

分布式协调服务器 Zookeeper

面试题暨知识点总结

第 1 次直播课

【Q-01】对于 zk 功能的开发，或者说是对于 zk 在具体应用场景中的解决方案的设计中，我们要着重考虑对哪两个 zk 特性的灵活使用？请谈一下你的看法。

【RA】对于 zk 功能的开发，或者说是对于 zk 在具体应用场景中的解决方案的设计中，我们要着重考虑 zk 的临时节点及 watcher 监听机制的灵活使用。

- 临时节点：其生命周期与创建它的客户端会话是绑定在一起的。客户端消失，则临时节点消失。
- watcher 机制：客户端可以对节点进行监听，监听其数据内容的变更、子节点列表的变更等。

【Q-02】对于 zk 的节点类型，谈一下你的认识。

【RA】每个 znode 根据节点类型的不同，具有不同的生命周期与特性。

- 持久节点：节点被创建后会一直保存在 zk 中，直到将其删除。
- 持久顺序节点：一个父节点可以为它的第一级子节点维护一份顺序，用于记录每个子节点创建的先后顺序。其在创建子节点时，会在子节点名称后添加数字后缀，作为该子节点的完整节点名。序号由 10 位数字组成，由这些子节点的父节点维护，从 0 开始计数。
- 临时节点：临时节点的生命周期与客户端的会话绑定在一起，会话消失则该节点就会被自动清理。
- 临时顺序节点：添加了创建序号的临时节点。

【Q-03】对于 zk 来说 watcher 机制非常重要，watcher 机制的工作原理是怎样的？谈一下你的认识。

【RA】当客户端想要监听 zk 中某节点的状态变化时，需要向该节点注册 watcher 监听。其首先会在客户端创建一个 watcher 对象，并为其添加相应的回调。当 zk 中对应的节点发生了相应的 watcher 事件后，zk 会向客户端发送事件通知，触发该 watcher 回调的执行。执行完毕，该 watcher 对象销毁。若要再次监听，则需再次注册。

【Q-04】对于 zk 官方给出了四种最典型的应用场景，配置维护就是之一。什么是配置维护？请谈一下你的看法。

【RA】分布式系统中，很多服务都是部署在集群中的，即多台服务器中部署着完全相同的应用，起着完全相同的作用。当然，集群中的这些服务器的配置文件是完全相同的。

若集群中服务器的配置文件需要进行修改，那么我们就需要逐台修改这些服务器中的配置文件。如果我们集群服务器比较少，那么这些修改还不是太麻烦，但如果集群服务器特别多，比如某些大型互联网公司的 Hadoop 集群有数千台服务器，那么纯手工的更改这些配置文件几乎就是一件不可能完成的任务。即使使用大量人力进行修改可行，但过多的人员参与，出错的概率大大提升，对于集群所形成的危险是很大的。

zk 可以通过一种发布订阅模式对集群中的配置文件进行统一管理，这就是配置维护。其作用与 Spring Cloud Config 的作用相同。

【Q-05】对于 zk 官方给出了四种最典型的应用场景，命名服务就是之一。不过，像 UUID、GUID 等就可以非常方便地生成几乎不会重复的 id，为什么还要那么麻烦的使用 zk 实现呢？

【RA】不错，UUID、GUID 可以非常方便地生成几乎不会重复的 id，但也存在问题。例如，id 过长，占用空间；虽然那么长，但还没有语义，无法从 id 中解析出业务数据。而使用 zk 生成的 id 一定是不重复的，且可以根据业务自定义 id 长度和语义。这是 UUID、GUID 无法替代的。

【Q-06】对于分布式日志系统，无论是日志源主机还是日志收集主机，都需要监控其存活状态。请设计一种方案，可以方便地监控日志收集主机的存活状态。

【RA】对于分布式日志系统中日志源主机存活状态的监控是比较简单的，只需要让日志源主机在 zk 中其对应的日志收集主机节点下创建一个临时节点，系统只需向这个日志收集主机注册子节点列表变更 watcher 监听即可。

但是对于日志收集主机就不同了，因为其是持久节点，主机宕机也不会引发持久节点的消失。此时可以这样实现：在日志收集主机对应的持久节点创建完毕后，马上在其下创建一个临时节点，然后让系统为该节点注册一个节点删除 watcher 监听。一旦日志收集主机宕机，该临时节点就会消失，然后就会触发该节点删除 watcher 的回调，然后就可以进行 Rebalance。

【Q-07】使用 DBMS 可以实现 Master 选举，实现原理是什么？存在什么问题？请谈一下你的看法。

【RA】使用 DBMS 实现 Master 选举的原理是，集群中所有主机都向关系型数据库中插入一条相同主键的记录，数据库会自动进行主键冲突检查，然后只允许一台主机插入成功。那么这个插入成功的主机就是 Master。

但该方案存在一个问题：若 Master 宕机，DBMS 无法通知其它 Slave，让其它 Slave 重新发起 Master 的再次选举。

【Q-08】使用 zk 可以实现 Master 选举，实现原理是什么？请谈一下你的看法。

【RA】使用 zk 实现 Master 选举的原理是，集群中所有主机都向 zk 中创建相同路径下的某持久节点注册子节点列表变更 watcher 监听，并在该节点下持久相同名称的临时节点，谁创建成功谁就是 Master。

当 Master 宕机，该临时节点消失，此时会触发其他主机 watcher 回调的执行。watcher 回调会重新抢注该节点下的临时节点，谁注册成功谁就是 Master。即可以实现 Master 宕机后的自动重新选举。

【Q-09】“数据复制总线”的功能可以由 MySQL 主备集群完成吗？数据复制总线与 zk 有什么关系。请谈一下你的认识。

【RA】MySQL 主备集群完成的是相同数据库间的数据备份，而“数据复制总线”则可以完成不同 DBMS 中异构 DB 间数据的复制、变更。

数据复制总线中复制任务是由复制器 replicator 完成的。为了保证复制任务的顺利完成，解决 replicator 的单点问题，replicator 采用了主备模式。即只有一个是可以工作的 RUNNING 状态，其它均为热备的 STANDBY 状态。谁是 RUNNING 状态？RUNNING 与 STANDBY 间如何实现协调同步的工作？RUNNING 状态的 replicator 宕机，哪个 STANDBY 的 replicator 变为 RUNNING 状态？这些都需要由 zk 来负责。

【Q-10】 我们可以通过上层业务向关系型数据库中添加额外的行锁、表锁或事务处理等来实现分布式锁。为什么还需要通过 zk 来实现分布式锁功能？请谈一下你的看法。

【RA】 我们可以通过上层业务向关系型数据库中添加额外的行锁、表锁或事务处理等来实现分布式锁。但目前绝大多数分布式系统的性能瓶颈都集中在数据库操作引发的数据库性能上，若再给数据库添加更多的额外功能，会势必会增加数据库的负担，降低整个系统的性能。此时可以使用第三方的 zk 来实现分布式锁，以减轻 DBMS 的压力。当然，zk 实现的分布式锁还可以实现非 DBMS 中共享数据的读写访问控制。

【Q-11】 什么是锁机制？锁都有哪些分类？请谈一下你的认识。

【RA】 锁机制是一种对共享数据读写访问的控制机制，其有两种类型：

- 排他锁（**eXclusive locks**）：也称为 X 锁、写锁。若事务 T 对数据对象 O 加了排他锁，那么在整个加锁期间，只允许事务 T 对 O 进行读写操作，其他任何事务都不能再对该数据对象进行任何操作，直到 T 释放了排他锁。
- 共享锁（**Shared locks**）：又称为 S 锁、读锁。若事务 T 对数据对象 O 加了共享锁，那么该事务只能对 O 进行读操作。当然，其它事务同时也可以对数据对象 O 再添加共享锁，且可以对 O 进行读操作。只有当所有共享锁全部释放后，才可对 O 添加排他锁。

第 2 次直播课

【Q-01】 一个 zk 客户端到底连接的是 zk 列表中的哪台 Server？请谈一下你的看法。

【Q-02】 (追问) 为什么要打散？请谈一下你的看法。

【Q-03】 (再追问, 看来是打破砂锅了) zk 客户端指定的要连接的 zk 集群地址，会被 shuffle 几次？请谈一下你的看法。

【Q-04】 第一个客户端将 zk 列表打散后，在打散的列表上采取轮询方式尝试连接。那么，第二个客户端又来连接 zk 集群，其是在前面打散的基础上采用轮询方式选择 Server，还是又重新打散后再进行轮询连接尝试？

【Q-05】 zk 客户端维护着会话超时管理，请谈一下你对此的认识。

【Q-06】 zk 是 CP 的，zk 集群在数据同步或 leader 选举时是不对外提供服务的，那岂不是用户体验非常不好？请谈一下你对此的看法。

【Q-07】 zk Client 在连接 zk 时会发生连接丢失事件，什么是连接丢失？请谈一下你的认识。

【Q-08】 zk Client 在连接 zk 时会发生会话转移事件，什么是会话转移？请谈一下你的认识。

【Q-09】 zk Client 在连接 zk 时会发生会话失效事件，什么是会话失效？请谈一下你的认识。

【Q-10】 zk 中的会话空闲超时管理采用的是分桶策略。请谈一下你对分桶策略的认识。

【Q-11】zk 中的会话空闲超时管理采用的是分桶策略。什么是会话桶？里面存放的是什么？从源码角度请谈一下你的认识。

【Q-12】zk 中的会话空闲超时管理采用的是分桶策略。会话桶中的会话会发生换桶，什么时候会进行换桶？如何换桶呢？从源码角度请谈一下你的认识。

【Q-13】zk 中的会话空闲超时管理采用的是分桶策略。该分桶策略中会话空闲超时判断发生在哪？超时发生后的处理发生在哪？都做了哪些处理呢？请谈一下你的认识。



分布式协调服务器

Zookeeper



2020

分布式协调服务器 Zookeeper

第1章 Zookeeper 理论基础

1.1 Zookeeper 简介

ZooKeeper 由雅虎研究院开发，后来捐赠给了 Apache。ZooKeeper 是一个开源的分布式应用程序协调服务器，其为分布式系统提供一致性服务。其一致性是通过基于 Paxos 算法的 ZAB 协议完成的。其主要功能包括：配置维护、域名服务、分布式同步、集群管理等。

zookeeper 的官网：<http://zookeeper.apache.org>

Welcome to Apache ZooKeeper™

Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

What is ZooKeeper?

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services correctly, it is often easier to use a centralized service like ZooKeeper.

Consul、Doozerd、Etcd

1.2 一致性

zk 是如何保证分布式系统的一致性的呢？是因为 zk 具有以下几方面的特点：

1.2.1 顺序一致性

从同一个客户端发起的多个事务请求（写操作请求），最终会严格按照其发起顺序记录到 zk 中。

1.2.2 原子性

所有事务请求的结果在集群中所有 Server 上的应用情况是一致的。要么全部应用成功，要么都没有成功，不会出现部分成功，部分失败的情况。

1.2.3 单一视图

无论客户端连接的是集群中的哪台 Server，其读取到的数据模型中的数据都是一致的。

1.2.4 可靠性

一旦某事务被成功应用到了 zk，则会一直被保留下来，除非另一个事务将其修改。

1.2.5 最终一致性

一旦一个事务被成功应用，zk 可以保证在一段较短的时间内，客户端最终一定能够从服务端读取到最新的数据。但不能保证实时读取到。

1.3 Paxos 算法

对于 zk 理论的学习，最重要也是最难的知识点就是 Paxos 算法。所以我们首先学习 Paxos 算法。

1.3.1 算法简介

Paxos 算法是莱斯利·兰伯特(Leslie Lamport)1990 年提出的一种基于消息传递的、具有高容错性的一致性算法。Google Chubby 的作者 Mike Burrows 说过，世上只有一种一致性算法，那就是 Paxos，所有其他一致性算法都是 Paxos 算法的不完整版。Paxos 算法是一种公认的晦涩难懂的算法，并且工程实现上也具有很大难度。较有名的 Paxos 工程实现有 Google Chubby、ZAB、微信的 PhxPaxos 等。

Paxos 算法是用于解决什么问题的呢？Paxos 算法要解决的问题是，在分布式系统中如何就某个决议达成一致。

1.3.2 Paxos 与拜占庭将军问题

拜占庭将军问题是由 Paxos 算法作者莱斯利·兰伯特提出的点对点通信中的基本问题。该问题要说明的含义是，在不可靠信道上试图通过消息传递的方式达到一致性是不可能的。所以，Paxos 算法的前提是不存在拜占庭将军问题，即信道是安全的、可靠的，集群节点间传

递的消息是不会被篡改的。

一般情况下，分布式系统中各个节点间采用两种通讯模型：共享内存（Shared Memory）、消息传递（Messages Passing）。而 Paxos 是基于消息传递通讯模型的。

1.3.3 算法描述

(1) 三种角色

在 Paxos 算法中有三种角色，分别具有三种不同的行为。但很多时候，一个进程可能同时充当着多种角色。

- Proposer: 提案者
- Acceptor: 表决者
- Learner: 同步者

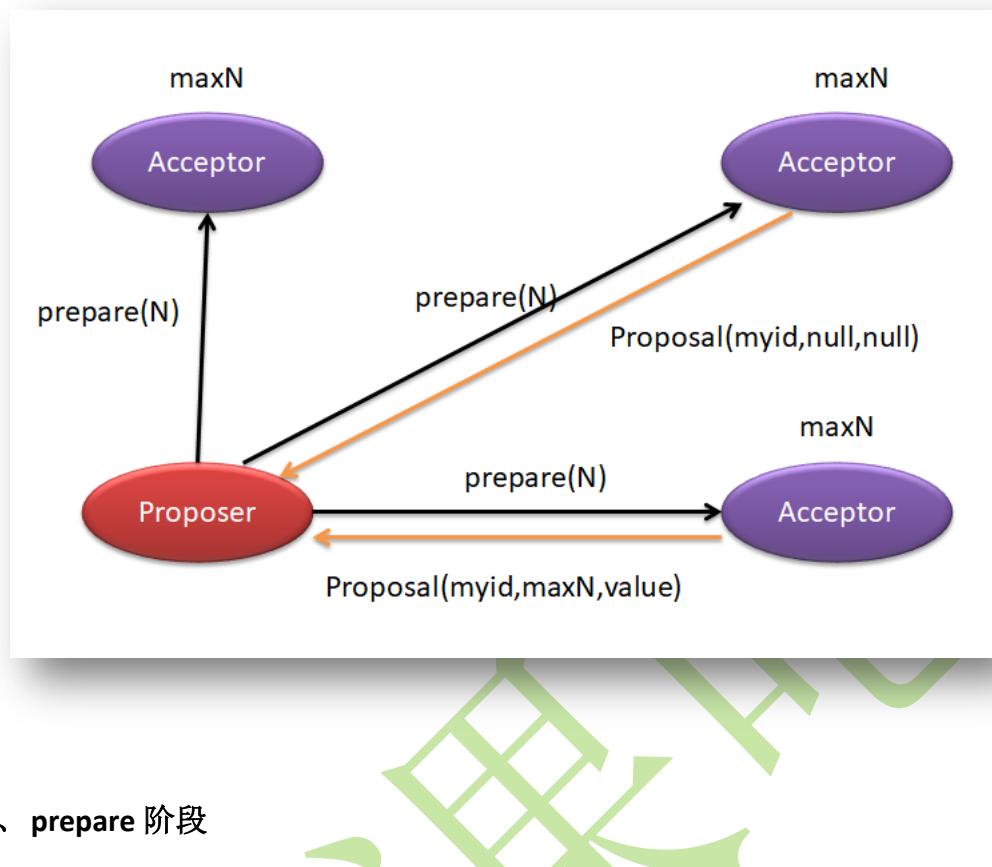
(2) Paxos 算法的一致性

Paxos 算法的一致性主要体现在以下几点：

- 每个提案者在提出提案时都会首先获取到一个具有全局唯一性的、递增的提案编号 N，即在整个集群中是唯一的编号 N，然后将该编号赋予其要提出的提案。
- 每个表决者在 accept 某提案后，会将该提案的编号 N 记录在本地，这样每个表决者中保存的已经被 accept 的提案中会存在一个编号最大的提案，其编号假设为 maxN。每个表决者仅会 accept 编号大于自己本地 maxN 的提案。
- 在众多提案中最终只能有一个提案被选定。
- 一旦一个提案被选定，则其它服务器会主动同步(Learn)该提案到本地。
- 没有提案被提出则不会有提案被选定。

(3) 算法过程描述

Paxos 算法的执行过程划分为两个阶段：准备阶段 prepare 与接受阶段 accept。



A、prepare 阶段

- 1) 提案者(Proposer)准备提交一个编号为 N 的提议，于是其首先向所有表决者(Acceptor)发送 **prepare(N)** 请求，用于试探集群是否支持该编号的提议。
- 2) 每个表决者(Acceptor)中都保存着自己曾经 **accept** 过的提议中的最大编号 **maxN**。当一个表决者接收到其它主机发送来的 **prepare(N)** 请求时，其会比较 N 与 **maxN** 的值。有以下几种情况：
 - a) 若 N 小于 **maxN**，则说明该提议已过时，当前表决者采取不回应或回应 **Error** 的方式来拒绝该 **prepare** 请求；
 - b) 若 N 大于 **maxN**，则说明该提议是可以接受的，当前表决者会首先将该 N 记录下来，并将其曾经已经 **accept** 的编号最大的提案 **Proposal(myid,maxN,value)** 反馈给提案者，以向提案者展示自己支持的提案意愿。其中第一个参数 **myid** 表示该提案的提案者标识 id，第二个参数表示其曾接受的提案的最大编号 **maxN**，第三个参数表示该提案的真正内容 **value**。当然，若当前表决者还未曾 **accept** 过任何提议，则会将 **Proposal(myid,null,null)** 反馈给提案者。
 - c) 在 **prepare** 阶段 N 不可能等于 **maxN**。这是由 N 的生成机制决定的。要获得 N 的值，其必定会在原来数值的基础上采用同步锁方式增一。

B、accept 阶段

- 1) 当提案者(Proposer)发出 **prepare(N)** 后，若收到了超过半数的表决者(Acceptor)的反馈，那么该提案者就会将其真正的提案 **Proposal(myid,N,value)** 发送给所有的表决者。
- 2) 当表决者(Acceptor)接收到提案者发送的 **Proposal(myid,N,value)** 提案后，会再次拿出自己曾经 **accept** 过的提议中的最大编号 **maxN**，或曾经记录下的 **prepare** 的最大编号，让 N

与它们进行比较，若 N 大于等于这两个编号，则当前表决者 accept 该提案，并反馈给提案者。若 N 小于这两个编号，则表决者采取不回应或回应 Error 的方式来拒绝该提议。

- 3) 若提案者没有接收到超过半数的表决者的 accept 反馈，则有两种可能的结果产生。一是放弃该提案，不再提出；二是重新进入 prepare 阶段，递增提案号，重新提出 prepare 请求。
- 4) 若提案者接收到的反馈数量超过了半数，则其会向外广播两类信息：
 - a) 向曾 accept 其提案的表决者发送“可执行数据同步信号”，即让它们执行其曾接收到的提案；
 - b) 向未曾向其发送 accept 反馈的表决者发送“提案 + 可执行数据同步信号”，即让它们接受到该提案后马上执行。

提问 1：

在 Prepare 阶段已经比较过了，并且已经通过了，为什么在 Accept 阶段还需要进行比较？

提问 2：

在 Prepare 阶段与 Accept 阶段都进行了比较，为什么在发送 COMMIT 信号量时无需进行比较？

1.3.4 Paxos 算法的活锁问题

前面所述的 Paxos 算法在实际工程应用过程中，根据不同的实际需求存在诸多不便之处，所以也就出现了很多对于基本 Paxos 算法的优化算法，以对 Paxos 算法进行改进，例如，Multi Paxos、Fast Paxos、EPaxos。

例如，Paxos 算法存在“活锁问题”，Fast Paxos 算法对 Paxos 算法进行了改进：只允许一个进程提交提案，即该进程具有对 N 的唯一操作权。该方式解决了“活锁”问题。

1.4 ZAB 协议

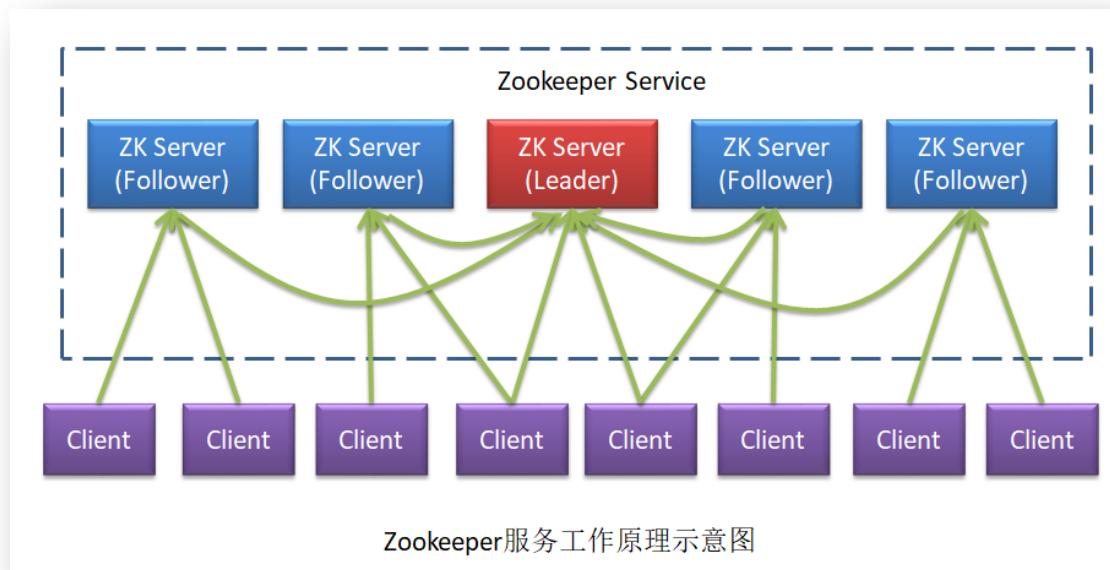
1.4.1 ZAB 协议简介

ZAB，Zookeeper Atomic Broadcast，zk 原子消息广播协议，是专为 ZooKeeper 设计的一种支持崩溃恢复的原子广播协议，在 Zookeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性。

Zookeeper 使用一个单一主进程来接收并处理客户端的所有事务请求，即写请求。当服务器数据的状态发生变更后，集群采用 ZAB 原子广播协议，以事务提案 Proposal 的形式广播到所有的副本进程上。ZAB 协议能够保证一个全局的变更序列，即可以为每一个事务分配一个全局的递增编号 xid。

当 Zookeeper 客户端连接到 Zookeeper 集群的一个节点后，若客户端提交的是读请求，那么当前节点就直接根据自己保存的数据对其进行响应；如果是写请求且当前节点不是 Leader，那么节点就会将该写请求转发给 Leader，Leader 会以提案的方式广播该写操作，只要有超过半数节点同意该写操作，则该写操作请求就会被提交。然后 Leader 会再次广播给

所有订阅者，即 Learner，通知它们同步数据。



1.4.2 ZAB 与 Paxos 的关系

ZAB 协议是 Paxos 算法的一种工业实现算法。但两者的设计目标不太一样。ZAB 协议主要用于构建一个高可用的分布式数据主从系统，即 Follower 是 Leader 的从机，Leader 挂了，马上就可以选举出一个新的 Leader，但平时它们都对外提供服务。而 Fast Paxos 算法则是用于构建一个分布式一致性状态机系统，确保系统中各个节点的状态都是一致的。

1.4.3 三类角色

为了避免 Zookeeper 的单点问题，zk 也是以集群的形式出现的。zk 集群中的角色主要有以下三类：

- **Leader:** 事务请求的唯一处理者，也可以处理读请求。
- **Follower:** 可以直接处理客户端的读请求，并向客户端响应；但其不会处理事务请求，其只会将客户端事务请求转发给 Leader 来处理；对 Leader 发起的事务提案具有表决权；同步 Leader 中的事务处理结果；Leader 选举过程的参与者，具有选举权与被选举权。（就好像正式工）
- **Observer:** 可以理解为不参与 Leader 选举的 Follower，在 Leader 选举过程中没有选举权与被选举权；同时，对于 Leader 的提案没有表决权。用于协助 Follower 处理更多的客户端读请求。**Observer** 的增加，会提高集群读请求处理的吞吐量，但不会增加事务请求的通过压力，不会增加 Leader 选举的压力。（就好像临时工）

Observer 设置为多少合适？是否是越多越好？

这三类角色在不同的情况下又有一些不同的名称：

- Learner: 学习者, 即要从 Leader 中同步数据的 Server, 即 Follower 与 Observer。
 $\text{Learner} = \text{Follower} + \text{Observer}$
- QuorumServer: QuorumPeer, Participant, 法定服务器, 法定主机, 参与者。在集群正常服务状态下, 具有表决权的服务器称为 QuorumServer, 或 QuorumPeer; 在 Leader 选举过程中, 具有选举权与被选举权的服务器, 称为 Participant。
 $\text{QuorumServer} = \text{Leader} + \text{Follower} = \text{Participant}$

1.4.4 三个数据

在 ZAB 中有三个很重要的数据:

- zxid: 其为一个 64 位长度的 Long 类型, 其中高 32 位表示 epoch, 低 32 位表示 xid。
- epoch: (时期、年号) 每个 Leader 选举结束后都会生成一个新的 epoch, 并会通知到集群中所有其它 Server, 包含 Follower 与 Observer。
- xid: 事务 id, 是一个流水号。

1.4.5 三种模式

ZAB 协议中对 zkServer 的状态描述有三种模式。这三种模式并没有十分明显的界线, 它们相互交织在一起。

- 恢复模式: 在集群启动过程中, 或 Leader 崩溃后, 系统都需要进入恢复模式, 以恢复系统对外提供服务的能力。其包含两个重要阶段: Leader 选举与初始化同步。
- 广播模式: 其分为两类: 初始化广播与更新广播。
- 同步模式: 其分为两类: 初始化同步与更新同步。

1.4.6 四种状态

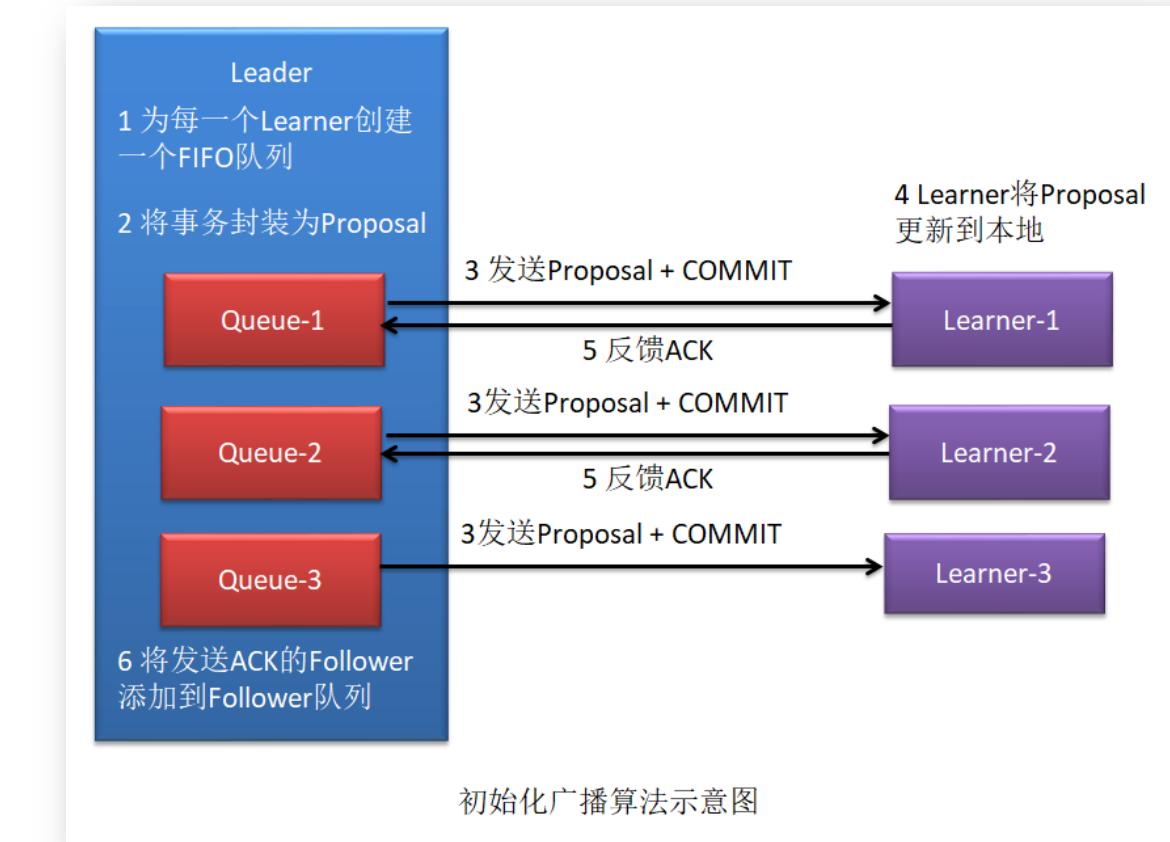
zk 集群中的每一台主机, 在不同的阶段会处于不同的状态。每一台主机具有四种状态。

- LOOKING: 选举状态
- FOLLOWING: Follower 的正常工作状态
- OBSERVING: Observer 的正常工作状态
- LEADING: Leader 的正常工作状态

1.4.7 同步模式与广播模式

(1) 初始广播

前面我们说过, 恢复模式具有两个阶段: Leader 选举与初始化同步(广播)。当完成 Leader 选举后, 此时的 Leader 还是一个准 Leader, 其要经过初始化同步后才能变为真正的 Leader。



具体过程如下：

- 1) 为了保证 Leader 向 Learner 发送提案的有序，Leader 会为每一个 Learner 服务器准备一个队列
- 2) Leader 将那些没有被各个 Learner 同步的事务封装为 Proposal
- 3) Leader 将这些 Proposal 逐条发给各个 Learner，并在每一个 Proposal 后都紧跟一个 COMMIT 消息，表示该事务已经被提交，Learner 可以直接接收并执行
- 4) Learner 接收来自于 Leader 的 Proposal，并将其更新到本地
- 5) 当 Learner 更新成功后，会向准 Leader 发送 ACK 信息
- 6) Leader 服务器在收到来自 Learner 的 ACK 后就会将该 Learner 加入到真正可用的 Follower 列表或 Observer 列表。没有反馈 ACK，或反馈了但 Leader 没有收到的 Learner，Leader 不会将其加入到相应列表。

(2) 消息广播算法



当集群中的 Learner 完成了初始化状态同步，那么整个 zk 集群就进入到了正常工作模式了。

如果集群中的 Learner 节点收到客户端的事务请求，那么这些 Learner 会将请求转发给 Leader 服务器。然后再执行如下的具体过程：

- 1) Leader 接收到事务请求后，为事务赋予一个全局唯一的 64 位自增 id，即 zxid，通过 zxid 的大小比较即可实现事务的有序性管理，然后将事务封装为一个 Proposal。
- 2) Leader 根据 Follower 列表获取到所有 Follower，然后再将 Proposal 通过这些 Follower 的队列将提案发送给各个 Follower。
- 3) 当 Follower 接收到提案后，会先将提案的 zxid 与本地记录的事务日志中的最大的 zxid 进行比较。若当前提案的 zxid 大于最大 zxid，则将当前提案记录到本地事务日志中，并向 Leader 返回一个 ACK。（提问学员）
- 4) 当 Leader 接收到过半的 ACKs 后，Leader 就会向所有 Follower 的队列发送 COMMIT 消息，向所有 Observer 的队列发送 Proposal。
- 5) 当 Follower 收到 COMMIT 消息后，就会将日志中的事务正式更新到本地。当 Observer 收到 Proposal 后，会直接将事务更新到本地。
- 6) 无论是 Follower 还是 Observer，在同步完成后都需要向 Leader 发送成功 ACK。

(3) Observer 的数量问题

Observer 数量一般与 Follower 数量相同。并不是 Observer 越多越好，因为 Observer 数

量的增多虽不会增加事务操作压力，但其需要从 Leader 同步数据，Observer 同步数据的时间是小于等于 Follower 同步数据的时间的。当 Follower 同步数据完成，Leader 的 Observer 列表中的 Observer 主机将结束同步。那些完成同步的 Observer 将会进入到另一个对外提供服务的列表。那么，那些没有同步了数据无法提供服务的 Observer 主机就形成了资源浪费。

所以，对于事务操作发生频繁的系统，不建议使用过多的 Observer。

- Leader 中存在两个关于 Observer 的列表：all（包含所有 Observer）与 service（包含与 Leader 同步过数据的 Observer）
- service 列表是动态变化的。对于没有进入到 service 列表中的 Observer，其会通过心跳与 Leader 进行连接，一旦连接成功，马上就会从 Leader 同步数据，同步完成后向 Leader 发送 ACK。Leader 在接收到其 ACK 后会将其添加到 service 列表。
- 若客户端连接上了不在 service 列表中的 Observer，那么这个 Observer 是不能提供服务的。因为该 Observer 的状态不是 Observing。这个状态是通过 Observer 与 Leader 间的心跳来维护的。
- Leader 中对于 Follower 也同样存在两个列表：all 与 service。其功能与 Observer 的相似。但不同点是，若 Leader 收到的 Follower 同步完成的 ACK 数量没有过半，则认为同步失败，会重新进行广播，让 Follower 重新进行同步。

1.4.8 恢复模式的三个原则

当集群正在启动过程中，或 Leader 崩溃后，集群就进入了恢复模式。对于要恢复的数据状态需要遵循三个原则。

(1) Leader 的主动出让原则

若集群中 Leader 收到的 Follower 心跳数量没有过半，此时 Leader 会自认为自己与集群的连接已经出现了问题，其会主动修改自己的状态为 LOOKING，去查找新的 Leader。为了防止集群出现脑裂。

而其它 Server 由于有过半的主机认为已经丢失了 Leader，所以它们会发起新的 Leader 选举，选出一个新的 Leader。

(2) 已被处理过的消息不能丢原则

当 Leader 收到超过半数 Follower 的 ACKs 后，就向各个 Follower 广播 COMMIT 消息，批准各个 Server 执行该写操作事务。当各个 Server 在接收到 Leader 的 COMMIT 消息后就会在本地执行该写操作，然后会向客户端响应写操作成功。

但是如果在非全部 Follower 收到 COMMIT 消息之前 Leader 就挂了，这将导致一种后果：部分 Server 已经执行了该事务，而部分 Server 尚未收到 COMMIT 消息，所以其并没有执行该事务。当新的 Leader 被选举出，集群经过恢复模式后需要保证所有 Server 上都执行了那些已经被部分 Server 执行过的事务。

(3) 被丢弃的消息不能再现原则

当在 Leader 新事务已经通过，其已经将该事务更新到了本地，但所有 Follower 还都没有收到 COMMIT 之前，Leader 宕机了（比前面叙述的宕机更早），此时，所有 Follower 根本就不知道该 Proposal 的存在。当新的 Leader 选举出来，整个集群进入正常服务状态后，之前挂了的 Leader 主机重新启动并注册成为了 Follower。若那个别人根本不知道的 Proposal 还保留在那个主机，那么其数据就会比其它主机多出了内容，导致整个系统状态的不一致。所以，该 Proposa 应该被丢弃。类似这样应该被丢弃的事务，是不能再次出现在集群中的，应该被清除。

1.4.9 Leader 选举

在集群启动过程中，或 Leader 宕机后，集群就进入了恢复模式。恢复模式中最重要的阶段就是 Leader 选举。

(1) Leader 选举中的基本概念

A、 myid

也称为 ServerId，这是 zk 集群中服务器的唯一标识。例如，有三个 zk 服务器，那么编号分别是 1,2,3。

B、 逻辑时钟

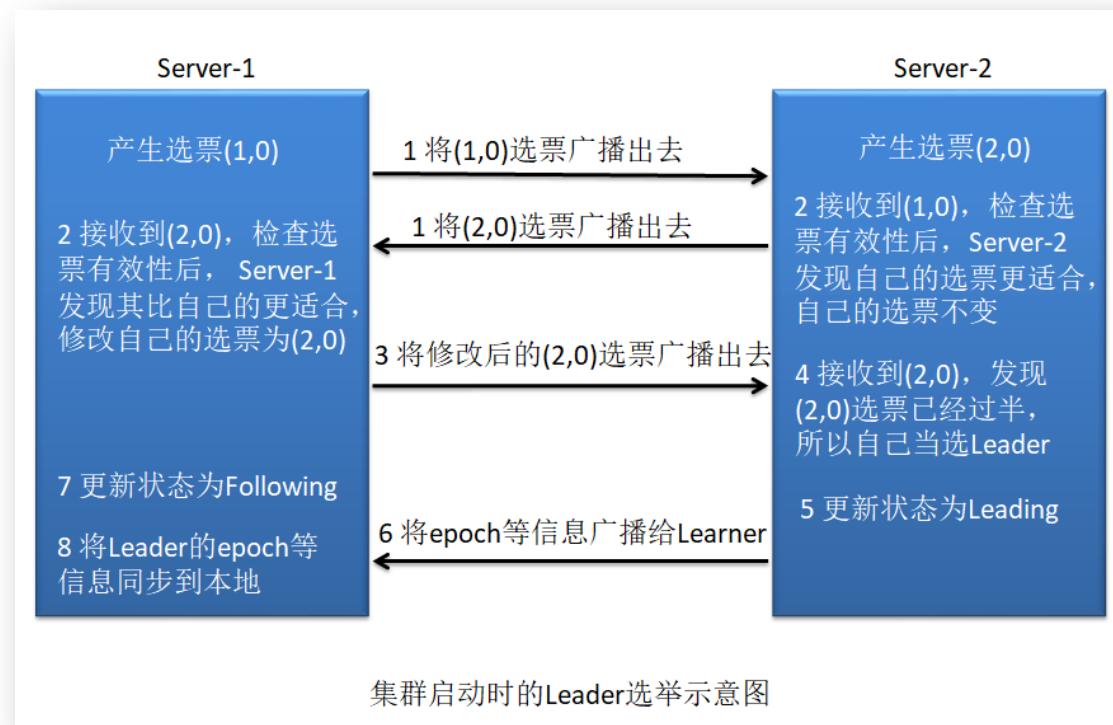
逻辑时钟，Logicalclock，是一个整型数，该概念在选举时称为 logicalclock，而在选举结束后称为 epoch。即 epoch 与 logicalclock 是同一个值，在不同情况下的不同名称。

(2) Leader 选举算法

在集群启动过程中的 Leader 选举过程（算法）与 Leader 断连后的 Leader 选举过程稍微有一些区别，基本相同。

A、 集群启动中的 Leader 选举

若进行 Leader 选举，则至少需要两台主机，这里以三台主机组成的集群为例。



在集群初始化阶段，当第一台服务器 Server1 启动时，其会给自己投票，然后发布自己的投票结果。投票包含所推举的服务器的 myid 和 ZID，使用(myid, ZID)来表示，此时 Server1 的投票为(1, 0)。由于其它机器还没有启动所以它收不到反馈信息，Server1 的状态一直属于 Looking，即属于非服务状态。

当第二台服务器 Server2 启动时，此时两台机器可以相互通信，每台机器都试图找到 Leader，选举过程如下：

(1) 每个 Server 发出一个投票。此时 Server1 的投票为(1, 0)，Server2 的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) 接受来自各个服务器的投票。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自 LOOKING 状态的服务器。

(3) 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行 PK，PK 规则如下：

- 优先检查 ZID。ZID 较大的服务器优先作为 Leader。
- 如果 ZID 相同，那么就比较 myid。myid 较大的服务器作为 Leader 服务器。

对于 Server1 而言，它的投票是(1, 0)，接收 Server2 的投票为(2, 0)。其首先会比较两者 ZID，均为 0，再比较 myid，此时 Server2 的 myid 最大，于是 Server1 更新自己的投票为(2, 0)，然后重新投票。对于 Server2 而言，其无须更新自己的投票，只是再次向集群中所有主机发出上一次投票信息即可。

(4) 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息。对于 Server1、Server2 而言，都统计出集群中已经有两台主机接受了(2, 0)的投票信息，此时便认为已经选出了新的 Leader，即 Server2。

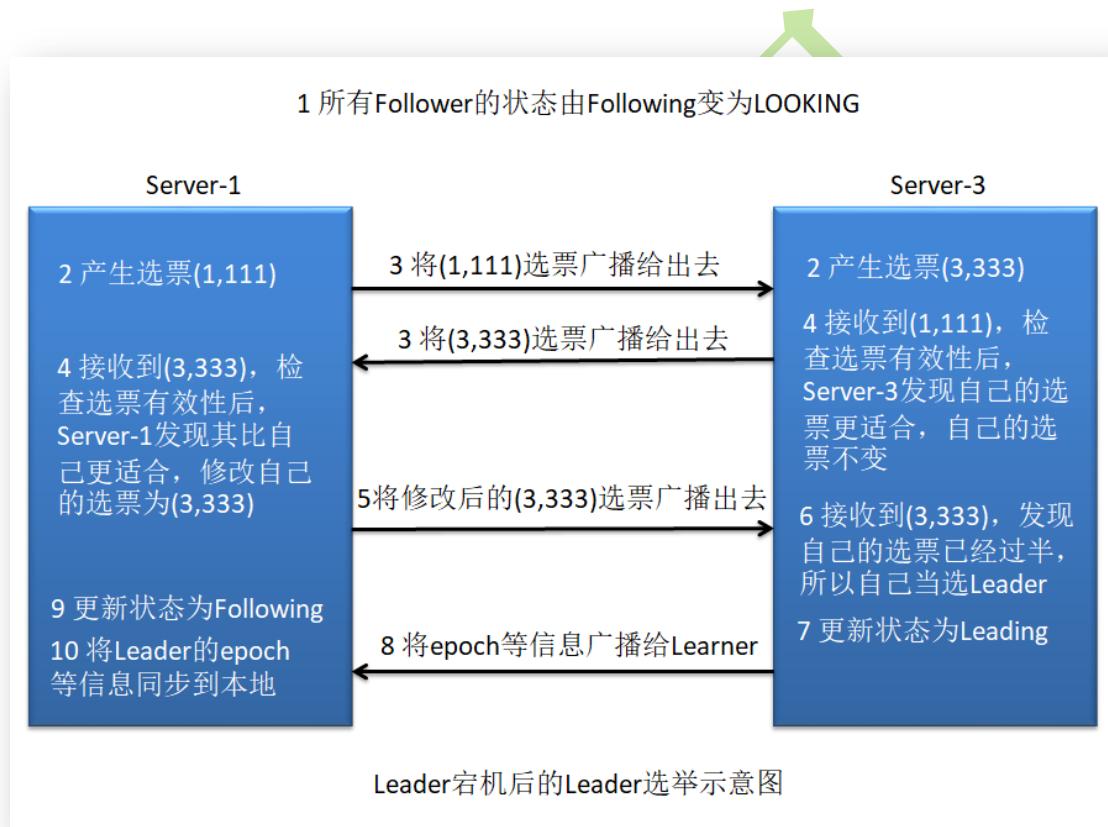
(5) 改变服务器状态。一旦确定了 Leader，每个服务器就会更新自己的状态，如果是

Follower，那么就变更为 FOLLOWING，如果是 Leader，就变更为 LEADING。

(6) 添加主机。在新的 Leader 选举出来后 Server3 启动，其想发出新一轮的选举。但由于当前集群中各个主机的状态并不是 LOOKING，而是各司其职的正常服务，所以其只能是以 Follower 的身份加入到集群中。

B、宕机后的 Leader 选举

在 Zookeeper 运行期间，Leader 与非 Leader 服务器各司其职，即便当有非 Leader 服务器宕机或新加入时也不会影响 Leader。但是若 Leader 服务器挂了，那么整个集群将暂停对外服务，进入新一轮的 Leader 选举，其过程和启动时期的 Leader 选举过程基本一致。



假设正在运行的有 Server1、Server2、Server3 三台服务器，当前 Leader 是 Server2，若某一时刻 Server2 挂了，此时便开始新一轮的 Leader 选举了。选举过程如下：

(1) 变更状态。Leader 挂后，余下的非 Observer 服务器都会将自己的服务器状态由 FOLLOWING 变更为 LOOKING，然后开始进入 Leader 选举过程。

(2) 每个 Server 会发出一个投票，仍然会首先投自己。不过，在运行期间每个服务器上的 ZID 可能是不同，此时假定 Server1 的 ZID 为 111，Server3 的 ZID 为 333；在第一轮投票中，Server1 和 Server3 都会投自己，产生投票(1, 111), (3, 333)，然后各自将投票发送给集群中所有机器。

(3) 接收来自各个服务器的投票。与启动时过程相同。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自 LOOKING 状态的服务器。

(4) 处理投票。与启动时过程相同。针对每一个投票，服务器都需要将别人的投票和自己的投票进行 PK。对于 Server1 而言，它的投票是(1, 111)，接收 Server3 的投票为(3, 333)。

其首先会比较两者的 ZXID，Server3 投票的 zxid 为 333 大于 Server1 投票的 zxid 的 111，于是 Server1 更新自己的投票为(3, 333)，然后重新投票。对于 Server3 而言，其无须更新自己的投票，只是再次向集群中所有主机发出上一次投票信息即可。

(5) 统计投票。与启动时过程相同。对于 Server1、Server2 而言，都统计出集群中已经有两台主机接受了(3, 333)的投票信息，此时便认为已经选出了新的 Leader，即 Server3。

(6) 改变服务器的状态。与启动时过程相同。一旦确定了 Leader，每个服务器就会更新自己的状态。Server1 变更为 FOLLOWING，Server3 变更为 LEADING。

1.5 高可用集群的容灾

1.5.1 服务器数量的奇数与偶数

前面我们说过，无论是写操作投票，还是 Leader 选举投票，都必须过半才能通过，也就是说若出现超过半数的主机宕机，则投票永远无法通过。基于该理论，由 5 台主机构成的集群，最多只允许 2 台宕机。而由 6 台构成的集群，其最多也只允许 2 台宕机。即，6 台与 5 台的容灾能力是相同的。基于此容灾能力的原因，建议使用奇数台主机构成集群，以避免资源浪费。

但从系统吞吐量上说，6 台主机的性能一定是高于 5 台的。所以使用 6 台主机并不是资源浪费。

1.5.2 容灾设计方案

对于一个高可用的系统，除了要设置多台主机部署为一个集群避免单点问题外，还需要考虑将集群部署在多个机房、多个楼宇。对于多个机房、楼宇中集群也是不能随意部署的，下面就多个机房的部署进行分析。

在多机房部署设计中，要充分考虑“过半原则”，也就是说，尽量要确保 zk 集群中有过半的机器能够正常运行。

(1) 三机房部署

在生产环境下，三机房部署是最常见的、容灾性最好的部署方案。

三机房部署中要求每个机房中的主机数量必须少于集群总数的一半。这样可以保证，三个机房中若有一个机房断电或断网，其它两个机房中的机器总数仍是过半的，集群仍可以正常对外提供服务。当然，若两个机房出现了问题，那么整个集群就瘫痪了。这种情况出现的概率要远低于一个机房出问题的情况。

(2) 双机房部署

zk 官网没有给出较好的双机房部署的容灾方案。只能是让其中一个机房占有超过半数的主机，使其做为主机房，而另一机房少于半数。当然，若主机房出现问题，则整个集群会

瘫痪。

1.6 CAP 定理

1.6.1 简介

CAP 定理指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性），三者不可兼得。

- 一致性 (C): 分布式系统中多个主机之间是否能够保持数据一致的特性。即，当系统数据发生更新操作后，各个主机中的数据仍然处于一致的状态。
- 可用性 (A): 系统提供的服务必须一直处于可用的状态，即对于用户的每一个请求，系统总是可以在**有限的时间内**对用户做出**响应**。
- 分区容错性 (P): 分布式系统在遇到任何网络分区故障时，仍能够保证对外提供满足一致性和可用性的服务。

对于分布式系统，网络环境相对是不可控的，出现网络分区是不可避免的，因此系统必须具备分区容错性。但其并不能同时保证一致性与可用性。CAP 原则对于一个分布式系统来说，只可能满足两项，即要么 CP，要么 AP。

1.6.2 BASE 理论

BASE 是 Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent（最终一致性）三个短语的简写。是 CAP 定理对于一致性与可用性权衡的结果。

BASE 理论的核心思想是：即使无法做到强一致性，但每个系统都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。

(1) 基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。

响应时间的损失：

功能上的损失：

(2) 软状态

软状态，是指允许系统数据存在的中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统主机间进行数据同步的过程存在一定延时。软状态，其实就是一种灰度状态，过渡状态。

(3) 最终一致性

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

1.6.3 ZK 与 CP

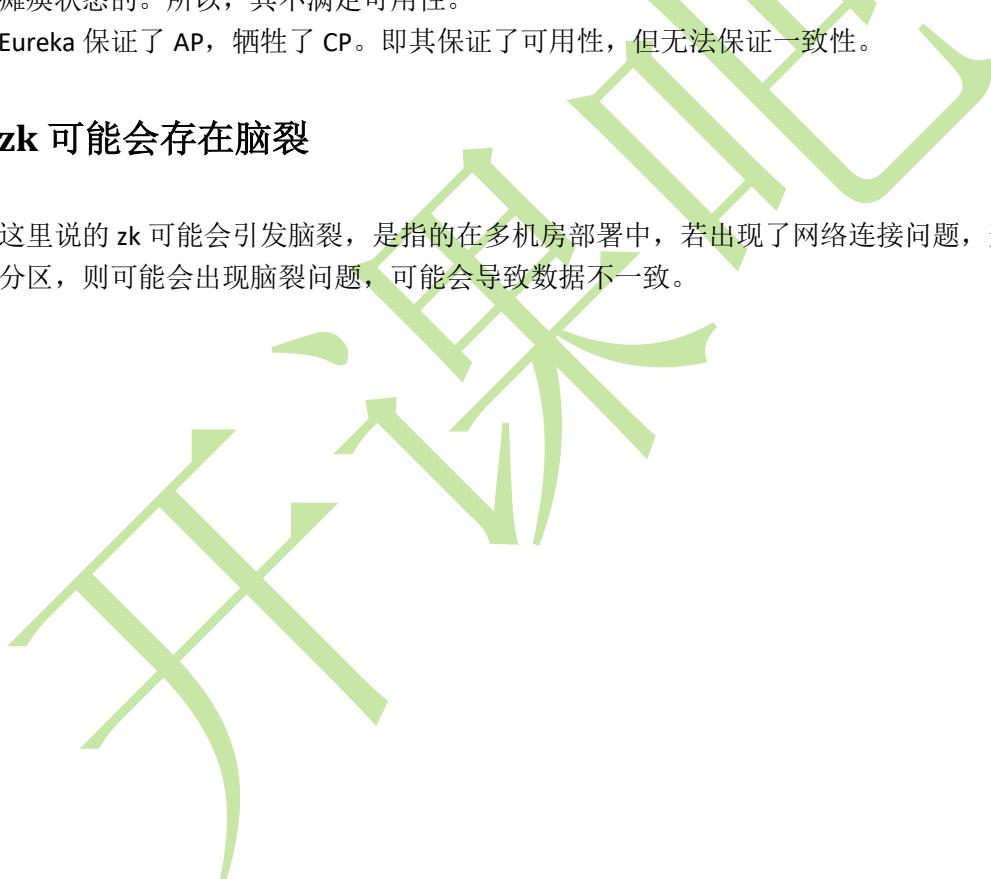
zk 遵循的是 CP 原则，即保证了一致性，但牺牲了可用性。体现在哪里呢？

当 Leader 宕机后，zk 集群会马上进行新的 Leader 的选举。但选举时长一般在 200 毫秒内，最长不超过 60 秒，整个选举期间 zk 集群是不接受客户端的读写操作的，即 zk 集群是处于瘫痪状态的。所以，其不满足可用性。

Eureka 保证了 AP，牺牲了 CP。即其保证了可用性，但无法保证一致性。

1.7 zk 可能会存在脑裂

这里说的 zk 可能会引发脑裂，是指的在多机房部署中，若出现了网络连接问题，形成多个分区，则可能会出现脑裂问题，可能会导致数据不一致。



第2章 Leader 的选举机制

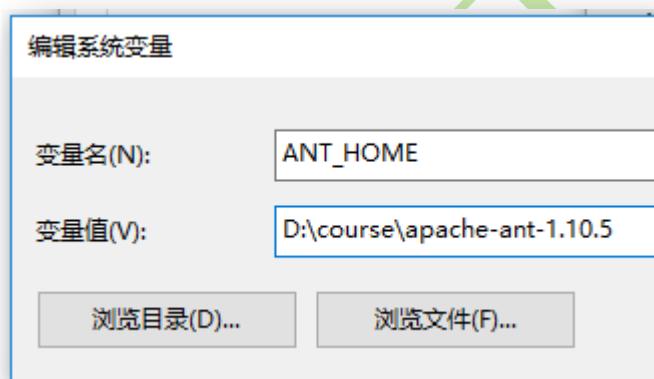
2.1 将 zookeeper 源码导入到 Idea

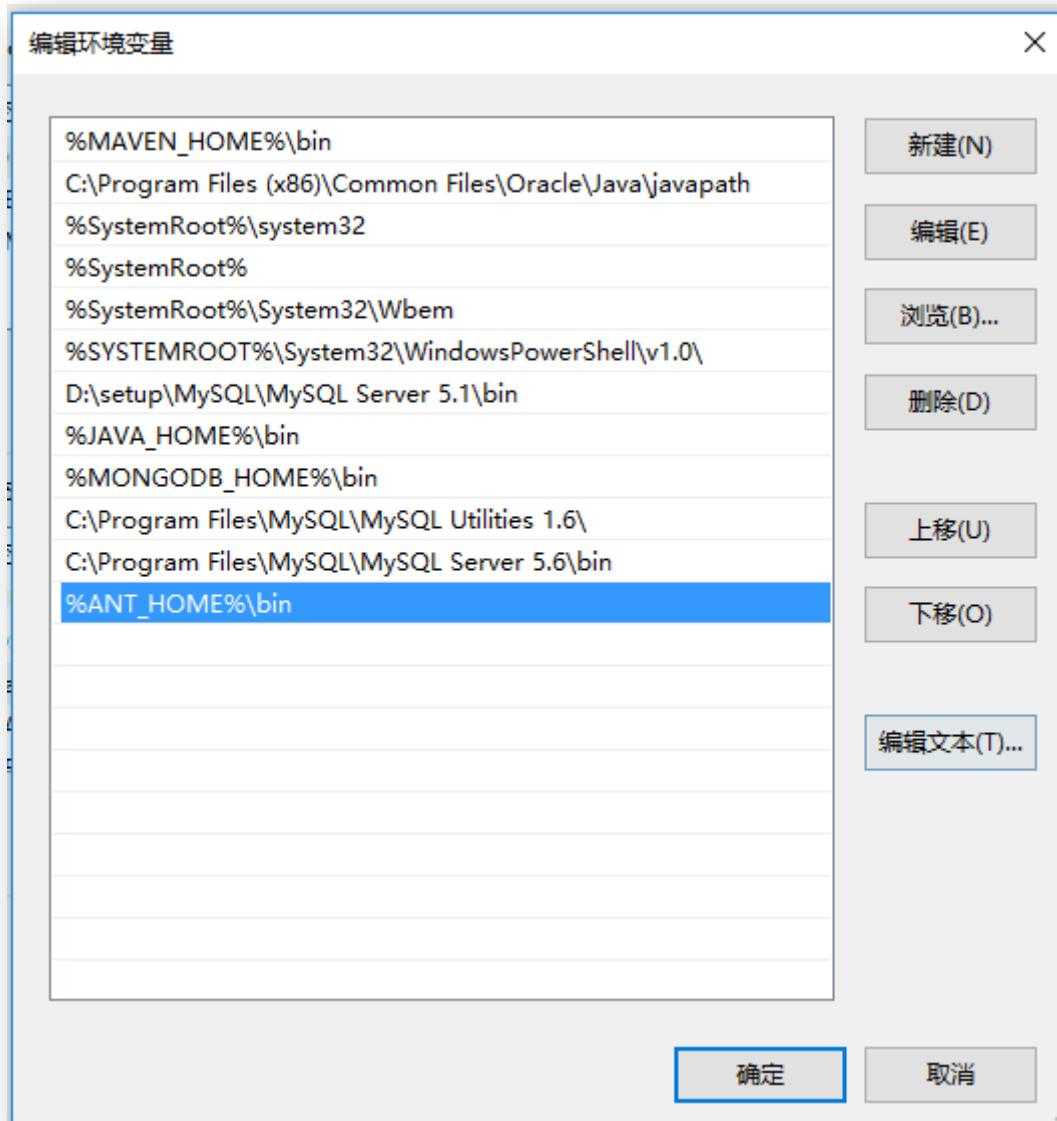
2.1.1 下载并安装 Ant

(1) 下载 Ant

Ant 官网：<http://ant.apache.org>

(2) 安装配置 Ant





在命令行窗口的任意目录下执行 `Ant -version` 命令，可以看到版本号，则说明 Ant 安装成功。



```
C:\Windows\system32\cmd.exe
C:\Users\Rey>ant -version
Apache Ant(TM) version 1.10.5 compiled on July 10 2018
C:\Users\Rey>
```

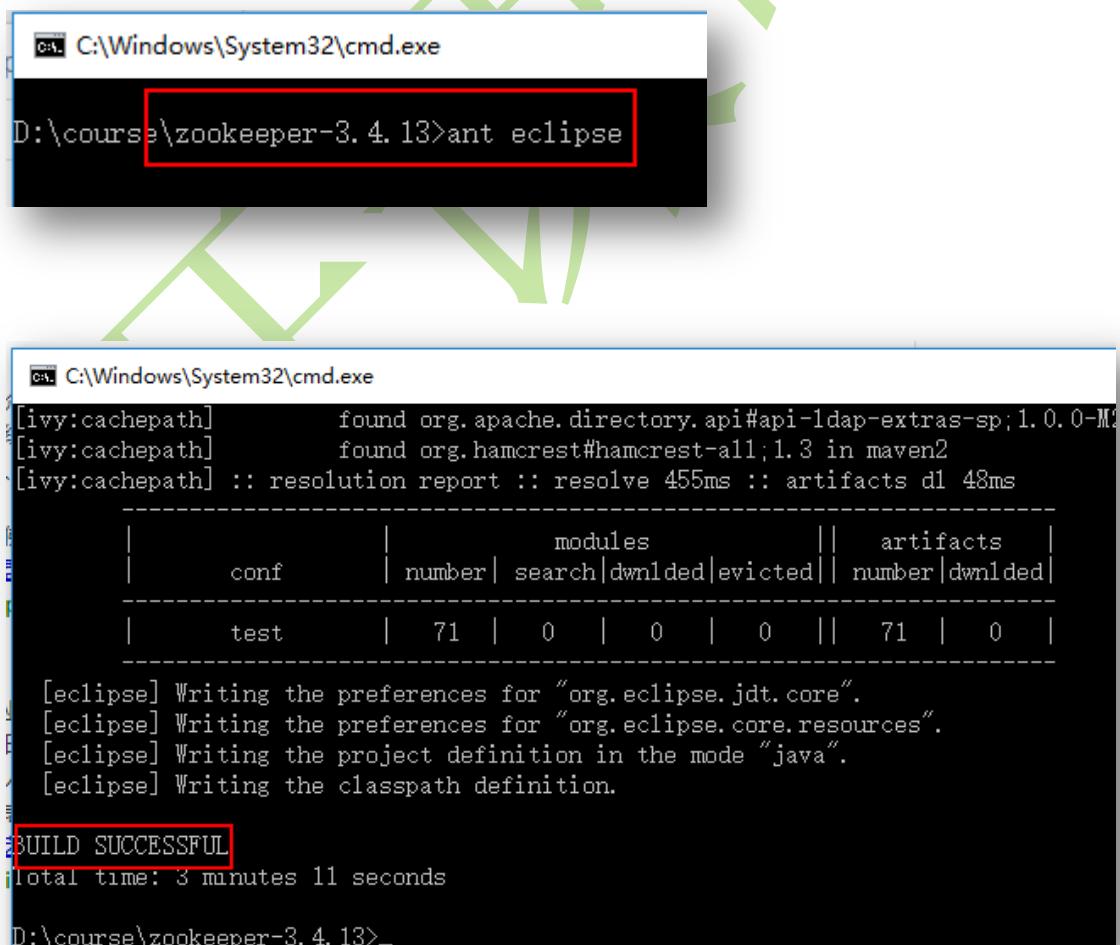
2.1.2 构建 Eclipse 工程

在命令行窗口中进入到 zk 解压目录，执行 `ant eclipse` 命令。其会根据 `build.xml` 文件对

当前的 zookeeper 源码进行构建，将其构建为一个 Eclipse 工程。不过，对于该构建过程，其 JDK 最好能够满足 build.xml 文件的要求。打开 build.xml 文件，可以看到 zk 源码是使用 JDK6 编写并编译的，所以最好使用该指定的 JDK，这样可以保证将来构建出的 Eclipse 工程中没有错误。

```
79      <property name="revision.properties" value="revisi
80      <property file="${basedir}/src/java/${revision.pro
81
82      <property name="javac.target" value="1.6" />
83      <property name="javac.source" value="1.6" />
84
85      <property name="src.dir" value="${basedir}/src" />
86      <property name="java.src.dir" value="${src.dir}/ja
87
```

不过，由于我们这里仅仅跟踪 Leader 的选举算法，本机使用的是 JDK8，构建出的该 Eclipse 工程中的相关代码并没有报错，所以这里就不更换 JDK 版本了。



The screenshot shows two command-line windows. The top window is titled 'C:\Windows\System32\cmd.exe' and contains the command 'D:\course\zookeeper-3.4.13>ant eclipse'. The bottom window is also titled 'C:\Windows\System32\cmd.exe' and displays the build log. The log includes Ivy resolution details, Eclipse preference writing, project definition creation, and finally 'BUILD SUCCESSFUL'.

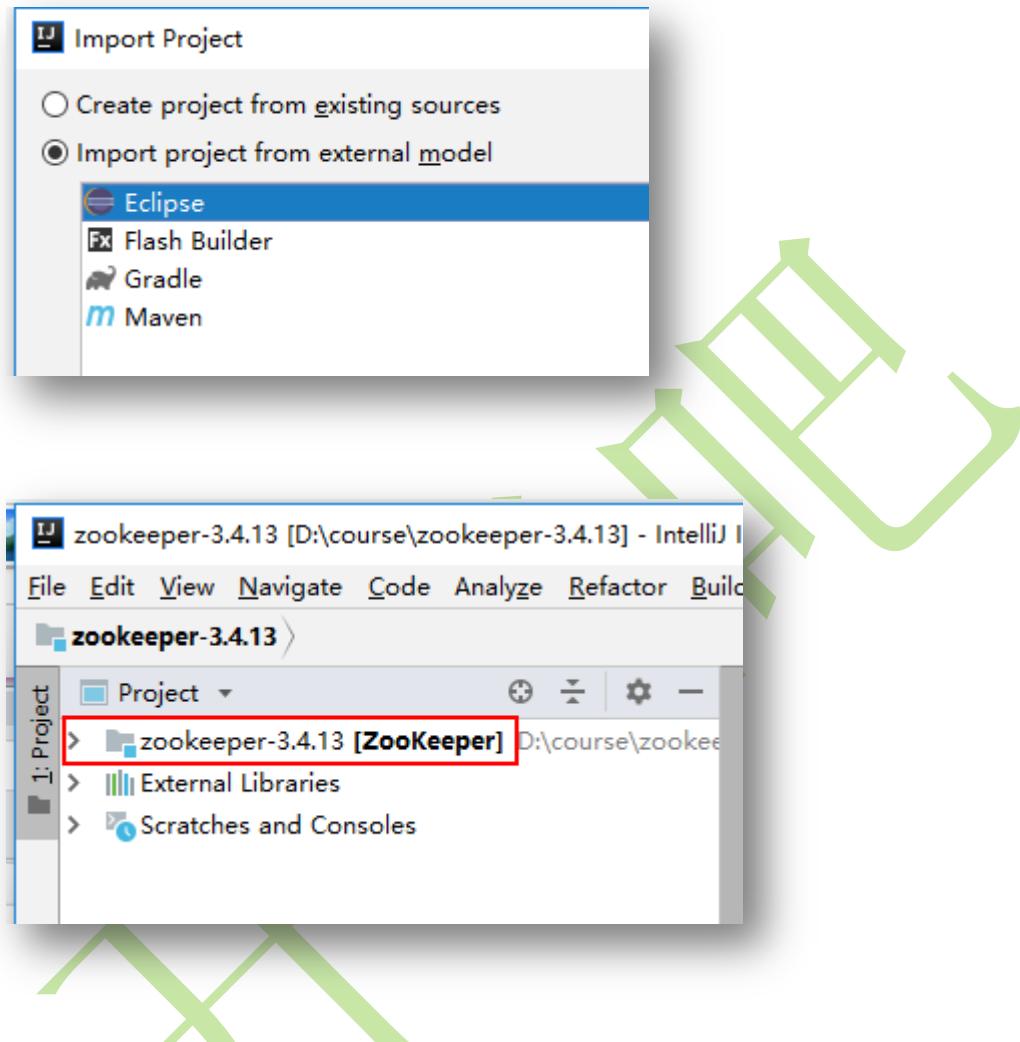
```
[ivy:cachepath]          found org.apache.directory.api#api-ldap-extras-sp;1.0.0-M2
[ivy:cachepath]          found org.hamcrest#hamcrest-all;1.3 in maven2
[ivy:cachepath] :: resolution report :: resolve 455ms :: artifacts dl 48ms
-----
|           |           modules   ||   artifacts  | | | | | |
|       conf    |   number| search|dwnlded|evicted||   number|dwnlded|
|           |       test   |   71   |   0    |   0    |   0    ||   71   |   0    |
-----
[eclipse] Writing the preferences for "org.eclipse.jdt.core".
[eclipse] Writing the preferences for "org.eclipse.core.resources".
[eclipse] Writing the project definition in the mode "java".
[eclipse] Writing the classpath definition.

[BUILD SUCCESSFUL]
Total time: 3 minutes 11 seconds

D:\course\zookeeper-3.4.13>
```

2.1.3 导入到 Idea

打开 Idea，选择导入工程，找到 zk 的源码解压目录，直接导入。



2.2 选举算法源码中的总思路

Zookeeper 的 Leader 选举类是 `FastLeaderElection`, 该类是 ZAB 协议在 Leader 选举中的工程应用，所以直接找到该类对其进行分析。该类中的最为重要的方法为 `lookForLeader()`, 是选举 Leader 的核心方法。该方法大体思路可以划分为以下几块：

2.2.1 选举前的准备工作

选举前需要做一些准备工作，例如，创建选举对象、创建选举过程中需要用到的集合、初始化选举时限等。

2.2.2 将自己作为初始化 Leader 投出去

在当前 Server 第一次投票时会先将自己作为 Leader，然后将自己的选票广播给其它所有 Server。

2.2.3 验证自己的投票与大家的投票谁更适合做 Leader

在“我选我”后，当前 Server 同样会接收到其它 Server 发送来的选票通知(Notification)。通过 while 循环，遍历所有接收到的选票通知，比较谁更适合做 Leader。若找到一个比自己更适合的 Leader，则修改自己选票，重新将新的选票广播出去。当然，每验证一个选票，则会将其记录到一个集合中，将来用于进行票数统计。

2.2.4 判断本轮选举是否应结束

其实在每次验证过谁更适合做 Leader 后，就会马上判断当前的选举是否可以结束了，即当前主机所推荐的这个选票是否过半了。若过半了，则直接完成后续的一些收尾工作，例如清空选举过程中所使用的集合，以备下次使用；再例如，生成最终的选票，以备其它 Server 来同步数据。若没有过半，则继续从队列中读取出下一个来自于其它主机的选票，然后进行验证。

2.2.5 无需选举的情况

对一些特殊情况的处理。

2.3 源码解读

需要注意，对源码的阅读主要包含两方面。一个是对重要类、重要成员变量、重要方法的注释的阅读；一个是对重要方法的业务逻辑的分析。

第3章 Zookeeper 的安装与集群搭建

3.1 安装单机 Zookeeper

3.1.1 下载 Zookeeper 安装包

在 <http://zookeeper.apache.org> 官网下载。

3.1.2 上传安装包

将下载的 Zookeeper 安装包上传到 zk04 主机的 /usr/tools 目录。

3.1.3 安装配置 zk

(1) 解压安装包

```
zk04 ~ [root@zk04 ~]# ll /usr/tools
total 203360
-rw-r--r-- 1 root root 185515842 Aug 14 10:08 jdk-8u144-linux-x64.tar.gz
-rw-r--r-- 1 root root 22724574 Oct  2 10:44 zookeeper-3.4.9.tar.gz
[root@zk04 ~]# tar -xzvf /usr/tools/zookeeper-3.4.9.tar.gz -C /usr/apps
```

(2) 创建软链接

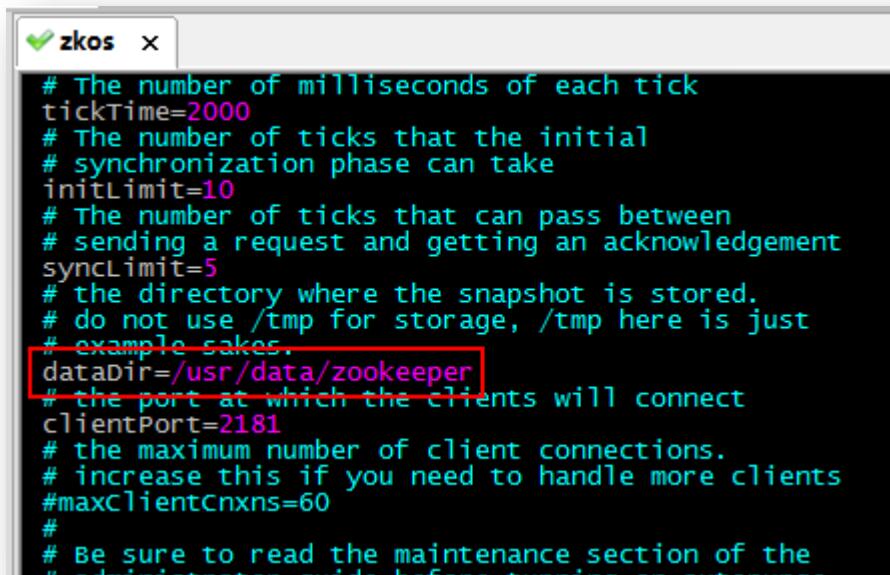
```
zk04 ~]# ll /usr/apps
total 8
lwxrwxrwx. 1 root root 23 Aug 15 01:24 jdk -> /usr/apps/jdk1.8.0_144/
drwxr-xr-x. 8 uucp 143 4096 Jul 22 13:11 jdk1.8.0_144
drwxr-xr-x. 10 1001 1001 4096 Aug 23 2016 zookeeper-3.4.9
[root@zk04 ~]#
[root@zk04 ~]# ln -s /usr/apps/zookeeper-3.4.9/ /usr/apps/zk
[root@zk04 ~]#
[root@zk04 ~]# ll /usr/apps
total 8
lwxrwxrwx. 1 root root 23 Aug 15 01:24 jdk -> /usr/apps/jdk1.8.0_144/
drwxr-xr-x. 8 uucp 143 4096 Jul 22 13:11 jdk1.8.0_144
lwxrwxrwx. 1 root root 26 Oct 3 00:37 zk -> /usr/apps/zookeeper-3.4.9/
drwxr-xr-x. 10 1001 1001 4096 Aug 23 2016 zookeeper-3.4.9
[root@zk04 ~]#
```

(3) 复制配置文件

复制 Zookeeper 安装目录下的 conf 目录中的 zoo_sample.cfg 文件，并命名为 zoo.cfg。

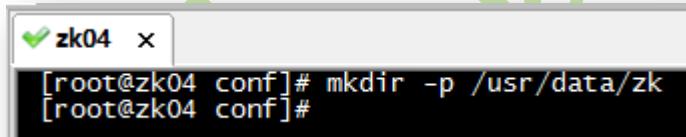
```
zk04 conf]# ll
total 12
-rw-rw-r--. 1 1001 1001 535 Aug 23 2016 configuration.xls
-rw-rw-r--. 1 1001 1001 2161 Aug 23 2016 log4j.properties
-rw-rw-r--. 1 1001 1001 922 Aug 23 2016 zoo_sample.cfg
[root@zk04 conf]#
[root@zk04 conf]# pwd
/usr/apps/zk/conf
[root@zk04 conf]# cp zoo_sample.cfg zoo.cfg
[root@zk04 conf]#
[root@zk04 conf]# ll
total 16
-rw-rw-r--. 1 1001 1001 535 Aug 23 2016 configuration.xls
-rw-rw-r--. 1 1001 1001 2161 Aug 23 2016 log4j.properties
-rw-r--r--. 1 root root 922 Oct 3 00:43 zoo.cfg
-rw-rw-r--. 1 1001 1001 922 Aug 23 2016 zoo_sample.cfg
[root@zk04 conf]#
```

(4) 修改配置文件



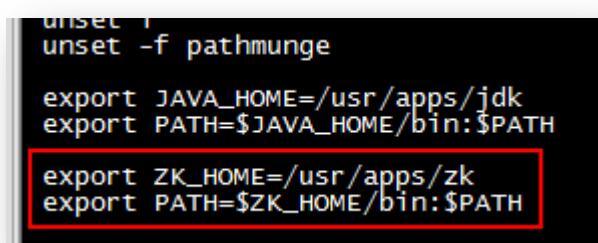
```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/usr/data/zookeeper
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# Be sure to read the maintenance section of the
# configuration guide before running ZooKeeper!
```

(5) 新建数据存放目录



```
[root@zk04 conf]# mkdir -p /usr/data/zk
[root@zk04 conf]#
```

(6) 注册 bin 目录

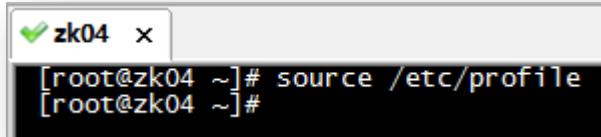


```
unset IFS
unset -f pathmunge

export JAVA_HOME=/usr/apps/jdk
export PATH=$JAVA_HOME/bin:$PATH

export ZK_HOME=/usr/apps/zk
export PATH=$ZK_HOME/bin:$PATH
```

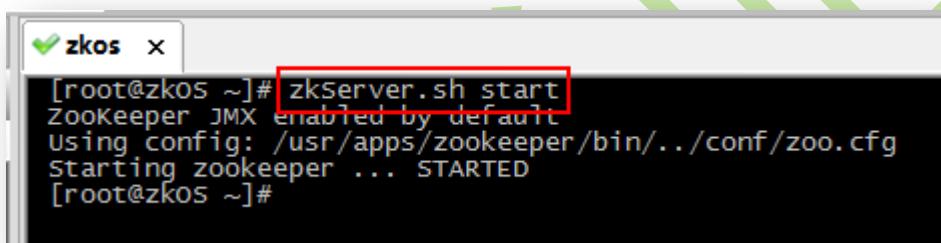
(7) 重新加载 profile 文件



```
[root@zk04 ~]# source /etc/profile  
[root@zk04 ~]#
```

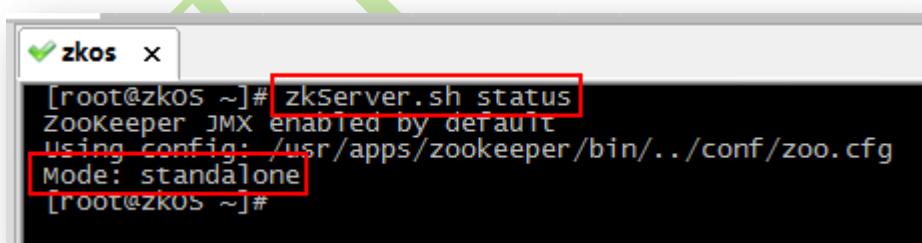
3.1.4 操作 zk

(1) 开启 zk



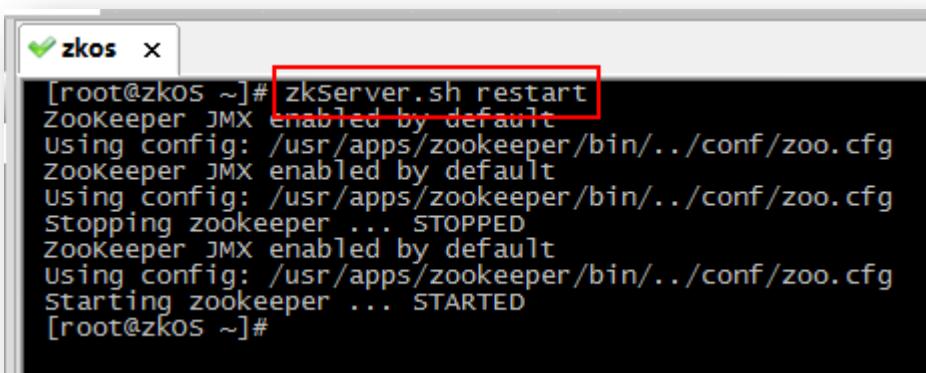
```
[root@zkos ~]# zkServer.sh start  
ZooKeeper JMX enabled by default  
using config: /usr/apps/zookeeper/bin/../conf/zoo.cfg  
starting zookeeper ... STARTED  
[root@zkos ~]#
```

(2) 查看状态



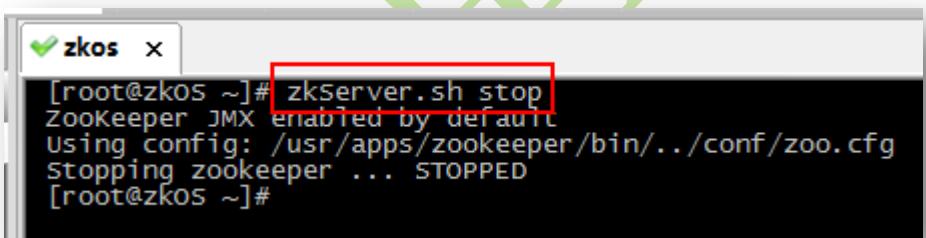
```
[root@zkos ~]# zkServer.sh status  
ZooKeeper JMX enabled by default  
using config: /usr/apps/zookeeper/bin/../conf/zoo.cfg  
Mode: standalone  
[root@zkos ~]#
```

(3) 重启 zk



```
[root@zkos ~]# zkserver.sh restart
ZooKeeper JMX enabled by default
Using config: /usr/apps/zookeeper/bin/../conf/zoo.cfg
ZooKeeper JMX enabled by default
Using config: /usr/apps/zookeeper/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
ZooKeeper JMX enabled by default
Using config: /usr/apps/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@zkos ~]#
```

(4) 停止 zk



```
[root@zkos ~]# zkServer.sh stop
ZooKeeper JMX enabled by default
Using config: /usr/apps/zookeeper/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@zkos ~]#
```

3.2 搭建 Zookeeper 集群

下面要搭建一个由四台 zk 构成的 zk 集群，其中一台为 Leader，两台 Follower，一台 Observer。

3.2.1 克隆并配置第一台主机

(1) 克隆并配置主机

克隆前面单机 Zookeeper 主机后，要修改如下配置文件：

- 修改主机名：/etc/hostname
- 修改网络配置：/etc/sysconfig/network-scripts/ifcfg-ens33

(2) 创建 myid 文件

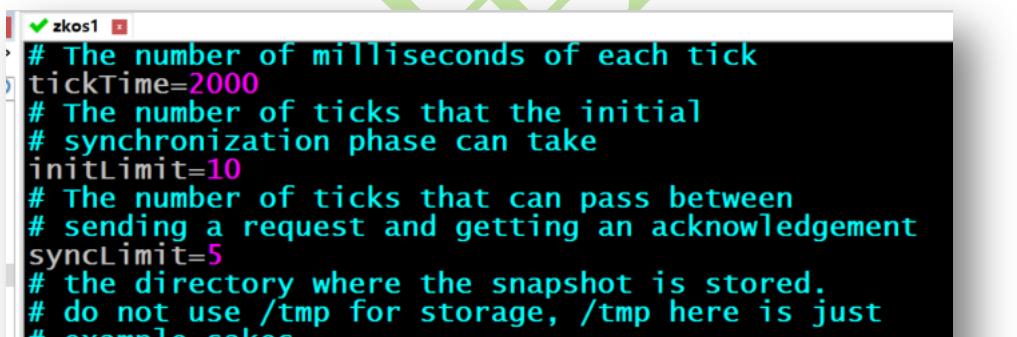
在/usr/data/zookeeper 目录中创建表示当前主机编号的 myid 文件，该编号为当前主机在集群中的唯一标识。



```
[root@zkos1 ~]# echo 1 > /usr/data/zookeeper/myid
[root@zkos1 ~]#
[root@zkos1 ~]# cat /usr/data/zookeeper/myid
1
[root@zkos1 ~]#
```

(3) 修改 zoo.cfg

在 zoo.cfg 文件中添加 zk 集群节点列表。



```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/usr/data/zookeeper
# the port at which the clients will connect
clientPort=2181

server.1=192.168.79.13:2888:3888
server.2=192.168.79.14:2888:3888
server.3=192.168.79.15:2888:3888
server.4=192.168.79.16:2888:3888:observer

# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns 60
```

3.2.2 克隆并配置另两台主机

克隆并配置另外两台主机的方式是相同的，下面以 zkos2 为例。

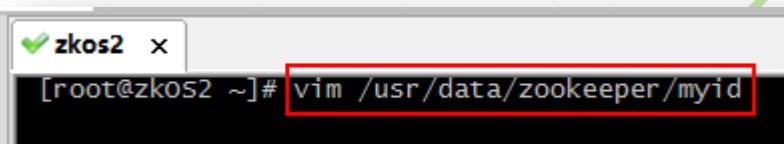
(1) 克隆主机

克隆前面 zk0s1 主机后，要修改如下配置文件：

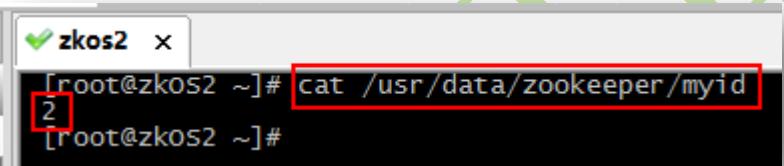
- 修改主机名：/etc/hostname
- 修改网络配置：/etc/sysconfig/network-scripts/ifcfg-ens33

(2) 修改 myid

修改 myid 的值与 zoo.cfg 中指定的主机编号相同。



```
[root@zkos2 ~]# vim /usr/data/zookeeper/myid
```



```
[root@zkos2 ~]# cat /usr/data/zookeeper/myid  
2  
[root@zkos2 ~]#
```

3.2.3 克隆并配置第四台主机

第四台主机即为要作 Observer 的主机。

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/usr/data/zookeeper
# the port at which the clients will connect
clientPort=2181
peerType=observer

server.1=192.168.79.13:2888:3888
server.2=192.168.79.14:2888:3888
server.3=192.168.79.15:2888:3888
server.4=192.168.79.16:2888:3888:observer

# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
```

3.2.4 启动 zk 集群

使用 zkServer.sh start 命令，逐个启动每一个 Zookeeper 节点主机。

3.3 伪集群的搭建

这里要搭建的集群与前面的集群相同，都由四台服务器组成，其中第四台为 Observer。伪集群的搭建与真实集群的搭建差不多。其主机步骤分为四大步。

3.3.1 复制配置文件

这里需要四个配置文件，都存放在 zk 安装目录的 conf 目录中。所以它们的文件名肯定是不同的。

```
✓ zkArtificialclusterOS conf]# ll
总用量 32
-rw-rw-r-- 1 2002 2002 535 3月 7 2019 configuration.xls
-rw-rw-r-- 1 2002 2002 2161 3月 7 2019 log4j.properties
-rw-r--r-- 1 root root 1076 9月 7 09:53 zoo1.cfg
-rw-r--r-- 1 root root 1076 9月 7 09:54 zoo2.cfg
-rw-r--r-- 1 root root 1076 9月 7 09:54 zoo3.cfg
-rw-r--r-- 1 root root 1095 9月 7 09:54 zoo4.cfg
-rw-r--r-- 1 root root 927 9月 3 19:08 zoo.cfg
-rw-rw-r-- 1 2002 2002 922 3月 7 2019 zoo_sample.cfg
[root@zkArtificialclusterOS conf]#
```

3.3.2 修改配置文件内容

以下是第四台 Server 的配置文件。

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/usr/data/zookeeper4
# the port at which the clients will connect
clientPort=2184
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
peerType=observer

server.1=192.168.59.157:2666:3666
server.2=192.168.59.157:2777:3777
server.3=192.168.59.157:2888:3888
server.4=192.168.59.157:2999:3999:observer
```

另外三个配置文件中指定的 dataDir 分别是 /usr/data/zookeeper1、/usr/data/zookeeper2、/usr/data/zookeeper3。clientPort 分别是 2181、2182、2183。当然，另外三个配置文件中不

要设置 `peerType=observer`。

3.3.3 创建数据目录

在`/usr/data` 下创建如下四个目录。



```
[root@zkArtificialClusterOS data]# ll
总用量 0
drwxr-xr-x 3 root root 51 9月    7 07:51 zookeeper
drwxr-xr-x 3 root root 63 9月    7 09:57 zookeeper1
drwxr-xr-x 3 root root 35 9月    7 09:59 zookeeper2
drwxr-xr-x 3 root root 63 9月    7 09:58 zookeeper3
drwxr-xr-x 3 root root 63 9月    7 09:58 zookeeper4
[root@zkArtificialClusterOS data]#
```

3.3.4 创建 myid 文件

分别在 `zookeeper1`、`zookeeper2`、`zookeeper3`、`zookeeper4` 四个目录中创建四个 `myid` 文件，内容分别为 1、2、3、4。



```
[root@zkArtificialClusterOS data]# cat zookeeper1/myid
1
[root@zkArtificialClusterOS data]#
```

3.3.5 集群启动

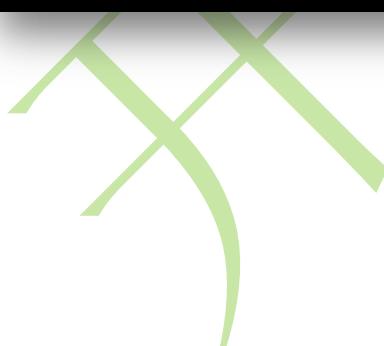
伪集群的启动需要指定每台 Server 启动所使用的配置文件。进入到 zk 的安装目录。

```
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh start conf/zoo1.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo1.cfg
Starting zookeeper ... STARTED
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh start conf/zoo2.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo2.cfg
Starting zookeeper ... STARTED
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh start conf/zoo3.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo3.cfg
Starting zookeeper ... STARTED
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh start conf/zoo4.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo4.cfg
Starting zookeeper ... STARTED
[root@zkArtificialClusterOS zookeeper]#
```

查看各个 Server 的状态。



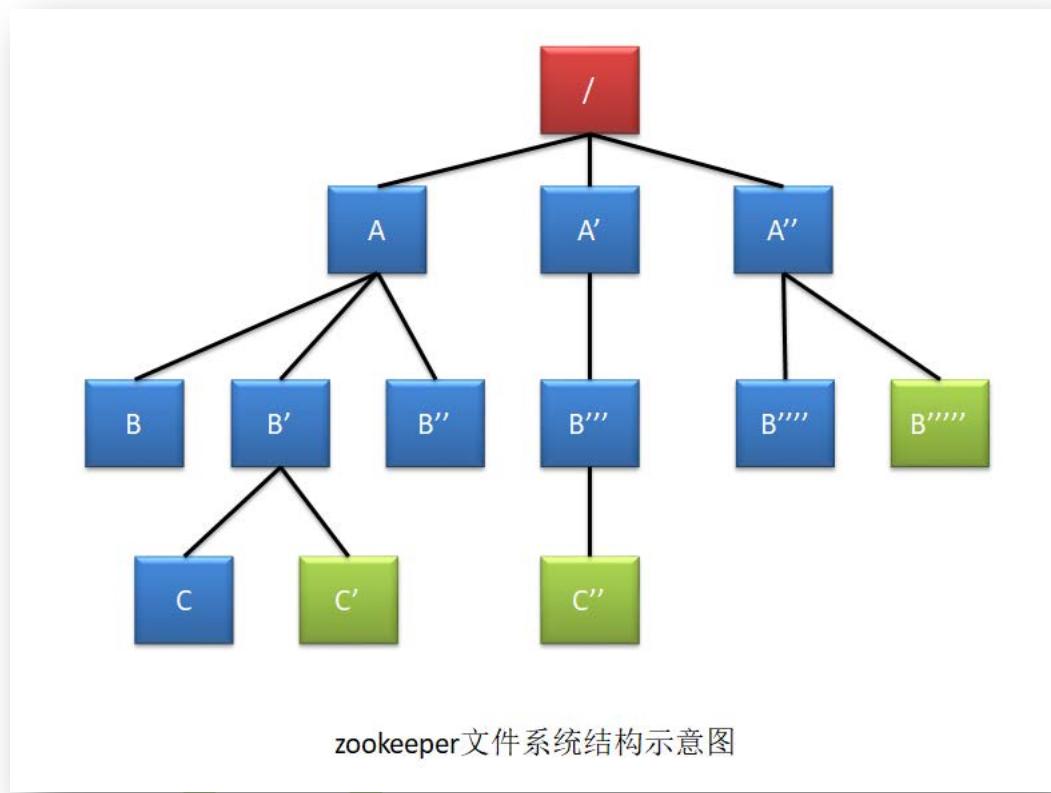
```
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh status conf/zoo1.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo1.cfg
Mode: leader
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh status conf/zoo2.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo2.cfg
Mode: follower
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh status conf/zoo3.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo3.cfg
Mode: follower
[root@zkArtificialClusterOS zookeeper]# bin/zkServer.sh status conf/zoo4.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo4.cfg
Mode: observer
[root@zkArtificialClusterOS zookeeper]#
```



第4章 Zookeeper 技术内幕

4.1 重要理论

4.1.1 数据模型 znode



zk 数据存储结构与标准的 Unix 文件系统非常相似，都是在根节点下挂很多子节点。zk 中没有引入传统文件系统中目录与文件的概念，而是使用了称为 znode 的数据节点概念。znode 是 zk 中数据的最小单元，每个 znode 上都可以保存数据，同时还可以挂载子节点，形成一个树形化命名空间。ZOO_USER_CFG

(1) 节点类型

- 每个 znode 根据节点类型的不同，具有不同的生命周期。
- **持久节点：**节点被创建后会一直保存在 zk 中，直到将其删除。
- **持久顺序节点：**一个父节点可以为它的第一级子节点维护一份顺序，用于记录每个子节点创建的先后顺序。其在创建子节点时，会在子节点名称后添加 10 个十进制数字（从 0 开始计数），作为该子节点的完整节点名。

- **临时节点:** 临时节点的生命周期与客户端的会话绑定在一起，会话消失则该节点就会被自动清理。临时节点不能有子节点，只能做叶子节点。
- **临时顺序节点:** 添加了创建序号的临时节点。

(2) 节点状态

- **cZxid:** Created Zxid，表示当前 znode 被创建时的事务 ID
- **ctime:** Created Time，表示当前 znode 被创建的时间
- **mZxid:** Modified Zxid，表示当前 znode 最后一次被修改时的事务 ID
- **mtime:** Modified Time，表示当前 znode 最后一次被修改时的时间
- **pZxid:** 表示当前 znode 的子节点列表最后一次被修改时的事务 ID。注意，只能是其子节点列表变更了才会引起 pZxid 的变更，子节点内容的修改不会影响 pZxid。
- **cversion:** Children Version，表示子节点的版本号。该版本号用于充当乐观锁。
- **dataVersion:** 表示当前 znode 数据的版本号。该版本号用于充当乐观锁。
- **aclVersion:** 表示当前 znode 的权限 ACL 的版本号。该版本号用于充当乐观锁。
- **ephemeralOwner:** 若当前 znode 是持久节点，则其值为 0；若为临时节点，则其值为创建该节点的会话的 SessionID。当会话消失后，会根据 SessionID 来查找与该会话相关的临时节点进行删除。
- **dataLength:** 当前 znode 中存放的数据的长度。
- **numChildren:** 当前 znode 所包含的子节点的个数。

4.1.2 会话

会话是 zk 中最重要的概念之一，客户端与服务端之间的任何交互操作都与会话相关。

ZooKeeper 客户端启动时，首先会与 zk 服务器建立一个 TCP 长连接。连接一旦建立，客户端会话的生命周期也就开始了。

(1) 会话状态

常见的会话状态有三种：

- **CONNECTING:** 连接中。Client 要创建一个连接，其首先会在本地创建一个 zk 对象，用于表示其所连接上的 Server。
- **CONNECTED:** 已连接。连接成功后，该连接的各种临时性数据会被初始化到 zk 对象中。
- **CLOSED:** 已关闭。连接关闭后，这个代表 Server 的 zk 对象会被删除。

(2) 会话连接超时管理—客户端维护

我们这里的会话连接超时管理指的是，客户端所发起的服务端连接时间记录，是从客户端当前会话第一次发起服务端连接的时间开始计时。

(3) 会话连接事件

客户端与服务端的长连接失效后，客户端将进行重连。在重连过程中客户端会产生三种会话连接事件：

- CONNECTION_LOSS: 连接丢失
- SESSION_MOVED: 会话转移
- SESSION_EXPIRED: 会话失效

(4) 会话空闲超时管理—服务端维护

服务器为每一个客户端的会话都记录着上一次交互后空闲的时长，及从上一次交互结束开始会话空闲超时的时间点。一旦空闲时长超时，服务端就会将该会话的 SessionId 从服务端清除。这也就是为什么客户端在空闲时需要定时向服务端发送心跳，就是为了维护这个会话长连接的。服务器是通过空闲超时管理来判断会话是否发生中断的。

服务端对于会话空闲超时管理，采用了一种特殊的方式——分桶策略。

A、分桶策略

分桶策略是指，将空闲超时时间相近的会话放到同一个桶中来进行管理，以减少管理的复杂度。在检查超时时，只需要检查桶中剩下的会话即可，因为没有超时的会话已经被移出了桶，而桶中存在的会话就是超时的会话。

zk 对于会话空闲的超时管理并非是精确的管理，即并非是一超时马上就执行相关的超时操作。

B、分桶依据

分桶的计算依据为：

$$\begin{aligned} \text{ExpirationTime} &= \text{CurrentTime} + \text{SessionTimeout} \\ \text{BucketTime} &= (\text{ExpirationTime}/\text{ExpirationInterval} + 1) * \text{ExpirationInterval} \end{aligned}$$

从以上公式可知，一个桶的大小为 ExpirationInterval 时间。只要 ExpirationTime 落入到同一个桶中，系统就会对其中的会话超时进行统一管理。

4.1.3 ACL

(1) ACL 简介

ACL 全称为 Access Control List（访问控制列表），是一种细粒度的权限管理策略，可以针对任意用户与组进行细粒度的权限控制。zk 利用 ACL 控制 znode 节点的访问权限，如节点

数据读写、节点创建、节点删除、读取子节点列表、设置节点权限等。

UGO, User、Group、Others，是一种粗粒度的权限管理策略。

(2) zk 的 ACL 维度

Unix/Linux 系统的 ACL 分为两个维度：组与权限，且目录的子目录或文件能够继承父目录的 ACL 的。而 Zookeeper 的 ACL 分为三个维度：授权策略 scheme、授权对象 id、用户权限 permission，子 znode 不会继承父 znode 的权限。

A、授权策略 scheme

授权策略用于确定权限验证过程中使用的检验策略（简单来说就是，通过什么来验证权限，或一个用户要访问某个节点，系统如何验证其身份），在 zk 中最常用的有四种策略。

- ip: 根据 IP 地址进行权限验证。
- digest: 根据用户名与密码进行验证。
- world: 对所有用户不做任何验证。
- super: 超级用户可以对任意节点进行任意操作。这种模式打开客户端的方式都与正常方式的不同。需要在打开客户端时添加一个系统属性。

B、授权对象 id

授权对象指的是权限赋予的用户。不同的授权策略具有不同类型的授权对象。下面是各个授权模式对应的授权对象 id。

- ip: 授权对象是 IP 地址。
- digest: 授权对象是“用户名 + 密码”。
- world: 其授权对象只有一个，即 anyone。
- Super: 与 digest 相同，极权对象为“用户名 + 密码”。

C、权限 Permission

权限指的是通过验证的用户可以对 znode 执行的操作。共有五种权限，不过 zk 支持自定义权限。

- c: Create, 允许授权对象在当前节点下创建子节点。
- d: Delete, 允许授权对象删除当前节点。
- r: Read, 允许授权对象读取当前节点的数据内容，及子节点列表。
- w: Write, 允许授权对象修改当前节点的数据内容，及子节点列表。
- a: Acl, 允许极权对象对当前节点进行 ACL 相关的设置。

4.1.4 Watcher 机制

zk 通过 Watcher 机制实现了发布/订阅模式。

(1) watcher 工作原理



(2) watcher 事件

对于同一个事件类型，在不同的通知状态中代表的含义是不同的。

客户端所处状态	事件类型（常量值）	触发条件	说明
SyncConnected	None (-1)	客户端与服务器成功建立会话	此时客户端与服务器处于连接状态
	NodeCreated (1)	Watcher 监听的对应数据节点被创建	
	NodeDeleted (2)	Watcher 监听的对应数据节点被删除	
	NodeDataChanged (3)	Watcher 监听的对应数据节点的数据内容发生变化	
	NodeChildrenChanged (4)	Watcher 监听的节点的子节点列表发生变化	
Disconnected (0)	None (-1)	客户端与 zk 断开连接	此时客户端与服务器处于连接断开状态
Expired (-112)	None (-1)	会话失效	此时客户端会话失效，通常会收到SessionExpiredException 异常
AuthFailed	None (-1)	使用错误的 scheme 进行权限检查	通常会收到AuthFailedException

(3) watcher 特性

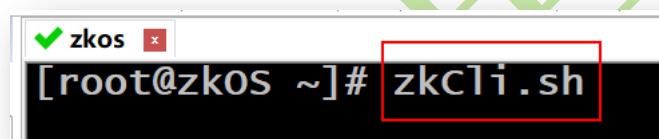
zk 的 watcher 机制具有非常重要的两个特性：

- **一次性**: 一旦一个 watcher 被触发, zk 就会将其从客户端的 WatchManager 中删除, 服务端中也会删除该 watcher。zk 的 watcher 机制不适合监听变化非常频繁的场景。
- **轻量级**: 真正传递给 Server 的是一个简易版的 watcher。回调逻辑存放在客户端, 没有在服务端。

4.2 客户端命令

4.2.1 启动客户端

(1) 连接本机 zk 服务器



```
[root@zkos ~]# zkcli.sh
```

(2) 连接其它 zk 服务器



```
[root@zkos ~]# zkcli.sh -server 192.168.59.117:2181
```

4.2.2 查看子节点-ls

查看根节点及/brokers 节点下所包含的所有子节点列表。

```
[zk: localhost:2181(CONNECTED) 0] ls /
[cluster, controller_epoch, brokers, zookeeper,
sumers, log_dir_event_notification, latest_produ
[zk: localhost:2181(CONNECTED) 1]
[zk: localhost:2181(CONNECTED) 1] ls /brokers
[ids, topics, seqid]
[zk: localhost:2181(CONNECTED) 2]
```

4.2.3 创建节点-create

(1) 创建永久节点

创建一个名称为 china 的 znode，其值为 999。

```
) 4] create /china 999
) 5] ls /
brokers, zookeeper, china, ad
ation, latest_producer_id_b1
) 6]
```

(2) 创建顺序节点

在/china 节点下创建了顺序子节点 beijing、shanghai、guangzhou，它们的数据内容分别为 bj、sh、gz。

```
[zk: localhost:2181(CONNECTED) 4] create -s /china/beijing bj
Created /china/beijing0000000001
[zk: localhost:2181(CONNECTED) 5] create -s /china/shanghai sh
Created /china/shanghai0000000002
[zk: localhost:2181(CONNECTED) 6] create -s /china/guangzhou gz
Created /china/guangzhou0000000003
[zk: localhost:2181(CONNECTED) 7] ls /china
[bj@localhost:2181(CONNECTED) 7] [beijing0000000001, shanghai0000000002, guangzhou0000000003]
[zk: localhost:2181(CONNECTED) 8]
```

(3) 创建临时节点

临时节点与持久节点的区别，在后面 get 命令中可以看到。

```
[zk: localhost:2181(CONNECTED) 8] create -e /china/aaa A
Created /china/aaa
[zk: localhost:2181(CONNECTED) 9] create -e /china/bbb B
Created /china/bbb
[zk: localhost:2181(CONNECTED) 10] create -e /china/ccc C
Created /china/ccc
[zk: localhost:2181(CONNECTED) 11] ls /china
[bj@localhost:2181(CONNECTED) 11] [beijing0000000001, aaa, shanghai0000000002, ccc, bbb, guangzh
[zk: localhost:2181(CONNECTED) 12]
```

4.2.4 获取节点信息-get

(1) 获取持久节点数据

```
✓ zkos x
[zk: localhost:2181(CONNECTED) 13] get /china
999
cZxid = 0x14a
ctime = Wed Jan 30 20:42:16 CST 2019
mZxid = 0x14a
mtime = Wed Jan 30 20:42:16 CST 2019
pZxid = 0x15a
cversion = 8
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 6
[zk: localhost:2181(CONNECTED) 14]
```

(2) 获取顺序节点信息

```
✓ zkos x
[zk: localhost:2181(CONNECTED) 14] get /china/beijing0000000001
bj
cZxid = 0x155
ctime = Thu Jan 31 15:05:00 CST 2019
mZxid = 0x155
mtime = Thu Jan 31 15:05:00 CST 2019
pZxid = 0x155
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 2
numChildren = 0
[zk: localhost:2181(CONNECTED) 15]
```

(3) 获取临时节点信息

```
[zk: localhost:2181(CONNECTED) 15] get /china/aaa
A
cZxid = 0x158
ctime = Thu Jan 31 15:13:31 CST 2019
mZxid = 0x158
mtime = Thu Jan 31 15:13:31 CST 2019
pZxid = 0x158
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x100000075680001
dataLength = 1
numChildren = 0
[zk: localhost:2181(CONNECTED) 16] get /china/bbb
B
cZxid = 0x159
ctime = Thu Jan 31 15:13:43 CST 2019
mZxid = 0x159
mtime = Thu Jan 31 15:13:43 CST 2019
pZxid = 0x159
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x100000075680001
dataLength = 1
numChildren = 0
[zk: localhost:2181(CONNECTED) 17]
```

4.2.5 更新节点数据内容-set

更新前:

```
✓ zkos x
[zk: localhost:2181(CONNECTED) 1] get /china
999
cZxid = 0x14a
ctime = Wed Jan 30 20:42:16 CST 2019
mZxid = 0x14a
mtime = Wed Jan 30 20:42:16 CST 2019
pZxid = 0x15b
cversion = 11
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 3
[zk: localhost:2181(CONNECTED) 2]
```



更新:

```
✓ zkos x
[zk: localhost:2181(CONNECTED) 2] set /china 888
cZxid = 0x14a
ctime = Wed Jan 30 20:42:16 CST 2019
mZxid = 0x15d
mtime = Thu Jan 31 16:18:24 CST 2019
pZxid = 0x15b
cversion = 11
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 3
[zk: localhost:2181(CONNECTED) 3]
```

```
[zk: localhost:2181(CONNECTED) 3] get /china  
888  
cZxid = 0x14a  
ctime = wed Jan 30 20:42:16 CST 2019  
mZxid = 0x15d  
mtime = Thu Jan 31 16:18:24 CST 2019  
pZxid = 0x15b  
cversion = 11  
dataVersion = 1  
aclVersion = 0  
ephemeralOwner = 0x0  
dataLength = 3  
numChildren = 3  
[zk: localhost:2181(CONNECTED) 4]
```

4.2.6 删除节点-delete

```
[zk: localhost:2181(CONNECTED) 4] delete /china/guangzhou0000000003  
[zk: localhost:2181(CONNECTED) 5]  
[zk: localhost:2181(CONNECTED) 5] ls /china  
[beijing0000000001, shanghai0000000002]  
[zk: localhost:2181(CONNECTED) 6]
```

若要删除具有子节点的节点，会报错。

```
[zk: localhost:2181(CONNECTED) 6] delete /china  
Node not empty: /china  
[zk: localhost:2181(CONNECTED) 7]
```

4.2.7 ACL 操作

(1) 查看权限-getAcl

```
[zk: localhost:2181(CONNECTED) 3] getAcl /china
'world, 'anyone
: cdrwa
[zk: localhost:2181(CONNECTED) 4]
```

(2) 设置权限

下面的命令是，首先增加了一个认证用户 zs，密码为 123，然后为/china 节点指定只有 zs 用户才可访问该节点，而访问权限为所有权限。

```
[zk: localhost:2181(CONNECTED) 7] addauth digest zs:123
[zk: localhost:2181(CONNECTED) 8] setAcl /china auth:zs:123:cdrwa
cXid = 0x14a
ctime = wed Jan 30 20:42:16 CST 2019
mZid = 0x15d
mtime = Thu Jan 31 16:18:24 CST 2019
pXid = 0x15e
cversion = 12
dataVersion = 1
aclVersion = 1
ephemeraOwner = 0x0
dataLength = 3
numChildren = 2
[zk: localhost:2181(CONNECTED) 9]
```

4.3 ZKClient 客户端

4.3.1 简介

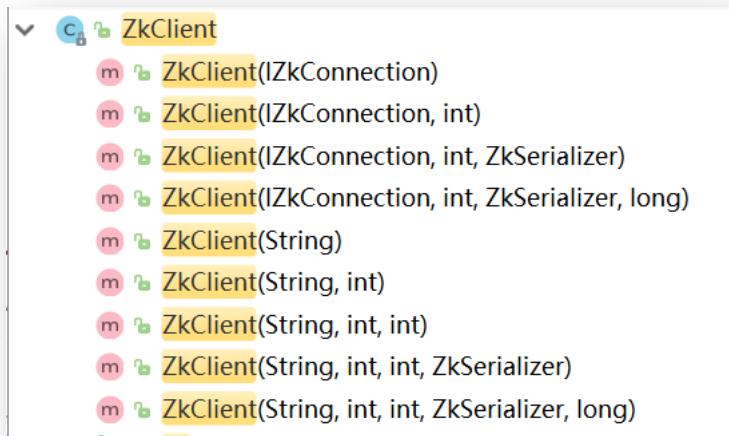
ZkClient 是一个开源客户端，在 Zookeeper 原生 API 接口的基础上进行了包装，更便于开发人员使用。内部实现了 Session 超时重连，Watcher 反复注册等功能。像 dubbo 等框架对其也进行了集成使用。

4.3.2 API 介绍

以下 API 方法均是 ZkClient 类中的方法。

(1) 创建会话

ZkClient 中提供了九个构造器用于创建会话。

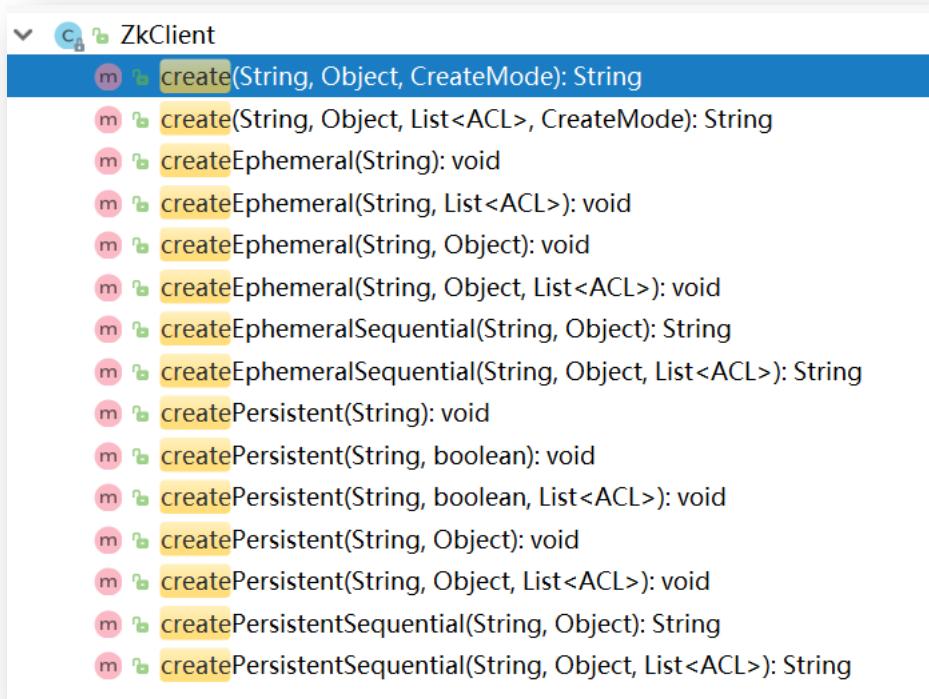


查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
zkServers	指定 zk 服务器列表，由英文状态逗号分隔的 host:port 字符串组成
connectionTimeout	设置连接创建超时时间，单位毫秒。在此时间内无法创建与 zk 的连接，则直接放弃连接，并抛出异常
sessionTimeout	设置会话超时时间，单位毫秒
zkSerializer	为会话指定序列化器。zk 节点内容仅支持字节数组 (byte[]) 类型，且 zk 不负责序列化。在创建 zkClient 时需要指定所要使用的序列化器，例如 Hessian 或 Kryo。默认使用 Java 自带的序列化方式进行对象的序列化。当为会话指定了序列化器后，客户端在进行读写操作时就会自动进行序列化与反序列化。
connection	IZkConnection 接口对象，是对 zk 原生 API 的最直接包装，是和 zk 最直接的交互层，包含了增删改查等一系列方法。该接口最常用的实现类是 zkClient 默认的实现类 ZkConnection，其可以完成绝大部分的业务需求。
operationRetryTimeout	设置重试超时时间，单位毫秒

(2) 创建节点

ZkClient 中提供了 15 个方法用于创建节点。

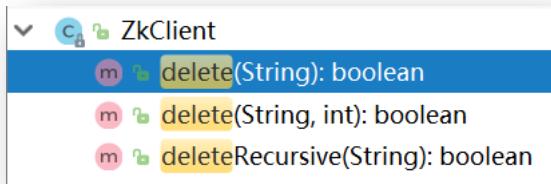


查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要创建的节点完整路径
data	节点的初始数据内容，可以传入 Object 类型及 null。zk 原生 API 中只允许向节点传入 byte[] 数据作为数据内容，但 zkClient 中具有自定义序列化器，所以可以传入各种类型对象。
mode	节点类型，CreateMode 枚举常量，常用的有四种类型。 <ul style="list-style-type: none">● PERSISTENT：持久型● PERSISTENT_SEQUENTIAL：持久顺序型● EPHEMERAL：临时型● EPHEMERAL_SEQUENTIAL：临时顺序型
acl	节点的 ACL 策略
callback	回调接口
context	执行回调时可以使用的上下文对象
createParents	是否级递归创建节点。zk 原生 API 中要创建的节点路径必须存在，即要创建子节点，父节点必须存在。但 zkClient 解决了这个问题，可以做递归节点创建。没有父节点，可以先自动创建了父节点，然后再在其下创建子节点。

(3) 删除节点

ZkClient 中提供了 3 个方法用于创建节点。

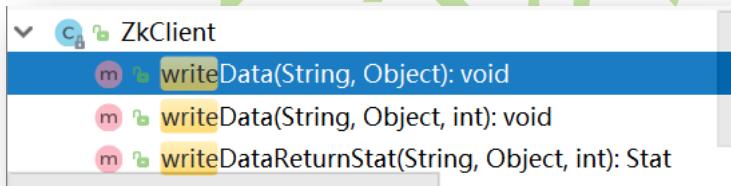


查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要删除的节点的完整路径
version	要删除的节点中包含的数据版本

(4) 更新数据

ZkClient 中提供了 3 个方法用于修改节点数据内容。

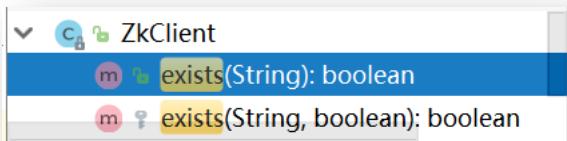


查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要更新的节点的完整路径
data	要采用的新数据值
expectedVersion	数据更新后要采用的数据版本号

(5) 检测节点是否存在

ZkClient 中提供了 2 个方法用于判断指定节点的存在性，但 `public` 方法就一个：只有一个参数的 `exists()` 方法。

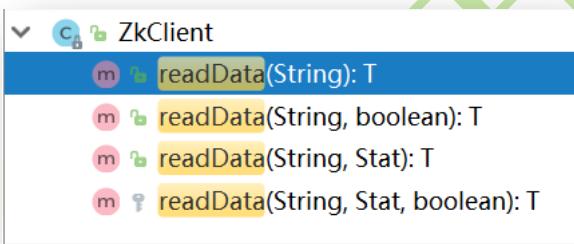


查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要判断存在性节点的完整路径
watch	要判断存在性节点及其子孙节点是否具有 watcher 监听

(6) 获取节点数据内容

ZkClient 中提供了 4 个方法用于获取节点数据内容，但 public 方法就三个。



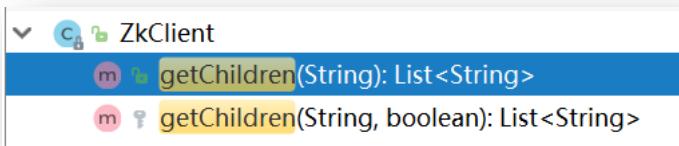
查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要读取数据内容的节点的完整路径
watch	指定节点及其子孙节点是否具有 watcher 监听
returnNullIfPathNotExists	这是个 boolean 值。默认情况下若指定的节点不存在，则会抛出 KeeperException\$NoNodeException 异常。设置该值为 true，若指定节点不存在，则直接返回 null 而不再抛出异常。
stat	指定当前节点的状态信息。不过，执行过后该 stat 值会被最新获取到的 stat 值给替换。

(7) 获取子节点列表

ZkClient 中提供了 2 个方法用于获取节点的子节点列表，但 public 方法就一个：只有一

一个参数的 getChildren()方法。



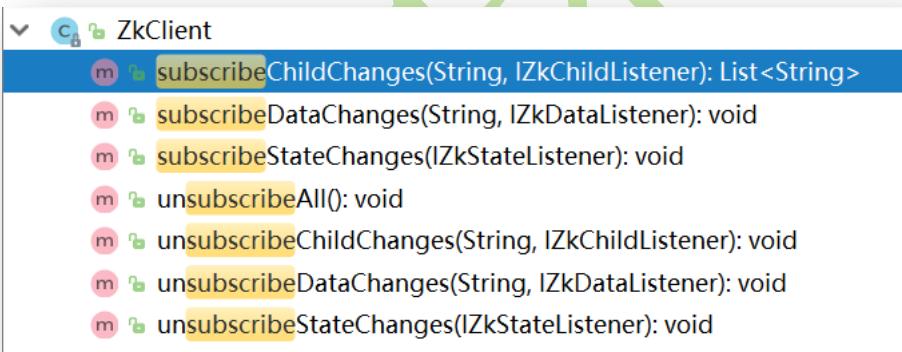
查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要获取子节点列表的节点的完整路径
watch	要获取子节点列表的节点及其子孙节点是否具有 watcher 监听

(8) watcher 注册

ZkClient 采用 Listener 来实现 Watcher 监听。客户端可以通过注册相关监听器来实现对 zk 服务端事件的订阅。

可以通过 subscribeXxx() 方法实现 watcher 注册，即相关事件订阅；通过 unsubscribeXxx() 方法取消相关事件的订阅。



查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
path	要操作节点的完整路径
IZkChildListener	子节点数量变化监听器
IZkDataListener	数据内容变化监听器
IZkStateListener	客户端与 zk 的会话连接状态变化监听器，可以监听新会话的创建、会话创建出错、连接状态改变。连接状态是系统定义好的枚举类型 Event. KeeperState 的常量

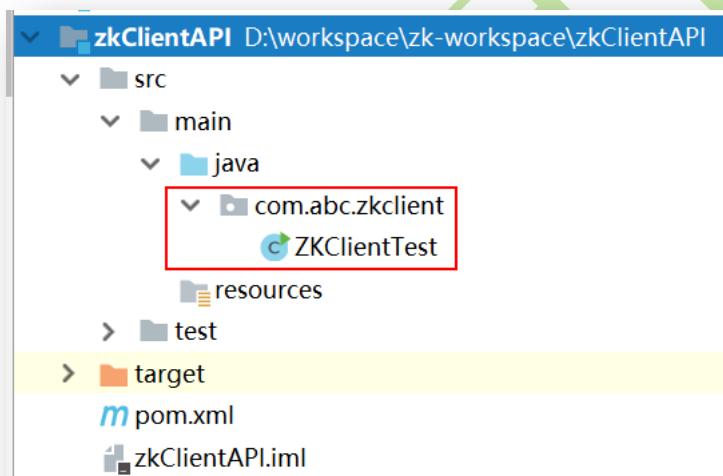
4.3.3 代码演示

(1) 创建工程

创建一个 Maven 的 Java 工程，并导入以下依赖。

```
<!--zkClient 依赖-->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

这里仅创建一个 ZkClient 的测试类即可。本例不适合使用 JUnit 测试。



(2) 代码

```
public class ZKClientTest {
    // 指定 zk 集群
    private static final String CLUSTER = "zkOS:2181";
    // 指定节点名称
    private static final String PATH = "/mylog";

    public static void main(String[] args) {
        // ----- 创建会话 -----
```

```
// 创建 zkClient
ZkClient zkClient = new ZkClient(CLUSTER);
// 为 zkClient 指定序列化器
zkClient.setZkSerializer(new SerializableSerializer());

// ----- 创建节点 -----
// 指定创建持久节点
CreateMode mode = CreateMode.PERSISTENT;
// 指定节点数据内容
String data = "first log";
// 创建节点
String nodeName = zkClient.create(PATH, data, mode);
System.out.println("新创建的节点名称为: " + nodeName);

// ----- 获取数据内容 -----
Object readData = zkClient.readData(PATH);
System.out.println("节点的数据内容为: " + readData);

// ----- 注册 watcher -----
zkClient.subscribeDataChanges(PATH, new IZkDataListener() {
    @Override
    public void handleDataChange(String dataPath, Object data) throws Exception
    {
        System.out.print("节点" + dataPath);
        System.out.println("的数据已经更新为了" + data);
    }

    @Override
    public void handleDataDeleted(String dataPath) throws Exception {
        System.out.println(dataPath + "的数据内容被删除");
    }
});

// ----- 更新数据内容 -----
zkClient.writeData(PATH, "second log");
String updatedData = zkClient.readData(PATH);
System.out.println("更新过的数据内容为: " + updatedData);

// ----- 删除节点 -----
zkClient.delete(PATH);

// ----- 判断节点存在性 -----
boolean isExists = zkClient.exists(PATH);
System.out.println(PATH + "节点仍存在吗? " + isExists);
```

```
}
```

4.4 Curator 客户端

4.4.1 简介

Curator 是 Netflix 公司开源的一套 zk 客户端框架，与 ZkClient 一样，其也封装了 zk 原生 API。其目前已经成为 Apache 的顶级项目。同时，Curator 还提供了一套易用性、可读性更强的 Fluent 风格的客户端 API 框架。

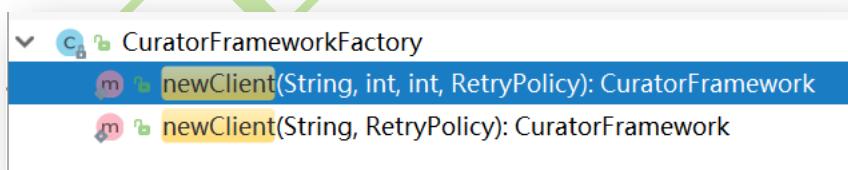
4.4.2 API 介绍

这里主要以 Fluent 风格客户端 API 为主进行介绍。

(1) 创建会话

A、普通 API 创建 newClient()

在 CuratorFrameworkFactory 类中提供了两个静态方法用于完成会话的创建。



查看这些方法的源码可以看到具体的参数名称，这些参数的意义为：

参数名	意义
connectString	指定 zk 服务器列表，由英文状态逗号分开的 host:port 字符串组成
sessionTimeoutMs	设置会话超时时间，单位毫秒，默认 60 秒
connectionTimeoutMs	设置连接超时时间，单位毫秒，默认 15 秒
retryPolicy	重试策略，内置有四种策略，分别由以下四个类的实例指定： ExponentialBackoffRetry、RetryNTimes、RetryOneTime、 RetryUntilElapsed

B、Fluent 风格创建

```
public class FluentTest {  
    public static void main(String[] args) throws Exception {  
        // 创建重试策略对象: 第1秒重试1次, 最多重试3次  
        ExponentialBackoffRetry retryPolicy = new ExponentialBackoffRetry(1000, 3);  
        // 创建客户端  
        CuratorFramework client = CuratorFrameworkFactory  
            .builder()  
            .connectString("zkOS:2181")  
            .sessionTimeoutMs(5000)  
            .connectionTimeoutMs(3000)  
            .retryPolicy(retryPolicy)  
            .namespace("logs")  
            .build();  
  
        // 开启客户端  
        client.start();  
    }  
}
```

(2) 创建节点 `create()`

下面以满足各种需求的举例方式分别讲解节点创建的方法。

说明：下面所使用的 `client` 为前面所创建的 `Curator` 客户端实例。

- 创建一个节点，初始内容为空
 - ◆ 语句：`client.create().forPath(path);`
 - ◆ 说明：默认创建的是持久节点，数据内容为空。
- 创建一个节点，附带初始内容
 - ◆ 语句：`client.create().forPath(path, "mydata".getBytes());`
 - ◆ 说明：Curator 在指定数据内容时，只能使用 `byte[]` 作为方法参数。
- 创建一个临时节点，初始内容为空
 - ◆ 语句：`client.create().withMode(CreateMode.EPHEMERAL).forPath(path);`
 - ◆ 说明：`CreateMode` 为枚举类型。
- 创建一个临时节点，并自动递归创建父节点
 - ◆ 语句：`client.create().creatingParentsIfNeeded().withMode(CreateMode.EPHEMERAL).forPath(path);`
 - ◆ 说明：若指定的节点多级父节点均不存在，则会自动创建。

(3) 删除节点 `delete()`

- 删除一个节点

- ◆ 语句: `client.delete().forPath(path);`
- ◆ 说明: 只能将叶子节点删除, 其父节点不会被删除。
- 删一个节点, 并递归删除其所有子节点
 - ◆ 语句: `client.delete().deletingChildrenIfNeeded().forPath(path);`
 - ◆ 说明: 该方法在使用时需谨慎。

(4) 更新数据 `setData()`

- 设置一个节点的数据内容
 - ◆ 语句: `client.setData().forPath(path, newData);`
 - ◆ 说明: 该方法具有返回值, 返回值为 Stat 状态对象。

(5) 检测节点是否存在 `checkExists()`

- 设置一个节点的数据内容
 - ◆ 语句: `Stat stat = client.checkExists().forPath(path);`
 - ◆ 说明: 该方法具有返回值, 返回值为 Stat 状态对象。若 stat 为 null, 说明该节点不存在, 否则说明节点是存在的。

(6) 获取节点数据内容 `getData()`

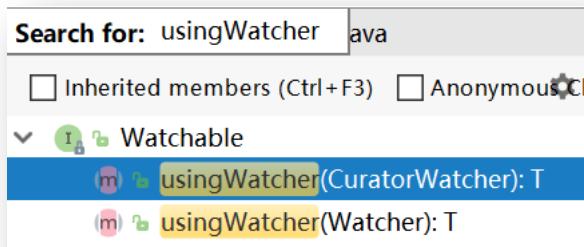
- 读取一个节点的数据内容
 - ◆ 语句: `byte[] data = client.getData().forPath(path);`
 - ◆ 说明: 其返回值为 byte[] 数组。

(7) 获取子节点列表 `getChildren()`

- 读取一个节点的所有子节点列表
 - ◆ 语句: `List<String> childrenNames = client.getChildren().forPath(path);`
 - ◆ 说明: 其返回值为 byte[] 数组。

(8) watcher 注册 `usingWatcher()`

curator 中绑定 watcher 的操作有三个: `checkExists()`、`getData()`、`getChildren()`。这三个方法的共性是, 它们都是用于获取的。这三个操作用于 watcher 注册的方法是相同的, 都是 `usingWatcher()` 方法。



这两个方法中的参数 CuratorWatcher 与 Watcher 都为接口。这两个接口中均包含一个 process() 方法，它们的区别是，CuratorWatcher 中的 process() 方法能够抛出异常，这样的话，该异常就可以被记录到日志中。

- 监听节点的存在性变化

```
Stat stat = client.checkExists().usingWatcher((CuratorWatcher) event -> {
    System.out.println("节点存在性发生变化");
}).forPath(path);
```

- 监听节点的内容变化

```
byte[] data = client.getData().usingWatcher((CuratorWatcher) event -> {
    System.out.println("节点数据内容发生变化");
}).forPath(path);
```

- 监听节点子节点列表变化

```
List<String> sons = client.getChildren().usingWatcher((CuratorWatcher) event -> {
    System.out.println("节点的子节点列表发生变化");
}).forPath(path);
```

4.4.3 代码演示

(1) 创建工程

创建一个 Maven 的 Java 工程，并导入以下依赖。

```
<!--curator 依赖-->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>2.12.0</version>
</dependency>
```

(2) 代码

```
ublic class FluentTest {
    public static void main(String[] args) throws Exception {
        // ----- 创建会话 -----
        // 创建重试策略对象: 第1秒重试1次, 最多重试3次
        ExponentialBackoffRetry retryPolicy = new ExponentialBackoffRetry(1000, 3);
        // 创建客户端
        CuratorFramework client = CuratorFrameworkFactory
            .builder()
            .connectString("zkOS:2181")
            .sessionTimeoutMs(15000)
            .connectionTimeoutMs(13000)
            .retryPolicy(retryPolicy)
            .namespace("logs")
            .build();
        // 开启客户端
        client.start();

        // 指定要创建和操作的节点, 注意, 其是相对于/Logs 节点的
        String nodePath = "/host";

        // ----- 创建节点 -----
        String nodeName = client.create().forPath(nodePath, "myhost".getBytes());
        System.out.println("新创建的节点名称为: " + nodeName);

        // ----- 获取数据内容并注册watcher -----
        byte[] data = client.getData().usingWatcher((CuratorWatcher) event -> {
            System.out.println(event.getPath() + "数据内容发生变化");
        }).forPath(nodePath);
        System.out.println("节点的数据内容为: " + new String(data));

        // ----- 更新数据内容 -----
        client.setData().forPath(nodePath, "newhost".getBytes());
        // 获取更新过的数据内容
        byte[] newData = client.getData().forPath(nodePath);
        System.out.println("更新过的数据内容为: " + new String(newData));

        // ----- 删除节点 -----
        client.delete().forPath(nodePath);

        // ----- 判断节点存在性 -----
        Stat stat = client.checkExists().forPath(nodePath);
```

```
boolean isExists = true;  
if(stat == null) {  
    isExists = false;  
}  
System.out.println(nodePath + "节点仍存在吗? " + isExists);  
}  
}
```



第5章 Zookeeper 典型应用场景

为进一步加强对 zk 的认识，理解 zk 的作用，下面再详细介绍一下 zk 在生产环境中的典型应用场景。

5.1 配置维护

5.1.1 什么是配置维护

分布式系统中，很多服务都是部署在集群中的，即多台服务器中部署着完全相同的应用，起着完全相同的作用。当然，集群中的这些服务器的配置文件是完全相同的。

若集群中服务器的配置文件需要进行修改，那么我们就需要逐台修改这些服务器中的配置文件。如果我们集群服务器比较少，那么这些修改还不是太麻烦，但如果集群服务器特别多，比如某些大型互联网公司的 Hadoop 集群有数千台服务器，那么纯手工的更改这些配置文件几乎就是一件不可能完成的任务。即使使用大量人力进行修改可行，但过多的人员参与，出错的概率大大提升，对于集群所形成的危险是很大的。

5.1.2 实现原理



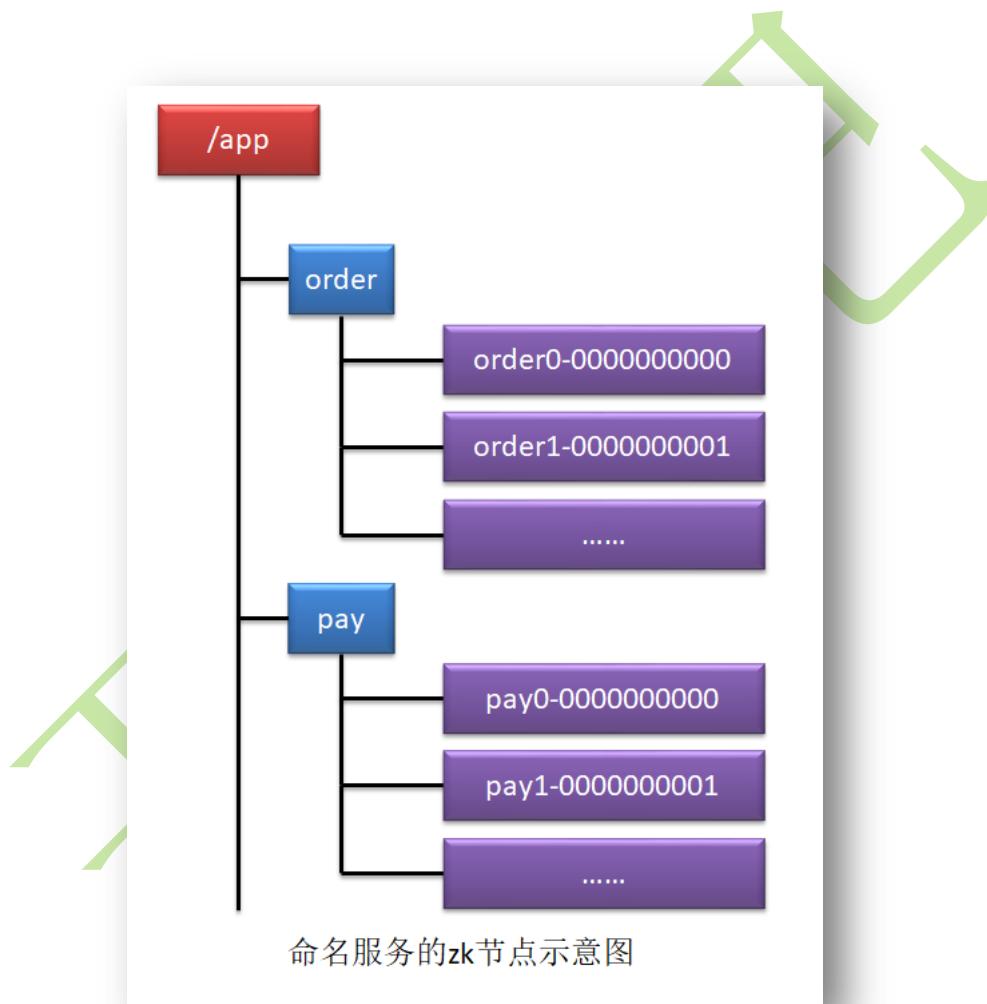
zk 可以通过“发布/订阅模型”实现对集群配置文件的管理与维护。“发布/订阅模型”分为推模式（Push）与拉模式（Pull）。zk 的“发布/订阅模型”采用的是推拉相结合的模式。

5.2 命名服务

5.2.1 什么是命名服务

命名服务是指可以为一定范围内的元素命名一个唯一标识，以与其它元素进行区分。在分布式系统中被命名的实体可以是集群中的主机、服务地址等。

5.2.2 实现原理



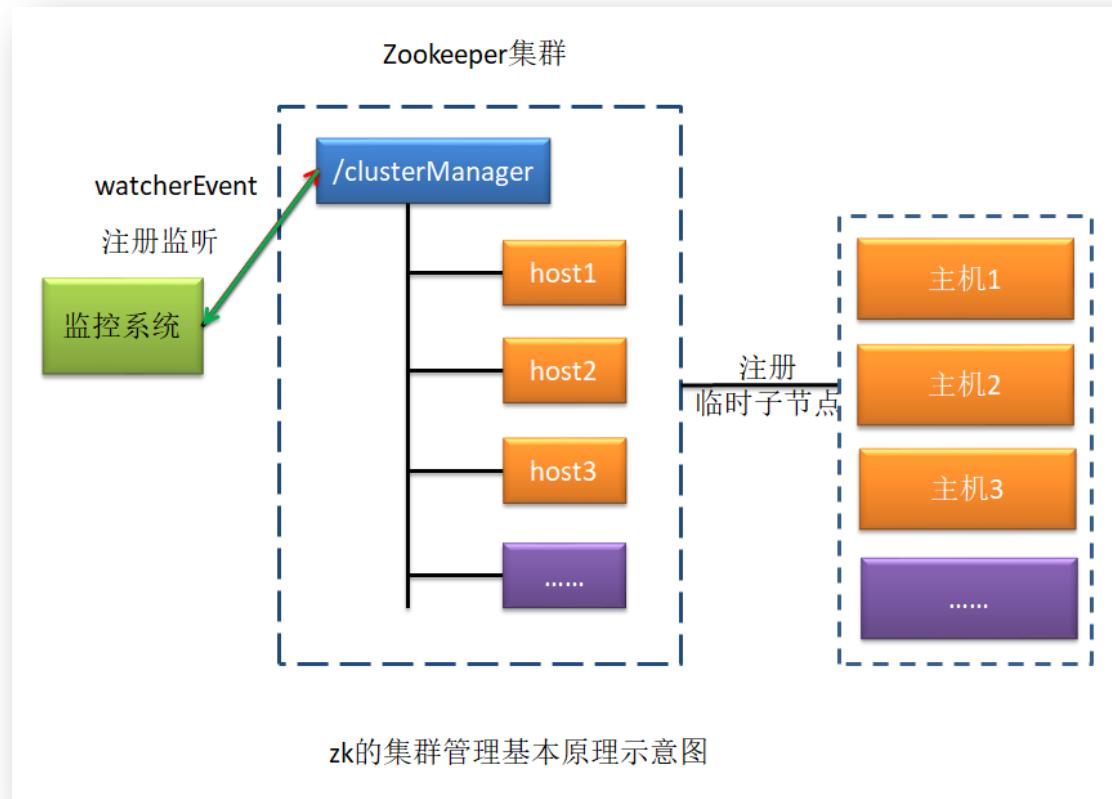
通过利用 zk 中节点路径不可重复的特点来实现命名服务的。当然，也可以配带上顺序节点的有序性来体现唯一标识的顺序性。

5.3 集群管理

对于集群，我们总是希望能够随时获取到当前集群中各个主机的运行时状态、当前集群中主机的存活状况等信息。通过 zk 可以实现对集群的随机监控。

5.3.1 基本原理

zk 进行集群管理的基本原理如下图所示。

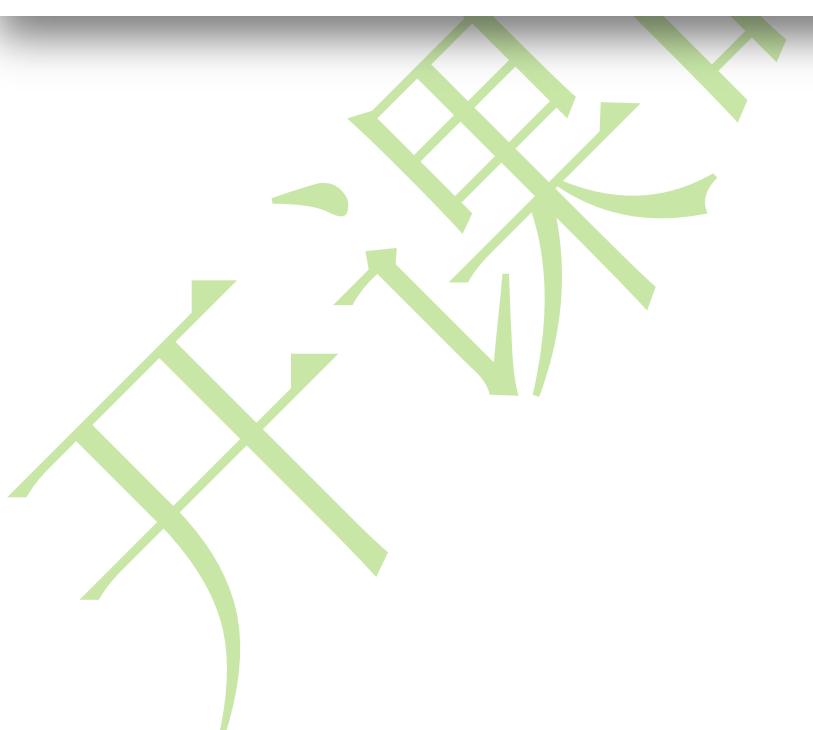
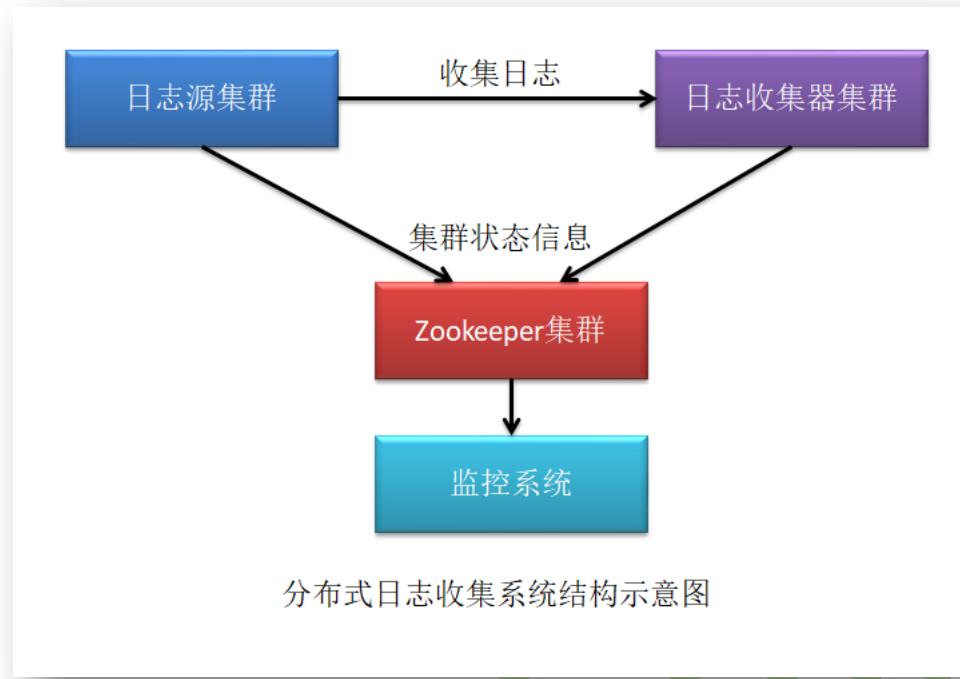


5.3.2 分布式日志收集系统

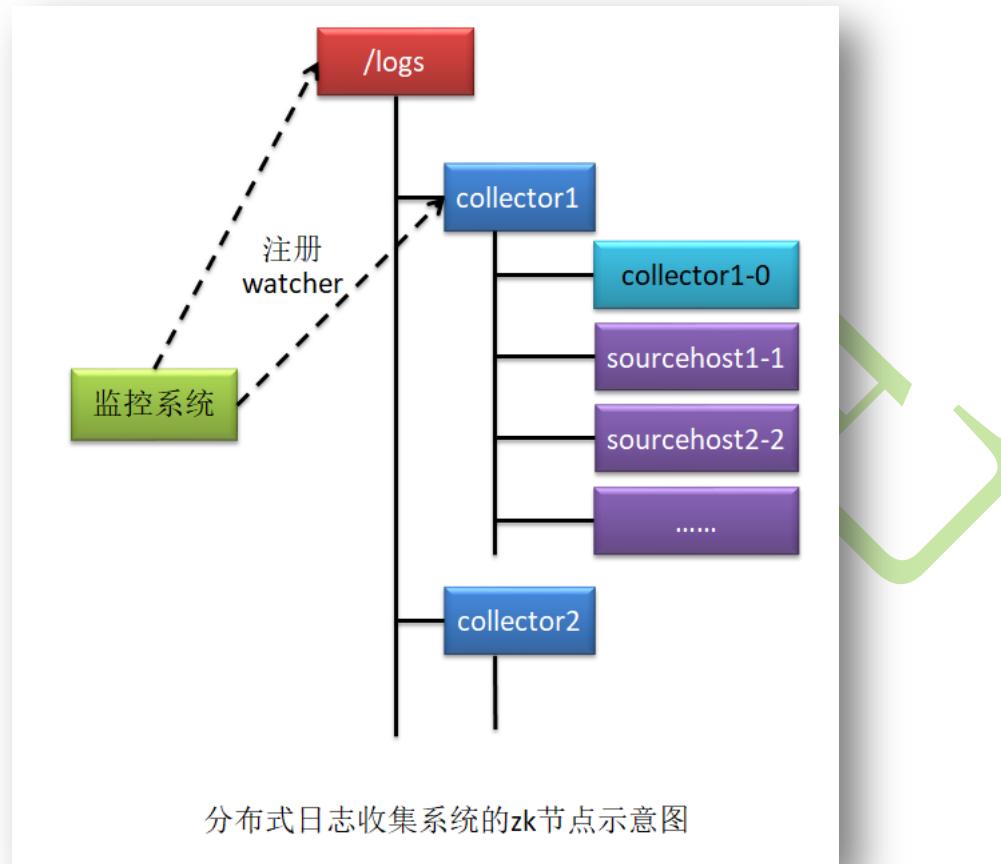
下面以分布式日志收集系统为例来分析 zk 对于集群的管理。

(1) 系统组成

首先要清楚，分布式日志收集系统由四部分组成：日志源集群、日志收集器集群，zk 集群，及监控系统。



(2) 系统工作原理



分布式日志收集系统的工作步骤有以下几步：

A、收集器的注册

在 zk 上创建各个收集器对应的节点。

B、任务分配

系统根据收集器的个数，将所有日志源集群主机分组，分别分配给各个收集器。

C、状态收集

这里的状态收集指的是两方面的收集：

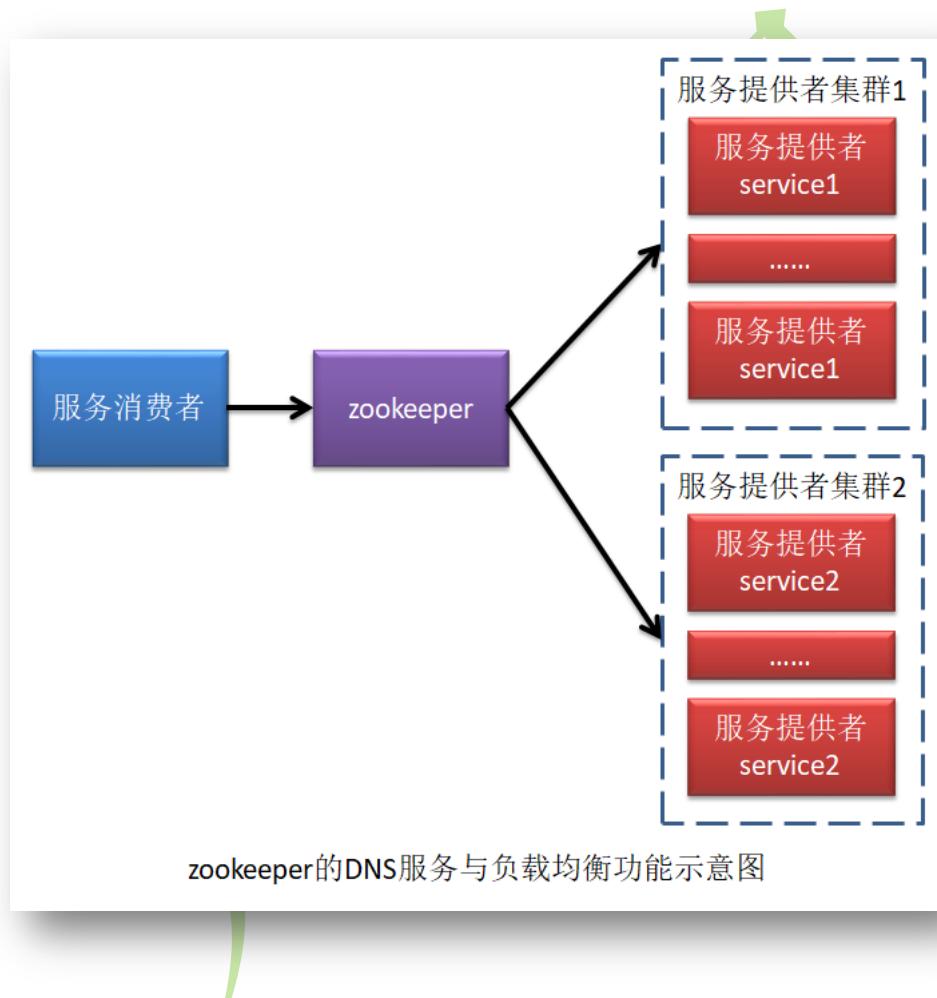
- 日志源主机状态，例如，日志源主机是否存活，其已经产生多少日志等
- 收集器的运行状态，例如，收集器本身已经收集了多少字节的日志、当前 CPU、内存的使用情况等

D、任务再分配 Rebalance

当出现收集器挂掉或扩容，就需要动态地进行日志收集任务再分配了，这个过程称为 Rebalance。只要发现某个收集器挂了，则系统进行任务再分配。

5.4 DNS 服务

zk 的 DNS 服务的功能主要是实现消费者与提供者的解耦合，防止提供者的单点问题，实现对提供者的负载均衡。



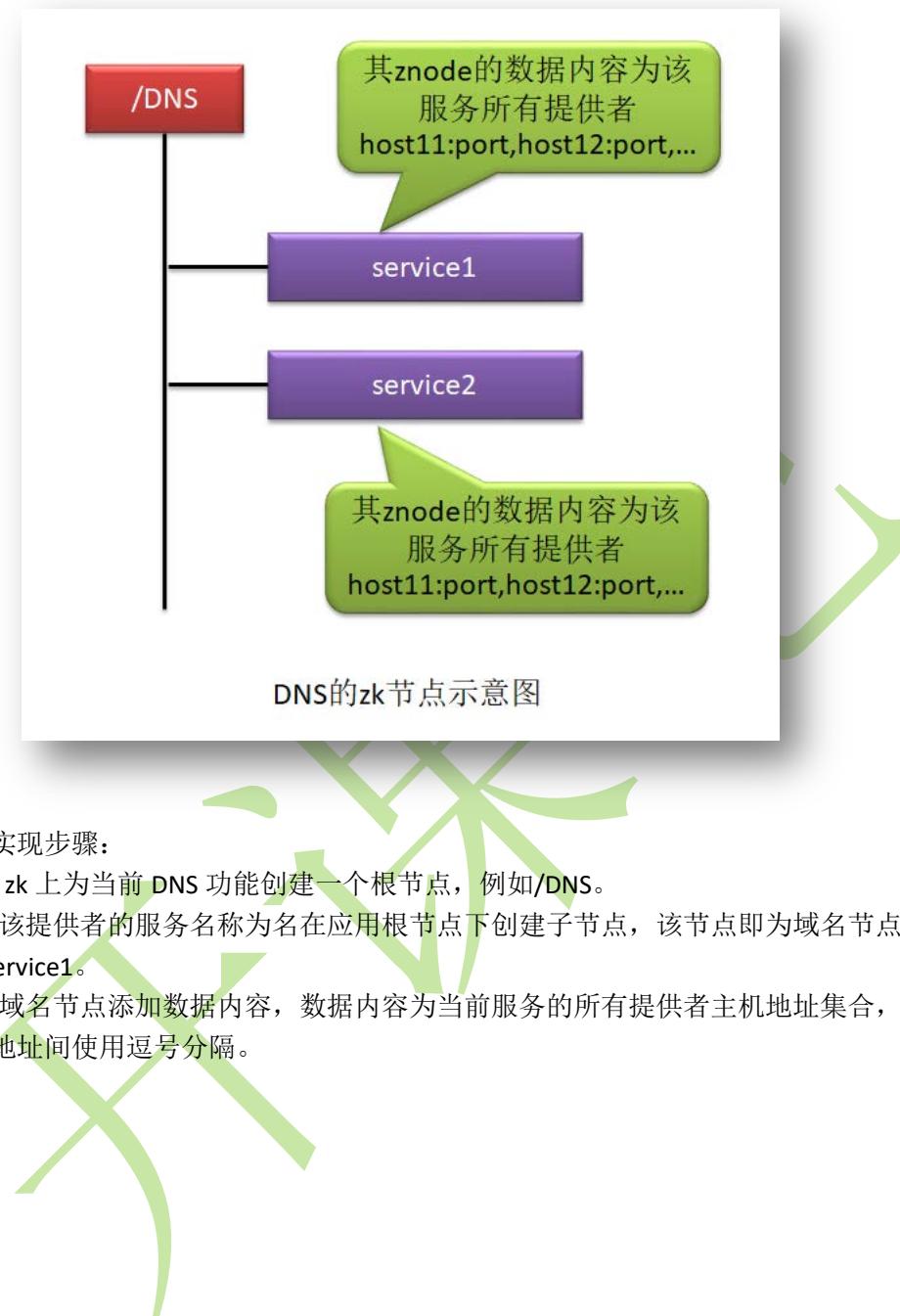
5.4.1 什么是 DNS

DNS, Domain Name System, 域名系统，即可以将一个名称与特定的主机 IP 加端口号进行绑定。zk 可以充当 DNS 的作用，完成域名到主机的映射。

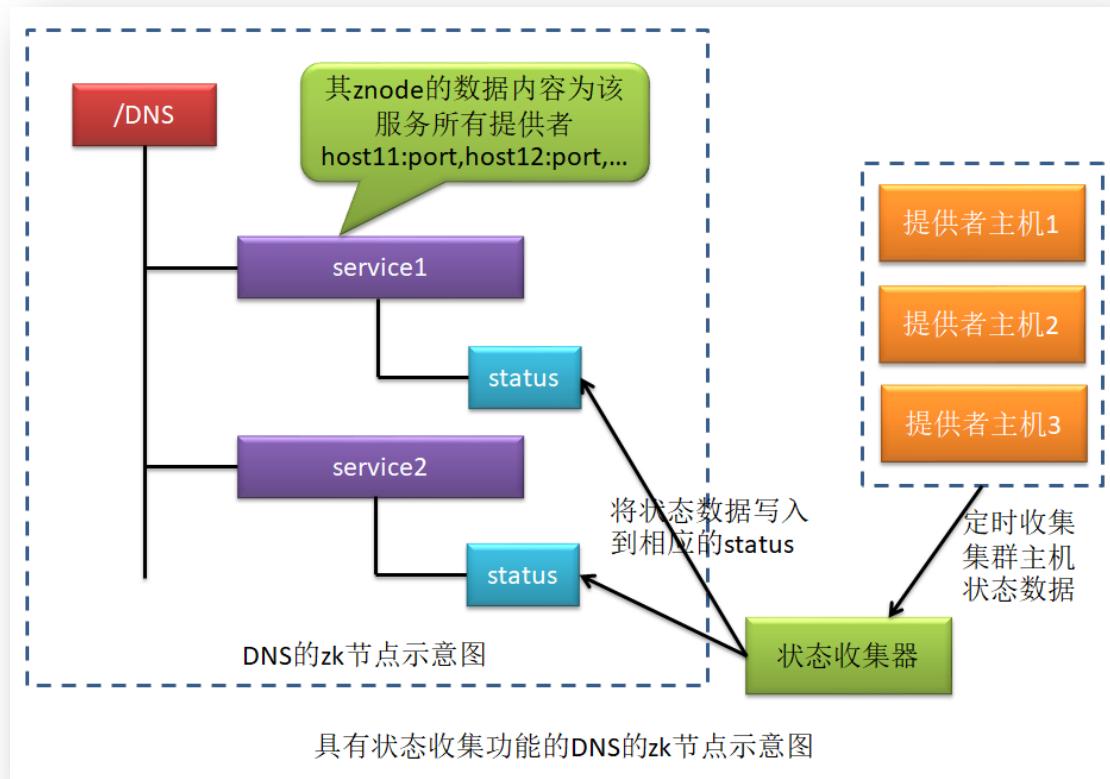
5.4.2 基本 DNS 实现原理

假设提供者应用程序 app1 与 app2 分别用于提供 service1 与 service2 两种服务，现要将

其注册到 zk 中，具体的实现步骤如下图所示。



5.4.3 具有状态收集功能的 DNS 实现原理



以上模型存在一个问题，如何获取各个提供者主机的健康状态、运行状态呢？可以为每一个域名节点再添加一个状态子节点，而该状态子节点的数据内容则为开发人员定义好的状态数据。这些状态数据是如何获取到的呢？是通过状态收集器（开发人员自行开发的）定期写入到 zk 的该节点中的。

阿里的 Dubbo 就是使用 Zookeeper 作为域名服务器的。

5.5 Master 选举

5.5.1 什么是 Master 选举

集群是分布式系统中不可或缺的组成部分，是为了解决分布式系统中计算单元的单点问题，水平扩展计算单元的处理能力的一种解决方案。

一般情况下，会在群集中选举出一个 Master，用于协调集群中的其它 Slave 主机，对于 Slave 主机的状态具有决定权。

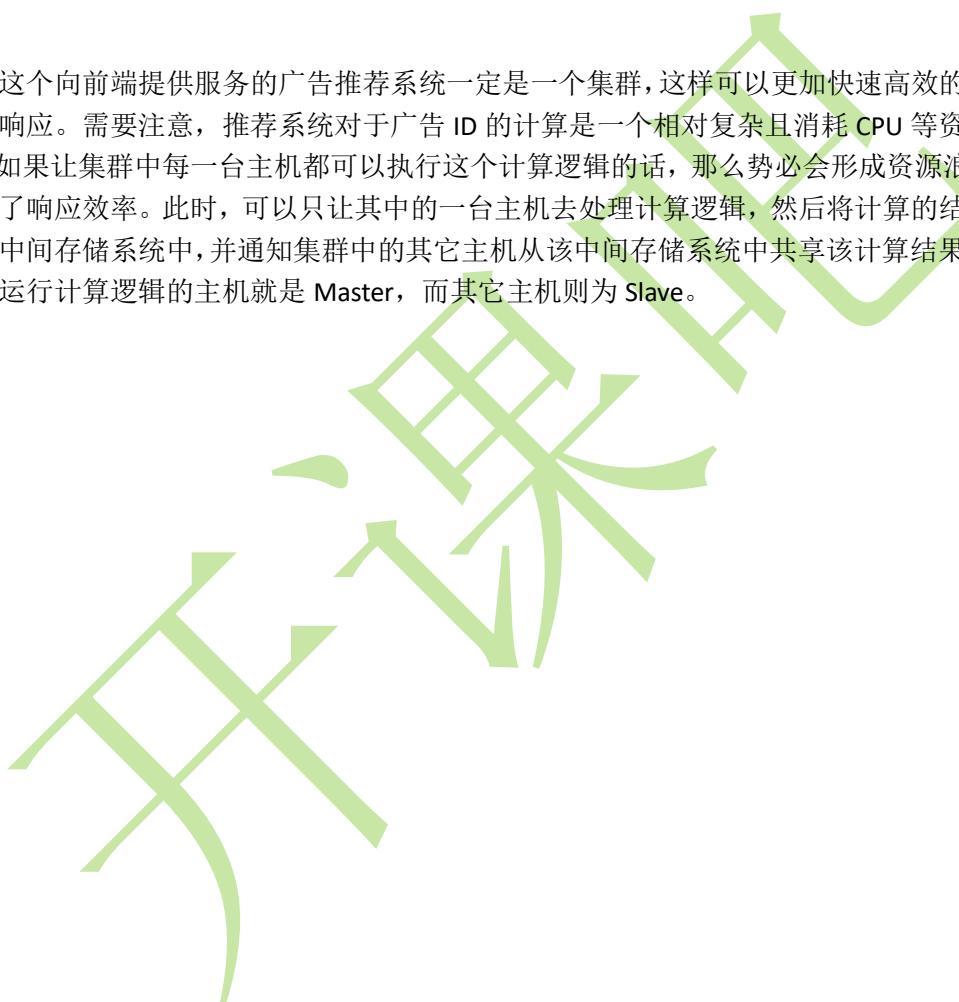
5.5.2 广告推荐系统

(1) 需求

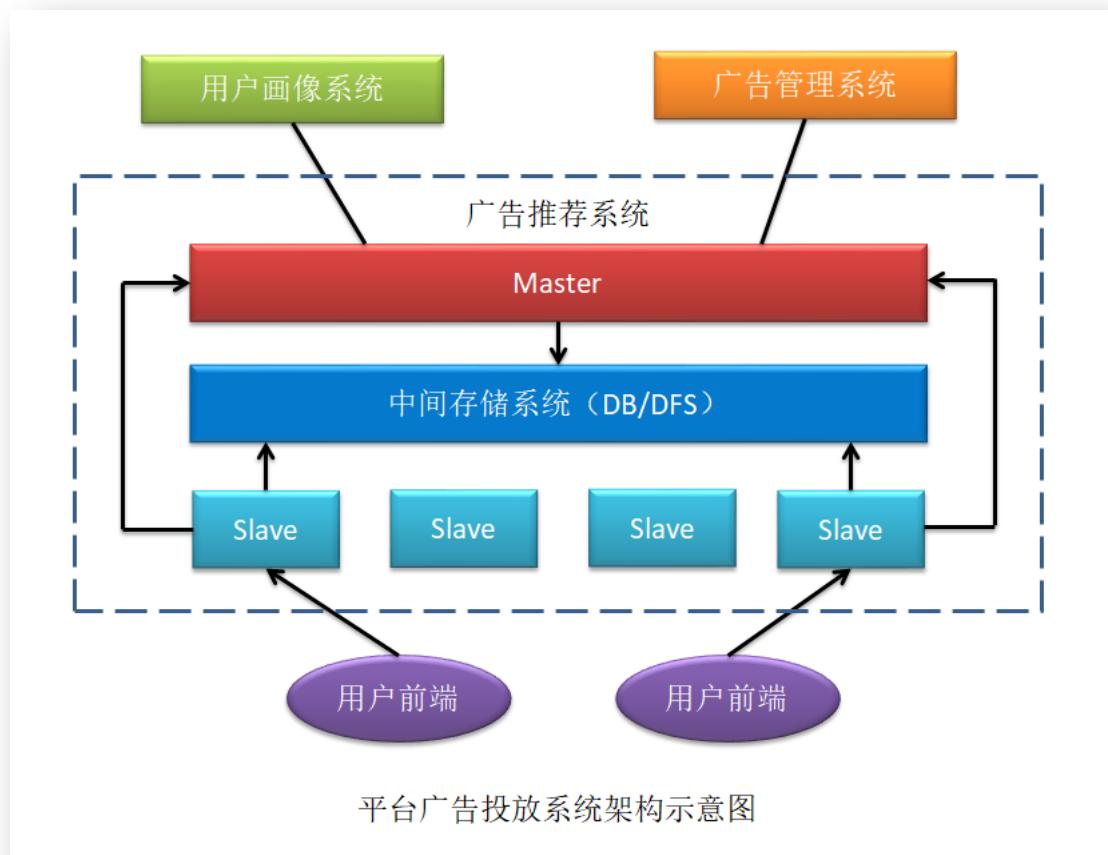
系统会根据用户画像，将用户归结为不同的种类。系统会为不同种类的用户推荐不同的广告。每个用户前端需要从广告推荐系统中获取到不同的广告 ID。

(2) 分析

这个向前端提供服务的广告推荐系统一定是一个集群，这样可以更加快速高效的为前端进行响应。需要注意，推荐系统对于广告 ID 的计算是一个相对复杂且消耗 CPU 等资源的过程。如果让集群中每一台主机都可以执行这个计算逻辑的话，那么势必会形成资源浪费，且降低了响应效率。此时，可以只让其中的一台主机去处理计算逻辑，然后将计算的结果写入到某中间存储系统中，并通知集群中的其它主机从该中间存储系统中共享该计算结果。那么，这个运行计算逻辑的主机就是 Master，而其它主机则为 Slave。



(3) 架构

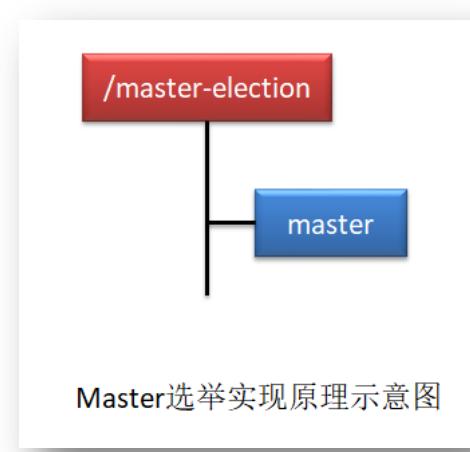


(4) Master 选举

使用 DBMS 的主键唯一特性可以实现 Master 的选举。让所有集群主机向数据库某表中插入主键相同的记录，由于 DBMS 具有主键冲突检查功能，所以其只能有一个主机插入成功，那么这个成功的主机即为 Master，其它为 Slave。

其存在的弊端是，仅使用 DBMS 的功能无法实现当 Master 崩机后对于 Slave 的通知，通知它们进行重新选举。

这个广告推荐系统集群中的 Master 是如何选举出来的呢？使用 zk 可以完成。使用 zk 中多个客户端对同一节点创建时，只有一个客户端可以成功的特性实现。



具体来说，由三步完成：

- Step1：多个客户端同时发起对同一临时节点/master-election/master 进行创建的请求，最终只能有一个客户端成功。这个成功的客户端主机就是 Master，其它客户端就是 Slave。
- Step2：让 Slave 都向这个临时节点的父节点/master-election 注册一个子节点列表的 watcher 监听。
- Step3：一旦该 Master 宕机，临时节点就会消失，zk 服务器就会向所有 Slave 发送子节点变更事件，Slave 在接收到事件后会调用相应的回调方法，该回调方法会重新向这个父节点创建相应的临时子节点。谁创建成功，谁就是新的 Master。

5.6 分布式同步

5.6.1 什么是分布式同步

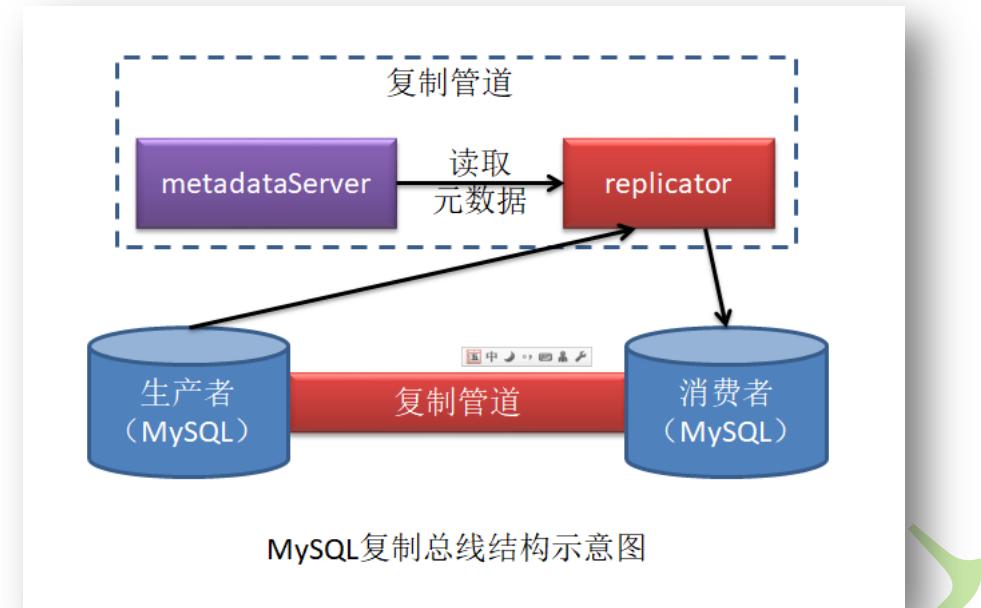
分布式同步，也称为分布式协调，是分布式系统中不可缺少的环节，是将不同的分布式组件有机结合起来的关键。对于一个在多台机器上运行的应用而言，通常需要一个协调者来控制整个系统的运行流程，例如执行的先后顺序，或执行与不执行等。

5.6.2 MySQL 数据复制总线

下面以“MySQL 数据复制总线”为例来分析 zk 的分布式同步服务。

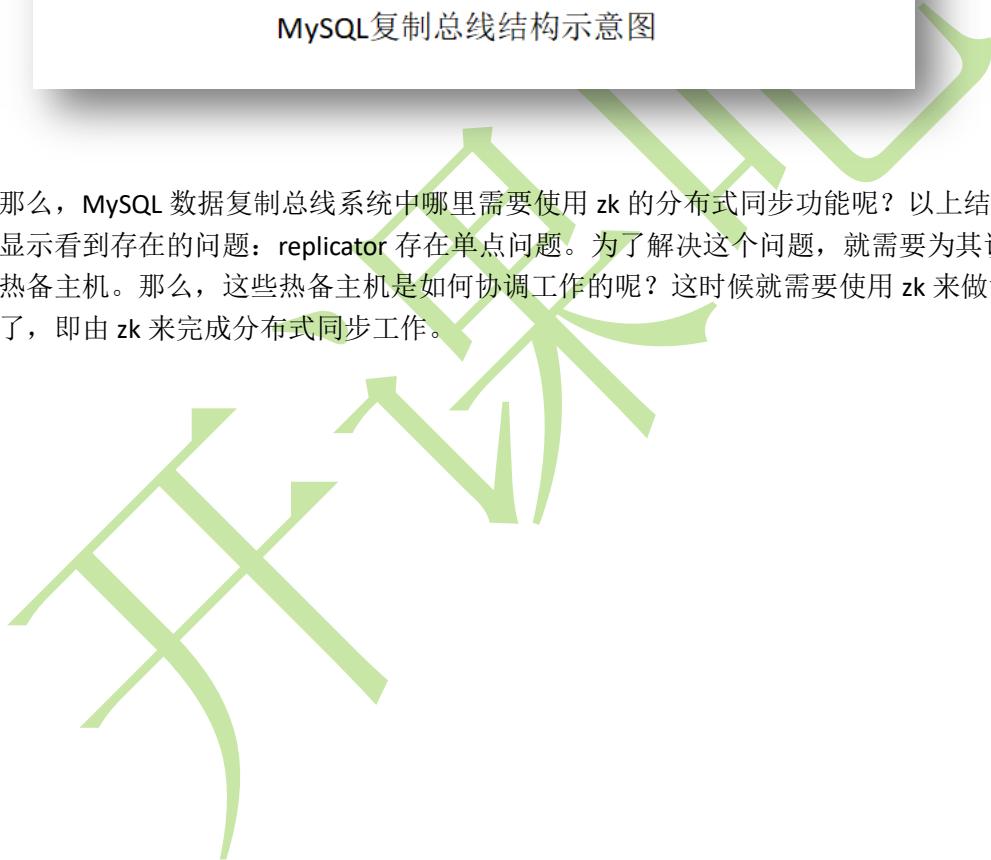
(1) 数据复制总线组成

MySQL 数据复制总线是一个实时数据复制框架，用于在不同的 MySQL 数据库实例间进行异步数据复制。其核心部分由三部分组成：生产者、复制管道、消费者。

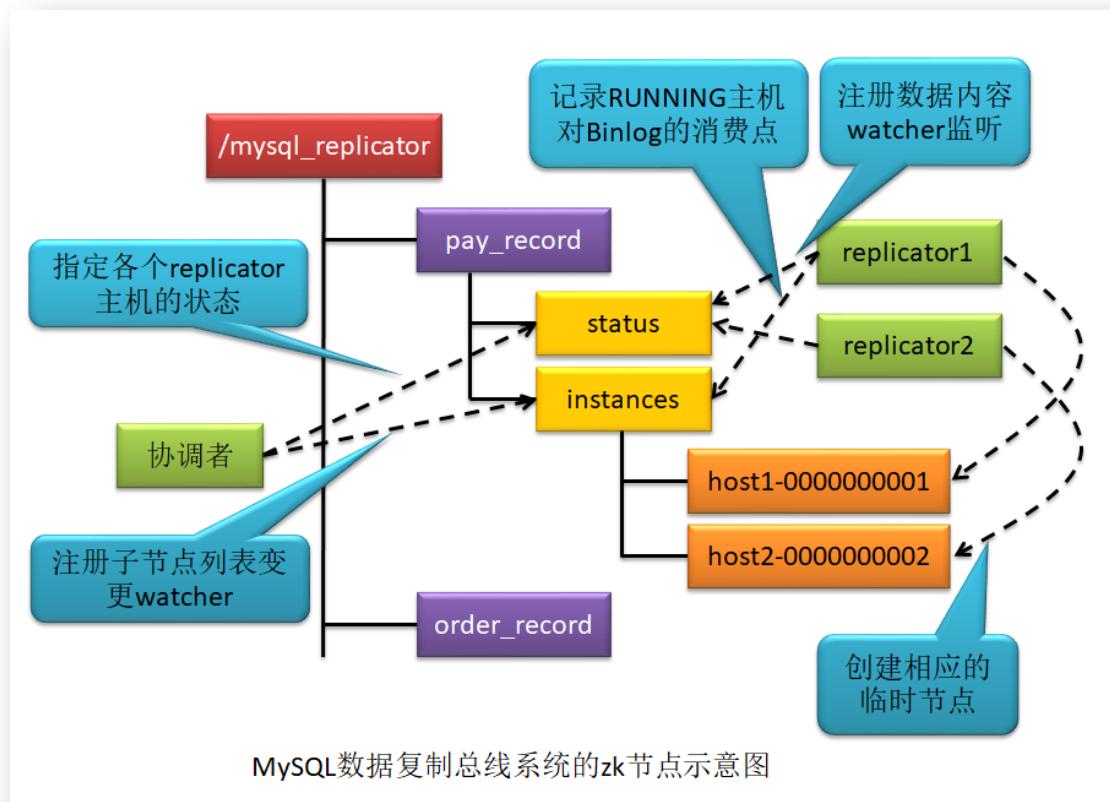


MySQL复制总线结构示意图

那么，MySQL 数据复制总线系统中哪里需要使用 zk 的分布式同步功能呢？以上结构中可以显示看到存在的问题：replicator 存在单点问题。为了解决这个问题，就需要为其设置多个热备主机。那么，这些热备主机是如何协调工作的呢？这时候就需要使用 zk 来做协调工作了，即由 zk 来完成分布式同步工作。



(2) 数据复制总线工作原理



MySQL 复制总线的工作步骤，总的来说分为三步：

A、复制任务注册

复制任务注册实际就是指不同的复制任务在 ZK 中创建不同的 znode，即将复制任务注册到 ZK 中。

B、replicator 热备

复制任务是由 replicator 主机完成的。为了防止 replicator 在复制过程中出现故障，replicator 采用热备容灾方案，即将同一个复制任务部署到多个不同的 replicator 主机上，但仅使一个处于 RUNNING 状态，而其它的主机则处于 STANDBY 状态。当 RUNNING 状态的主机出现故障，无法完成复制任务时，使某一个 STANDBY 状态主机转换为 RUNNING 状态，继续完成复制任务。

C、主备切换

当 RUNNING 态的主机出现宕机，则该主机对应的子节点马上就被删除了，然后在当前

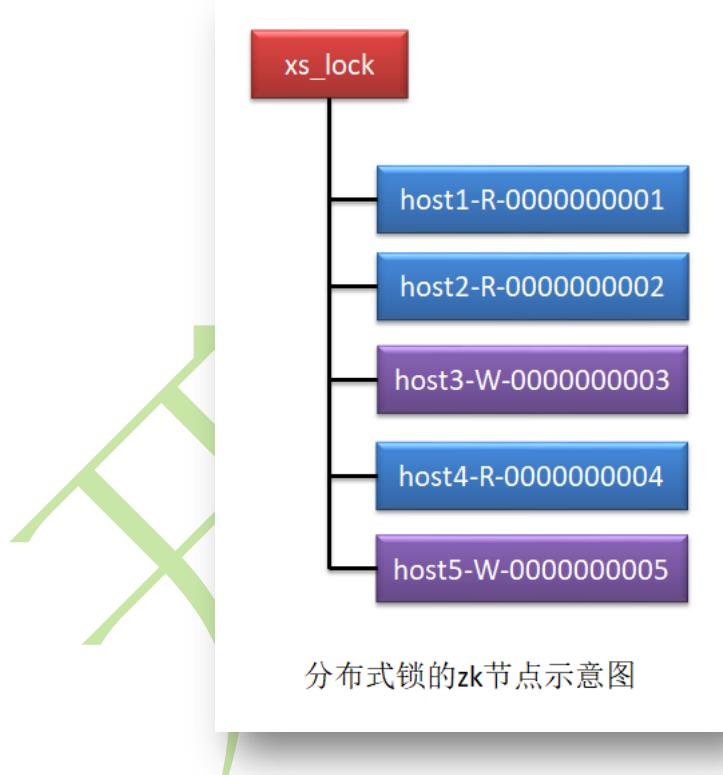
处于 STANDBY 状态中的 replicator 中找到序号最小的子节点，然后将其状态马上修改为 RUNNING，完成“主备切换”。

5.7 分布式锁

分布式锁是控制分布式系统同步访问共享资源的一种方式。Zookeeper 可以实现分布式锁功能。根据用户操作类型的不同，可以分为排他锁与共享锁。

5.7.1 分布式锁的实现

在 zk 上对于分布式锁的实现，使用的是类似于 “/xs_lock/[hostname]-请求类型-序号”的临时顺序节点。当客户端发出读写请求时会在 zk 中创建不同的节点。根据读写操作的不同及当前节点与之前节点的序号关系来执行不同的逻辑。



具体实现步骤：

- Step1：当一个客户端向某资源发出读/写请求时，若发现其为第一个请求，则首先会在 zk 中创建一个根节点。若节点已经存在，则无需创建。
- Step2：根节点已经存在了，客户端在根节点上注册子节点列表变更的 watcher 监听。
- Step3：watcher 注册完毕后，其会在根节点下创建一个读/写操作的临时顺序节点。
- Step4：节点创建完毕后，其就会马上触发客户端的 watcher 回调的执行。回调方法首先会将子节点列表读取，然后会查看序号比自己小的节点，并根据读写操作的不同，执行不同的逻辑。
- Step5：客户端读写操作完毕，其与 zk 的连接断开，则 zk 中该会话对应的节点消失。

5.7.2 分布式锁的改进

前面的实现方式存在“羊群效应”，为了解决其所带来的性能下降，可以对前述分布式锁的实现进行改进。

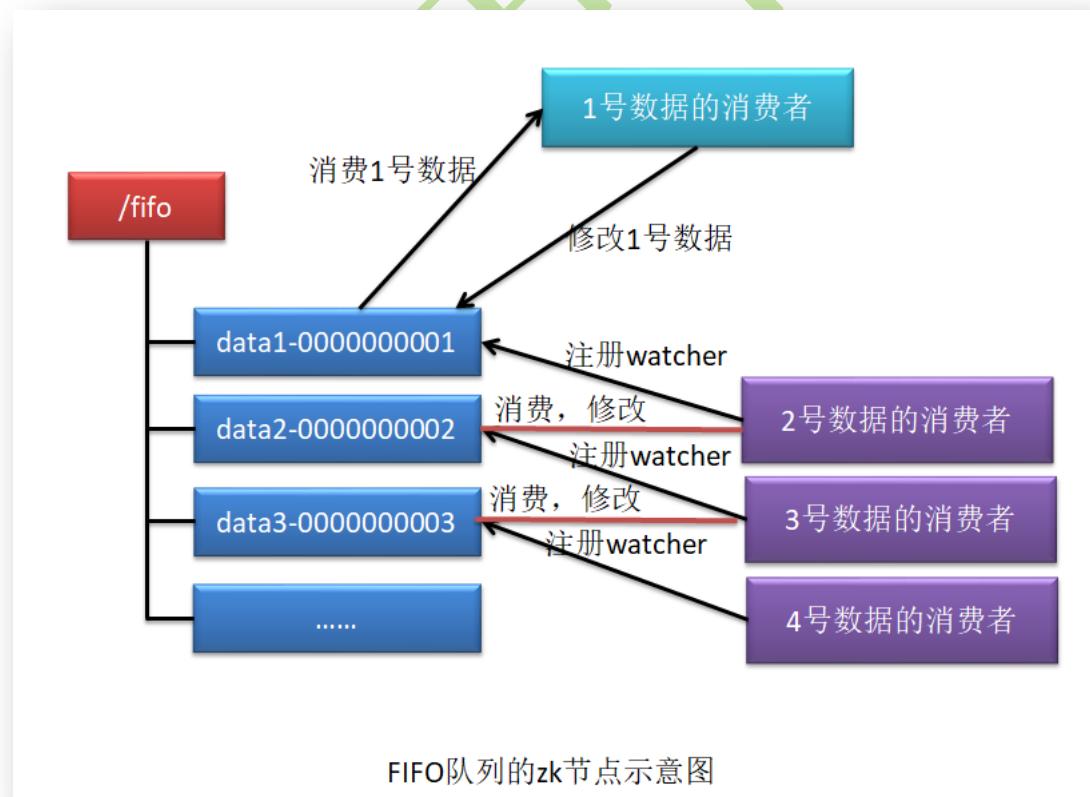
由于一个操作而引发了大量的低效或无用的操作的执行，这种情况称为羊群效应。

当客户端请求发出后，在 zk 中创建相应的临时顺序节点后马上获取当前的 /xs_lock 的所有子节点列表，但任何客户端都不向 /xs_lock 注册用于监听子节点列表变化的 watcher。而是改为根据请求类型的不同向“对其有影响的”子节点注册 watcher。

5.8 分布式队列

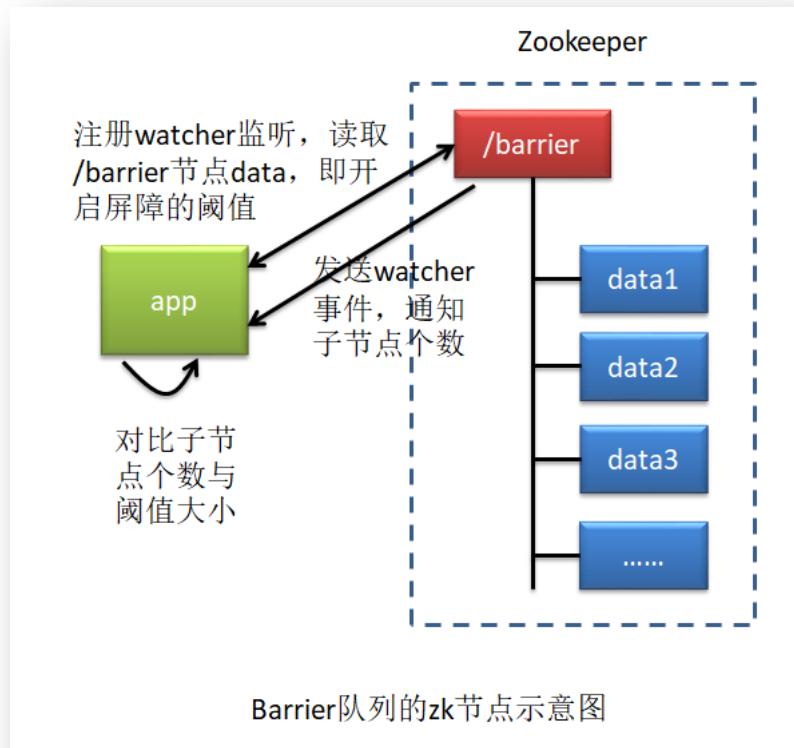
说到分布式队列，我们马上可以想到 RabbitMQ、Kafka 等分布式消息队列中间件产品。zk 也可以实现简单的消息队列。

5.8.1 FIFO 队列



zk 实现 FIFO 队列的思路是：利用顺序节点的有序性，为每个数据在 zk 中都创建一个相应的节点。然后为每个节点都注册 watcher 监听。一个节点被消费，则会引发消费者消费下一个节点，直到消费完毕。

5.8.2 分布式屏障 Barrier 队列



Barrier，屏障、障碍物。Barrier 队列是分布式系统中的一种同步协调器，规定了一个队列中的元素必须全部聚齐后才能继续执行后面的任务，否则一直等待。其常见于大规模分布式并行计算的应用场景中：最终的合并计算需要基于很多并行计算的子结果来进行。

zk 对于 Barrier 的实现原理是，在 zk 中创建一个/barrier 节点，其数据内容设置为屏障打开的阈值，即当其下的子节点数量达到该阈值后，app 才可进行最终的计算，否则一直等待。每一个并行运算完成，都会在/barrier 下创建一个子节点，直到所有并行运算完成。