ZooKeeper 拥有一个层次的命名空间。（like distributed）

注意：ZooKeeper 中不许使用相对路径。

一 ZooKeeper 数据模型

1） Znode

ZooKeeper 目录树中的每一个节点对应着一个 Znode 每个 Znode 维护者一个属性结构。

dataVersion 版本号
ctime 创建时间
mtime 修改时间

每当 Znode 的数据改变时，它相应的版本号会增加。 每当客户端检索数据时，他同时检索版本号。 并且如果一个客户端如果进行了节点的更新或删除操作。他也必须提供要被操作的数据版本号。 如果所提供的数据版本号与实际的不匹配，那么这个操作将会失败。

Znode 是客户端要访问的 ZooKeeper 实体，包含以下几个特征。

Watches 客户端可以在节点上设置 watch (监视器)。当节点状态发生改变时，将会处发 watch 对应的操作。 当 watch 被触发时， Zookeeper 将会向 客户端发送且仅发送一个通知， 因为 watch 只能被触发一次。

数据访问 Zookeeper 中的每个节点上存储的数据需要被原子性的操作。读- 所有。 写- 替换所有。 节点中有(ACL)访问控制列表，列表中规定了用户的权限。（特定用户对应特定权限）

临时节点 ZK 在创建候确定类型且不能被改变（临时节点，永久节点）。临时节点依赖于会话。 一旦会话结束，临时节点删除。 ZK 临时节点不允许拥有子节点。相反，永久节点的生命周期不依赖于会话。只有在客户端执行删除操作时才会被删除。

顺序节点 （唯一性保证）创建 Znode 的时候，用户可以请求在 ZK的路径结尾添加一个递增计数。这个计数对于此节点的父节点来说是唯一的。

2） ZooKeeper 中的时间

Zxid

使 ZooKeeper 节点状态改变的每一个操作都将使节点接收到一个 zxid 格式的时间戳，并且这个时间戳是全局有序的。 也就是说，每一个对节点的改变都将产生一个唯一的 zxid。如果 zxid1 的值小于 zxid2 的值， 那么zxid1 所对应的事件发生在 2 所对应的事件 之前。 实际上， ZooKeeper 的每个节点维护者 3 个 zxid 值，分别为：cZxid、mZxid 和 pZxid。 cZxid 是节点的创建时间锁对应的 Zxid 格式时间戳， mZxid 是节点的修改时间所对应的 Zxid 格式时间戳。

版本号

对节点的每一个操作都将致使这个节点的版本号增加。 每个节点维护着三个版本号，它们 version（节点数据版本号）、cversion（子节点版本号）、avevsion（节点所拥有的 ACL 的版本号）。

3） ZooKeeper 节点属性结构

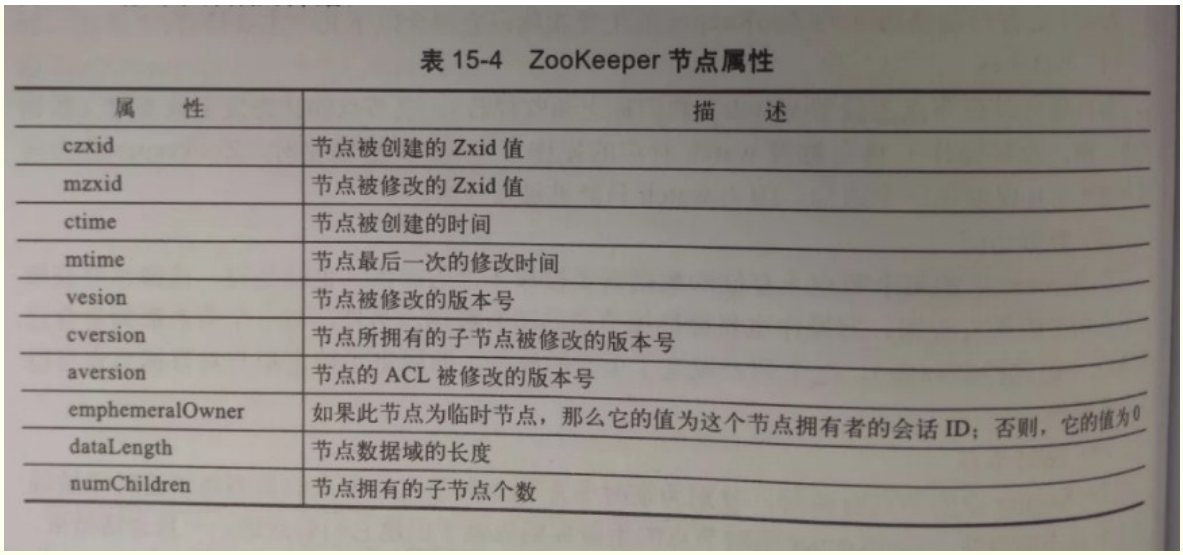


表 15-4 ZooKeeper 节点属性

属 性	描 述
czxid	节点被创建的 Zxid 值
mzxid	节点被修改的 Zxid 值
ctime	节点被创建的时间
mtime	节点最后一次的修改时间
vesion	节点被修改的版本号
cversion	节点所拥有的子节点被修改的版本号
aversion	节点的 ACL 被修改的版本号
emphemeralOwner	如果此节点为临时节点，那么它的值为这个节点拥有者的会话 ID；否则，它的值为 0
dataLength	节点数据域的长度
numChildren	节点拥有的子节点个数

二 ZooKeeper会话及状态

1） 服务器与客户端之间会话

在ZooKeeper中，客户端和服务端建立连接后，会话随之建立，生成一个全局唯一的会话ID(Session ID)。服务器和客户端之间维持的是一个长连接，在SESSION_TIMEOUT时间内，服务器会确定客户端是否正常连接(客户端会定时向服务器发送heart_beat，服务器重置下次SESSION_TIMEOUT时间)。因此，在正常情况下，Session一直有效，并且ZK集群所有机器上都保存这个Session信息。在出现网络或其它问题情况下（例如客户端所连接的那台ZK机器挂了，或是其它原因的网络闪断），客户端与当前连接的那台服务器之间连接断了，这个时候客户端会主动在地址列表寻址（实例化ZK对象的时候传入构造方法的那个参数connectString）中选择新的地址进行连接。

2） 连接断开

在这个过程中，两类异常CONNECTIONLOSS(连接断开)和SESSIONEXPIRED(Session过期)。

连接断开(CONNECTIONLOSS)一般发生在网络的闪断或是客户端所连接的服务器挂机的时候。ZooKeeper客户端自己会首先感知到这个异常,具体逻辑是在如下方法中触发的：一种场景是Server服务器挂了，这个时候，ZK客户端首选会捕获异常。核心流程如下: ZK客户端捕获“连接断开”异常 ——> 获取一个新的ZK地址 ——> 尝试连接 在这个流程中，我们可以发现，整个过程不需要开发者额外的程序介入，都是ZK客户端自己会进行的，并且，使用的会话ID都是同一个，所以结论就是：发生CONNECTIONLOSS的情况，应用不需要做什么事情，等待ZK客户端建立新的连接即可。

3） 会话超时

SESSIONEXPIRED发生在上面蓝色文字部分，这个通常是ZK客户端与服务器的连接断了，试图连接上新的ZK机器，但是这个过程如果耗时过长，超过了SESSION_TIMEOUT 后还没有成功连接上服务器，那么服务器认为这个Session已经结束了（服务器无法确认是因为其它异常原因还是客户端主动结束会话），由于在ZK中，很多数据和状态都是和会话绑定的，一旦会话失效，那么ZK就开始清除和这个会话有关的信息，包括这个会话创建的临时节点和注册的所有Watcher。在这之后，由于网络恢复后，客户端可能会重新连接上服务器，但是很不幸，服务器会告诉客户端一个异常：SESSIONEXPIRED（会话过期）。此时客户端的状态变成 CLOSED状态，应用要做的事情就是的看自己应用的复杂程序了，要重新实例zookeeper对象，然后重新操作所有临时数据（包括临时节点和注册Watcher），总之，会话超时在ZK使用过程中是真实存在的。

所以这里也简单总结下，一旦发生会话超时，那么存储在ZK上的所有临时数据与注册的订阅者都会被移除，此时需要重新创建一个ZooKeeper客户端实例，需要自己编码做一些额外的处理。

4） 会话时间（Session Time）会话超时时间控制先确认 ZooKeeper 客户端设置大小

在ZooKeeper API 实例化一个ZK客户端的时候，需要设置一个会话的超时时间。这里需要注意的一点是，客户端并不是可以随意设置这个会话超时时间，在ZK服务器端对会话超时时间是有限制的，主要是minSessionTimeout和maxSessionTimeout这两个参数设置的。（详细查看这个文章《ZooKeeper管理员指南》）Session超时时间限制，如果客户端设置的超时时间不在这个范围，那么会被强制设置为最大或最小时间。 默认的Session超时时间是在2 * tickTime ~ 20 * tickTime。所以，如果应用对于这个会话超时时间有特殊的需求的话，一定要和ZK管理员沟通好，确认好服务端是否设置了对会话时间的限制。

三 ZooKeeper watches（观察者）

1） Zookeeper 可以为所有的读操作设置一个watch，包括getData()、getChildren()和exists()。都有一个开关可以在操作的同时再设置一个watch。（实际上是为了让所有细节都通知到ZooKeeper）在ZooKeeper中，Watch是一个一次性触发器，会在被设置watch的数据发生变化的时候，发送给设置watch的客户端。

watch的定义中有三个关键点：

一次性触发器

一个watch事件将会在数据发生变更时发送给客户端。例如，如果客户端执行操作getData（“/znode1”, true），而后/znode1 发生变更或是删除了，客户端都会得到一个/znode1 的watch事件。如果/znode1 再次发生变更，则在客户端没有设置新的watch的情况下，是不会再给这个客户端发送watch事件的。

发送客户端

一个事件会发送给客户端，但可能在操作成功的返回值到达发起变动的客户端之前，这个事件还没有送达watch的客户端。Watch是异步发送的。但ZooKeeper保证了一个顺序：一个客户端在收到watch事件之前，一定不会看到它设置过watch的值的变动。网络时延和其他因素可能会导致不同的客户端看到watch和更新返回值的时间不同。但关键点是，每个客户端所看到的每件事都是有顺序的。（ZooKeeper 为watch 提供了一个有序的一致性）

设置 watch 数据

可以认为ZooKeeper维护了两个watch列表：数据 watch和 子 watch。getData()和exists() 设置 data watch，而 getChildren()设置child watch。或者，可以认为watch是根据返回值设置的。getData()和exists()返回节点本身的信息，而getChildren()返回子节点的列表。一个成功的 setData() 触发znode上设置的data watch。一个 create() 操作会触发被创建的znode上的数据watch，以及其父节点上的child watch。而一个成功的?delete()操作将会同时触发一个znode的data watch和child watch（因为这样就没有子节点了），同时也会触发其父节点的child watch。（需验证）

Watch由client连接上的ZooKeeper服务器在本地维护。减小设置、维护和分发watch的开销。当一个客户端连接到一个新的服务器上时，任何事件都可能会触发 watch 。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。（通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失）。

ZooKeeper对Watch提供了什么保障

Watch与其他事件、其他watch以及异步回复都是有序的。

ZooKeeper客户端库保证所有事件都会按顺序分发。

客户端会保障它在看到相应的znode的新数据之前接收到watch事件。

从ZooKeeper接收到的watch事件顺序一定和ZooKeeper服务所看到的事件顺序是一致的。

Watch是一次性触发器，如果你得到了一个watch事件，而你希望在以后发生变更时继续得到通知，你应该再设置一个watch。

因为watch是一次性触发器，而获得事件再发送一个新的设置watch的请求这一过程会有延时，所以你无法确保你看到了所有发生在ZooKeeper上的一个节点上的事件。所以请处理好在这个时间窗口中可能会发生多次znode变更的这种情况。（你可以不处理，但至少请认识到这一点）。

四 ZooKeeper ACL（父子节点权限独立。）

1） 在Zookeeper中，node的ACL是没有继承关系的，是独立控制的。 zk的节点可以扩展。

表 15-5 ACL 权限

权 限	权限描述
CREATE（创建）	创建子节点
READ（读）	从节点获取数据或列出节点的所有子节点
WRITE（写）	设置节点的数据
DELETE（删除）	删除子节点
ADMIN（管理员）	可以设置权限

五 ZooKeeper 一致性

Zookeeper的实现是有Client、Server构成，Server端提供了一个一致性复制、存储服务，Client端会提供一些具体的语义，比如分布式锁、选举算法、分布式互斥等。从存储内容来说，Server端更多的是存储一些数据的状态，而非数据内容本身，因此Zookeeper可以作为一个小文件系统使用。 数据状态的存储量相对不大，完全可以全部加载到内存中，从而极大地消除了通信延迟。

Server可以崩溃后（Crash）重启。考虑到容错性，Server必须“记住”之前的数据状态，因此数据需要持久化，但吞吐量很高时，磁盘的IO便成为系统瓶颈，其解决办法是使用缓存，把随机写变为连续写。

- 全序（Total order）：如果消息a在消息b之前发送，则所有Server应该看到相同的结果
- 因果顺序（Causal order）：如果消息a在消息b之前发生（a导致了b），并被一起发送，则a始终在b之前被执行。（也就是说永远都是顺序加载。）

为了保证上述两个安全属性，Zookeeper使用了TCP协议和Leader。通过使用TCP协议保证了消息的全序特性（先发先到），通过Leader解决了因果顺序问题：先到Leader的先执行。因为有了Leader，Zookeeper的架构就变为：Master-Slave模式，但在该模式中Master（Leader）会Crash，因此，Zookeeper引入了Leader选举算法，以保证系统的健壮性。归纳起来Zookeeper整个工作分两个阶段：

1. Atomic Broadcast (原子性保证)

同一时刻存在一个Leader节点，其他节点称为“Follower”，

写：

Leader->执行-***watches***

客户端->更新->Leader->执行-***watches***

客户端->更新-> Follower->Leader->执行-***watches***

读(无特殊要求)：

Client->Follower

Client->Leader

读(最新)：

Client->leader

Zookeeper设计的读写比例是2：1。

- 因为只有一个Leader，Leader提交到Follower的请求一定会被接受（没有其他Leader干扰）
- 不需要所有的Follower都响应成功，只要一个多数派即可
- 如果这个时候有个渣渣Leader复活了，那么这个Leader的属性里面，会有一个

通俗地说，如果有2f+1个节点，允许f个节点失败。

A Leader B 死了 - C 正常。 不用选举。 A死 选举失败

六 ZooKeeper Leader选举

Leader Election

Leader选举主要是依赖Paxos算法，这里仅考虑Leader选举带来的一些问题。Leader选举遇到的最大问题是，”新老交互“的问题，新Leader是否要继续老Leader的状态。这里要按老Leader Crash的时机点分几种情况：

1. 老Leader在COMMIT前Crash（已经提交到本地）
2. 老Leader在COMMIT后Crash，但有部分Follower接收到了Commit请求

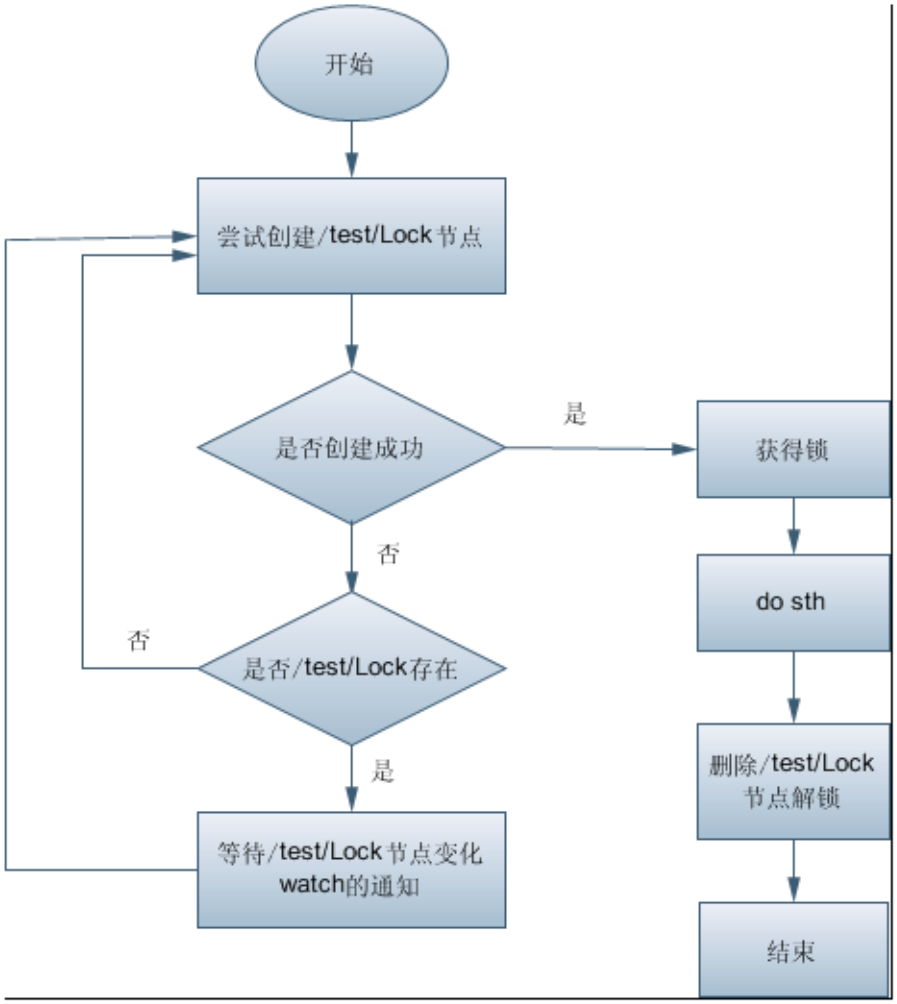
第一种情况，这些数据只有老Leader自己知道，当老Leader重启后，需要与新Leader同步并把这些数据从本地删除，以维持状态一致。

第二种情况，新Leader应该能通过一个多数派获得老Leader提交的最新数据

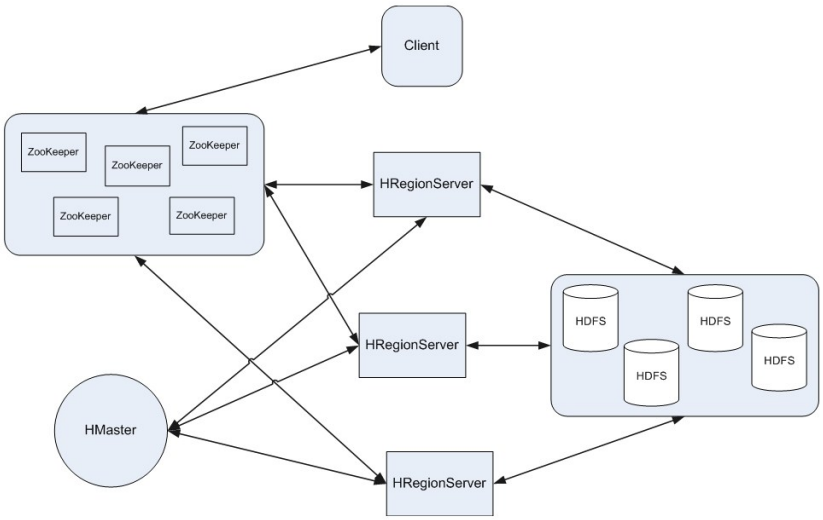
老Leader重启后，可能还会认为自己是Leader，可能会继续发送未完成的请求，从而因为两个Leader同时存在导致算法过程失败，解决办法是把Leader信息加入每条消息的id中，Zookeeper中称为zxid，zxid为一64位数字，高32位为leader信息又称为epoch，每次leader转换时递增；低32位为消息编号，Leader转换时应该从0重新开始编号。通过zxid，Follower能很容易发现请求是否来自老Leader，从而拒绝老Leader的请求。

因为在老Leader中存在着数据删除（情况1），因此Zookeeper的数据存储要支持补偿操作，这也就需要像数据库一样记录log。

七 ZooKeeper 锁 同其他锁一样。



下面是一些 zookeeper 在 hadoop 生态圈的一个形态



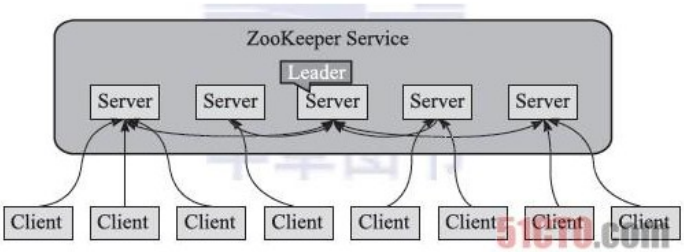
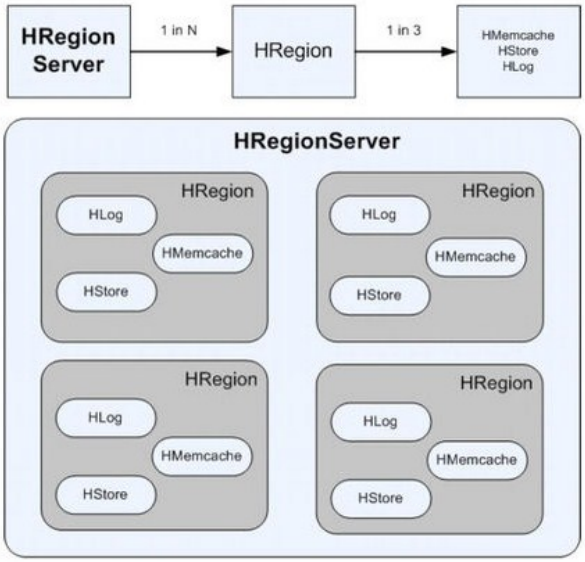


图 1-7 ZooKeeper 的结构示意



1.经过Map、Reduce运算后产生的结果看上去是被写入到HBase了，但是其实HBase中HLog和StoreFile中的文件在进行flush to disk操作时，这两个文件存储到了HDFS的DataNode中，HDFS才是永久存储。

2.ZooKeeper跟Hadoop Core、HBase有什么关系呢？ZooKeeper都提供了哪些服务呢？主要有：管理Hadoop集群中的NameNode，HBase中HBaseMaster的选举，Servers之间状态

同步等。具体一点，细一点说，单只HBase中ZooKeeper实例负责的工作就有：存储HBase的Schema，实时监控HRegionServer,存储所有Region的寻址入口，当然还有最常见的功能就是保证HBase集群中只有一个Master。

ZooKeeper 将按照如下方式实现加锁的操作：

1 ） ZooKeeper 调用 create （ ）方法来创建一个路径格式为“ _locknode_/lock- ”的节点，此节点类型为sequence （连续）和 ephemeral （临时）。也就是说，创建的节点为临时节点，并且所有的节点连续编号，即“lock-i ”的格式。

2 ）在创建的锁节点上调用 getChildren （ ）方法，来获取锁目录下的最小编号节点，并且不设置 watch 。

3 ）步骤 2 中获取的节点恰好是步骤 1 中客户端创建的节点，那么此客户端获得此种类型的锁，然后退出操作。

4 ）客户端在锁目录上调用 exists （ ）方法，并且设置 watch 来监视锁目录下比自己小一个的连续临时节点的状态。

5 ）如果监视节点状态发生变化，则跳转到第 2 步，继续进行后续的操作，直到退出锁竞争。

God has given me a gift. Only one. I am the most complete fighter in the world. My whole life, I have trained. I must prove I am worthy of someting.

rocky_24