1. ZooKeeper 是什么？

ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

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。

Zookeeper 保证了如下分布式一致性特性：

1. 顺序一致性
2. 原子性
3. 单一视图
4. 可靠性
5. 实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，**如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。**对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个 zookeeper 最新的 zxid。

2. ZooKeeper 提供了什么？

1. 文件系统
2. 通知机制

3. Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，**这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。**

4. ZAB 协议？

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。

ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 zookeeper 集群刚刚启动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

5. 四种类型的数据节点 Znode

- **PERSISTENT-持久节点**
- 除非手动删除，否则节点一直存在于 Zookeeper 上
- **EPHEMERAL-临时节点**
- 临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。
- **PERSISTENT_SEQUENTIAL-持久顺序节点**
- 基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。
- **EPHEMERAL_SEQUENTIAL-临时顺序节点**
- 基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

6. Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

工作机制：

- 客户端注册 watcher
- 服务端处理 watcher
- 客户端回调 watcher

Watcher 特性总结：

- 一次性
- 无论是服务端还是客户端，一旦一个 Watcher 被触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。
- 客户端串行执行
- 客户端 Watcher 回调的过程是一个串行同步的过程。
- 轻量
- Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。
- 客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。
- watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。
- 注册 watcher getData、exists、getChildren
- 触发 watcher create、delete、setData
- 当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。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。

7. 客户端注册 Watcher 实现

- 调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象
- 标记请求 request，封装 Watcher 到 WatchRegistration
- 封装成 Packet 对象，发服务端发送 request
- 收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理
- 请求返回，完成注册。

8. 服务端处理 Watcher 实现

1. 服务端接收 Watcher 并存储
2. 接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在 WatcherManager 的 WatchTable 和 watch2Paths 中去。
3. Watcher 触发
4. 以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例：

- 封装 WatchedEvent
- 将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个 WatchedEvent 对象
- 查询 Watcher
- 从 WatchTable 中根据节点路径查找 Watcher
- 没找到；说明没有客户端在该数据节点上注册过 Watcher
- 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）
- 调用 process 方法来触发 Watcher
- 这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知。

9. 客户端回调 Watcher

客户端 SendThread 线程接收事件通知，交由 EventThread 线程回调 Watcher。客户端的 Watcher 机制同样是一次性的，一旦被触发后，该 Watcher 就失效了。

10. ACL 权限控制机制

UGO（User/Group/Others）

目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

ACL（Access Control List）访问控制列表

包括三个方面：

权限模式（Scheme）

- IP：从 IP 地址粒度进行权限控制
- Digest：最常用，用类似于 username:password 的权限标识来进行权限配置，便于区分不同应用来进行权限控制
- World：最开放的权限控制方式，是一种特殊的 digest 模式，只有一个权限标识“world:anyone”
- Super：超级用户

授权对象

- 授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器灯。

权限 Permission

- CREATE：数据节点创建权限，允许授权对象在该 Znode 下创建子节点
- DELETE：子节点删除权限，允许授权对象删除该数据节点的子节点

- READ：数据节点的读取权限，允许授权对象访问该数据节点并读取其数据内容或子节点列表等
- WRITE：数据节点更新权限，允许授权对象对该数据节点进行更新操作
- ADMIN：数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关设置操作

11. Chroot 特性

3.2.0 版本后，添加了 Chroot 特性，该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot，那么该客户端对服务器的任何操作，都将会被限制在其自己的命名空间下。

通过设置 Chroot，能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应，在那些多个应用公用一个 Zookeeper 进群的场景下，对实现不同应用间的相互隔离非常有帮助。

12. 会话管理

分桶策略：将类似的会话放在同一区块中进行管理，以便于 Zookeeper 对会话进行不同区块的隔离处理以及同一区块的统一处理。

分配原则：每个会话的“下次超时时间点”（ExpirationTime）

计算公式：

ExpirationTime_ = currentTime + sessionTimeoutExpirationTime = (ExpirationTime_ / ExpirationInrerval + 1) * ExpirationInterval , ExpirationInterval 是指 Zookeeper 会话超时检查时间间隔，默认 tickTime

13. 服务器角色

Leader

- 事务请求的唯一调度和处理者，保证集群事务处理的顺序性
- 集群内部各服务的调度者

Follower

- 处理客户端的非事务请求，转发事务请求给 Leader 服务器
- 参与事务请求 Proposal 的投票
- 参与 Leader 选举投票

Observer

- 3.3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力
- 处理客户端的非事务请求，转发事务请求给 Leader 服务器
- 不参与任何形式的投票

14. Zookeeper 下 Server 工作状态

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。

- LOOKING：寻找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader，因此需要进入 Leader 选举状态。
- FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。
- LEADING：领导者状态。表明当前服务器角色是 Leader。
- OBSERVING：观察者状态。表明当前服务器角色是 Observer。

15. Leader 选举

Leader 选举是保证分布式数据一致性的关键所在。当 Zookeeper 集群中的一台服务器出现以下两种情况之一时，需要进入 Leader 选举。

(1) 服务器初始化启动。

(2) 服务器运行期间无法和 Leader 保持连接。

下面就两种情况进行分析讲解。

1. 服务器启动时期的 Leader 选举

若进行 Leader 选举，则至少需要两台机器，这里选取 3 台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器 Server1 启动时，其单独无法进行和完成 Leader 选举，当第二台服务器 Server2 启动时，此时两台机器可以相互通信，每台机器都试图找到 Leader，于是进入 Leader 选举过程。选举过程如下

(1) 每个 Server 发出一个投票。由于是初始情况，Server1 和 Server2 都会将自己作为 Leader 服务器来进行投票，每次投票会包含所推举的服务器的 myid 和 ZXID，使用(myid, ZXID)来表示，此时 Server1 的投票为(1, 0)，Server2 的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) 接受来自各个服务器的投票。集群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、是否来自 LOOKING 状态的服务器。

(3) 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行 PK，PK 规则如下

- 优先检查 ZXID。ZXID 比较大的服务器优先作为 Leader。

- 如果 ZXID 相同，那么就比较 myid。myid 较大的服务器作为 Leader 服务器。

对于 Server1 而言，它的投票是(1, 0)，接收 Server2 的投票为(2, 0)，首先会比较两者的 ZXID，均为 0，再比较 myid，此时 Server2 的 myid 最大，于是更新自己的投票为(2, 0)，然后重新投票，对于 Server2 而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

(4) 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于 Server1、Server2 而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了 Leader。

(5) 改变服务器状态。一旦确定了 Leader，每个服务器就会更新自己的状态，如果是 Follower，那么就变更为 FOLLOWING，如果是 Leader，就变更为 LEADING。

2. 服务器运行时期的 Leader 选举

在 Zookeeper 运行期间,Leader 与非 Leader 服务器各司其职,即便当有非 Leader 服务器宕机或新加入，此时也不会影响 Leader，但是一旦 Leader 服务器挂了，那么整个集群将暂停对外服务，进入新一轮 Leader 选举，其过程和启动时期的 Leader 选举过程基本一致。假设正在运行的有 Server1、Server2、Server3 三台服务器,当前 Leader 是 Server2,若某一时刻 Leader 挂了,此时便开始 Leader 选举。选举过程如下

(1) 变更状态。Leader 挂后，余下的非 Observer 服务器都会讲自己的服务器状态变更为 LOOKING，然后开始进入 Leader 选举过程。

(2) 每个 Server 会发出一个投票。在运行期间,每个服务器上的 ZXID 可能不同，此时假定 Server1 的 ZXID 为 123，Server3 的 ZXID 为 122；在第一轮投票中，Server1 和 Server3 都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。

(3) 接收来自各个服务器的投票。与启动时过程相同。

(4) 处理投票。与启动时过程相同，此时，Server1 将会成为 Leader。

(5) 统计投票。与启动时过程相同。

(6) 改变服务器的状态。与启动时过程相同。

2.2 Leader 选举算法分析

在 3.4.0 后的 Zookeeper 的版本只保留了 TCP 版本的 FastLeaderElection 选举算法。当一台机器进入 Leader 选举时，当前集群可能会处于以下两种状态

- 集群中已经存在 Leader。

- 集群中不存在 Leader。

对于集群中已经存在 Leader 而言，此种情况一般都是某台机器启动得较晚，在其启动之前，集群已经在正常工作，对这种情况，该机器试图去选举 Leader 时，会被告知当前服务器的 Leader 信息，对于该机器而言，仅仅需要和 Leader 机器建立起连接，并进行状态同步即可。而在集群中不存在 Leader 情况下则会相对复杂，其步骤如下

(1) 第一次投票。无论哪种导致进行 Leader 选举，集群的所有机器都处于试图选举出一个 Leader 的状态，即 LOOKING 状态，LOOKING 机器会向所有其他机器发送消息，该消息称为投票。投票中包含了 SID（服务器的唯一标识）和 ZXID（事务 ID），(SID, ZXID)形式来标识一次投票信息。假定 Zookeeper 由 5 台机器组成，SID 分别为 1、2、3、4、5，ZXID 分别为 9、9、9、8、8，并且此时 SID 为 2 的机器是 Leader 机器，某一时刻，1、2 所在机器出现故障，因此集群开始进行 Leader 选举。在第一次投票时，每台机器都会将自己作为投票对象，于是 SID 为 3、4、5 的机器投票情况分别为(3, 9)，(4, 8)， (5, 8)。

(2) 变更投票。每台机器发出投票后，也会收到其他机器的投票，每台机器会根据一定规则来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票，这个规则也是整个 Leader 选举算法的核心所在，其中术语描述如下

- vote_sid：接收到的投票中所推举 Leader 服务器的 SID。

- vote_zxid：接收到的投票中所推举 Leader 服务器的 ZXID。

- self_sid：当前服务器自己的 SID。

- self_zxid：当前服务器自己的 ZXID。

每次对收到的投票的处理，都是对(vote_sid, vote_zxid)和(self_sid, self_zxid)对比的过程。

规则一：如果 vote_zxid 大于 self_zxid，就认可当前收到的投票，并再次将该投票发送出去。

规则二：如果 vote_zxid 小于 self_zxid，那么坚持自己的投票，不做任何变更。

规则三：如果 vote_zxid 等于 self_zxid，那么就对比两者的 SID，如果 vote_sid 大于 self_sid，那么就认可当前收到的投票，并再次将该投票发送出去。

规则四：如果 vote_zxid 等于 self_zxid，并且 vote_sid 小于 self_sid，那么坚持自己的投票，不做任何变更。

结合上面规则，给出下面的集群变更过程。

(3) 确定 Leader。经过第二轮投票后，集群中的每台机器都会再次接收到其他机器的投票，然后开始统计投票，如果一台机器收到了超过半数的相同投票，那么这个投票对应的 SID 机器即为 Leader。此时 Server3 将成为 Leader。

由上面规则可知，通常那台服务器上的数据越新（ZXID 会越大），其成为 Leader 的可能性越大，也就越能够保证数据的恢复。如果 ZXID 相同，则 SID 越大机会越大。

2.3 Leader 选举实现细节

1. 服务器状态

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。

LOOKING：寻找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader，因此需要进入 Leader 选举状态。

FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。

LEADING：领导者状态。表明当前服务器角色是 Leader。

OBSERVING：观察者状态。表明当前服务器角色是 Observer。

2. 投票数据结构

每个投票中包含了两个最基本的信息，所推举服务器的 SID 和 ZXID，投票（Vote）在 Zookeeper 中包含字段如下

id：被推举的 Leader 的 SID。

zxid：被推举的 Leader 事务 ID。

electionEpoch：逻辑时钟，用来判断多个投票是否在同一轮选举周期中，该值在服务端是一个自增序列，每次进入新一轮的投票后，都会对该值进行加 1 操作。

peerEpoch：被推举的 Leader 的 epoch。

state：当前服务器的状态。

3. QuorumCnxManager：网络 I/O

每台服务器在启动的过程中，会启动一个 QuorumPeerManager，负责各台服务器之间的底层 Leader 选举过程中的网络通信。

(1) 消息队列。QuorumCnxManager 内部维护了一系列的队列，用来保存接收到的、待发送的消息以及消息的发送器，除接收队列以外，其他队列都按照 SID 分组形

成队列集合，如一个集群中除了自身还有 3 台机器，那么就会为这 3 台机器分别创建一个发送队列，互不干扰。

- recvQueue：消息接收队列，用于存放那些从其他服务器接收到的消息。

- queueSendMap：消息发送队列，用于保存那些待发送的消息，按照 SID 进行分组。

- senderWorkerMap：发送器集合，每个 SenderWorker 消息发送器，都对应一台远程 Zookeeper 服务器，负责消息的发送，也按照 SID 进行分组。

- lastMessageSent：最近发送过的消息，为每个 SID 保留最近发送过的一个消息。

(2) 建立连接。为了能够相互投票，Zookeeper 集群中的所有机器都需要两两建立起网络连接。QuorumCnxManager 在启动时会创建一个 ServerSocket 来监听 Leader 选举的通信端口(默认为 3888)。开启监听后，Zookeeper 能够不断地接收到来自其他服务器的创建连接请求，在接收到其他服务器的 TCP 连接请求时，会进行处理。为了避免两台机器之间重复地创建 TCP 连接，Zookeeper 只允许 SID 大的服务器主动和其他机器建立连接，否则断开连接。在接收到创建连接请求后，服务器通过对比自己和远程服务器的 SID 值来判断是否接收连接请求，如果当前服务器发现自己的 SID 更大，那么会断开当前连接，然后自己主动和远程服务器建立连接。一旦连接建立，就会根据远程服务器的 SID 来创建相应的消息发送器 SendWorker 和消息接收器 RecvWorker，并启动。

(3) 消息接收与发送。消息接收：由消息接收器 RecvWorker 负责，由于 Zookeeper 为每个远程服务器都分配一个单独的 RecvWorker，因此，每个 RecvWorker 只需要不断地从这个 TCP 连接中读取消息，并将其保存到 recvQueue 队列中。消息发送：由于 Zookeeper 为每个远程服务器都分配一个单独的 SendWorker，因此，每个 SendWorker 只需要不断地从对应的消息发送队列中获取出一个消息发送即可，同时将这个消息放入 lastMessageSent 中。在 SendWorker 中，一旦 Zookeeper 发现针对当前服务器的消息发送队列为空，那么此时需要从 lastMessageSent 中取出一个最近发送过的消息来进行再次发送，这是为了解决接收方在消息接收前或者接收到消息后服务器挂了，导致消息尚未被正确处理。同时，Zookeeper 能够保证接收方在处理消息时，会对重复消息进行正确的处理。

4. FastLeaderElection：选举算法核心

- 外部投票：特指其他服务器发来的投票。

- 内部投票：服务器自身当前的投票。

- 选举轮次：Zookeeper 服务器 Leader 选举的轮次，即 logicalclock。

- PK：对内部投票和外部投票进行对比来确定是否需要变更内部投票。

(1) 选票管理

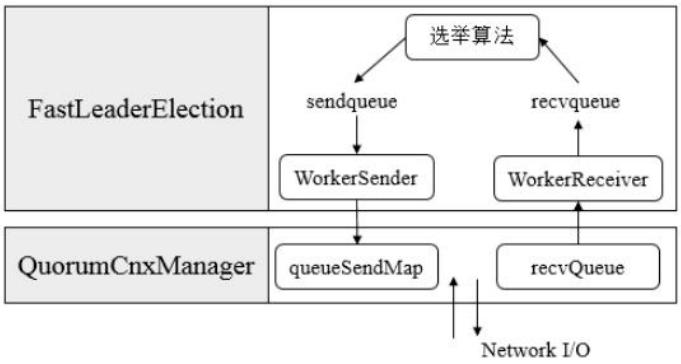
• sendqueue：选票发送队列，用于保存待发送的选票。

• recvqueue：选票接收队列，用于保存接收到的外部投票。

• WorkerReceiver：选票接收器。其会不断地从 QuorumCnxManager 中获取其他服务器发来的选举消息，并将其转换成一个选票，然后保存到 recvqueue 中，在选票接收过程中，如果发现该外部选票的选举轮次小于当前服务器的，那么忽略该外部投票，同时立即发送自己的内部投票。

• WorkerSender：选票发送器，不断地从 sendqueue 中获取待发送的选票，并将其传递到底层 QuorumCnxManager 中。

(2) 算法核心



上图展示了 FastLeaderElection 模块是如何与底层网络 I/O 进行交互的。Leader 选举的基本流程如下

1. 自增选举轮次。Zookeeper 规定所有有效的投票都必须在同一轮次中，在开始新一轮投票时，会首先对 logicalclock 进行自增操作。

2. 初始化选票。在开始进行新一轮投票之前，每个服务器都会初始化自身的选票，并且在初始化阶段，每台服务器都会将自己推举为 Leader。

3. 发送初始化选票。完成选票的初始化后，服务器就会发起第一次投票。Zookeeper 会将刚刚初始化好的选票放入 sendqueue 中，由发送器 WorkerSender 负责发送出去。

4. 接收外部投票。每台服务器会不断地从 recvqueue 队列中获取外部选票。如果服务器发现无法获取到任何外部投票，那么就会立即确认自己是否和集群中其他服务器保持着有效的连接，如果没有连接，则马上建立连接，如果已经建立了连接，则再次发送自己当前的内部投票。

5. 判断选举轮次。在发送完初始化选票之后，接着开始处理外部投票。在处理外部投票时，会根据选举轮次来进行不同的处理。

• 外部投票的选举轮次大于内部投票。若服务器自身的选举轮次落后于该外部投票对应服务器的选举轮次,那么就会立即更新自己的选举轮次(logicalclock)，并且清空所有已经收到的投票，然后使用初始化的投票来进行 PK 以确定是否变更内部投票。最终再将内部投票发送出去。

• 外部投票的选举轮次小于内部投票。若服务器接收的外选票的选举轮次落后于自身的选举轮次，那么 Zookeeper 就会直接忽略该外部投票，不做任何处理，并返回步骤 4。

• 外部投票的选举轮次等于内部投票。此时可以开始进行选票 PK。

6. 选票 PK。在进行选票 PK 时，符合任意一个条件就需要变更投票。

• 若外部投票中推举的 Leader 服务器的选举轮次大于内部投票,那么需要变更投票。

• 若选举轮次一致，那么就对比两者的 ZXID，若外部投票的 ZXID 大，那么需要变更投票。

• 若两者的 ZXID 一致，那么就对比两者的 SID，若外部投票的 SID 大，那么就需要变更投票。

7. 变更投票。经过 PK 后，若确定了外部投票优于内部投票，那么就变更投票，即使用外部投票的选票信息来覆盖内部投票，变更完成后，再次将这个变更后的内部投票发送出去。

8. 选票归档。无论是否变更了投票，都会将刚刚收到的那份外部投票放入选票集合 recvset 中进行归档。recvset 用于记录当前服务器在本轮次的 Leader 选举中收到的所有外部投票（按照服务队的 SID 区别，如{(1, vote1), (2, vote2)...}）。

9. 统计投票。完成选票归档后，就可以开始统计投票，统计投票是为了统计集群中是否已经有过半的服务器认可了当前的内部投票,如果确定已经有过半服务器认可了该投票，则终止投票。否则返回步骤 4。

10. 更新服务器状态。若已经确定可以终止投票，那么就开始更新服务器状态，服务器首选判断当前被过半服务器认可的投票所对应的 Leader 服务器是否是自己，若是自己，则将自己的服务器状态更新为 LEADING，若不是，则根据具体情况来确定自己是 FOLLOWING 或是 OBSERVING。

以上 10 个步骤就是 FastLeaderElection 的核心,其中步骤 4-9 会经过几轮循环，直到有 Leader 选举产生。

16. 数据同步

整个集群完成 Leader 选举之后，Learner（Follower 和 Observer 的统称）回向 Leader 服务器进行注册。当 Learner 服务器想 Leader 服务器完成注册后，进入数据同步环节。

数据同步流程：（均以消息传递的方式进行）

i. Learner 向 Learder 注册

ii. 数据同步

iii. 同步确认

Zookeeper 的数据同步通常分为四类：

- 直接差异化同步（DIFF 同步）
- 先回滚再差异化同步（TRUNC+DIFF 同步）
- 仅回滚同步（TRUNC 同步）
- 全量同步（SNAP 同步）

在进行数据同步前，Leader 服务器会完成数据同步初始化：

- peerLastZxid：从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid（该 Learner 服务器最后处理的 ZXID）
- minCommittedLog：Leader 服务器 Proposal 缓存队列 committedLog 中最小 ZXID
- maxCommittedLog：Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID

直接差异化同步（DIFF 同步）

场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog 之间

整个直接差异化同步过程中涉及的 Leader 和 Learner 之间的数据包通信如图 7-50 所示。

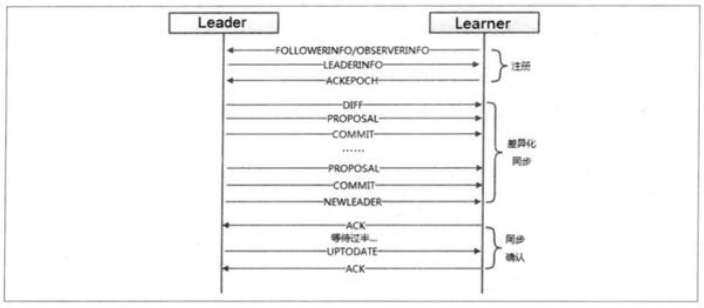


图 7-50. 直接差异化同步方式中 Leader 和 Learner 之间的数据通信

先回滚再差异化同步（TRUNC+DIFF 同步）

场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚--回滚到 Leader 服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID

仅回滚同步（TRUNC 同步）

场景：peerLastZxid 大于 maxCommittedLog

全量同步（SNAP 同步）

场景一：peerLastZxid 小于 minCommittedLog

场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid

17. zookeeper 是如何保证事务的顺序一致性的？

zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期; 纪元; 世; 新时代)用来标识 leader 周期，如果有新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

18. 分布式集群中为什么会有 Master？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

19. zk 节点宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；

如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)

2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)

20. zookeeper 负载均衡和 nginx 负载均衡区别

zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件；但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

21. Zookeeper 有哪几种几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

22. 集群最少要几台机器，集群规则是怎样的?

集群规则为 2N+1 台，N>0，即 3 台。

23. 集群支持动态添加机器吗？

其实就是水平扩容了，Zookeeper 在这方面不太好。两种方式：

- 全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。

- 逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5 版本开始支持动态扩容。

24. Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的?

不是。官方声明: 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。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。

一般是客户端执行 getData(“/节点 A”,true)，如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没有设置 watch 事件，就不再给客户端发送。

在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。

25. Zookeeper 的 java 客户端都有哪些？

java 客户端：zk 自带的 zkclient 及 Apache 开源的 Curator。

26. chubby 是什么，和 zookeeper 比你怎么看？

chubby 是 google 的，完全实现 paxos 算法，不开源。zookeeper 是 chubby 的开源实现，使用 zab 协议，paxos 算法的变种。

27. 说几个 zookeeper 常用的命令。

常用命令：ls get set create delete 等。

28. ZAB 和 Paxos 算法的联系与区别？

相同点：

- 两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行
- Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交

- ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot

不同点：

- ZAB 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。

29. Zookeeper 的典型应用场景

Zookeeper 是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对 Zookeeper 中丰富的数据节点进行交叉使用，配合 Watcher 事件通知机制，可以非常方便的构建一系列分布式应用中年都会涉及的核心功能，如：

1. 数据发布/订阅
2. 负载均衡
3. 命名服务
4. 分布式协调/通知
5. 集群管理
6. Master 选举
7. 分布式锁
8. 分布式队列

1. 数据发布/订阅

介绍

数据发布/订阅系统，即所谓的配置中心，顾名思义就是发布者发布数据供订阅者进行数据订阅。

目的

- 动态获取数据（配置信息）
- 实现数据（配置信息）的集中式管理和数据的动态更新

设计模式

- Push 模式
- Pull 模式

数据（配置信息）特性：

- 数据量通常比较小

- 数据内容在运行时会发生动态更新
- 集群中各机器共享，配置一致

如：机器列表信息、运行时开关配置、数据库配置信息等

基于 Zookeeper 的实现方式

- 数据存储：将数据（配置信息）存储到 Zookeeper 上的一个数据节点
- 数据获取：应用在启动初始化节点从 Zookeeper 数据节点读取数据，并在该节点上注册一个数据变更 Watcher
- 数据变更：当变更数据时，更新 Zookeeper 对应节点数据，Zookeeper 会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

2. 负载均衡

zk 的命名服务

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。

对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

7.zk 的命名服务（文件系统）

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

8.zk 的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。

9.Zookeeper 集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：是否有机器退出和加入、选举 master。

对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

10.Zookeeper 分布式锁（文件系统、通知机制）

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类,我们将 zookeeper 上的一个 znode 看作是一把锁,通过 createznode 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端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。

对于第二类， /distribute_lock 已经预先存在,所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

11.获取分布式锁的流程

clipboard.png

在获取分布式锁的时候在 locker 节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用 createNode 方法在 locker 下创建临时顺序节点，

然后调用 getChildren(“locker”)来获取 locker 下面的所有子节点，注意此时不用设置任何 Watcher。客户端获取到所有的子节点 path 之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非 locker 所有子节点中最小的，说明自己还没有获取到锁,此时客户端需要找到比自己小的那个节点,然后对其调用 exist()方法,同时对其注册事件监听器。之后,让这个被关注的节点删除,则客户端的 Watcher 会收到相应通知，此时再次判断自己创建的节点是否是 locker 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。

clipboard.png

代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于 BaseDistributedLock，实现了基于 Zookeeper 实现分布式锁的细节。

12.Zookeeper 队列管理（文件系统、通知机制）

两种类型的队列：

1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类,在约定目录下创建临时目录节点,监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下 Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。