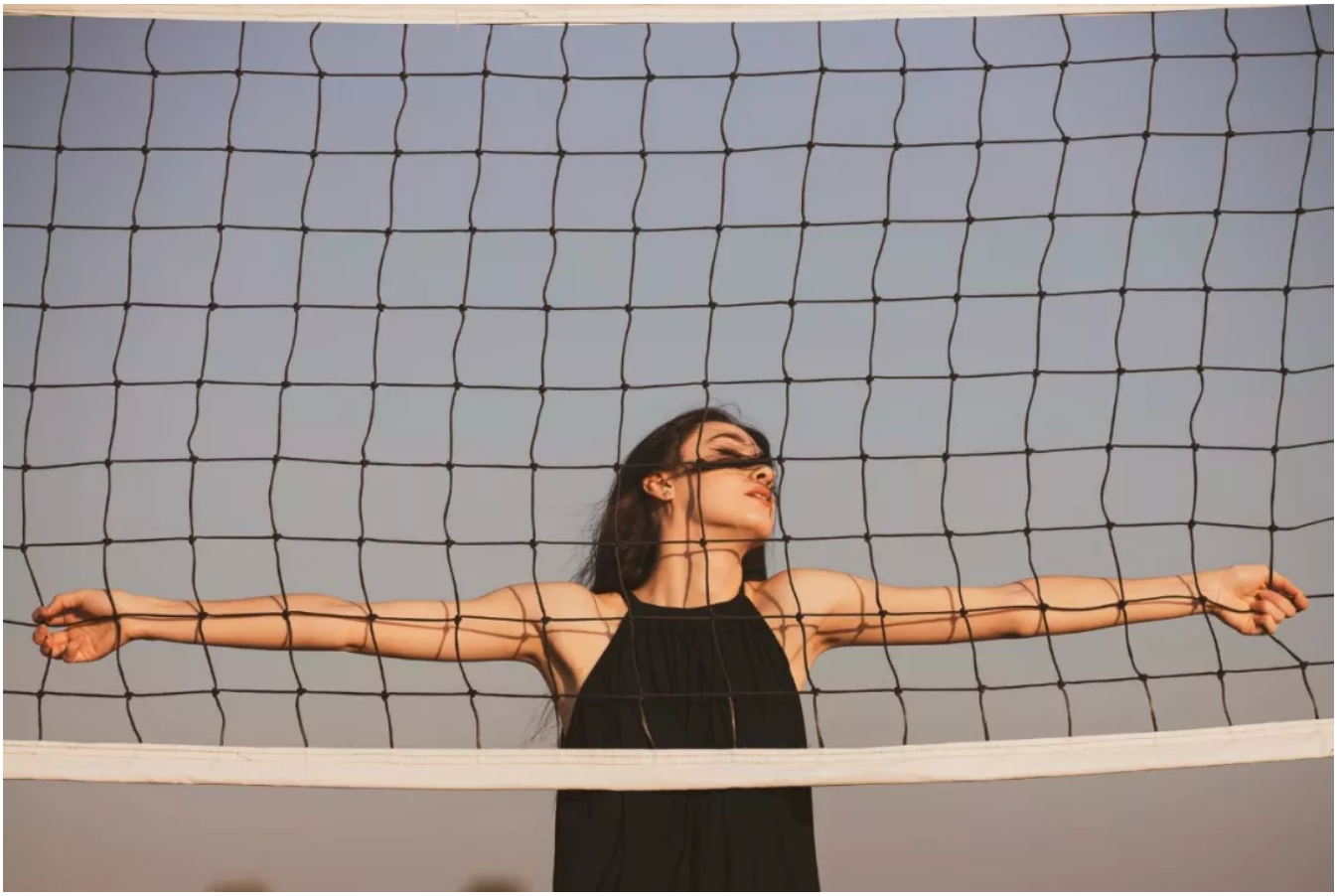
很遗憾，没有一篇文章能讲清楚ZooKeeper

原创 崔皓 51CTO技术栈 2019-11-22

> 互联网时代是信息爆发的时代，信息的高并发催生了分布式系统的广泛应用。



图片来自 Pexels

作为分布式系统解决方案的 ZooKeeper，被广泛应用于多个分布式场景。例如：数据发布/订阅，负载均衡，命名服务，集群管理等等。

因此，ZooKeeper 在分布式系统中扮演着重要的角色，今天通过一个简单的例子来看看它的实现原理。

从一个简单的例子开始

在分布式系统中经常会遇到这种情况，多个应用读取同一个配置。例如：A，B 两个应用都会读取配置 C 中的内容，一旦 C 中的内容出现变化，会通知 A 和 B。

一般的做法是在 A，B 中按照时钟频率询问 C 的变化，或者使用观察者模式来监听 C 的变化，发现变化以后再更新 A 和 B。那么 ZooKeeper 如何协调这种场景？

ZooKeeper 会建立一个 ZooKeeper 服务器，暂且称为 ZServer，用它来存放 C 的值。为 A，B 两个应用分别生成两个客户端，称作 ClientA 和 ClientB。

这两个客户端连接到 ZooKeeper 的服务器，并获取其中存放的 C。保存 C 值的地方在 ZooKeeper 服务端（Server）中称为 ZNode。

ClientA 和 ClientB 通过 ZooKeeper Server 获取 C 的值

ZNode

通过上面的例子，客户端 ClientA 和 ClientB 需要读取 C 的内容。这个 C 就作为树的叶子节点存放在 ZooKeeper 的 ZNode 中。

通常来说，为了提高效率 ZNode 是被存放在内存中的。ZNode 的数据模型是一棵树（ZNode Tree）。

好像我们从上图中看到的一样，树中的每个节点都可以存放数据，并且每个节点下面都可以存放叶子节点。

ZooKeeper 客户端通过 “/” 作为访问路径，访问数据。例如可以通过路径 “/RootNode/C” 来访问 C 变量。

为了方便客户端调用，ZooKeeper 会暴露一些命令：

访问 Znode 命令

作为存储媒介来说，ZNode分为持久节点和临时节点：

- **持久节点（PERSISTENT），**该数据节点被创建后，就一直存在于 ZooKeeper 服务器上，除非删除操作（delete）清除该节点。
- **临时节点（EPHEMERAL），**该数据节点的生命周期会和客户端（Client）会话（Session）绑定在一起。如果客户端（Client）会话丢失了，那么节点就自动清除掉。

如果把临时节点看成资源的话，当客户端和服务端产生会话并生成临时节点，一旦客户端与服务器中断联系，节点资源会被从 ZNode 中删除。

顺序节点（SEQUENTIAL），ZNode 节点被分配唯一个单调递增的整数。例如多个客户端在服务器 /tasks 上申请节点时，根据客户端申请的先后顺序，将数字追加到 /tasks/task 后面。

如果有三个客户端申请节点资源，那么在 /tasks 下面建立三个顺序节点，分别是 /tasks/task1，/tasks/task2，/tasks/task3。

顺序节点，在处理分布式事务的时候非常有帮助，当多个客户端（Client）协作工作的时候，会按照一定的顺序执行。

如果将上面的两类节点和顺序节点进行组合的话，就有四种节点类型，分别是持久节点，持久顺序节点，临时节点，临时顺序节点。

Watcher

上面说了 ZooKeeper 用来存放数据的 ZNode，并且把 C 的值存储在里面。如果 C 被更新了，两个客户端（ClientA、ClientB）如何获得通知呢？

ZooKeeper 客户端（Client）会在指定的节点（/RootNote/C）上注册一个 Watcher，ZNode 上的 C 被更新的时候，服务端就会通知 ClientA 和 ClientB。

通过三步来实现：

- **客户端注册 Watcher**
- **服务端处理 Watcher**
- **客户端回调 Watcher**

Watcher 注册，处理，回调

①客户端注册 Watcher

ZooKeeper 客户端创建 Watcher 的实例对象：

同时这个 Watcher 会保存在客户端本地，一直作为和服务端会话的 Watcher。

客户端可以通过 getData，getChildren 和 exist 方法来向服务端注册 Watcher。

客户端注册 Watcher 简图

同时需要注意的是在客户端发送 Watcher 到服务端注册的时候，会将这个要发送的 Watcher 在本地的 ZKWatchManager 中保存。

这样做的好处，就是当获得服务端的注册成功的信息以后，就不用将 Watcher 的具体内容回传给客户端了。

客户端只用在接到服务端响应以后，从本地的 ZKWatchManager 中获取 Watch 的信息进行处理即可。

②服务端处理 Watcher

服务端收到客户端的请求以后，交给 FinalRequestProcessor 处理，这个进程会去 ZNode 中获取对应的数据，同时会把 Watch 加入到 WatchManager 中。

这样下次这节点上的数据被更改了以后，就会通知注册 Watch 的客户端了。

服务端处理 Watch 过程

③客户端回调 Watcher

客户端在响应客户端 Watcher 注册以后，会发送 WathcerEvent 事件。作为客户端有对应的回调函数接受这个消息。

这里会通过 readResponse 方法统一处理：

在 SendTread 接受到服务端的通知以后，会将事件通过 EventThread.queueEvent 发送给 EventThread。

正如前面提到的，在客户端注册时，已经将 Watcher 的具体内容保存在 ZKWatchManager 一样了。

所以，EventTread 通过 EventType 就可以知道哪个 Watcher 被响应了（数据变化了）。

然后从 ZKWatchManager 取出具体 Watch 放到 waitingEvent 队列等待处理。

最后，由 EventThread 中的 processEvent 方法依次处理数据更新的响应。

版本（Version）

介绍完了 Watcher 机制，回头再来谈谈 ZNode 的版本（Version）。如果有一个客户端（ClientD），它尝试修改 C 的值，此时其他两个客户端会收到通知，并且进行后续的业务处理了。

那么在分布式系统中，会出现这么一种情况：在 ClientD 对 C 进行写入操作的时候，又有一个 ClientE 也对 C 进行写入操作。这两个 Client 会去竞争 C 资源，通常这种情况需要对 C 进行加锁操作。

两个 Client 竞争一个资源

因此引入 ZNode 版本（Version）概念。版本是用来保证分布式数据原子性操作的。

ZNode 的版本（Version）信息保存在 ZNode 的 Stat 对象中。有如下三种：

本例只关注“数据节点内容的版本号”，也就是 Version。

如果说 ClientD 和 ClientE 对 C 进行写入操作视作是一个事务的话。在执行写入操作之前，两个事务分别会获取节点上的值，即节点保存的数据和节点的版本号（Version）。

以乐观锁为例，对数据的写入会分成以下三个阶段：数据读取，写入校验和数据写入。例如 C 上的数据是 1， Version 是 0。

此时 ClientD 和 ClientE 都获取了这些信息。假设 ClientD 先做写入操作，在做写入校验的时候，发现之前获得的 Version 和节点上的 Version 是相同的，都是 1，因此直接执行数据写入。

写入以后，Version 由原来的 1 变成了 2。当 ClientE 做写入校验时，发现自己持有的 Version=1 和节点当前的 Version=2，不一样。于是，写入失败，重新获取 Version 和节点数据，再次尝试写入。

除了上述方案以外，还可以利用 ZNode 的有序性。在 C 下面建立多个有序的子节点。每当一个 Client 准备写入数据的时候，创建一个临时有序的节点。

节点的顺序根据 FIFO 算法，保证先申请写入的 Client 排在其前面。每个节点都有一个序号，后面申请的节点按照序号依次递增。

ClientD，ClientE 分别建立子 ZNode

每个 Client 在执行修改 C 操作的时候，都要检查有没有比自己序号小的节点，如果存在那么就进入等待。

直到比自己序号小的节点进行完毕以后，才轮到自己执行修改操作。从而保证了事物处理的顺序性。

会话（Session）

说完版本（Version）的概念，例子从原来的 ClientAB 已经扩充到了 ClientDE。这些客户端都会和 ZooKeeper 的服务端进行通信，或读取数据或修改数据。

我们将客户端与服务端完成的这种连接称为会话。ZooKeeper 的会话有 Connecting，Connected，Reconnecting，Reconnected 和 Close 这几种状态。

并且在服务端由专门的进程来管理他们，客户端初始化的时候就会根据配置自动连接服务器，从而建立会话，客户端连接服务器时会话处于 Connecting 状态。

一旦连接完成，就会进入 Connected 状态。如果出现延迟或者短暂失联，客户端会自动重连，Reconnecting 和 Reconnected 状态也就应运而生。

如果长时间超时，或者客户端断开服务器，ZooKeeper 会清理掉会话，以及该会话创建的临时数据节点，并且关闭和客户端的连接。

Session 作为会话实体，用来代表客户端会话，其包括 4 个属性：

- **SessionID**，用来全局唯一识别会话。
- **TimeOut**，会话超时事件。客户端在创造 Session 实例的时候，会设置一个会话超时的时间。
- **TickTime**，下次会话超时时间点。后面“分桶策略”会用到。
- **isClosing**，当服务端如果检测到会话超时失效了，会通过设置这个属性将会话关闭。

既然，会话是客户端与服务器之间的连接。在服务器端由 SessionTracker 管理会话。

SessionTracker 有一个工作就是，将超时的会话清除掉。于是“分桶策略”就登场了。

由于每个会话在生成的时候都会定义超时时间，通过当前时间+超时时间可以算出会话的过期时间。

由于 SessionTracker 不是实时监听会话超时，它是按照一定时间周期来监听的。

也就是说，如果没有到达 SessionTracker 的检查时间周期，即使有会话过期，SessionTracker 也不会去清除。由此，就引入会话超时计算公式，也就是 TickTime 的计算公式。

TickTime=（（当前时间+会话过期时间）/检查时间间隔+1）*检查时间间隔。

将这个值计算出来以后，SessionTracker 会把对应的会话按照这个时间放在对应的时间轴上面。SessionTracker 在对应的 TickTime 检查会话是否过期。

计算会话下次的过期时间

每当客户端连接上服务器都会做激活操作，同时每隔一段时间客户端会向服务器发送心跳检测。

服务器收到激活或者心跳检测以后，会重新计算会话过期时间，根据“分桶策略”进行重新调整。把会话从“老的区块”放到”新的区块“中去。

重新计算过期时间并且调整“分桶策略”

对于超时的会话，SessionTracker 也会做如下清理工作：

- 标记会话状态为“已关闭”，也就是设置 isClosing 为 True。
- 发起“会话关闭”的请求，让关闭操作在整个集群生效。
- 收集需要清理的临时节点。

- 添加“节点删除”的事务变更。
- 删除临时节点
- 移除会话
- 关闭客户端与服务端的连接

会话关闭以后客户端就无法从服务端获取/写入数据了。

服务群组（Leader，Follower，Observer）

前面提到了客户端如何通过会话与服务端保持联系，以及服务端是如何管理客户端会话（Session）的。

我们继续思考一下，这么多的服务端都依赖一个 ZooKeeper 服务器。一旦服务挂了，客户端就无法工作了。

为了提高 ZooKeeper 服务的可靠性，引入服务器集群的概念。从原来的单个服务器，扩充成多个服务器，即使某一台服务器挂了，其他的服务器也可以顶上来。

ZooKeeper 的服务器集群

这样看起来不错了，新的问题是，存在多个 ZooKeeper 服务器，那么客户端的请求发给哪台呢？服务器之间如何同步数据呢？如果一个服务挂掉了其他的服务器如何替代？这里介绍两个概念 Leader 和 Follower。

Leader 服务器，是事务请求（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。也是集群内部服务器的调度者。

它是整个集群的老大，其他的服务器接到事务请求都会转交给它，让它协调处理。

Follower 服务器，处理非事务请求（读操作），转发事务请求给 Leader 服务器。参与选举 Leader 的投票和事务请求 Proposal 的投票。

既然 Leader 是集群的老大，那么这个老大是如何产生的。ZooKeeper 有仲裁机制，通过服务器的选举产生这个 Leader，按照少数服从多数的原则。

因此，集群中服务器的个数一般都是奇数，例如：1，3，5。当然这里是建议。关于选举和仲裁都有一定的算法，一起来看看吧。

当众多服务器启动的时候，互相都不知道谁是 Leader，因此都会进入 Looking 状态，也就是在网络中寻找 Leader。

寻找的过程也是投票的过程，每个服务器会将服务器 ID 和事务 ID 作为投票信息发送给网络中其他的服务器。假设称它为投票信息 VOTE，它包括：（ServerID，ZXID）。

其中，ServerID 是服务器注册的 ID，随着服务器启动的顺序自动增加，后启动的服务器 ServerID 就大；ZXID 是服务器处理事物的 ID，随着事物的增加自动增加，同样后提交的事务 ZXID 也大一些。

其他的服务器收到 VOTE 信息以后会和自己的 VOTE 信息（ServerID，ZXID）进行比较。

如果收到的 VOTE（ServerID，ZXID）中的 ZXID 比自己的 ZXID 要大，那么把自己的 VOTE 修改成收到的 VOTE。

如果 ZXID 一样大，那么就比较 ServerID，将大的那个 ServerID 作为自己 VOTE 的 ServerID，转发给其他服务器。

再简单点说，如果事务 ID（ZXID）比自己的事务 ID（ZXID）要大，就把票投给这个服务器。如果事务 ID 一样，就把票投给 ServerID 大的服务器。

来个具体的例子，有三个服务器，他们的投票值分别是：

- **S1 (1，6)**
- **S2 (2，5)**
- **S3 (3，5)**

三个服务器分别把自己的 VOTE 发给其他两台服务器，S2 和 S3 收到 VOTE 以后发现 ZXID 为 6 的来自 S1 的 VOTE 比自己持有的 ZXID 要大，因此把自己的 VOTE 修改为（1，6）投出去，因此 S1 称为 Leader。

Leader 选举实例

同样，如果 S1 作为 Leader，因为某种原因挂掉或者长时间没有响应请求，其他的服务器也会进入 Looking 状态，开启投票仲裁模式寻找下一个 Leader。

成为新 Leader 以后会通过广播的方式将 ZNode 上的数据同步到其他的 Follower。

Leader 有了，整个服务器集群有了领袖，它可以处理客户端的事物请求。客户端的请求可以发给集群中任意一台服务器，无论是哪个服务器都会将事物请求转交给 Leader。

Leader 在将数据写入 ZNode 之前会向 ZooKeeper 的其他 Follower 进行广播。

这里广播用到了 ZAB 协议（Atomic Broadcast Protocol）是 Paxos 协议的实践。说白了就是一个两段提交。

PS：对分布式事务比较了解的同学应该知道两段提交和三段提交。

这里 ZooKeeper 通过以下方式实现两段提交：

- Leader 向所有 Follower 发送一个 PROPOSAL。
- 当 Follower 接收到 PROPOSAL 后，返回给 Leader 一个 ACK 消息，表示我收到 PROPOSAL，并且准备好了。
- Leader 仲裁数量（过半数）的 Follower 发送的 ACK 后（包括 Leader 自己），会发送消息通知 Follower 进行 COMMIT。
- 收到 COMMIT 以后，Follower 就开始干活，将数据写入到 ZNode 中。

ZAB 广播 PROPOSAL

选举了 Leader 领导集群，Leader 接受到 Client 的请求以后，也可以协调 Follower 工作了。

那么如果 Client 很多的情况下，特别是这些客户端都是做读操作的时候，ZooKeeper 服务器如何处理如此多的请求呢？这里引入 Observer 的概念。

Observer 和 Follower 基本一致，对于非事务请求（读操作），可以直接返回节点中的信息（数据从 Leader 中同步过来的）。

对于事务请求（写操作），会转交给 Leader 做统一处理。Observer 的存在就是为了解决大量客户端读请求。

Observer 和 Follower 的区别是，Observer 不参与仲裁投票，选举 Leader。

Observer 加入 Leader 和 Follower 大家庭

总结

全文用了一个简单的例子讲 ZooKeeper 的主要特性和实现原理，最后做个总结。

ZooKeeper 被用来协调和管理分布式系统，发挥着重要的作用。分布式系统由于其特性，应用分布在不同的物理主机或者网络中。

为了让它们协同工作，ZooKeeper 中的 ZNode 成为统一协调的重要部分，客户端通过 Client 间接到服务端的 ZNode 上，监听 ZNode 数据的变化。

同时 ZNode 支持的持久，临时和顺序性，以及版本（Version）控制，这些特性支持了分布式事务和锁的功能。

如果说，每一个 ZooKeeperClient 对 Server 的写入操作都是一次事务的话，ZooKeeper 服务端维护了大量的事务，并且通过“分桶策略”来管理它们，保证了 Client 与 Server 端协调工作。

为了提高 Server 的可靠性，ZooKeeper 引入了 Server 集群的概念。通过仲裁机制选举 Leader 来领导其他 Follower。

事物都由 Leader 来处理，通过两段提交的方式对其他 Server 发起广播。为了增强对非事务请求的处理效率，ZooKeeper 加入了 Observer 来帮忙。

ZooKeeper 包含的内容远不止上面说的这些，由于篇幅的原因无法一一道来。

为了方便大家理解，文中将一些原理做了简化处理，希望有机会和大家做深入的探讨，咱们下次见。

作者：崔皓

简介：十六年开发和架构经验，曾担任过惠普武汉交付中心技术专家，需求分析师，项目经理，后在创业公司担任技术/产品经理。善于学习，乐于分享。目前专注于技术架构与研发管理。

编辑：陶家龙、孙淑娟

征稿：有投稿、寻求报道意向技术人请联络 editor@51cto.com