北京郵電大學

分布计算环境

北京邮电大学计算机学院

Chapter 2

分布式系统的基本原理

主要内容

- 体系结构
- 进程
- 通信
- 命名
- 一致性和复制
- 容错
- 安全
- 例：ZooKeeper

Zookeeper主要内容

- **ZooKeeper是什么**
- **ZooKeeper应用举例**
- **ZooKeeper基本工作原理**

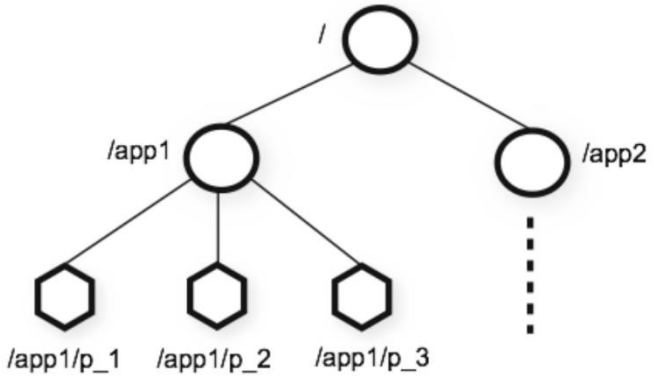
- **主要参考文献：从Paxos到ZooKeeper 分布式一致性原理与实践**

Zookeeper是什么

- **ZooKeeper是一个开源的高可用分布式协调系统，由雅虎创建，并于2010年正式成为Apache顶级项目**
  - **由Apache Hadoop的子项目发展而来，是Google Chubby的开源实现**
- **主要设计目标是将复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的API接口提供给用户使用**
  - 基于这些接口，可以方便地实现数据发布订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master选举、分布式锁、分布式队列等
- **Zookeeper通常由一组机器构成一个集群提供服务**
  - **一般建议部署奇数台服务器：**搭建一个最大允许N台服务器宕机的Zookeeper集群，最少需要部署2N+1台服务器

Zookeeper的数据模型

- **ZooKeeper维护一棵树：Znode Tree，即一个名字空间**
  - **每个数据节点Znode会有自己的数据内容和一系列属性**



- **Znode分为两类**
  - 持久节点：一旦创建就一直保存在ZK中，除非主动删除
  - 临时节点：生命期与客户端会话绑定，会话失效，被删除
  - 两类节点均分有序和无序两类
- **Znode Tree保存在内存中，并定时向磁盘保存快照**
- **ZK保证Znode Tree中数据访问的顺序一致性、高可用性和高性能**

Zookeeper提供的基本接口

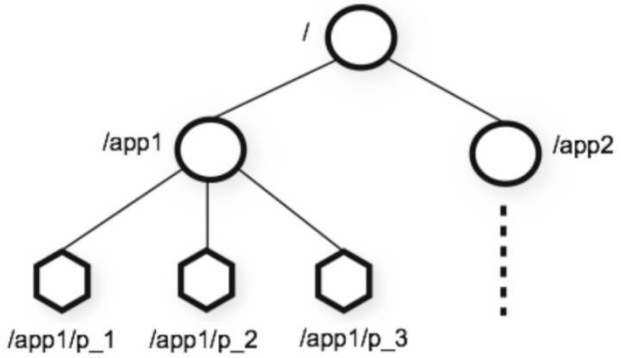
- ZK为用户提供的接口主要用于Znode Tree中节点的创建、删除、读取、更新等
- 创建会话：客户端和ZK通信，需要先创建一个会话

```
ZooKeeper myZK = new ZooKeeper("domain1.book.zookeeper:2181", 5000, myWatcher1);
```

- 创建节点：

```
String path1 = myZK.create("/ZK_test-ephemeral-", "".getBytes(),
                           Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
```

- 删除节点：myZK.delete(path, ….);
- 读取节点数据：
  - myZK.getChildren(path, myWatcher2,…..,);
  - myZK.getData(path,myWatcher3,…..);



Zookeeper提供的基本接口

- **更新数据**
  - myZK.setData(path, data, …);
- **检查节点是否存在**
  - myZK.exists(path, myWatcher4);
- **权限控制**
  - myZK.addAuthInfo(“digest”, “foo:true”.getBytes());

- **为方便基于ZooKeeper进行应用开发，有第三方ZooKeeper客户端产品在ZK原生API上进行封装，实现了更加方便易用的客户端开发包，如**
  - ZkClient，实现了会话超时重连、Watcher反复注册等功能

Zookeeper主要内容

- **ZooKeeper是什么**
- **ZooKeeper应用举例**
- **ZooKeeper基本工作原理**

Zookeeper的应用

- **通过对ZooKeeper中丰富的数据节点类型进行综合使用，配合Watcher时间通知机制，可以非常方便地构建一系列的分布式应用都会涉及的核心功能，如**
  - 数据发布订阅
  - 负载均衡
  - 命名服务
  - 分布式协调/通知
  - 集群管理
  - Master选举
  - 分布式锁
  - 分布式队列
  - ……

例：基于数据发布订阅的配置管理

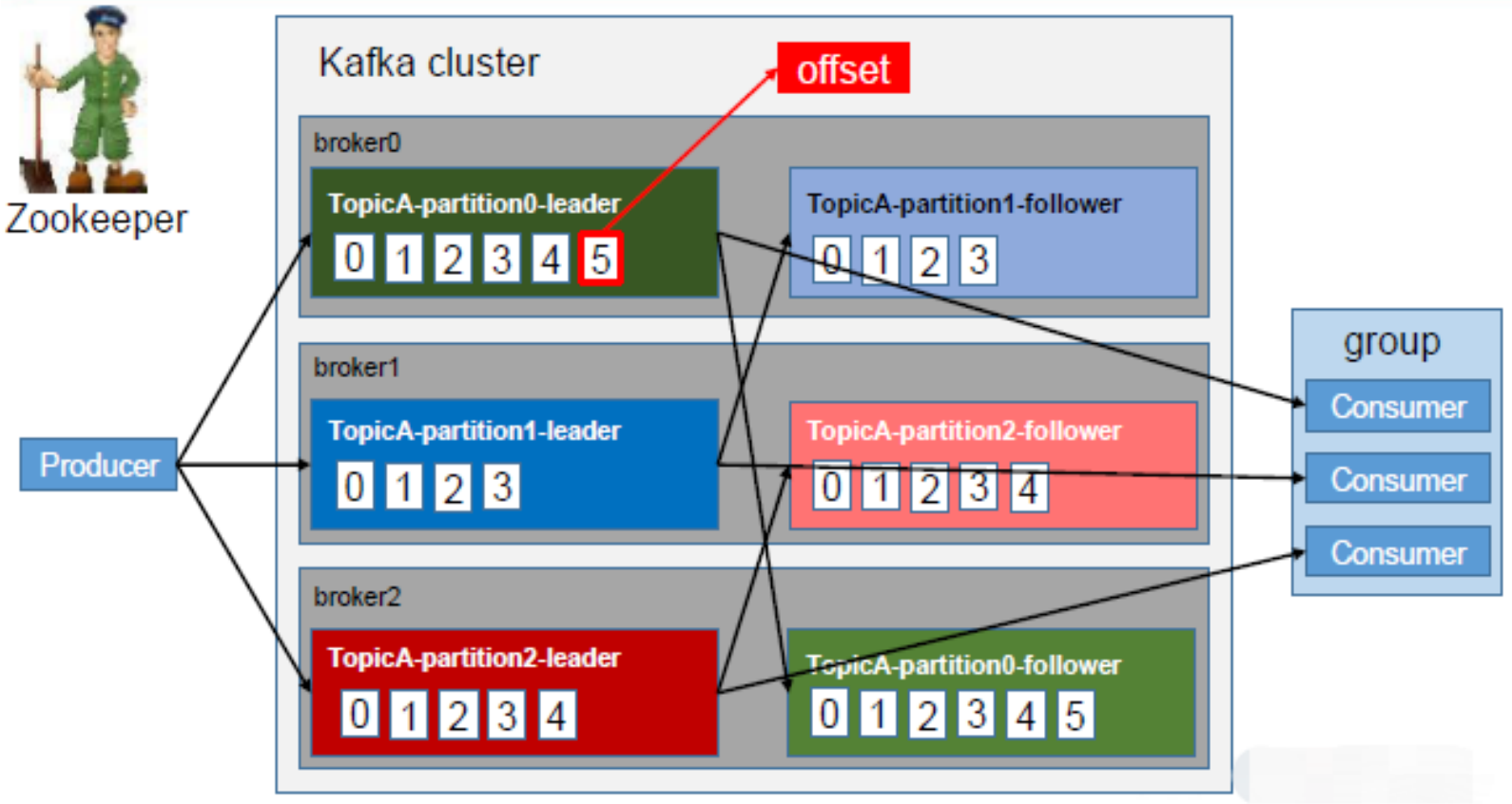
- **分布式系统中要使用一些通用的配置信息：机器列表、数据库配置等**
  - 在系统运行时动态可变，且集群中各机器共享
- **使用ZooKeeper进行配置管理**
  - 首先将初始化配置存储到ZooKeeper上：选取或创建一个数据节点用于配置信息的存储
  - 集群中每个机器节点在启动初始化阶段，读取配置信息，同时在该节点注册一个数据变更的Watcher
  - 当有变更时，更新该节点，ZooKeeper将变更通知发布给该分布式应用系统集群中的所有机器节点，则各个机器节点重新进行数据获取，进行相应的配置更改

例：Master选举

- **在分布式系统中，常常有主（Master）备（Slave）**
  - 如客户端的写请求由Master处理
- **使用ZooKeeper实现Master选举的基本原理：ZooKeeper的一致性能力，能够保证在分布式高并发下节点创建的唯一性**
  - 没有Master时，客户端集群往ZooKeeper上创建一个同名临时节点，ZK能够保证只有一个客户端能够成功创建这个节点，成为Master
  - 其它没有创建成功的节点为Slave，并注册一个节点变更Watcher，一旦当前Master宕机，宕机Master创建的临时节点被删除，通知到所有节点，重新进行Master选举

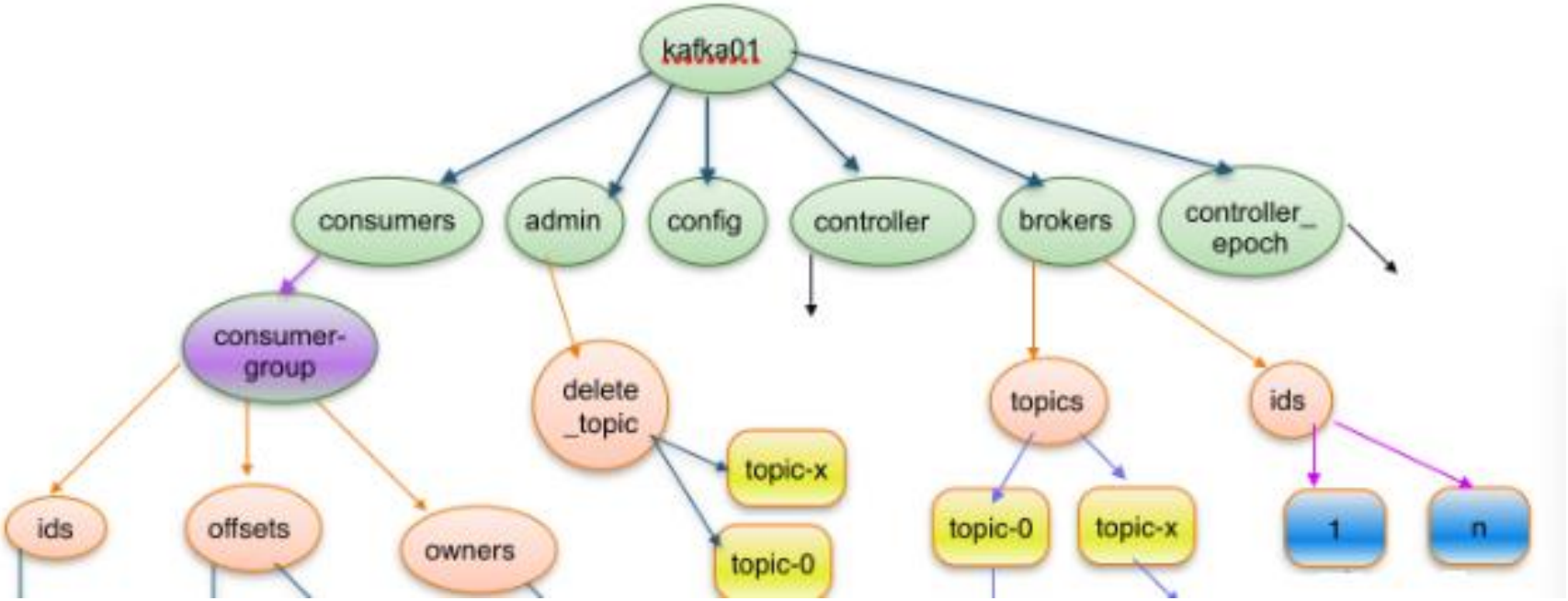
例：在Kafka中的应用

Kafka回顾：发布订阅消息中间件



Kafka在ZK中的数据模型概览

- Kafka集群通过在ZooKeeper中操纵一个记录了各类元数据的Znode Tree，来协调整个集群的工作
- Kafka新版本逐渐减少了对ZK的依赖，但是broker仍然依赖于ZK
- 这里基于老版本介绍



Broker注册

- **Broker分布式部署且相互独立，需要一个注册系统将整个集群中的Broker管理起来：ZooKeeper**
- **在ZK上有一个节点，用于记录Broker服务器列表**
  - /brokers/ids
- **每个Broker服务器启动时，到该节点下创建属于自己的节点**
  - /brokers/ids/[0…N]: 如 /brokers/ids/3
    - ➔Kafka使用全局唯一的数字来指代每个Broker服务器
  - 把自己的IP地址和端口号写到该节点中
- **Broker创建的节点是临时节点，一旦宕机，该节点被删除**
  - 节点变化可以动态表示Broker的可用性

Topic注册

- **同一个Topic的消息会被分成多个分区并将其分布在多个Broker上，这些分区信息以及与Broker的对应关系也由Zookeeper进行维护，由专门的节点记录**
  - /borkers/topics
- **每个Topic都以/brokers/topics/[topic]的形式被记录，如**
  - /brokers/topics/login
- **Broker服务器启动后，会到对应Topic节点（/brokers/topics）上注册自己的Broker ID并写入针对该Topic的分区总数，如**
  - /brokers/topics/login/3->2
  - 该节点为临时节点

生产者负载均衡

- 同一个Topic消息会被分区并分布在多个Broker上，因此，生产者需要将消息合理地发送到这些分布的Broker上
- 生产者对ZooKeeper上记录的一些节点的事件注册Watcher监听，即可根据实际情况进行负载均衡，如
  - Broker的新增与减少
  - Topic的新增与减少
  - Broker与Topic关联关系的变化

消费者负载均衡

- Kafka中的消费者同样需要进行负载均衡，以便多个消费者能合理地从对应的Broker服务器上接收消息
- Kafka消费者是分组的，不同组之间互不干扰
- 消费者负载均衡考虑同一个消费组内部的消息消费策略

分区与消费者关系

- Kafka为每个消费组分配一个全局唯一的Group ID，为每个消费者分配一个Consumer ID
- 每个消息分区只能被同组的一个消费者进行消费，因此，需要在 Zookeeper 上记录 消息分区 与 Consumer 之间的关系
- 每个消费者一旦确定了对一个消息分区的消费权力，就将其 Consumer ID 写入到 Zookeeper 对应消息分区的临时节点上，例如
  - /consumers/[group_id]/owners/[topic]/[broker_id-partition_id]
- 其中，[broker_id-partition_id]就是一个 消息分区 的标识，节点内容就是该消息分区上消费者的Consumer ID

消息消费进度Offset记录

- Kafka在消费者对指定消息分区进行消息消费时，需要定时地将分区消息的消费进度Offset记录到Zookeeper上
  - 以便在该消费者进行重启或者其他消费者重新接管该消息分区的消息消费后，能够从之前的进度开始继续进行消息消费
- Offset在Zookeeper中由一个专门节点进行记录，其节点路径为
  - /consumers/[group_id]/offsets/[topic]/[broker_id-partition_id]
- [broker_id-partition_id]就是一个 消息分区 的标识，节点内容就是Offset的值

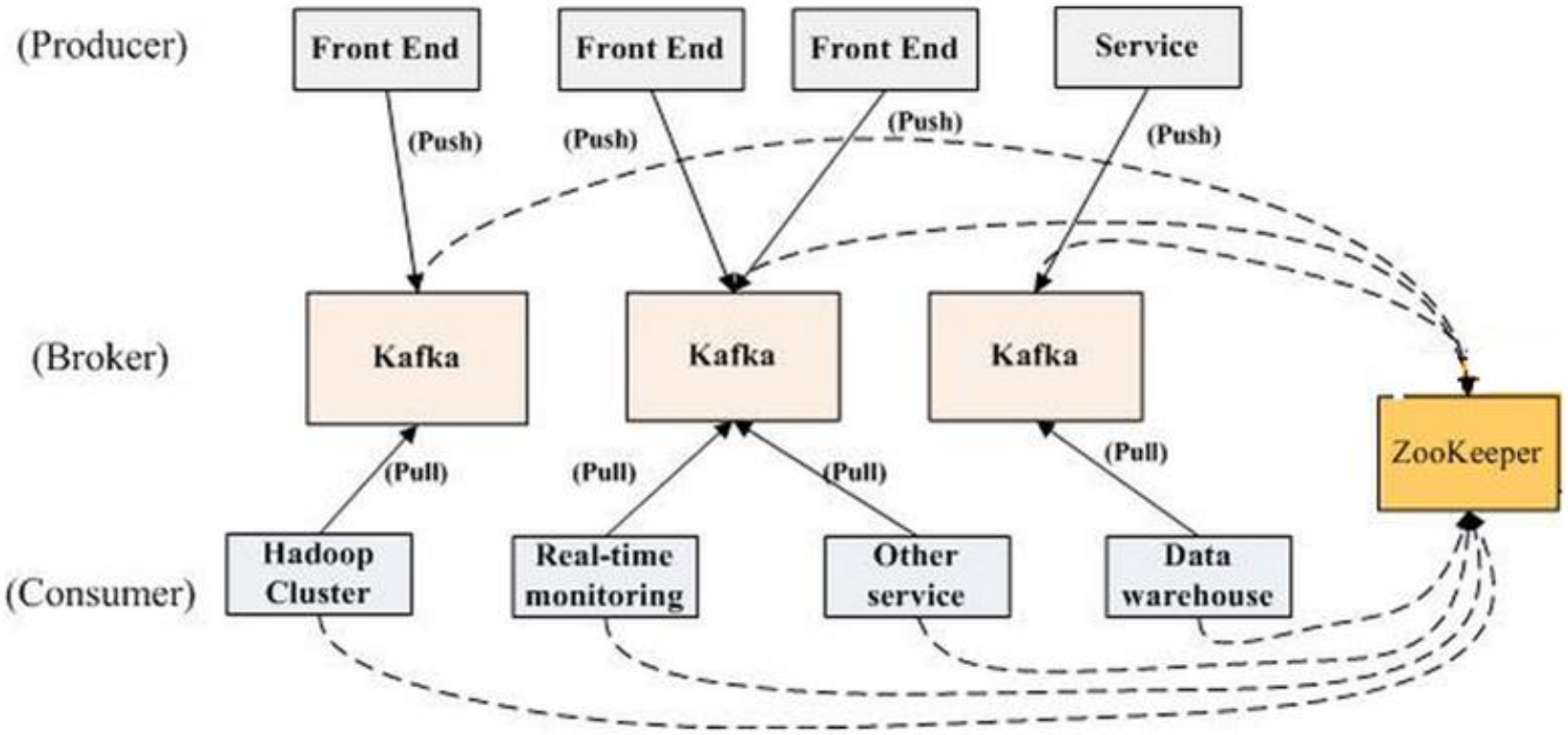
消费者注册

- **消费者服务器在初始化启动时加入消费者分组的过程为**
  - 1. 注册到消费组：每个消费者服务器启动时，会到Zookeeper的指定节点下创建一个属于自己的消费者节点，如/consumers/[group_id]/ids/[consumer_id]，并将自己订阅的Topic信息写入该临时节点
  - 2.对消费组中消费者的变化注册监听：即对/consumers/[group_id]/ids 节点注册子节点变化的Watcher监听，一旦发现消费者新增或减少，就触发消费者的负载均衡
  - 3.对Broker服务器的变化注册监听：对/broker/ids/[0-N]中的节点进行监听，如果发现Broker服务器列表发生变化，就根据具体情况来决定是否需要进行消费者负载均衡
  - 4.进行消费者负载均衡：为了让同一个Topic下不同分区的消息尽量均衡地被多个消费者消费，进行消费者与消息分区的分配
    - ➔ 具体方法略

分区Leader确定

- 可能的方法：leader在zk上创建一个临时节点，所有Follower对此节点注册监听。当leader宕机时，ISR（一组与Leader数据一致的副本）里的所有Follower都尝试创建该节点，创建成功者即是新的Leader，其它副本为Follower
- 但当kafka集群业务量很大，partition达到成千上万时，若某broker宕机会涉及很多分区调整，大量Watch事件被触发，造成Zookeeper负载过重
  - ZooKeeper不适合大量的写
- 因此，Kafka在所有broker中选出一个controller，所有分区的Leader都由controller决定
  - Controller在Zookeeper注册Watch，一旦有Broker宕机，注册的Watch得到通知，对该Broker上的Leader分区重新确定Leader：获取相应分区的ISR，从中确定一个副本为Leader

Kafka 与ZooKeeper关系小结



- 通过 Zookeeper 管理集群配置，选举 leader，以及在broker集群或 Consumer Group 发生变化时进行 rebalance
- 生产者可对ZooKeeper上记录的一些节点的事件注册Watcher监听，从而根据实际情况进行负载均衡

Zookeeper主要内容

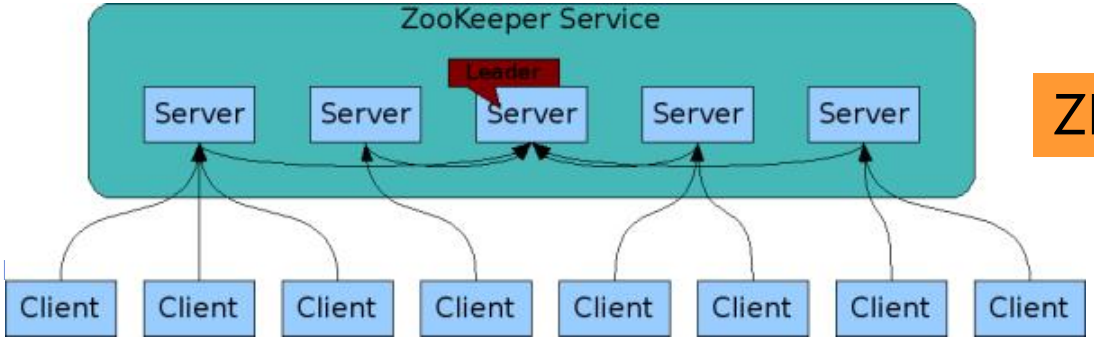
- ZooKeeper是什么
- ZooKeeper应用举例
- ZooKeeper基本工作原理

ZooKeeper的基本需求

- ZooKeeper需要满足高性能、高可用、且具有严格的顺序访问控制能力（主要是写操作的严格顺序性）等需求，要保证如下分布一致性特性
  - 顺序一致性：从一个客户端发起的事务请求，最终会严格地按照其发起顺序被应用到ZooKeeper中去
  - 原子性：所有事务请求的处理结果在整个ZK集群机器上的应用结果是一致的
  - 单一视图：无论客户端连接哪一个ZooKeeper服务器，看到的服务器数据模型都是一致的
  - 可靠性：一旦服务端成功地应用了一个事务，并完成对客户的响应，那么该事务引起的服务端状态变更将一直被保留，直到另一个事务对其进行了变更
  - 实时性：在一定的时间段内，客户端一定能够从服务端读取到最新的数据状态

ZooKeeper集群中的角色

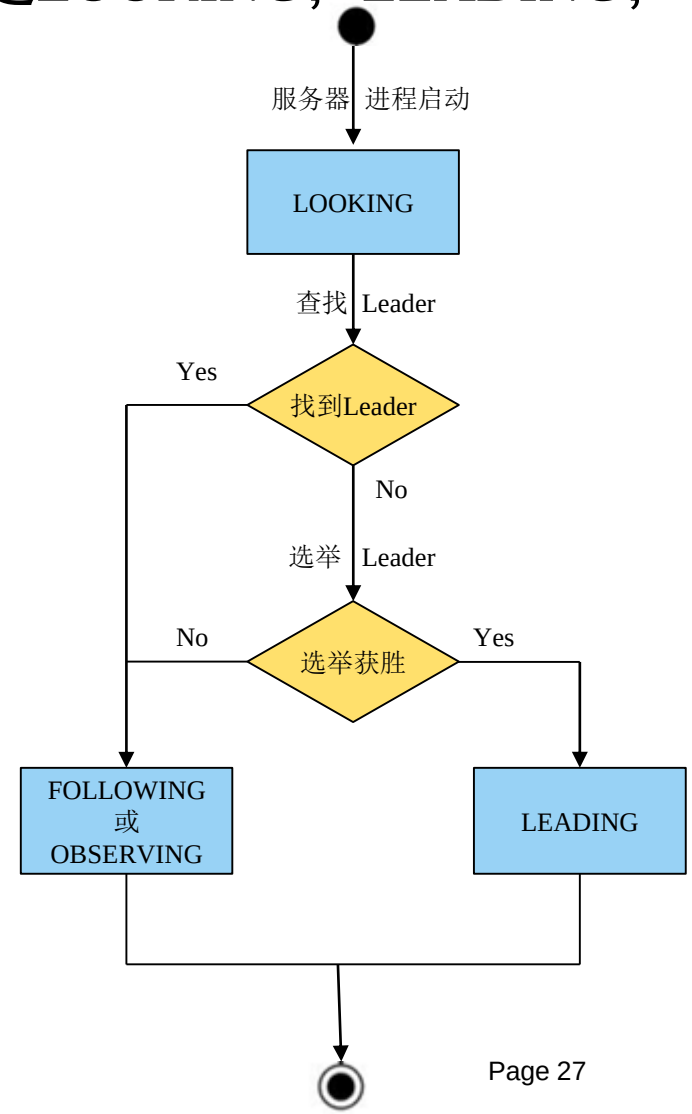
- Leader：ZK 集群工作机制的核心
  - 集群内部各个服务节点的调度者
  - 事务请求的唯一调度和处理者，保证集群事务处理的顺序性
    - ➔每个事务proposal会分配一个全局单调递增的唯一ID：zxid
- Follower：
  - 直接处理非事务请求，对于事务请求，转发给 Leader
  - Proposal 投票：Leader 上执行事务时，需要 Follower 投票，过半数同意 Leader 才真正执行
  - Leader 选举投票
- Observer：不影响事务处理能力，提升 ZK 集群的非事务处理能力
  - 直接处理非事务请求，对于事务请求，转发给 Leader



ZK适合读多写少的场合

ZooKeeper集群中服务器的状态

- 服务器（进程）具有四种状态，分别是LOOKING，LEADING，FOLLOWING， OBSERVING
- 服务器的状态转移



状态的转移以及在LEADING，FOLLOWING， OBSERVING状态中保证一致性地处理客户的请求，都是基于ZAB协议实现的

ZAB协议

- ZAB（Zookeeper Atomic Broadcast，Zookeeper原子广播）协议是Zookeeper保证数据一致性的核心算法
  - 借鉴了Paxos算法，是特别为Zookeeper设计的支持崩溃恢复的原子广播协议，也借鉴了2PC两阶段提交协议，但过半数提交
    - ➔Paxos算法是lamport提出的目前公认的解决分布式一致性问题最有效的算法之一
- ZooKeeper主要依赖ZAB协议来实现数据一致性
  - 基于该协议，zk实现了一种基于主备模型（即Leader和Follower模型）的系统架构来保证集群中各个副本之间数据的一致
  - 基于该协议，进行Leader选举、同步、广播以便一致地处理事务
- ZAB协议的特性
  - 确保那些已经在 Leader 服务器上提交（Commit）的事务最终被所有的服务器提交
  - 确保丢弃那些只在 Leader 上被提出而没有被提交的事务

Leader选举

- 当ZK集群初始化或Leader宕机时，服务器进入LOOKING状态，进行Leader选举
- Leader竞选基本规则：服务器持有的数据越新（zxid越大）、服务器ID（sid）越大（配置），则越可能为Leader
  - 开始时所有节点推荐自己为Leader，广播自己的 (sid，zxid)
  - 选举过程中接收到更合适推荐人（(sid,zxid)更大），则更改推荐人，并广播到所有节点

    ➔先比较zxid；zxid相同，比较sid
  - 当任意一个服务器节点收到的投票数，超过了法定数量(quorum，超过半数)，则升级为 Leader，状态改为LEADING，其它节点根据具体情况，状态改为FOLLOWING或OBSERVING

服务器同步

- **Leader选完之后，ZooKeeper集群进入同步阶段**
  - Leader从支持它的过半Follower中发现出具有最新事务历史集合的Follower，将其历史事务集合作为初始化事务集合，记为H
  - Leader向Follower广播这个H
  - Follower如果可以参与此次同步，会接受H中的事务，并反馈给Leader
  - 当Leader收到过半数的Follower的反馈消息，就向所有Follower广播commit消息
  - Follower收到Commit消息，提交所有H中未处理的事务

接受新的事务请求（广播）

- 完成同步后，ZK集群就可以正式接收客户端新的事务请求了
- Zookeeper 客户端会随机地连接到 zookeeper 集群中的一个节点，如果是读请求，就直接从当前节点中读取数据；如果是写请求，该节点会向 Leader 转发这个请求，由Leader进行统一处理
  - Leader接受到新的事务请求后，生成新的事务Proposal，并按照zxid的顺序，向所有Follower发送提案
  - Follower根据接受消息的先后顺序来处理这些提案，将其写到历史事务集合中，然后反馈ACK给Leader
  - Leader接受到来自过半Follower针对该提案的ACK，就发送Commit消息给所有Follower，commit这个提案，其自身也提交这个事务
  - Follower收到该提案的Commit消息后，就提交这个提案

    ➔此时，之前的提案必定已经提交

重新选举Leader

- Leader进程与所有Follower进程之间通过心跳检测机制来感知彼此的情况
- 如果在超时时间内Leader无法从过半的Follower进程那里接收到心跳消息，则Leader会终止这一周期的领导，转换为LOOKING状态。所有Follower也会放弃这个Leader，进入LOOKING状态。然后开始新一轮Leader选举，开始进入下一周期
  - 选举----同步----广播（接收新事务请求）

ZooKeeper一致性特性回顾

- 顺序一致性：从一个客户端发起的事务请求，最终会严格地按照其发起顺序被应用到ZooKeeper中去
  - 事务请求由Leader负责，事务zxid单调递增
- 原子性：所有事务的处理结果在整个ZK集群机器上是一致的
  - ZAB协议确保那些已经在 Leader 服务器上提交（Commit）的事务最终被所有的服务器提交
- 单一视图：无论客户端连接哪一个ZK服务器，数据都一致
- 可靠性：成功提交的事务结果一直保留，直到另一个事务对其进行了变更
  - 集群复制，事务日志，数据快照
- 实时性：在一定的时间段内，客户端一定能够从服务端读取到最新的数据状态
  - 多个Follower和Observer，最终一致性，过半数提交
  - Tnode Tree保存在服务器进程内存中

本章总结

- **体系结构**
  - 软件体系结构，样式：分层、面向对象、以数据为中心、基于事件
  - 系统体系结构：集中式、非集中式、混合式
- **进程**
  - 进程、线程
  - 客户：胖瘦情况根据需要
    - ➔客户进程中的多线程
    - ➔客户进程对分布透明性的支持
  - 服务器：迭代服务器、并发服务器
    - ➔客户如何找到服务器？
    - ➔客户如何中断服务器的工作？
    - ➔服务器的状态问题

本章总结（2）

- **通信**
  - 分层通信：OSI七层、中间件协议、通信的持久性和瞬时性，同步性和异步性
  - RPC和RMI：同步
  - 面向消息的通信：瞬时（如socket、MPI）、持久（如kafka）
  - 面向流的通信：QoS、流同步
  - 多播通信
- **命名**
  - 命名系统：名字和位置的无关性，名字到地址的 解析
  - 非结构化命名、结构化命名、基于属性的命名
- **复制和一致性**
  - 复制：为了性能或可伸缩性、容错
  - 要求副本的一致：松弛的一致性
  - 多种一致性模型：以数据为中心的（如顺序一致性）、以用户为中心的（如单调读一致性）

本章总结（3）

- **容错**
  - 故障类型、失效类型、冗余
  - 客户服务器容错问题：RPC
  - 进程容错技术：进程组、两军问题、拜占庭将军问题
  - 事务容错技术：两阶段提交
  - 恢复方法：分布式检查点、消息日志
- **安全**
  - 安全性概述：安全威胁、安全机制、加密（对称、非对称加密系统）
  - 安全通道：身份认证、会话密钥、数字签名
  - 访问控制：访问控制矩阵、权能、防火墙、拒绝服务
  - 安全管理：密钥管理、授权管理
- **例：ZooKeeper**