

kafka的简介

一、简介

1.1 概述

Kafka是最初由Linkedin公司开发，是一个分布式、分区的、多副本的、多订阅者，基于zookeeper协调的分布式日志系统（也可以当做MQ系统），常见可以用于web/nginx日志、访问日志，消息服务等等，Linkedin于2010年贡献给了Apache基金会并成为顶级开源项目。

主要应用场景是：日志收集系统和消息系统。

Kafka主要设计目标如下：

- 以时间复杂度为 $O(1)$ 的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间的访问性能。
- 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条消息的传输。
- 支持Kafka Server间的消息分区，及分布式消费，同时保证每个partition内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。
- Scale out:支持在线水平扩展

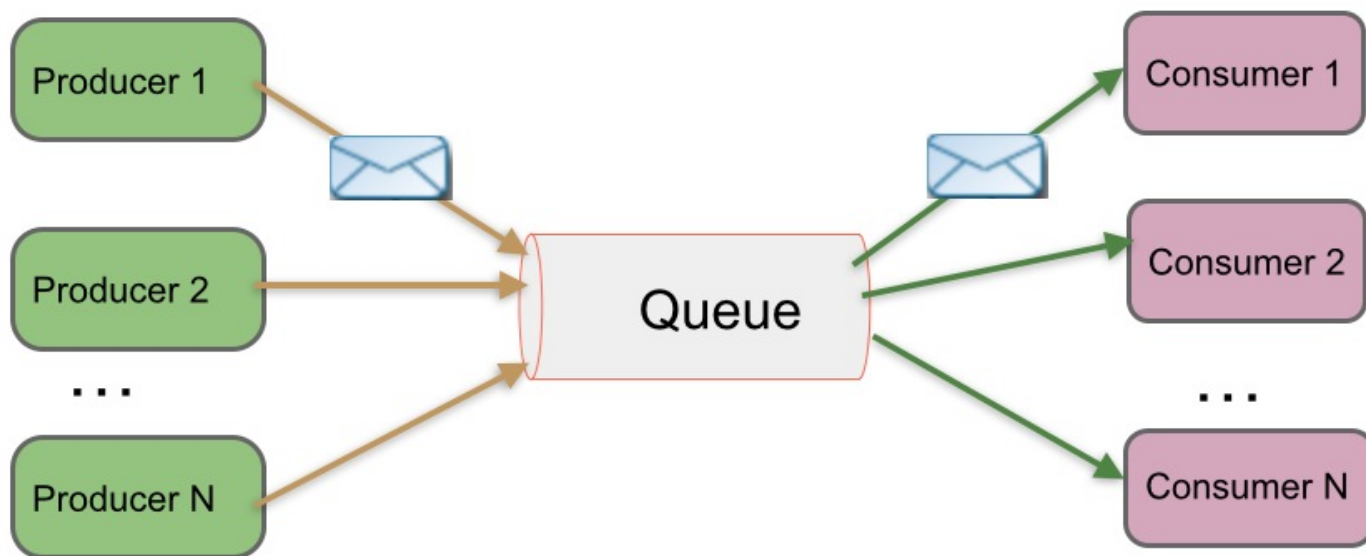
1.2 消息系统介绍

一个消息系统负责将数据从一个应用传递到另外一个应用，应用只需关注于数据，无需关注数据在两个或多个应用间是如何传递的。分布式消息传递基于可靠的消息队列，在客户端应用和消息系统之间异步传递消息。有两种主要的消息传递模式：**点对点传递模式**、**发布-订阅模式**。大部分的消息系统选用发布-订阅模式。**Kafka就是一种发布-订阅模式**。

1.3 点对点消息传递模式

在点对点消息系统中，消息持久化到一个队列中。此时，将有一个或多个消费者消费队列中的数据。但是一条消息只能被消费一次。当一个消费者消费了队列中的某条数据之后，该条数据则从消息队列中删除。该模式即使有多个消费者同时消费数据，也能保证数据处理的顺序。这种架构描述示意图如下：

消息队列-点对点

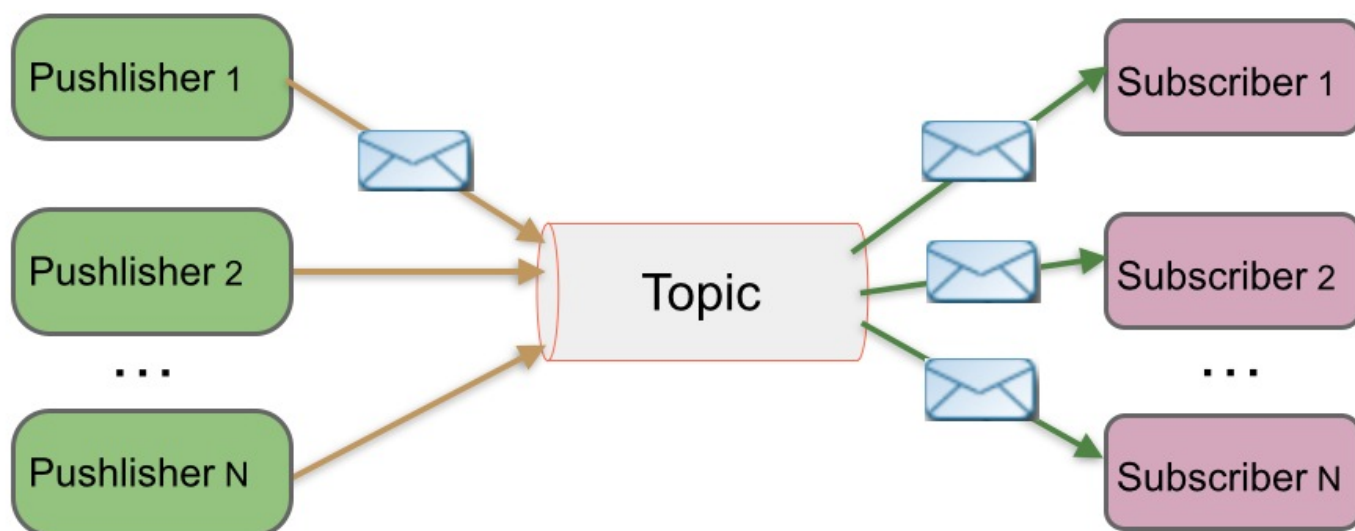


生产者发送一条消息到queue，只有一个消费者能收到。

1.4 发布-订阅消息传递模式

在发布-订阅消息系统中，消息被持久化到一个topic中。与点对点消息系统不同的是，消费者可以订阅一个或多个topic，消费者可以消费该topic中所有的数据，同一条数据可以被多个消费者消费，数据被消费后不会立马删除。在发布-订阅消息系统中，消息的生产者称为发布者，消费者称为订阅者。该模式的示例图如下：

消息队列-发布订阅



发布者发送到topic的消息，只有订阅了topic的订阅者才会收到消息。

二、Kafka的优点

2.1 解耦

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

2.2 冗余（副本）

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

2.3 扩展性

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

2.4 灵活性&峰值处理能力

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

2.5 可恢复性

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

2.6 顺序保证

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka保证一个Partition内的消息的有序性。

2.7 缓冲

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行——写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

2.8 异步通信

很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

三、常用Message Queue对比

3.1 RabbitMQ

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

3.2 Redis

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于

RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

3.3 ZeroMQ

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演这个服务器角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty作为传输模块）。

3.4 ActiveMQ

ActiveMQ是Apache下的一个子项目。类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

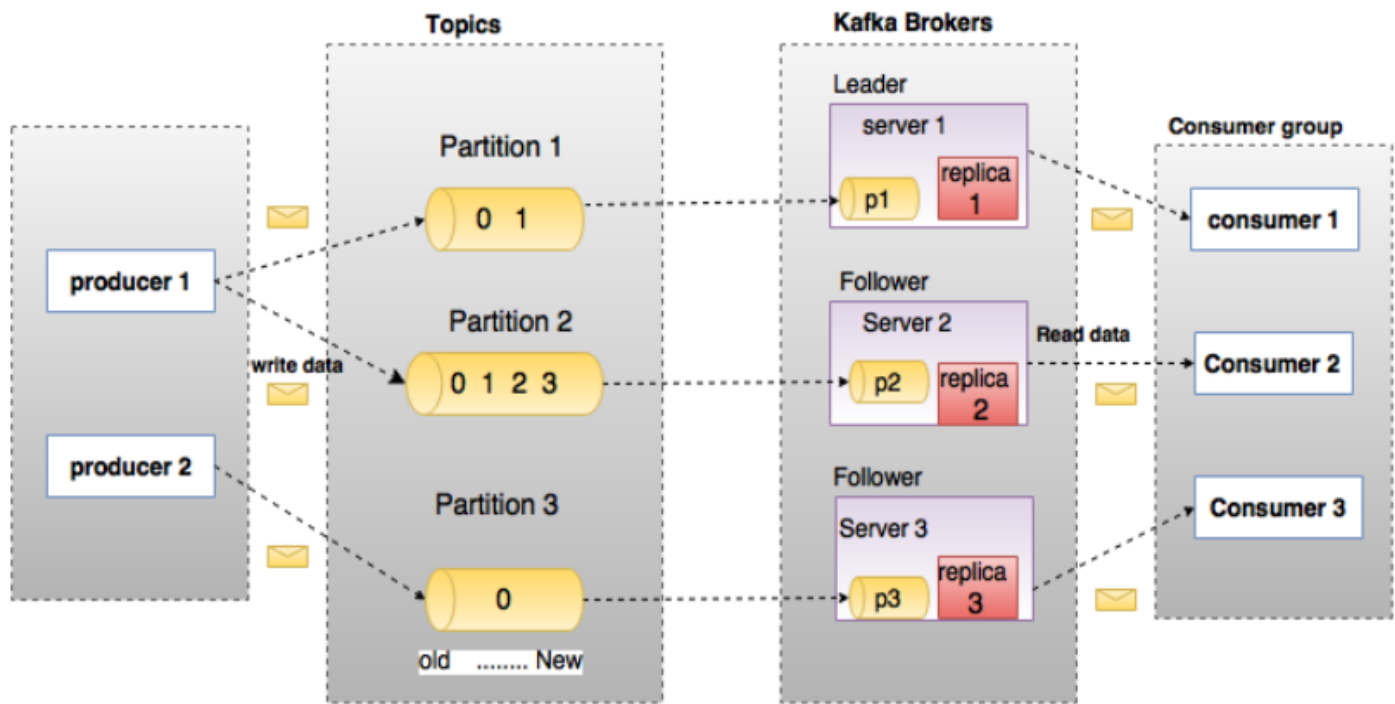
3.5 Kafka/Jafka

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

四、Kafka中的术语解释

4.1 概述

在深入理解Kafka之前，先介绍一下Kafka中的术语。下图展示了Kafka的相关术语以及之间的关系：



上图中一个topic配置了3个partition。Partition1有两个offset：0和1。Partition2有4个offset。Partition3有1个offset。副本的id和副本所在的机器的id恰好相同。

如果一个topic的副本数为3，那么Kafka将在集群中为每个partition创建3个相同的副本。集群中的每个broker存储一个或多个partition。多个producer和consumer可同时生产和消费数据。

4.2 broker

Kafka 集群包含一个或多个服务器，服务器节点称为broker。

broker存储topic的数据。如果某topic有N个partition，集群有N个broker，那么每个broker存储该topic的一个partition。

如果某topic有N个partition，集群有(N+M)个broker，那么其中有N个broker存储该topic的一个partition，剩下的M个broker不存储该topic的partition数据。

如果某topic有N个partition，集群中broker数目少于N个，那么一个broker存储该topic的一个或多个partition。在实际生产环境中，尽量避免这种情况的发生，这种情况容易导致Kafka集群数据不均衡。

4.3 Topic

每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个broker上但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处）

类似于数据库的表名

4.3 Partition

topic中的数据分割为一个或多个partition。每个topic至少有一个partition。每个partition中的数据使用多个segment文件存储。partition中的数据是有序的，不同partition间的数据丢失了数据的顺序。如果topic有多个partition，消费数据时就不能保证数据的顺序。在需要严格保证消息的消费顺序的场景下，需要将partition数目设为1。

4.4 Producer

生产者即数据的发布者，该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后，broker将该消息追加到当前用于追加数据的segment文件中。生产者发送的消息，存储到一个partition中，生产者也可以指定数据存储的partition。

4.5 Consumer

消费者可以从broker中读取数据。消费者可以消费多个topic中的数据。

4.6 Consumer Group

每个Consumer属于一个特定的Consumer Group（可为每个Consumer指定group name，若不指定group name则属于默认的group）。

4.7 Leader

每个partition有多个副本，其中有且仅有一个作为Leader，Leader是当前负责数据的读写的partition。

4.8 Follower

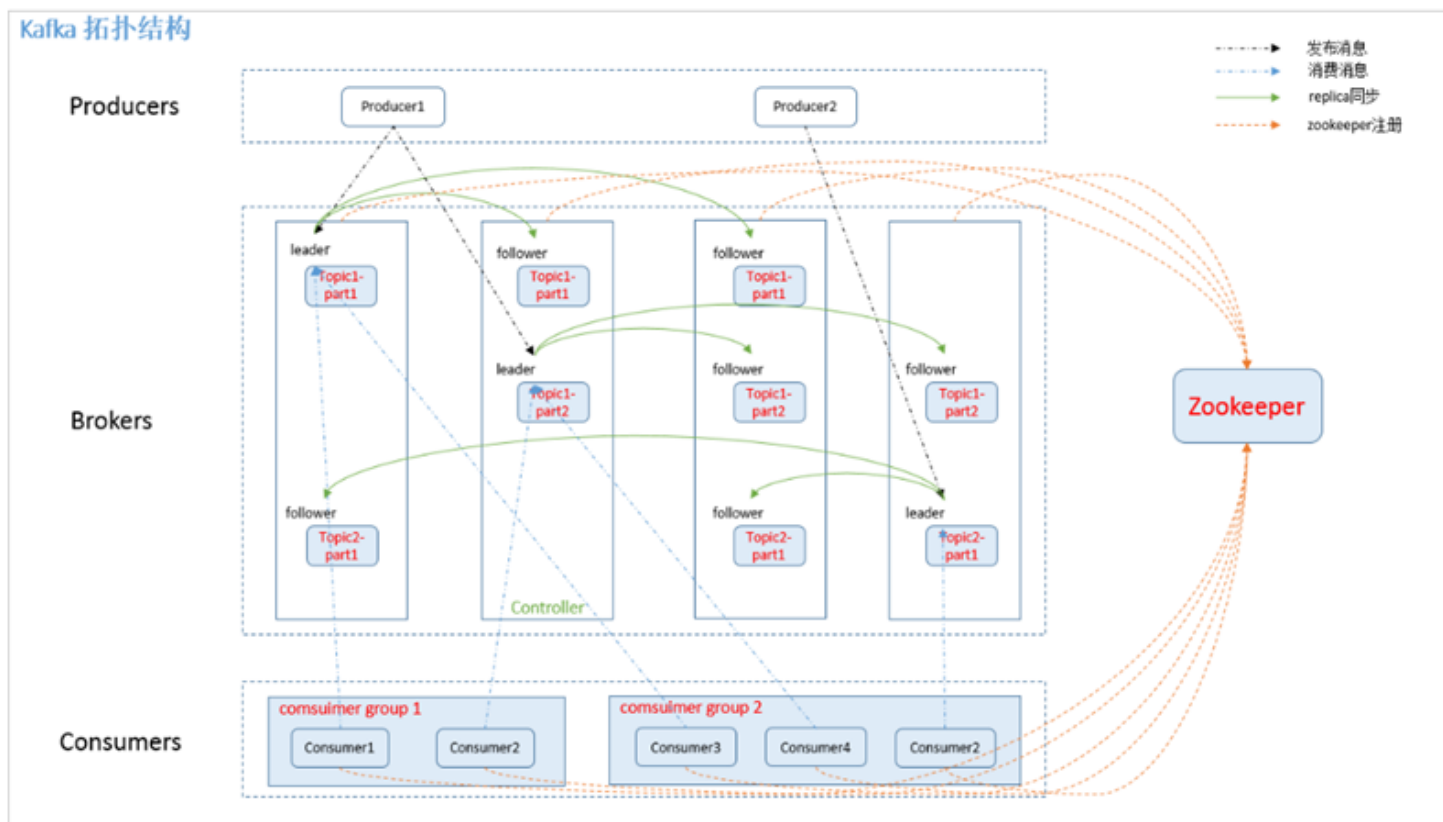
Follower跟随Leader，所有写请求都通过Leader路由，数据变更会广播给所有Follower，Follower与Leader保持数据同步。如果Leader失效，则从Follower中选举出一个新的Leader。当Follower与Leader挂掉、卡住或者同步太慢，leader会把这个follower从“in sync replicas”（ISR）列表中删除，重新创建一个Follower。

参考

<https://www.cnblogs.com/qingyunzong/p/9004509.html>

一、Kafka的架构

一、Kafka的架构

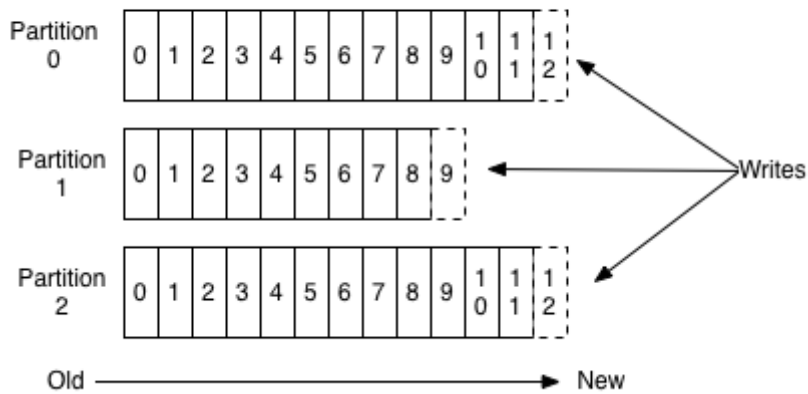


如上图所示，一个典型的Kafka集群中包含若干Producer（可以是web前端产生的Page View，或者是服务器日志，系统CPU、Memory等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干Consumer Group，以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在Consumer Group发生变化时进行rebalance。Producer使用push模式将消息发布到broker，Consumer使用pull模式从broker订阅并消费消息。

二、Topics和Partition

Topic在逻辑上可以被认为是一个queue，每条消费都必须指定它的Topic，可以简单理解为必须指明把这条消息放进哪个queue里。为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，每个Partition在物理上对应一个文件夹，该文件夹下存储这个Partition的所有消息和索引文件。创建一个topic时，同时可以指定分区数目，分区数越多，其吞吐量也越大，但是需要的资源也越多，同时也会导致更高的不可用性，kafka在接收到生产者发送的消息之后，会根据均衡策略将消息存储到不同的分区中。因为每条消息都被append到该Partition中，属于顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）。

Anatomy of a Topic



对于传统的message queue而言，一般会删除已经被消费的消息，而Kafka集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此Kafka提供两种策略删除旧数据。一是基于时间，二是基于Partition文件大小。例如可以通过配置 `$KAFKA_HOME/config/server.properties`，让Kafka删除一周前的数据，也可在Partition文件超过1GB时删除旧数据，配置如下所示：

```
# The minimum age of a log file to be eligible for deletion
log.retention.hours=168
# The maximum size of a log segment file. When this size is reached a new log segment will be created.
log.segment.bytes=1073741824
# The interval at which log segments are checked to see if they can be deleted according to the retention policies
log.retention.check.interval.ms=300000
# If log.cleaner.enable=true is set the cleaner will be enabled and individual logs can then be marked for log compaction
log.cleaner.enable=false
```

因为Kafka读取特定消息的时间复杂度为 $O(1)$ ，即与文件大小无关，所以这里删除过期文件与提高Kafka性能无关。选择怎样的删除策略只与磁盘以及具体的需求有关。另外，Kafka会为每一个Consumer Group保留一些metadata信息——当前消费的消息的position，也即offset。这个offset由Consumer控制。正常情况下Consumer会在消费完一条消息后递增该offset。当然，Consumer也可将offset设成一个较小的值，重新消费一些消息。因为offset由Consumer控制，所以Kafka broker是无状态的，它不需要标记哪些消息被哪些消费过，也不需要通过broker去保证同一个Consumer Group只有一个Consumer能消费某一条消息，因此也就不需要锁机制，这也为Kafka的高吞吐率提供了有力保障。

三、Producer消息路由

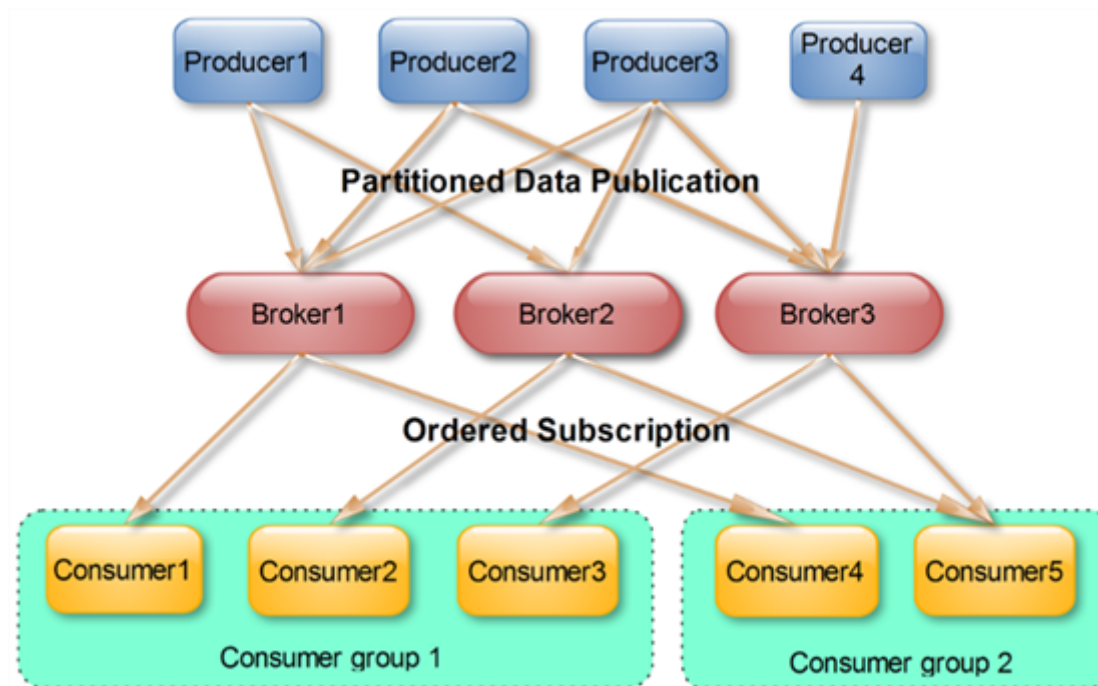
Producer发送消息到broker时，会根据Partition机制选择将其存储到哪一个Partition。如果Partition机制设置合理，所有消息可以均匀分布到不同的Partition里，这样就实现了负载均衡。如果一个Topic对应一个文件，那这个文件所在的机器I/O将会成为这个Topic的性能瓶颈，而有了Partition后，不同的消息可以并行写入不同broker的不同Partition里，极大的提高了吞吐率。可以在

\$KAFKA_HOME/config/server.properties中通过配置项num.partitions来指定新建Topic的默认Partition数量，也可在创建Topic时通过参数指定，同时也可以Topic创建之后通过Kafka提供的工具修改。

在发送一条消息时，可以指定这条消息的key，Producer根据这个key和Partition机制来判断应该将这条消息发送到哪个Partition。Partition机制可以通过指定Producer的partition.class这一参数来指定，该class必须实现kafka.producer.Partitioner接口。

四、Consumer Group

使用Consumer high level API时，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。



这是Kafka用来实现一个Topic消息的广播（发给所有的Consumer）和单播（发给某一个Consumer）的手段。一个Topic可以对应多个Consumer Group。如果需实现广播，只要每个Consumer有一个独立的Group就可以了。要实现单播只要所有的Consumer在同一个Group里。用Consumer Group还可以将Consumer进行自由的分组而不需要多次发送消息到不同的Topic。

实际上，Kafka的设计理念之一就是同时提供离线处理和实时处理。根据这一特性，可以使用Storm这种实时流处理系统对消息进行实时在线处理，同时使用Hadoop这种批处理系统进行离线处理，还可以同时将数据实时备份到另一个数据中心，只需要保证这三个操作所使用的Consumer属于不同的Consumer Group即可。

五、Push vs. Pull

作为一个消息系统，Kafka遵循了传统的方式，选择由Producer向broker push消息并由Consumer从broker pull消息。一些logging-centric system，比如Facebook的Scribe和Cloudera的Flume，采用push模式。事实上，push模式和pull模式各有优劣。

push模式很难适应消费速率不同的消费者，因为消息发送速率是由broker决定的。push模式的目标是尽可能以最快速度传递消息，但是这样很容易造成Consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据Consumer的消费能力以适当的速率消费消息。

对于Kafka而言，pull模式更合适。pull模式可简化broker的设计，Consumer可自主控制消费消息的速率，同时Consumer可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

六、Kafka delivery guarantee

有这么几种可能的delivery guarantee：

At most once	消息可能会丢，但绝不会重复传输
At least one	消息绝不会丢，但可能会重复传输
Exactly once	每条消息肯定会被传输一次且仅传输一次，很多时候这是用户所想要的。

当Producer向broker发送消息时，一旦这条消息被commit，因数replication的存在，它就不会丢。但是如果Producer发送数据给broker后，遇到网络问题而造成通信中断，那Producer就无法判断该条消息是否已经commit。虽然Kafka无法确定网络故障期间发生了什么，但是Producer可以生成一种类似于主键的东西，发生故障时幂等性的重试多次，这样就做到了Exactly once。

接下来讨论的是消息从broker到Consumer的delivery guarantee语义。（仅针对Kafka consumer high level API）。Consumer在从broker读取消息后，可以选择commit，该操作会在Zookeeper中保存该Consumer在该Partition中读取的消息的offset。该Consumer下一次再读该Partition时会从下一条开始读取。如未commit，下一次读取的开始位置会跟上一次commit之后的开始位置相同。当然可以将Consumer设置为autocommit，即Consumer一旦读到数据立即自动commit。如果只讨论这一读取消息的过程，那Kafka是确保了Exactly once。但实际使用中应用程序并非在Consumer读取完数据就结束了，而是要进行进一步处理，而数据处理与commit的顺序在很大程度上决定了消息从broker和consumer的delivery guarantee semantic。

Kafka默认保证At least once，并且允许通过设置Producer异步提交来实现At most once。而Exactly once要求与外部存储系统协作，幸运的是Kafka提供的offset可以非常直接非常容易得使用这种方式。

参考

kafka的高可用

一、高可用的由来

1.1 为何需要Replication

在Kafka在0.8以前的版本中，是没有Replication的，一旦某一个Broker宕机，则其上所有的Partition数据都不可被消费，这与Kafka数据持久性及Delivery Guarantee的设计目标相悖。同时Producer都不能再将数据存于这些Partition中。

如果Producer使用同步模式则Producer会在尝试重新发送message.send.max.retries（默认值为3）次后抛出Exception，用户可以选择停止发送后续数据也可选择继续选择发送。而前者会造成数据的阻塞，后者会造成本应发往该Broker的数据的丢失。

如果Producer使用异步模式，则Producer会尝试重新发送message.send.max.retries（默认值为3）次后记录该异常并继续发送后续数据，这会造成数据丢失并且用户只能通过日志发现该问题。同时，Kafka的Producer并未对异步模式提供callback接口。

由此可见，在没有Replication的情况下，一旦某机器宕机或者某个Broker停止工作则会造成整个系统的可用性降低。随着集群规模的增加，整个集群中出现该类异常的几率大大增加，因此对于生产系统而言Replication机制的引入非常重要。

1.2 Leader Election

引入Replication之后，同一个Partition可能会有多个Replica，而这时需要在这些Replication之间选出一个Leader，Producer和Consumer只与这个Leader交互，其它Replica作为Follower从Leader中复制数据。

因为需要保证同一个Partition的多个Replica之间的数据一致性（其中一个宕机后其它Replica必须要能继续服务并且即不能造成数据重复也不能造成数据丢失）。如果没有一个Leader，所有Replica都可同时读/写数据，那就需要保证多个Replica之间互相（ $N \times N$ 条通路）同步数据，数据的一致性和有序性非常难保证，大大增加了Replication实现的复杂性，同时也增加了出现异常的几率。而引入Leader后，只有Leader负责数据读写，Follower只向Leader顺序Fetch数据（ N 条通路），系统更加简单且高效。

二、Kafka HA设计解析

2.1 如何将所有Replica均匀分布到整个集群

为了更好的做负载均衡，Kafka尽量将所有的Partition均匀分配到整个集群上。一个典型的部署方式是一个Topic的Partition数量大于Broker的数量。同时为了提高Kafka的容错能力，也需要将同一个Partition的Replica尽量分散到不同的机器。实际上，如果所有的Replica都在同一个Broker上，那一旦该Broker宕机，该Partition的所有Replica都无法工作，也就达不到HA的效果。同时，如果某个Broker宕机了，需要保证它上面的负载可以被均匀的分配到其它幸存的所有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 $(i \bmod n)$ 个Broker上
- 3.将第i个Partition的第j个Replica分配到第 $((i + j) \bmod n)$ 个Broker上

2.2 Data Replication（副本策略）

Kafka的高可靠性的保障来源于其健壮的副本（replication）策略。

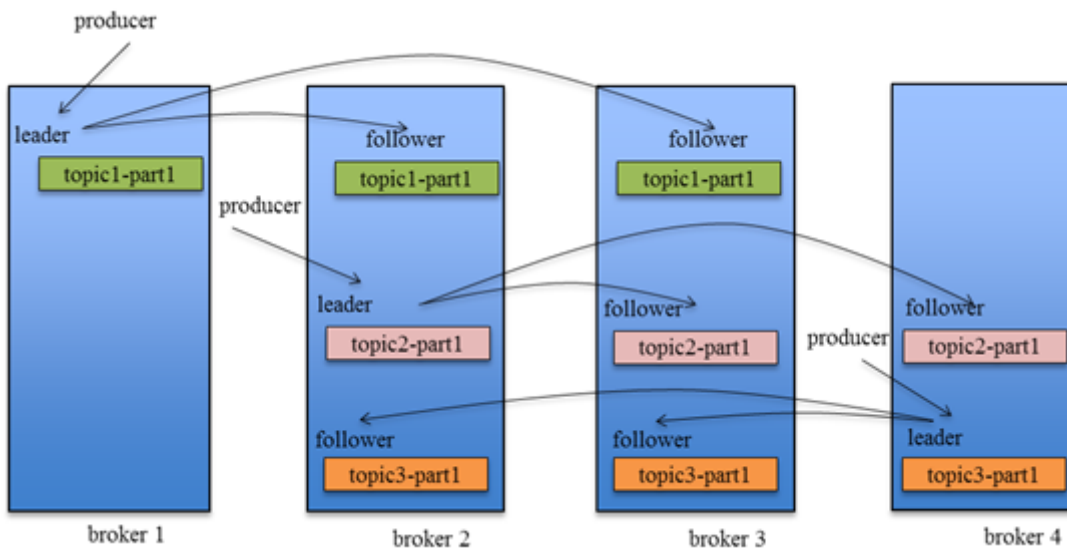
2.2.1 消息传递同步策略

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，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。

Consumer读消息也是从Leader读取，只有被commit过的消息才会暴露给Consumer。

Kafka Replication的数据流如下图所示：



2.2.2 ACK前需要保证有多少个备份

对于Kafka而言，定义一个Broker是否“活着”包含两个条件：

- 一是它必须维护与ZooKeeper的session（这个通过ZooKeeper的Heartbeat机制来实现）。
- 二是Follower必须能够及时将Leader的消息复制过来，不能“落后太多”。

Leader会跟踪与其保持同步的Replica列表，该列表称为ISR（即in-sync Replica）。如果一个Follower宕机，或者落后太多，Leader将把它从ISR中移除。这里所描述的“落后太多”指Follower复制的消息落后于Leader后的条数超过预定值（该值可在\$KAFKA_HOME/config/server.properties中通过replica.lag.max.messages配置，其默认值是4000）或者Follower超过一定时间（该值可在\$KAFKA_HOME/config/server.properties中通过replica.lag.time.max.ms来配置，其默认值是10000）未向Leader发送fetch请求。

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，完全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（高吞吐率是Kafka非常重要的一个特性）。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。

需要说明的是，Kafka只解决fail/recover，不处理“Byzantine”（“拜占庭”）问题。一条消息只有被ISR里的所有Follower都从Leader复制过去才会被认为已提交。这样就避免了部分数据被写进了Leader，还没来得及被任何Follower复制就宕机了，而造成数据丢失（Consumer无法消费这些数据）。而对于Producer而言，它可以选择是否等待消息commit，这可以通过request.required.acks来设置。这种机制确保了只要ISR有一个或以上的Follower，一条被commit的消息就不会丢失。

2.2.3 Leader Election算法

Leader选举本质上是一个分布式锁，有两种方式实现基于ZooKeeper的分布式锁：

- 节点名称唯一性：多个客户端创建一个节点，只有成功创建节点的客户端才能获得锁
- 临时顺序节点：所有客户端在某个目录下创建自己的临时顺序节点，只有序号最小的才获得锁

一种非常常用的选举leader的方式是“Majority Vote”（“少数服从多数”），但Kafka并未采用这种方式。这种模式下，如果我们有 $2f+1$ 个Replica（包含Leader和Follower），那在commit之前必须保证有 $f+1$ 个Replica复制完消息，为了保证正确选出新的Leader，fail的Replica不能超过 f 个。因为在剩下的任意 $f+1$ 个Replica里，至少有一个Replica包含有最新的所有消息。这种方式有个很大的优势，系统的latency只取决于最快的几个Broker，而非最慢那个。Majority Vote也有一些劣势，为了保证Leader Election的正常进行，它所能容忍的fail的follower个数比较少。如果要容忍1个follower挂掉，必须要有3个以上的Replica，如果要容忍2个Follower挂掉，必须要有5个以上的Replica。也就是说，在生产环境下为了保证较高的容错程度，必须要有大量的Replica，而大量的Replica又会在大数据量下导致性能的急剧下降。这就是这种算法更多用在ZooKeeper这种共享集群配置的系统而很少在需要存储大量数据的系统中使用的原因。例如HDFS的HA Feature是基于majority-vote-based journal，但是它的数据存储并没有使用这种方式。

Kafka在ZooKeeper中动态维护了一个ISR（in-sync replicas），这个ISR里的所有Replica都跟上了leader，只有ISR里的成员才有被选为Leader的可能。在这种模式下，对于 $f+1$ 个Replica，一个Partition能在保证不丢失已经commit的消息的前提下容忍 f 个Replica的失败。在大多数使用场景中，这种模式是非常有利的。事实上，为了容忍 f 个Replica的失败，Majority Vote和ISR在commit前需要等待的Replica数量是一样的，但是ISR需要的总的Replica的个数几乎是Majority Vote的一半。

虽然Majority Vote与ISR相比有不需等待最慢的Broker这一优势，但是Kafka作者认为Kafka可以通过Producer选择是否被commit阻塞来改善这一问题，并且节省下来的Replica和磁盘使得ISR模式仍然值得。

2.2.4 如何处理所有Replica都不工作

在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某个Partition的所有Replica都宕机了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

- 1.等待ISR中的任一个Replica“活”过来，并且选它作为Leader
- 2.选择第一个“活”过来的Replica（不一定是ISR中的）作为Leader

这就需要在可用性和一致性当中作出一个简单的折衷。如果一定要等待ISR中的Replica“活”过来，那不可用的时间就可能会相对较长。而且如果ISR中的所有Replica都无法“活”过来了，或者数据都丢失了，这个Partition将永远不可用。选择第一个“活”过来的Replica作为Leader，而这个Replica不是ISR中的Replica，那即使它并不保证已经包含了所有已commit的消息，它也会成为Leader而作为consumer的数据源（前文有说明，所有读写都由Leader完成）。Kafka0.8.*使用了第二种方式。根据Kafka的文档，在

以后的版本中，Kafka支持用户通过配置选择这两种方式中的一种，从而根据不同的使用场景选择高可用性还是强一致性。

2.2.5 选举Leader

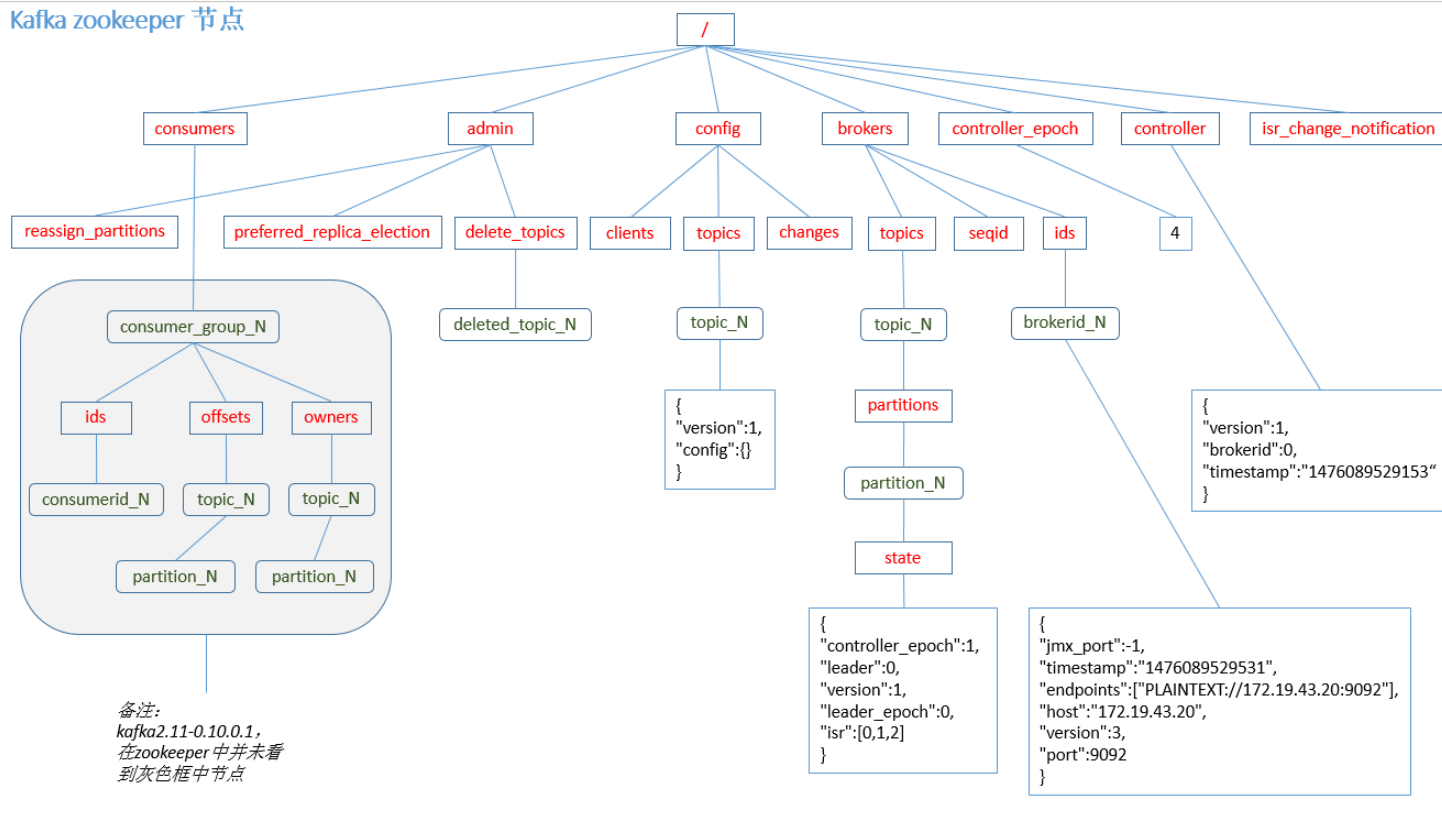
最简单最直观的方案是，所有Follower都在ZooKeeper上设置一个Watch，一旦Leader宕机，其对应的ephemeral znode会自动删除，此时所有Follower都尝试创建该节点，而创建成功者（ZooKeeper保证只有一个能创建成功）即是新的Leader，其它Replica即为Follower。

但是该方法会有3个问题：

- 1.split-brain 这是由ZooKeeper的特性引起的，虽然ZooKeeper能保证所有Watch按顺序触发，但并不能保证同一时刻所有Replica“看”到的状态是一样的，这就可能造成不同Replica的响应不一致
- 2.herd effect 如果宕机的那个Broker上的Partition比较多，会造成多个Watch被触发，造成集群内大量的调整
- 3.ZooKeeper负载过重 每个Replica都要为此在ZooKeeper上注册一个Watch，当集群规模增加到几千个Partition时ZooKeeper负载会过重。

Kafka 0.8.*的Leader Election方案解决了上述问题，它在所有broker中选出一个controller，所有Partition的Leader选举都由controller决定。controller会将Leader的改变直接通过RPC的方式（比ZooKeeper Queue的方式更高效）通知需为此作为响应的Broker。同时controller也负责增删Topic以及Replica的重新分配。

三、HA相关ZooKeeper结构



3.1 admin

该目录下znode只有在有相关操作时才会存在，操作结束时将其删除

/admin/reassign_partitions用于将一些Partition分配到不同的broker集合上。对于每个待重新分配的Partition，Kafka会在该znode上存储其所有的Replica和相应的Broker id。该znode由管理进程创建并且一旦重新分配成功它将会被自动移除。

3.2 broker

即/brokers/ids/[brokerId] 存储“活着”的broker信息。

topic注册信息 (/brokers/topics/[topic])，存储该topic的所有partition的所有replica所在的broker id，第一个replica即为preferred replica，对一个给定的partition，它在同一个broker上最多只有一个replica,因此broker id可作为replica id。

3.3 controller

/controller -> int (broker id of the controller)存储当前controller的信息

/controller_epoch -> int (epoch)直接以整数形式存储controller epoch，而非像其它znode一样以JSON字符串形式存储。

四、producer发布消息

4.1 写入方式

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 partition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率）。

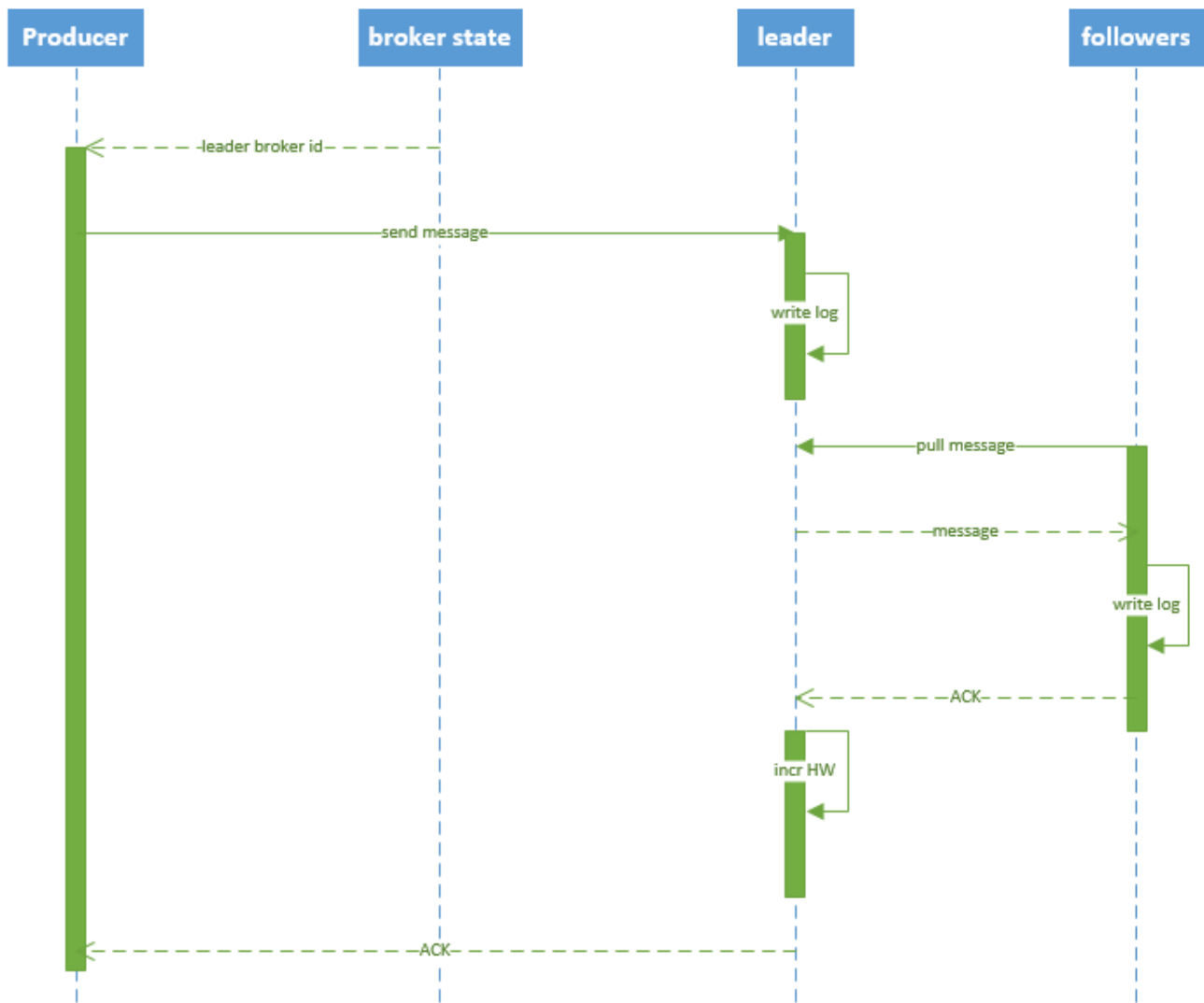
4.2 消息路由

producer 发送消息到 broker 时，会根据分区算法选择将其存储到哪一个 partition。其路由机制为：

- 1、指定了 partition，则直接使用；
- 2、未指定 partition 但指定 key，通过对 key 的 value 进行hash 选出一个 partition
- 3、partition 和 key 都未指定，使用轮询选出一个 partition。

4.3 写入流程

producer 写入消息序列图如下所示：



流程说明：

- 1、producer 先从 zookeeper 的 `"/brokers/.../state"` 节点找到该 partition 的 leader
- 2、producer 将消息发送给该 leader
- 3、leader 将消息写入本地 log
- 4、followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK
- 5、leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW (high watermark, 最后 commit 的 offset) 并向 producer

五、broker保存消息

5.1 存储方式

物理上把 topic 分成一个或多个 partition (对应 `server.properties` 中的 `num.partitions=3` 配置)，每个 partition 物理上对应一个文件夹 (该文件夹存储该 partition 的所有消息和索引文件)，如下：

```

drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-0/
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-1/
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-2/
-rw-r--r--  1 root root    0 Oct 10 00:48 .lock
-rw-r--r--  1 root root   54 Oct 10 00:48 meta.properties
-rw-r--r--  1 root root 1254 Oct 10 04:56 recovery-point-offset-checkpoint
-rw-r--r--  1 root root 1256 Oct 10 04:57 replication-offset-checkpoint
feng@ubuntu:/tmp/kafka/kafka-logs-1$ cd demoTopic2-0
feng@ubuntu:/tmp/kafka/kafka-logs-1/demoTopic2-0$ ll
total 12
drwxr-xr-x  2 root root    4096 Oct 10 01:54 ./
drwxr-xr-x 56 root root    4096 Oct 10 04:57 ../
-rw-r--r--  1 root root 10485760 Oct 10 01:54 00000000000000000000.index
-rw-r--r--  1 root root    188 Oct 10 02:28 00000000000000000000.log
feng@ubuntu:/tmp/kafka/kafka-logs-1/demoTopic2-0$

```

5.2 存储策略

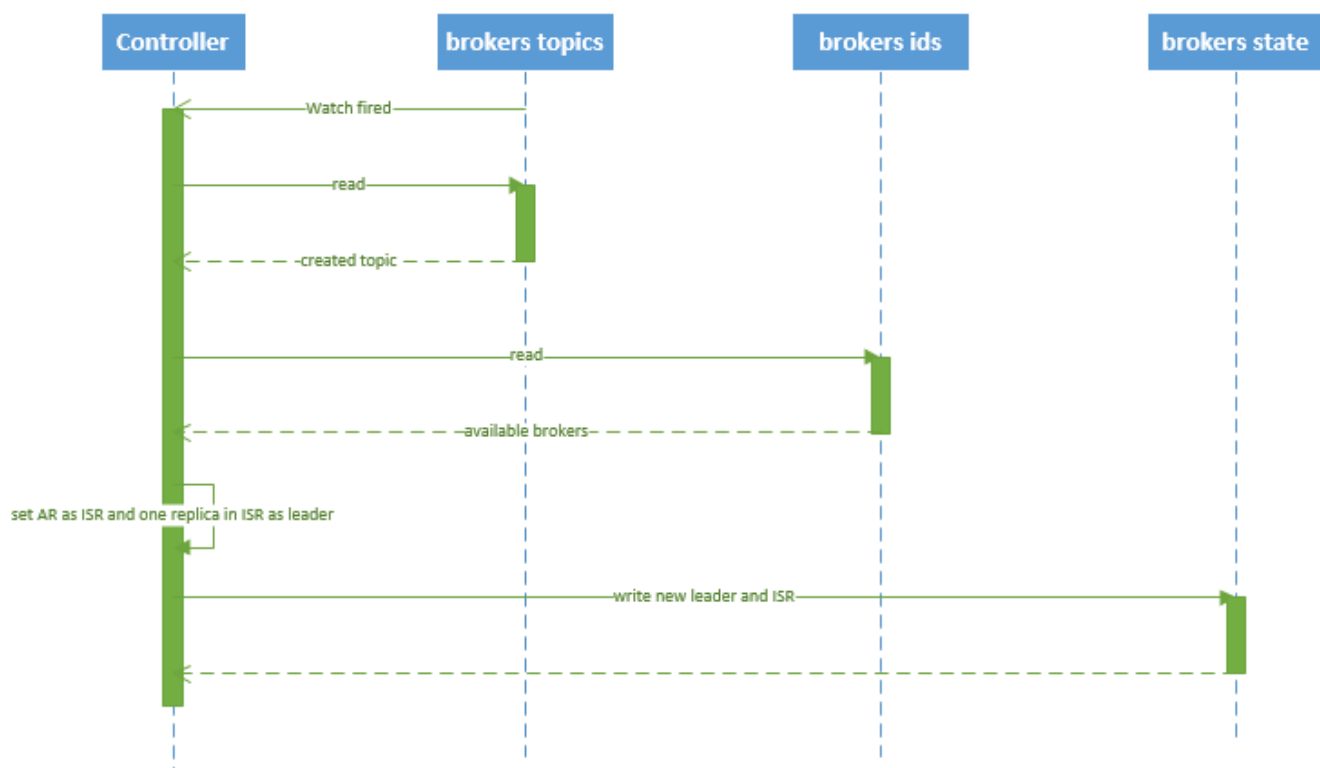
无论消息是否被消费，kafka 都会保留所有消息。有两种策略可以删除旧数据：

- 1、基于时间：log.retention.hours=168
- 2、基于大小：log.retention.bytes=1073741824

六、Topic的创建和删除

6.1 创建topic

创建 topic 的序列图如下所示：

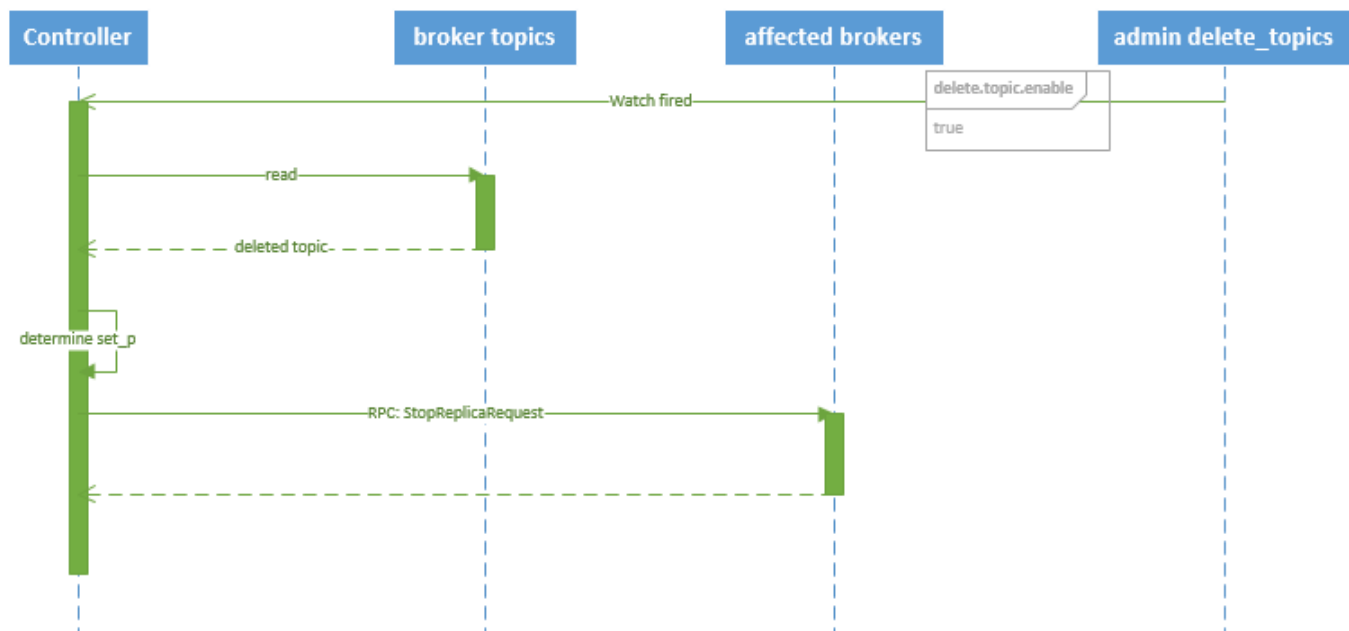


流程说明：

- 1、controller 在 ZooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被创建，则 controller 会通过 watch
- 2、controller 从 /brokers/ids 读取当前所有可用的 broker 列表，对于 set_p 中的每一个 partition：
 - 2.1、从分配给该 partition 的所有 replica（称为 AR）中任选一个可用的 broker 作为新的 leader，并将 AR 设置为新的
 - 2.2、将新的 leader 和 ISR 写入 /brokers/topics/[topic]/partitions/[partition]/state
- 3、controller 通过 RPC 向相关的 broker 发送 LeaderAndISRRequest。

6.2 删除topic

删除 topic 的序列图如下所示：

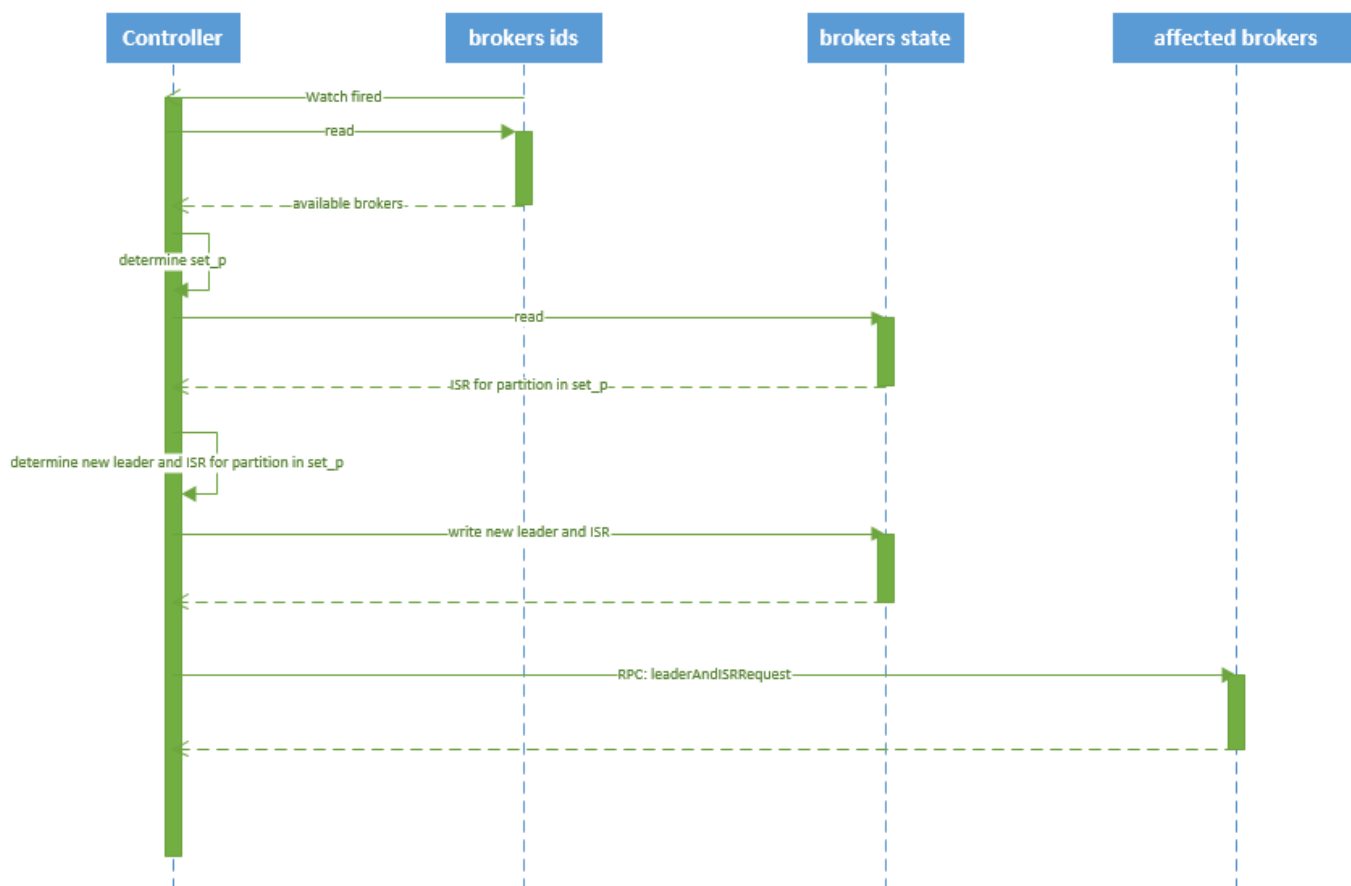


流程说明：

- 1、controller 在 zooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被删除，则 controller 会通过 watch
- 2、若 delete.topic.enable=false，结束；否则 controller 注册在 /admin/delete_topics 上的 watch 被 fire，contro

七、broker failover

kafka broker failover 序列图如下所示：



流程说明：

- 1、controller 在 zookeeper 的 /brokers/ids/[brokerId] 节点注册 Watcher，当 broker 宕机时 zookeeper 会 fire watch
- 2、controller 从 /brokers/ids 节点读取可用broker
- 3、controller决定set_p，该集合包含宕机 broker 上的所有 partition
- 4、对 set_p 中的每一个 partition
 - 4.1、从/brokers/topics/[topic]/partitions/[partition]/state 节点读取 ISR
 - 4.2、决定新 leader
 - 4.3、将新 leader、ISR、controller_epoch 和 leader_epoch 等信息写入 state 节点
- 5、通过 RPC 向相关 broker 发送 leaderAndISRRequest 命令

八、controller failover

当 controller 宕机时会触发 controller failover。每个 broker 都会在 zookeeper 的 "/controller" 节点注册 watcher，当 controller 宕机时 zookeeper 中的临时节点消失，所有存活的 broker 收到 fire 的通知，每个 broker 都尝试创建新的 controller path，只有一个竞选成功并当选为 controller。

当新的 controller 当选时，会触发 KafkaController.onControllerFailover 方法，在该方法中完成如下操作：

- 1、 读取并增加 Controller Epoch。
- 2、 在 reassignedPartitions Patch(/admin/reassign_partitions) 上注册 watcher。
- 3、 在 preferredReplicaElection Path(/admin/preferred_replica_election) 上注册 watcher。
- 4、 通过 partitionStateMachine 在 broker Topics Patch(/brokers/topics) 上注册 watcher。
- 5、 若 delete.topic.enable=true (默认值是 false) , 则 partitionStateMachine 在 Delete Topic Patch(/admin/delete_topics) 上注册 watcher。
- 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=true (默认值是true) , 则启动 partition-rebalance 线程。
- 12、 若 delete.topic.enable=true 且Delete Topic Patch(/admin/delete_topics)中有值, 则删除相应的Topic。

参考

<https://www.cnblogs.com/qingyunzong/p/9004703.html>

kafka的安装

一、下载

下载地址:

<http://kafka.apache.org/downloads.html>

<http://mirrors.hust.edu.cn/apache/>

[回到顶部](#)

二、安装前提 (zookeeper安装)

参考http://www.cnblogs.com/qingyunzong/p/8634335.html#_label4_0

三、安装

此处使用版本为kafka_2.11-0.8.2.0.tgz

2.1 上传解压缩


```
[hadoop@hadoop1 ~]$ tar -zxvf kafka_2.11-0.8.2.0.tgz -C apps
[hadoop@hadoop1 ~]$ cd apps/
[hadoop@hadoop1 apps]$ ln -s kafka_2.11-0.8.2.0/ kafka
```

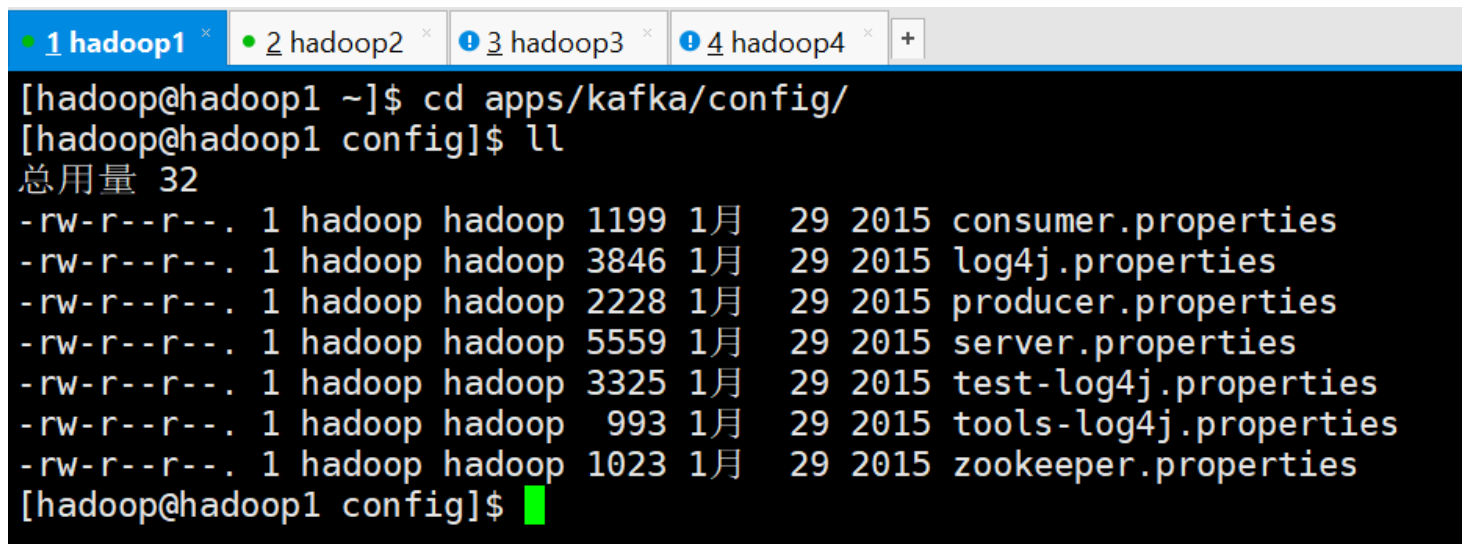
2.2 修改配置文件

进入kafka的安装配置目录

```
[hadoop@hadoop1 ~]$ cd apps/kafka/config/
```

主要关注：**server.properties** 这个文件即可，我们可以发现在目录下：

有很多文件，这里可以发现Zookeeper文件，我们可以根据Kafka内带的zk集群来启动，但是建议使用独立的zk集群



The image shows a terminal window with four tabs labeled '1 hadoop1', '2 hadoop2', '3 hadoop3', and '4 hadoop4'. The active tab is '1 hadoop1'. The terminal output shows the user navigating to the kafka config directory and listing its contents. The output is as follows:

```
[hadoop@hadoop1 ~]$ cd apps/kafka/config/
[hadoop@hadoop1 config]$ ll
总用量 32
-rw-r--r--. 1 hadoop hadoop 1199 1月 29 2015 consumer.properties
-rw-r--r--. 1 hadoop hadoop 3846 1月 29 2015 log4j.properties
-rw-r--r--. 1 hadoop hadoop 2228 1月 29 2015 producer.properties
-rw-r--r--. 1 hadoop hadoop 5559 1月 29 2015 server.properties
-rw-r--r--. 1 hadoop hadoop 3325 1月 29 2015 test-log4j.properties
-rw-r--r--. 1 hadoop hadoop 993 1月 29 2015 tools-log4j.properties
-rw-r--r--. 1 hadoop hadoop 1023 1月 29 2015 zookeeper.properties
[hadoop@hadoop1 config]$
```

server.properties (**broker.id**和**host.name**每个节点都不相同)

```

//当前机器在集群中的唯一标识，和zookeeper的myid性质一样
broker.id=0
//当前kafka对外提供服务的端口默认是9092
port=9092
//这个参数默认是关闭的，在0.8.1有个bug，DNS解析问题，失败率的问题。
host.name=hadoop1
//这个是borker进行网络处理的线程数
num.network.threads=3
//这个是borker进行I/O处理的线程数
num.io.threads=8
//发送缓冲区buffer大小，数据不是一下子就发送的，先回存储到缓冲区了到达一定的大小后在发送，能提高性能
socket.send.buffer.bytes=102400
//kafka接收缓冲区大小，当数据到达一定大小后在序列化到磁盘
socket.receive.buffer.bytes=102400
//这个参数是向kafka请求消息或者向kafka发送消息的请请求的最大数，这个值不能超过java的堆栈大小
socket.request.max.bytes=104857600
//消息存放的目录，这个目录可以配置为“，”逗号分割的表达式，上面的num.io.threads要大于这个目录的个数这个目录，
//如果配置多个目录，新创建的topic他把消息持久化的地方是，当前以逗号分割的目录中，那个分区数最少就放那一个
log.dirs=/home/hadoop/log/kafka-logs
//默认的分区分数，一个topic默认1个分区分数
num.partitions=1
//每个数据目录用来日志恢复的线程数目
num.recovery.threads.per.data.dir=1
//默认消息的最大持久化时间，168小时，7天
log.retention.hours=168
//这个参数是：因为kafka的消息是以追加的形式落地到文件，当超过这个值的时候，kafka会新起一个文件
log.segment.bytes=1073741824
//每隔300000毫秒去检查上面配置的log失效时间
log.retention.check.interval.ms=300000
//是否启用log压缩，一般不用启用，启用的话可以提高性能
log.cleaner.enable=false
//设置zookeeper的连接端口
zookeeper.connect=192.168.123.102:2181,192.168.123.103:2181,192.168.123.104:2181
//设置zookeeper的连接超时时间
zookeeper.connection.timeout.ms=6000

```

producer.properties

```

metadata.broker.list=192.168.123.102:9092,192.168.123.103:9092,192.168.123.104:9092

```

consumer.properties

```

zookeeper.connect=192.168.123.102:2181,192.168.123.103:2181,192.168.123.104:2181

```

2.3 将kafka的安装包分发到其他节点

```
[hadoop@hadoop1 apps]$ scp -r kafka_2.11-0.8.2.0/ hadoop2:$PWD
[hadoop@hadoop1 apps]$ scp -r kafka_2.11-0.8.2.0/ hadoop3:$PWD
[hadoop@hadoop1 apps]$ scp -r kafka_2.11-0.8.2.0/ hadoop4:$PWD
```

2.4 创建软连接

```
[hadoop@hadoop1 apps]$ ln -s kafka_2.11-0.8.2.0/ kafka
```

2.5 修改环境变量

```
[hadoop@hadoop1 ~]$ vi .bashrc
#Kafka
export KAFKA_HOME=/home/hadoop/apps/kafka
export PATH=$PATH:$KAFKA_HOME/bin
```

保存使其立即生效

```
[hadoop@hadoop1 ~]$ source ~/.bashrc
```

三、启动

3.1 首先启动zookeeper集群

所有zookeeper节点都需要执行

```
[hadoop@hadoop1 ~]$ zkServer.sh start
```

3.2 启动Kafka集群服务

```
[hadoop@hadoop1 kafka]$ bin/kafka-server-start.sh config/server.properties
```

hadoop1

```
1 hadoop1 x 2 hadoop2 x 3 hadoop3 x 4 hadoop4 x +
ent)
[2018-05-08 09:10:21,096] INFO Log directory '/home/hadoop/log/kafka-logs' not found, creating it
. (kafka.log.LogManager)
[2018-05-08 09:10:21,147] INFO Loading logs. (kafka.log.LogManager)
[2018-05-08 09:10:21,549] INFO Logs loading complete. (kafka.log.LogManager)
[2018-05-08 09:10:21,550] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogMan
ager)
[2018-05-08 09:10:21,595] INFO Starting log flusher with a default period of 9223372036854775807
ms. (kafka.log.LogManager)
[2018-05-08 09:10:21,998] INFO Awaiting socket connections on hadoop1:9092. (kafka.network.Accept
or)
[2018-05-08 09:10:22,007] INFO [Socket Server on Broker 0], Started (kafka.network.SocketServer)
[2018-05-08 09:10:22,750] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.
utils.Mx4jLoader$)
[2018-05-08 09:10:22,875] INFO 0 successfully elected as leader (kafka.server.ZookeeperLeaderElec
tor)
[2018-05-08 09:10:23,164] INFO Registered broker 0 at path /brokers/ids/0 with address hadoop1:90
92. (kafka.utils.ZkUtils$)
[2018-05-08 09:10:23,234] INFO [Kafka Server 0], started (kafka.server.KafkaServer)
[2018-05-08 09:10:23,314] INFO New leader is 0 (kafka.server.ZookeeperLeaderElector$LeaderChangeL
istener)
```

Hadoop2

```
1 hadoop1 x 2 hadoop2 x 3 hadoop3 x 4 hadoop4 x +
[2018-05-08 09:11:25,204] INFO Socket connection established to 192.168.123.104/192.168.123.104:2
181, initiating session (org.apache.zookeeper.ClientCnxn)
[2018-05-08 09:11:25,222] INFO Session establishment complete on server 192.168.123.104/192.168.1
23.104:2181, sessionId = 0x3633d46e5fa0001, negotiated timeout = 6000 (org.apache.zookeeper.Clien
tCnxn)
[2018-05-08 09:11:25,230] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkCli
ent)
[2018-05-08 09:11:25,293] INFO Loading logs. (kafka.log.LogManager)
[2018-05-08 09:11:25,297] INFO Logs loading complete. (kafka.log.LogManager)
[2018-05-08 09:11:25,299] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogMan
ager)
[2018-05-08 09:11:25,303] INFO Starting log flusher with a default period of 9223372036854775807
ms. (kafka.log.LogManager)
[2018-05-08 09:11:25,355] INFO Awaiting socket connections on hadoop2:9092. (kafka.network.Accept
or)
[2018-05-08 09:11:25,361] INFO [Socket Server on Broker 1], Started (kafka.network.SocketServer)
[2018-05-08 09:11:25,627] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.
utils.Mx4jLoader$)
[2018-05-08 09:11:26,092] INFO Registered broker 1 at path /brokers/ids/1 with address hadoop2:90
92. (kafka.utils.ZkUtils$)
[2018-05-08 09:11:26,210] INFO [Kafka Server 1], started (kafka.server.KafkaServer)
```

hadoop3

```
• 1 hadoop1 × • 2 hadoop2 × • 3 hadoop3 × • 4 hadoop4 × +
[2018-05-08 09:12:03,129] INFO Socket connection established to 192.168.123.104/192.168.123.104:2181, initiating session (org.apache.zookeeper.ClientCnxn)
[2018-05-08 09:12:03,143] INFO Session establishment complete on server 192.168.123.104/192.168.123.104:2181, sessionId = 0x3633d46e5fa0002, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)
[2018-05-08 09:12:03,170] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
[2018-05-08 09:12:03,219] INFO Loading logs. (kafka.log.LogManager)
[2018-05-08 09:12:03,224] INFO Logs loading complete. (kafka.log.LogManager)
[2018-05-08 09:12:03,226] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogManager)
[2018-05-08 09:12:03,230] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogManager)
[2018-05-08 09:12:03,278] INFO Awaiting socket connections on hadoop3:9092. (kafka.network.Acceptor)
[2018-05-08 09:12:03,279] INFO [Socket Server on Broker 2], Started (kafka.network.SocketServer)
[2018-05-08 09:12:03,510] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader$)
[2018-05-08 09:12:03,965] INFO Registered broker 2 at path /brokers/ids/2 with address hadoop3:9092. (kafka.utils.ZkUtils$)
[2018-05-08 09:12:04,025] INFO [Kafka Server 2], started (kafka.server.KafkaServer)
```

hadoop4

```
• 1 hadoop1 × • 2 hadoop2 × • 3 hadoop3 × • 4 hadoop4 × +
[2018-05-08 09:12:24,443] INFO Socket connection established to 192.168.123.104/192.168.123.104:2181, initiating session (org.apache.zookeeper.ClientCnxn)
[2018-05-08 09:12:24,453] INFO Session establishment complete on server 192.168.123.104/192.168.123.104:2181, sessionId = 0x3633d46e5fa0003, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)
[2018-05-08 09:12:24,455] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
[2018-05-08 09:12:24,534] INFO Loading logs. (kafka.log.LogManager)
[2018-05-08 09:12:24,545] INFO Logs loading complete. (kafka.log.LogManager)
[2018-05-08 09:12:24,554] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogManager)
[2018-05-08 09:12:24,556] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogManager)
[2018-05-08 09:12:24,611] INFO Awaiting socket connections on hadoop4:9092. (kafka.network.Acceptor)
[2018-05-08 09:12:24,612] INFO [Socket Server on Broker 3], Started (kafka.network.SocketServer)
[2018-05-08 09:12:24,837] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader$)
[2018-05-08 09:12:25,179] INFO Registered broker 3 at path /brokers/ids/3 with address hadoop4:9092. (kafka.utils.ZkUtils$)
[2018-05-08 09:12:25,249] INFO [Kafka Server 3], started (kafka.server.KafkaServer)
```

3.3 创建的topic

```
[hadoop@hadoop1 kafka]$ bin/kafka-topics.sh --create --zookeeper hadoop1:2181 --replication-factor 3 --partitions
```

```
• 1 hadoop1 × • 2 hadoop2 × • 3 hadoop3 × • 4 hadoop4 × • 5 hadoop1 × +
[2018-05-08 09:25:22,390] INFO Created log for partition [topic2,1] in /home/hadoop/log/kafka-log
s with properties {segment.index.bytes -> 10485760, file.delete.delay.ms -> 60000, segment.bytes
-> 1073741824, flush.ms -> 9223372036854775807, delete.retention.ms -> 86400000, index.interval.b
ytes -> 4096, retention.bytes -> -1, min.insync.replicas -> 1, cleanup.policy -> delete, unclean.
leader.election.enable -> true, segment.ms -> 604800000, max.message.bytes -> 1000012, flush.mess
ages -> 9223372036854775807, min.cleanable.dirty.ratio -> 0.5, retention.ms -> 604800000, segment
.jitter.ms -> 0}. (kafka.log.LogManager)
[2018-05-08 09:25:22,392] WARN Partition [topic2,1] on broker 0: No checkpointed highwatermark is
found for partition [topic2,1] (kafka.cluster.Partition)
[2018-05-08 09:25:22,425] INFO Completed load of log topic2-0 with log end offset 0 (kafka.log.Lo
g)
[2018-05-08 09:25:22,434] INFO Created log for partition [topic2,0] in /home/hadoop/log/kafka-log
s with properties {segment.index.bytes -> 10485760, file.delete.delay.ms -> 60000, segment.bytes
-> 1073741824, flush.ms -> 9223372036854775807, delete.retention.ms -> 86400000, index.interval.b
ytes -> 4096, retention.bytes -> -1, min.insync.replicas -> 1, cleanup.policy -> delete, unclean.
leader.election.enable -> true, segment.ms -> 604800000, max.message.bytes -> 1000012, flush.mess
ages -> 9223372036854775807, min.cleanable.dirty.ratio -> 0.5, retention.ms -> 604800000, segment
.jitter.ms -> 0}. (kafka.log.LogManager)
[2018-05-08 09:25:22,438] WARN Partition [topic2,0] on broker 0: No checkpointed highwatermark is
found for partition [topic2,0] (kafka.cluster.Partition)
[2018-05-08 09:25:22,439] INFO [ReplicaFetcherManager on broker 0] Removed fetcher for partitions
[topic2,0] (kafka.server.ReplicaFetcherManager)
```

3.4 查看topic副本信息

```
[hadoop@hadoop1 kafka]$ bin/kafka-topics.sh --describe --zookeeper hadoop1:2181 --topic topic2
```

```
• 1 hadoop1 × • 2 hadoop2 × • 3 hadoop3 × • 4 hadoop4 × • 5 hadoop1 × +
[hadoop@hadoop1 kafka]$ bin/kafka-topics.sh --describe --zookeeper hadoop1:2181 --topic topic2
Topic:topic2    PartitionCount:3      ReplicationFactor:3      Configs:
    Topic: topic2    Partition: 0      Leader: 3      Replicas: 3,0,1 Isr: 3,0,1
    Topic: topic2    Partition: 1      Leader: 0      Replicas: 0,1,2 Isr: 0,1,2
    Topic: topic2    Partition: 2      Leader: 1      Replicas: 1,2,3 Isr: 1,2,3
[hadoop@hadoop1 kafka]$
```

3.5 查看已经创建的topic信息

```
[hadoop@hadoop1 kafka]$ bin/kafka-topics.sh --list --zookeeper hadoop1:2181
```

```
• 1 hadoop1 × • 2 hadoop2 × • 3 hadoop3 × • 4 hadoop4 × • 5 hadoop1 × +
[hadoop@hadoop1 kafka]$ bin/kafka-topics.sh --list --zookeeper hadoop1:2181
topic1
topic2
[hadoop@hadoop1 kafka]$
```

3.6 生产者发送消息

```
[hadoop@hadoop1 kafka]$ bin/kafka-console-producer.sh --broker-list hadoop1:9092 --topic topic2
```

```
1 hadoop1 x 2 hadoop2 x 3 hadoop3 x 4 hadoop4 x 5 hadoop1 x +
[hadoop@hadoop1 kafka]$ bin/kafka-console-producer.sh --broker-list hadoop1:9092 --topic topic2
[2018-05-08 09:43:52,004] WARN Property topic is not valid (kafka.utils.VerifiableProperties)
hello kafka
```

hadoop1显示接收到消息

3.7 消费者消费消息

在hadoop2上消费消息

```
[hadoop@hadoop2 kafka]$ bin/kafka-console-consumer.sh --zookeeper hadoop1:2181 --from-beginning --topic topic2
```

```
1 hadoop1 x 2 hadoop2 x 3 hadoop3 x 4 hadoop4 x 5 hadoop1 x 6 hadoop2 x +
[hadoop@hadoop2 kafka]$ bin/kafka-console-consumer.sh --zookeeper hadoop1:2181 --from-beginning -
-topic topic2
hello kafka
```

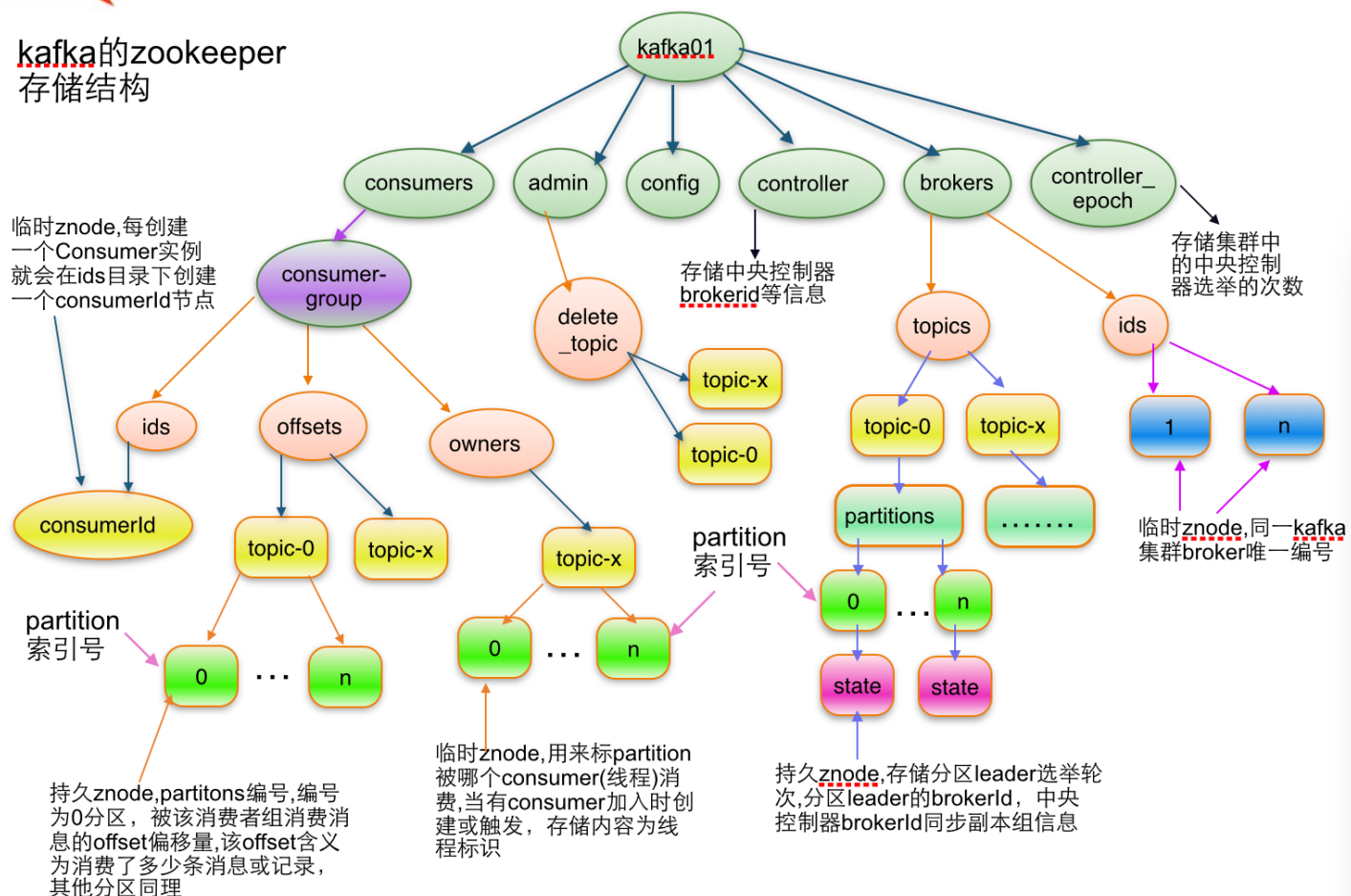
参考

<https://www.cnblogs.com/qingyunzong/p/9005062.html>

Kafka在zookeeper中存储

一、Kafka在zookeeper中存储结构图

kafka的zookeeper存储结构



<http://blog.csdn.net/lizhitao>

二、分析

2.1 topic注册信息

/brokers/topics/[topic]:

存储某个topic的partitions所有分配信息

```
[zk: localhost:2181(CONNECTED) 1] get /brokers/topics/topic2
```



```
[zk: localhost:2181(CONNECTED) 1] get /brokers/topics/topic2
{"version":1,"partitions":{"2":[1,2,3],"1":[0,1,2],"0":[3,0,1]}}
cZxid = 0x220000003d
ctime = Tue May 08 09:25:22 CST 2018
mZxid = 0x220000003d
mtime = Tue May 08 09:25:22 CST 2018
pZxid = 0x2200000041
cversion = 1
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 64
numChildren = 1
[zk: localhost:2181(CONNECTED) 2] █
```

Schema:

```
{
  "version": "版本编号目前固定为数字1",
  "partitions": {
    "partitionId编号": [
      同步副本组brokerId列表
    ],
    "partitionId编号": [
      同步副本组brokerId列表
    ],
    .....
  }
}
```

Example:

```
{
  "version": 1,
  "partitions": {
    "2": [1, 2, 3],
    "1": [0, 1, 2],
    "0": [3, 0, 1],
  }
}
```

2.2 partition状态信息

/brokers/topics/[topic]/partitions/[0...N] 其中[0..N]表示partition索引号

/brokers/topics/[topic]/partitions/[partitionId]/state

```
[zk: localhost:2181(CONNECTED) 2] ls /brokers/topics/topic2/partitions
[0, 1, 2]
[zk: localhost:2181(CONNECTED) 3] get /brokers/topics/topic2/partitions/0/state
{"controller_epoch":1,"leader":3,"version":1,"leader_epoch":0,"isr":[3,0,1]}
cZxid = 0x22000000046
ctime = Tue May 08 09:25:22 CST 2018
mZxid = 0x22000000046
mtime = Tue May 08 09:25:22 CST 2018
pZxid = 0x22000000046
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 76
numChildren = 0
[zk: localhost:2181(CONNECTED) 4] █
```

Schema:

```
{
  "controller_epoch": 表示kafka集群中的中央控制器选举次数,
  "leader": 表示该partition选举leader的brokerId,
  "version": 版本编号默认为1,
  "leader_epoch": 该partition leader选举次数,
  "isr": [同步副本组brokerId列表]
}
```

Example:

```
{
  "controller_epoch": 1,
  "leader": 3,
  "version": 1,
  "leader_epoch": 0,
  "isr": [3, 0, 1]
}
```

2.3 Broker注册信息

/brokers/ids/[0...N]

每个broker的配置文件都需要指定一个数字类型的id(全局不可重复),此节点为临时znode(EPHEMERAL)

```
[zk: localhost:2181(CONNECTED) 4] ls /brokers/ids
[0, 1, 2, 3]
[zk: localhost:2181(CONNECTED) 5] get /brokers/ids/0
{"jmx_port":-1,"timestamp":"1525741823119","host":"hadoop1","version":1,"port":9092}
cZxid = 0x2200000019
ctime = Tue May 08 09:10:23 CST 2018
mZxid = 0x2200000019
mtime = Tue May 08 09:10:23 CST 2018
pZxid = 0x2200000019
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x2633d46e5a20000
dataLength = 84
numChildren = 0
[zk: localhost:2181(CONNECTED) 6] █
```

Schema:

```
{
  "jmx_port": jmx端口号,
  "timestamp": kafka broker初始启动时的时间戳,
  "host": 主机名或ip地址,
  "version": 版本号默认为1,
  "port": kafka broker的服务端口号,由server.properties中参数port确定
}
```

Example:

```
{
  "jmx_port": -1,
  "timestamp": "1525741823119"
  "version": 1,
  "host": "hadoop1",
  "port": 9092
}
```

2.4 Controller epoch

/controller_epoch --> int (epoch)

此值为一个数字,kafka集群中第一个broker第一次启动时为1,以后只要集群中center controller中央控制器所在broker变更或挂掉,就会重新选举新的center controller,每次center controller变更controller_epoch值就会 + 1;

```
[zk: localhost:2181(CONNECTED) 6] get /controller_epoch
1
cZxid = 0x2200000017
ctime = Tue May 08 09:10:22 CST 2018
mZxid = 0x2200000017
mtime = Tue May 08 09:10:22 CST 2018
pZxid = 0x2200000017
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 1
numChildren = 0
[zk: localhost:2181(CONNECTED) 7] █
```

2.5 Controller注册信息

/controller -> int (broker id of the controller) 存储center controller中央控制器所在kafka broker的信息

```
[zk: localhost:2181(CONNECTED) 7] get /controller
{"version":1,"brokerid":0,"timestamp":"1525741822769"}
cZxid = 0x2200000015
ctime = Tue May 08 09:10:22 CST 2018
mZxid = 0x2200000015
mtime = Tue May 08 09:10:22 CST 2018
pZxid = 0x2200000015
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x2633d46e5a20000
dataLength = 54
numChildren = 0
[zk: localhost:2181(CONNECTED) 8] █
```

Schema:

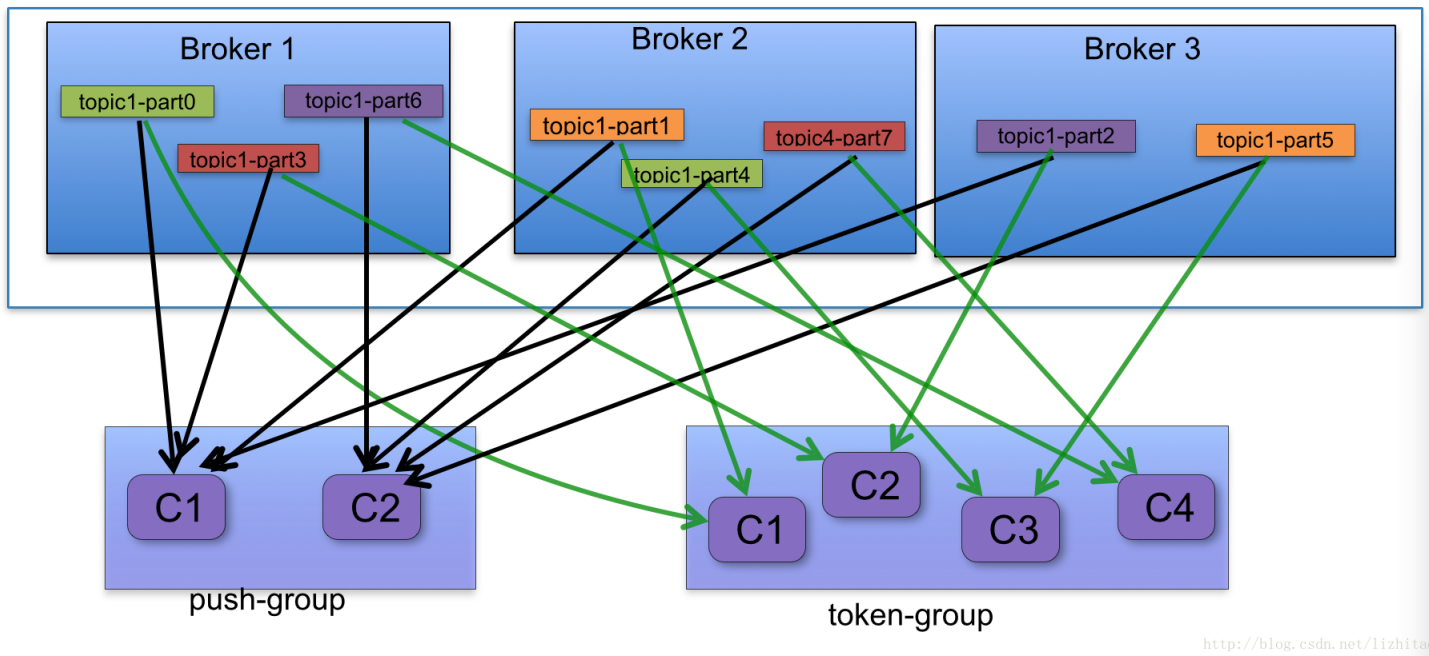
```
{  
  "version": 版本号默认为1,  
  "brokerid": kafka集群中broker唯一编号,  
  "timestamp": kafka broker中央控制器变更时的时间戳  
}
```

Example:

```
{  
  "version": 1,  
  "brokerid": 0,  
  "timestamp": "1525741822769"  
}
```

2.6 补充Consumer and Consumer group

kafka cluster



<http://blog.csdn.net/lizhitao>

- a.每个consumer客户端被创建时,会向zookeeper注册自己的信息;
- b.此作用主要是为了"负载均衡".
- c.同一个Consumer Group中的Consumers, Kafka将相应Topic中的每个消息只发送给其中一个Consumer.
- d.Consumer Group中的每个Consumer读取Topic的一个或多个Partitions, 并且是唯一的Consumer;
- e.一个Consumer group的多个consumer的所有线程依次有序地消费一个topic的所有partitions,如果Consumer group中所有consumer总线程大于partitions数量, 则会出现空闲情况;

举例说明:

kafka集群中创建一个topic为report-log 4 partitions 索引编号为0,1,2,3

假如有目前有三个消费者node：注意-->一个consumer中一个消费线程可以消费一个或多个partition.

如果每个consumer创建一个consumer thread线程,各个node消费情况如下, node1消费索引编号为0,1分区, node2消费索引编号为2,node3消费索引编号为3

如果每个consumer创建2个consumer thread线程, 各个node消费情况如下(是从consumer node先后启动状态来确定的), node1消费索引编号为0,1分区; node2消费索引编号为2,3; node3为空闲状态

总结:

从以上可知, Consumer Group中各个consumer是根据先后启动的顺序有序消费一个topic的所有partitions的。

如果Consumer Group中所有consumer的总线程数大于partitions数量, 则可能consumer thread或consumer会出现空闲状态。

2.7 Consumer均衡算法

当一个group中,有consumer加入或者离开时,会触发partitions均衡.均衡的最终目的,是提升topic的并发消费能力.

- 1) 假如topic1,具有如下partitions: P0,P1,P2,P3
- 2) 加入group中,有如下consumer: C0,C1
- 3) 首先根据partition索引号对partitions排序: P0,P1,P2,P3
- 4) 根据(consumer.id + '-' + thread序号)排序: C0,C1
- 5) 计算倍数: $M = [P0,P1,P2,P3].size / [C0,C1].size$,本例值 $M=2$ (向上取整)
- 6) 然后依次分配partitions: $C0 = [P0,P1], C1=[P2,P3]$,即 $Ci = [P(i * M), P((i + 1) * M - 1)]$

2.8 Consumer注册信息

每个consumer都有一个唯一的ID(consumerId可以通过配置文件指定,也可以由系统生成),此id用来标记消费者信息.

/consumers/[groupId]/ids/[consumerIdString]

是一个临时的znode,此节点的值请看consumerIdString产生规则,即表示此consumer目前所消费的topic + partitions列表.

consumerId产生规则:

```

String consumerUuid = null;
if(config.consumerId!=null && config.consumerId)
    consumerUuid = consumerId;
else {
    String uuid = UUID.randomUUID();
    consumerUuid = "%s-%d-%s".format(
        InetAddress.getLocalHost().getHostName(), System.currentTimeMillis(),
        uuid.getMostSignificantBits().toHexString().substring(0,8));
}
String consumerIdString = config.groupId + "_" + consumerUuid;

```

```
[zk: localhost:2181(CONNECTED) 11] get /consumers/console-consumer-2304/ids/console-consumer-2304_hadoop2-1525747
```

```

[zk: localhost:2181(CONNECTED) 9] ls /consumers
[console-consumer-2304]
[zk: localhost:2181(CONNECTED) 10] ls /consumers/console-consumer-2304/ids
[console-consumer-2304_hadoop2-1525747915241-6b48ff32]
[zk: localhost:2181(CONNECTED) 11] get /consumers/console-consumer-2304/ids/console-consumer-2304_hadoop2-1525747915241-6b48ff32
{"version":1,"subscription":{"topic2":1},"pattern":"white_list","timestamp":"1525747915336"}
cZxid = 0x22000000080
ctime = Tue May 08 10:51:55 CST 2018
mZxid = 0x22000000080
mtime = Tue May 08 10:51:55 CST 2018
pZxid = 0x22000000080
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x1633d470e10000e
dataLength = 92
numChildren = 0
[zk: localhost:2181(CONNECTED) 12] █

```

```
Schema:
{
  "version": 版本编号默认为1,
  "subscription": { //订阅topic列表
    "topic名称": consumer中topic消费者线程数
  },
  "pattern": "static",
  "timestamp": "consumer启动时的时间戳"
}
```

```
Example:
{
  "version": 1,
  "subscription": {
    "topic2": 1
  },
  "pattern": "white_list",
  "timestamp": "1525747915336"
}
```

2.9 Consumer owner

/consumers/[groupId]/owners/[topic]/[partitionId] -> consumerIdString + threadId索引编号

a) 首先进行"Consumer Id注册";

b) 然后在"Consumer id 注册"节点下注册一个watch用来监听当前group中其他consumer的"退出"和"加入";只要此znode path下节点列表变更,都会触发此group下consumer的负载均衡.(比如一个consumer失效,那么其他consumer接管partitions).

c) 在"Broker id 注册"节点下,注册一个watch用来监听broker的存活情况;如果broker列表变更,将会触发所有的groups下的consumer重新balance.


```

[zk: localhost:2181(CONNECTED) 12] ls /consumers
[console-consumer-2304]
[zk: localhost:2181(CONNECTED) 13] ls /consumers/console-consumer-2304/owners
[topic2]
[zk: localhost:2181(CONNECTED) 14] ls /consumers/console-consumer-2304/owners/topic2
[0, 1, 2]
[zk: localhost:2181(CONNECTED) 15] get /consumers/console-consumer-2304/owners/topic2/0
console-consumer-2304_hadoop2-1525747915241-6b48ff32-0
cZxid = 0x22000000086
ctime = Tue May 08 10:51:55 CST 2018
mZxid = 0x22000000086
mtime = Tue May 08 10:51:55 CST 2018
pZxid = 0x22000000086
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x1633d470e10000e
dataLength = 54
numChildren = 0
[zk: localhost:2181(CONNECTED) 16] █

```

2.10 Consumer offset

/consumers/[groupId]/offsets/[topic]/[partitionId] -> long (offset)

用来跟踪每个consumer目前所消费的partition中最大的offset

此znode为持久节点,可以看出offset跟group_id有关,以表明当消费者组(consumer group)中一个消费者失效,

重新触发balance,其他consumer可以继续消费.

```

[zk: localhost:2181(CONNECTED) 16] get /consumers/console-consumer-2304/offsets/topic2/0
0
cZxid = 0x2200000008c
ctime = Tue May 08 10:52:55 CST 2018
mZxid = 0x2200000008c
mtime = Tue May 08 10:52:55 CST 2018
pZxid = 0x2200000008c
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 1
numChildren = 0
[zk: localhost:2181(CONNECTED) 17] █

```

2.11 Re-assign partitions

/admin/reassign_partitions

```

{
  "fields":[
    {
      "name":"version",
      "type":"int",
      "doc":"version id"
    },
    {
      "name":"partitions",
      "type":{
        "type":"array",
        "items":{
          "fields":[
            {
              "name":"topic",
              "type":"string",
              "doc":"topic of the partition to be reassigned"
            },
            {
              "name":"partition",
              "type":"int",
              "doc":"the partition to be reassigned"
            },
            {
              "name":"replicas",
              "type":"array",
              "items":"int",
              "doc":"a list of replica ids"
            }
          ],
          "doc":"an array of partitions to be reassigned to new replicas"
        }
      }
    }
  ]
}

```

Example:

```

{
  "version": 1,
  "partitions":
  [
    {
      "topic": "Foo",
      "partition": 1,
      "replicas": [0, 1, 3]
    }
  ]
}

```

2.12 Preferred replication election

/admin/preferred_replica_election

```

{
  "fields":[
    {
      "name":"version",
      "type":"int",
      "doc":"version id"
    },
    {
      "name":"partitions",
      "type":{
        "type":"array",
        "items":{
          "fields":[
            {
              "name":"topic",
              "type":"string",
              "doc":"topic of the partition for which preferred replica election should be triggered"
            },
            {
              "name":"partition",
              "type":"int",
              "doc":"the partition for which preferred replica election should be triggered"
            }
          ],
          "doc":"an array of partitions for which preferred replica election should be triggered"
        }
      }
    }
  ]
}

```

例子:

```

{
  "version": 1,
  "partitions":
  [
    {
      "topic": "Foo",
      "partition": 1
    },
    {
      "topic": "Bar",
      "partition": 0
    }
  ]
}

```

2.13 删除topics

/admin/delete_topics

```
[zk: localhost:2181(CONNECTED) 19] get /admin/delete_topics
null
cZxid = 0x220000000e
ctime = Tue May 08 09:10:16 CST 2018
mZxid = 0x220000000e
mtime = Tue May 08 09:10:16 CST 2018
pZxid = 0x220000000e
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 0
[zk: localhost:2181(CONNECTED) 20] █
```

Schema:

```
{ "fields":
  [ {"name": "version", "type": "int", "doc": "version id"},
    {"name": "topics",
      "type": { "type": "array", "items": "string", "doc": "an array of topics to be deleted"}
    } ]
}
```

例子:

```
{
  "version": 1,
  "topics": ["foo", "bar"]
}
```

2.4 Topic配置

/config/topics/[topic_name]

```
[zk: localhost:2181(CONNECTED) 20] get /config/topics/topic2
{"version":1,"config":{}}
cZxid = 0x220000003c
ctime = Tue May 08 09:25:22 CST 2018
mZxid = 0x220000003c
mtime = Tue May 08 09:25:22 CST 2018
pZxid = 0x220000003c
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 25
numChildren = 0
[zk: localhost:2181(CONNECTED) 21] █
```

参考

<https://www.cnblogs.com/qingyunzong/p/9007107.html>