**概述**

zookeeper 是一个分布式协调服务的开源框架。主要用来解决分布式集群中应用系统的一致性问题。例如怎么避免同时操作统一数据造成脏读的问题。

zookeeper 本质上市一个分布式小文件存储系统。 提供基于类似于文件系统的目录树方式的数据存储，并且可以对书中的节点进行有效管理，从而用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理。 注入： 统一命名服务、分布式配置管理、分布式消息队列、分布式锁、分布式协调等功能。

zookeeper=文件系统+通知机制

**特性**

一个leader，多个follower组成的集群。集群中只要有半数以上节点存活，集群就能正常服务；

1. 全局数据一致： 每个server保存一份相同的数据副本，client无论连接到哪个server，展示的数据都是一致的。 这是最重要的特性
2. 可靠性：如果消息被其中一台服务器接收，那么将被所有服务器接收
3. 顺序性：包括全局有序和偏序两种； 全局有序是指如果一台服务器行消息a比b先发布，那么所有服务器上a在b之前发布；偏序是指如果一个消息b在消息a后被同一个发送者发布，a 必排在b前面
4. 数据更新原子性： 一次数据更新要么成功（半数以上节点成功），要么失败，不存在中间状态
5. 实时性： zookeeper 保证客户端将在一个时间间隔范围内，获得服务器的消息更新，或者服务器失效的信息。

**集群角色**

**leader：**

zookeeper 集群工作的核心

事务请求（写操作）的唯一调度和处理这，保证集群事务处理额顺序性

集群内部各个服务器的调度者

对于create/ serdata delete 等有写操作的请求，则需要统一转发给leader处理，leader需要决定编号，执行操作，这个过程称为一个事务

**follower：**

处理客户端非事务（读操作）的请求，转发事务请求给leader

参与集群leader 选举投票

**observer：**

此外，如果集群很大，可以添加观察者角色

观察者角色，观察zookeeper集群的最新状态变化并将这些状态同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则会转发给leader服务器进行处理。

不会参与任何形式的投票，只提供非事务服务。 通常用于在不影响集群事务处理的前提下提升集群的非事务处理能力。

**集群搭建**

1. 需要安装jre。检查时间同步。配置主机名称到IP地址映射配置。监测防火墙是否关闭。
2. 从官网下载对应的版本，解压到/opt/
3. 修改zookeeper配置文件
   1. cp zoo\_sample.cfg zoo.cfg
   2. 修改如下配置：

datadir=/var/lib/zookeeper

server.1=ceph-93:2888:3888 (心跳端口，选举端口)

server.2=ceph-94:2888:3888

server.3=ceph-95:2888:3888

1. 在数据目录设置myid,注意id 跟配置文件中要匹配
2. 启动zookeeper

/opt/zookeeper/bin/zkServer.sh start

或者配置成服务文件，用systemctl 来控制

[Unit]

Description=zookeeper

After=network.target

[Service]

User=zookeeper

WorkingDirectory=/var/lib/ali/zookeeper

ExecStart=/opt/zookeeper/bin/zkServer.sh start

ExecStop=/opt/zookeeper/bin/zkServer.sh stop

Type=forking

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

1. 查看状态

zkServer.sh status

想用observer模式，可以在对应节点的配置文件添加如下配置：

1. peerType=observer
2. 其次在配置文件中指定哪些文件被指定为observer

server.1:localhost:2888:3888:observer

配置参数：

ticktime: 2000 心跳，2000ms，

initlimit: 10 10个ticktime，leader 跟fowllower 通信时间，启动时 通信延时

syncLimit ： 5 5个ticktime， leader 跟follower 通信时间，启动后允许的通信延时。如果超过，则认为follower掉线

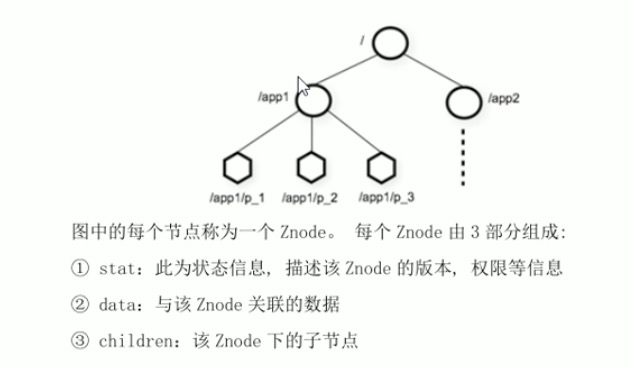
datadir：

ClientPort：

**数据模型**

zookeeper 的数据模型，在结构上和标准文件系统非常相似，拥有一个层次的命名空间，都是采用树形层次结构，zookeeper树种的每个节点被称为有个znode。 和文件系统的目录树一样，zookeeper树种的每个节点可以拥有子节点。

一个znode只能存1M数据



不同之处：

1. znode兼具文件和目录两种特点，既像文件一样维护着数据目录、原信息、acl、时间戳等数据结构，又像目录一样可以作为路径表示的一部分，并可以具有子znode。用户对znoode具有增删改查等操作
2. znode具有原子性操作，读操作讲获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。 另外每个节点都拥有自己的acl，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。
3. znode存储数据大小有限制。zookeeper 虽然可以关联一些数据，通常保存分布式应用中的配置文件信心，状态信息，通常以kb 为单位
4. zookeeper通过路径引用，路径必须是绝对路径。 节点必须是唯一的。

**节点类型**

znode 有两种分别为临时节点和永久节点

临时节点： 该节点的声明周期依赖于创建他们的绘画，一旦会话结束，临时节点将被自动删除（也可以手动删除）。临时节点不允许拥有子节点。

永久节点： 该节点的声明周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才被删除

znode 还有一个序列化的特性，如果创建的时候指定的话，改节点的名字后面会自动追加一个不断增加的序列号。

这样变回存在四种类型的znode节点，分别是：

PERSISTENT: 永久节点

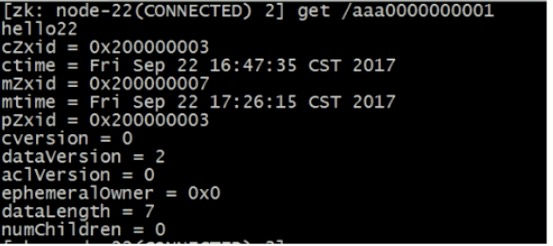
EPHEMERAL: 临时节点

persistent\_sequential: 永久节点、序列化

ephemeral\_sequential： 临时节点、序列化

**节点属性**

znode都包含了一系列的属性，通过get命令，可以获得节点的属性。



dataversion： 数据版本号，每次对节点进行set操作，dataversion的值都会增加1. 即使设置的是相同数据。 可以优先避免数据更新是出现先后顺序的问题

cversion：子节点的版本号。当znode的子节点有变化时，cversion的值就会增加1

aclversion： acl的版本号

czxid： znode创建的事务id

mzxid： znode被修改的事务id，即每次对znode的修改都会更新mzxid

对于zk来说，每次你的变化都会产唯一的事务id，通过id可以确定更新操作的先后顺序。

ctime： 创建时间

mtime： 节点最新一次更新发生的时间

ephemeralOwner： 如果该节点为临时节点，该值便是与改节点绑定的session id。 如果不是，值为0

**zookeeper shell**

./bin/zkCli.sh -server ip 默认连接本机

连上后 输入help 查看可执行的操作

**创建节点**：

create [-s] [-e] path data acl

-s 表示序列化

-e 表示临时节点

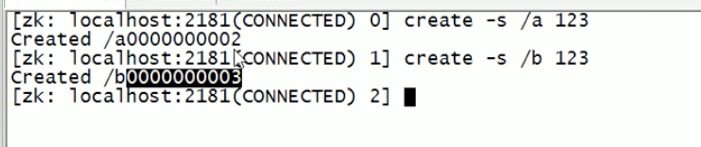
创建永久节点

create /hellozk 123

创建临时节点. 断开连接后果几秒才自动删除

create -e /zktmp 123456

创建序列化节点



**读取节点：**

ls path [watch] 可以查看第一级的子节点

get path [watch] 可以获得指定节点的内容和属性

ls2 path [watch] 查看节点的属性，子节点

**更新节点：**

set path value [version]

**删除节点**

delete path [version]

如果节点存在子节点，需要先删除子节点。

rmr path 可以递归删除节点。

**quota：**

setquota -n| -b val path 对接点增加限制

n: 表示子节点的最大个数，这个计数包括他自己。如果实际子节点超过了限制，也能创建成功；只会在日志里面有WARN

b: 表示数据值的最大长度

val 子节点最大个数活数据值的最大长度

path 节点路径

listquota path 列出指定节点的quota

delquota [-n|-b] path 删除quota

**其他命令**

history 列出命令历史

redo 该命令可以重新执行指定的命令，命令编号可以用history 查看

**watcher**

zookeeper 提供了分布式数据发布/订阅功能，一个典型的发布、订阅模型系统定义了一种一对多的订阅关系，能让多个订阅者同时监听某一个主体对象，当这个主题对象自身状态变化是，会通知所有订阅者，是他们能够做出想听的处理。

zookeeper 中，引入了watcher机制，来实现这种分布式通知的功能。 zookeeper允许客户端向服务端注册一个watcher监听，当服务端的一些事情触发了这个watcher，那么就会想指定的客户端发送一个时间通知来实现分布式的通知功能。

触发时间的种类很多，如：节点创建，节点书暗处，节点改变，子节点改变等。

总的来说，可以概括watcher 为一下三个过程： 客户端向服务端注册watcher，服务端时间发生触发watcher，客户端回电watcher 得到出发时间的情况。

**watcher 机制特点：**

**一次性触发：** 时间发生触发监听，一个watcher event 就会被发送到设置监听的客户端，这种效果是一次性的，后续再次发生同样的事情，不会再次触发。

**事件封装：**

zookeeper 使用watchedEvent 对象来封装服务端时间并传递。

watchedEvent 包含了每一个事件的三个基本属性：

通知状态keeperstate、事件类型eventtype、节点路径path

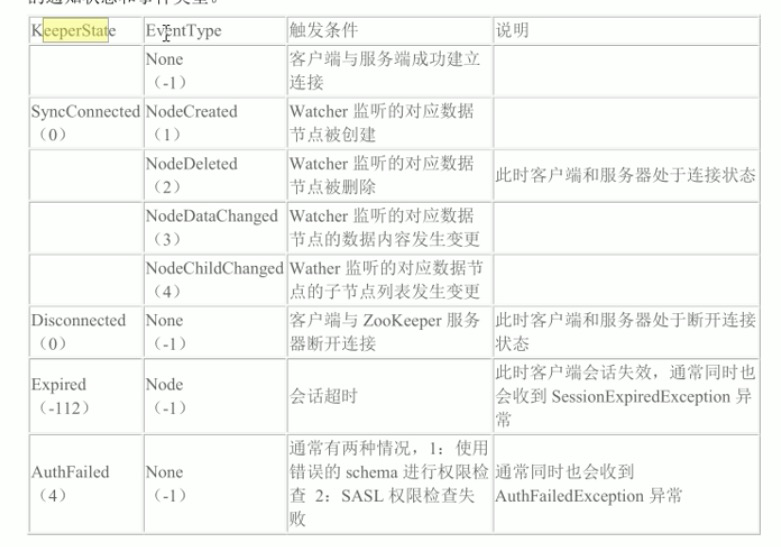
**event异步发送：**

wtcher的通知时间从服务端发送到客户端是异步的。

**先注册再触发：**

zookeeper 中的watch机制，必须客户端先去服务端注册监听，这样时间发生才会通知客户端

**事件类型**



其中连接状态事件（type=None，path=null）不需要客户端注册，客户端只要有需要直接处理就行了

**shell 操作watcher**

命令中带watch 选项的命令才支持设置watcher

设置节点数据变动监听：

create /watchertest 123

get /watchertest watch

另开一个连接

set watchertest 123455

第一个连接会收到

WatchedEvent state:SyncConnected type:NodeDataChanged path:/watchertest

ls /watchertest watch # 监听子节点的变化。

**zookeeper java api**

略

**zookeeper 集群选举**

zookeeper 默认的算法是FastLeaderElection，才去投票数大于板书则胜出的逻辑

服务器ID：

编号越大在选择算法中权重越大

选举状态：

looking，竞选状态

following， 随从状态，同步leader状态，参与投票

observing，观察状态，同步leader状态，不参与投票

leading 领导者状态。

数据ID

服务器中存放的最新数据version

值越大说明数据越新，在选举算法中权重越大

逻辑时钟

也叫投票的次数，同一轮投票过程中的逻辑时钟是相同的。 每投完一次票，这个数据就会增加，然后与接收到的其他服务器返回的投票信息中的数值相比，根据不同的值做出不同的判断。

**全新集群**

每个机器都给自己投票，投票数过半选举结束。

假设目前有5台机器，每台服务器均没有数据，他们的编号分别是1、2、3、4、5。 选举过程如下

1. 服务器1 启动，给自己投票，然后发投票信息，由于其他机器还没有启动，所以他收不到反馈信息，服务器的状态一直属于looking
2. 服务器2 启动，给自己投票，同时与之前启动的服务器1交换结果，由于服务器2 的编号大，所以服务器2胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是looking状态。
3. 服务器3 启动，给自己投票，同时与之前启动的服务器1、2交换信息，由于服务器的编号最大，所以服务器3 胜出。 此时投票数正好大于板书，所以服务区3 成为领导者，服务器1、2成为follower
4. 服务器4 启动，给自己投票。 同时与之前启动的1、2、3 交换信息，尽管服务器4 的编号大，但之前服务器3 已经胜出，所以服务器4 成为小弟
5. 服务器5 启动后，后面的逻辑通服务器4

**非全新集群**

对于运行正产的zookeeper集群，中途有机器down掉，需要重新选举时，选举过程就需要加入数据ID、服务器ID、和逻辑时钟。

数据ID： 数据新的version就大，数据每次更新都会更新version

服务器ID： 就是我们配置的myid的值，每个机器一个

逻辑时钟： 这个值从0 开始，每次选举对应一个值。 如果在同义词选举中，这个值是一直的。

这样选举标准就变成：

1. 逻辑时钟小的选举结果被忽略。
2. 统一逻辑时钟后，数据id大的胜出；
3. 数据id 相同的情况下，服务器id 大的胜出。

**zookeeper 典型应用**

统一命名服务，统一配置管理，统一集群管理，服务器节点动态上下线，软负载均衡

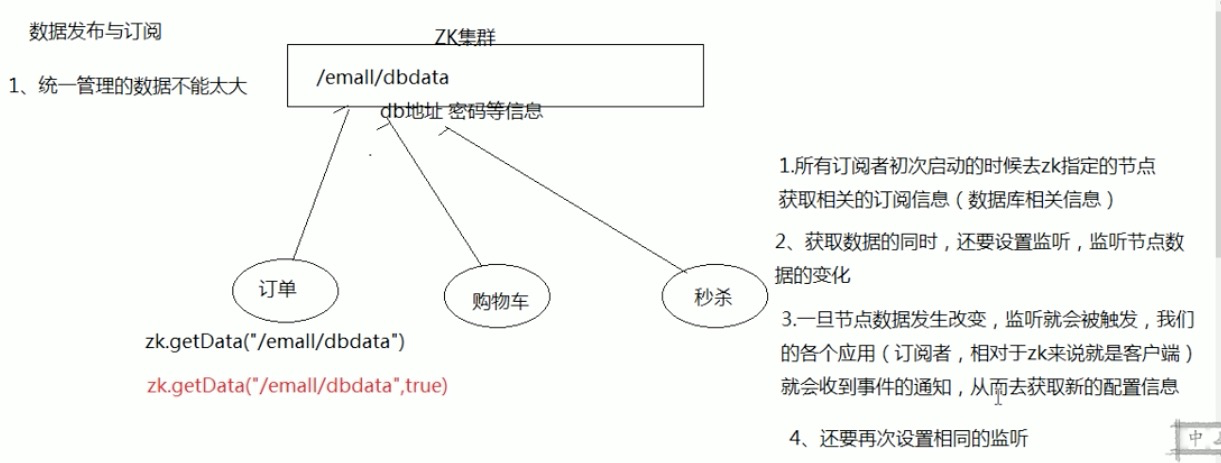
**数据发布与订阅（配置中心）**

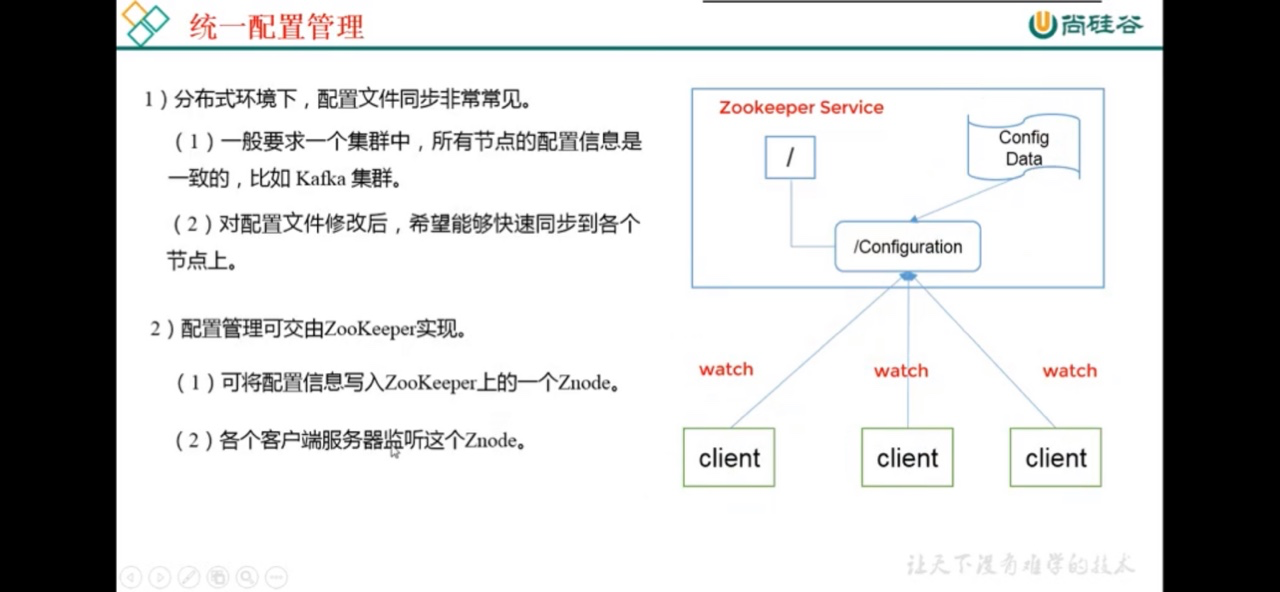
发布与订阅模型，即所谓的配置中心，顾名思义就是发布者讲数据发布到zk节点上，供订阅者动态获取数据，实现配置信息的集中式管路与动态更新。

应用在启动的时候回主动来获取一次配置，同时，在节点上注册一个watcher，这样一来，以后每次配置有更新的时候，都会实时通知到订阅的客户端，从而达到获取最新配置信息的目的。比如：

分布式搜索服务中，索引的源数据和服务器集群机器的阶段状态存在zk的一些指定节点，供各个客户端订阅使用。

注意： 适合数据量很小的场景，这样数据更新可能会比较快。





**命名服务**

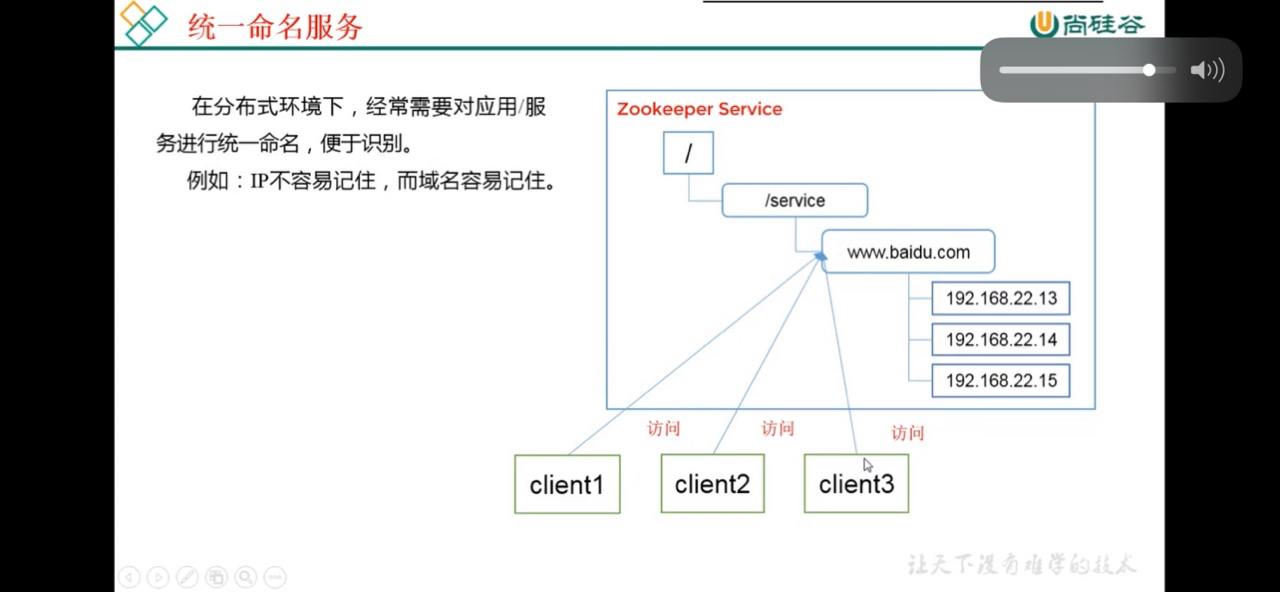
在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。 被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等。 这些我们都可以统称他们为名字（name）。 其中较为常见的就是一些分布式服务框架中的服务地址列表。 通过调用zk提供的创建节点的API，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。

阿里巴巴几圈开源的分布式服务框架dubbo中使用zookeeper来作为其命名服务，维护全局的服务地址列表

from 另一篇文章

在分布式环境下，经常需要对应用进行统一命名，便于识别。

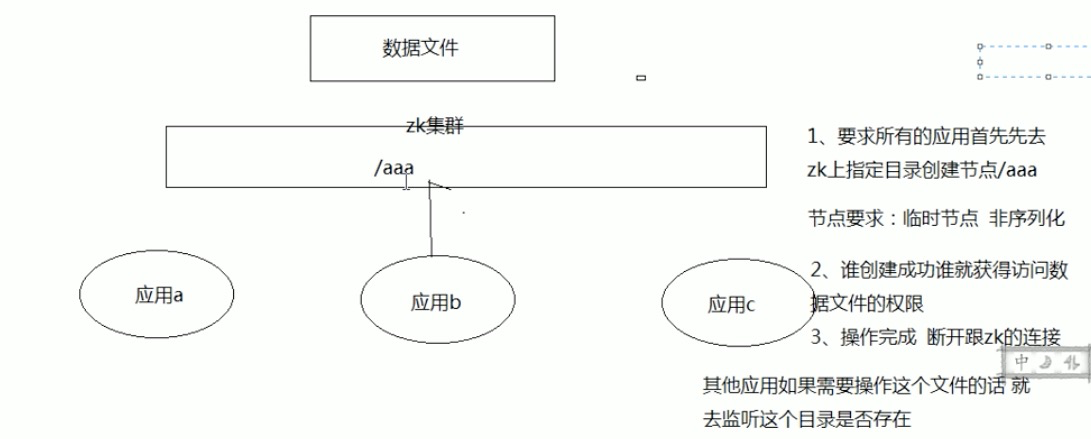
例如IP不容易记住，而域名容易记住。



**分布式锁**

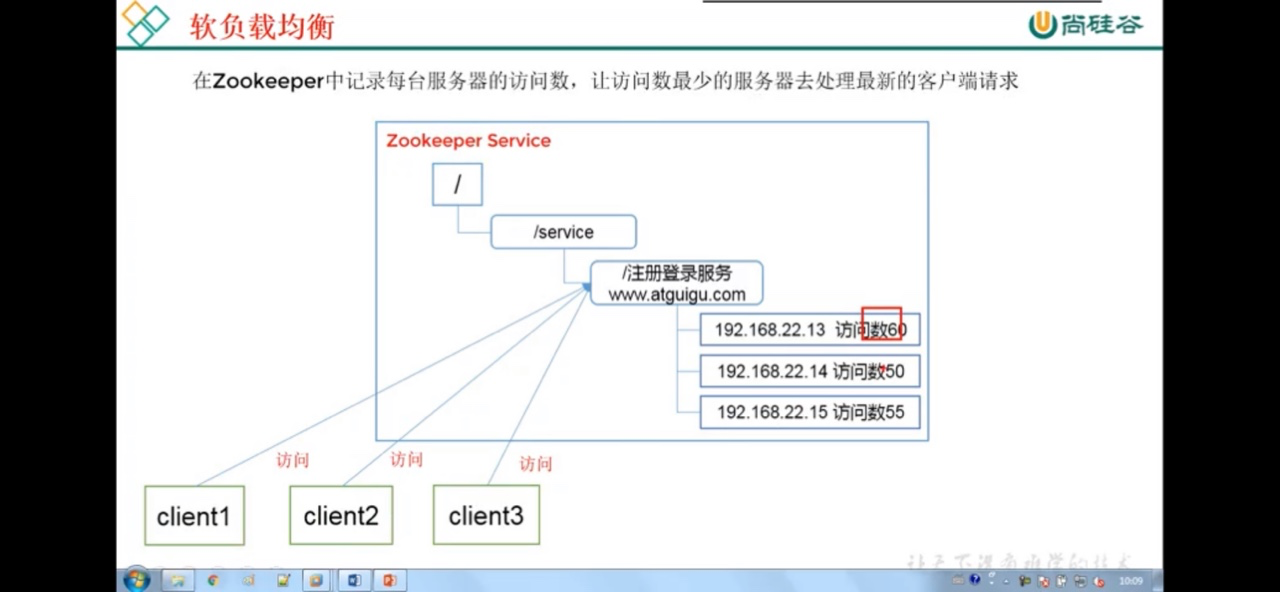
分布式锁，这个主要得益于zookeeper保证了数据的强一致性。 锁服务可以分为两类，一个是保持独占，另一个是控制时序。

所谓保持独占，就是所有视图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法就是把zk上的一个znode看做是一把锁，通过create znode 的方式来实现。 所有客户端都去创建/distribute\_lock 节点，最终成功创建的那个客户单也即拥有了这把锁。

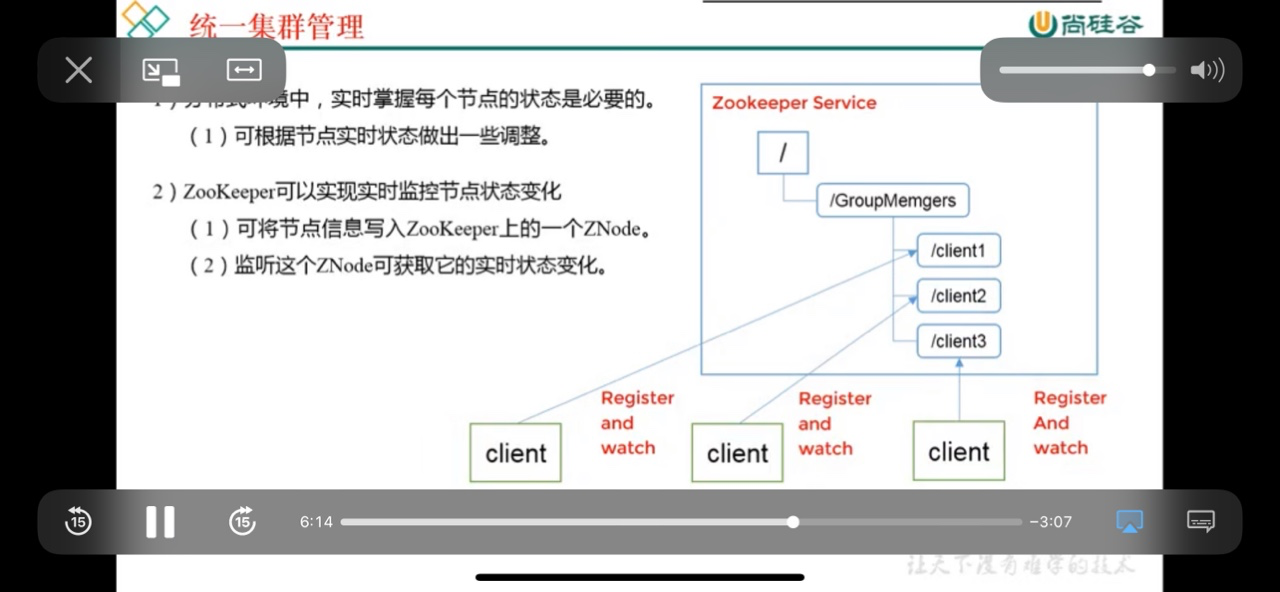


控制时序，就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局的时序了。 做法和上面的基本类似，只是这里/distribute\_lock 已经预先存在，客户端在他线面创建临时有序节点（这个可以通过节点的属性控制：createMode.Ephemeral.\_sequential来指定）。 zk的父节点（/distribute\_lock） 维持一份唯一sequence，保证子节点创建的时序性，从而也形成了每个客户端的全局时序

**软负载均衡**



**统一集群管理**



**服务器动态上下线**

