# 相关概念

### zk的数据结构

ZooKeeper 数据模型的结构与Unix 文件系统很类似，整体上可以看作是一棵树，每个节

点称做一个 ZNode。

ZooKeeper的数据模型是层次模型，层次模型常见于文件系统。层次模型和key-value模型

是两种主流的数据模型。ZooKeeper使用文件系统模型主要基于以下两点考虑:

1. 文件系统的树形结构便于表达数据之间的层次关系

2. 文件系统的树形结构便于为不同的应用分配独立的命名空间( namespace )

ZooKeeper的层次模型称作Data Tree，Data Tree的每个节点叫作Znode。不同于文件系

统，每个节点都可以保存数据，每一个 ZNode 默认能够存储 1MB 的数据，每个 ZNode

都可以通过其路径唯一标识，每个节点都有一个版本(version)，版本从0开始计数。

### 2、节点分类

一个znode可以使持久性的，也可以是临时性的:

1. 持久节点(PERSISTENT): 这样的znode在创建之后即使发生ZooKeeper集群宕机或

者client宕机也不会丢失。

2. 临时节点(EPHEMERAL ): client宕机或者client在指定的timeout时间内没有给

ZooKeeper集群发消息，这样的znode就会消失。

如果上面两种znode具备顺序性，又有以下两种znode :

3. 持久顺序节点(PERSISTENT\_SEQUENTIAL): znode除了具备持久性znode的特点之

外，znode的名字具备顺序性。

4. 临时顺序节点(EPHEMERAL\_SEQUENTIAL): znode除了具备临时性znode的特点之

外，zorde的名字具备顺序性。

zookeeper主要用到的是以上4种节点。

5. Container节点 (3.5.3版本新增)：Container容器节点，当容器中没有任何子节点，

该容器节点会被zk定期删除（定时任务默认60s 检查一次)。和持久节点的区别是 ZK 服务端启动后，会有一个单独的线程去扫描，所有的容器节点，当发现容器节点的子节点数量为0时，会自动删除该节点。可以用于 leader 或者锁的场景中。

### 3、节点状态信息

cZxid ：Znode创建的事务id。

ctime：节点创建时的时间戳。

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，每次的变化都会产生一个唯一的事务id，zxid（ZooKeeper Transaction

Id），通过zxid，可以确定更新操作的先后顺序。例如，如果zxid1小于zxid2，说明zxid1

操作先于zxid2发生，zxid对于整个zk都是唯一的，即使操作的是不同的znode。

pZxid: 表示该节点的子节点列表最后一次修改的事务ID，添加子节点或删除子节

点就会影响子节点列表，但是修改子节点的数据内容则不影响该ID（注意: 只有子节

点列表变更了才会变更pzxid，子节点内容变更不会影响pzxid）

mtime：节点最新一次更新发生时的时间戳.

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。

dataVersion：数据版本号，每次对节点进行set操作，dataVersion的值都会增加1（即使设置的是相同的数据），可有效避免了数据更新时出现的先后顺序问题。

ephemeralOwner:如果该节点为临时节点, ephemeralOwner值表示与该节点绑定的session id。如果不是, ephemeralOwner值为0(持久节点)。

在client和server通信之前,首先需要建立连接,该连接称为session。连接建立后,如果发生连

接超时、授权失败,或者显式关闭连接,连接便处于closed状态, 此时session结束。

dataLength ： 数据的长度

numChildren ：子节点的数量（只统计直接子节点的数量）

### 4、监听通知（watcher）机制

一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。

Zookeeper采用了Watcher机制实现数据的发布订阅功能，多个订阅者可同时监听某一特定主题对象，当该主题对象的自身状态发生变化时例如节点内容改变、节点下的子节点列表改变等，会实时、主动通知所有订阅者。

watcher机制事件上与观察者模式类似，也可看作是一种观察者模式在分布式场景下的实现方式。

watcher的过程：

1. 客户端向服务端注册watcher

2. 服务端事件发生触发watcher

3. 客户端回调watcher得到触发事件情况

注意：Zookeeper中的watch机制，必须客户端先去服务端注册监听，这样事件发送才会触发监听，通知给客户端。

支持的事件类型：

None: 连接建立事件

NodeCreated： 节点创建

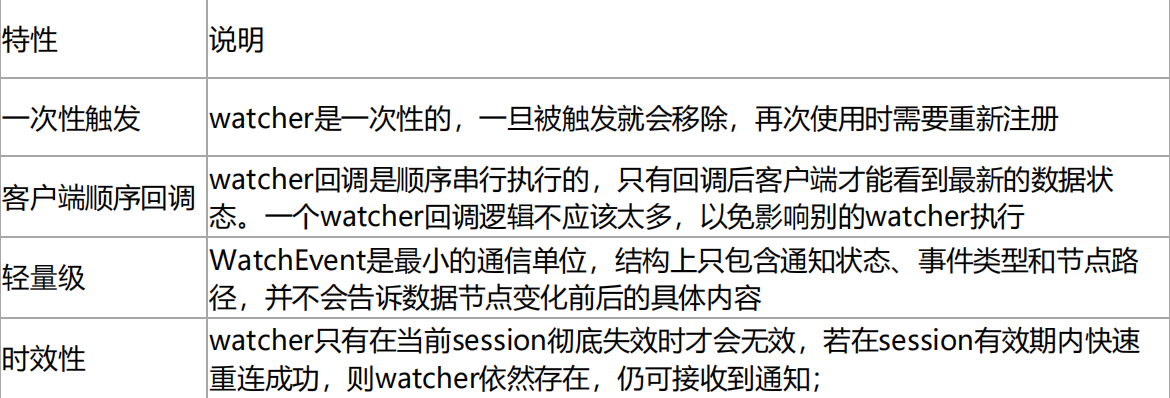
NodeDeleted： 节点删除

NodeDataChanged：节点数据变化

NodeChildrenChanged：子节点列表变化

DataWatchRemoved：节点监听被移除

ChildWatchRemoved：子节点监听被移除



### 5、应用场景

注册中心,数据发布/订阅（常用于实现配置中心）,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

使用zk的顺序节点可以用来生成分布式订单号

# zk集群

### 集群角色

客户端随机访问集群中的任意节点处理请求,如果是写请求,则由接到请求的非主节点将请求转发到主节点

Leader： 领导者。

事务请求（写操作）的唯一调度者和处理者，保证集群事务处理的顺序性；集群内部各

个服务器的调度者。对于create、setData、delete等有写操作的请求，则要统一转发给

leader处理，leader需要决定编号、执行操作，这个过程称为事务。

Follower: 跟随者

处理客户端非事务（读操作）请求（可以直接响应），转发事务请求给Leader；参与集群

Leader选举投票。

Observer: 观察者

对于非事务请求可以独立处理（读操作），对于事务性请求会转发给leader处理。

Observer节点接收来自leader的inform信息，更新自己的本地存储，不参与提交和选举投

票。通常在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

Observer应用场景：

提升集群的读性能。因为Observer和不参与提交和选举的投票过程，所以可以

通过往集群里面添加observer节点来提高整个集群的读性能。

跨数据中心部署。 比如需要部署一个北京和香港两地都可以使用的zookeeper

集群服务，并且要求北京和香港客户的读请求延迟都很低。解决方案就是把香港的节

点都设置为observer。

### 2、leader选举

Leader选举成功后要与所有的follower保持ping消息,如果超过一半的follower ping无返回ack则leader会改变状态为looking继续选举,follower会死循环的接收主节点的ping消息,如果接收不到ping消息会自动改变自身状态为looking,然后尝试重新选举

zookeeper 的 leader 选举存在两个阶段，一个是服务器启动时 leader 选举，另一个是运

行过程中 leader 服务器宕机。

在分析选举原理前，先介绍几个重要的参数：

服务器 ID(myid)：编号越大在选举算法中权重越大

事务 ID(zxid)：值越大说明数据越新，权重越大

逻辑时钟(epoch-logicalclock)：同一轮投票过程中的逻辑时钟值是相同的，每投完一次值会增加

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

服务器启动时的 leader 选举

每个节点启动的时候都 LOOKING 观望状态，接下来就开始进行选举主流程。这里选取三

台机器组成的集群为例。第一台服务器 server1启动时，无法进行 leader 选举，当第二台

服务器 server2 启动时，两台机器可以相互通信，进入 leader 选举过程。

（1）每台 server 发出一个投票，由于是初始情况，server1 和 server2 都将自己作为 leader 服务器进行投票，每次投票包含所推举的服务器myid、zxid、epoch，使用（myid，zxid）表示，此时 server1 投票为（1,0），server2 投票为（2,0），然后将各自投票发送给集群中其他机器。

（2）接收来自各个服务器的投票。集群中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（epoch）、是否来自 LOOKING 状态的服务器。

（3）分别处理投票。针对每一次投票，服务器都需要将其他服务器的投票和自

己的投票进行对比，对比规则如下：

a. 优先比较 epoch

b. 检查 zxid，zxid 比较大的服务器优先作为 leader

c. 如果 zxid 相同，那么就比较 myid，myid 较大的服务器作为 leader 服务器

（4）统计投票。每次投票后，服务器统计投票信息，判断是都有过半机器接收

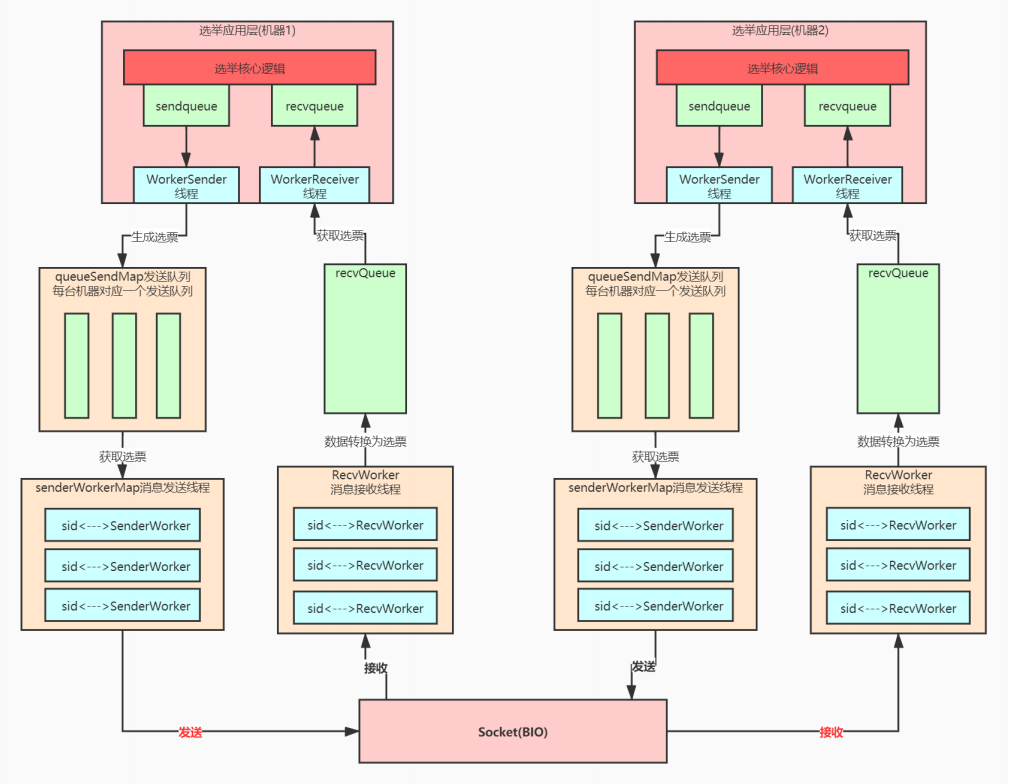
到相同的投票信息。server1、server2 都统计出集群中有两台机器接受了（2,0）的

投票信息，此时已经选出了 server2 为 leader 节点。

（5）改变服务器状态。一旦确定了 leader，每个服务器响应更新自己的状态，

如果是 follower，那么就变更为 FOLLOWING，如果是 Leader，变更为

LEADING。此时 server3继续启动，直接加入变更自己为 FOLLOWING。



### 3、zab协议

在 Zookeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性。

ZAB 协议分为两部分：

消息广播

崩溃恢复

**消息广播**

Zookeeper 使用单一的主进程 Leader 来接收和处理客户端所有事务请求，并采用 ZAB 协

议的原子广播协议，将事务请求以 Proposal 提议广播到所有 Follower 节点，当集群中有

过半的Follower 服务器进行正确的 ACK 反馈，那么Leader就会再次向所有的 Follower 服

务器发送commit 消息，将此次提案进行提交。这个过程可以简称为 2pc 事务提交，整个

流程可以参考下图，注意 Observer 节点只负责同步 Leader 数据，不参与 2PC 数据同步

过程。

**崩溃恢复**

在正常情况消息下广播能运行良好，但是一旦 Leader 服务器出现崩溃，或者由于网络原理

导致 Leader 服务器失去了与过半 Follower 的通信，那么就会进入崩溃恢复模式，需要选

举出一个新的 Leader 服务器。在这个过程中可能会出现两种数据不一致性的隐患，需要

ZAB 协议的特性进行避免。

Leader 服务器将消息 commit 发出后，立即崩溃

Leader 服务器刚提出 proposal 后，立即崩溃

ZAB 协议的恢复模式使用了以下策略：

选举 zxid 最大的节点作为新的 leader

新 leader 将事务日志中尚未提交的消息进行处理

**数据同步**

当崩溃恢复之后，需要在正式工作之前（接收客户端请求），Leader服务器首先确认事务是否都已经被过半的 Follwer 提交了，即是否完成了数据同步。目的是为了保持数据一致。

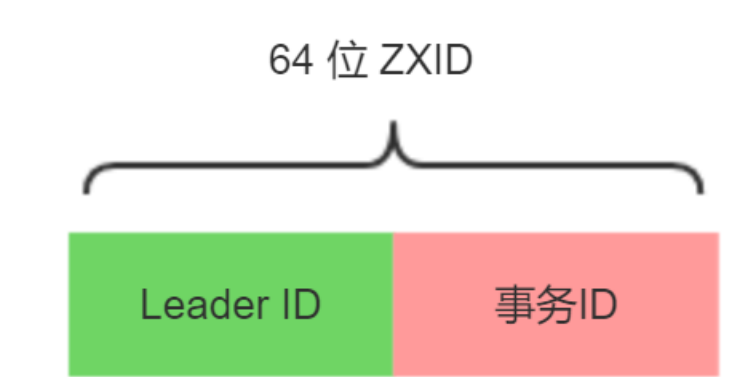
当 Follwer 服务器成功同步之后，Leader 会将这些服务器加入到可用服务器列表中。

实际上，Leader服务器处理或丢弃事务都是依赖着ZXID的，那么这个ZXID如何生成呢？

答：在 ZAB 协议的事务编号 ZXID 设计中，ZXID 是一个 64 位的数字，其中低 32 位可以看作是一个简单的递增的计数器，针对客户端的每一个事务请求，Leader都会产生一个新的事务 Proposal 并对该计数器进行+1 操作。而高32位则代表了Leader服务器上取出本地日志中最大事务 Proposal 的 ZXID，并从该 ZXID中解析出对应的 epoch值(leader选举周期)，当一轮新的选举结束后，会对这个值加一，并且事务id又从0开始自增。

高32位代表了每代 Leader的唯一性，低32代表了每代 Leader中事务的唯一性。同时，也能让Follwer通过高32位识别不同的Leader。简化了数据恢复流程。

基于这样的策略：当 Follower连接上Leader之后，Leader服务器会根据自己服务器上最后被提交的ZXID和Follower上的ZXID进行比对，比对结果要么回滚，要么和Leader同步。



### 4、zk客户端

#### ZooKeeper官方的Java客户端API。

第三方的Java客户端API，比如Curator。

**ZooKeeper官方API有一些不足之处，具体如下：**

1、ZooKeeper的Watcher监测是一次性的，每次触发之后都需要重新进行注册。∙会话超时之后没有实现重连机制。

2、异常处理烦琐，ZooKeeper提供了很多异常，对于开发人员来说可能根本不知道应该如何处理这些抛出的异常。

3、仅提供了简单的byte[]数组类型的接口，没有提供Java POJO级别的序列化数据处理接口。

4、创建节点时如果抛出异常，需要自行检查节点是否存在。

5、无法实现级联删除

#### Curator客户端

Curator是Netflix公司开源的一套ZooKeeper客户端框架，和ZkClient一样它解决了非常底

层的细节开发工作，包括连接、重连、反复注册Watcher的问题以及NodeExistsException

异常等。

Curator是Apache基金会的顶级项目之一，Curator具有更加完善的文档，另外还提供了一

套易用性和可读性更强的Fluent风格的客户端API框架。

Curator还为ZooKeeper客户端框架提供了一些比较普遍的、开箱即用的、分布式开发用的

解决方案，例如Recipe、共享锁服务、Master选举机制和分布式计算器等，帮助开发者避

免了“重复造轮子”的无效开发工作。

curator-framework是对ZooKeeper的底层API的一些封装。

curator-client提供了一些客户端的操作，例如重试策略等。

curator-recipes封装了一些高级特性，如：Cache事件监听、选举、分布式锁、分布式计数器、分布式Barrier等。

Curator连接客户端的重试机制



**Curator Caches:**

Curator 引入了 Cache 来实现对 Zookeeper 服务端事件监听，Cache 事件监听可以理解

为一个本地缓存视图与远程 Zookeeper 视图的对比过程。Cache 提供了反复注册的功能。

Cache 分为两类注册类型：节点监听和子节点监听。

NodeCache对节点进行监听 PathChildrenCache对子节点进行监听 TreeCache维护了treeNode,可以对当前节点下的所有节点进行监听

**异步接口：**

Curator 引入了BackgroundCallback 接口，用来处理服务器端返回来的信息，这个处理过

程是在异步线程中调用，默认在EventThread 中调用，也可以自定义线程池。

### 5、分布式锁实现

1、基于数据库的分布式锁。db操作性能较差，并且有锁表的风险，一般不考虑。

2、基于Redis的分布式锁。适用于并发量很大、性能要求很高而可靠性问题可以通过其

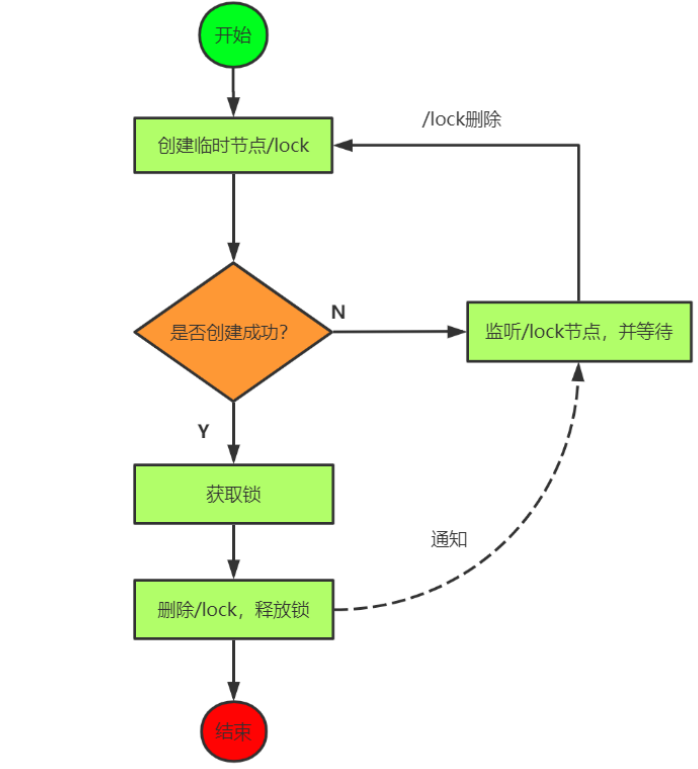
他方案去弥补的场景。

1. 基于ZooKeeper的分布式锁。适用于高可靠（高可用），而并发量不是太高的场景。

基于zookeeper实现分布式锁方案：公平锁实现与非公平锁实现

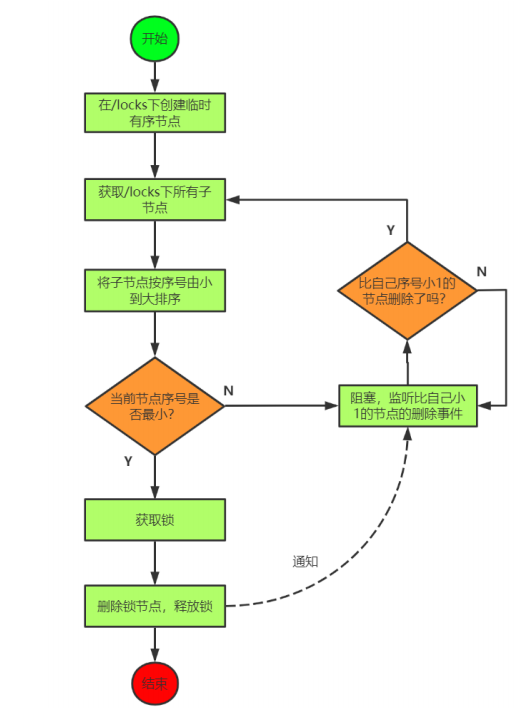
**公平锁实现：**

所有线程创建同一个路径的临时节点，当创建成功的线程直接获取到锁，其余线程则监听该节点，直到节点删除所有线程继续抢锁，也就存在羊群效应问题。

****

**公平锁实现**：

公平锁实现使用临时有序节点，所有线程均先创建有序节点，然后获取该路径下的所有有序节点，对比当前线程的节点序号是不是最小，如果最小则获取到锁，如果非最小则监听序号数减一的节点，做到按先后顺序获取锁，且不存在多线程抢锁。



Curator 由ZooKeeper分布式锁（如InterProcessMutex）的封装，具备高可用、可重入、阻塞锁特性，可解决失效死锁问题，使用起来也较为简单。

**缺点**：因为需要频繁的创建和删除节点，性能上不如Redis。在高性能、高并发的应用场景下，不建议使用ZooKeeper的分布式锁。而由于ZooKeeper 的高可用性，因此在并发量不是太高的应用场景中，还是推荐使用ZooKeeper的分布式锁。

