Kafka设计解析：Kafka High Availability

《Kafka剖析：Kafka背景及架构介绍》
《Kafka设计解析：Kafka High Availability（上）》
《Kafka设计解析：Kafka High Availability（下）》
《Kafka设计解析：Replication工具》
《Kafka设计解析：Kafka Consumer解析》

Kafka在0.8以前的版本中，并不提供High Availablity机制，一旦一个或多个Broker宕机，则宕机期间其上所有Partition都无法继续提供服务。若该Broker永远不能再恢复，亦或磁盘故障，则其上数据将丢失。而Kafka的设计目标之一即是提供数据持久化，同时对于分布式系统来说，尤其当集群规模上升到一定程度后，一台或者多台机器宕机的可能性大大提高，对Failover要求非常高。因此，Kafka从0.8开始提供High Availability机制。本文从Data Replication和Leader Election两方面介绍了Kafka的HA机制。

Kafka为何需要High Available

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。

1、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。
2、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。

为何需要Leader Election

注意：本文所述Leader Election主要指Replica之间的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×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。

Kafka HA设计解析

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 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

Data Replication

Kafka的Data Replication需要解决如下问题：
1、怎样Propagate消息
2、在向Producer发送ACK前需要保证有多少个Replica已经收到该消息
3、怎样处理某个Replica不工作的情况
4、怎样处理Failed Replica恢复回来的情况

Propagate消息

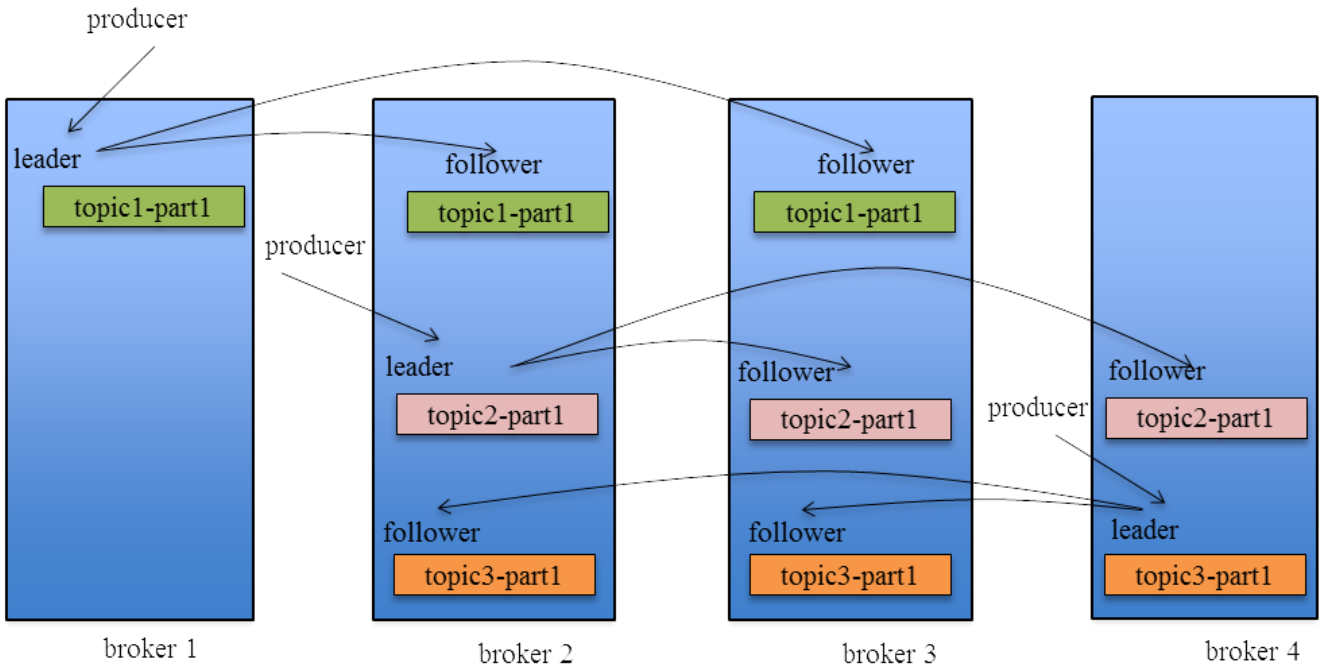
Producer在发布消息到某个Partition时，先通过ZooKeeper找到该Partition的Leader，然后无论该Topic的Replication Factor为多少（也即该Partition有多少个Replica），Producer只将该消息发送到该Partition的Leader。Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。Follower在收到该消息并写入其Log后，向Leader发送ACK。一旦Leader收到了ISR中的所有Replica的ACK，该消息就被认为已经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 Replication的数据流如下图所示：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

ACK前需要保证有多少个备份

和大部分分布式系统一样，Kafka处理失败需要明确定义一个Broker是否“活着”。对于Kafka而言，Kafka存活包含两个条件，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。

Leader Election算法

上文说明了Kafka是如何做Replication的，另外一个很重要的问题是当Leader宕机了，怎样在Follower中选举出新的Leader。因为Follower可能落后许多或者crash了，所以必须确保选择“最新”的Follower作为新的Leader。一个基本的原则就是，如果Leader不在了，新的Leader必须拥有原来的Leader commit过的所有消息。这就需要作一个折衷，如果Leader在标明一条消息被commit前等待更多的Follower确认，那在它宕机之后就有更多的Follower可以作为新的Leader，但这也会造成吞吐率的下降。

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中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。

实际上，Leader Election算法非常多，比如ZooKeeper的Zab, Raft和Viewstamped Replication。而Kafka所使用的Leader Election算法更像微软的PacificA算法。

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。

如何处理所有Replica都不工作

上文提到，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。

如何选举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

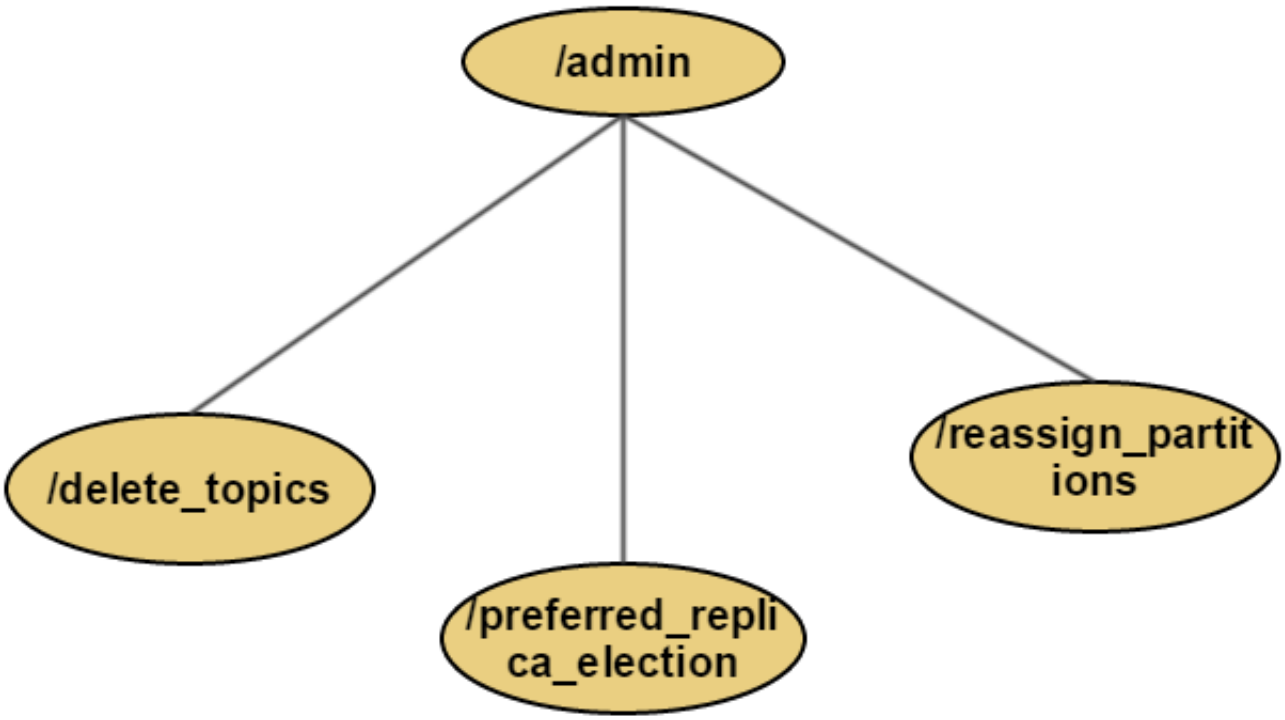
3、ZooKeeper负载过重 每个Replica都要为此在ZooKeeper上注册一个Watch，当集群规模增加到几千个Partition时ZooKeeper负载会过重。

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。

HA相关ZooKeeper结构

首先声明本节所示ZooKeeper结构中，实线框代表路径名是固定的，而虚线框代表路径名与业务相关

admin（该目录下znode只有在有相关操作时才会存在，操作结束时会将其删除）



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

/admin/preferred_replica_election数据结构

```
{
  "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 triggere
d"
            },
            {
```

```
                "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"
        }
      }
  ]
}
```

Example:

```
{
  "version": 1,
  "partitions":
    [
      {
        "topic": "topic1",
        "partition": 8
      },
      {
        "topic": "topic2",
        "partition": 16
      }
    ]
}
```

/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。其数据结构如下：

```
{
  "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]
      }
    ]
}
```

/admin/delete_topics数据结构：

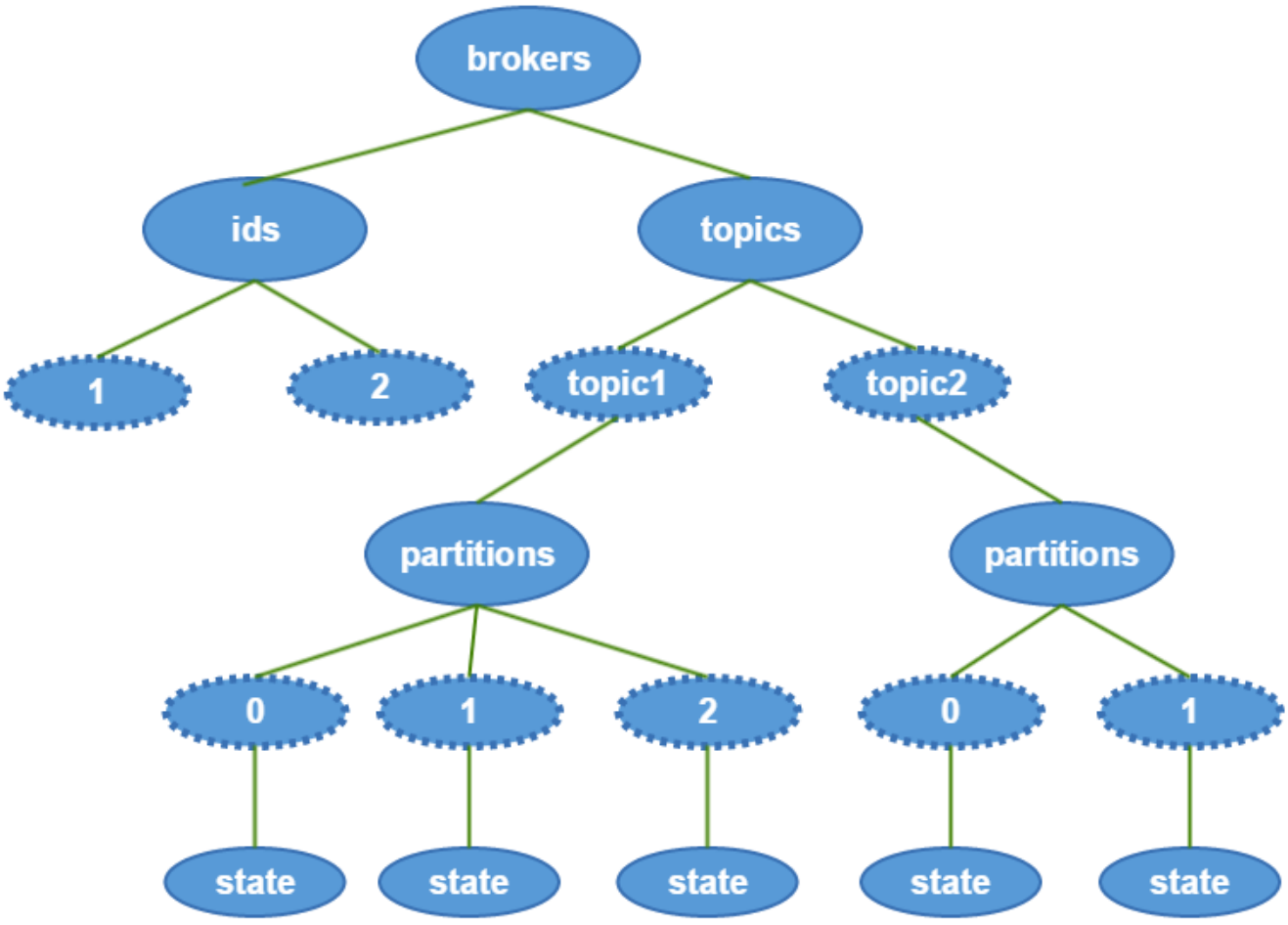
Schema:

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

Example:

```
{
  "version": 1,
  "topics": ["topic4", "topic5"]
}
```

brokers



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

broker（即/brokers/ids/[brokerId]）存储“活着”的broker信息。数据结构如下：

```
Schema:
{ "fields":
    [ {"name": "version", "type": "int", "doc": "version id"},
      {"name": "host", "type": "string", "doc": "ip address or host name of the broker"},
      {"name": "port", "type": "int", "doc": "port of the broker"},
      {"name": "jmx_port", "type": "int", "doc": "port for jmx"}
    ]
}

Example:
{
    "jmx_port":-1,
    "host":"node1",
    "version":1,
    "port":9092
}
```

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。数据结构如下：

```
Schema:
{ "fields" :
    [ {"name": "version", "type": "int", "doc": "version id"},
      {"name": "partitions",
       "type": {"type": "map",
                "values": {"type": "array", "items": "int", "doc": "a list of replica ids"},
                "doc": "a map from partition id to replica list"},
      }
    ]
}
Example:
{
    "version":1,
    "partitions":
       {"12":[6],
        "8":[2],
        "4":[6],
        "11":[5],
        "9":[3],
```

```
        "5":[7],
        "10":[4],
        "6":[8],
        "1":[3],
        "0":[2],
        "2":[4],
        "7":[1],
        "3":[5]}
}
```

partition state（/brokers/topics/[topic]/partitions/[partitionId]/state）结构如下：

```
Schema:
{ "fields":
   [ {"name": "version", "type": "int", "doc": "version id"},
     {"name": "isr",
      "type": {"type": "array",
               "items": "int",
               "doc": "an array of the id of replicas in isr"}
     },
     {"name": "leader", "type": "int", "doc": "id of the leader replica"},
     {"name": "controller_epoch", "type": "int", "doc": "epoch of the controller that last updated
the leader and isr info"},
     {"name": "leader_epoch", "type": "int", "doc": "epoch of the leader"}
   ]
}

Example:
{
   "controller_epoch":29,
   "leader":2,
   "version":1,
   "leader_epoch":48,
   "isr":[2]
}
```

controller

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

Schema:

```
{ "fields":
   [ {"name": "version", "type": "int", "doc": "version id"},
     {"name": "brokerid", "type": "int", "doc": "broker id of the controller"}
   ]
}
```

Example:

```
{
   "version":1,
      "brokerid":8
}
```

/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。