# Zookeeper简介

ZooKeeper 是一个开源的分布式协调服务。分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

# 重要概念

## 会话

Session 指的是 ZooKeeper  服务器与客户端建立连接产生的会话。客户端启动的时候，首先会与服务器建立一个 TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了。Session 的 sessionTimeout 值用来设置一个客户端会话的超时时间。在为客户端创建会话之前，服务端首先会为每个客户端都分配一个全局唯一的 sessionID。

## Znode

ZooKeeper 将所有数据存储在内存中，数据模型是一棵树（Znode Tree)，由斜杠（/）的进行分割的路径，就是一个 Znode。

在 Zookeeper 中，Node 可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个 ZNode 被创建了，除非主动进行 ZNode 的移除操作，否则这个 ZNode 将一直保存在 ZooKeeper 上。

而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。

## 版本

Zookeeper 的每个 ZNode 上都会存储数据，对应于每个 ZNode，Zookeeper 都会为其维护一个叫作 Stat 的数据结构。

Stat 中记录了这个 ZNode 的三个数据版本，分别是：

version（当前 ZNode 的版本）

cversion（当前 ZNode 子节点的版本）

aversion（当前 ZNode 的 ACL 版本）

## watcher

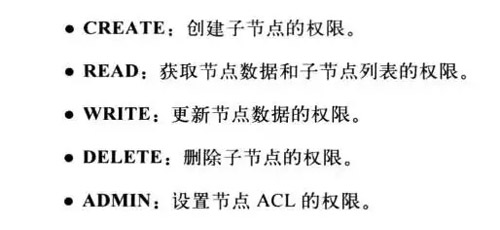
Watcher（事件监听器），是 ZooKeeper 中的一个很重要的特性。

ZooKeeper 允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，ZooKeeper 服务端会将事件通知到感兴趣的客户端上去，该机制是 ZooKeeper 实现分布式协调服务的重要特性。

## ACL

ZooKeeper 采用 ACL（AccessControlLists）策略来进行权限控制，类似于  UNIX 文件系统的权限控制。

ZooKeeper 定义了 5 种权限，如下图：



# Zookeeper特点

**顺序一致性**：从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。

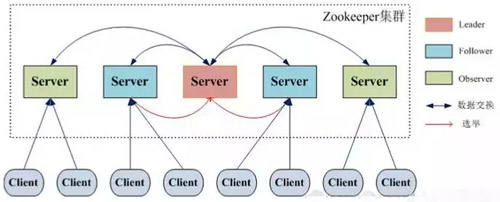
**原子性**：所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。

**单一系统映像**：无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。

**可靠性**：一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

# Zookeeper集群角色

Zookeeper集群模式并未采用master/slave，而是引入了leader/follower/observer三种角色。如下图所示：



ZooKeeper 集群中的所有机器通过一个 Leader 选举过程来选定一台称为 “Leader” 的机器。

Leader 既可以为客户端提供写服务又能提供读服务。除了 Leader 外，Follower 和  Observer 都只能提供读服务。

Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。



# ZAB协议

ZAB 协议全称：Zookeeper Atomic Broadcast（Zookeeper 原子广播协议）。是zookeeper数据一致性的核心算法。

## 主要实现

1.使用一个单一的主进程来接收并处理客户端的所有事务请求，并采用 ZAB 的原子广播协议，将服务器数据的状态变更以事务 Proposal 的形式广播到所有的副本进程上去。

2.保证一个全局的变更序列被顺序应用。

ZooKeeper是一个树形结构，很多操作都要先检查才能确定能不能执行，比如P1的事务t1可能是创建节点“/a”，t2可能是创建节点“/a/aa”，只有先创建了父节点“/a”，才能创建子节点“/a/aa”。为了保证这一点，ZAB要保证同一个leader的发起的事务要按顺序被apply，同时还要保证只有先前的leader的所有事务都被apply之后，新选的leader才能在发起事务。

3.当前主进程出现异常情况的时候，依旧能够正常工作。

## 实现原理

所有事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被称为 Leader服务器，而余下的其他服务器则成为 Follower 服务器。 Leader 服务器负责将一个客户端事务请求转换成一个事务proposal（提议），并将该 Proposal分发给集群中所有的Follower服务器。之后 Leader 服务器需要等待所有Follower 服务器的反馈,一旦超过半数的Follower服务器进行了正确的反馈后，那么 Leader 就会再次向所有的 Follower服务器分发Commit消息，要求其将前一个proposal进行提交。

## ZAB协议两种模式

ZAB 协议包括两种基本的模式，分别是崩溃恢复和消息广播。

当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时， ZAB 协议就会进入恢复模式并选举产生新的 Leader 服务器。当选举产生了新的Leader 服务器同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出恢复模式。

当集群中已经有过半的 Follower 服务器完成了和 Leader 服务器的状态同步，那么整个服务框架就可以进入消息广播模式了。当一台同样遵守 ZAB 协议的服务器启动后加入到集群中时，如果此时集群中已经存在一个 Leader 服务器在负责进行消息广播 ， 那么新加人的服务器就会自觉地进人数据恢复模式：找到 Leader 所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。

下面重点讲解崩溃回复和消息广播的过程。

### 消息广播模式

ZAB 协议的消息广播过程使用的是一个原子广播协议，类似一个 **二阶段提交过程**。对于客户端发送的写请求，全部由 Leader 接收。Leader 服务器会为每一个 Follower 服务器都各自分配一个单独的队列，然后将需要广播的事务 Proposal 依次放入这些队列中去，并且根据 FIFO策略进行消息发送。每一个 Follower 服务器在接收到这个事务 Proposal 之后，都会首先将其以事务日志的形式写入到本地磁盘中去，并且在成功写入后反馈给 Leader 服务器一个 Ack 响应。当 Leader 服务器接收到超过半数 Follower 的 Ack 响应后，就会广播一个Commit 消息给所有的 Follower 服务器以通知其进行事务提交，同时 Leader 自身也会完成对事务的提交。

### 崩溃恢复模式

Leader 服务器出现崩溃，或者说由于网络原因导致 Leader 服务器失去了与过半 Follower 的联系，那么就会进入崩溃恢复模式。Leader 选举算法不仅仅需要让 Leader自己知道其自身已经被选举为 Leader ,同时还需要让集群中的所有其他机器也能够快速地感知到选举产生的新的 Leader 服务器。

刚刚我们说消息广播过程中，Leader 崩溃怎么办？还能保证数据一致吗？如果 Leader 先本地提交了，然后 commit 请求没有发送出去，怎么办？

实际上，当 Leader 崩溃，即进入我们开头所说的崩溃恢复模式（崩溃即：Leader 失去与过半 Follwer 的联系）。下面来详细讲述。

假设1：Leader 在复制数据给所有 Follwer 之后崩溃，怎么办？  
假设2：Leader 在收到 Ack 并提交了自己，同时发送了部分 commit 出去之后崩溃怎么办？

针对这些问题，ZAB 定义了 2 个原则：

ZAB 协议确保那些已经在 Leader 提交的事务最终会被所有服务器提交。

ZAB 协议确保丢弃那些只在 Leader 提出/复制，但没有提交的事务。

所以，ZAB 设计了下面这样一个选举算法：

能够确保提交已经被 Leader 提交的事务，同时丢弃已经被跳过的事务。

针对这个要求，如果让 Leader 选举算法能够保证新选举出来的 Leader 服务器拥有集群总所有机器编号（即 ZXID 最大）的事务，那么就能够保证这个新选举出来的 Leader 一定具有所有已经提交的提案。而且这么做有一个好处是：可以省去 Leader 服务器检查事务的提交和丢弃工作的这一步操作。

# 分布式锁

首先我们先来看看使用zk实现分布式锁的原理，在zk中是使用文件目录的格式存放节点内容，其中节点类型分为：

- 持久节点（PERSISTENT ）：节点创建后，一直存在，直到主动删除了该节点。

- 临时节点（EPHEMERAL）：生命周期和客户端会话绑定，一旦客户端会话失效，这个节点就会自动删除。

- 序列节点（SEQUENTIAL ）：多个线程创建同一个顺序节点时候，每个线程会得到一个带有编号的节点，节点编号是递增不重复的。

这里我们就使用临时顺序节点来实现分布式锁。

分布式锁实现：

- 创建临时顺序节点,比如/root/node，假设返回结果为nodeId。

- 获取/root下所有孩子节点，用自己创建的nodeId的序号与所有子节点比较，看看自己是不是编号最小的。如果是最小的则就相当于获取到了锁，如果自己不是最小的，则从所有子节点里面获取比自己次小的一个节点，然后设置监听该节点的事件，然后挂起当前线程。

- 当最小编号的线程获取锁，处理完业务后删除自己对应的nodeId，删除后会激活比自己大一号的节点的线程从阻塞变为运行态，被激活的线程应该就是当前node序列号最小的了，然后就会获取到锁。

# 常用命令

一、服务端

1、启动zk服务

sh bin/zkServer.sh start

2、查看zk服务状态

sh bin/zkServer.sh status

3、停止zk服务

sh bin/zkServer.sh stop

4、重启zk服务

sh bin/zkServer.sh restart

二、客户端

1、连接zk服务

./zkCli.sh -server 127.0.0.1:2181

三、客户端连接zk服务后

1、查看根节点（ZNode）下的所有节点

[zk: 127.0.0.1:2181(CONNECTED) 1] ls /

2、查看某个目录包含的所有节点及time、version等信息

[zk: 127.0.0.1:2181(CONNECTED) 1] ls2 /

3、创建节点，并设置初始内容

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

其中，-s或-e分别指定节点为顺序或临时节点，默认情况下创建的为持久节点。acl命令是用来做权限控制的，如

[zk: 127.0.0.1:2181(CONNECTED) 1] create /test "hello"

注意：zk不能递归创建节点，也就是如果父节点不存在，无法创建子节点

例如根节点下只有zookeeper,apps,iot,test

[zk: localhost:2181(CONNECTED) 8] ls /

[zookeeper, test, iot, apps]

如果执行命令create /aaaa/bbbbb "hello"，则会提示如下信息

Node does not exist: /aaaa/bbbbb

4、获取指定节点的信息

[zk: localhost:2181(CONNECTED) 2] get /test

显示内容如下：

"hello"

cZxid = 0xe53

ctime = Thu Mar 29 10:36:40 CST 2018

mZxid = 0xe53

mtime = Thu Mar 29 10:36:40 CST 2018

pZxid = 0xe53

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 7

numChildren = 0

5、修改指定节点关联的数据

set path data [version]

其中data为更新内容，version是指定当然更新是基于Znode的哪个版本进行更新的。

[zk: localhost:2181(CONNECTED) 9] set /test "test"

显示内容如下：

cZxid = 0xe53

ctime = Thu Mar 29 10:36:40 CST 2018

mZxid = 0xe57

mtime = Thu Mar 29 10:43:42 CST 2018

pZxid = 0xe53

cversion = 0

dataVersion = 1

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 6

numChildren = 0

6、删除节点

delete path [version]

version是指定当然更新是基于Znode的哪个版本进行删除的。

[zk: localhost:2181(CONNECTED) 0] delete /test

[zk: localhost:2181(CONNECTED) 1] ls /

[zookeeper, iot, apps]

[zk: localhost:2181(CONNECTED) 2]

# Java API



# 监听机制

**1、zookeeper的Watch机制**

一个zk的节点可以被监控，包括这个目录中存储的数据的修改，子节点目录的变化，一旦变化可以通知设置监控的客户端，这个功能是zookeeper对于应用最重要的特性，通过这个特性可以实现的功能包括配置的集中管理，集群管理，分布式锁等等。

**watch机制官方说明：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。**

**2、zookeeper机制的特点**

**1) 一次性的触发器（one-time trigger）**

当数据改变的时候，那么一个Watch事件会产生并且被发送到客户端中。但是客户端只会收到一次这样的通知，如果以后这个数据再次发生改变的时候，之前设置Watch的客户端将不会再次收到改变的通知，因为Watch机制规定了它是一个一次性的触发器。

当设置监视的数据发生改变时，该监视事件会被发送到客户端，例如，如果客户端调用了 getData("/znode1", true) 并且稍后 /znode1 节点上的数据发生了改变或者被删除了，客户端将会获取到 /znode1 发生变化的监视事件，而如果 /znode1 再一次发生了变化，除非客户端再次对 /znode1 设置监视，否则客户端不会收到事件通知。

**2)发送给客户端（Sent to the client）**

这个表明了Watch的通知事件是从服务器发送给客户端的，是异步的，这就表明不同的客户端收到的Watch的时间可能不同，但是ZooKeeper有保证：当一个客户端在看到Watch事件之前是不会看到结点数据的变化的。例如：A=3，此时在上面设置了一次Watch，如果A突然变成4了，那么客户端会先收到Watch事件的通知，然后才会看到A=4。

Zookeeper 客户端和服务端是通过 Socket 进行通信的，由于网络存在故障，所以监视事件很有可能不会成功地到达客户端，监视事件是异步发送至监视者的，Zookeeper 本身提供了保序性(ordering guarantee)：即客户端只有首先看到了监视事件后，才会感知到它所设置监视的 znode 发生了变化(a client will never see a change for which it has set a watch until it first sees the watch event). 网络延迟或者其他因素可能导致不同的客户端在不同的时刻感知某一监视事件，但是不同的客户端所看到的一切具有一致的顺序。

**3)被设置Watch的数据（The data for which the watch was set）**

这意味着 znode 节点本身具有不同的改变方式。你也可以想象 Zookeeper 维护了两条监视链表：

数据监视和子节点监视(data watches and child watches)

getData() and exists() 设置数据监视，getChildren() 设置子节点监视。 或者，你也可以想象 Zookeeper 设置的不同监视返回不同的数据，getData() 和 exists() 返回 znode 节点的相关信息，而 getChildren() 返回子节点列表。

因此， setData() 会触发设置在某一节点上所设置的数据监视(假定数据设置成功)，而一次成功的 create() 操作则会出发当前节点上所设置的数据监视以及父节点的子节点监视。一次成功的 delete() 操作将会触发当前节点的数据监视和子节点监视事件，同时也会触发该节点父节点的child watch。

**3、各种watch触发的情况总结**

可以注册watcher的方法：getData、exists、getChildren。

可以触发watcher的方法：create、delete、setData。连接断开的情况下触发的watcher会丢失。

一个Watcher实例是一个回调函数，被回调一次后就被移除了。如果还需要关注数据的变化，需要再次注册watcher。

**New ZooKeeper时注册的watcher叫default watcher，它不是一次性的，只对client的连接状态变化作出反应。**

**表格 具体操作触发的具体事件类型**

|  |  |  |
| --- | --- | --- |
|  | event For "/path" | event For "/path/child" |
| create("/path") | EventType.NodeCreated | 无 |
| delete("/path") | EventType.NodeDeleted | 无 |
| setData("/path") | EventType.NodeDataChanged | 无 |
| create("/path/child") | EventType.NodeChildrenChanged（getChild） | EventType.NodeCreated |
| delete("/path/child") | EventType.NodeChildrenChanged（getChild） | EventType.NodeDeleted |
| setData("/path/child") | 无 | EventType.NodeDataChanged |

表格说明：第一列为具体操作，第二列和第三列为具体节点或者子节点触发的类型，"无"表示不会触发。

**表格 设置监听方法对应后续会触发的事件类型的关系**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| event For "/path" | Default Watcher | exists("/path") | getData("/path") | getChildren("/path") |
| EventType.None | √ | √ | √ | √ |
| EventType.NodeCreated |  | √ | √ |  |
| EventType.NodeDeleted |  | √ | √ |  |
| EventType.NodeDataChanged |  | √ | √ |  |
| EventType.NodeChildrenChanged |  | √ | √ | √ |

表格说明：第一列为会触发的事件类型，第一行为设置监听的方法，即调用了那个设置监听的方法，后续触发对应的事件类型。

**表格 设置监听的方法与对应后续调用会触发监听的方法的关系**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | exits("/path") | getData("/path") | getChildren("/path") | exits("/path/child") | getData("/path/child") | getChildren("/path/child") |
| create("/path") | √ | √ | 会报错 |  |  |  |
| delete("/path") | √ | √ | √（这个要注意） |  |  |  |
| setData("/path") | √ | √ |  |  |  |  |
| create("/path/child") | √ | √ | √ |  |  |  |
| delete("/path/child") |  |  | √ | √ | √ | √ |
| setData("/path/child") |  |  |  | √ | √ |  |
|  |  |  |  |  |  |  |

表格说明：第一列为触发监听的方法，第一行为设置监听方法，这张表展示了设置监听方法与后续调用会触发该监听的方法的对应关系

# ZK与CAP

CAP理论告诉我们，一个分布式系统不可能同时满足以下三种

一致性（C:Consistency）

可用性（A:Available）

分区容错性（P:Partition Tolerance）

这三个基本需求，最多只能同时满足其中的两项，**因为P是必须的,因此往往选择就在CP或者AP中**。

**在此ZooKeeper保证的是CP**

**分析：可用性（A:Available）**

**不能保证每次服务请求的可用性**。任何时刻对ZooKeeper的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性；但是它不能保证每次服务请求的可用性（注：也就是在极端环境下，ZooKeeper可能会丢弃一些请求，消费者程序需要重新请求才能获得结果）。所以说，ZooKeeper不能保证服务可用性。

**进行leader选举时集群都是不可用**。在使用ZooKeeper获取服务列表时，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。所以说，ZooKeeper不能保证服务可用性。

作者：BakerZhu  
链接：https://www.jianshu.com/p/4445842be566  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

# 总结

 getChildren()方法仅仅监控对应节点直接子目录的一次变化，但是只会监控直接子节点的增减情况，不会监控数据变化情况！若要每次对应节点发生增减变化都被监测到，那么每次都得先调用getChildren()方法获取一遍节点的子节点列表

exists()方法仅仅监控对应节点的一次数据变化，无论是数据修改还是删除！若要每次对应节点发生变化都被监测到，那么每次都得先调用exists()方法获取一遍节点状态！

getData()方法仅仅监控对应节点的一次数据变化，无论是数据修改还是删除！若要每次对应节点发生变化都被监测到，那么每次都得先调用getData()方法获取一遍数据！