestChannel的请求队列获取客户端请求
- sendResponse 会由Kafka将响应结果发送给RequestChannel的指定

Processor的响应队列

Handler的职责是从requestChannel中的requestQueue取出Request, 处理以后再将Response添加到requestChannel中的responseQueue中。 Processor和Handler互相通信都通过RequestChannel的请求或响应队列



现在我们终于可以理清客户端发送请求到服务端处理请求的路径了：

```

NetworkClient --- ClientRequest(Send) --- KafkaChannel ==> SocketSelector
  
```

Request

请求转发给KafkaApis处理, 请求内容都在ByteBuffer buffer中. requestId表示请求类型, 有PRODUCE,FETCH等。

提前定义keyToNameAndDeserializerMap, 根据requestId, 再传入buffer, 就可以得到请求类型对应的Request

```
case class Request(processor: Int, connectionId: String, session: Session, offset: Long) {
    //首先获取请求类型对应的ID
    val requestId = buffer.getShort()

    // NOTE: this map only includes the server-side request/response types
    private val keyToNameAndDeserializerMap: Map[Short, (ByteBuffer, Function[ByteBuffer, Any])] =
        Map(ApiKeys.PRODUCE.id -> ProducerRequest.readFrom, ApiKeys.FETCH.id -> FetchRequest.readFrom,
            ApiKeys.FETCH_METADATA.id -> FetchMetadataRequest.readFrom, ApiKeys.PARTITION_OFFSETS.id -> PartitionOffsetRequest.readFrom,
            ApiKeys.PARTITION_OFFSETS_METADATA.id -> PartitionOffsetMetadataRequest.readFrom, ApiKeys.PARTITION_OFFSETS_FETCH.id -> PartitionOffsetFetchRequest.readFrom,
            ApiKeys.PARTITION_OFFSETS_METADATA_FETCH.id -> PartitionOffsetMetadataFetchRequest.readFrom)

    //通过ByteBuffer构造请求对象,比如ProducerRequest
    val requestObj = keyToNameAndDeserializerMap.get(requestId).map { case (key, (buffer, deserializer)) =>
        deserializer(buffer)
    }.get

    //请求头和请求内容
    val header: RequestHeader = if (requestObj == null) {buffer.readRequestHeader()} else requestObj.getHeader()
    val body: AbstractRequest = if (requestObj == null) AbstractRequest() else requestObj.getBody()

    //重置ByteBuffer为空
    buffer = null
}
```

KafkaRequestHandler

KafkaServer会创建KafkaRequestHandlerPool, 在HandlerPool中会启动numThreads个KafkaRequestHandler线程

KafkaRequestHandler线程从requestChannel的requestQueue中获取Request请求，交给KafkaApis的handle处理

```
def run() {
    while(true) {
        var req : RequestChannel.Request = null
        while (req == null) {
            req = requestChannel.receiveRequest(300)
        }
        apis.handle(req)
    }
}
```

注意RequestChannel是怎样从KafkaServer一直传递给KafkaRequestHandler的

```

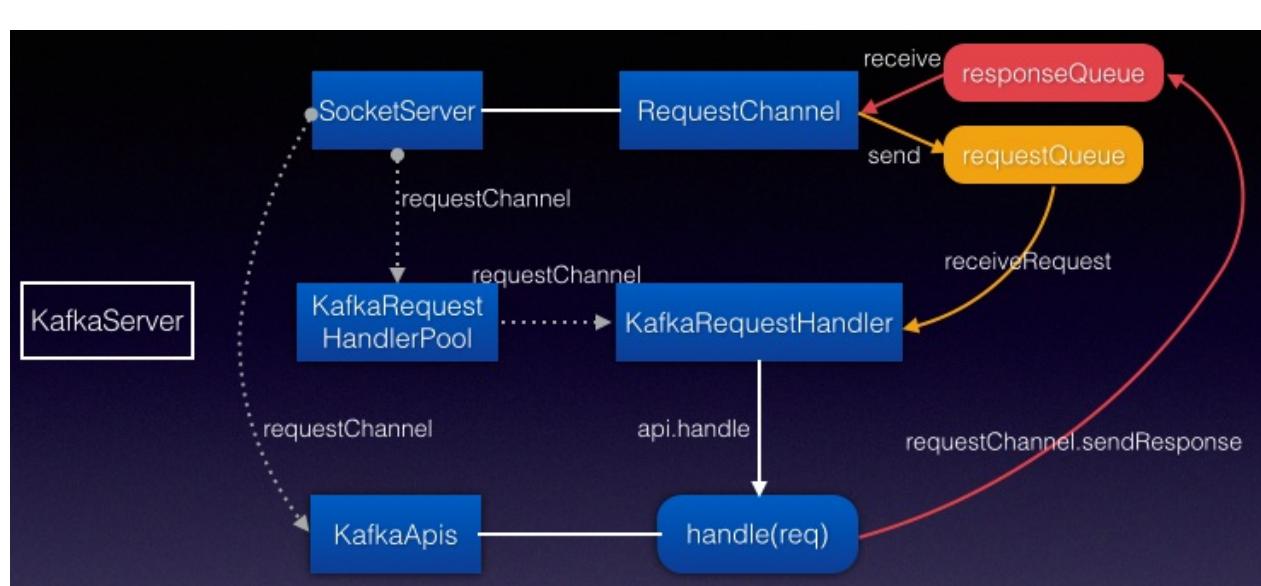
class KafkaServer {
    def startup() {
        socketServer = new SocketServer(config, metrics, kafkaMetricsTopic)
        socketServer.startup()

        apis = new KafkaApis(socketServer.requestChannel, replicaManager)
        requestHandlerPool = new KafkaRequestHandlerPool(config.brokerId)
    }
}

class KafkaRequestHandlerPool(val brokerId: Int, val requestChannel: RequestChannel) {
    for(i <- 0 until numThreads) {
        runnables(i) = new KafkaRequestHandler(i, brokerId, aggregateId)
        threads(i) = Utils.daemonThread("kafka-request-handler-" + i, i)
        threads(i).start()
    }
}

class KafkaRequestHandler(id: Int, brokerId: Int, val requestChannel: RequestChannel) {
}

```



Ref

- <https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Internals>
- <http://colobu.com/2015/03/23/kafka-internals/>
- <https://www.zybuluo.com/jewes/note/59978>

kafka-LogAppend

KafkaApis

上篇KafkaRequestHandler会将Request请求转发给KafkaApis处理:根据不同的请求类型调用不同的方法进行处理.

```
class KafkaApis(val requestChannel: RequestChannel, val replicaManager: ReplicaManager,
               val controller: KafkaController, val zkUtils: ZkUtils)
    extends RequestHandler with RequestDecoder with ResponseEncoder {
    // Top-level method that handles all requests and multiplexes to
    def handle(request: RequestChannel.Request) {
        try {
            ApiKeys.forName(request.requestId) match {
                case ApiKeys.PRODUCE => handleProducerRequest(request)
                case ApiKeys.FETCH => handleFetchRequest(request)
                // ...
            }
        } catch {
            case e: Throwable =>
                if (request.requestObj == null) {
                    val response = request.body.getErrorResponse(request.header)
                    val respHeader = new ResponseHeader(request.header.correlationId)
                    // If request doesn't have a default error response, we
                    // need to create one
                    if (response == null) requestChannel.closeConnection(request)
                    else requestChannel.sendResponse(new Response(request, response))
                }
        }
    }
}
```

SocketChannel的RequestChannel用于KafkaApis,因为响应和请求都需要放到RequestChannel的队列中被处理.

比如在handle出现异常时,会将Response加入到requestChannel的请求队列中(closeConnection也是一种Resp).

handleProducerRequest

有两个Callback函数定义,但是真正调用是通过

- ① replicaManager.appendMessages 触发的:
- ② sendResponseCallback->③ produceResponseCallback->④ ProducerResponse->⑤ requestChannel.sendResponse

真正的发送ProducerResponse是在produceResponseCallback中:如果不需要ack,则这一整条链路都不会调用.

sendResponseCallback的参数responseStatus是响应状态,所以①中要确保会生成它,然后使用回调的函数调用.

```

def handleProducerRequest(request: RequestChannel.Request) {
    val produceRequest = request.requestObj.asInstanceOf[ProducerRe
    val numBytesAppended = produceRequest.sizeInBytes

    //partition是对Map分成两组:有授权的(true)和没有授权的(false).返回值类
    val (authorizedRequestInfo, unauthorizedRequestInfo) = produce
        //data是TopicAndPartition->MessageSet, 在判断是否授权时不需要Message
        case (topicAndPartition, _) => authorize(request.session, Wr
    }

    // *****
    // ② the callback for sending a produce response 发送Producer的
    def sendResponseCallback(responseStatus: Map[TopicAndPartition,
        // ③ 生成Response的Callback, 即将Response放入RequestChannel的re
        def produceResponseCallback(delayTimeMs: Int) {
            if (produceRequest.requiredAcks == 0) {
                if (errorInResponse) {
                    requestChannel.closeConnection(request.processor, request)
                } else {
                    requestChannel.noOperation(request.processor, request)
                }
            } else {
                // ④ Producer请求需要等待ack(1, -1), 创建ProducerResponse, 然后将
                val response = ProducerResponse(produceRequest.correlationId,
                    // ⑤ 将响应信息推入到请求通道中, 最后会发送给Producer客户端
                    requestChannel.sendResponse(new RequestChannel.Response(
                }
            }
        }
    }
}

```

```

    // ③ quotaManagers会调用produceResponseCallback, 它本身做了一些
    quotaManagers(ApiKeys.PRODUCE.id).recordAndMaybeThrottle(pro
}
// *****

// ① 将回调函数传给replicaManager.appendMessages方法, 所以回调函数真
if (authorizedRequestInfo.isEmpty) sendResponseCallback(Map.emp
else { // call the replica manager to append messages to the re
    replicaManager.appendMessages(produceRequest.ackTimeoutMs.tol
    produceRequest.emptyData()
}
}

```

handleProducerRequest处理Producer的请求: 将Request请求转换为ProducerRequest.

ProducerRequest含有生产者的 Partition->消息集 :authorizedRequestInfo,会被写入到日志中.

Producer端的 request.required.acks 配置项,用来控制什么时候返回响应给客户端:

acks	what happen
0	The producer never waits for an ack 生产者不会等待一个ack,收到消息后直接返回给客户端
1	The producer gets an ack after the leader replica has received the data 当写Leader成功后就返回,其他的replica都是通过fetcher去同步的,所以kafka是异步写
-1	The producer gets an ack after all ISRs receive the data 要等待所有的replicas都成功后才能返回.

responseCallback

responseCallback作为参数传入appendMessages方法中,是因为首先要把生产者的消息写到本地日志后,才能判断是否要发送响应给客户端.

接着还有一种情况是延迟的操作,则回调函数的调用又会被拖后到

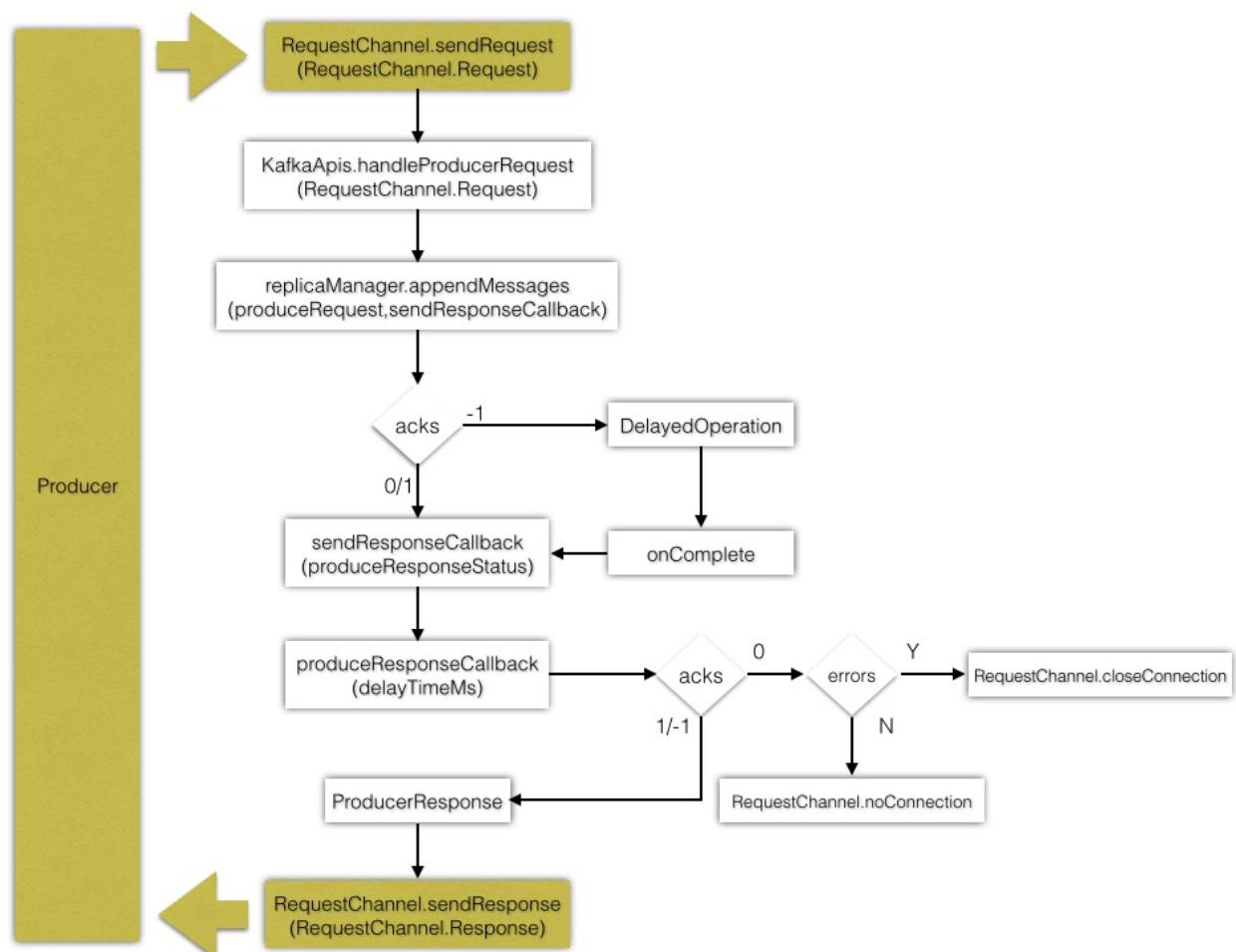
DelayedOperation.onComplete时。

由于responseCallback回到函数需要一个表示响应状态的参数,因此这里要负责生成responseStatus.

```

def appendMessages(timeout: Long, requiredAcks: Short, internalTopic: TopicAndPartition,
                   responseCallback: Map[TopicAndPartition, ProducerRecord])
    if (isValidRequiredAcks(requiredAcks)) {
        // 首先追加到本地日志中... appendToLocalLog, 然后判断是否要立即返回
        if (delayedRequestRequired(requiredAcks, messagesPerPartition))
            // 创建延迟的DelayedProduce, 在完成后, 调用onComplete, 也会触发onComplete
        } else {
            // we can respond immediately 对于acks=0,1的情况, 现在可以返回
            val produceResponseStatus = produceStatus.mapValues(status => responseCallback(produceResponseStatus))
        }
    }
}

```



ReplicaManager.appendMessages

- 将消息追加到Partition的leader replicas，并且等待它们复制到其他副本上.
- 回调函数只有在超时或者需要acks时才会被调用

由于Producer客户端在发送ClientRequest的时候已经根据Partition找到了所属的Leader,

所以ClientRequest一定是发送给Leader节点的SocketServer的(也是KafkaServer节点),

所以接收ProducerRequest一定是消息的Partition对应的Leader节点,可以放心地往本地写数据.

```
def appendMessages(timeout: Long, requiredAcks: Short, internalTopicsAll: Map[TopicAndPartition, TopicPartition], messages: Seq[MessageAndOffset]): Future[LocalProduceResults] = {
    if (isValidRequiredAcks(requiredAcks)) {
        // 添加到本地日志(为什么是本地? 本地指的是KafkaServer节点, 同时也是TopicAndPartition)
        val localProduceResults = appendToLocalLog(internalTopicsAll, messages)
        // 下面是response的一些处理, 数据在appendToLocalLog方法中已经被成功写入
    }
}
```

messagesPerPartition因为是TopicAndPartition到消息集的映射, topic用于验证是否能够允许写.

Partition是消息最终存储的所在文件夹,调用

partition.appendMessagesToLeader(messages),

表示将消息messages追加到指定的这个partition(而这个Partition一定是Leader replicas).

```

private def appendToLocalLog(internalTopicsAllowed: Boolean, messages: Seq[Message], requiredAcks: Short): Map[TopicAndPartition, LogAppendResult] = {
  messagesPerPartition.map { case (topicAndPartition, messages) =>
    // reject appending to internal topics if it is not allowed.
    if (!Topic.InternalTopics.contains(topicAndPartition.topic))
      val partitionOpt = getPartition(topicAndPartition.topic, topicAndPartition.partitionId)
      val info = partitionOpt match {
        case Some(partition) => partition.appendMessagesToLeader(messages, requiredAcks)
      }
      (topicAndPartition, LogAppendResult(info))
    }
  }
}

```

topicAndPartition的源头是由Producer客户端构造的,服务端需要根据topic和partitionId构造出属于自己的Partition用于写日志

Partition是怎么来的:getOrCreatePartition的调用链

= becomeLeaderOrFollower <- KafkaApis.handleLeaderAndFollowerRequest

```

private val allPartitions = new Pool[(String, Int), Partition]

def getOrCreatePartition(topic: String, partitionId: Int): Partition = {
  var partition = allPartitions.get((topic, partitionId))
  if (partition == null) {
    allPartitions.putIfNotExists((topic, partitionId), new Partition)
    partition = allPartitions.get((topic, partitionId))
  }
  partition
}

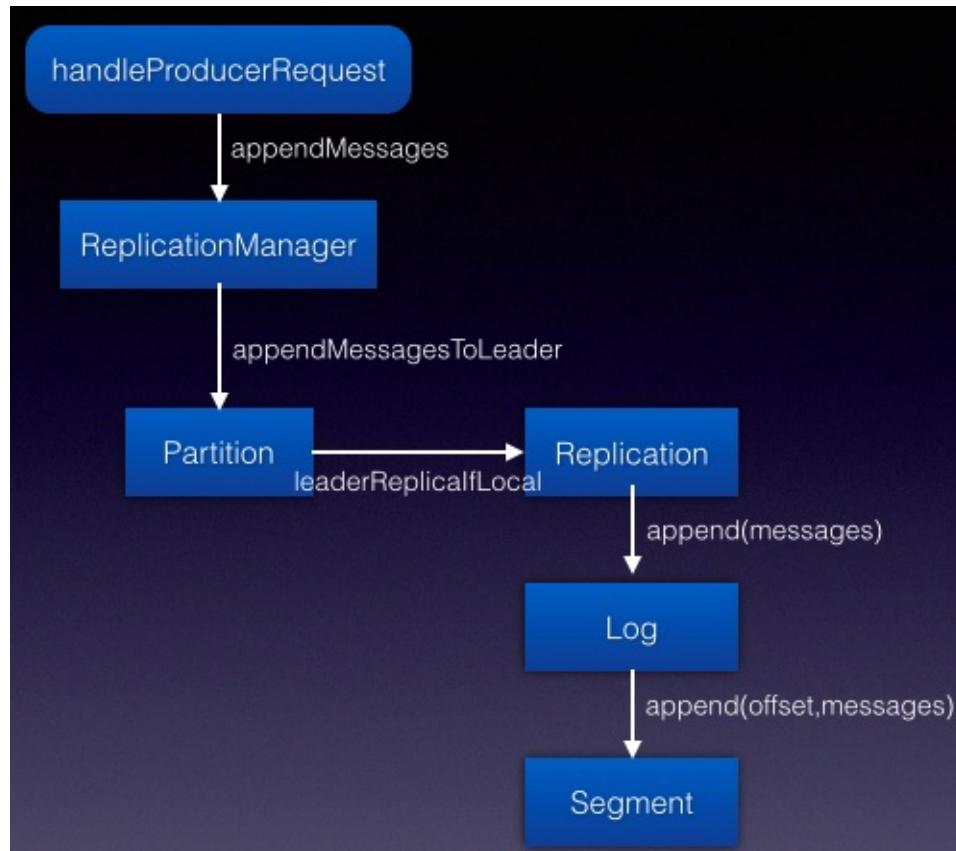
def getPartition(topic: String, partitionId: Int): Option[Partition] = {
  val partition = allPartitions.get((topic, partitionId))
  if (partition == null) None
  else Some(partition)
}

```

Partition.appendMessagesToLeader

消息写到Partition的Leader replica,在确定Partition的Replica后,往Replication的Log实例追加数据.

调用链从ReplicationManager -> Partition -> Replica -> Log, 消息messages最终写到日志文件中.



```

def appendMessagesToLeader(messages: ByteBufferMessageSet, requiredAcks: Int) {
    val (info, leaderHWIncremented) = inReadLock(leaderIsrUpdateLock)
    val leaderReplicaOpt = leaderReplicaIfLocal()
    leaderReplicaOpt match {
        case Some(leaderReplica) =>
            val log = leaderReplica.log.get
            val minIsr = log.config.minInSyncReplicas
            val inSyncSize = inSyncReplicas.size
            // Avoid writing to leader if there are not enough insync replicas
            if (inSyncSize < minIsr && requiredAcks == -1) throw new IllegalStateException("Not enough in sync replicas")
            val info = log.append(messages, assignOffsets = true)
            replicaManager.tryCompleteDelayedFetch(new TopicPartition(
                info.topic, info.partition), maybeIncrementLeaderHW(leaderReplica))
    }
}
if (leaderHWIncremented) tryCompleteDelayedRequests() // some logic here
info
}

```

leaderReplicaIfLocal

判断leaderReplica是否是本地的, 是比较leaderReplicaOpt(这个变量是volatile)和localBrokerId是否相同.

localBrokerId在Kafka节点启动时就是确定的, 即同一个KafkaServer节点的所有Partition的BrokerId都是相等的.

而leaderReplicaOpt会在 makeLeader 和 makeFollower 中被修改(也都被becomeLeaderOrFollower调用)

assignedReplicaMap是一个Map: 因为一个Partition的Replication有Leader和Follower.

它们都是分布在不同的Broker节点上. 所以Partition持有所有这些Replication, 就可以知道Replication的分布情况.

```

class Partition(val topic: String, val partitionId: Int, time: Time)
  private val localBrokerId = replicaManager.config.brokerId      //本地BrokerId
  @volatile var leaderReplicaIdOpt: Option[Int] = None
  private val assignedReplicaMap = new Pool[Int, Replica]

  def leaderReplicaIfLocal(): Option[Replica] = {
    leaderReplicaIdOpt match {
      case Some(leaderReplicaId) => if (leaderReplicaId == localBrokerId) Some(getReplica(leaderReplicaId))
      case None => None
    }
  }

  def getReplica(replicaId: Int = localBrokerId): Option[Replica] =
    val replica = assignedReplicaMap.get(replicaId)
    if (replica == null) None
    else Some(replica)
  }
}

```

问题: 具体Replica是如何被创建出来的, 以及local和remote的含义分别是什么?
会在下一篇介绍.

Log.append

将消息转换为LogAppendInfo, 并交给Log中最新的Segment处理, 因为一个Partition的Log分成多个Segment.

追加消息只到最新的Segment中, 旧的Segment一旦关闭后, 就不能再添加数据. 所以必要的话会创建新的Segment.

- ① LogOffsetMetadata(Log的Offset元数据)的当前值作为本次LogAppendInfo的firstOffset
- ② 给MessageSet分配Offset: 每条消息的offset都是单调递增的
- ③ LogAppendInfo的lastOffset是这一批消息的最后一条消息的offset
- ④ 添加消息到当前或者新创建的Segment
- ⑤ 更新Log End Offset为当前最后一条消息的offset的下一条

```

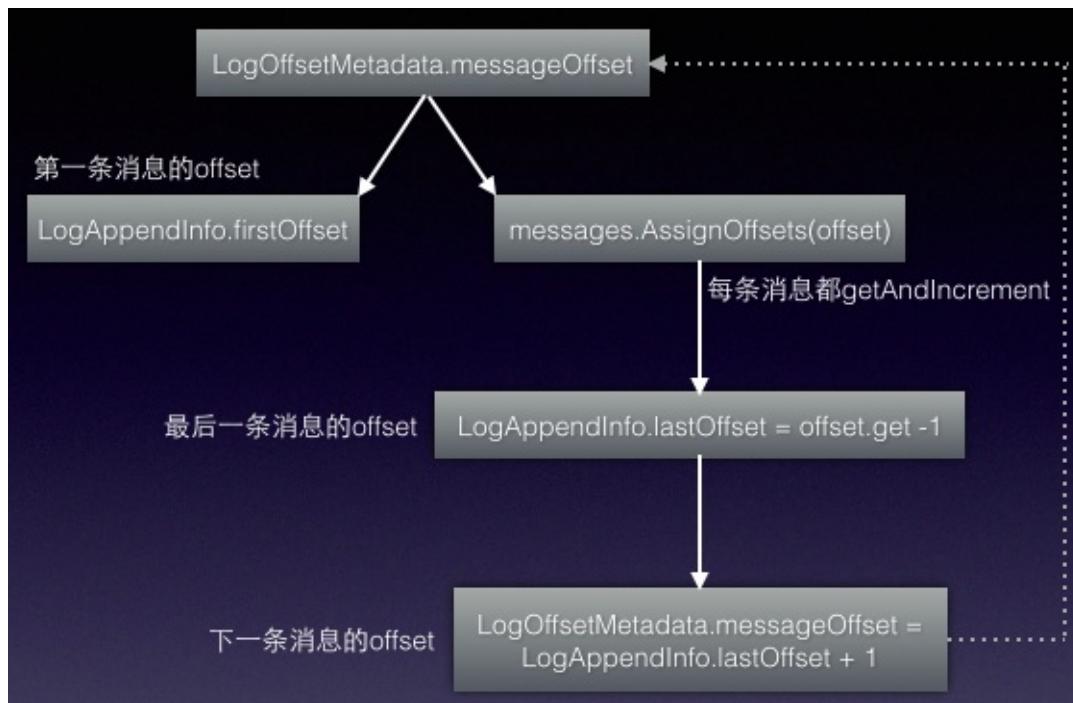
@volatile var nextOffsetMetadata = new LogOffsetMetadata(activeSe

def append(messages: ByteBufferMessageSet, assignOffsets: Boolean) {
    val appendInfo = analyzeAndValidateMessageSet(messages)
    // trim any invalid bytes or partial messages before appending
    var validMessages = trimInvalidBytes(messages, appendInfo)
    // they are valid, insert them in the log
    lock synchronized {
        appendInfo.firstOffset = nextOffsetMetadata.messageOffset
        if(assignOffsets) {
            val offset = new AtomicLong(nextOffsetMetadata.messageOffset)
            validMessages = validMessages.validateMessagesAndAssignOffset(offset)
            appendInfo.lastOffset = offset.get + 1
        }
        val segment = maybeRoll(validMessages.sizeInBytes)
        segment.append(appendInfo.firstOffset, validMessages)
        updateLogEndOffset(appendInfo.lastOffset + 1)
        if(unflushedMessages >= config.flushInterval) flush()
        appendInfo
    }
}

// 直接修改nextOffsetMetadata实例，获取messageOffset时比读取文件要快
private def updateLogEndOffset(messageOffset: Long) {
    nextOffsetMetadata = new LogOffsetMetadata(messageOffset, activeSe
}

```

追加一批消息,开始时从LogOffsetMetadata获取最新的messageoffset,
 最后更新LogOffsetMetadata的messageOffset为 最后一条消息的offset+1 ,
 这样下一次从LogOffsetMetadata获取的messageOffset是紧接着上一次的offset之后.



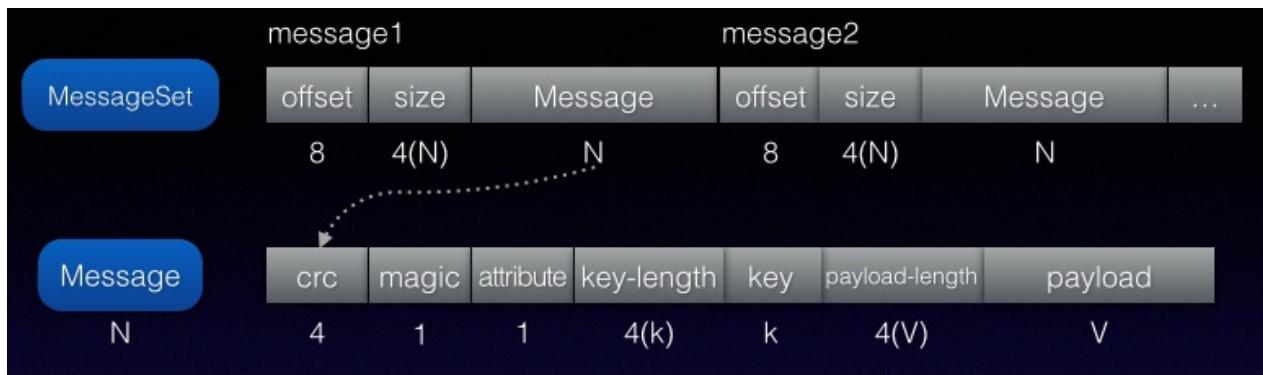
MessageSet

`MessageSet` : A set of messages with offsets. The format of each message is as follows:

- 8 byte `message_offset` number
- 4 byte `size` containing an integer N
- N byte `message`

`Message` : The format of an N byte message is the following:

- 4 byte CRC32 of the message
- 1 byte "magic" identifier to allow format changes, value is 0 currently
- 1 byte "attributes" identifier(compression enabled, type of codec used)
- 4 byte `key_length`, containing length K
- K byte `key`
- 4 byte `payload_length`, containing length V
- V byte `payload`



ByteBufferMessageSet的shallowIterator返回MessageAndOffset迭代器,验证集合中的每条消息是否合法.

```

def shallowIterator: Iterator[MessageAndOffset] = internalIterator

private def internalIterator(isShallow: Boolean = false): Iterator[MessageAndOffset] = new IteratorTemplate[MessageAndOffset] {
    var topIter = buffer.slice() // buffer是
    def makeNextOuter: MessageAndOffset = {
        if (topIter.remaining < 12) return allDone() // if there
        val offset = topIter.getLong() // 读取8byte
        val size = topIter.getInt() // 读取4byte
        if (size < Message.MinHeaderSize) throw new InvalidMessageException("Message size is less than min header size")
        if (topIter.remaining < size) return allDone() // we have
        val message = topIter.slice() // read the
        message.limit(size) // size表示
        topIter.position(topIter.position + size) // 通过上面的
        val newMessage = new Message(message) // 已经得到
        new MessageAndOffset(newMessage, offset) // Message
    }
    override def makeNext(): MessageAndOffset = makeNextOuter
}
}

```

analyzeAndValidateMessageSet

除了检查每条消息是否和CRC匹配,消息的大小是否有效,还返回了如下信息:

- First offset in the message set 消息集中第一个offset
- Last offset in the message set 消息集中最后一个offset

- Number of messages 消息数量
- Number of valid bytes 有效字节数
- Whether the offsets are monotonically increasing 偏移量是否单调递增

```

private def analyzeAndValidateMessageSet(messages: ByteBufferMessageSet)
    var shallowMessageCount = 0 // 消息数量
    var validBytesCount = 0 // 有效字节数
    var firstOffset, lastOffset = -1L // 第一条消息和最后一条(在遍历前)
    var monotonic = true // 是否单调变化(单调递增)
    for(messageAndOffset <- messages.shallowIterator) {
        if(firstOffset < 0) firstOffset = messageAndOffset.offset
        if(lastOffset >= messageAndOffset.offset) monotonic = false
        lastOffset = messageAndOffset.offset
        val m = messageAndOffset.message
        val messageSize = MessageSet.entrySize(m)
        m.ensureValid()
        shallowMessageCount += 1
        validBytesCount += messageSize
    }
    LogAppendInfo(firstOffset, lastOffset, sourceCodec, targetCodec)
}

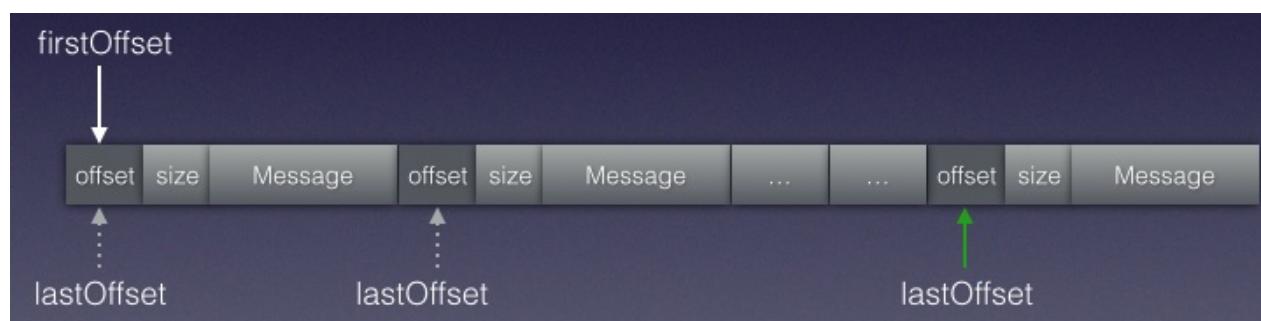
```

因为MessageSet包含多条Message.firstOffset<0只会在第一条消息时执行,firstOffset=第一条消息的offset.

lastOffset在遍历每条消息时,都更改为当前消息的offset. 初始时为-1,不执行if, 并赋值为第一条消息的offset,

第二条消息时,lastOffset是第一条消息的offset,messageAndOffset现在是第二条消息,如果是单调递增的offset,

越往后消息的offset都比前一条offset要高,则if语句同样都不会执行.直到最后一条消息的offset设置为lastOffset.

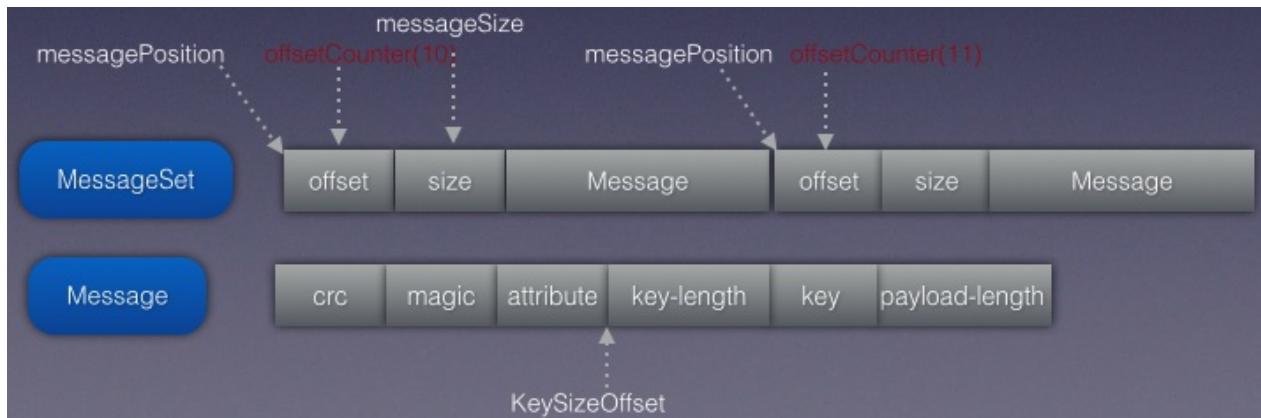


validateMessagesAndAssignOffsets

LogAppendInfo的firstOffset虽然上面已经计算出来了,但是真正是由

LogOffsetMetadata.messageOffset确定的

lastOffset也要根据初始化值重新计算: offsetCounter 在循环每条消息时不断递增1,代表的是下一条消息的offset.



```

private[kafka] def validateMessagesAndAssignOffsets(offsetCounter: Long)
  if(sourceCodec == NoCompressionCodec && targetCodec == NoCompressionCodec)
    var messagePosition = 0
    buffer.mark()           // 先标记
    while(messagePosition < sizeInBytes - MessageSet.LogOverhead)
      buffer.position(messagePosition)
      buffer.putLong(offsetCounter.getAndIncrement())
      val messageSize = buffer.getInt()
      val positionAfterKeySize = buffer.position + Message.KeySizeOffset
      if (compactedTopic && positionAfterKeySize < sizeInBytes) {
        buffer.position(buffer.position() + Message.KeySizeOffset)
      }
      val keySize = buffer.getInt()
    }
    messagePosition += MessageSet.LogOverhead + messageSize
  }
  buffer.reset()           // 重置的时候，回到最开始标记的地方
  this
}
}

```

在遍历完MessageSet后,offset的值是初始值+所有消息数量,最后更新下LogAppendInfo的lastOffset.

```

appendInfo.firstOffset = nextOffsetMetadata.messageOffset
if(assignOffsets) { // assign offsets to the message set
    val offset = new AtomicLong(nextOffsetMetadata.messageOffset)
    validMessages = validMessages.validateMessagesAndAssignOffset
    appendInfo.lastOffset = offset.get - 1
}

```



为什么在最后要减去1: 因为每条消息的offsetCounter在获取后就递增1, 所以它表示的是下一条消息的offset.

假设初始值nextOffsetMetadata的offset=10, 第一条消息的offset=10, 递增1; 第二条消息的offset=11, 递增1.

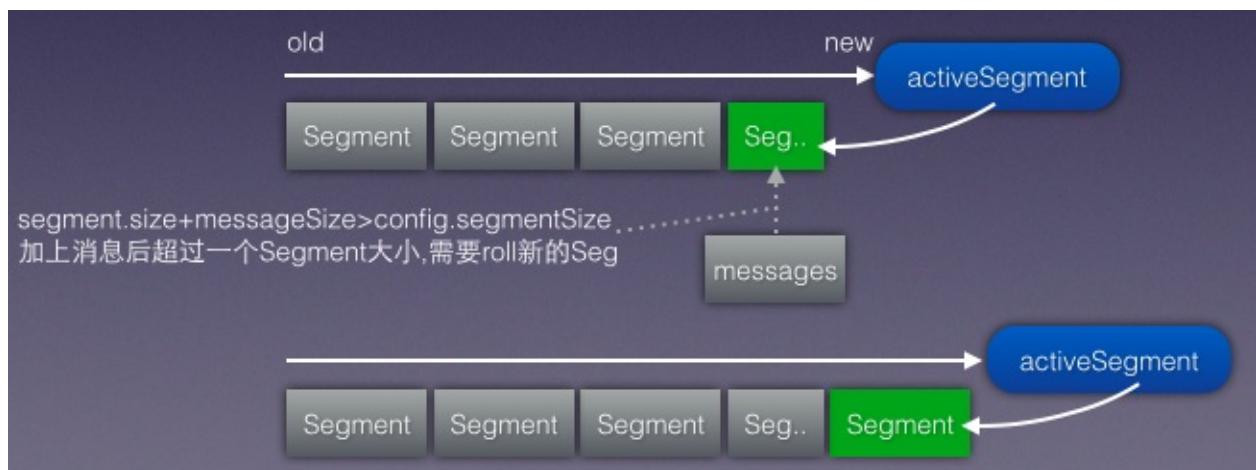
假设有5条消息, 最后的offset返回值=15, 而实际上最有一条消息的offset=lastOffset应该是 $15 - 1 = 14$

	message1	message2	message3	message4	message5	
offset	10	11	12	13	14	<- lastOffset
increment	11	12	13	14	15	<- offsetCount

LogSegment Roll

发生Roll是有一定时机的:

- 当前Segment+本次需要追加的消息大小超过Segment的阈值
- 索引满了(Segment是由log和index组成, 不仅log满了要Roll, index满了也要Roll)
- 离上次创建Segment的时间到达一定需要滚动的时间



```
private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] = new ConcurrentSkipListMap[Long, LogSegment]
loadSegments() // 加载Segments,会填充到segments链式Map中(因为

def activeSegment = segments.lastEntry.getValue // segments中最后一个Segment, 总是有效的
def logEndOffset: Long = nextOffsetMetadata.messageOffset // 从哪里开始写入

private def maybeRoll(messagesSize: Int): LogSegment = {
    val segment = activeSegment // segments中最后一个Segment, 总是有效的
    if (segment.size > config.segmentSize - messagesSize || segment.size == 0)
        roll() // 创建新的Segment, 此时如果调用activeSegment()，将返回新的Segment
    } else segment // 使用当前的Segment, 不需要创建新的Segment
}
```

当需要(滚动地)创建一个新的Segment,以当前的 `logEndOffset` 为起点,作为新 Segment的 `segmentBaseOffset` .

```

def roll(): LogSegment = {
    lock synchronized {
        val newOffset = logEndOffset // 最新的offset
        val logFile = logFilename(dir, newOffset) // Segment文件名
        val indexFile = indexFilename(dir, newOffset) // Segment索引文件名

        segments.lastEntry() match {
            case null => // 新创建的segment
            case entry => {
                entry.getValue.index.trimToValidSize() // 对index进行截断
                entry.getValue.log.trim()
            }
        }
        val segment = new LogSegment(dir, startOffset = newOffset,
        val prev = addSegment(segment) // 将新创建的segment添加到segments中
        // We need to update the segment base offset and append position
        updateLogEndOffset(nextOffsetMetadata.messageOffset)
        // schedule an asynchronous flush of the old segment
        scheduler.schedule("flush-log", () => flush(newOffset), delay)
        segment
    }
}

// Add the given segment to the segments in this log. If this segment
// segments的Key是每个Segment的baseOffset, 所以一个Partition中所有Segments的
def addSegment(segment: LogSegment) = this.segments.put(segment.key,

```

每个Segment由log和index组成,log是FileMessageSet,含有真正的消息.index是OffsetIndex:逻辑的offset到物理文件位置的索引.

Segment间有这样的关系:之前任何的segment的offset < 每个Segment的baseOffset <= 当前Segment任何消息最小的offset.

- 当前Segment的第一条消息的offset会作为这个Segment中所有消息的segmentBaseOffset
- 所以右边的式子成立,最小的offset=segmentBaseOffset, 后面消息的offset一定大于这个值
- 每个Segment都是按照时间顺序创建的,后面的Segment的最小的baseOffset一定大于前面的Segment的最大的offset



```
class LogSegment(val log: FileMessageSet, val index: OffsetIndex,
                 val baseOffset: Long, val indexIntervalBytes: Int)
  def this(dir: File, startOffset: Long, indexIntervalBytes: Int, r
          this(new FileMessageSet(file = Log.logFilename(dir, startOffset),
          new OffsetIndex(file = Log.indexFilename(dir, startOffset),
          startOffset, indexIntervalBytes, rollJitterMs, time)
}
```

updateLogEndOffset

有两个地方调用了updateLogEndOffset, 第一次就是在addSegment之后(貌似取值后又设置进去,没啥变化),

第二次是在往Segment添加消息之后(更新为LogAppendInfo.lastOffset+1,即最后一条消息的offset的下一个)

实际上在addSegment之后,activeSegment会变成新创建的那个Segment. 虽然messageOffset没有变化,但是后面两个都变了.

```
private def updateLogEndOffset(messageOffset: Long) {
  nextOffsetMetadata = new LogOffsetMetadata(messageOffset, active)
}
```

也就是说追加到roll segment,发生了两次updateLogEndOffset,而如果是已有的Segment,则只有一次update操作.

每次update操作都是新创建一个LogOffsetMetadata对象,nextOffsetMetadata是一个var变量,表示下一个offset的元数据.

虽然LogOffsetMetadata创建的时机不是很稳定的,但是它的作用域仍然是整个 Partition 范围的.

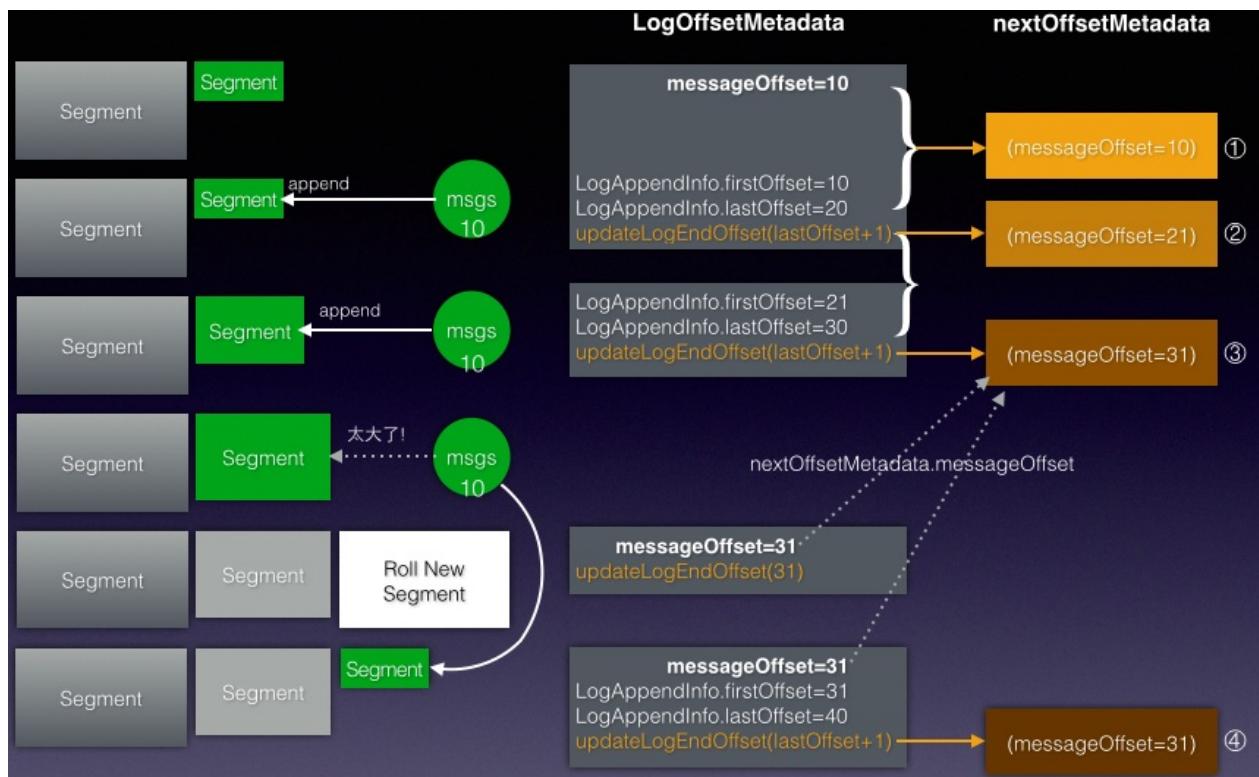
因为它的messageOffset要不断地更新:即使没有创建Segment,在每次追加数据到已

有Segment后都要

执行 `updateLogEndOffset(appendInfo.lastOffset + 1)`, 将messages最后一条消息的下一个作为其

`messageOffset`, 正如变量名 `nextOffsetMetadata`, 表示的是下一个 `offset: nextOffset` 的元数据信息

注意: 在同一批messages中, 并不需要每条消息都要更新LogEndOffset, 只有不同批次的messages才需要更新.



LogOffsetMetadata

- [1] the message offset 消息的偏移量, 可以认为是绝对偏移量, 即同一个Partition的所有Segment的绝对偏移位置
- [2] the base message offset of the located segment 当前Segment的base offset. 可以认为是相对偏移量
- [3] the physical position on the located segment 在Segment中的物理位置, 用于读取消息的具体内容

```
case class LogOffsetMetadata(messageOffset: Long, segmentBaseOffset: Long)
    // check if this offset is already on an older segment compared with the given offset
    def offsetOnOlderSegment(that: LogOffsetMetadata): Boolean = this.messageOffset < that.messageOffset
    // check if this offset is on the same segment with the given offset
    def offsetOnSameSegment(that: LogOffsetMetadata): Boolean = this.segmentBaseOffset == that.segmentBaseOffset

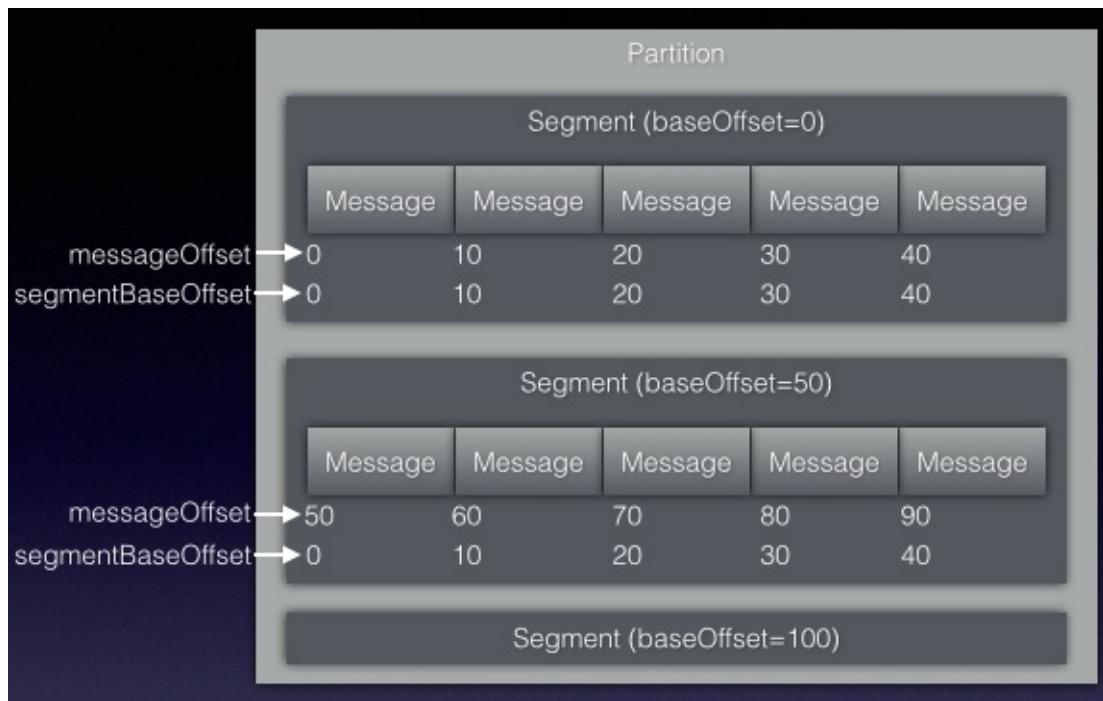
    // check if this offset is before the given offset
    def precedes(that: LogOffsetMetadata): Boolean = this.messageOffset < that.messageOffset
    // compute the number of messages between this offset to the given offset
    def offsetDiff(that: LogOffsetMetadata): Long = this.messageOffset - that.messageOffset

    // compute the number of bytes between this offset to the given offset
    def positionDiff(that: LogOffsetMetadata): Int = this.relativePositionInSegment - that.relativePositionInSegment
}
```

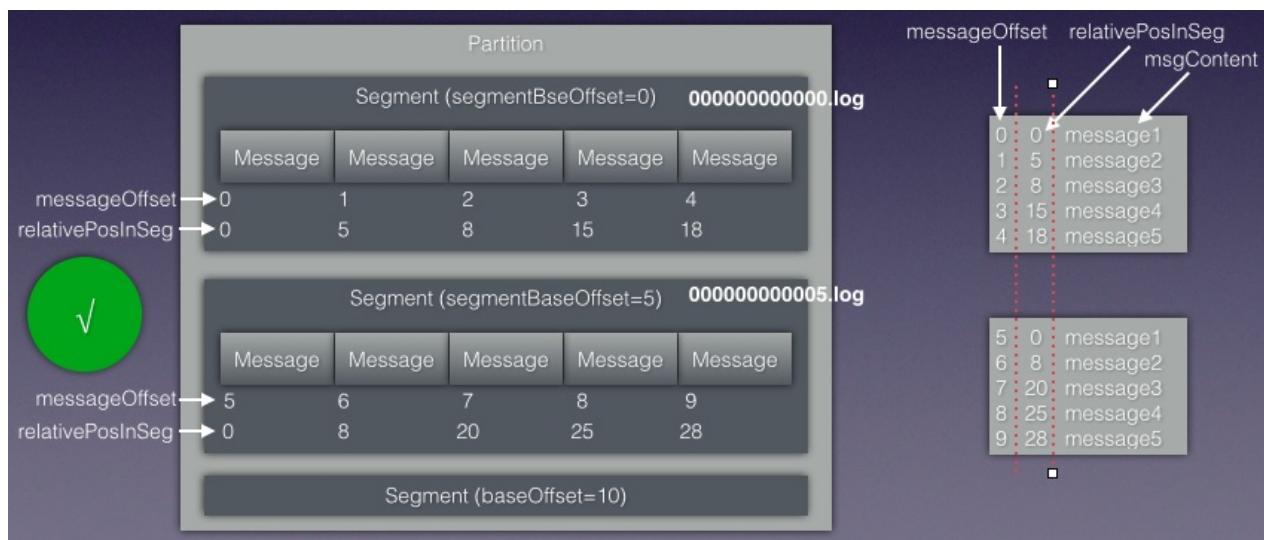
因为baseOffset是第一条消息的offset,所以同一个Segment的所有消息的baseOffset都是一样的.

- messageOffset: 针对Partition全局的绝对偏移量
- segmentBaseOffset: 每个Segment的第一条消息在Partition的messageOffset
- relativePositionInSegment: 只有在同一个Segment中,两个offset相减,表示两个offset之间的字节数

原先以为在同一个Segment中每条消息的segmentBaseOffset表示这条消息对于最开始消息的 相对偏移量 ,其实是错的.



实际上每条消息的offset都是单调递增的(连续的).而relativePositionInSegment是不连续的(因为每条消息的长度不同)



LogSegment append

messages追加到Segment的起始位置是:

```
segment.append	appendInfo.firstOffset, validMessages)
```

写入数据到log文件的同时,还要间隔indexIntervalBytes大小写入一条索引条目(并不是每条消息都写入一个索引)

```
//Append the given messages starting with the given offset. Add a new entry to the index.
def append(offset: Long, messages: ByteBufferMessageSet) {
    if (messages.sizeInBytes > 0) {
        if (bytesSinceLastIndexEntry > indexIntervalBytes) {
            index.append(offset, log.sizeInBytes()) // append to index
            this.bytesSinceLastIndexEntry = 0 // success
        }
        log.append(messages) // append to log
        this.bytesSinceLastIndexEntry += messages.sizeInBytes //统计
    }
}
```

FileMessageSet

创建LogSegment时,知道segmentBaseOffset,得到log和index的文件名,创建log文件,获取文件的FileChannel.

```
def append(messages: ByteBufferMessageSet) {
    val written = messages.writeTo(channel, 0, messages.sizeInBytes)
    _size.getAndAdd(written)
}

def sizeInBytes(): Int = _size.get()
```

ByteBufferMessageSet提供了写数据的方法,writeTo提供的 offset,size 是为了让channel从指定的offset开始写,
一共写入size大小(不是MessageSet中的offset和size概念!). 而这里需要把全部数据都写入channel中.

```

class ByteBufferMessageSet(val buffer: ByteBuffer) extends MessageSet {
    // Ignore offset and size from input. We just want to write the whole buffer.
    def writeTo(channel: GatheringByteChannel, offset: Long, size: Int): Unit = {
        buffer.mark()
        var written = 0
        while(written < sizeInBytes) written += channel.write(buffer)
        buffer.reset()
        written
    }
}

```

FileMessageSet的构造函数有start和end,提供这两个参数可以返回消息文件的子集/切片.

searchFor给定目标offset和起始位置,返回 \geq 目标offset最近的那个的文件位置.
它的作用是给定一个offset,要拉取这个offset之后的消息(这部分消息就是一个子集).

要确保startingPosition位于消息的边界,不过一般startPos=0,而不是文件随便某个位置.

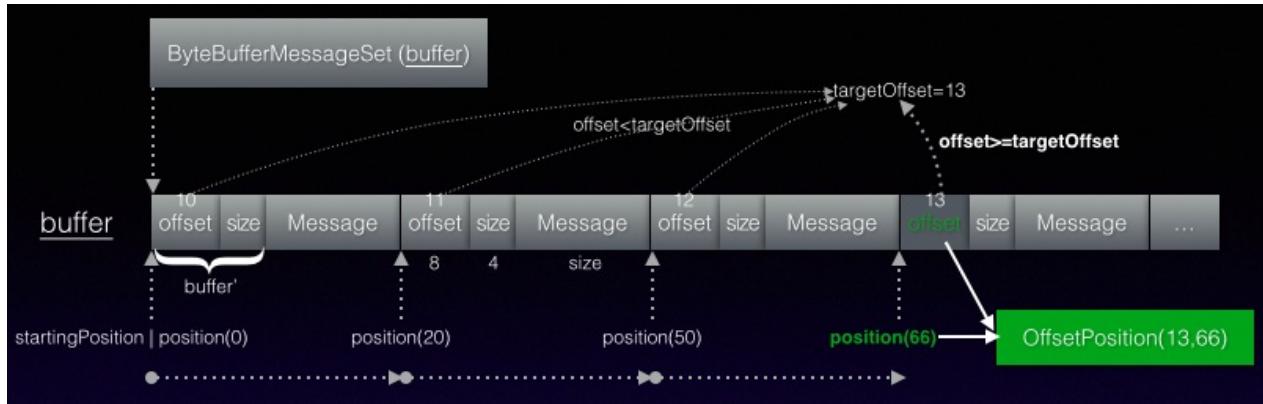
```

def searchFor(targetOffset: Long, startingPosition: Int): OffsetPosition = {
    var position = startingPosition
    val buffer = ByteBuffer.allocate(MessageSet.LogOverhead)          // 为读取准备一个缓冲区
    val size = sizeInBytes()                                         // 得到文件大小
    while(position + MessageSet.LogOverhead < size) {
        buffer.rewind()      // 在循环的过程中, 由于读取buffer, 指针后移, 所以重置指针
        channel.read(buffer, position)                                // 从文件的postition位置读取数据到buffer中, 读取
        buffer.rewind()      // 从文件的postition位置读取数据到buffer中, 读取
        val offset = buffer.getLong()                                 // 读取到的偏移量
        if(offset >= targetOffset) return OffsetPosition(offset, position)
        val messageSize = buffer.getInt()                            // 读取到的消息大小
        position += MessageSet.LogOverhead + messageSize           // 移动到下一个消息
    }
    null
}

```

OffsetPosition是这条消息的逻辑log偏移量和在日志文件中的物理位置(即在文件中的位置)

因为MessageSet中每条消息的大小都存在size中,所以不必真正读取这条消息就可以找到需要的offset.



OffsetIndex

OffsetIndex和FileMessageSet都是在创建LogSegment时同时创建. OffsetIndex还需要`baseOffset=segmentBaseOffset`.

- 索引文件映射offset到文件的物理位置(类似上面的OffsetPosition),它不会对每条消息都索引,所以是稀疏的.
- 由于offset和position的值不像消息内容变长, 所以每个索引条目都固定为8bytes. 即offset和position分别4byte
- MessageSet中是8byte的offset,而索引中是4byte,所以索引中存储的offset是减去`segmentBaseOffset`的值才放的下
- 由于是稀疏索引,所以可以把索引文件通过内存映射的方式将整个索引文件都放入内存中,可以加快offset的查询
- 因为offset是有序的,所以查询指定的offset时,使用二分查找就可以确定offset在哪个位置(当然也知道position了)
- 可能指定的offset在索引文件中(0,5,8)不存在,但是可以找到小于等于指定offset(6)的最大offset(5)
- 由于索引文件随着消息的追加而不断变化,打开索引文件: 空 的允许 追加 的 可变 索引或 不可变 的 只读 的索引文件
- 当索引文件发生roll时,索引文件都不会变化了(类似日志文件的滚动),这时可以把可变的索引文件转为不可变的索引文件

OffsetIndex中每个条目的物理格式是4字节的 offset和4字节的文件位置(消息在log中的物理位置).

已经知道绝对offset和position的对应关系的(OffsetPosition),绝对offset-baseOffset=相对offset

假设baseOffset=50, 消息1的offset=55(这是一个绝对offset,针对partition范围), 相对offset=55-50=5 消息2的offset=56,相对offset=56-50=6,即索引条目中的offset都是相对于segmentBaseOffset的差值.

这样只需要4个字节就可以放的下,而且每条消息的offset是单调递增的,以一个基线为准,存储的数据量就会少很多.

带来的额外工作是需要将相对offset转换为全局/绝对offset,因为外部给的offset一定是一个绝对offset.

```
// Append an entry for the given offset/location pair to the index.
def append(offset: Long, position: Int) {
    inLock(lock) {
        if (size.get == 0 || offset > lastOffset) {
            this.mmap.putInt((offset - baseOffset).toInt) // 相对于segmentBaseOffset
            this.mmap.putInt(position) // offset对segmentBaseOffset的偏移量
            this.size.incrementAndGet()
            this.lastOffset = offset
            require(entries * 8 == mmap.position, entries + " entries took " +
                (mmap.position - entries * 8) + " bytes")
        }
    }
}
```

二分查找,先将目标offset减去baseOffset,因为索引存的都是相对offset.

```
// Find the slot in which the largest offset less than or equal to targetOffset
// -1 if the least entry in the index is larger than the target offset
private def indexSlotFor(idx: ByteBuffer, targetOffset: Long): Int = {
    val relOffset = targetOffset - baseOffset // we only care about relative offsets
    if(entries == 0) return -1 // check if there are any entries
    if(relativeOffset(idx, 0) > relOffset) return -1 // check if first entry is greater than target
    // binary search for the entry
    var lo = 0
    var hi = entries-1
    while(lo < hi) {
        val mid = ceil(hi/2.0 + lo/2.0).toInt
        val found = relativeOffset(idx, mid)
        if(found == relOffset) return mid
        else if(found < relOffset) lo = mid
        else hi = mid - 1
    }
    lo
}
```

Ref

- <http://www.cnblogs.com/fxjwind/p/4913703.html>
- <http://www.cnblogs.com/huxi2b/p/4583249.html>

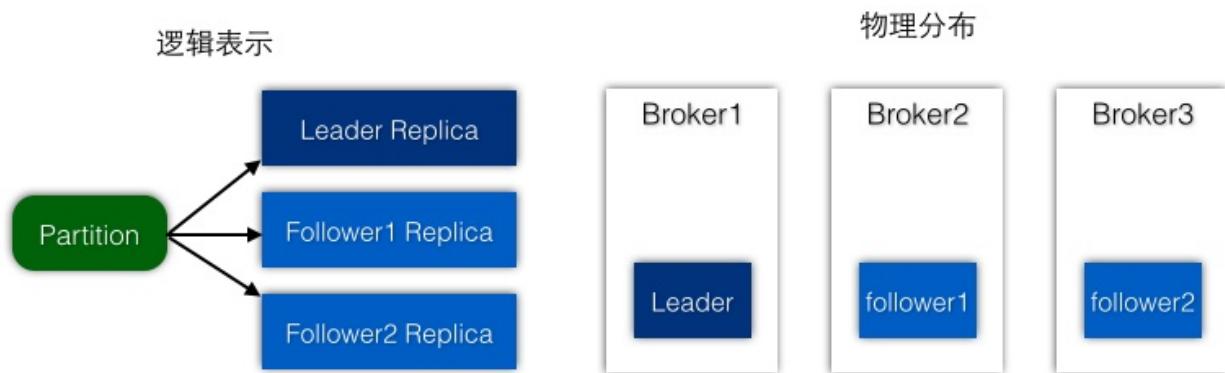
kafka-ISR

Partition & Replica

首先来看Partition的Replication副本是个什么概念. 每个Partition都可以有多个Replication.

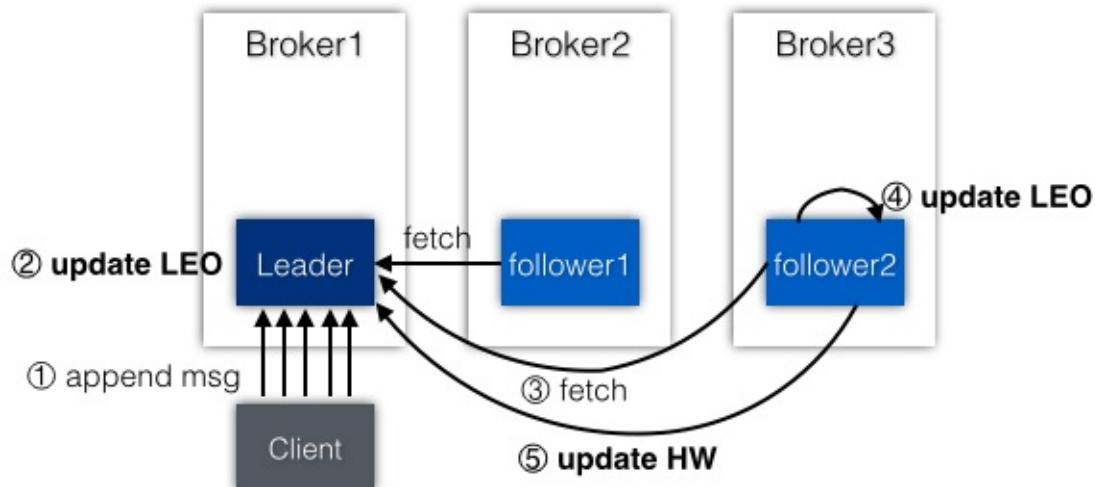
其中Leader副本负责读写, Follower副本负责从Leader拉取数据. Replica分布在不同的Broker上.

所以Replica只需要两个属性: Partition(分区), brokerid(所在的Kafka节点).



一个Replication有两个重要的元数据: HighWatermark和LogEndOffset.

- HighWatermark是用来确保消费者能获取到的消息的最高水位,超过这个水位的消息是不会被客户端看到的.
- 由于Leader负责读写,所以HW只能由Leader更新,但是什么时候更新,可能由follower在更新LEO时通知Leader修改.
- LogEndOffset是所有的Replica都会有的:Leader在消息追加后会更新,follower在从Leader抓取消息也也会更新.



问:最后一个可选的Log是怎么用来判断是否是本地的Replica? Local和Remote又是针对什么而言?

答:一个Partition的所有Replicas都是保存在Leader节点的内存中的. 而不是让每个节点自己管理自己的信息,

如果这样的话,每个Kafka节点的信息就都是不一样的! 而分布式集群管理是要有一个中心来管理所有节点信息的.

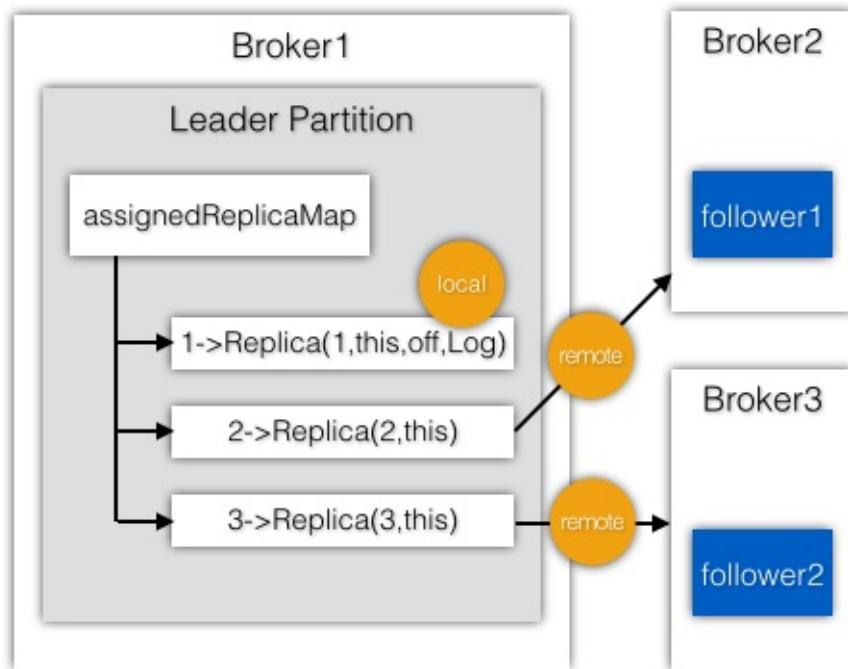
因为现在是在Leader节点上,并且是由Leader管理所有的Replicas,Leader自己就是Local,其他Replicas都是Remote!

即 Leader Replica = Local Replica , Follower Repicas = Remote Replicas .

问:因为Partitions是分布在不同的Kafka节点,本身每个节点记录的Partitions就是不一样的了.

答:没错,分布式节点保存的信息是不一样的,但是同一个Partition的所有Replicas应该是交给Leader管理的.

而follower上的Replicas并不需要管理自己的状态,因为Leader替他们管理好了.



```

class Replica(val brokerId: Int, val partition: Partition, time: Time) {
    // the high watermark offset value, in non-leader replicas only
    @volatile private[this] var highWatermarkMetadata: LogOffsetMetadata = LogOffsetMetadata.Earliest
    // the log end offset value, kept in all replicas;
    // for local replica it is the log's end offset, for remote replica it is the offset of the last log entry
    @volatile private[this] var logEndOffsetMetadata: LogOffsetMetadata = LogOffsetMetadata.Earliest

    def isLocal: Boolean = log match {
        case Some(l) => true      // 创建Replica时指定Log时，则表示是本地的Replica
        case None => false
    }

    // 设置LEO：不应该在Local Replication的Partition上设置LEO。对于Local Replication，LEO由Log对象提供
    private def logEndOffset_=_(newLogEndOffset: LogOffsetMetadata) {
        if (!isLocal) logEndOffsetMetadata = newLogEndOffset
    }

    // 获取LEO
    def logEndOffset = if (isLocal) log.get.logEndOffsetMetadata else logEndOffsetMetadata

    // 设置HW：不应该在非Local Replication的Partition上设置HW，即只能在Local Replication的Partition上设置HW
    def highWatermark_=_(newHighWatermark: LogOffsetMetadata) {
        if (isLocal) highWatermarkMetadata = newHighWatermark
    }

    // 获取HW
    def highWatermark = highWatermarkMetadata
}

```

在Partition中创建Replica,如果不在 assignedReplicaMap 中,根据是否是本地(Leader)来创建Replica实例.

Local的Replica比Remote的多了offset和Log. Replica的isLocal()会根据是否有Log判断是不是Local.

```

def getOrCreateReplica(replicaId: Int = localBrokerId): Replica = {
    val replicaOpt = getReplica(replicaId)
    replicaOpt match {
        case Some(replica) => replica
        case None =>
            if (isReplicaLocal(replicaId)) {
                val config = LogConfig.fromProps(logManager.defaultConfig)
                // TopicAndPartition的Log
                val log = logManager.createLog(TopicAndPartition(topic, partition))
                // log.dirs是所有TopicAndPartition的父目录，而checkpoints也是
                val checkpoint = replicaManager.highWatermarkCheckpoints()
                val offsetMap = checkpoint.read
                // checkpoints中记录了所有Partition的offset信息，所以可以根据topic和partition
                if (!offsetMap.contains(TopicAndPartition(topic, partition))) {
                    val offset = offsetMap.getOrElse(TopicAndPartition(topic, partition), 0L)
                    // 本地的是Leader，所以要给出offset和Log.
                    val localReplica = new Replica(replicaId, this, time, offset)
                    addReplicaIfNotExists(localReplica)
                } else {
                    // 远程的是follower.
                    val remoteReplica = new Replica(replicaId, this, time)
                    addReplicaIfNotExists(remoteReplica)
                }
                getReplica(replicaId).get
            }
    }
}

```

注意上面的 `log.dir` 是TopicAndPartition的目录(每个TopicAndPartition目录是唯一的).

`log.dir.getParentFile` 指的是配置文件中的 `log.dirs`,而不是Partition的目录.

而 `log.dirs` 是server.properties的log.dirs配置项,它是所有TopicAndPartition的父目录.

getOrCreateReplica被调用的地方是在
ReplicaManager.becomeLeaderOrFollower->makeFollowers

ReplicaManager->OffsetCheckpoint

Local Replica的offset来源于High watermark的checkpoint(因为HW很重要,所有需要做检查点).

ReplicaManager的highWatermarkCheckpoints是一个Map:日志目录(log.dirs)->OffsetCheckpoint.

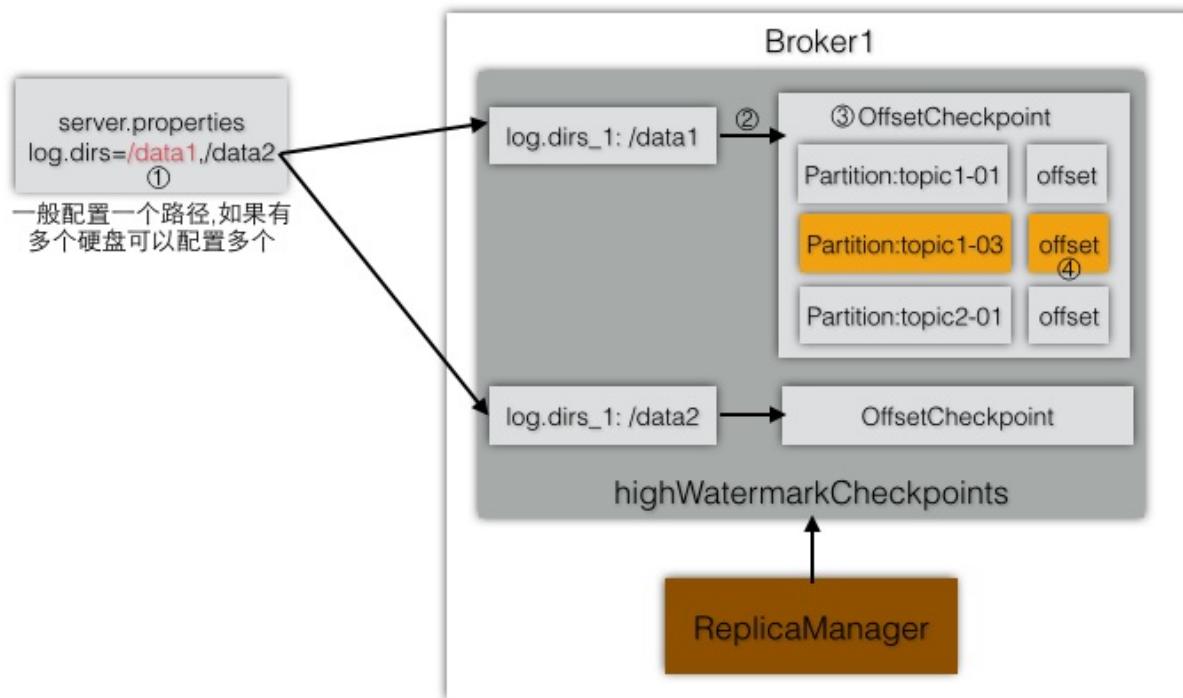
```
val highWatermarkCheckpoints = config.logDirs.map(dir =>
  (new File(dir).getAbsolutePath, new OffsetCheckpoint(new File(dir).getAbsolutePath)))
.toMap
```

下面的/data/kafka目录是server.properties中的log.dirs的配置项,会在这个目录下生成checkpoint文件.

```
→ kafka 11
-rw-r--r-- 1 zhengqh staff      0B  1 14 16:13 recovery-point-offset
-rw-r--r-- 1 zhengqh staff     18B  1 14 16:13 replication-offset
drwxr-xr-x 4 zhengqh staff   136B  1 14 16:13 wikipedia-0
→ kafka 11 wikipedia-0
-rw-r--r-- 1 zhengqh staff    10M  1 14 16:13 00000000000000000000000000000000
-rw-r--r-- 1 zhengqh staff      0B  1 14 16:13 00000000000000000000000000000000
```

ReplicaManager是管理所有的Partition的,而checkpoints里记录的offsetMap是所有Partition共用的.

所以可以根据某个特定的TopicAndPartition找到它在checkpoints中对应的offset,用来创建Replica.



因为OffsetCheckpoint记录的是TopicAndPartition到offset的映射关系.所以这个类中只是文件的读写操作.

```
class OffsetCheckpoint(val file: File) extends Logging {
  def write(offsets: Map[TopicAndPartition, Long]) {
    // write the current version and the number of entries, then the offsets
    offsets.foreach { case (topicPart, offset) =>
      writer.write("%s %d %d".format(topicPart.topic, topicPart.partition, offset))
      writer.newLine()
    }
  }
  def read(): Map[TopicAndPartition, Long] = {
    var offsets = Map[TopicAndPartition, Long]()
    line = reader.readLine()
    while(line != null) {
      val pieces = line.split("\\s+")
      val topic = pieces(0)
      val partition = pieces(1).toInt
      val offset = pieces(2).toLong
      offsets += (TopicAndPartition(topic, partition) -> offset)
      line = reader.readLine()
    }
  }
}
```

ReplicaManager是在 `becomeLeaderOrFollower` 调度Checkpoint的写入,这个线程是定时运行的,确保HW是最新的.

```

private val highWatermarkCheckPointThreadStarted = new AtomicBoolean(false)
private val allPartitions = new Pool[(String, Int), Partition]

def startHighWaterMarksCheckPointThread() = {
    if(highWatermarkCheckPointThreadStarted.compareAndSet(false, true))
        scheduler.schedule("highwatermark-checkpoint", checkpointHighWatermarks)
}

// Flushes the highwatermark value for all partitions to the highwatermark
def checkpointHighWatermarks() {
    val replicas = allPartitions.values.map(_.getReplica(config.brokers))
    val replicasByDir = replicas.filter(_.log.isDefined).groupByKey(_.topic)
    for((dir, reps) <- replicasByDir) {
        val hwms = reps.map(r => (new TopicAndPartition(r) -> r.highWatermark))
        highWatermarkCheckpoints(dir).write(hwms)
    }
}

```

如何根据Partition得到offset: Partition->Replica->LogOffsetMetadata->messageOffset.

- 找出所有的Partitions, 获取其Replica, 确保有Replica的Partition
- 根据log.dirs重新分组(确保有Log), 每个log.dirs对应了replicas列表
- 对每个log.dirs, 循环replicas, 转换成TopicAndPartition到HW的映射
- 最终往每个log.dirs写入了属于这个dir的TopicAndPartition->offset信息

问: HW是由Leader更新的,Broker节点存储的并不都是Leader Partitions,在做checkpoint时是只对Leader Partition做吗?

答:不是的,ReplicaManager是对所有Partitions做checkpoint的. 虽然只有Leader更新HW,但是Leader也会将HW广播给follower的.

当follower向leader fetch request的时候,leader会把hw也传给follower,这样follower也有了hw信息. 这样同一个Partition的所有Replica都有了hw信息. 目的是即使Leader挂掉了,这个hw仍然会保留在其他Replica上,其他replica成为leader后,hw也不会丢失.

问:Replica的highWatermark_=方法不是只有Leader才能调用吗(isLocal),那么leader广播给follower的hw是如何被更新的?

ReplicaManager.allPartitions

allPartitions被放入也是在 `becomeLeaderOrFollower` 中.这说明一个Partition在 Leader和follower转换时是要做很多工作的.

```
def getOrCreatePartition(topic: String, partitionId: Int): Partition
    var partition = allPartitions.get((topic, partitionId))
    if (partition == null) {
        allPartitions.putIfNotExists((topic, partitionId), new Partition)
        partition = allPartitions.get((topic, partitionId))
    }
    partition
}
```

allPartitions记录的是当前节点的所有Partitions.这些Partitions并不都是Leader.

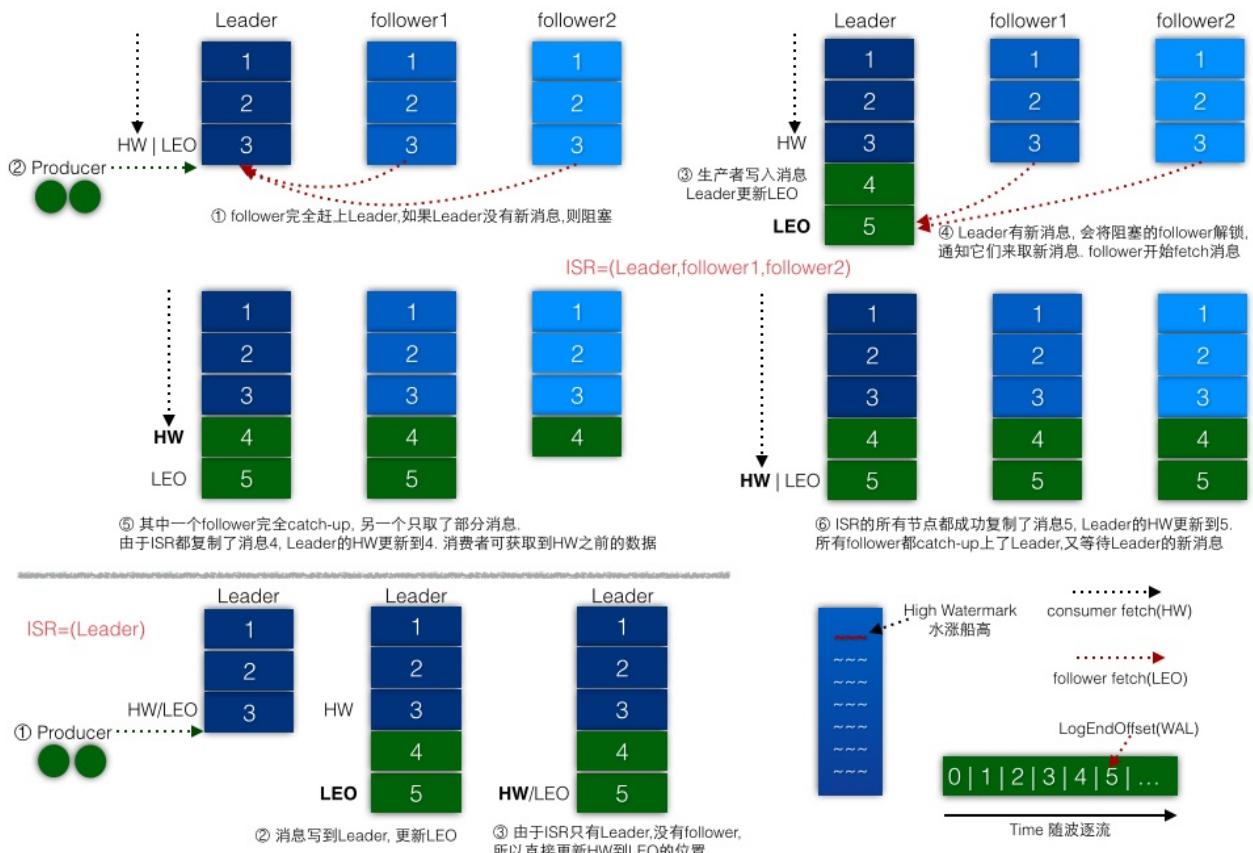
```
private def getLeaderPartitions(): List[Partition] = {
    allPartitions.values.filter(_.leaderReplicaIfLocal().isDefined)
}
private def maybeShrinkIsr() {
    allPartitions.values.foreach(_.maybeShrinkIsr(config.replicaLag))
}
```

留个坑: `becomeLeaderOrFollower`的调用以及ISR收缩时对HW的影响.

ReplicaManager.appendMessages

在`Log.append`之后,需要通知follower获取这些新的消息.因为现在Leader的LEO已经更新了,follower需要及时获取落后的消息

如果Partition的ISR只有1个,说明没有其他Replication,则Leader的LEO更新后,其HW也要一起更新(当然这是特殊情况)



为了更方便地查看在append之后的操作, 我把相关的代码都列在一起了. 之前分析过的就省略了.

```
//*****ReplicationManager.scala*****
private def appendToLocalLog(internalTopicsAllowed: Boolean, messagesPerPartition: Map[TopicPartition, ByteBufferMessageSet], requiredAcks: Int, requiredReplicas: Int, requiredOffsetSync: Boolean): Unit = {
  messagesPerPartition.map { case (topicAndPartition, messages) =>
    val info = partition.appendMessagesToLeader(messages.asInstanceOf[ByteBufferMessageSet], requiredAcks, requiredReplicas, requiredOffsetSync)
    (topicAndPartition, LogAppendResult(info))
  }
}

//*****Partition.scala*****
def appendMessagesToLeader(messages: ByteBufferMessageSet, requiredAcks: Int, requiredReplicas: Int, requiredOffsetSync: Boolean): LogAppendResult = {
  val (info, leaderHWIncremented) = inReadLock(leaderIsrUpdateLock) {
    // ① 写到Partition的Leader的那个Log
    val log = leaderReplica.log.get
    val info = log.append(messages, assignOffsets = true)
    // ② probably unblock some follower fetch requests since
    replicaManager.tryCompleteDelayedFetch(new TopicPartition(topicAndPartition.topic, topicAndPartition.partition))
    // ③ we may need to increment high watermark since ISR changes
    (info, maybeIncrementLeaderHW(leaderReplica))
  }
}
```

```

    // some delayed operations may be unblocked after HW changed
    // ④ 如果HW改变了,则一些延迟的请求(针对消费者)需要被解锁.当然如果HW没有
    if (leaderHWIncremented) tryCompleteDelayedRequests()
}

//*****ReplicationManager.scala*****
def appendMessages(timeout: Long, ... ){
    val localProduceResults = appendToLocalLog(internalTopicsAllocated)
    val produceStatus = localProduceResults.map { case (topicAndPartition, result) =>
        topicAndPartition -> ProducePartitionStatus(result.info.lastOffset)
    }
    if (delayedRequestRequired(requiredAcks, messagesPerPartition)) {
        // ⑤ create delayed produce operation 创建延迟的Produce操作
        val produceMetadata = ProduceMetadata(requiredAcks, produceStatus)
        val delayedProduce = new DelayedProduce(timeout, produceMetadata)
        // create a list of (topic, partition) pairs to use as keys
        val producerRequestKeys = messagesPerPartition.keys.map(new TopicPartition(_))
        // ⑥ try to complete the request immediately, otherwise put it in purgatory
        // this is because while the delayed produce operation is being processed,
        // we can't respond to the client until it's completed
        delayedProducePurgatory.tryCompleteElseWatch(delayedProduce)
    } else {
        // ⑦ we can respond immediately 如果不需要等待ISR同步数据,在成功写到Leader之后,就可以返回响应给Producer了
        val produceResponseStatus = produceStatus.mapValues(status => new ProducerResponseStatus(status))
        responseCallback(produceResponseStatus)
    }
}

```

①-④ 添加到Leader的本地日志后,要解锁follower的fetch,这样ISR中的follower才能尽快地同步新增的消息.

⑤-⑥ 如果需要ISR中的follower replicas同步数据,则需要创建延迟的ProducerRequest(生产请求),并交给专门的Purgatory处理

⑦ 不需要等待ISR的数据同步,在成功写到Leader之后,就可以返回响应给Producer了

注意这里创建的DelayedProduce和Producer相关的几个对象的关系.

- produceStatus: 生产者的状态,主要是TopicAndPartition和ProducePartitionStatus的映射.会有多个Partition对应自己的状态
- produceMetadata: 包含了上面的生产者状态,以及requiredAcks来自Producer的acks设置

- responseCallback: 回到函数, 来自于appendMessages外面传入的(来源于 KafkaApis 定义的回调函数<--客户端回调)
- produceResponseStatus: 最终返回给客户端的响应信息, 在请求完成时, 会把响应状态传给回调函数, 然后调用回调函数, 完成客户端的回调

ReplicaManager->DelayOperation

这里先简单看下生产请求需要被延迟的场景, 后面一篇专门讲为什么需要延迟(以及使用缓存来保存和移除操作)

delayedRequestRequired

在appendToLocalLog后, 满足下面的所有条件时, 将会产生一个延迟的Produce请求, 并且等待replication完成:

- required acks = -1 : Producer需要等待所有ISR接收数据(这个最重要了, 不过由客户端指定的)
- there is data to append : 有数据(废话, 没数据怎么叫生产消息)
- at least one partition append was successful 至少一个Partition追加成功(一次请求有多个Partition)

```
private def delayedRequestRequired(requiredAcks: Short, messagesPerPartition: Map[TopicAndPartition, Seq[LogAppendResult]]) = {
    requiredAcks == -1 && messagesPerPartition.size > 0 && localProducer != null
}
```

DelayedProduce

localProduceResults -> (TopicAndPartition, LogAppendResult) -> Produce	本地(Leader)的生产结果 -> 日志追加结果 -> Produce
--	--------------------------------------

延迟的Produce操作会被ReplicaManager创建(new DelayedProduce), 并被 ProduceOperationPurgatory 监视.

```
// A delayed produce operation that can be created by the replica manager
class DelayedProduce(delayMs: Long, produceMetadata: ProduceMetadata,
                      responseCallback: Map[TopicAndPartition, ProduceResponse]) extends DelayedOperation {
    override val expirationMs = delayMs + System.currentTimeMillis()
}
```

DelayedProduce的 delayMs 以及 acks 源头来自于 KafkaApis.handleProducerRequest 的 ProduceRequest:

```
replicaManager.appendMessages(produceRequest.ackTimeoutMs.toLong)
```

DelayedOperationPurgatory

ReplicaManager 的 append 操作出现了三个 tryCompleteXXX(Fetch, Request, ElseWath) 都是交给 DelayedOperationPurgatory 炼狱工厂.

```
class ReplicaManager {
    val delayedProducePurgatory = new DelayedOperationPurgatory[DelayedProduce]
    val delayedFetchPurgatory = new DelayedOperationPurgatory[DelayedFetch]
    def tryCompleteDelayedFetch(key: DelayedOperationKey) { val completionFuture = delayedFetchPurgatory.tryComplete(key) }
    def tryCompleteDelayedProduce(key: DelayedOperationKey) { val completionFuture = delayedProducePurgatory.tryComplete(key) }

    // A helper purgatory class for bookkeeping delayed operations with their purgatory names
    class DelayedOperationPurgatory[T <: DelayedOperation](purgatoryName: String) {
        private var pendingOperations = Map[DelayedOperationKey, T]()
        private var completedOperations = Map[DelayedOperationKey, T]()

        def tryComplete(key: DelayedOperationKey): Future[T] = {
            pendingOperations.get(key).map { op => completedOperations += (key, op) }.getOrElse(Future.successful(null))
        }
        def tryComplete(key: DelayedOperationKey, op: T): Unit = pendingOperations += (key, op)
    }
}
```

A. 尝试完成 延迟的fetch请求 的触发条件:

- A.1 Partition 的 HW 发生变化 (正常的 fetch-- 即 consumer 的 fetch, 因为 HW 是针对消费者而言, 消费者最多只能到 HW)
- A.2 新的 MessageSet 追加到本地日志 (follower 的 fetch, 新的 MessageSet 有新的 LEO, 而 follower 是跟踪 LEO 的)

B. 尝试完成 延迟的Produce请求 的触发条件:

- B.1 Partition的HW发生变化 (对于acks=-1的情况)
- B.2 收到了一个follower副本的fetch操作 (对于acks>1的情况? acks不是只有-1,0,1三个值吗?)

Partition->IncrementLeaderHW

HW表示的是所有ISR中的节点都已经复制完的消息.也是消费者所能获取到的消息的最大offset,所以叫做high watermark.

注意Leader Partition保存了ISR信息.所以可以看到
maybeIncrementLeaderHW()是在appendToLocalLog()内一起执行的

C. 增加(Leader)Partition的Hight Watermark(HW)的触发条件:

- C.1 (Leader)Partition的ISR发生变化 (假设某个很慢的节点落后很多从ISR中移除,而其他节点大部分都catch-up,就可以更新HW)
- C.2 任何Replication的LEO 变化 (ISR中的followers有任何一个节点LEO改变,看看所有ISR是否都复制了,然后更新HW)

```
private def maybeIncrementLeaderHW(leaderReplica: Replica): Boolean
    // 所有inSync副本中最小的LEO(因为每个follower的LEO都可能不一样), 表示所有
    val allLogEndOffsets = inSyncReplicas.map(_.logEndOffset)
    val newHighWatermark = allLogEndOffsets.min(new LogOffsetMetadata)

    // Leader本身的hw, 是旧的
    val oldHighWatermark = leaderReplica.highWatermark // 是一个LogOffsetMetadata
    if(oldHighWatermark.precedes(newHighWatermark)) { // 比较Leader和旧的
        leaderReplica.highWatermark = newHighWatermark // Leader小的
        true
    }else false // Returns true if the HW was incremented, and
}
```

在Log中append的updateLogEndOffset(LogAppendInfo.lastOffset+1)更新的是
LogOffsetMetadata的messageOffset

而这里的oldHighWatermark和newHighWatermark也都是LogOffsetMetadata(最重

要的就是messageOffset字段了)!

所以实际比较的是两个LogOffsetMetadata的messageOffset, 只有 Leader的HW 小于 ISR最小的LEO ,才更新Leader的HW.

- leaderReplica 是Partition的Leader副本,一个Partition只有一个Leader,读写都发生在Leader上
- inSyncReplicas 是Partition的follower中追赶上Leader的副本(并不是所有Follower都是InSync)

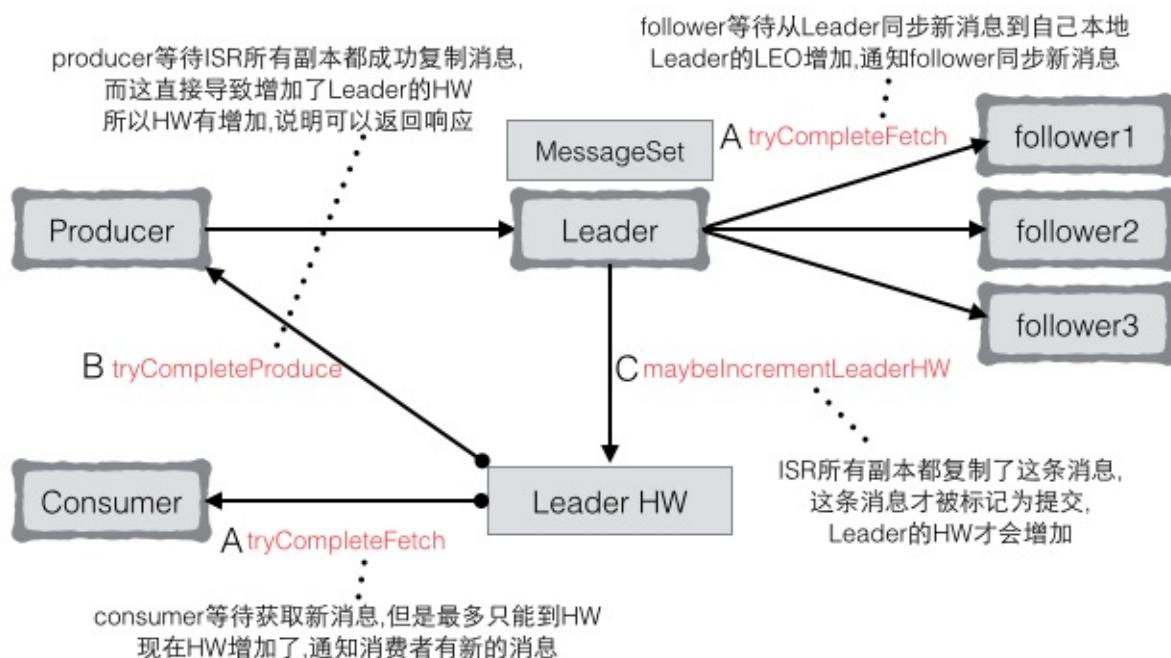
这两个属性都是Partiton级别的,即Partiton要知道它的leader是哪个,以及这个leader管理的isr列表是什么.

其他

delay operation complete

触发条件(延迟请求以及增加HW)中关于ISR的部分都是环环相扣的:

- leader有新消息写到本地日志(生产者写新数据) --> A.2 --> DelayedFetch
- leader replication的LEO发生变化(追加了新消息) --> C.2 --> HW
- follower向Leader发起fetch请求(ISR的follower会和Leader保持同步) --> B.2 --> DelayedProduce
- follower所在replication的LEO发生变化(拉取了新消息到本地) --> C.2 --> HW
- 所有replication的LEO发生变化,Leader的HW也会变化(成功提交了消息) --> C.2 --> HW
- consumer读取至多Leader的HW,HW变化了,解锁consumer --> A.1 --> DelayedFetch
- producer等待ISR都同步成功,导致HW变化,就可以返回响应 --> B.1 --> DelayedProduce



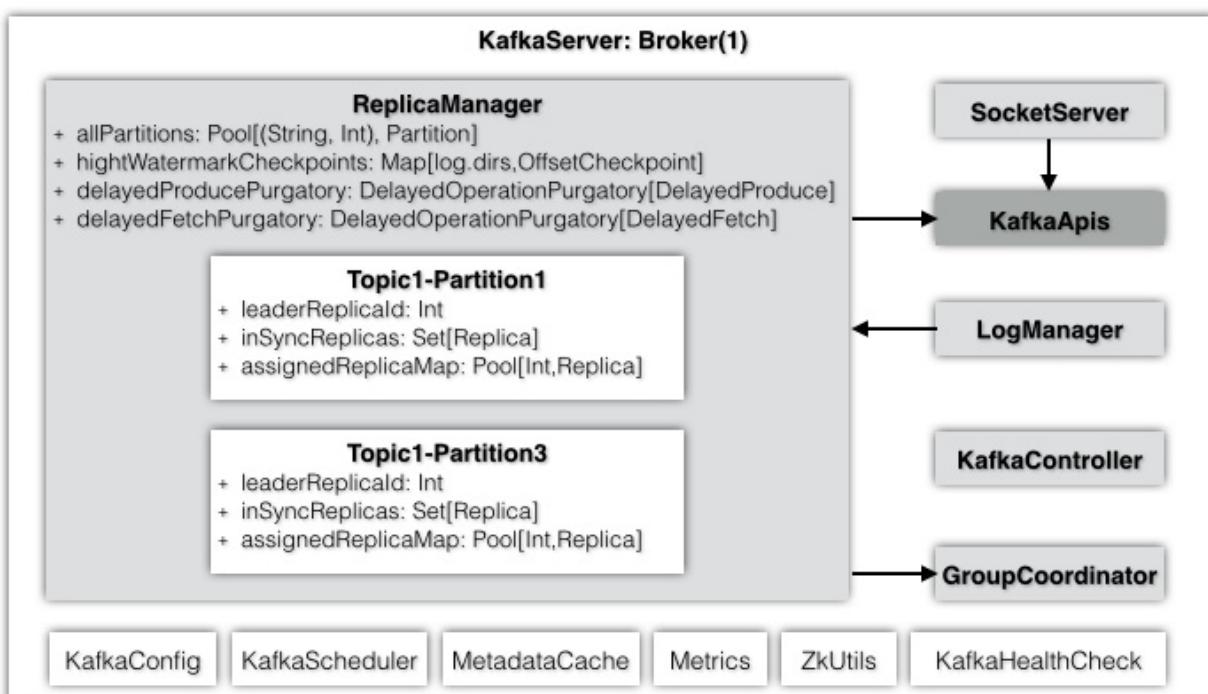
Partition and Replica Creation

ReplicaManager负责管理Partition, 所以是在ReplicaManager创建Partition的:
getOrCreatePartition

Partition负责管理Replica, 所以是在Partition中创建Replica的: getOrCreateReplica

ReplicaManager要管理Broker上的所有Partition, allPartitions: Pool[(String, Int), Partition]

Partition也要管理分配的所有Replica, 还有Leader Replica和ISR.



结语

- Partition副本由Leader和follower组成,只有ISR列表中的副本是仅仅跟着Leader的
- Leader管理了ISR列表,只有ISR列表中的所有副本都复制了消息,才能认为这条消息是提交的
- Leader和follower副本都叫做Replica,同一个Partition的不同副本分布在不同Broker上
- Replica很重要的两个信息是HighWatermark(HW)和LogEndOffset(LEO)
- 只有Leader Partition负责客户端的读写,follower从Leader同步数据
- 所有Replica都会对HW做checkpoint,Leader会在follower的拉取请求时广播HW给follower

Ref

- <http://www.jasongj.com/2015/04/24/KafkaColumn2/>
- <https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication>

kafka-Consumer-Scala

Consumer Example(old and high-level)

消费者示例, 指定要消费的topic和线程数, 返回每个topic对应的KafkaStream列表, 每个线程对应一个KafkaStream.

下面的示例中只使用了一个线程, 所以通过streams.get(0)获取到该线程对应的 KafkaStream. 然后从流中读取出消息.

topicCountMap表示客户端可以同时消费多个topic, 那为什么要设置线程数呢? 因为一个topic有多个partition分布在多个broker节点上. 即使是同一个broker, 也可能有这个topic的多个partition. 用不同的线程来隔离不同的partition.

```
ConsumerConfig conf = new ConsumerConfig(props);
ConsumerConnector consumer = kafka.consumer.Consumer.createJavaConsumerConnector(conf);
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put(topic, new Integer(1));
Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createConsumer(topicCountMap);

List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);
KafkaStream<byte[], byte[]> stream = streams.get(0);
ConsumerIterator<byte[], byte[]> it = stream.iterator();
while (it.hasNext()){
    System.out.println("message: " + new String(it.next().message()));
}
```

ConsumerConnector

Consumer定义在ConsumerConnector接口同一个文件中. 它默认创建的 ConsumerConnector是基于ZK的ZookeeperConsumerConnector

```

object Consumer extends Logging {
    def createJavaConsumerConnector(config: ConsumerConfig): kafka.javaapi.consumer.ConsumerConnector = {
        val consumerConnect = new kafka.javaapi.consumer.ZookeeperConsumerConnector(config)
        consumerConnect
    }
}

```

ConsumerConnector主要有创建消息流(createMessageStreams)和提交offset(commitOffsets)两种方法.

Consumer会根据消息流消费数据, 并且定时提交offset. 由客户端自己保存offset是kafka采用pull拉取消息的一个附带工作.

```

trait ConsumerConnector {
    def createMessageStreams(topicCountMap: Map[String, Int]): Map[String, ConsumerStream]
    def createMessageStreams[K, V](topicCountMap: Map[String, Int], keyDecoder: Decoder[K], valueDecoder: Decoder[V]): Map[String, ConsumerStream[K, V]]
    def createMessageStreamsByFilter[K, V](topicFilter: TopicFilter, keyDecoder: Decoder[K], valueDecoder: Decoder[V]): Map[String, ConsumerStream[K, V]]
    def commitOffsets(retryOnFailure: Boolean)
    def commitOffsets(offsetsToCommit: immutable.Map[TopicAndPartition, OffsetAndTimestamp])
    def setConsumerRebalanceListener(listener: ConsumerRebalanceListener)
    def shutdown()
}

```

ZookeeperConsumerConnector

一个Consumer会创建一个ZookeeperConsumerConnector, 代表一个消费者进程.

- fetcher: 消费者获取数据, 使用ConsumerFetcherManager fetcher线程抓取数据
- zkUtils: 消费者要和ZK通信, 除了注册自己, 还有其他信息也会写到ZK中
- topicThreadIdAndQueues: 消费者会指定自己消费哪些topic, 并指定线程数, 所以topicThreadId都对应一个队列
- messageStreamCreated: 消费者会创建消息流, 每个队列都对应一个消息流
- offsetsChannel: offset可以存储在ZK或者kafka中, 如果存在kafka里, 像其他请求一样, 需要和Broker通信
- 还有其他几个Listener监听器, 分别用于topicPartition的更新, 负载均衡, 消费者重

新负载等

```

private[kafka] class ZookeeperConsumerConnector(val config: ConsumerConfig)
    extends ConsumerConnector with Logging with KafkaMetricsGroup
{
    private var fetcher: Option[ConsumerFetcherManager] = None
    private var zkUtils: ZkUtils = null
    private var topicRegistry = new Pool[String, Pool[Int, Partition]]
    private val checkpointedZkOffsets = new Pool[TopicAndPartition, Long]
    private val topicThreadIdAndQueues = new Pool[(String, ConsumerThreadId), Queue[OffsetAndWatermark]]
    private val scheduler = new KafkaScheduler(threads = 1, threadNamePrefix = "consumer")
    private val messageStreamCreated = new AtomicBoolean(false)
    private var offsetsChannel: BlockingChannel = null
    private var sessionExpirationListener: ZKSessionExpireListener = null
    private var topicPartitionChangeListener: ZKTopicPartitionChangeListener = null
    private var loadBalancerListener: ZKRebalancerListener = null
    private var wildcardTopicWatcher: ZookeeperTopicEventWatcher = null
    private var consumerRebalanceListener: ConsumerRebalanceListener = null

    connectZk()                                // ① 创建ZkUtils,会创建对应的ZkConsumer
    createFetcher()                             // ② 创建ConsumerFetcherManager
    ensureOffsetManagerConnected()             // ③ 确保连接上OffsetManager.
    if (config.autoCommitEnable) {           // ④ 启动定时提交offset线程
        scheduler.startup
        scheduler.schedule("kafka-consumer-autocommit", autoCommit, delay)
    }
}

```

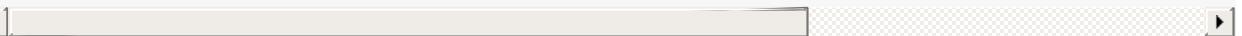
zk and broker

- ① /brokers -> topics 和 ids : 集群中所有的topics,以及所有的brokers.
- ② /brokers/ids/broker_id -> 主机的基本信息,包括主机地址和端口号

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
```

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
```

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
{"jmx_port":10055,"timestamp":"1453380999577","host":"192.168.48.153","port":10055,"version":1}
```



- ③ /brokers/topics/topic_name -> topic的每个partition,以及分配的replicas(AR)
- ④ /brokers/topics/topic_name/partitions/partition_id/state -> 这个partition的leader,ISR

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
{"version":1,"partitions":[{"2":[5,4],"1":[4,3],"0":[3,5]}]} ←
```

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
[2, 1, 0]
```

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
[state]
```

```
[zk: 192.168.47.83:2181,192.168.47.84:2181,192.168.47.86:2181(CONNECTED) 1 topics, 1000ms]
{"controller_epoch":1775,"leader":3,"version":1,"leader_epoch":145,
```



Partition Information

Partition	Leader	Replicas	In Sync Replicas
0	3	(3,5)	(3,5)
1	4	(4,3)	(3,4)
2	5	(5,4)	(4,5)

上图是kafka manager中某个topic的PartitionInfo, 集群只有3个节点,这个topic有3个partition,2个副本.

② Broker node registry

```
/brokers/ids/0 --> { "host" : "host:port", "topics" : {"topic1": [
```

每个Broker节点在自己启动的时候,会在/brokers下创建一个逻辑节点. 内容包括了Broker的主机和端口, Broker服务的所有topic, 以及分配到当前Broker的这个topic的partition列表(并不是topic的全部partition,会将所有partition分布在不同的brokers).

A consumer subscribes to event changes of the broker node registry.

当Broker挂掉的时候,在这个Broker上的所有Partition都丢失了,而Partition是给消费者服务的.

所以Broker挂掉后在做迁移的时候,会将其上的Partition转移到其他Broker上,因此消费者要消费的Partition也跟着变化.

③ Broker topic registry

```
/brokers/topics/topic1 -> {"version":1,"partitions":{ "2": [5,4], "1": [3,2] }}
```

虽然topic是在/brokers下,但是这个topic的信息是全局的.在创建topic的时候,这个topic的每个partition的编号以及replicas.

具体每个partition的Leader以及isr信息则是

在 /brokers/topics/topic_name/partitions/partition_id/state

zk and consumer

Consumer id registry: /consumers/[group_id]/ids/[consumer_id] -> topic1,...topicN

每个消费者会将它的id注册为临时znode并且将它所消费的topic设置为znode的值,当客户端(消费者)退出时,znode(consumer_id)会被删除.

A consumer subscribes to event changes of the consumer id registry within its group.

每个consumer会订阅它所在的消费组中关于consumer_id注册的更新事件. 为什么要注册呢,因为Kafka只会将一条消息发送到一个消费组中唯一的一个消费者.

如果某个消费者挂了,它要把本来发给挂的消费者的消费转给这个消费组中其他的消费者.同理,有新消费者加入消费组时,也会进行负载均衡.

Partition owner registry:

```
/consumers/[group_id]/owner/[topic]/[broker_id-partition_id] -->
consumer_node_id
```

在消费时,每个topic的partition只能被一个消费者组中的唯一的一个消费者消费.在每次重新负载的时候,这个映射策略就会重新构建.

Consumer offset tracking:

```
/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id] -->
offset_counter_value
```

每个消费者都要跟踪自己消费的每个Partition最近的offset.表示自己读取到Partition的最新位置.

由于一个Partition只能被消费组中的一个消费者消费,所以offset是以 消费组 为级别的,而不是消费者.

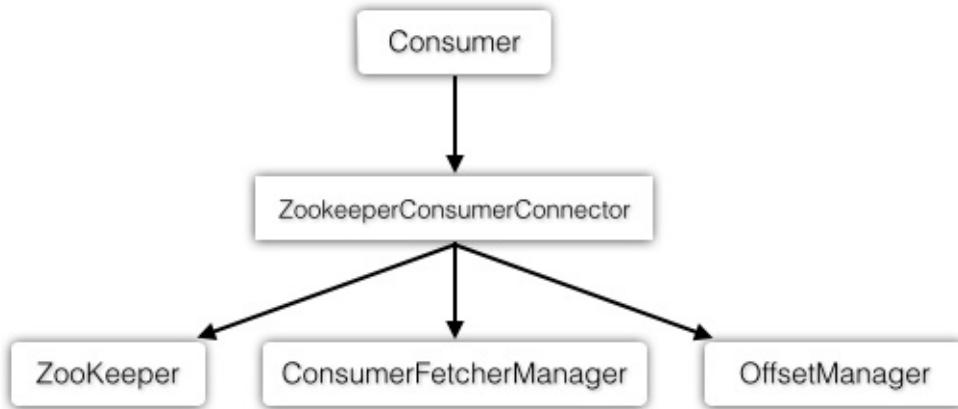
因为如果原来的消费者挂了后,应当将这个Partition交给同一个消费组中别的消费者,而此时offset是没有变化的.

一个partition可以被不同的 消费者组 中的不同消费者消费, 所以不同的消费者组必须维护他们各自对该partition消费的最新的offset

init

在创建ZookeeperConsumerConnector时,有几个初始化方法需要事先执行.

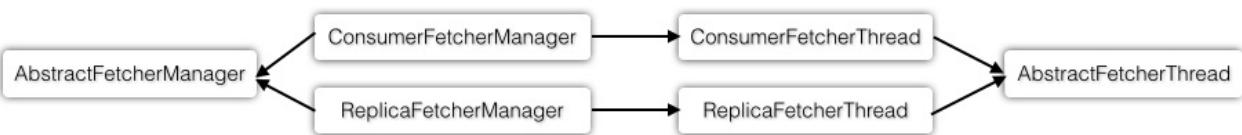
- 因为消费者要和ZK通信,所以connectZk会确保连接上ZooKeeper
- 消费者要消费数据,需要有抓取线程,所有的抓取线程交给 ConsumerFetcherManager统一管理
- 由消费者客户端自己保存offset,而消费者会消费多个topic的多个partition.
- 类似多个数据抓取线程有管理类,多个partition的offset管理类 OffsetManager是一个GroupCoordinator
- 定时提交线程会使用OffsetManager建立的通道定时提交offset到zk或者kafka.



AbstractFetcherManager

每个消费者都有自己的ConsumerFetcherManager.fetch动作不仅只有消费者有,Partition的副本也会拉取Leader的数据.

createFetcherThread抽象方法对于Consumer和Replica会分别创建ConsumerFetcherThread和ReplicaFetcherThread.



由于消费者可以消费多个topic的多个partition.每个TopicPartition组合都会有一个fetcherId.

所以fetcherThreadMap的key实际上在由(broker_id, topic_id, partition_id)组成的.针对每个source broker的每个partition都会有拉取线程,即拉取是针对partition级别拉取数据的.

```

abstract class AbstractFetcherManager(protected val name: String, c
  // map of (source broker_id, fetcher_id per source broker) => fe
  private val fetcherThreadMap = new mutable.HashMap[BrokerAndFetch
}
case class BrokerAndFetcherId(broker: BrokerEndPoint, fetcherId: In
case class BrokerAndInitialOffset(broker: BrokerEndPoint, initOffset
  
```

所以BrokerAndFetcherId可以表示Broker上某个topic的PartitionId, 而BrokerAndInitialOffset只是Broker级别的offset.

addFetcherForPartitions的参数中BrokerAndInitialOffset是和TopicAndPartition有关

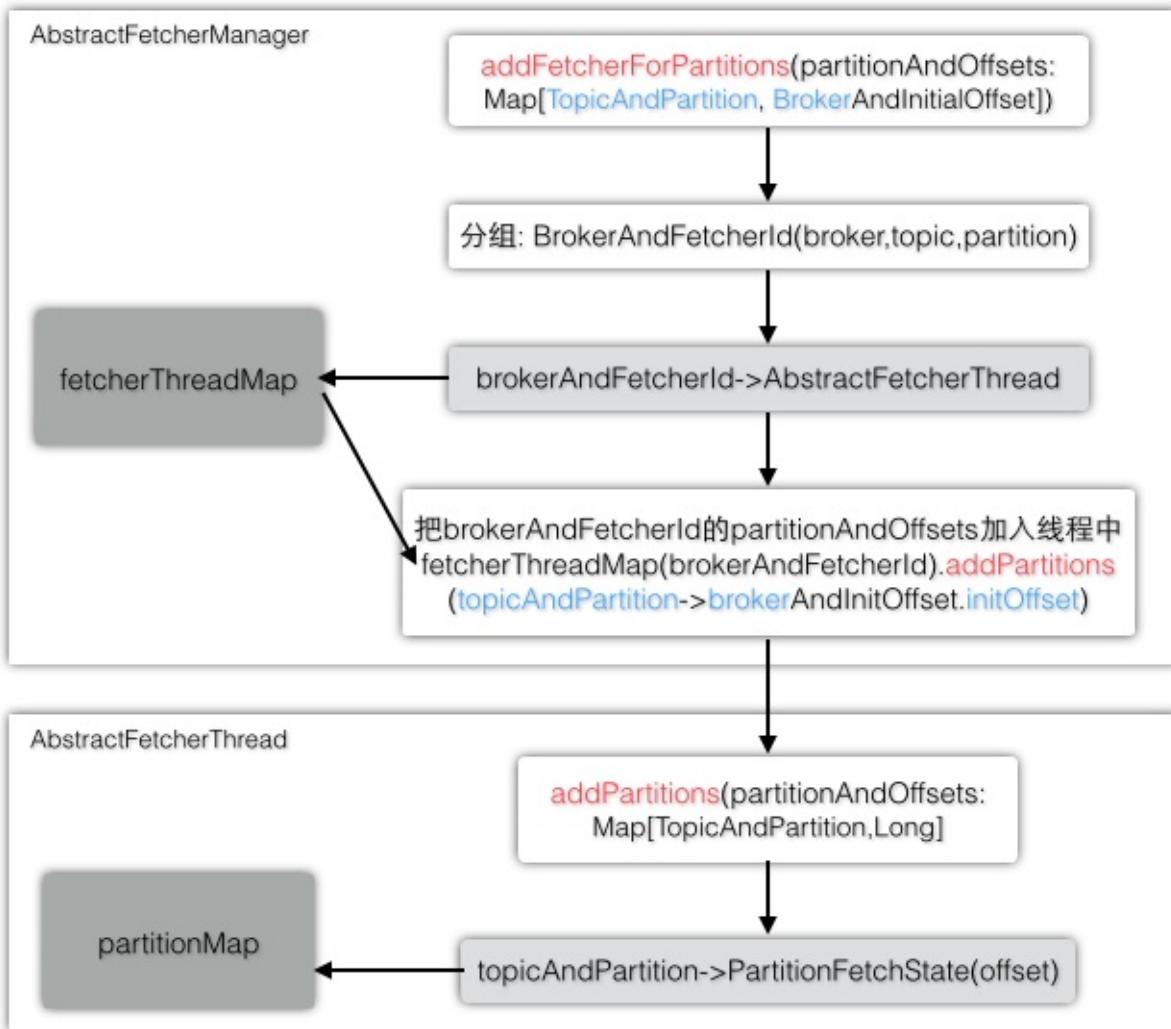
的,即Partition的offset.

为Partition添加Fetcher是为Partition创建Fetcher线程. 因为Fetcher线程是用来抓取Partition的消息.

```
// to be defined in subclass to create a specific fetcher
def createFetcherThread(fetcherId: Int, sourceBroker: BrokerEndPoint)
    : AbstractFetcherThread = ...

def addFetcherForPartitions(partitionAndOffsets: Map[TopicAndPartition,
    BrokerAndFetcherId] = partitionAndOffsets.groupBy{ case (topicAndPartition, brokerAndInitOffset) =>
    BrokerAndFetcherId(brokerAndInitOffset.broker, getFetcherId(topicAndPartition))}
    // 根据broker-topic-partition分组. 所以相同partition的只会有一个fetcher
    val partitionsPerFetcher = partitionsPerFetcherId.brokers.map{ broker =>
        BrokerAndFetcherId(broker, getFetcherId(partitionAndOffsets))}.groupByKey{ _, values =>
        BrokerAndFetcherId(values.head.broker, getFetcherId(values.head))}
        // 分组之后的value仍然不变,还是partitionAndOffsets,但是相同的partition
        for ((brokerAndFetcherId, partitionAndOffsets) <- partitionsPerFetcher) {
            // 在这里想要为每个fetcherId创建拉取线程的. 如果在缓存中直接返回,否则
            var fetcherThread: AbstractFetcherThread = null
            fetcherThreadMap.get(brokerAndFetcherId) match {
                case Some(f) => fetcherThread = f
                case None =>
                    fetcherThread = createFetcherThread(brokerAndFetcherId)
                    fetcherThreadMap.put(brokerAndFetcherId, fetcherThread)
                    fetcherThread.start // 启动刚刚创建的拉取线程
            }
        }
    }

    // 由于partitionAndOffsets现在已经是在同一个partition里. 取得所有partitions
    fetcherThreadMap(brokerAndFetcherId).addPartitions(partitionAndOffsets)
        case (topicAndPartition, brokerAndInitOffset) => topicAndPartition
    })
}
}
```



AbstractFetcherThread addPartitions

Consumer和Replica的FetcherManager都会负责将自己要抓取的partitionAndOffsets传给对应的Fetcher线程.

```

AbstractFetcherManager.addFetcherForPartitions(Map<TopicAndPartition, BrokerAndInitialOffset> partitions)
| -- LeaderFinderThread in ConsumerFetcherManager.dowork() (kafka.consumer.Consumer)
| -- ReplicaManager.makeFollowers(int, int, Map<Partition, Partition> partitions)
    
```

抓取线程也是用partitionMap缓存来保存每个TopicAndPartition的抓取状态.即 管理者负责线程 相关,而 线程负责状态 相关.

Partition的状态就是offset信息.但是拉取状态并不是实时更新的,PartitionFetchState还包括了isActive表示是否延迟.

对一个Partition延迟,判断isActive状态后,用延迟时间封装到DelayedItem.一般出错的拉取会被延迟back-off毫秒.

```

// Abstract class for fetching data from multiple partitions from 1
abstract class AbstractFetcherThread(name: String, clientId: String,
  private val partitionMap = new mutable.HashMap[TopicAndPartition, Long]
    partitionMapLock = new ReentrantLock()
    partitionMapCond = new Condition(partitionMapLock)
    partitionMapLock.lock()

  def addPartitions(partitionAndOffsets: Map[TopicAndPartition, Long])
    partitionMapLock.lockInterruptibly()
    try {
      for ((topicAndPartition, offset) <- partitionAndOffsets) {
        // If the partitionMap already has the topic/partition, then update
        if (!partitionMap.contains(topicAndPartition))
          partitionMap.put(topicAndPartition,
            if (PartitionTopicInfo.isOffsetInvalid(offset)) new PartitionFetchState(offset)
            else new PartitionFetchState(offset))
      }
      partitionMapCond.signalAll()
    } finally partitionMapLock.unlock()
  }
}

```



FetchRequest & PartitionData

拉取请求指定要拉取哪个TopicAndPartition(offset来自于PartitionFetchState), PartitionData返回要拉取的消息集.

```

type REQ <: FetchRequest //拉取请求的子类
type PD <: PartitionData //Partition数据, 即拉取结果

trait FetchRequest { //定义了拉取接口
  def isEmpty: Boolean
  def offset(topicAndPartition: TopicAndPartition): Long
}

trait PartitionData {
  def errorCode: Short
  def exception: Option[Throwable]
  def toByteBufferMessageSet: ByteBufferMessageSet
  def highWatermark: Long
}

```

FetchRequest和PartitionData也有Consumer和Replica之分.

ConsumerFetcherThread中的方法交给了underlying(类似于装饰模式).

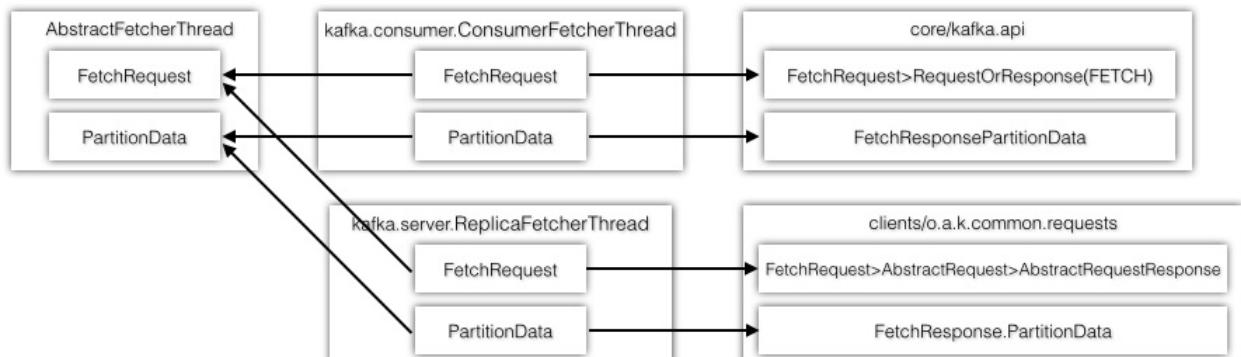
```
object ConsumerFetcherThread {
    class FetchRequest(val underlying: kafka.api.FetchRequest) extends RequestOrResponse {
        def isEmpty: Boolean = underlying.requestInfo.isEmpty
        def offset(topicAndPartition: TopicAndPartition): Long = underlying.offset(topicAndPartition)
    }
    class PartitionData(val underlying: FetchResponsePartitionData) extends ResponseOrData {
        def errorCode: Short = underlying.error
        def toByteBufferMessageSet: ByteBufferMessageSet = underlying.messageSet
        def highWatermark: Long = underlying.hw
        def exception: Option[Throwable] = if (errorCode == ErrorMapping.NoError) None else Some(new KafkaException(errorCode))
    }
}
```

来自于kafka.api的FetchRequest才是真正面向KafkaApis的请求.PartitionFetchInfo除了offset还有fetchSize.

RequestOrResponse是作为KafkaApis中数据传递的介质接口. 参数requestId表示了请求的类型(PRODUCE, FETCH等)

```
case class PartitionFetchInfo(offset: Long, fetchSize: Int)

case class FetchRequest(versionId: Short = FetchRequest.CurrentVersion,
                       clientId: String = ConsumerConfig.DefaultClientId,
                       maxWait: Int = FetchRequest.DefaultMaxWait,
                       requestInfo: Map[TopicAndPartition, PartitionFetchInfo] = Map.empty,
                       extends RequestOrResponse(Some(ApiKeys.FETCH.id)))
```



ConsumerFetcherThread.buildFetchRequest

AbstractFetcherThread的doWork会抽象出
buildFetchRequest, ConsumerFetcherThread会使用FetchRequestBuilder
build出来的是和kafka.api.FetchRequestBuilder相同文件下的
kafka.api.FetchRequest, 作为underlying.

注意其中Partition的offset最开始源自
于 AbstractFetcherManager .addFetcherForPartitions的BrokerAndInitialOffset
然后获取brokerAndInitOffset.initOffset作
为 AbstractFetcherThread .addPartitions方法参数Map的value,
并转化为 PartitionFetchState 加入到partitionMap中, 在buildFetchRequest又
转化为了 PartitionFetchInfo .

注意: 上面只是一种来源, 拉取线程在拉取数据之后, 会 更新这批数据最后一条消息的下
一个offset 作为 partitionMap 中 Partition 的
最新 PartitionFetchState, 所以下一次调用 buildFetchRequest 构建新的 FetchRequest
时, PartitionFetchInfo 的 offset 也是最新的.

```

class ConsumerFetcherThread(...){
    private val fetchRequestBuilder = new FetchRequestBuilder().
        clientId(clientId).replicaId(Request.OrdinaryConsumerId).maxWaitTime(
            minBytes(config.fetchMinBytes).requestVersion(kafka.api.FetchRe
        )

    // partitionMap来自于AbstractFetcherThread.addPartitions或者delayP
    protected def buildFetchRequest(partitionMap: collection.Map[TopicAndP
        partitionMap.foreach { case ((topicAndPartition, partitionFetchState)
            if (partitionFetchState.isActive)
                fetchRequestBuilder.addFetch(topicAndPartition.topic, topicAndP
        }
        new FetchRequest(fetchRequestBuilder.build()) //构造器模式,在最尾
    }
}

class FetchRequestBuilder() {
    private val requestMap = new collection.mutable.HashMap[TopicAndPartitio
        def addFetch(topic: String, partition: Int, offset: Long, fetchSize: I
            requestMap.put(TopicAndPartition(topic, partition), PartitionFetchS
            this
        }
        def build() = {
            val fetchRequest = FetchRequest(versionId, correlationId.getAndIncr
            requestMap.clear()
            fetchRequest
        }
    }
}

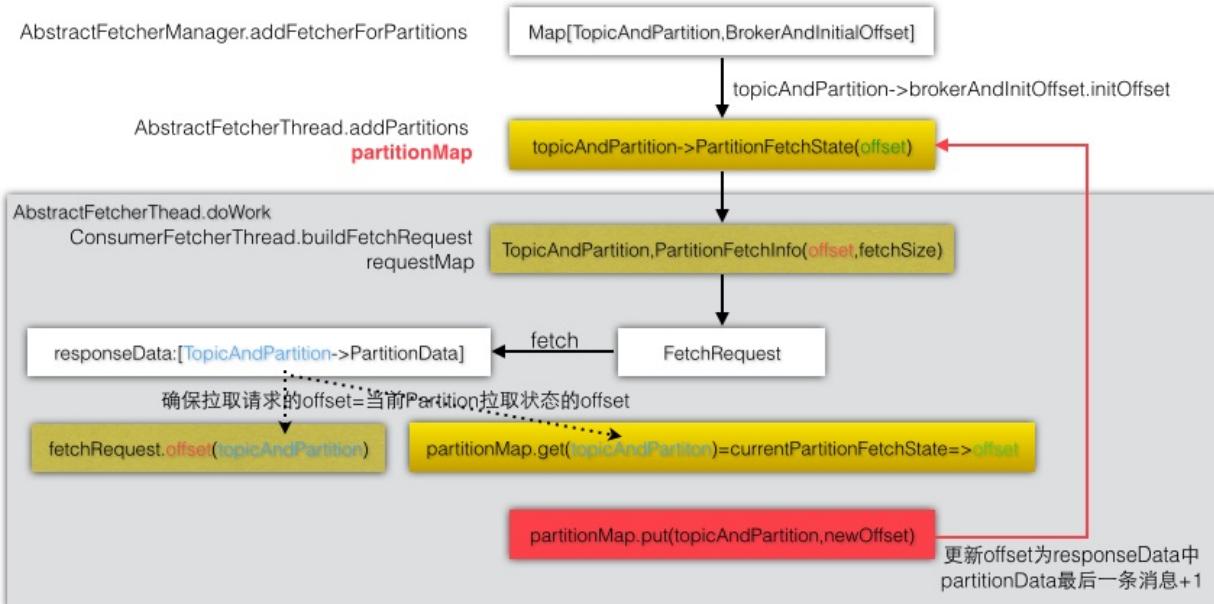
```

AbstractFetcherThread doWork

AbstractFetcherThread定义了多个回调方法,它的doWork方法会构建FetchRequest,然后处理拉取请求.

因为拉取分为Consumer和Replica,所以将具体的拉取动作要留给子类自己实现.

注意:下面的partitionMap在addPartitions中被添加.在doWork拉取到数据后被更新offset,表示最新拉取的位置



```

abstract class AbstractFetcherThread(..){
    private val partitionMap = new mutable.HashMap[TopicAndPartition,
        Map[BrokerAndInitialOffset, PartitionFetchState]]()

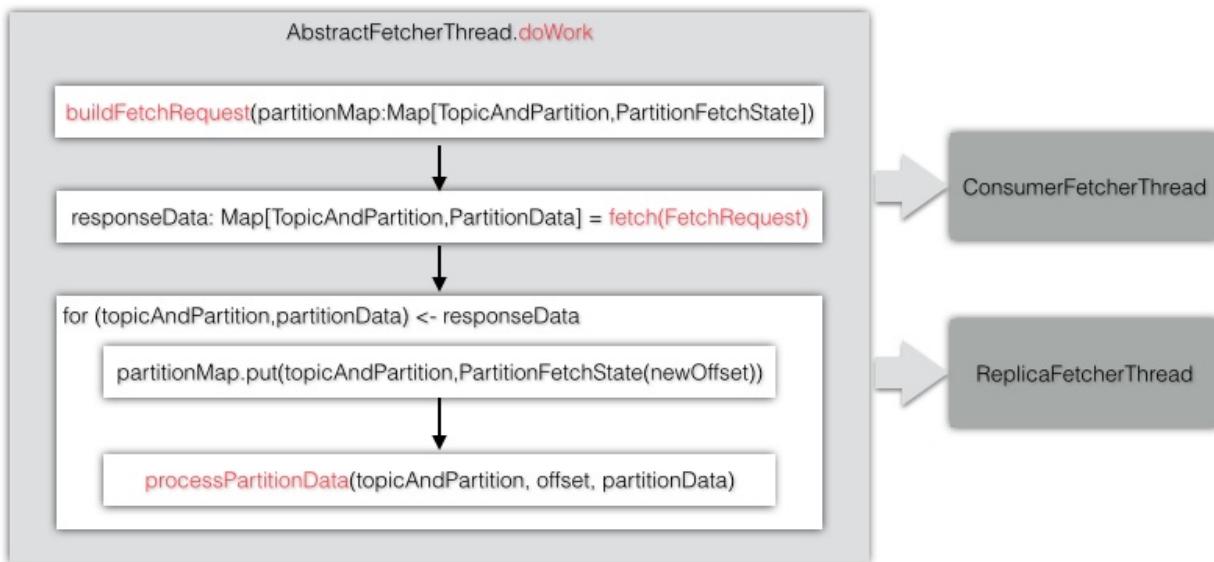
    // ① 根据partitionMap构建FetchRequest请求
    protected def buildFetchRequest(partitionMap: Map[TopicAndPartition, Map[BrokerAndInitialOffset, PartitionFetchState]]): FetchRequest = {
        // ② 根据抓取请求向Broker拉取消息
        protected def fetch(fetchRequest: REQ): Map[TopicAndPartition, PartitionFetchInfo] = {
            // ③ process fetched data 处理抓取到的数据
            def processPartitionData(topicAndPartition: TopicAndPartition, fetchData: PartitionFetchData): PartitionFetchInfo = {
                // ④ handle a partition whose offset is out of range and return null
                def handleOffsetOutOfRange(topicAndPartition: TopicAndPartition): Option[PartitionFetchInfo] = {
                    // ⑤ deal with partitions with errors, potentially due to leader
                    def handlePartitionsWithErrors(partitions: Iterable[TopicAndPartition]): Map[TopicAndPartition, PartitionFetchInfo] = {
                        // 拉取线程工作, doWork是被循环调用的, 所以一旦partionMap发生了变化(比如从leader变成了follower)
                        override def doWork() {
                            val fetchRequest = inLock(partitionMapLock) {
                                val fetchRequest = buildFetchRequest(partitionMap)
                                // 如果没有拉取请求, 则延迟back-off毫秒后继续发送请求
                                if (fetchRequest.isEmpty) partitionMapCond.await(fetchBackOffMs)
                                fetchRequest
                            }
                            if (!fetchRequest.isEmpty) processFetchRequest(fetchRequest)
                        }
                        private def processFetchRequest(fetchRequest: REQ) {
                            // ...
                        }
                    }
                }
            }
        }
    }
}
  
```

```

val partitionsWithError = new mutable.HashSet[TopicAndPartition]
var responseData: Map[TopicAndPartition, PD] = Map.empty
responseData = fetch(fetchRequest)
responseData.foreach { case (topicAndPartition, partitionData)
    // 响应结果:TopicAndPartition->PartitionData,根据TopicAndPartition
    partitionMap.get(topicAndPartition).foreach(currentPartitionF
    // we append to the log if the current offset is defined ar
    // fetchRequest是由partitionMap通过buildFetchRequest构建出来的
    if (fetchRequest.offset(topicAndPartition) == currentPartitio
    Errors.forCode(partitionData.errorCode) match {
        case Errors.NONE =>
            // responseData的PartitionData,包含了拉取的消息内容
            val messages = partitionData.toByteBufferMessageSet
            // 最后一条消息的offset+1,为新的offset,即下一次要拉取的offset
            val newOffset = messages.shallowIterator.toSeq.lastOffset
            case Some(m: MessageAndOffset) => m.nextOffset
            case None => currentPartitionFetchState.offset
        }
        // 更新partitionMap中的Partition拉取状态,这样下次请求时
        partitionMap.put(topicAndPartition, new PartitionFetchState(
            processPartitionData(topicAndPartition, currentPartitionF
        case Errors.OFFSET_OUT_OF_RANGE => partitionMap.put(topicAn
        case _ => if (isRunning.get) partitionsWithError += topic
    }
}
}

if (partitionsWithError.nonEmpty) {
    handlePartitionsWithErrors(partitionsWithError)
}
}
}

```



基本上我们把关于Fetcher的Manager和Thread的抽象类都分析完了,现在看看Consumer是如何Fetch消息的.

createMessageStreams

由ConsumerConnector创建消息流,需要指定解码器,因为要将日志反序列化(生产者写消息时对消息序列化到日志文件).

`consume`并不真正的消费数据,只是初始化存放数据的queue.真正消费数据的是对该queue进行shallow iterator.

在kafka的运行过程中,会有其他的线程将数据放入partition对应的queue中. 而queue是用于KafkaStream的.

一旦数据添加到queue后,KafkaStream的阻塞队列就有数据了,消费者就可以从队列中消费消息.

```

def createMessageStreams[K,V](topicCountMap: Map[String,Int], keyDecoder, valueDecoder)
    consume(topicCountMap, keyDecoder, valueDecoder)
}
def consume[K, V](topicCountMap: scala.collection.Map[String,Int]
    val topicCount = TopicCount.constructTopicCount(consumerIdString)
    val topicThreadIds = topicCount.getConsumerThreadIdsPerTopic

    // make a list of (queue,stream) pairs, one pair for each thread
    val queuesAndStreams = topicThreadIds.values.map(threadIdSet =>
        threadIdSet.map(_ => {
            val queue = new LinkedBlockingQueue[FetchedDataChunk](config.consumerQueueSize)
            val stream = new KafkaStream[K,V](queue, config.consumerTimeoutMs)
            (queue, stream)
        })
    ).flatten.toList //threadIdSet是个集合,外层的topicThreadIds.values是个map

    val dirs = new ZKGroupDirs(config.groupId) //ZKGroupDirs
    registerConsumerInZK(dirs, consumerIdString, topicCount) //registerConsumerInZK
    reinitializeConsumer(topicCount, queuesAndStreams) //reinitializeConsumer

    // 返回KafkaStream, 每个Topic都对应了多个KafkaStream. 数量和topicCount一致
    loadBalancerListener.kafkaMessageAndMetadataStreams.asInstanceOf[Map[TopicPartition, KafkaStream[K,V]]]
}

```

consumerIdString 会返回当前Consumer在哪个ConsumerGroup的编号.每个 consumer在消费组中的编号都是唯一的.

一个消费者,对一个topic可以使用多个线程一起消费(一个进程可以有多个线程).当然一个消费者也可以消费多个topic.

```

def makeConsumerThreadIdsPerTopic(consumerIdString: String, topicCountMap: Map[String, Int]): Map[String, Set[ConsumerThreadId]] = {
    val consumerThreadIdsPerTopicMap = new mutable.HashMap[String, Set[ConsumerThreadId]]()
    for ((topic, nConsumers) <- topicCountMap) {
        val consumerSet = new mutable.HashSet[ConsumerThreadId]()
        for (i <- 0 until nConsumers)
            consumerSet += ConsumerThreadId(consumerIdString, i)
        consumerThreadIdsPerTopicMap.put(topic, consumerSet)
    }
    consumerThreadIdsPerTopicMap
}

```

假设消费者C1声明了topic1:2, topic2:3.

topicThreadIds=consumerThreadIdsPerTopicMap.

topicThreadIds.values = [(C1_1,C1_2), (C1_1,C1_2,C1_3)]一共有5个线程,queuesAndStreams也有5个元素.

```

consumerThreadIdsPerTopicMap = {
    topic1: [C1_1, C1_2],
    topic2: [C1_1, C1_2, C1_3]
}
topicThreadIds.values = [
    [C1_1, C1_2],
    [C1_1, C1_2, C1_3]
]
threadIdSet循环[C1_1, C1_2]时, 生成两个queue->stream pair.
threadIdSet循环[C1_1, C1_2, C1_3]时, 生成三个queue->stream pair.
queuesAndStreams = [
    (LinkedBlockingQueue_1,KafkaStream_1),           //topic1:C1_1
    (LinkedBlockingQueue_2,KafkaStream_2),           //topic1:C1_2
    (LinkedBlockingQueue_3,KafkaStream_3),           //topic2:C1_1
    (LinkedBlockingQueue_4,KafkaStream_4),           //topic2:C1_2
    (LinkedBlockingQueue_5,KafkaStream_5),           //topic2:C1_3
]

```

对于消费者而言,它只要指定要消费的topic和线程数量就可以了,其他具体这个topic分成多少个partition,
以及topic-partition是分布是哪个broker上,对于客户端而言都是透明的.

客户端关注的是我的每个线程都对应了一个队列,每个队列都是一个消息流就可以了.

在Producer以及前面分析的Fetcher,都是以Broker-Topic-Partition为级别的.

AbstractFetcherManager的fetcherThreadMap就是以brokerAndFetcherId来创建拉取线程的.

而消费者是通过拉取线程才有数据可以消费的,所以客户端的每个线程实际上也是针对Partition级别的.

registerConsumerInZK

消费者需要向ZK注册一个临时节点,节点内容为订阅的topic.

```
private def registerConsumerInZK(dirs: ZKGroupDirs, consumerIdString: String): Unit = {
    val consumerRegistrationInfo = Json.encode(Map("version" -> 1,
    "consumerId" -> consumerIdString))
    val zkWatchedEphemeral = new ZKCheckedEphemeral(dirs.consumerRegistrationPath)
    zkWatchedEphemeral.create()
}
```

问题:什么时候这个节点会被删除掉呢? Consumer进程挂掉时,或者Session失效时删除临时节点. 重连时会重新创建.

reinitializeConsumer listener

当前Consumer在ZK注册之后,需要重新初始化Consumer. 对于全新的消费者,注册多个监听器,在zk的对应节点的注册事件发生时,会回调监听器的方法.

- 将topic对应的消费者线程id及对应的LinkedBlockingQueue放入topicThreadIdAndQueues中,LinkedBlockingQueue是真正存放数据的queue
- ① 注册 sessionExpirationListener ,监听状态变化事件. 在session失效重新创建session时调用
- ② 向 /consumers/group/ids 注册Child变更事件的 loadBalancerListener ,当消费组下的消费者发生变化时调用
- ③ 向 /brokers/topics/topic 注册Data变更事件的 topicPartitionChangeListener ,在topic数据发生变化时调用
- 显式调用 loadBalancerListener.syncedRebalance() ,会调用reblance方法进行consumer的初始化工作

```

private def reinitializeConsumer[K,V](topicCount: TopicCount, queueSize: Int, zkClient: ZkClient, config: ConsumerConfig, brokerTopicsPath: String) {
    val dirs = new ZKGroupDirs(config.groupId)
    // ② listener to consumer and partition changes
    if (loadBalancerListener == null) {
        val topicStreamsMap = new mutable.HashMap[String, List[KafkaStream[_, _]]]
        loadBalancerListener = new ZKRebalancerListener(config.groupId, topicStreamsMap)
    }
    // ① create listener for session expired event if not exist yet
    if (sessionExpirationListener == null) sessionExpirationListener = new ZKSessionExpireListener()
    // ③ create listener for topic partition change event if not exist yet
    if (topicPartitionChangeListener == null) topicPartitionChangeListener = new ZKTopicPartitionChangeListener()

    // listener to consumer and partition changes
    zkUtils.zkClient.subscribeStateChanges(sessionExpirationListener)
    zkUtils.zkClient.subscribeChildChanges(dirs.consumerRegistryDir)
    // register on broker partition path changes.
    topicStreamsMap.foreach { topicAndStreams =>
        zkUtils.zkClient.subscribeDataChanges(BrokerTopicsPath + "/" + topicAndStreams._1)
    }

    // explicitly trigger load balancing for this consumer
    loadBalancerListener.syncedRebalance()
}

```

ZKRebalancerListener传入ZKSessionExpireListener和ZKTopicPartitionChangeListener.它们都会使用ZKRebalancerListener完成自己的工作.

ZKSessionExpireListener

当Session失效时,新的会话建立时,立即进行rebalance操作.

```
class ZKSessionExpireListener(val dirs: ZKGroupDirs, val consumerIdString: String) extends IZkStateListener {
    def handleNewSession() {
        loadBalancerListener.resetState()
        registerConsumerInZK(dirs, consumerIdString, topicCount)
        loadBalancerListener.syncedRebalance()
    }
}
```

ZKTopicPartitionChangeListener

当topic的数据变化时,通过触发的方式启动rebalance操作.

```
class ZKTopicPartitionChangeListener(val loadBalancerListener: ZkLoadBalancer) extends IZkDataListener {
    def handleDataChange(dataPath : String, data: Object) {
        loadBalancerListener.rebalanceEventTriggered()
    }
}
```

ZKRebalancerListener watcher

```

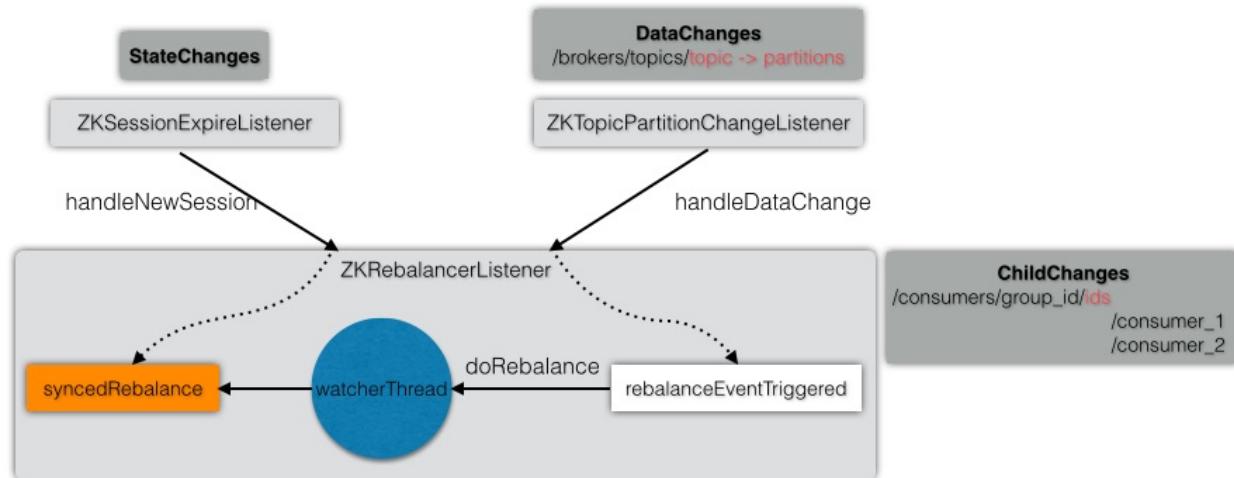
class ZKRebalancerListener(val group: String, val consumerIdString: String,
                           val kafkaMessageAndMetadataStreams: Map[TopicPartition, KafkaMessageAndMetadata])
    extends IZkChildListener {
  private var isWatcherTriggered = false
  private val lock = new ReentrantLock
  private val cond = lock.newCondition()

  private val watcherExecutorThread = new Thread(consumerIdString + " rebalance watcher")
  override def run() {
    var doRebalance = false
    while (!isShuttingDown.get) {
      lock.lock()
      try {
        // 如果isWatcherTriggered=false，则不会触发syncedRebalance
        if (!isWatcherTriggered)
          cond.await(1000, TimeUnit.MILLISECONDS) // wake up
      } finally {
        // 不管isWatcherTriggered值是多少，在每次循环时，都会执行。如果doRebalance = isWatcherTriggered
        // 重新设置isWatcherTriggered=false，因为其他线程触发一次
        isWatcherTriggered = false
        lock.unlock()
      }
      if (doRebalance) syncedRebalance // 只有每次rebalance时才会调用
    }
  }
}

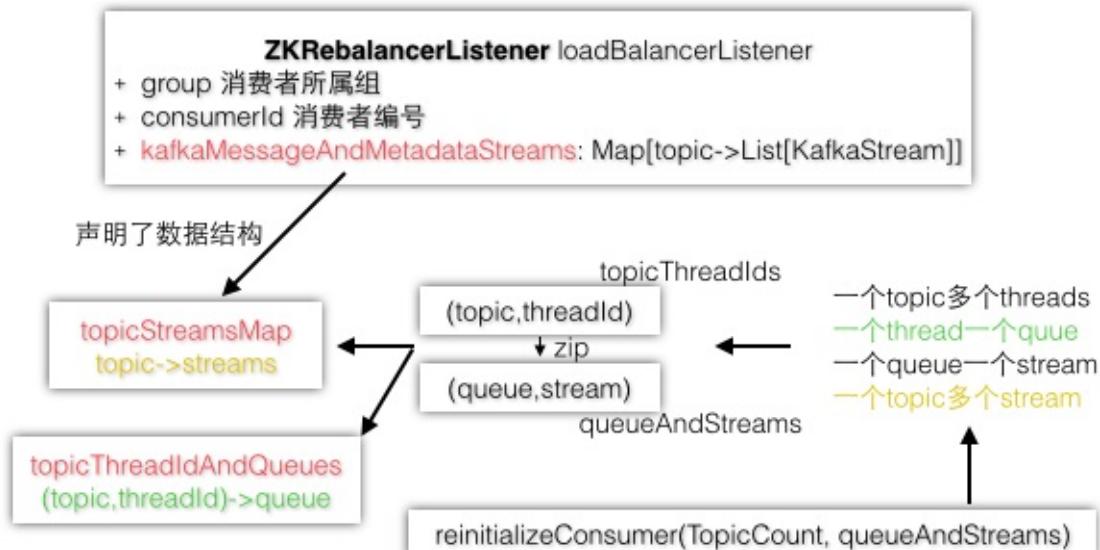
watcherExecutorThread.start()

// 触发rebalance开始进行，修改isWatcherTriggered标志位，触发cond条件变量
def rebalanceEventTriggered() {
  inLock(lock) {
    isWatcherTriggered = true
    cond.signalAll()
  }
}

```



`reinitializeConsumer`的topicStreamsMap是从(topic,thread)->(queue,stream)根据topic获取stream得来的.



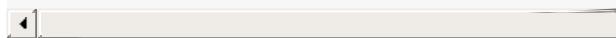
```

    val topicStreamsMap = loadBalancerListener.kafkaMessageAndMetadata
    // map of {topic -> Set(thread-1, thread-2, ...)}
    val consumerThreadIdsPerTopic: Map[String, Set[ConsumerThreadId]] =
    // list of (Queue, KafkaStreams): (queue1, stream1), (queue2, stream2)
    val allQueuesAndStreams = queuesAndStreams

    // (topic,thread-1), (topic,thread-2), ...
    val topicThreadIds = consumerThreadIdsPerTopic.map {
      case(topic, threadIds) => threadIds.map((topic, _)) //一个topic可能有多个线程
    }.flatten
    // (topic,thread-1), (queue1, stream1)
    // (topic,thread-2), (queue2, stream2)
    val threadQueueStreamPairs = topicThreadIds.zip(allQueuesAndStreams)

    threadQueueStreamPairs.foreach(e => {
      val topicThreadId = e._1 // (topic,threadId)
      val q = e._2._1           // Queue
      topicThreadIdAndQueues.put(topicThreadId, q)
    })

    val groupedByTopic = threadQueueStreamPairs.groupBy(_._1._1)
    // 根据topic分组之后, groupedByTopic的每个元素的_1为topic,_2为属于这个topic的所有队列和流
    groupedByTopic.foreach(e => {
      val topic = e._1
      // e._2是一个List[((topic,thread),(queue,stream))],下面收集这个1
      val streams = e._2.map(_._2._2).toList
      topicStreamsMap += (topic -> streams)
    })
  
```



ZKRebalancerListener rebalance

- ① 关闭数据抓取线程，获取之前为topic设置的存放数据的queue并清空该queue
- ② 释放partition的ownership,删除partition和consumer的对应关系
- ③ 为各个partition重新分配threadid
- 获取partition最新的offset并重新初始化新的PartitionTopicInfo(queue存放数据,两个offset为partition最新的offset)

- ④ 重新将partition对应的新的consumer信息写入zookeeper
- ⑤ 重新创建partition的fetcher线程



```

private def rebalance(cluster: Cluster): Boolean = {
  val myTopicThreadIdsMap = TopicCount.constructTopicCount(gro...
  val brokers = zkUtils.getAllBrokersInCluster()
  if (brokers.size == 0) {
    zkUtils.zkClient.subscribeChildChanges(BrokerIdsPath, load...
    true
  } else {
    // ① 停止fetcher线程防止数据重复. 如果当前调整失败了, 被释放的partit...
    closeFetchers(cluster, kafkaMessageAndMetadataStreams, myTo...
    // ② 释放topicRegistry中topic-partition的owner
    releasePartitionOwnership(topicRegistry)
    // ③ 为partition重新分配消费者....
    // ④ 为partition添加consumer owner
    if(reflectPartitionOwnershipDecision(partitionAssignment))
      allTopicsOwnedPartitionsCount = partitionAssignment.size
      topicRegistry = currentTopicRegistry
      // ⑤ 创建拉取线程
      updateFetcher(cluster)
      true
    }
  }
}
  
```

PartitionOwnership

topicRegistry的数据结构是: `topic -> (partition -> PartitionTopicInfo)` , 表示现有的topic注册信息.

当partition被consumer所拥有后, 会在zk中创

建 `/consumers/[group_id]/owner/[topic]/[partition_id]` -->

consumer_node_id

释放所有partition的ownership, 数据来源于topicRegistry的topic-partition(消费者所属的group_id也是确定的).

所以deletePartitionOwnershipFromZK会删除

除 /consumers/[group_id]/owner/[topic]/[partition_id] 节点.

这样partition没有了owner,说明这个partition不会被consumer消费了,也就相当于consumer释放了partition.

```

private def releasePartitionOwnership(localTopicRegistry: Pool[TopicInfo])
    for ((topic, infos) <- localTopicRegistry) {
        for(partition <- infos.keys) {
            deletePartitionOwnershipFromZK(topic, partition)
        }
        localTopicRegistry.remove(topic)
    }
    allTopicsOwnedPartitionsCount = 0
}
private def deletePartitionOwnershipFromZK(topic: String, partition: Int)
    val topicDirs = new ZKGroupTopicDirs(group, topic)
    val znode = topicDirs.consumerOwnerDir + "/" + partition
    zkUtils.deletePath(znode)
}

```

重建ownership. 参数partitionAssignment会指定partition(TopicAndPartition)要分配给哪个consumer(ConsumerThreadId)消费的.

```

private def reflectPartitionOwnershipDecision(partitionAssignment: Map[TopicPartition, PartitionOwner])
  var successfullyOwnedPartitions : List[(String, Int)] = Nil
  val partitionOwnershipSuccessful = partitionAssignment.map {
    val topic = partitionOwner._1.topic
    val partition = partitionOwner._1.partition
    val consumerThreadId = partitionOwner._2
    zkUtils.createEphemeralPathExpectConflict(partitionOwnerPath)
  }
  // 成功创建的节点,加入到列表中
  successfullyOwnedPartitions ::= (topic, partition)
  true
}
// 判断上面的创建节点操作(为consumer分配partition)是否有错误,一旦有-
val hasPartitionOwnershipFailed = partitionOwnershipSuccessful.size > 0
if(hasPartitionOwnershipFailed > 0) {
  successfullyOwnedPartitions.foreach(topicAndPartition => doNotDelete)
  false
} else true
}

```

关于consumer的注册节点出现的地方有:开始时的registerConsumerInZK,以及这里的先释放再注册.

AssignmentContext

AssignmentContext是PartitionAssignor要为某个消费者分配的上下文.因为消费者只订阅了特定的topic,所以首先要选出topic.

每个topic都是有partitions map,表示这个topic有哪些partition,以及对应的replicas.进而得到partitionsForTopic.

consumersForTopic:要在当前消费者所在的消费组中,找到所有订阅了这个topic的消费者.以topic为粒度,统计所有的线程数.

```

class AssignmentContext(group: String, val consumerId: String, ex:
    // 当前消费者的消费topic和消费线程, 因为指定了consumerId, 所以是针对当前co
    val myTopicThreadIds: collection.Map[String, collection.Set[Consu
        val myTopicCount = TopicCount.constructTopicCount(group, consum
        myTopicCount.getConsumerThreadIdsPerTopic
    }
    // 属于某个topic的所有partitions. 当然topic是当前消费者订阅的范围内, 其他
    val partitionsForTopic: collection.Map[String, Seq[Int]] = zkUtil
        // 在当前消费组内, 属于某个topic的所有consumers
        val consumersForTopic: collection.Map[String, List[ConsumerThread
        val consumers: Seq[String] = zkUtils.getConsumersInGroup(group).s
    }

```

`/brokers/topics/topic_name` 记录的是topic的partition分配情况. 获取partition信息, 读取的是partitions字段(partitionMap).

```

def getPartitionsForTopics(topics: Seq[String]): mutable.Map[String, Seq[Int]] = {
    getPartitionAssignmentForTopics(topics).map { topicAndPartitionMap =>
        val topic = topicAndPartitionMap._1
        val partitionMap = topicAndPartitionMap._2
        (topic -> partitionMap.keys.toSeq.sortWith((s, t) => s < t))
    }
}

```

每个消费者都可以指定消费的topic和线程数, 对于同一个消费组中多个消费者可以指定消费同一个topic或不同topic.

获取topic的所有consumers时, 统计所有消费这个topic的消费者线程(原先以消费者, 现在转换为以topic).

注意: 这里是取出当前消费组下所有消费者的所有topic, 并没有过滤出属于当前消费者感兴趣的topics.

```

def getConsumersPerTopic(group: String, excludeInternalTopics: Boolean): Map[String, List[ConsumerThread]] = {
    val dirs = new ZKGroupDirs(group)
    // 当前消费组下所有的consumers
    val consumers = getChildrenParentMayNotExist(dirs.consumerRegistrationPath)
    val consumersPerTopicMap = new mutable.HashMap[String, List[ConsumerThread]]()
    for (consumer <- consumers) {
        // 每个consumer都指定了topic和thread-count
        val topicCount = TopicCount.constructTopicCount(group, consumer)
        for ((topic, consumerThreadIdSet) <- topicCount.getConsumerThreads) {
            // 现在是在同一个consumer里, 对同一个topic, 有多个thread-id
            for (consumerThreadId <- consumerThreadIdSet) {
                // 最后按照topic分, 而不是按照consumer分, 比如多个consumer都消费一个topic
                consumersPerTopicMap.get(topic) match {
                    case Some(curConsumers) => consumersPerTopicMap.put(topic, curConsumers :+ consumer)
                    case _ => consumersPerTopicMap.put(topic, List(consumer))
                }
            }
        }
    }
    for ((topic, consumerList) <- consumersPerTopicMap) consumersPerTopicMap.put(topic, consumerList)
    consumersPerTopicMap
}

```

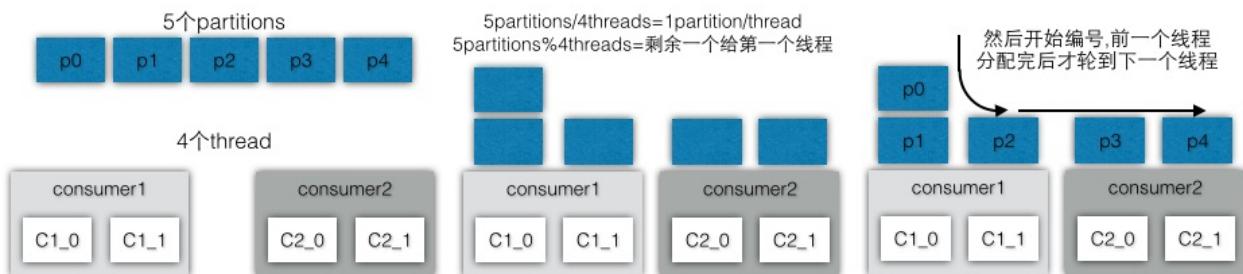
PartitionAssignor

将可用的partitions以及消费者线程排序, 将partitions处在线程数, 表示每个线程(不是消费者数量)平均可以分到几个partition.

如果除不尽, 剩余的会分给前面几个消费者线程. 比如有两个消费者, 每个都是两个线程, 一共有5个可用的partitions:(p0-p4).

每个消费者线程(一共四个线程)可以获取到至少一共partition($5/4=1$), 剩余一个($5\%4=1$)partition分给第一个线程.

最后的分配结果为: p0 -> C1-0, p1 -> C1-0, p2 -> C1-1, p3 -> C2-0, p4 -> C2-1

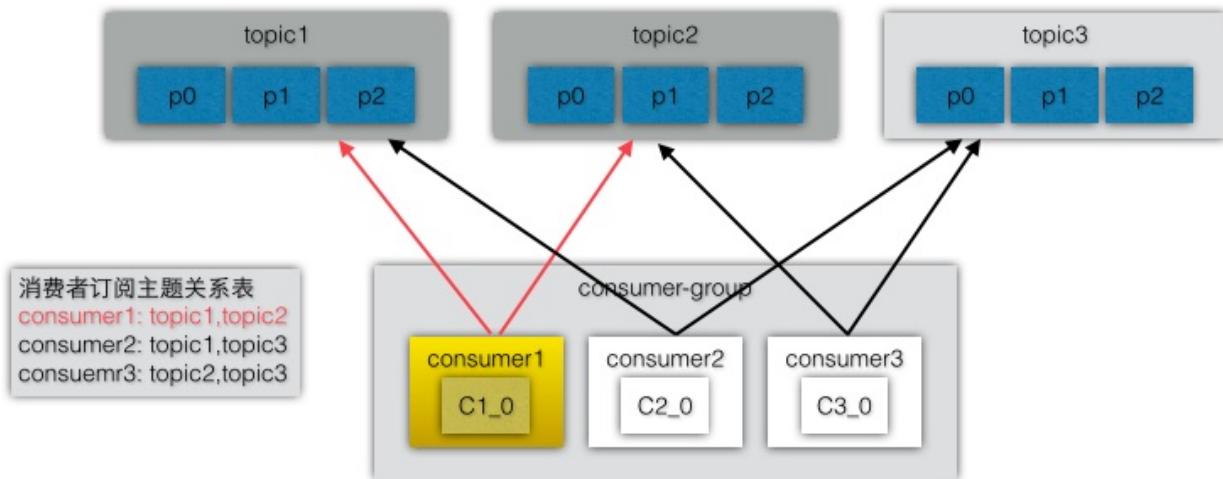


```

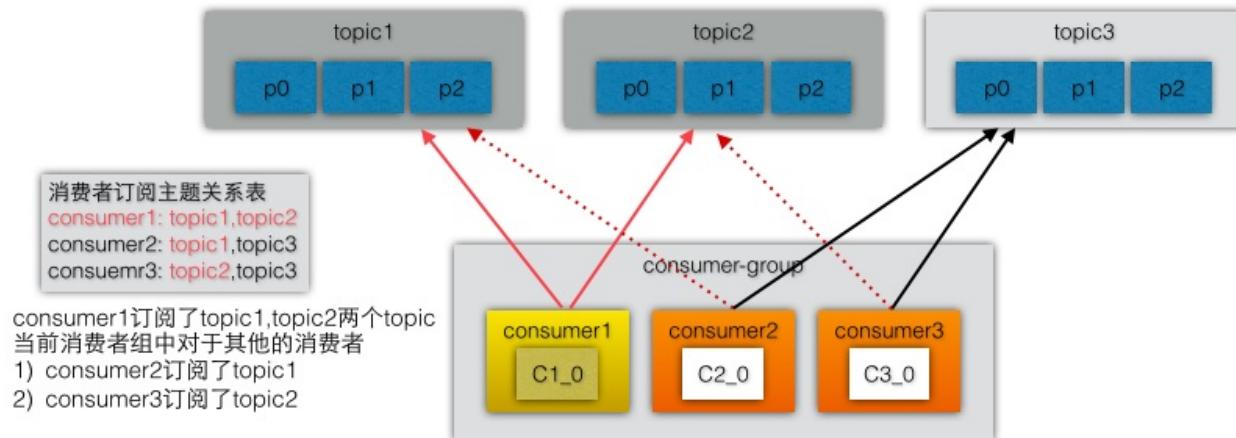
class RangeAssignor() extends PartitionAssignor with Logging {
  def assign(ctx: AssignmentContext) = {
    val valueFactory = (topic: String) => new mutable.HashMap[TopicAndPartition, ConsumerThreadId]
    // consumerThreadId -> (TopicAndPartition -> ConsumerThreadId)
    val partitionAssignment = new Pool[String, mutable.Map[TopicAndPartition, ConsumerThreadId]](valueFactory)
    for (topic <- ctx.myTopicThreadIds.keySet) {
      val curConsumers = ctx.consumersForTopic(topic)
      val curPartitions: Seq[Int] = ctx.partitionsForTopic(topic)
      val nPartsPerConsumer = curPartitions.size / curConsumers.size
      val nConsumersWithExtraPart = curPartitions.size % curConsumers.size
      for (consumerThreadId <- curConsumers) {
        val myConsumerPosition = curConsumers.indexOf(consumerThreadId)
        val startPart = nPartsPerConsumer * myConsumerPosition + myConsumerPosition
        val nParts = nPartsPerConsumer + (if (myConsumerPosition + 1 == curConsumers.size) 1 else 0)
        // Range-partition the sorted partitions to consumers for topic
        if (nParts > 0)
          for (i <- startPart until startPart + nParts) {
            val partition = curPartitions(i)
            // record the partition ownership decision 记录partition ownership
            val assignmentForConsumer = partitionAssignment.getAndCompute(partition, consumerThreadId)
            assignmentForConsumer += (TopicAndPartition(topic, partition), consumerThreadId)
          }
      }
    }
    // 前面的for循环中的consumers是和当前消费者有相同topic的, 如果消费者的topic不同
    // assign Map.empty for the consumers which are not associated with the topic
    ctx.consumers.foreach(consumerId => partitionAssignment.getAndCompute(TopicAndPartition(consumerId, 0), consumerId))
  }
}

```

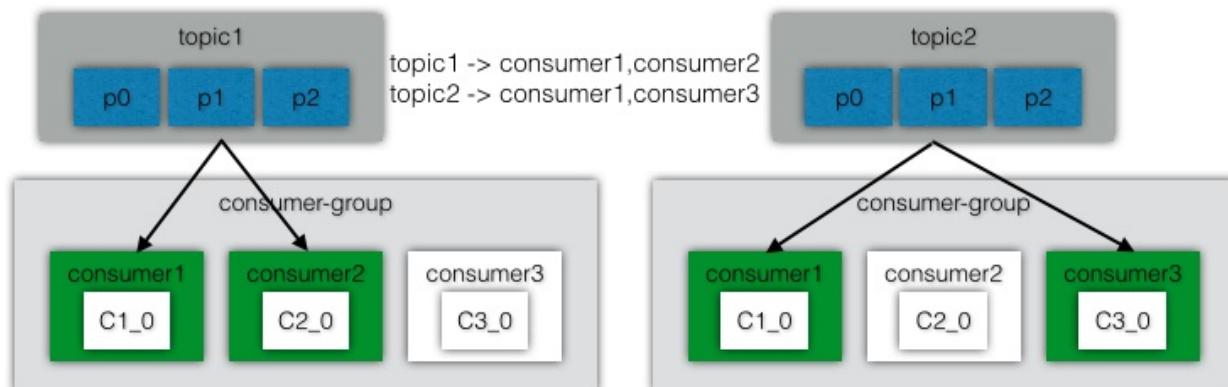
假设有如下的消费者-订阅topic的关系.



首先找到和consumer1有相同订阅主题的消费者



对于consumer1的所有topic,分配给有这个订阅关系的所有订阅者



假设还有一个consumer4,订阅的是topic4. 则这部分执行的就是最后的ctx.consumers...

PartitionAssignment -> PartitionTopicInfo

这是ZKRebalancerListener.rebalance的第三步.首先为当前消费者创建AssignmentContext上下文.

上面知道partitionAssignor.assign的返回值是所有consumer的分配结果(虽然有些consumer在本次中并没有分到partition)

partitionAssignment的结构是 consumerThreadId -> (TopicAndPartition -> ConsumerThreadId) ,

所以获取当前consumer只要传入assignmentContext.consumerId就可以得到当前消费者的PartitionAssignment.

```
// ③ 为partition重新选择consumer
val assignmentContext = new AssignmentContext(group, consumer)
val globalPartitionAssignment = partitionAssignor.assign(as
val partitionAssignment = globalPartitionAssignment.get(as
val currentTopicRegistry = new Pool[String, Pool[Int, Parti

// fetch current offsets for all topic-partitions
val topicPartitions = partitionAssignment.keySet.toSeq
val offsetFetchResponseOpt = fetchOffsets(topicPartitions)
val offsetFetchResponse = offsetFetchResponseOpt.get
topicPartitions.foreach(topicAndPartition => {
    val (topic, partition) = topicAndPartition.asTuple
    val offset = offsetFetchResponse.requestInfo(topicAndParti
    val threadId = partitionAssignment(topicAndPartition)
    addPartitionTopicInfo(currentTopicRegistry, partition,
})

// ④ ⑤ 注册到zk上，并创建新的fetcher线程
if(reflectPartitionOwnershipDecision(partitionAssignment))
    topicRegistry = currentTopicRegistry // topicRegistry
    updateFetcher(cluster)
}
```

PartitionAssignment的key是TopicAndPartition,根据所有topicAndPartitions获取这些partitions的offsets.

返回值offsetFetchResponse包含了对应的offset, 最终根据这些信息创建对应的PartitionTopicInfo.

PartitionTopicInfo包含Partition和Topic:队列用来存放数据

(topicThreadIdAndQueues在reinitializeConsumer中放入), consumedOffset和fetchedOffset都是来自于offset参数,即上面offsetFetchResponse中partition的offset.

currentTopicRegistry这个结构也很重要: topic -> (partition -> PartitionTopicInfo)

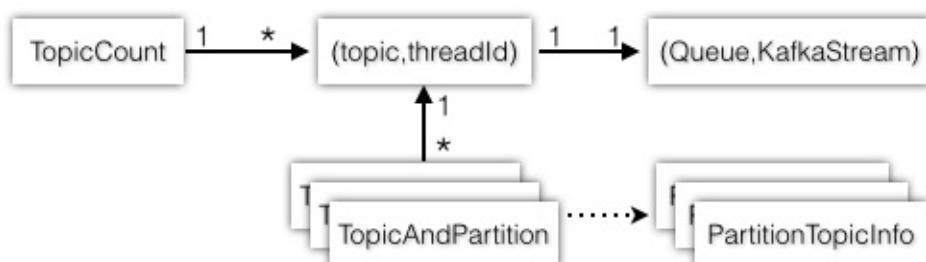
key是topic,正如名称所示是topic的注册信息(topicRegistry).又因为来自于fetchOffsets,所以表示的是最新当前的.

```
private def addPartitionTopicInfo(currentTopicRegistry: Pool[St
                                partition: Int, topic: String]
  val partTopicInfoMap = currentTopicRegistry.getAndMaybePut(t
  val queue = topicThreadIdAndQueues.get((topic, consumerThread
  val consumedOffset = new AtomicLong(offset)
  val fetchedOffset = new AtomicLong(offset)
  val partTopicInfo = new PartitionTopicInfo(topic, partition, qu
  partTopicInfoMap.put(partition, partTopicInfo)
  checkpointedZkOffsets.put(TopicAndPartition(topic, partition))
}
}
```

注意:一个threadId可以消费同一个topic的多个partition. 而一个threadId对应一个queue.

所以一个queue也就可以消费多个partition. 即对于不同的partition,可能使用同一个队列来消费.

比如消费者设置了一个线程,就只有一个队列,而partition分了两个给它,这样一个队列就要处理两个partition了.



updateFetcher & closeFetchers

这里我们看到了在ZooKeeperConsumerConnector初始化时创建的fetcher(ConsumerFetcherManager)终于派上用场了.allPartitionInfos是分配给Consumer的Partition列表(但是这里还不知道Leader的,所以在Manager中要自己寻找Leader).

```

private def updateFetcher(cluster: Cluster) {    // update part:
  var allPartitionInfos : List[PartitionTopicInfo] = Nil
  for (partitionInfos <- topicRegistry.values)    // topicRegist
    for (partition <- partitionInfos.values)        // PartitionTop
      allPartitionInfos ::= partition
  fetcher match {
    case Some(f) => f.startConnections(allPartitionInfos, clust
  }
}

```



创建Fetcher是为PartitionInfos准备开始连接,在rebalance时一开始要先closeFetchers就是关闭已经建立的连接.

relevantTopicThreadIdsMap是当前消费者的topic->threadIds,要从topicThreadIdAndQueues过滤出需要清除的queues.

DataStructure	Explain
relevantTopicThreadIdsMap	消费者注册的topic->threadIds
topicThreadIdAndQueues	消费者的(topic,threadId)->queue
messageStreams	消费者注册的topic->List[KafkaStream]消息流

关闭Fetcher时要注意: 先提交offset,然后才停止消费者. 因为在停止消费者的时候当前的数据块中还会有点残留数据.

因为这时候还没有释放partiton的ownership(即partition还归当前consumer所有),强制提交offset,

这样拥有这个partition的下一个消费者线程(rebalance后),就可以使用已经提交的offset了,确保不中断.

因为fetcher线程已经关闭了(stopConnections),这是消费者能得到的最后一个数据块,以后不会有,直到平衡结束,fetcher重新开始

topicThreadIdAndQueues 来自于 topicThreadIds, 所以它的 topic 应该都在 relevantTopicThreadIdsMap 的 topics 中.

为什么还要过滤呢? 注释中说到在本次平衡之后, 只需要清理可能不再属于这个消费者的队列(部分的 topicPartition 抓取队列).

```

private def closeFetchers(cluster: Cluster, messageStreams: Map[TopicPartition, MessageStream])
    // only clear the fetcher queues for certain topic partitions
    val queuesTobeCleared = topicThreadIdAndQueues.filter(q => relevantTopicThreadIdsMap.contains(q.topic))
    closeFetchersForQueues(cluster, messageStreams, queuesTobeCleared)
}

private def closeFetchersForQueues(cluster: Cluster, messageStreams: Map[TopicPartition, MessageStream])
    val allPartitionInfos = topicRegistry.values.map(p => p.value)
    for (partitionInfo ← allPartitionInfos) {
        fetcher match {
            case Some(f) =>
                f.stopConnections          // 停止FetcherManager管理的所有Fetcher
                clearFetcherQueues(allPartitionInfos, cluster, queuesTobeCleared)
                if (config.autoCommitEnable) commitOffsets(true)
        }
    }
}

private def clearFetcherQueues(topicInfos: Iterable[PartitionTopicInfo])
    // Clear all but the currently iterated upon chunk in the consumer
    queuesTobeCleared.foreach(_.clear)
    // Also clear the currently iterated upon chunk in the consumer
    if (messageStreams != null) messageStreams.foreach(_.foreach(_.2.foreach(_.clear)))
}

```

问题: 新创建的 ZKRebalancerListener 中

kafkaMessageAndMetadataStreams(即这里的 messageStreams) 为 空的 Map. 如何清空里面的数据? 实际上 KafkaStream 只是一个迭代器, 在运行过程中会有数据放入到这个流中, 这样流就有数据了.

ConsumerFetcherManager

topicInfos 是上一步 updateFetcher 的 topicRegistry, 是分配给给 consumer 的注册信息: topic->(partition->PartitionTopicInfo).

Fetcher 线程要抓取数据关心的是 PartitionTopicInfo, 首先要找出 Partition Leader(因为只向 Leader Partition 发起抓取请求).

初始时假设所有topicInfos(PartitionTopicInfo)都找不到Leader,即同时加入partitionMap和noLeaderPartitionSet.

在LeaderFinderThread线程中如果找到Leader,则从noLeaderPartitionSet中移除.

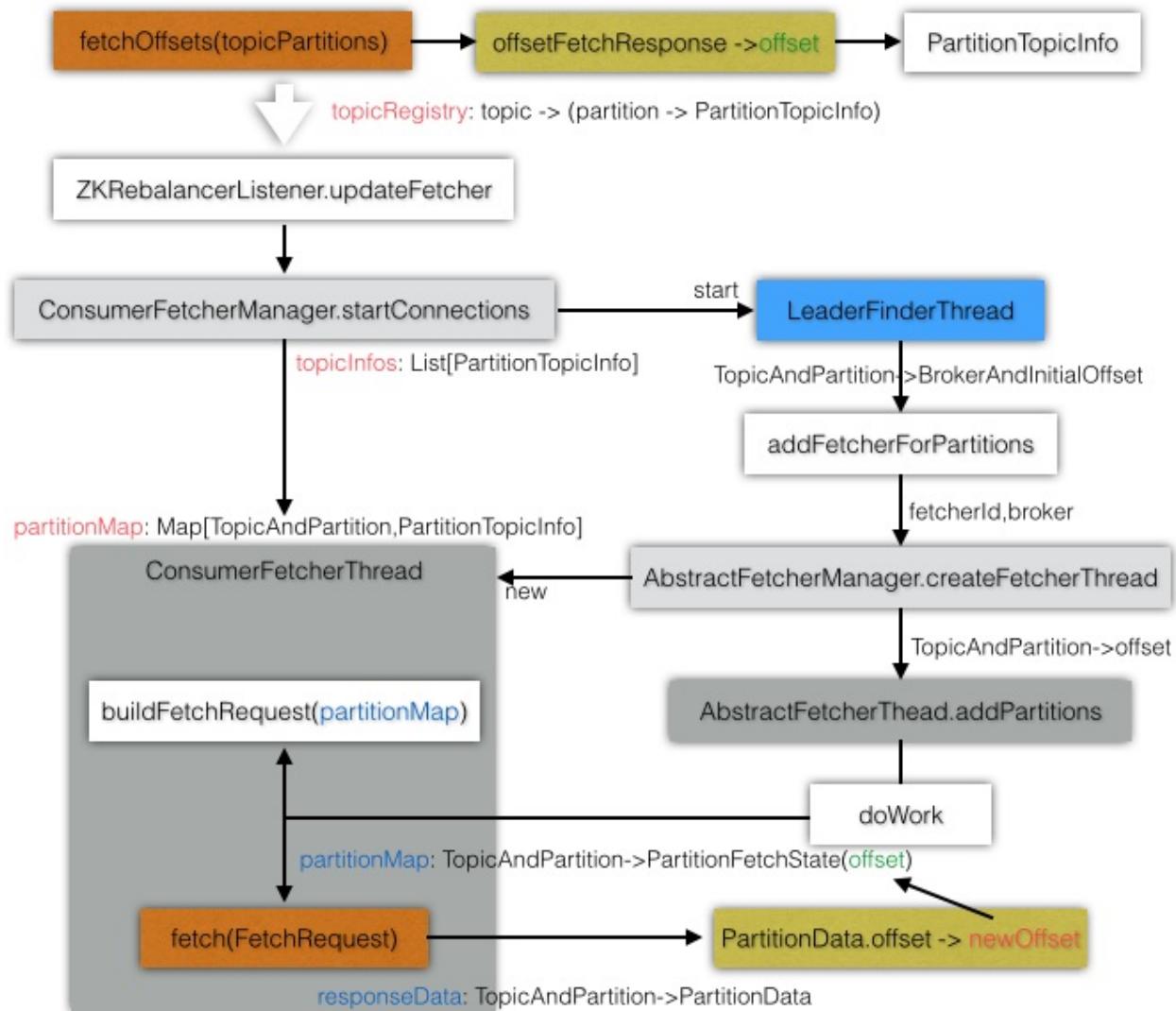
```
def startConnections(topicInfos: Iterable[PartitionTopicInfo], c: ConsumerConfig) {
    val leaderFinderThread = new LeaderFinderThread(consumerIdString + "-leader-finder")
    leaderFinderThread.start()

    inLock(lock) {
        partitionMap = topicInfos.map(tpi => (TopicAndPartition(tpi.topic, tpi.partition), tpi))
        this.cluster = cluster
        noLeaderPartitionSet ++= topicInfos.map(tpi => TopicAndPartition(tpi.topic, tpi.partition))
        cond.signalAll()
    }
}
```

ConsumerFetcherManager管理了当前Consumer的所有Fetcher线程.

注意ConsumerFetcherThread构造函数中的partitionMap和构建FetchRequest时的partitionMap是不同的.

不过它们的相同点是都有offset信息.并且都有fetch操作.



Ref

- <http://uohzoaix.github.io/studies>