Kafka原理学习

C-3 创建: 蒋树奇, 最后修改: 蒋树奇 2016-07-22 11:21

Kafka下的重要概念：

broker：服务器

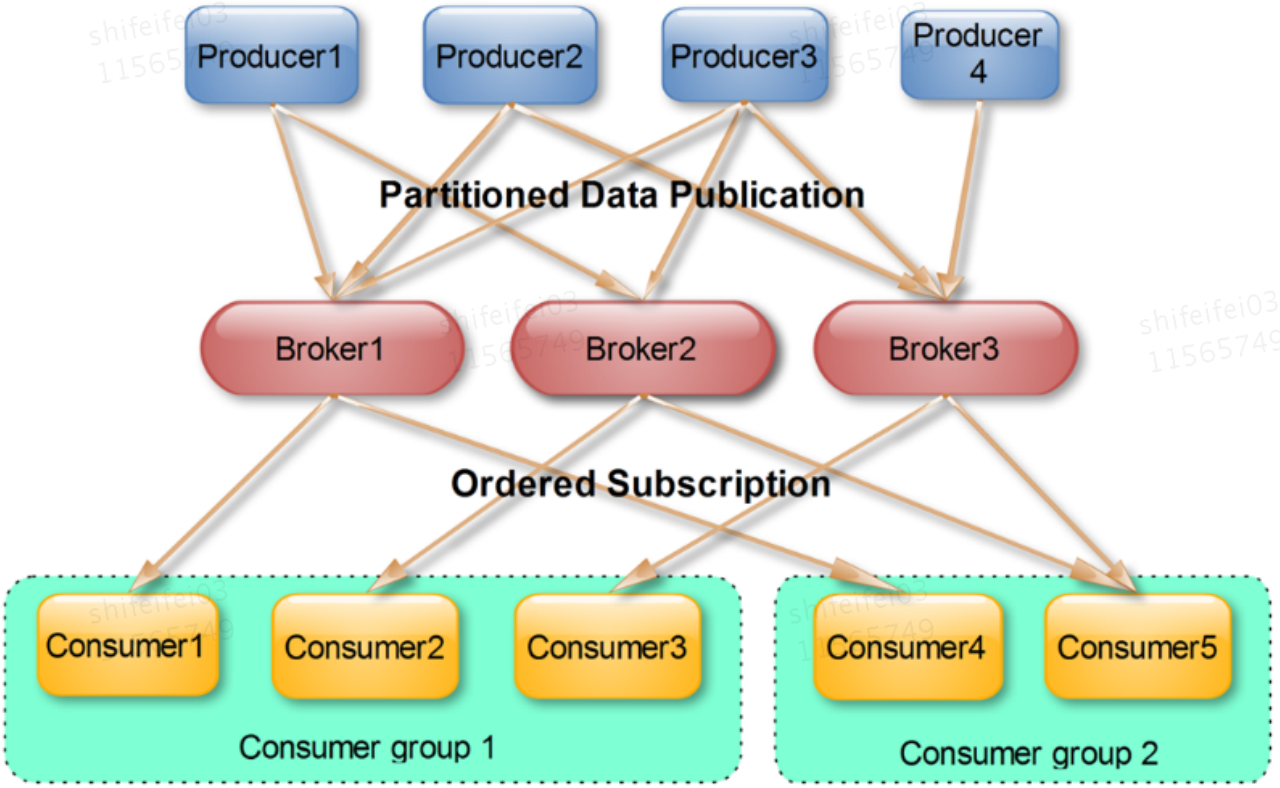
topic 每个消息都要通过topic来区分

partition：每个topic 包含一个或多个partition

Producer: 发消息到broker

Consumer：消费者，从broker读取消息

Consumer Group: 每个Consumer都属于一个group，可以指定，如果不指定那么数目默认group



Message 持久化机制

一个Topic 对应多个Partition（分区），一个partition对应一个实际文件夹，文件夹下存储这个partition的所有消息和索引文件。

每个日志文件都是log Entrie序列。

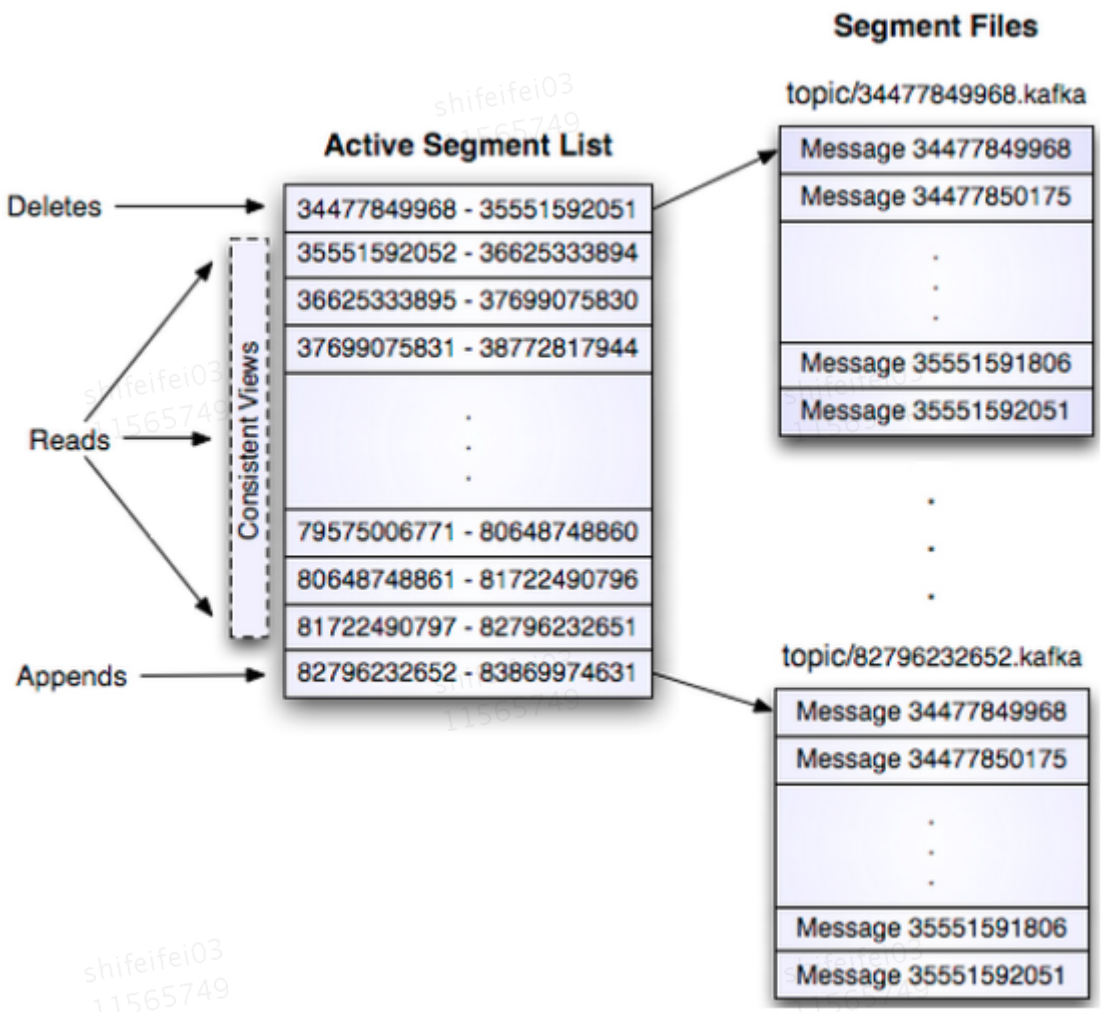
每个Log Entrie 包含 一个4字节的整数值（值N+ 5），1个字节的"magic value"，4个字节的CRC校验码，其后跟N个字节的消息体。

- *message length： 4 bytes (value: 1+4+n)*
- *"magic" value： 1 byte*
- *crc： 4 bytes*

- *payload： n bytes*

Log Entrie 由多个文件来记录，分成了多个segment，每个segment 以该条记录第一个log entire的offset命名，后缀名是 .kafka。

另外会有一个索引文件，标明每个segment下的offset范围。 结构如下图：



每条消息顺序写文件，append到 partition中。 实践证明，顺序写磁盘速度快于随机写内存。

Broker 可以根据 key来决定存到那个partition下面，通过一个合理的机制，可以使得消息均匀分散到不同的partition下面，实现了负载均衡和并发操作。可以配置message的key和对应规则，均匀的分配Message到不同的partition。

kafka不会在读取消息后就删除掉，而是会一直保存。删除策略有两种，一个是按照partition的尺寸，大于某个尺寸后删除；二是按照时间删除。

因为索引机制，kafka随机查询某条消息的时间复杂度为O(1)。

消息消费

Consumer 通过offset来消费消息，每次消费掉一个消息后应该递增offset。kafka broker本身是无状态的，并不记录消息是否被消费。Consumer可以重置较小的offset来重新消费消息。

Consumer 每个partition对应一个线程

Consumer Group : 同一Topic的一条消息只能被同一个Consumer Group内的一个Consumer消费，但多个Consumer Group可同时消费这一消息。

一个Topic可以对应多个consumer group， 如果想要广播，那么只需要每个消费者一个 consumer group 就行了。如果想单播，那么多有的消费者在一个Consumer group 即可。

Producer 向 Broker push, consumer 向 Broker pull 消息。

消息传递

有三种类型的消息传递模式：

- **at most once**: 消息可能会丢，但是绝不会重复；
- **at least once**: 消息绝不会丢，但是可能重复；
- **exactly once**: 消息不多不少，正好一次。

Producer 向broker 发送请求，默认会commit，这样消息就不会丢。如果由于网络问题，没有收到commit，这样producer不知道是不是收到，所以会重新发送。这样就保证了 at least once。

可以通过设置producer发送方式为异步发送，这样就是at most once。

Broker向Consumer发送消息，Consumer可以commit，这样在zk中会保存consumer到这个partition的offset，并可以设置autoCommit。这样下次继续往后读取，基本做到了exactly once。

但是Consumer接收到消息后还需要处理消息，可能在这个过程crash。所以如果提前commit，则相当于消息丢失；如果之后commit，又不能确定前面处理了没有，可能会重复处理。

所以客户端在使用mq时，一般本身业务逻辑要保证幂等性。

Kafka 高可用性

防止某个broker宕机后服务不可用，需要做水平扩展。

多partition：设置某个topic下partition的数量大于broker的数量，这样partition会分配到多个broker上。

多个replica，每个Partition 有多个副本，不同的副本也要放在不同的broker上。

Kafka分配Replica的算法如下：

1. 将所有Broker（假设共n个Broker）和待分配的Partition排序
2. 将第i个Partition分配到第（i mod n）个Broker上
3. 将第i个Partition的第j个Replica分配到第（(i + j) mode n）个Broker上

在有多个备份的情况下，当producer来消息，不可能直接找所有的备份。所以这些replica首先要找出一个leader，然后producer直接与leader通信。

Producer从zk中找到topic下Partition的leader，这个partition会将消息持久化。 其他的 replica 会向leader pull 消息，然后会返回给leader ack。

当leader收到所有ISR的ACK后，会想producer发送ack消息。Follower 发送ack时，并不是持久化完成后再发，而是收到就发。

Leader会跟踪与其保持同步的Replica列表，该列表称为ISR（即in-sync Replica）。ISR有两个要求，一是该broker始终保持着zk的session连接；二是log版本不能落后leader太多，这个阈值可以配置。如果超过阈值，leader就会把follower从isr中移除。

ISR相当于是介于同步复制和异步复制之间的一种模式，如果全部同步，即所有follower都写完log再commit，吞吐量会太低；如果全异步，则如果leader宕机，follower 会丢失数据。 所以通过维护一个ISR，在两种模式直接找了一种平衡。

如果全down了怎么办?

- 等ISR里面的第一个启动后作为leader
- 等所有的replica ，有一个启动了，就作为leader

这两种如何选择，就看对一致性和可用性的不同要求了。

如何选Leader？

Kafka会从所有的broker中选一台作为controller，由controller作为监控着管理者，通过rpc调用，来任命不用的leader。同时负责topic创建和Replica重新分配。

Broker 宕机处理步骤：

1.Controler在ZK上注册watch，broker如果宕机，对应的broker结点就会删除。这时就会触发Controller的Watch。

2.Controller算出set_p集合，这个集合包含了宕机的Broker上所有的partition。

3.对set_p中的每个partition重新计算ISR和leader：

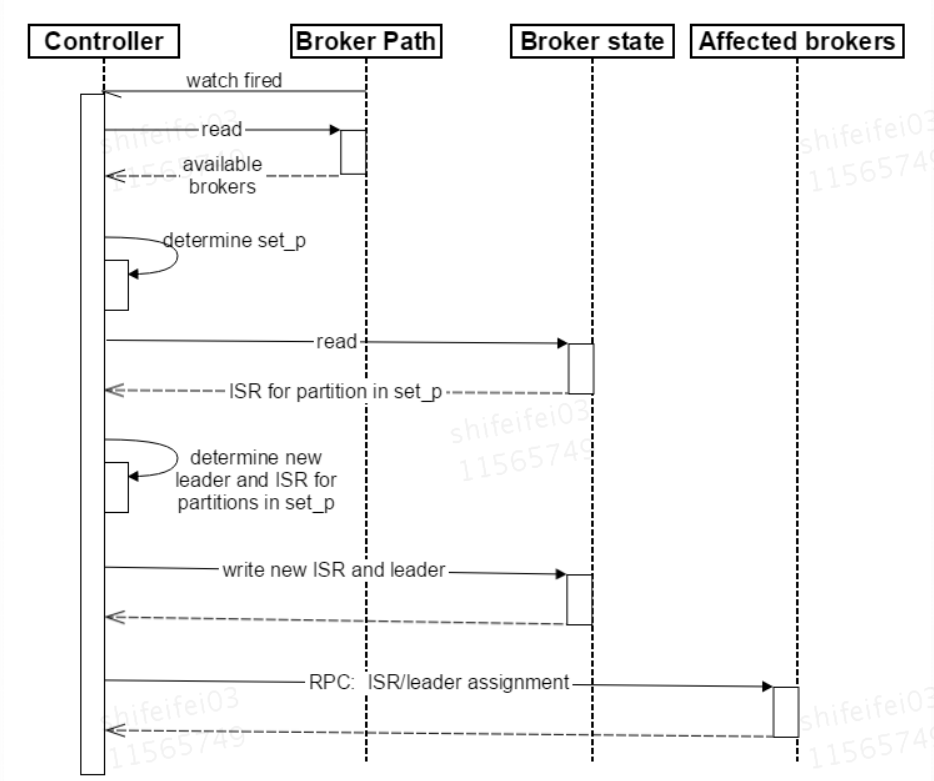
1）从 */brokers/topics/[topic]/partitions/[partition]/state* 读取该Partition当前的ISR

2）如果需要重新任命Leader，如果ISR中有存活的replica ，则任命为leader。如果ISR为空，则从存活的replica里选一个作为leader。

3）将新的Leader，ISR和新的leader_epoch及controller_epoch写入*/brokers/topics/[topic]/partitions/[partition]/state*。注意，该操作只有Controller版本在3.1至3.3的过程中无变化时才会执行，否则跳转到3.1。

4.通过RPC向set_p相关的broker发送LeaderAndISRRequest命令。

时序图如下：



Broker启动处理：

Broker启动后首先根据其ID在Zookeeper的/brokers/idszonde下创建临时子节点（Ephemeral node），创建成功后Controller的ReplicaStateMachine注册其上的Broker Change Watch会被fire，从而通过回调KafkaController.onBrokerStartup方法完成以下步骤：

1.向所有新启动的Broker发送UpdateMetadataRequest，其定义如下。

2.将新启动的Broker上的所有Replica设置为OnlineReplica状态，同时这些Broker会为这些Partition启动high watermark线程。

3.通过partitionStateMachine触发OnlinePartitionStateChange。

Controller宕机处理

每个Broker都会在Controller Path (/controller)上注册一个Watch。

当前Controller失效时，对应的Controller Path会自动消失（因为它是Ephemeral Node），此时该Watch被fire，所有“活”着的Broker都会去竞选成为新的Controller（创建新的Controller Path），但是只会有一个竞选成功（这点由Zookeeper保证）。

竞选成功者即为新的Leader，竞选失败者则重新在新的Controller Path上注册Watch。因为Zookeeper的Watch是一次性的，被fire一次之后即失效，所以需要重新注册。

Broker成功竞选为新Controller后会触发KafkaController.onControllerFailover方法，并在该方法中完成如下操作：

1.读取并增加Controller Epoch。

2.在ReassignedPartitions Path(/admin/reassign_partitions)上注册Watch。

3.在PreferredReplicaElection Path(/admin/preferred_replica_election)上注册Watch。

4.通过partitionStateMachine在Broker Topics Patch(/brokers/topics)上注册Watch。

5.若delete.topic.enable设置为true（默认值是false），则partitionStateMachine在Delete Topic Patch(/admin/delete_topics)上注册Watch。

6.通过replicaStateMachine在Broker Ids Patch(/brokers/ids)上注册Watch。

7.初始化ControllerContext对象，设置当前所有Topic，“活”着的Broker列表，所有Partition的Leader及ISR等。

8.启动replicaStateMachine和partitionStateMachine。

9.将brokerState状态设置为RunningAsController。

10.将每个Partition的Leadership信息发送给所有“活”着的Broker。

11.若auto.leader.rebalance.enable配置为true（默认值是true），则启动partition-rebalance线程。

12.若delete.topic.enable设置为true且Delete Topic Patch(/admin/delete_topics)中有值，则删除相应的Topic。

Broker的通信机制

整个网络通信模块基于Java NIO开发，并采用Reactor模式，其中包含1个Acceptor负责接受客户请求，N个Processor负责读写数据，M个Handler处理业务逻辑。

1.Acceptor的主要职责是监听并接受客户端（请求发起方，包括但不限于Producer，Consumer，Controller，Admin Tool）的连接请求，并建立和客户端的数据传输通道，然后为该客户端指定一个Processor，至此它对该客户端该次请求的任务就结束了，它可以去响应下一个客户端的连接请求了。

2.Processor主要负责从客户端读取数据并将响应返回给客户端，它本身并不处理具体的业务逻辑，并且其内部维护了一个队列来保存分配给它的所有SocketChannel。Processor的run方法会循环从队列中取出新的SocketChannel并将其SelectionKey.OP_READ注册到selector上，然后循环处理已就绪的读（请求）和写（响应）。Processor读取完数据后，将其封装成Request对象并将其交给RequestChannel。

3.RequestChannel是Processor和KafkaRequestHandler交换数据的地方，它包含一个队列requestQueue用来存放Processor加入的Request，KafkaRequestHandler会从里面取出Request来处理；同时它还包含一个respondQueue，用来存放KafkaRequestHandler处理完Request后返还给客户端的Response。

4.Processor会通过processNewResponses方法依次将requestChannel中responseQueue保存的Response取出，并将对应的SelectionKey.OP_WRITE事件注册到selector上。当selector的select方法返回时，对检测到的可写通道，调用write方法将Response返回给客户端。

5.KafkaRequestHandler循环从RequestChannel中取Request并交给kafka.server.KafkaApis处理具体的业务逻辑。

Follower从Leader Fetch数据

Follower 向 Leader 获取数据 与Consumer向broker获取数据都是用FetchRequest来进行请求。

Leader收到Fetch请求后，Kafka通过KafkaApis.handleFetchRequest响应该请求，响应过程如下：

1.replicaManager根据请求读出数据存入dataRead中。

2.如果该请求来自Follower则更新其相应的LEO（log end offset）以及相应Partition的High Watermark；

3.根据dataRead算出可读消息长度（单位为字节）并存入bytesReadable中。

4.满足下面4个条件中的1个，则立即将相应的数据返回

1）Fetch请求不希望等待，即fetchRequest.macWait <= 0

2）Fetch请求不要求一定能取到消息，即fetchRequest.numPartitions <= 0，也即requestInfo为空

3）有足够的数据可供返回，即bytesReadable >= fetchRequest.minBytes

4）读取数据时发生异常

若不满足以上4个条件，FetchRequest将不会立即返回，并将该请求封装成DelayedFetch。检查该DeplayedFetch是否满足，若满足则返回请求，否则将该请求加入Watch列表。