

分布式协调服务Zookeeper

讲师 | 2025年9月

目录

CONTENTS

- 1 分布式系统概述
- 2 ZooKeeper简介
- 3 ZooKeeper原理
- 4 ZooKeeper应用场景

The background features a complex network diagram with numerous nodes (represented by small grey and black dots) connected by thin, light grey lines. These connections form a dense web that fills the entire slide, with some areas appearing more concentrated than others. The overall aesthetic is technical and modern.

1 chapter

分布式系统概述

- ✓ 什么是分布式系统
- ✓ CAP定理
- ✓ BASE理论

➤ 概念

- 将硬件或软件组件（服务）分布在不同的网络计算机上
- 通过消息传递进行通信和协调

➤ 特点

- 分布性
- 对等性
 - 平等：无主从之分
 - 独立：拥有自己的CPU和内存，独立处理数据
- 并发性
 - 外部：承载多个客户端的并发访问
 - 内部：作业（Job）被分解成多个任务（Task），并发运行在不同的节点上
- 故障独立性
 - 部分节点出现故障不影响整个系统的正常使用

➤ 典型问题

- 通信异常
- 网络分区（脑裂）
 - 系统分裂为两个甚至多个局部小集群（分区）
 - 各分区独立运行，同时提供服务，从而导致混乱
- 节点故障
 - 宕机或僵死
- 三态
 - 成功、失败和超时

➤ C (Consistency, 一致性)

- 含义：同一时刻，数据在不同节点的多个副本是否具有完全相同的值
- 类型
 - 强一致性：数据更新完成后，同一时刻，不同的读操作都能获得最新的值
 - 弱一致性：数据更新完成后，同一时刻，不同的读操作不一定都能获得最新的值，也无法保证多长时间之后可以获得最新的值

➤ A (Availability, 可用性)

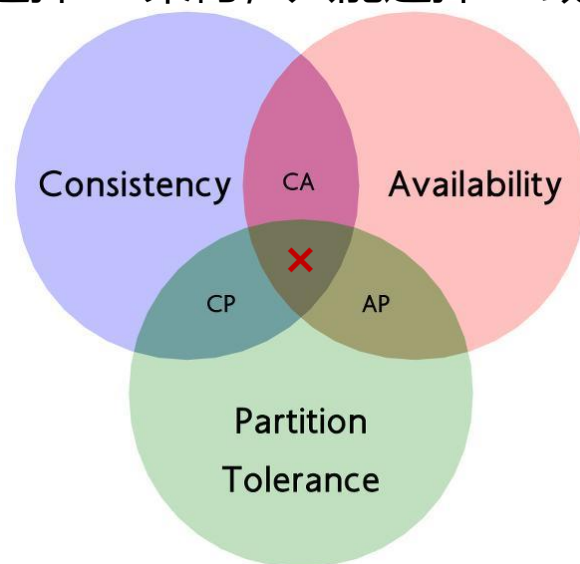
- 含义：对于每一次请求，系统是否都能在有限（指定）的时间内做出响应

➤ P (Partition Tolerance, 分区容错性)

- 含义：当发生网络分区时，系统仍能对外提供满足一致性C和可用性A的服务

➤ CAP定理

- 表述：分布式系统在同一时间片段内，不可能同时满足一致性C、可用性A和分区容错性P，最多只能满足其中的两项
- 理解
 - “满足”意味着100%，满足C → 满足强一致性，满足A → 满足绝对可用性
 - 对分布式系统而言，网络分区无法避免，满足P是前提条件，所以不可能选择CA架构，只能选择CP或AP架构
 - ✓ 例如：发生网络分区时，某个节点正在进行写操作
 - (1) 如果为了保证C，必须禁止其他节点的读写操作，那就与A冲突了
 - (2) 如果为了保证A，其他节点正常读写，那就与C冲突了
 - 选择CP或AP架构，关键在业务场景
 - ✓ 例如：对于必须确保强一致性的银行业务，只能选择CP



- BA (Basically Availability, 基本可用性)
 - 当系统发生故障时, 在确保核心功能和指标有效的提前下, 允许损失部分可用性, 包括响应时间上的损失、非核心功能上的损失等
- S (Soft State, 软状态)
 - 允许数据存在中间状态 (暂时未更新), 且该状态不会影响整体可用性
 - 允许不同节点上的数据副本的同步过程存在一定延时
- EC (Eventually Consistency, 最终一致性)
 - 分布在不同节点上的数据副本, 在经过一定时间的同步后, 最终达到一致状态
 - 例如: Zookeeper、HDFS QJM写事务的过半策略
 - 弱一致性的升级版

➤ BASE理论

- 表述：分布式系统在满足分区容错性P的同时，允许数据软状态S的存在，并实现基本可用性BA和最终一致性EC
- 理解
 - 在满足P的前提下，对CAP中的强一致性A和绝对可用性C进行适度妥协
 - ✓ $A \rightarrow BA$ 、 $C \rightarrow EC$
 - ✓ 通过容忍部分数据的暂时不一致（软状态），即牺牲数据的强一致性（确保最终一致性），以确保系统的核心功能和指标有效（基本可用）
 - CAP定理的延伸，CAP的 $C+P$ / $A+P \rightarrow$ BASE的 $EC+BA+P$
 - 对大规模互联网系统分布式实践的总结






2 chapter

ZooKeeper简介

- ✓ 什么是ZooKeeper
- ✓ 基本特性

- Google Chubby的开源实现，由Hadoop子项目发展而来
- 高可用、高容错和高性能的分布式协调系统
- 将复杂易错的分布式一致性服务封装起来，形成高效可靠的原语集，并提供简单易用的访问接口
- 为大型分布式系统提供关键、共性、高效和可靠的协调服务
 - 数据发布订阅
 - 分布式锁
 - 统一命名
 - 配置管理
 - 集群管理（Master选举、节点上下线动态监测）
- 大数据开源技术体系的基础组件，无可替代
 - 开源免费、简单易用、高可用、高容错、高性能
 - 被HDFS、YARN、Kafka和HBase等技术高度依赖

- 最终一致性
 - 保证数据在各节点的副本最终能够达到一致状态，这是ZK最重要的功能
- 有限实时性
 - 不能保证一定能读到最新数据，要想实时获取最新数据，应先调用sync()进行强制同步
- 原子性
 - 集群所有服务器或者都成功执行了某一事务，或者都没有，不存在第三种情况
- 顺序性
 - 从同一客户端发起的事务请求，最终会严格按照其发起顺序应用到ZK中
- 可靠性
 - 服务端一旦成功执行了事务，则该事务所引起的数据变更会被一直保留
- 单一视图
 - 无论客户端连接到哪一台服务器，所看到的数据模型都是一致的



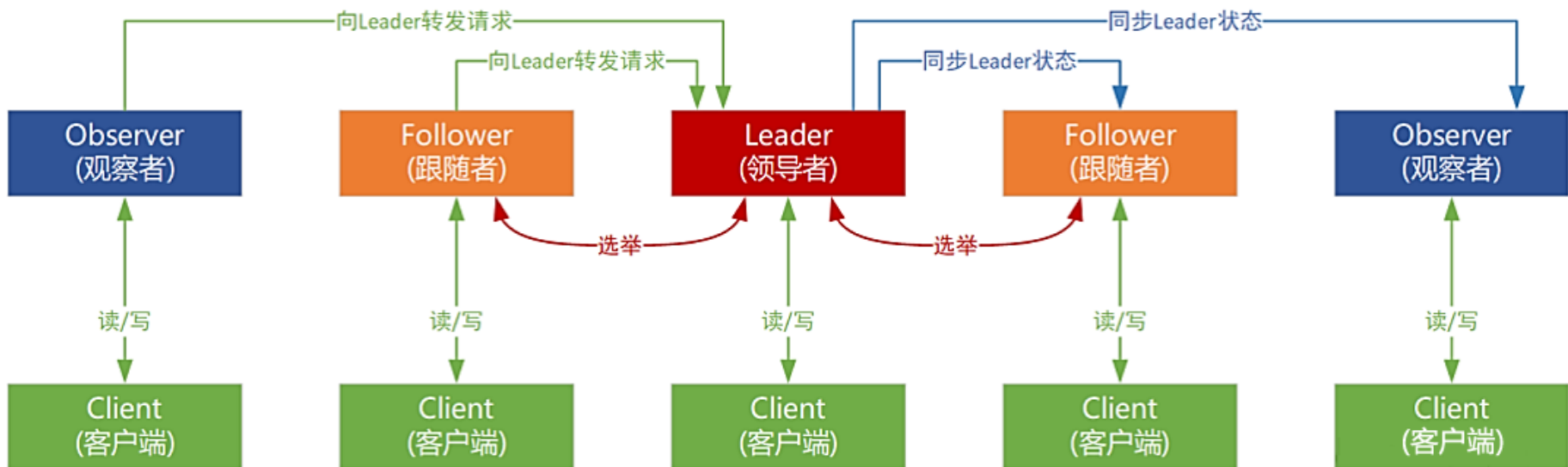
3 chapter

ZooKeeper原理

- ✓ 系统架构
- ✓ 数据模型
- ✓ 数据读写
- ✓ Leader选举
- ✓ Watcher机制

➤ 系统架构图 (Master/Slave)

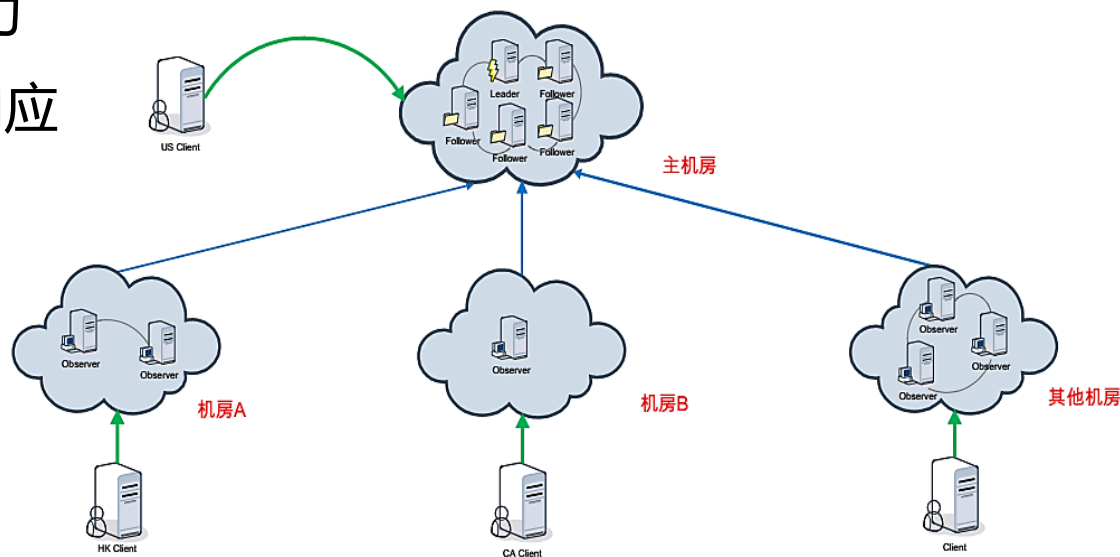
- 四种角色：Leader、Follower、Observer和Client
- Leader和Follower遵循“Quorum仲裁机制（过半策略）”，参与事务处理和Leader选举的投票
 - 若Leader+Follower的数量为 m ，则 $Quorum = m/2 + 1$ ，只有投票数 $\geq Quorum$ ，事务和选举才判定成功
- Leader+Follower的数量最好是奇数（ $2n+1$ ），在不影响事务处理的前提下，最多可容忍 n 台宕机
 - 例如：对于节点数为5和6的两个集群，①二者都最多容忍2台宕机，所以容灾能力相同；②前者的Quorum数为3，后者为4，前者更小，事务处理和选举的效率更高



角 色		职 责
Leader（领导者 / Master / 集群中唯一）		<ul style="list-style-type: none">接收Follower和Observer转发的写请求事务的发起和决议更新系统状态
Leaner（学习者 / Slave）	Follower（跟随者）	<ul style="list-style-type: none">处理客户端读请求，将写请求转发给Leader参与Leader发起的事务处理，负责本节点的数据变更和事务提交参与Leader选举同步Leader状态（保持心跳连接）
	Observer（观察者）	<ul style="list-style-type: none">处理客户端读请求，将写请求转发给Leader不参与Leader发起的事务处理不参与Leader选举同步Leader状态（保持心跳连接）
Client（客户端）		发起读写请求

➤ 为什么要引入Observer (ZK 3.3.0)

- 在不影响写性能的前提下，提升读操作的性能和吞吐量
 - Follower数量越多，写操作的性能越差，吞吐量越小
 - Observer不参与事务处理，数量再多也不影响集群的写性能
 - Observer不参与选主和事务处理，即使宕机也不影响集群的可用性
- 提升集群的可扩展性，以及大规模访问的承载能力
- 跨中心部署Observer，为本地读请求提供快速响应
 - Leader和Follower部署在一个中心，很大程度上降低了网络延时对写性能的影响
 - 由于Observer要转发写请求和同步Leader状态，所以跨中心部署并不能彻底消除网络延时
 - 例如：阿里开源的跨机房同步系统Otter

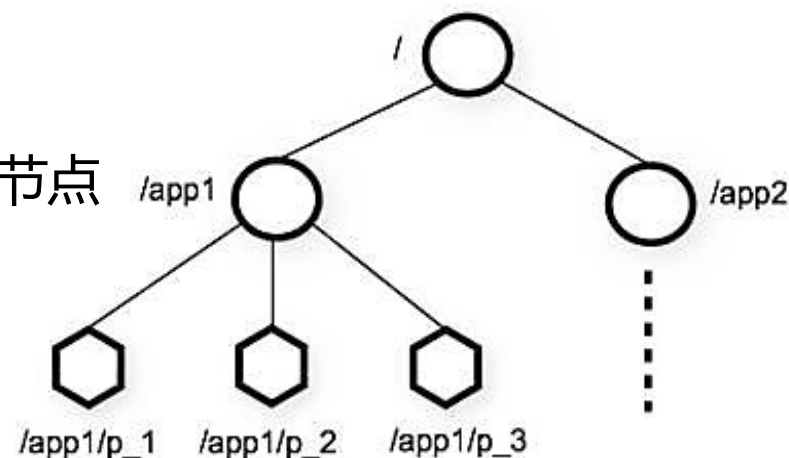


➤ Znode

- ZK的最小数据单元
- 节点有数据存储功能（非数据库），大小不能超过1M
- 节点中的数据应尽可能小，数据过大会导致ZK性能明显下降
- 如果确实要存储较大的数据，可将其放在数据库中，而存储地址放在Znode中
 - 以HBase为例，元数据放在RegionServer中，元数据的寻址入口放在ZK中

➤ Znode Tree

- Znode通过挂载子节点而形成的一个树状层次化命名空间
- 结构类似于Linux文件系统，但没有目录和文件，只有节点和子节点
- 绝对路径（非相对路径）作为节点名称，用于唯一标识节点
- 根节点为 “/” ，非根节点为不能以 “/” 结尾



➤ 会话 (Session)

- 客户端为实现Znode读写操作而与ZK服务器建立的TCP长连接，即会话
- 在SessionTimeout时间内，客户端会通过心跳检测 (Ping) 或发送读写请求来激活和保持会话，但服务器要是是一直未收到客户端消息，就会判定超时并关闭会话
- 在某一会话中，客户端请求以FIFO方式顺序执行

➤ Znode节点类型

- 持久节点 (PERSISTENT)
 - 节点默认类型，生命周期不依赖于客户端会话，只有客户端执行删除操作时，节点才会消失
 - 可以拥有子节点，也可以是叶节点
- 临时节点 (EPHEMERAL)
 - 生命周期依赖于客户端会话，当会话结束时，节点会自动删除 (也可手动删除)
 - 不能拥有子节点，只能是叶节点

➤ Znode节点类型

• 顺序节点 (SEQUENTIAL)

- 带顺序编号的持久或临时节点
- 创建顺序节点时，在路径后面会自动添加一个10位的节点编号（计数器），如
 <path1>0000000001, <path2>0000000002,, 该编号在同一父节点下是唯一的

• 持久顺序节点 (PERSISTENT_SEQUENTIAL)

• 临时顺序节点 (EPHEMERAL_SEQUENTIAL)

➤ Znode版本

- Znode版本是形如0,1,2,...,N的单调递增数字，不是形如v1.2.1的软件版本
- dataVersion (数据版本)
 - 当对Znode中的数据进行更新操作时，dataVersion自增1（即使数据值未发生变化）
- cVersion (子节点版本)
 - 当Znode的子节点有变化时，cVersion自增1

➤ Znode版本

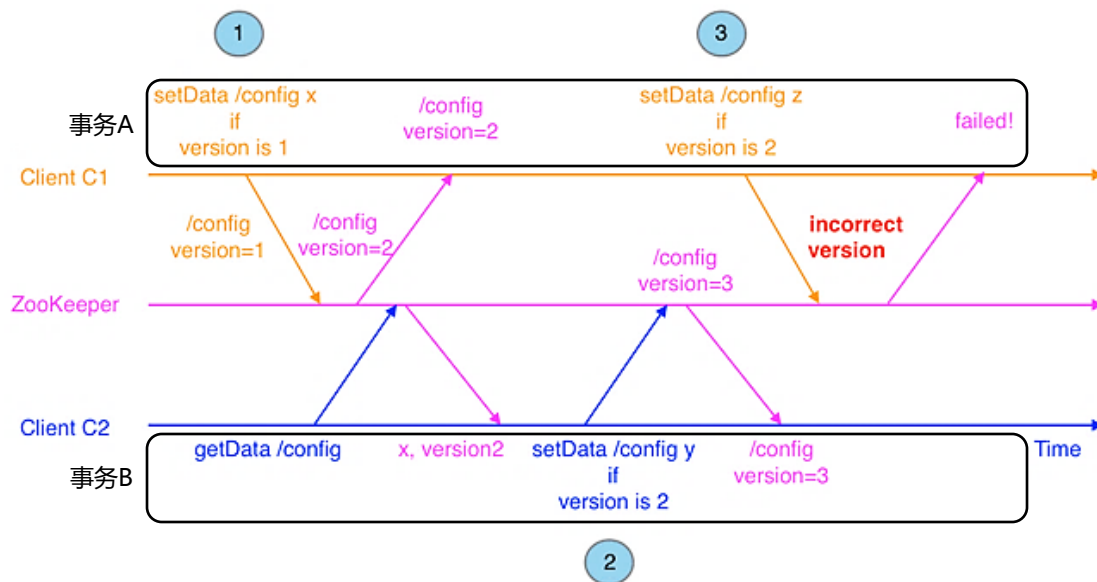
- aclVersion (ACL版本)

- 当Znode的ACL (Access Control List, 访问控制列表) 发生变更时, aclVersion自增1

- 利用版本确保分布式事务操作的原子性

- 悲观锁: 事务A执行期间, 数据全程加锁, 其他事务只能等待; 适用于数据更新并发度较高的场景

- 乐观锁: 事务A提交更新请求前, 先检查数据是否被其他事务修改过 (通过比较版本进行写入校验), 如修改过, 则相关事务必须回滚; 适用于数据更新并发度不高的场景



```
// 获取当前请求中的version
version = setDataRequest.getVersion();
// 获取当前服务器上该数据的最新Version
int currentVersion = nodeRecord.stat.getVersion();
If (version != -1 && version != currentVersion) {
    throw new KeeperException.BadVersionException(path);
}
version = currentVersion + 1;
```

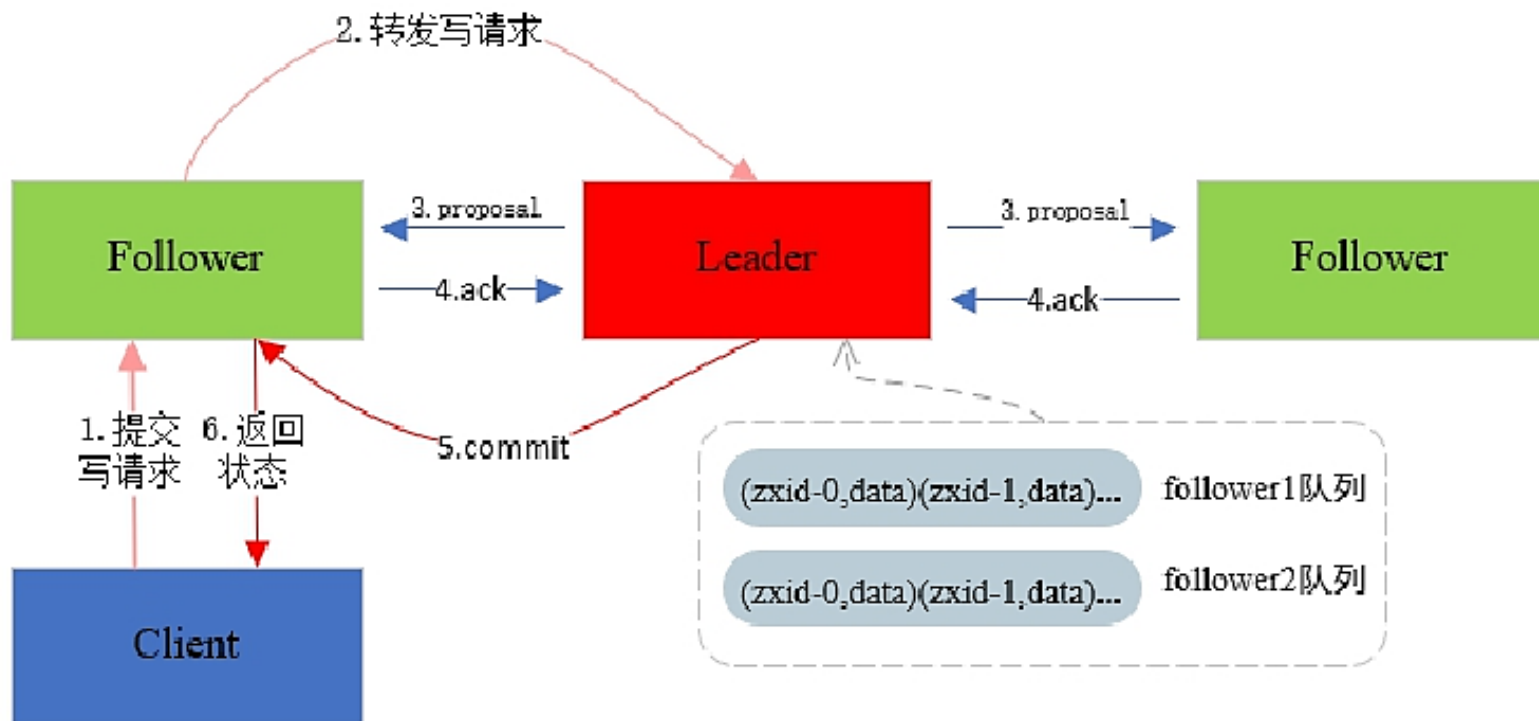

➤ Znode属性

属性	说明
cZxid	Znode创建时的事务ID
cTime	Znode的创建时间
mZxid	Znode最后一次更新的事务ID
mTime	Znode最后一次更新的时间
pZxid	Znode子节点列表最后一次修改时的事务ID
dataVersion	数据版本号
cVersion	子节点版本号
aclVersion	ACL版本号
ephemeralOwner	创建临时节点的会话的SessionID（持久节点的值为0）
dataLength	数据长度
numChildren	子节点个数

➤ ZAB协议

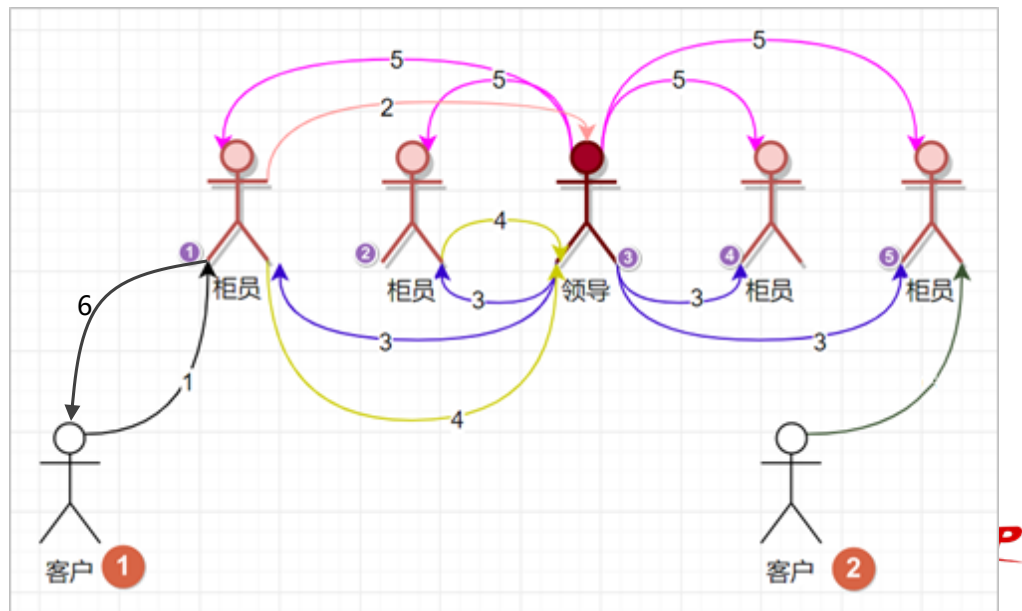
- Zookeeper Atomic Broadcast, ZK原子广播协议
- 基于该协议, ZK实现了Master/Slave架构下集群各节点副本数据的最终一致性
- 包括两种模式: 正常运行时的消息广播模式、Leader故障时的崩溃恢复模式

➤ 数据写入



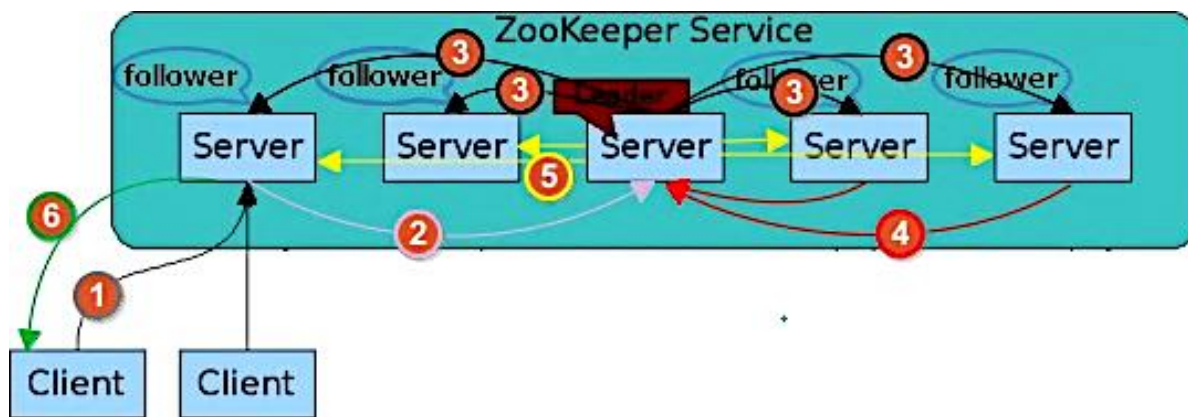
➤ 数据写入 - 举例

- (1) 客户①找到柜员①，说：昨天少给我存了1000，现在要加回来
- (2) 柜员①说：对不起先生，我没权决定，请稍等，向领导柜员③汇报一下
- (3) 柜员③收到消息后，经查账发现是搞错了，但按照规定必须向柜员①②④⑤征求意见
(广播Proposal)
- (4) 柜员①②反馈同意，柜员④⑤还在忙其他事情，但因为已经过半数的柜员（包括Leader）同意，所以Leader做出决定同意补钱（事务Commit）
- (5) 柜员③告知所有下属，登记此事并生效
- (6) 柜员①答复客户①，给您账号里加了1000



➤ 数据写入 - 基本步骤

- (1) Follower接收客户端写请求，并将其转发给Leader
- (2) Leader接收写请求，为其分配一个全局唯一的事务ID, Zxid (64位/单调递增)
- (3) Leader生成一个形如(Zxid, data)的事务提案Proposal (data是事务体)，并将其放入各Follower对应的FIFO队列中 (通过TCP协议实现)，再按照FIFO策略把Proposal广播出去
- (4) Follower接收Proposal后，先以事务日志的形式落盘，再向Leader发送ack
- (5) 当Leader接收到超过半数的ack之后 (包括Leader自己)，会向Follower发送commit命令，要求提交事务，同时自己在本地commit
- (6) Follower收到commit命令后提交事务，同时向客户端反馈结果



➤ 数据恢复

(1) 当Leader无法提供服务时，快速选举出一个新Leader（算法见3.4节）

- 原则：在所有参选Follower中，新Leader拥有Zxid最大的已提交事务
- 确保数据最新：拥有最新的事务数据
- 确保性能最优：软硬件条件最好、数据处理能力最强

(2) Follower与新Leader同步数据

- 原则：对于某个事务，如果至少一个Follower接收到Proposal并最终执行了commit，那么该事务数据就应该被永久保留
- 例如：如果Leader把某个Proposal广播给所有Follower之后宕机，数据该怎么处理？

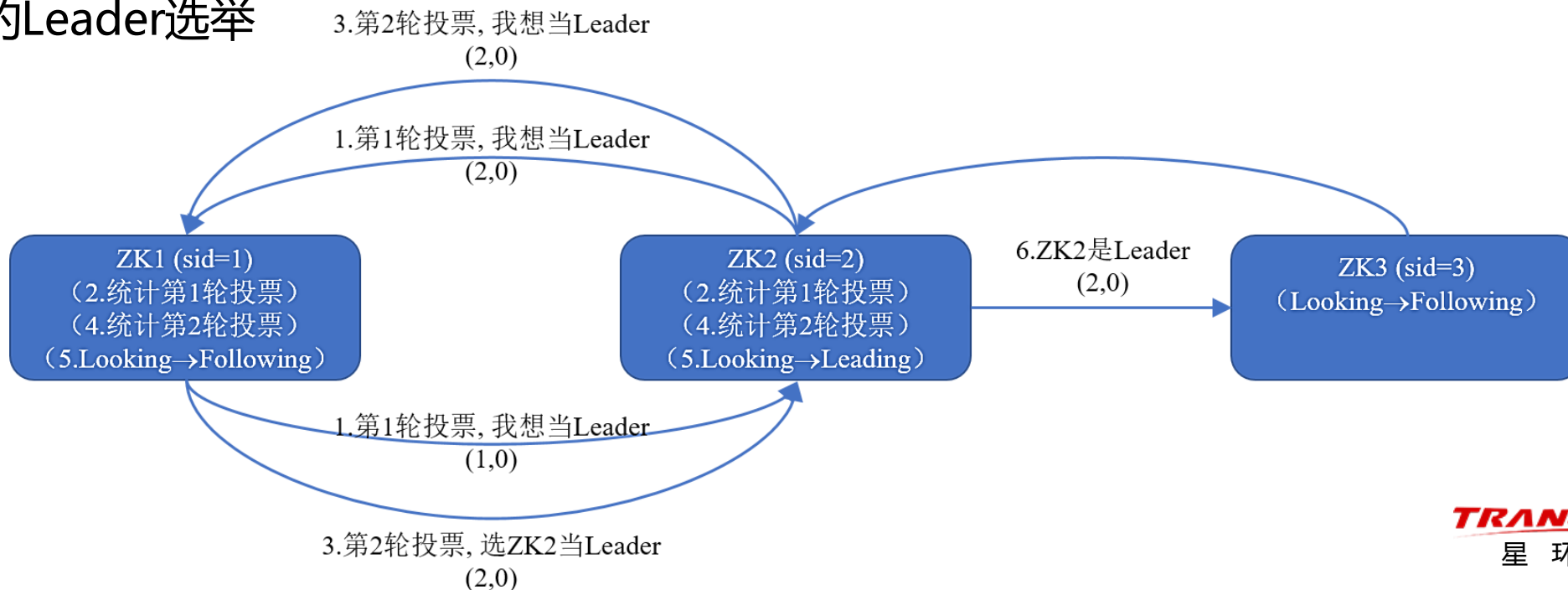
➤ 数据读取

- 客户端直接从Follower或Observer读取数据
- 如果要确保读到最新数据，应该先调用sync()进行强制同步

➤ 服务器的四种状态

- LOOKING: 寻找Leader状态, 表示当前集群没有Leader, 需要进行选举
- LEADING: 领导者状态, 当前角色为Leader
- FOLLOWING: 跟随者状态, Leader选举已完成, 当前角色为Follower
- OBSERVING: 观察者状态, 当前角色为Observer, 不参与选举和写事务

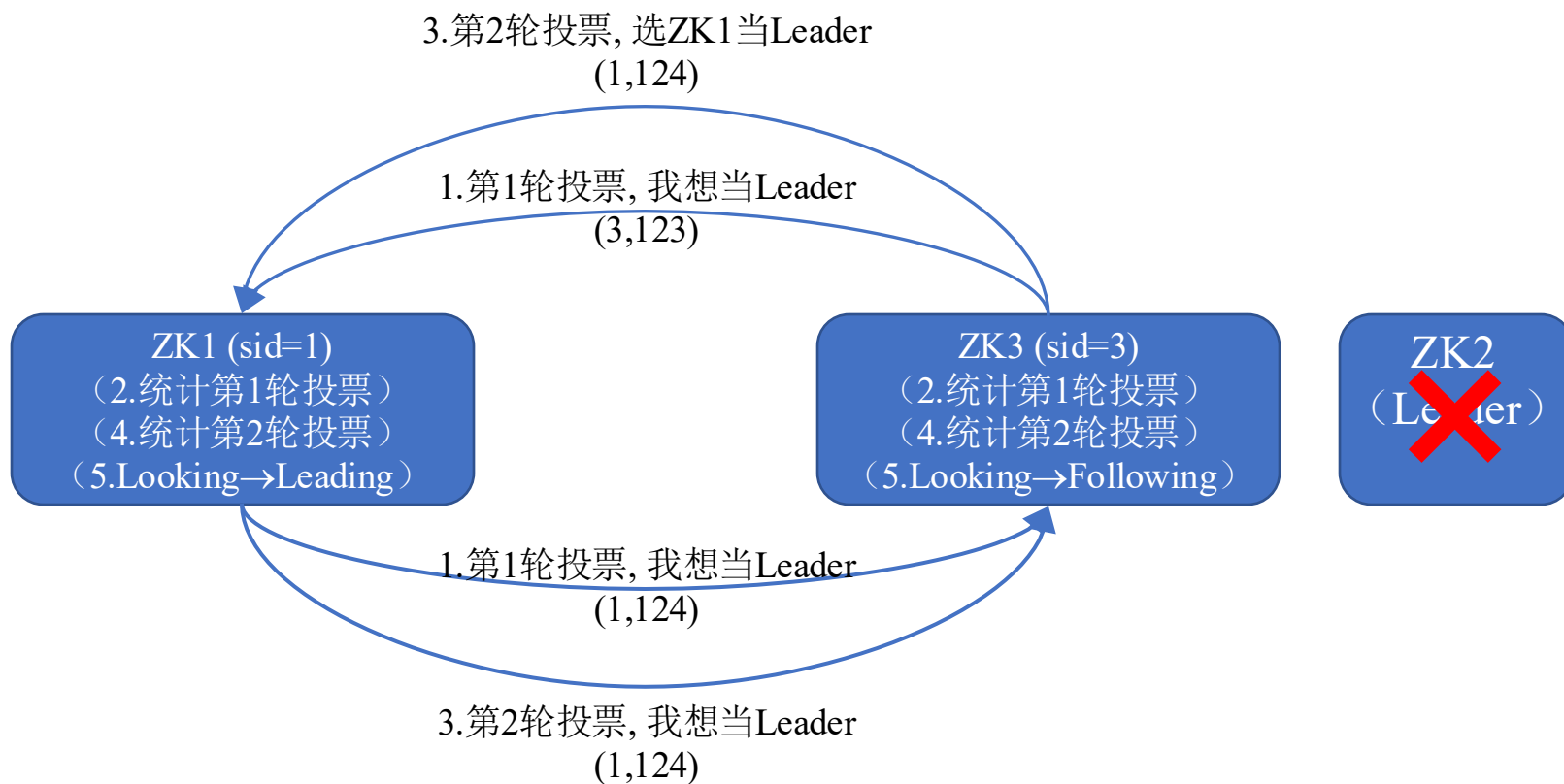
➤ 启动期间的Leader选举



启动期间的 Leader 选举

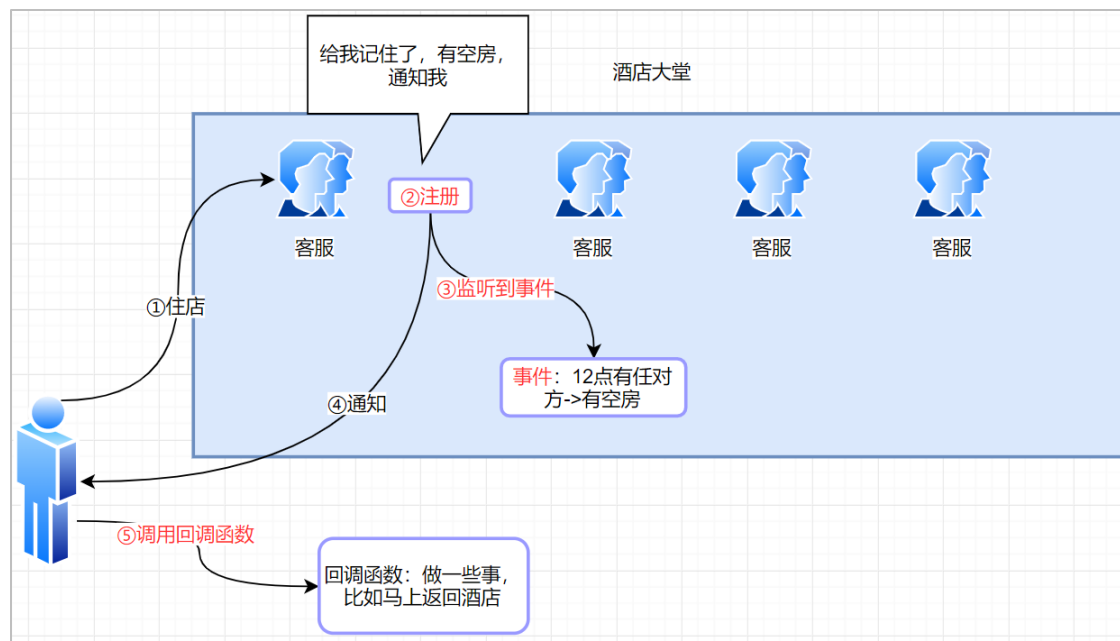
- ZK1 启动: 投票给自己 (1,0), 因未达多数派 (集群需要 ≥ 2), 继续保持 LOOKING。
- ZK2 启动:
 - 初始时 ZK1 投 (1,0), ZK2 投 (2,0);
 - 投票交换后, 按规则 (2,0) 优先, ZK1 更新为投 (2,0);
 - 虽然此时已形成多数派趋势, 但需要新一轮确认;
 - 下一轮投票时, 两者一致投 (2,0), 多数派确认, ZK2 成为 Leader, ZK1 成为 Follower。
- ZK3 启动: 进入 LOOKING, 广播投票; 收到 Leader 信息后, 确认已有 Leader, 切换为 FOLLOWING 并进行数据同步。

➤ 运行期间的 Leader 选举




➤ 工作机制：观察者模式在分布式场景下的实现

- (1) 注册Watcher监听器：客户端向ZK的某个Znode注册一个Watcher监听器
- (2) 存储Watcher对象：客户端把Watcher对象存储到本地的WatchManager中
- (3) 处理Watcher事件：当服务端的指定事件触发了Watcher，会向客户端发送事件通知
- (4) 回调Watcher对象：客户端根据通知状态和事件类型回调WatchManager中的Watcher对象，执行相应的业务逻辑



➤ Watcher特性

特性	说明
一次性	Watcher是一次性的，一旦触发就会被删除，再次使用时需重新注册
轻量级	<ul style="list-style-type: none">• WatchEvent是轻量级的通信单元• 仅包含通知状态、事件类型和节点路径，不包含数据节点变化前后的具体内容
时效性	<ul style="list-style-type: none">• Watcher只有在当前Session结束后才会失效• 如果在Session有效期内快速重连成功，则Watcher依然有效，仍可接收到通知
顺序回调	<ul style="list-style-type: none">• Watcher回调是顺序串行的• 回调逻辑不应太多，以免影响其他Watcher执行• 只有回调后客户端才能看到最新的数据状态



4 chapter

ZooKeeper应用场景

- ✓ 分布式锁
- ✓ 统一命名
- ✓ 配置管理
- ✓ 集群管理

➤ 实现方式一

- 依据

(1) 多个客户端同时创建同一个Znode节点（节点路径名唯一），只有一个客户端能够成功

(2) Watcher机制

- 步骤

(1) 多个客户端同时在ZK上竞争创建临时节点“/Lock”（简称Lock节点），创建成功的客户端获得锁，并执行事务

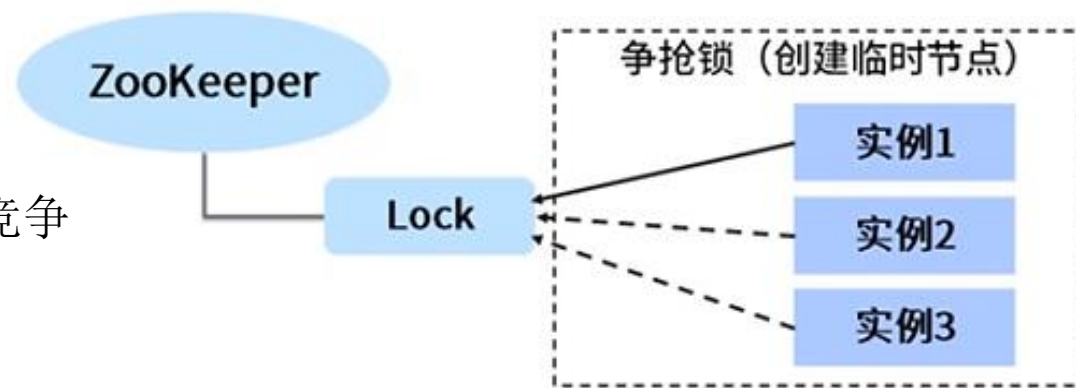
(2) 其他客户端注册Watcher监听器，监听Lock节点

(3) 事务完成后，获得锁的客户端会删除Lock节点，释放锁，同时触发Watcher，通知其他客户端

(4) 其他客户端再次竞争创建Lock节点

- 缺点

– 产生羊群效应，即当锁被释放后，如果抢占锁资源的竞争客户端太多，势必会影响性能



➤ 实现方式二

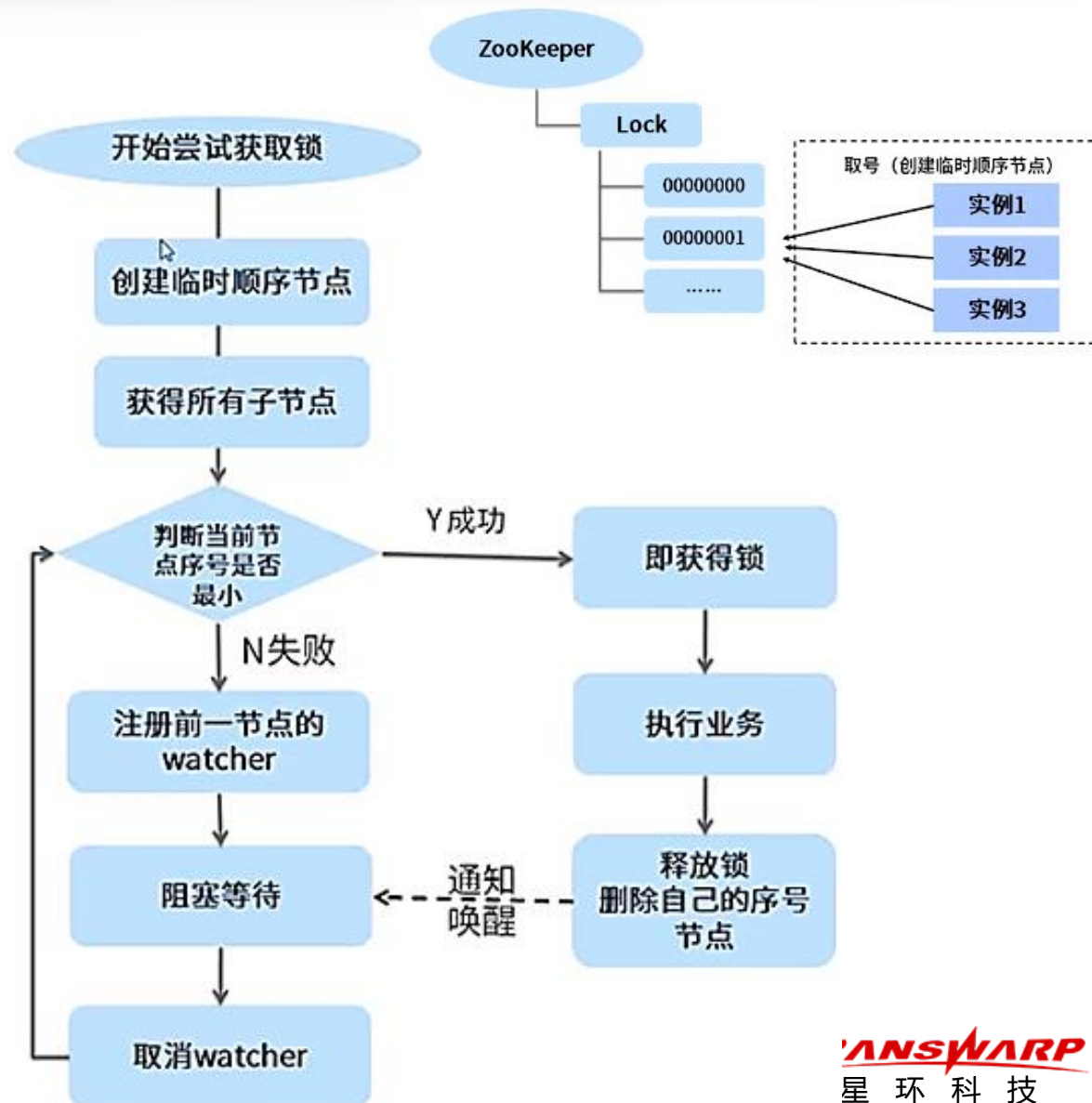
- 依据：临时顺序节点 + Watcher机制

- 步骤

(1) 客户端在永久节点“/Lock”下创建临时顺序子节点，第一个客户端创建的子节点为“/Lock/Lock-0”，第二个为“/Lock/Lock-1”，以此类推

(2) 客户端获取Lock节点的子节点列表，判断其创建的子节点的序号是否最小，如果是则获得锁，否则就监听序号排在其前一位的子节点

(3) 锁释放后，对应的子节点被删除，该节点序号后一位的子节点得到通知，重复步骤②直至获得锁

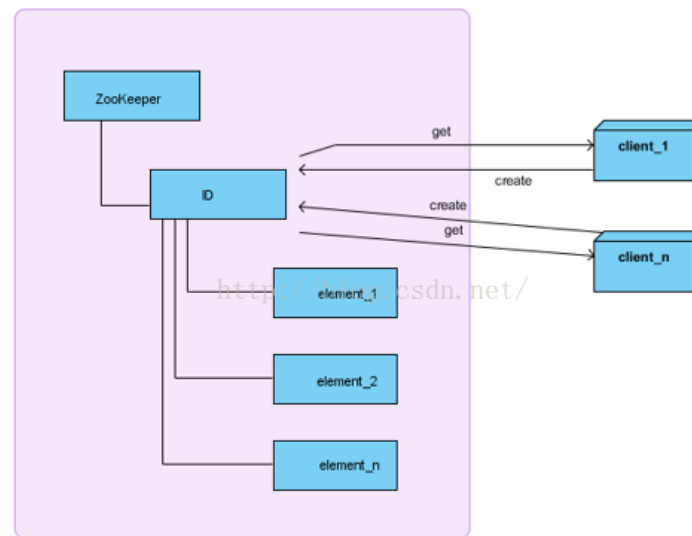
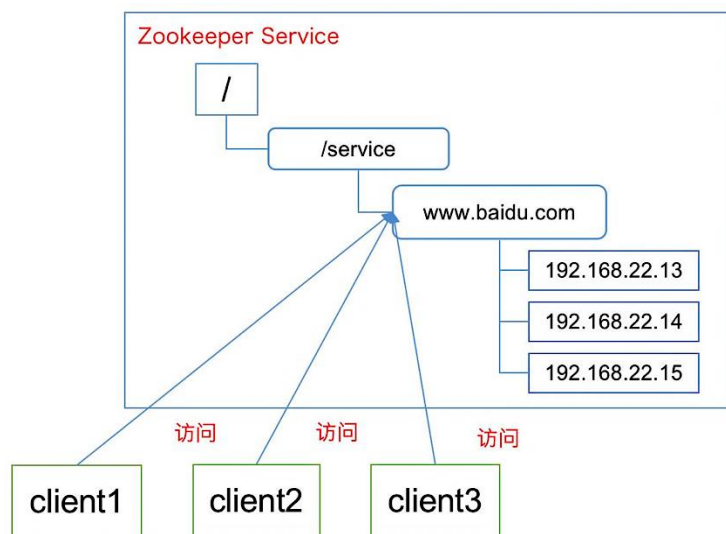


➤ 服务统一命名

- 类似于JNDI，记录域名与IP之间的对应关系，域名作为访问入口
- 按照Znode Tree层次结构组织服务名称
- 将服务的地址、目录和提供者等信息存入Znode，通过服务名称来获取相关信息

➤ 全局唯一ID

- UUID虽然可以保证分布式环境下的编号唯一，但缺点是无序、存储空间大和查询效率低
- 利用Znode顺序节点，可以实现分布式环境下的编号生成器（ID生成器）

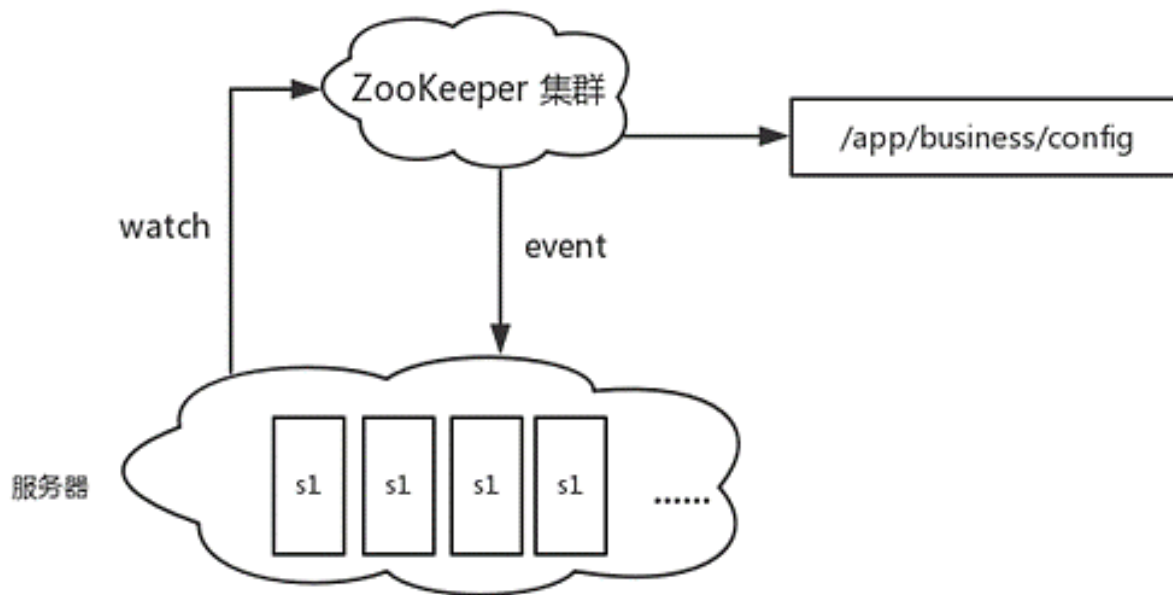


➤ 分布式环境下的配置信息同步

- 集群中所有节点的配置信息需保持一致
- 配置信息修改后，应快速同步到其他节点上

➤ 利用Znode和Watcher实现统一配置管理

- 将配置信息写入Znode
- 各节点监听Znode，一旦Znode的数据被修改，将通知各节点进行更新



- 实时掌握分布式系统中各节点的状态是集群管理的前提和基础
- 利用Znode和Watcher实现集群管理
 - 将节点的状态信息写入Znode，利用Watcher监听Znode，以获取节点的实时状态变化
 - 节点上下线动态监测
 - 新节点启动后，先在ZK中创建临时顺序Znode，这时会触发父节点上的监听器，并通知集群管理节点Master有新节点上线
 - 节点发生故障，失去与ZK的心跳连接，它创建的临时顺序Znode被自动删除，这时会触发父节点上的监听器，并通知Master节点下线
 - Master选举
 - 如果Master节点宕机，失去与ZK的心跳连接，那么它创建的临时Znode被自动删除，这时会触发该节点的Watcher，并通知所有Standby节点去竞争创建临时Znode
 - 成功创建临时Znode的Standby节点成为新的Master
 - 其他Standby节点在该Znode上注册监听器，等待下一次选举



Q&A

TRANSWARP
星环科技

温故知新

- 为什么说BASE理论是CAP定理的延伸？
- 请简述ZAB协议的基本内容。
- 为什么ZK的Leader和Follower数量最好是奇数，而不是偶数？
- 为什么要在ZK引入Observer这一角色？
- 请简述ZK数据写入的基本步骤。
- 请简述ZK运行期间的Leader选举过程。
- 如何利用临时顺序节点实现分布式锁？

