

24 | Kafka的协调服务ZooKeeper：实现分布式系统的“瑞士军刀”

李玥 · 消息队列高手课



你好，我是李玥。

上节课我带你一起学习了 RocketMQ NameServer 的源代码，RocketMQ 的 NameServer 虽然设计非常简洁，但很好地解决了路由寻址的问题。

而 Kafka 却采用了完全不同的设计思路，它选择使用 ZooKeeper 这样一个分布式协调服务来实现和 RocketMQ 的 NameServer 差不多的功能。

这节课我先带大家简单了解一下 ZooKeeper，然后再来一起学习一下 Kafka 是如何借助 ZooKeeper 来构建集群，实现路由寻址的。

ZooKeeper 的作用是什么？

Apache ZooKeeper 它是一个非常特殊的中间件，为什么这么说呢？一般来说，像中间件类的开源产品，大多遵循“做一件事，并做好它。”这样的 UNIX 哲学，每个软件都专注于一种功

能上。而 ZooKeeper 更像是一个“瑞士军刀”，它提供了很多基本的操作，能实现什么样的功能更多取决于使用者如何来使用它。

ZooKeeper 作为一个分布式的协调服务框架，主要用来解决分布式集群中，应用系统需要面对的各种通用的一致性问题。ZooKeeper 本身可以部署为一个集群，集群的各个节点之间可以通过选举来产生一个 Leader，选举遵循半数以上的原则，所以一般集群需要部署奇数个节点。

ZooKeeper 最核心的功能是，它提供了一个分布式的存储系统，数据的组织方式类似于 UNIX 文件系统的树形结构。由于这是一个可以保证一致性的存储系统，所以你可以放心地在你的应用集群中读写 ZooKeeper 的数据，而不用担心数据一致性的问题。分布式系统中一些需要整个集群所有节点都访问的元数据，比如集群节点信息、公共配置信息等，特别适合保存在 ZooKeeper 中。

在这个树形的存储结构中，每个节点被称为一个“ZNode”。ZooKeeper 提供了一种特殊的 ZNode 类型：临时节点。这种临时节点有一个特性：如果创建临时节点的客户端与 ZooKeeper 集群失去连接，这个临时节点就会自动消失。在 ZooKeeper 内部，它维护了 ZooKeeper 集群与所有客户端的心跳，通过判断心跳的状态，来确定是否需要删除客户端创建的临时节点。

ZooKeeper 还提供了一种订阅 ZNode 状态变化的通知机制：Watcher，一旦 ZNode 或者它的子节点状态发生了变化，订阅的客户端会立即收到通知。

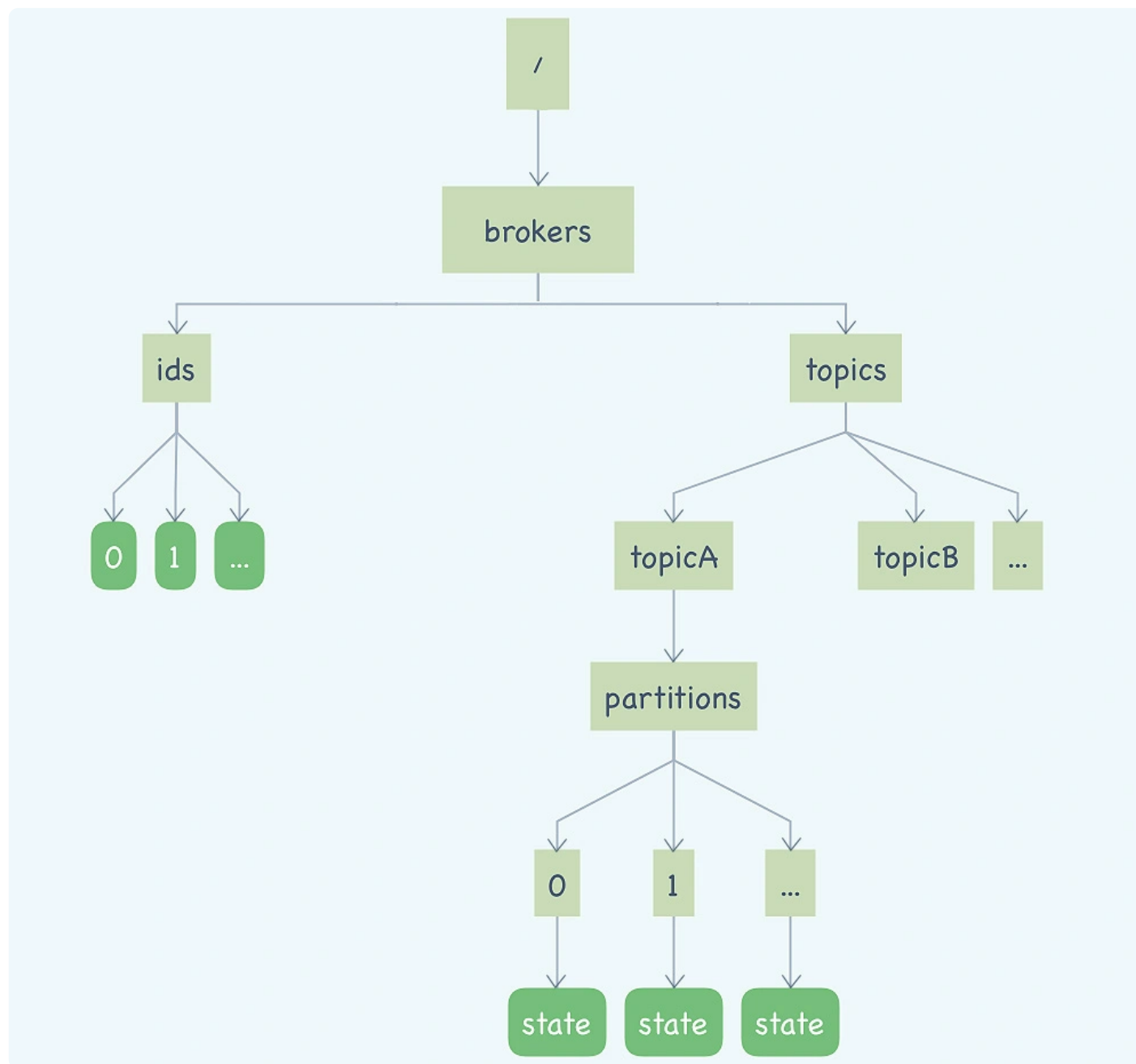
利用 ZooKeeper 临时节点和 Watcher 机制，我们很容易随时来获取业务集群中每个节点的存活状态，并且可以监控业务集群的节点变化情况，当有节点上下线时，都可以收到来自 ZooKeeper 的通知。

此外，我们还可以用 ZooKeeper 来实现业务集群的快速选举、节点间的简单通信、分布式锁等很多功能。

下面我带你一起来看一下 Kafka 是如何来使用 ZooKeeper 的。

Kafka 在 ZooKeeper 中保存了哪些信息？

首先我们来看一下 Kafka 在 ZooKeeper 都保存了哪些信息，我把这些 ZNode 整理了一张图方便你来学习。



你可能在网上看到过和这个图类似的其他版本的图，这些图中绘制的 ZNode 比我们这张图要多一些，这些图大都是描述的 0.8.x 的旧版本的情况，最新版本的 Kafka 已经将消费位置管理等一些原本依赖 ZooKeeper 实现的功能，替换成了其他的实现方式。

图中圆角的矩形是临时节点，直角矩形是持久化的节点。

我们从左往右来看，左侧这棵树保存的是 Kafka 的 Broker 信息，`/brokers/ids/[0...N]`，每个临时节点对应着一个在线的 Broker，Broker 启动后会创建一个临时节点，代表 Broker 已经加入集群可以提供服务了，节点名称就是 BrokerID，节点内保存了包括 Broker 的地址、版本号、启动时间等等一些 Broker 的基本信息。如果 Broker 宕机或者与 ZooKeeper 集群失联了，这个临时节点也会随之消失。

右侧部分的这棵树保存的就是主题和分区的信息。`/brokers/topics/` 节点下面的每个子节点都是一个主题，节点的名称就是主题名称。每个主题节点下面都包含一个固定的 partitions 节点，partitions 节点的子节点就是主题下的所有分区，节点名称就是分区编号。

每个分区节点下面是一个名为 state 的临时节点，节点中保存着分区当前的 leader 和所有的 ISR 的 BrokerID。这个 state 临时节点是由这个分区当前的 Leader Broker 创建的。如果这个分区的 Leader Broker 宕机了，对应的这个 state 临时节点也会消失，直到新的 Leader 被选举出来，再次创建 state 临时节点。

Kafka 客户端如何找到对应的 Broker?

那 Kafka 客户端如何找到主题、队列对应的 Broker 呢？其实，通过上面 ZooKeeper 中的数据结构，你应该已经可以猜的八九不离十了。是的，先根据主题和队列，在右边的树中找到分区对应的 state 临时节点，我们刚刚说过，state 节点中保存了这个分区 Leader 的 BrokerID。拿到这个 Leader 的 BrokerID 后，再去左侧的树中，找到 BrokerID 对应的临时节点，就可以获取到 Broker 真正的访问地址了。

在《[21 | Kafka Consumer 源码分析：消息消费的实现过程](#)》这一节课中，我讲过，Kafka 的客户端并不会去直接连接 ZooKeeper，它只会和 Broker 进行远程通信，那我们可以合理推测一下，ZooKeeper 上的元数据应该是通过 Broker 中转给每个客户端的。

下面我们一起看一下 Kafka 的源代码，来验证一下我们的猜测是不是正确的。

在之前的课程中，我和大家讲过，客户端真正与服务端发生网络传输是在 `org.apache.kafka.clients.NetworkClient#poll` 方法中实现的，我们一直跟踪这个调用链：



```
1 NetworkClient#poll() -> DefaultMetadataUpdater#maybeUpdate(long) -> DefaultMetadata
```

[复制代码](#)

直到 `maybeUpdate(long, Node)` 这个方法，在这个方法里面，Kafka 构造了一个更新元数据的请求：

[复制代码](#)

```
1 private long maybeUpdate(long now, Node node) {
2     String nodeConnectionId = node.idString();
3
4     if (canSendRequest(nodeConnectionId, now)) {
5         // 构建一个更新元数据的请求的构造器
6         Metadata.MetadataRequestAndVersion metadataRequestAndVersion = metadata.n
7         inProgressRequestVersion = metadataRequestAndVersion.requestVersion;
8         MetadataRequest.Builder metadataRequest = metadataRequestAndVersion.reque
9         log.debug("Sending metadata request {} to node {}", metadataRequest, node
10        // 发送更新元数据的请求
11        sendInternalMetadataRequest(metadataRequest, nodeConnectionId, now);
12        return defaultRequestTimeoutMs;
13    }
14
15    //...
16 }
```

这段代码先构造了更新元数据的请求的构造器，然后调用 `sendInternalMetadataRequest()` 把这个请求放到待发送的队列中。这里面有两个地方我需要特别说明一下。

第一点是，在这个方法里面创建的并不是一个真正的更新元数据的 `MetadataRequest`，而是一个用于构造 `MetadataRequest` 的构造器 `MetadataRequest.Builder`，等到真正要发送请求之前，Kafka 才会调用 `Builder.buid()` 方法把这个 `MetadataRequest` 构建出来然后发送出去。而且，不仅是元数据的请求，所有的请求都是这样来处理的。

第二点是，调用 `sendInternalMetadataRequest()` 方法时，这个请求也并没有被真正发出去，依然是保存在待发送的队列中，然后择机来异步批量发送。


请求的具体内容封装在 `org.apache.kafka.common.requests.MetadataRequest` 这个对象中，它包含的信息很简单，只有一个主题的列表，来表明需要获取哪些主题的元数据，另外还

有一个布尔类型的字段 `allowAutoTopicCreation`，表示是否允许自动创建主题。

然后我们再来看下，在 Broker 中，Kafka 是怎么来处理这个更新元数据的请求的。

Broker 处理所有 RPC 请求的入口类在 `kafka.server.KafkaApis#handle` 这个方法里面，我们找到对应处理更新元数据的方法

`handleTopicMetadataRequest(RequestChannel.Request)`，这段代码是用 Scala 语言编写的：

 复制代码

```
1  def handleTopicMetadataRequest(request: RequestChannel.Request) {
2      val metadataRequest = request.body[MetadataRequest]
3      val requestVersion = request.header.apiVersion
4
5      // 计算需要获取哪些主题的元数据
6      val topics =
7          // 在旧版本的协议中，每次都获取所有主题的元数据
8          if (requestVersion == 0) {
9              if (metadataRequest.topics() == null || metadataRequest.topics.isEmpty)
10                 metadataCache.getAllTopics()
11             else
12                 metadataRequest.topics.asScala.toSet
13         } else {
14             if (metadataRequest.isAllTopics)
15                 metadataCache.getAllTopics()
16             else
17                 metadataRequest.topics.asScala.toSet
18         }
19
20     // 省略掉鉴权相关代码
21     // ...
22
23     val topicMetadata =
24         if (authorizedTopics.isEmpty)
25             Seq.empty[MetadataResponse.TopicMetadata]
26         else
27             // 从元数据缓存过滤出相关主题的元数据
28             getTopicMetadata(metadataRequest.allowAutoTopicCreation, authorizedTopics
29                 errorUnavailableEndpoints, errorUnavailableListeners)
30
31     // ...
32     // 获取所有Broker列表
33     val brokers = metadataCache.getAliveBrokers
34 }
```

```
35     trace("Sending topic metadata %s and brokers %s for correlation id %d to client",
36           brokers.mkString(", "), request.header.correlationId, request.header.clientId)
37
38     // 构建Response并发送
39     sendResponseMaybeThrottle(request, requestThrottleMs =>
40       new MetadataResponse(
41         requestThrottleMs,
42         brokers.flatMap(_.getNode(request.context.listenerName)).asJava,
43         clusterId,
44         metadataCache.getControllerId.getOrElse(MetadataResponse.NO_CONTROLLER_ID),
45         completeTopicMetadata.asJava
46       ))
47   }
```

这段代码的主要逻辑是，先根据请求中的主题列表，去本地的元数据缓存 `MetadataCache` 中过滤出相应主题的元数据，也就是我们上面那张图中，右半部分的那棵树的子集，然后再去本地元数据缓存中获取所有 Broker 的集合，也就是上图中左半部分那棵树，最后把这两部分合在一起，作为响应返回给客户端。

Kafka 在每个 Broker 中都维护了一份和 ZooKeeper 中一样的元数据缓存，并不是每次客户端请求元数据就去读一次 ZooKeeper。由于 ZooKeeper 提供了 Watcher 这种监控机制，Kafka 可以感知到 ZooKeeper 中的元数据变化，从而及时更新 Broker 中的元数据缓存。

这样就完成了一次完整的更新元数据的流程。通过分析代码，可以证实，我们开始的猜测都是没有问题的。

小结

最后我们对这节课的内容做一个总结。

首先，我们简单的介绍了 ZooKeeper，它是一个分布式的协调服务，它的核心服务是一个高可用、高可靠的一致性存储，在此基础上，提供了包括读写元数据、节点监控、选举、节点间通信和分布式锁等很多功能，**这些功能可以极大方便我们快速开发一个分布式的集群系统。**

但是，ZooKeeper 也并不是完美的，在使用的时候你需要注意几个问题：

1. 不要往 ZooKeeper 里面写入大量数据，它不是一个真正意义上的存储系统，只适合存放少量的数据。依据服务器配置的不同，ZooKeeper 在写入超过几百 MB 数据之后，性能和稳定性都会严重下降。
2. 不要让业务集群的可用性依赖于 ZooKeeper 的可用性，什么意思呢？你的系统可以使用 Zookeeper，但你要留一手，要考虑如果 Zookeeper 集群宕机了，你的业务集群最好还能提供服务。因为 ZooKeeper 的选举过程是比较慢的，而它对网络的抖动又比较敏感，一旦触发选举，这段时间内的 ZooKeeper 是不能提供任何服务的。

Kafka 主要使用 ZooKeeper 来保存它的元数据、监控 Broker 和分区的存活状态，并利用 ZooKeeper 来进行选举。

Kafka 在 ZooKeeper 中保存的元数据，主要就是 Broker 的列表和主题分区信息两棵树。这份元数据同时也被缓存到每一个 Broker 中。客户端并不直接和 ZooKeeper 来通信，而是在需要的时候，通过 RPC 请求去 Broker 上拉取它关心的主题的元数据，然后保存到客户端的元数据缓存中，以便支撑客户端生产和消费。

可以看到，目前 Kafka 的这种设计，集群的可用性是严重依赖 ZooKeeper 的，也就是说，如果 ZooKeeper 集群不能提供服务，那整个 Kafka 集群也就不能提供服务了，这其实是一个不太好的设计。

如果你需要部署大规模的 Kafka 集群，建议的方式是，拆分成多个互相独立的小集群部署，每个小集群都使用一组独立的 ZooKeeper 提供服务。这样，每个 ZooKeeper 中存储的数据相对比较少，并且如果某个 ZooKeeper 集群故障，只会影响到一个小的 Kafka 集群，故障的影响面相对小一些。

Kafka 的开发者也意识到了这个问题，目前正在讨论开发一个元数据服务来替代 ZooKeeper，感兴趣的同学可以看一下他们的 [Proposal](#)。

思考题

本节课的思考题是这样的，请你顺着我们这节课源码分析的思路继续深挖进去，看一下 Broker 中的元数据缓存，又是如何与 ZooKeeper 中的元数据保持同步的呢？欢迎在留言区

写下你的想法。

感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (19)



书中迷梦

2019-11-11

个人感觉使用ZK不太适合做注册中心，因为作为注册中心首先要保证的是AP而不是CP，因为注册中心不是数据库

共 8 条评论 >

👍 31



饭粒

2019-11-17

ZooKeeper 集群宕机后 Kafka 不可以依靠客户端自身的元数据缓存或者 Broker 的元数据缓存工作吗？

作者回复: 你可以试验一下，实际上是不行的，因为Kafka它很多地方都依赖zk的watch机制，不仅仅是元数据缓存的问题。



👍 17



游弋云端

2019-09-26

这里就出现了一个矛盾，ZK理论上抽象了分布式协调服务的核心功能，让各个分布式组件不在单独去关注协调服务本身，但大家似乎又在努力的避免去依赖ZK，从而需要自己去实现这套逻辑。哎，分久必合，合久必分。

共 2 条评论 >

👍 11



jack

2019-10-01

老师，有两个问题：1、hadoop集群的高可用也是用zookeeper保证的，是不是也要拆分成小集群，2、那么提供服务的时候是否需要路由，将不同的业务分发到不同的集群上呢？

作者回复: Hadoop并不需要这么做，原因是，Hadoop它有它自己的NamingService，也就是namenode。ZK的作用只是协调namenode主从复制，namenode出现问题的时候，做主从切换使用。



👍 10



老杨

2019-09-23

老师能否讲讲RocketMQ或Kafka是如何做高可用，落地过程中有哪些坑，已经最佳实践有哪些？

作者回复: 通过学习5、22、23、24这几节课，你应该已经掌握了RocketMQ和Kafka的高可用的实现原理。具体操作上，你需要去看一下他们的对应的配置文档，应该可以正确的配置出可靠的集群。

至于所谓的“坑儿”，这俩产品都已经非常成熟了，只要你正确的配置和使用，基本上不会遇到什么“坑儿”的。



👍 6



山头

2019-09-26

老师，每个分区节点下面是一个名为 state 的临时节点，节点中保存着分区当前的 leader 和所有的 ISR 的 BrokerID。，这里的leader是指的一个分区的master,isr指的是从？

作者回复: ISR是所有“保持同步的节点”，包括Leader



👍 5



slam

2019-09-20

最后说到kafka严重依赖zk集群的可用性，有计划搞个服务替换，这里搞一套服务跟zk有什么区别？kafka的可用性不是也会依赖这个新服务吗？

作者回复: 他们肯定希望新的服务比zk更稳定

共 2 条评论 >

👍 3



海神名

2019-09-19

老师，听说kafka对部署环境资源要求较低，请问具体体现在哪些方面呢？有没有具体的推荐配置表？

作者回复: 主流的消息队列都对服务器的配置没有太多要求，如果你的业务不是处理海量消息，配置很低的服务也都可以满足需求。

共 2 条评论 >

👍 3



山头

2019-09-26

老师，你讲的很好，通过这节课懂了很多，能否说说一个消息发出去，我是如何路由到文件里去的呢？比如通过hash取模会对应到一个broker,又通过一个操作选择了一个队列，接着又怎么去选择写队列里的一个文件，思路模糊是没法造轮子的，请老师指正

作者回复: 我会考虑在后面的答疑中讲解这部分内容。



👍 2



山头

2019-09-26

老师，consumergroup和主题或者队列是什么关系，几对几的关系？还是consumergroup和分区有关系？是几对几

作者回复: 这个问题我在03和08两节课中有详细的描述，你可以复习一下。

共 2 条评论 >

👍 2



业余草

2019-09-19

kafka的优缺点都很明显，在架构平衡方便可以为我们借鉴！



👍 2



study的程序员

2021-05-07

某个分区的leader挂了，state节点消失，zk如何知道分区副本所在的broker是哪些？这个信息不是保存在state节点上的吗？难道在ids下的broker节点也会保存自身有哪些副本分区？zk

是这样冗余保存的吗？



挖掘机

2020-10-24

老师你好，既然kafka不会主动请求zk，那么客户端在本地没有元数据缓存的情况下写数据时是随便选一个broker进行连接，根据返回的元数据信息再连接对应的broker是吗？



周星平

2020-10-13

ZooKeeper 中的元数据修改通过Watcher通知broker删除缓存，下次查询时先访问ZooKeeper 中的元数据再缓存到broker本地；broker应该还需要心跳检测zookeeper是否存活吧



Berton.Yip.

2020-04-17

学到良多，谢谢老师讲解！



张瑞浩

2020-03-25

kafka初始化阶段怎么得知broker的信息的？

共 2 条评论 >



ub8

2019-12-15

使用临时节点来做动态扩容



Pantheon

2019-09-22

既然客户端是和borker进行通信获取元数据信息,为啥客户端在连kafka的时候还要指定zk的地址

作者回复: 新版本的Kafka不需要指定ZK地址。



许童童

2019-09-19

感谢老师的分享。

