**Etcd**

# 1.etcd与zookeeper

## 1.1.CAP原则

Zookeeper和etcd都是强一致性，当发生网络分区时，选择了CP，两者的一致性协议raft和ZAB都是paxos的变体，都是通过多数派达成决议。

## 1.2.逻辑结构

Zookeeper从逻辑上看是一种目录结构，而etcd从逻辑上看是K-V结构，因为etcd的key可以是任意字符串，所以仍然可以模拟目录，同样，etcd可以表达父子关系，etcd在存储上实现了key 的有序排序，因此/a/b，/a/b/c，/a/b/d在存储中是顺序排列的，所以可以通过定位到key=/a/b并依次顺序扫描，找到/a/b/c，/a/b/d孩子目录，从而实现父子目录关系。

其本质上是一个有序的K-V存储。

## 1.3.临时节点

在实现服务发现时，一般会用到zookeeper的临时节点，只要client与zookeeper之间session会话没有中断或过期，临时节点就一直存在，client掉线后session就会过期，对应的临时节点就会被自动删除。

Etcd中没有临时节点的概念，但是支持lease租约机制，即etcd支持申请定时器，比如：可以申请一个TTL=10s的lease，会返回一个lease ID标识定时器，可以set一个key同时携带lease ID，即可实现一个自动过期的key，在etcd中，一个lease可以关联任意多的key，当lease过期后所有关联的key将被自动删除。

## 1.4.事件模型

用zookeeper实现服务发现，一般会getChildrenAndWatch来获取一个目录下所有在线节点，这个API会先获取当前孩子列表并同时注册一个观察器，，每当zookeeper发现孩子有变动，就会发送一个通知事件给客户端，同时关闭观察器，此时再次调用getChildrenWatch再次获取最新的孩子列表并重新注册观察器。

Zookeeper的事件模型非常可靠，不会出现发生了更新而客户端不知道的情况，特点如下：

* 事件不包含数据，仅仅通知变化
* 多次连续的更新，通知会合并为一个，客户端收到通知再次拉取数据，会跳过中间多个版本，只拿到最新的数据

Etcd事件是包含数据的，通常情况下连续的更新不会被合并通知，而是逐条通知到客户端。

# 2.etcd原理分析

Etcd基于raft实现分布式数据一致，其本质是一个K-V存储，在内存中维护一个B树，它是有序的，结构如下：

Type treeIndex struct{

Sync.RWMutex

Tree \*btree.BTree

}

当存储大量的K-V时，如果用的value较大，全部放在内存B树中内存耗费过大，所以etcd将用户的value保存在磁盘中。Etcd是纯内存索引，数据在磁盘持久化，磁盘上使用bbolt的纯K-V存储引擎。

## 2.1.MVCC

Etcd每次数据变化都会通知，并且通知里携带有变化后数据的内容，etcd通过MVCC提供历史数据变更记录。

每一个事务有唯一ID，全局递增不重复，称为main ID，一个事务可以包含多个修改操作，每一个操作称为一个revision，从0递增编号，称为sub ID。每个revision由（main ID， sub ID）唯一标识。

在内存索引中，每个用户原始key都有一个key\_index结构，里面维护了多版本信息：

Type struct keyIndex

{

Byte Key[]; //原始key值

Revision Modified; //最后一次对应的revision

Generations []generation; //多版本历史修改

}

Type struct generation

{

int64 Ver； //当前代有多少版本

revision Created； //创建本代的revision

revision Revs[]; //(main ID, sub ID)

}

Generations[i]为第i代，标记一代的结束是key被删除（tombstone），当一个key从无到右，generations[0]会被创建，其中created字段记录了引起本次key创建的revision信息，当用户继续更新这个key的时候，generations[0].revs数组会不断追加本次revision信息。

在bbolt中，每个revision都将作为key，即序列化（revision.main+revision.sub）作为key，通过内存btree在keyIndex.generations[0].revs中找到最后一条revision，即可去bbolt中读取对应数据。

内存btree维护的用户key🡪keyIndex的映射，keyIndex内维护多版本的revision信息，revision可以映射到磁盘bbolt中的value，而在bbolt中value的存储结构是json。

## 2.2.watch机制

Etcd的时间通知机制是基于MVCC实现的，client可以提供一个要监听的revision.main作为watch的起始ID，只要etcd当前的全局自增事务ID > watch起始ID，etcd就会将MVCC在bbolt中存储的所有历史revision数据，逐一顺序推送给client。

Zookeeper总是获取最新数据并建立一个一次性的监听后续的变化。而etcd支持client从任意历史版本开始订阅时间，并且会推送当前的数据快照给client。

其工作原理如下：

Etcd会保存每个client发来的watch请求，watch请求可以关注一个key，或者一个key前缀（监听区间），etcd会有一个协程不断遍历所有watch请求，每个watch对象都维护了其watch的key事件推送到了哪个revision。

Etcd会拿着这个revision.main去bbolt中顺序遍历，遍历的时候会反序列化value，判断其中key值是否为watch请求关注的key，如果是则返回给client。每次会从所有watcher中选出一批watcher组成group进行处理，称为watchGroup，这批watcher中观察的最小的revision.main作为bbolt的遍历起始位置。

Type Struct Watcher

{

pb.watchClient Remote；

sync.RWMutex mu；

map[string] \*watchGrpcStream stream;

}

watchGroup分为两种

* 一种是synced，表示group的watcher都已经同步完毕，等待更新
* 一种是unsynced，表示group的watcher数据同步落后于当前最新的变更，正在追赶

当etcd收到client的watch请求，如果请求中携带revision参数，则比较revision和当前内存btree中的revision：

* Request\_revision >= cur\_revision，放入synced组
* Request\_revision < cur\_revision，放入unsynced组,etcd会启动一个后台协程持续同步unsynced的watcher，然后将其迁移到synced

在这种机制下，etcd实现了任意版本的watch，没有v2版本1000历史event的限制。

## 2.3.v2与v3

### 2.3.1.v2

Etcd v2是纯内存实现，并未将数据实时写入磁盘，持久化机制很简单，就是将store整合序列化成json写入文件，数据在内存中是一个简单的树结构。

Store中有一个全局currentIndex，每次变更，index++，然后每个event都会关联到一个currentIndex，当client调用watch（参数中增加wait=true）：

* 如果请求中有waitIndex，并且waitIndex小于currentIndex，则从eventHistory表中查询index<=waitIndex，并且和watch key匹配的event，返回数据。
* 如果请求中没有waitIndex或是历史表中没有waitIndex，则放入watchHub中，每个key会关联一个watcher列表，当有操作变更时，变更生成的event会放入eventHistory中，同时通知和该key相关的watcher。

注意：eventHistory长度为1000，即如果client停了许久，然后重新watch时候，可能和该waitIndex相关的event被淘汰了，这种情况变更丢失。

### 2.3.2.v3

Etcd v3将store分为两部分，一部分是内存中的索引kvindex（btree），一部分是boltdb（支持事务）KV存储，key为revision，value为key-value组合，以及创建版本和lease。

# 3.etcd应用场景

1. 提供存储以及获取数据的接口，通过raft协议保证etcd集群中多个节点数据的强一致性，用于存储元信息以及共享配置。
2. 提供监听机制，client可以监听某个key或某些key的变更，用于监听和推送变更
3. 提供key的过期和续约机制，client通过定时刷新来实现续约，用于集群监控以及服务注册发现
4. 提供原子的CAS和CAD（compare and delete）（v2通过接口参数实现，v3通过批量事务实现），用于分布式锁和leader选举。

# 4.zookeeper原理分析

## 4.1.zookeeper数据结构

Zookeeper的核心保存结构是一个dataTree数据结构，其内部是一个Map<String,DataNode>，其中key是path，DataNode是保存数据的核心数据结构。

Struct DataTree

{

concurrentHashMap<String, DataNode> Nodes; //代表所有节点

//临时节点，key为sessionID

ConcurrentHashMap<Long, HashSet<String>> ephemerals;

}

Struct DataNode

{

Byte[] data； //保存节点内容

Long acl； //保存data的acl规则

Set<String> children； //该节点下子节点的path

}

## 4.2.快照数据

异步线程生成快照，称为fuzzy快照，在快照文件生成过程中，仍然有新事务提交，快照文件不是精确到某一时刻的，而是模糊的。

生成时机：

每次事务日志写满，并切换一次事务日志文件，会生成快照文件。

## 4.3.初始化

Zookeeper服务器启动时，首先会进行数据初始化，将磁盘中数据加载到内存，恢复现场。恢复过程如下：

获取最新的100个快照，从最近一次快照开始解析快照文件校验checksum，直至恢复成功，返回最新的ZXID，根据ZXID从事务日志开始恢复数据。然后开始与follower节点的同步。