## 1、Zookeeper应用场景介绍

 ZooKeeper是一个高可用的分布式数据管理与系统协调框架。基于对Paxos算法的实现，使该框架保证了分布式环境中数据的强一致性，也正是基 于这样的特性，使得zookeeper能够应用于很多场景。网上对zk的使用场景也有不少介绍，本文将结合作者身边的项目例子，系统的对zk的使用场景进 行归类介绍。 值得注意的是，zk并不是生来就为这些场景设计，都是后来众多开发者根据框架的特性，摸索出来的典型使用方法。因此，也非常欢迎你分享你在ZK使用上的奇 技淫巧。

### 数据发布与订阅（配置管理）

发布与订阅即所谓的配置管理，顾名思义就是将数据发布到zk节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，地址列表等就非常适合使用。

1. 索引信息和集群中机器节点状态存放在zk的一些指定节点，供各个客户端订阅使用。

2. 系统日志（经过处理后的）存储，这些日志通常2-3天后被清除。

3. 应用中用到的一些配置信息集中管理，在应用启动的时候主动来获取一次，并且在节点上注册一个Watcher，以后每次配置有更新，实时通知到应用，获取最新配置信息。

4. 业务逻辑中需要用到的一些全局变量，比如一些消息中间件的消息队列通常有个offset，这个offset存放在zk上，这样集群中每个发送者都能知道当前的发送进度。

5. 系统中有些信息需要动态获取，并且还会存在人工手动去修改这个信息。以前通常是暴露出接口，例如JMX接口，有了zk后，只要将这些信息存放到zk节点上即可。

### 1.2、Name Service

这个主要是作为分布式命名服务，通过调用zk的create node api，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。

### 1.3、分布通知/协调

ZooKeeper 中特有watcher注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对 ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容及子节点的），其中一个系统update了znode，那么另一个系统能 够收到通知，并作出相应处理。

1. 另一种心跳检测机制：检测系统和被检测系统之间并不直接关联起来，而是通过zk上某个节点关联，大大减少系统耦合。

2. 另一种系统调度模式：某系统有控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改 了ZK上某些节点的状态，而zk就把这些变化通知给他们注册Watcher的客户端，即推送系统，于是，作出相应的推送任务。

3. 另一种工作汇报模式：一些类似于任务分发系统，子任务启动后，到zk来注册一个临时节点，并且定时将自己的进度进行汇报（将进度写回这个临时节点），这样任务管理者就能够实时知道任务进度。

总之，使用zookeeper来进行分布式通知和协调能够大大降低系统之间的耦合。

### 1.4、分布式锁

分布式锁，这个主要得益于ZooKeeper为我们保证了数据的强一致性，即用户只要完全相信每时每刻，zk集群中任意节点（一个zk server）上的相同znode的数据是一定是相同的。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

  所谓保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把zk上的一个znode看作是一把锁，通过create znode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。

控制时序，就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distribute\_lock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制：CreateMode.EPHEMERAL\_SEQUENTIAL来指 定）。Zk的父节点（/distribute\_lock）维持一份sequence,保证子节点创建的时序性，从而也形成了每个客户端的全局时序。

### 1.5、集群管理

1. 集群机器监 控：这通常用于那种对集群中机器状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。这样的场景中，往往有一个监控系统，实时检测集群 机器是否存活。过去的做法通常是：监控系统通过某种手段（比如ping）定时检测每个机器，或者每个机器自己定时向监控系统汇报“我还活着”。 这种做法可行，但是存在两个比较明显的问题：1. 集群中机器有变动的时候，牵连修改的东西比较多。2. 有一定的延时。

利 用ZooKeeper有两个特性，就可以实时另一种集群机器存活性监控系统：a. 客户端在节点 x 上注册一个Watcher，那么如果 x 的子节点变化了，会通知该客户端。b. 创建EPHEMERAL类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。

例 如，监控系统在 /clusterServers 节点上注册一个Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。  
2. Master选举则是zookeeper中最为经典的使用场景了。

在 分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑（例如一些耗时的计算，网络I/O处理），往往只需要让整个集群中的某一台机器进行执行， 其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能，于是这个master选举便是这种场景下的碰到的主要问题。

利用ZooKeeper的强一致性，能够保证在分布式高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /currentMaster 节点，最终一定只有一个客户端请求能够创建成功。

利用这个特性，就能很轻易的在分布式环境中进行集群选取了。

另外，这种场景演化一下，就是动态Master选举。这就要用到 EPHEMERAL\_SEQUENTIAL类型节点的特性了。

上 文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终 在ZK上创建结果的一种可能情况是这样： /currentMaster/{sessionId}-1 , /currentMaster/{sessionId}-2 , /currentMaster/{sessionId}-3 ….. 每次选取序列号最小的那个机器作为Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是Master了。

1. 在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此之间索引数据一致。因此让集群中的Master来进行全量索引的生成， 然后同步到集群中其它机器。

2. 另外，Master选举的容灾措施是，可以随时进行手动指定master，就是说应用在zk在无法获取master信息时，可以通过比如http方式，向 一个地方获取master。

### 1.6、分布式队列

队列方面，我目前感觉有两种，一种是常规的先进先出队列，另一种是要等到队列成员聚齐之后的才统一按序执行。

对于第二种先进先出队列，和分布式锁服务中的控制时序场景基本原理一致，这里不再赘述。

第 二种队列其实是在FIFO队列的基础上作了一个增强。通常可以在 /queue 这个znode下预先建立一个/queue/num 节点，并且赋值为n（或者直接给/queue赋值n），表示队列大小，之后每次有队列成员加入后，就判断下是否已经到达队列大小，决定是否可以开始执行 了。这种用法的典型场景是，分布式环境中，一个大任务Task A，需要在很多子任务完成（或条件就绪）情况下才能进行。这个时候，凡是其中一个子任务完成（就绪），那么就去 /taskList 下建立自己的临时时序节点（CreateMode.EPHEMERAL\_SEQUENTIAL），当 /taskList 发现自己下面的子节点满足指定个数，就可以进行下一步按序进行处理了。

|  |  |  |
| --- | --- | --- |
| 场景类别 | 典型场景描述（ZK特性，使用方法） | 应用中的具体使用 |

## 2、Zookeeper的基本概念

### 2.1、zookeeper角色

多个zookeeper服务组成了一个zookeeper服务**集群**(其中2n+1个服务允许n个失效)。zookeeper系统模型以及其中的角色介绍如下：

|  |  |
| --- | --- |
| C:\Documents and Settings\liws0\桌面\20150511163557_494.jpg | C:\Documents and Settings\liws0\桌面\20150413150011_139.jpg |

集群中有两种角色，其中一个zookeeper服务为leader角色，剩下的所有zookeeper服务为follower角色；leader负责写服务和数据同步，follower负责提供读服务，leader失效后会在follower中重新选举新的leader。

### 2.2、设计目的

1、最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

2、可靠性：具有简单、健壮、良好的性能，如果消息m被一台服务器接受，那么它将被所有的服务器接受。

3、实时性：Zookeeper保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

4、等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

5、原子性：更新只能成功或者失败，没有中间状态。

6、顺序性：包括全局有序和偏序两种。全局有序是指如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布；偏序是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

# 3、****ZooKeeper****的工作原理

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做**Zab协议**。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动时或者在leader崩溃后，Zab就进入了恢复模式，当leader被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。**状态同步**保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了**递增的事务id号（zxid）**来标识事务。所有的提议[proposal]都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch，用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

|  |  |
| --- | --- |
| 每个Server在工作过程中有**三种状态：** | |
| LOOKING | 当前Server不知道leader是谁，正在搜寻。 |
| LEADING | 当前Server即为选举出来的leader。 |
| FOLLOWING | leader已经选举出来，当前Server与之同步。 |

## 3.1 选主流程

当leade崩溃或者leader失去大多数的follower时，这时候zk进入恢复模式；恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。

Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。

### 3.1.1、basic paxos选举流程

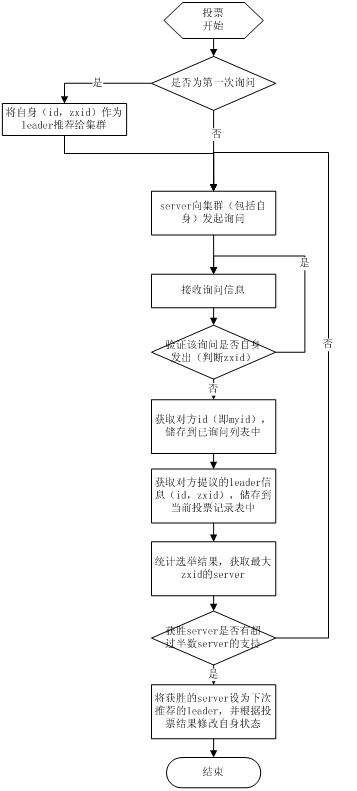
1 .选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

2 .选举线程首先向所有Server发起一次询问(包括自己)；

3 .选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

4.  收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

5.  线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数， 设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。

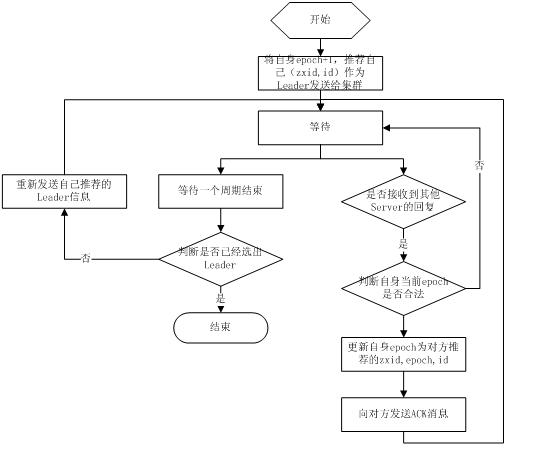


通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1。

每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。

### 3.1.2、fast paxos选举流程

fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和 zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。其流程图如下所示：



## 3.2、同步流程

选完leader以后，zk就进入状态同步过程。

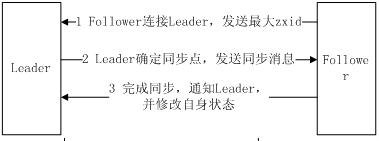
1. leader等待server连接；

2 .Follower连接leader，将最大的zxid发送给leader；

3 .Leader根据follower的zxid确定同步点；

4 .完成同步后通知follower 已经成为uptodate状态；

5 .Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。



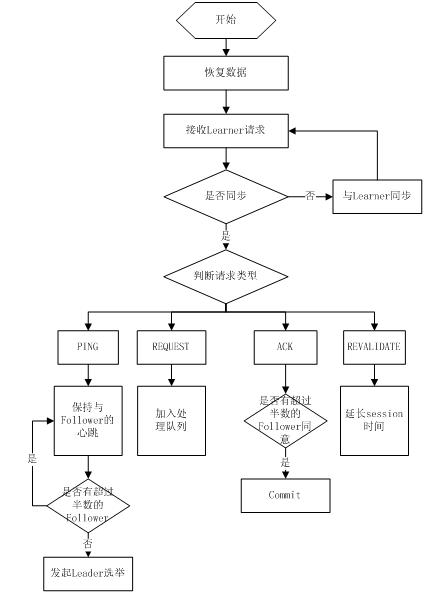
## 3.3、工作流程

### 3.3.1、Leader工作流程

Leader主要有三个功能：

1 .恢复数据；

2 .维持与Learner的心跳，接收Learner请求并判断Learner的请求消息类型；

3 .Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。**PING消息**是指Learner的心跳信息；**REQUEST消息**是Follower发送的提议信息，包括写请求及同步请求；**ACK消息**是Follower的对提议 的回复，超过半数的Follower通过，则commit该提议；**REVALIDATE消息**是用来延长SESSION有效时间。  
Leader的工作流程简图如下所示，在实际实现中，流程要比下图复杂得多，启动了三个线程来实现功能。

### 3.3.2、Follower工作流程

Follower主要有四个功能：

1. 向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；

2 .接收Leader消息并进行处理；

3 .接收Client的请求，如果为写请求，发送给Leader进行投票；

4 .返回Client结果。

Follower的消息循环处理如下几种来自Leader的消息：

1 .PING消息： 心跳消息；

2 .PROPOSAL消息：Leader发起的提案，要求Follower投票；

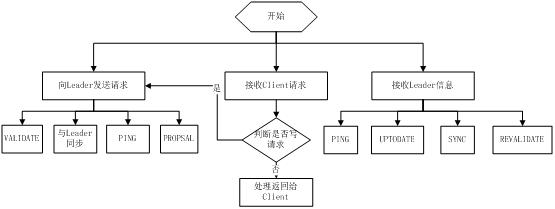
3 .COMMIT消息：服务器端最新一次提案的信息；

4 .UPTODATE消息：表明同步完成；

5 .REVALIDATE消息：根据Leader的REVALIDATE结果，关闭待revalidate的session还是允许其接受消息；

6 .SYNC消息：返回SYNC结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

Follower的工作流程简图如下所示，在实际实现中，Follower是通过5个线程来实现功能的。



### 3.3.3、observer工作流程

observer流程和Follower的唯一不同是：observer不会参加leader发起的投票。