Zookeeper

#### Zookeeper简介

Zookeeper是一个为分布式应用所设计的开源协调服务，提供同步，配置管理，分组和命名服务。

设计目标：

1. 简单化

Zookeeper允许分布式的进程通过共享体系内的命名空间来进行协调，其组织与标准的文件系统非常相似，是由一些寄存器组成，在zookeeper中，这些寄存器称之为znode。Zookeeper的数据存放在内存中，因此具有高吞吐量、低延迟的特性。

1. 健壮性

Zookeeper维护着一个处于内存中的状态镜像，一个位于存储器中的交换日志和快照。只要大部分服务器可用，那么zookeeper服务就是可用的。服务器与客户端之间通过TCP连接，并通过TCP来发送请求、获得响应、获取检测事件以及发送心跳。

1. 有序性

Zookeeper可以为每一次更新操作赋予一个版本号，此版本号全局有效，且不可重复。

1. 速度优势

Zookeeper在读取主要负载的时候特别快，当读工作比写工作更多的时候，执行性能更好。

#### Zookeeper基础

1. 角色



1. 节点

1）节点分类

a）PERSISTENT-持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

b）PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

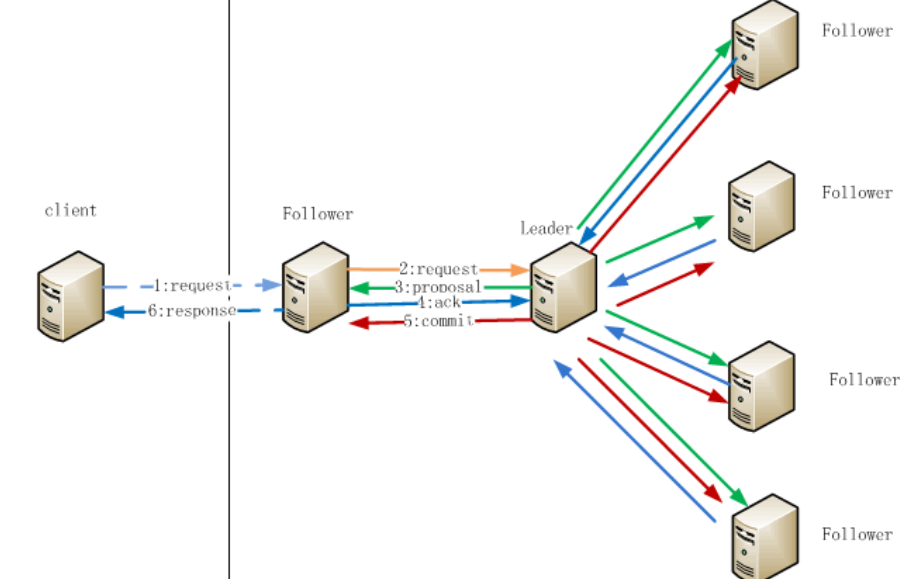
c）EPHEMERAL-临时目录节点

客户端与zookeeper断开连接后，该节点被删除

d）EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

2）节点数据操作流程



　注：1.在Client向Follwer发出一个写的请求

　　 　 2.Follwer把请求发送给Leader

　　 　3.Leader接收到以后开始发起投票并通知Follwer进行投票

　 　　4.Follwer把投票结果发送给Leader

5.Leader将结果汇总后如果需要写入，则开始写入同时把写入操作通知给 Leader，然后commit;

6.Follwer把请求结果返回给Client

　　　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。

1. 监听

1） 一个节点可以注册多个watcher，但是分成两种情况，当一个watcher实例多次注册时，zkClient也只会通知一次；当多个不同的watcher实例都注册时，zkClient会依次进行通知。

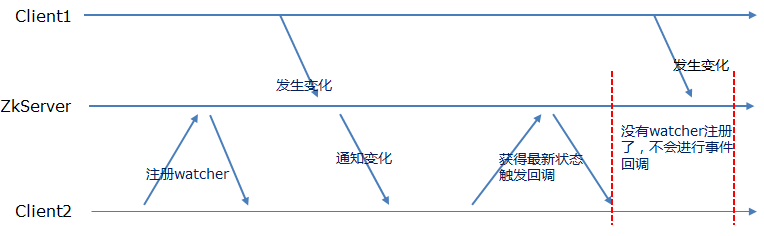
2） 监控同一个节点X的一个watcher实例，通过exist、getData等注册方式多次注册的，zkClient也只会通知一次。

3） zk.getData(“/node-x”,watcher)这种注册方式可以监控节点的NodeCreated事件，实际上是不行的（或者说没有意义）。当一个节点还不存在时，zk.getData这样设置的watcher是会抛出KeeperException$NoNodeException异常的，这次注册会失败，watcher也不会起作用；一旦node-x节点存在了，那这个节点的NodeCreated事件又有什么意义呢？

4） 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。

5） 轻量级设计，WatchedEvent 是 ZooKeeper 整个 Watcher 通知机制的最小通知单元，这个数据结构中只包含三部分的内容：通知状态、事件类型和节点路径。也就是说，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。例如针对 NodeDataChanged 事件，ZooKeeper 的 Watcher 只会通知客户指定数据节点的数据内容发生了变更，而对于原始数据以及变更后的新数据都无法从这个事件中直接获取到，而是需要客户端主动重新去获取数据，这也是 ZooKeeper 的 Watcher 机制的一个非常重要的特性。

6） 监听真空期



#### Zookeeper工作原理

1. 原子广播

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：

1. 恢复模式：当服务启动或者在领导者崩溃后，Zab就进入了恢复模式。
2. 广播模式：当领导者被选举出来，且大多数的server完成了和leader的状 态同步以后，恢复模式就结束了。
3. 顺序一致性

为了保证事务的顺序一致性。实现中zxid是一个64位的数字，它高32位是用epoch用来标志leader关系是否改变，每次一个新的leader选举出来，都会拥有一个新的epoch。低32位用来递增计数。

（1）Serverid：在配置server时，给定的服务器的标示id。

（2）Zxid：服务器在运行时产生的数据id，zxid越大，表示数据越新。对节点 状态改变的每一个操作，都将使节点收到一个zxid格式的时间戳，且时间戳全局有序。

（3）Epoch：选举的轮数，即逻辑时钟。随着选举的轮数++

（4）版本号：对节点的每一个操作，都将使节点版本号增加，每个节点维护三个版本号：version（节点数据版本号），cversion（子节点版本号），aversion（节点所拥有的ACL版本号）。

1. 设计原则
2. 最终一致性：客户端（Client）无论连接到哪个zk的节点，展示给他的视 图都是一样的。
3. 可靠性：消息message被到一台服务器接受，那么它到任何服务器都被接受。
4. 实时性：zk保证在一个时间间隔范围内获得服务器的更新信息，或者服务器失效信息。但是由于网络延时等一些其他原因，zk不能保证两个客户端同时得到更新或者失效信息。
5. 等待无关：慢的或者失效的客户端（Client）不得干预快速的client的请求，使得每个client都能有效的等待。
6. 原子性：更新只能成功或者失败，没有其他中间信息。
7. 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。
8. 选举算法

ZooKeeper 的选举算法有两种：一种是基于 Basic Paxos（Google Chubby 采用）实现的，另外 一种是基于 Fast Paxos（ZooKeeper 采用）算法实现的。系统默认的选举算法为 Fast Paxos。 并且 ZooKeeper 在 3.4.0 版本后只保留了 FastLeaderElection 算法。

1. basic paxos
2. 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；
3. 选举线程首先向所有Server发起一次询问(包括自己)；
4. 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；
5. 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；
6. 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。
7. 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.
8. fast paxos

Paxos描述了这样一个场景，有一个叫做Paxos的小岛(Island)上面住了一批居民，岛上面所有的事情由一些特殊的人决定，他们叫做议员(Senator)。议员的总数(Senator Count)是确定的，不能更改。岛上每次环境事务的变更都需要通过一个提议(Proposal)，每个提议都有一个编号(PID)，这个编号是一直增长的，不能倒退。每个提议都需要超过半数((Senator Count)/2 +1)的议员同意才能生效。每个议员只会同意大于当前编号的提议，包括已生效的和未生效的。如果议员收到小于等于当前编号的提议，他会拒绝，并告知对方：你的提议已经有人提过了。这里的当前编号是每个议员在自己记事本上面记录的编号，他不断更新这个编号。整个议会不能保证所有议员记事本上的编号总是相同的。现在议会有一个目标：保证所有的议员对于提议都能达成一致的看法。

好，现在议会开始运作，所有议员一开始记事本上面记录的编号都是0。有一个议员发了一个提议：将电费设定为1元/度。他首先看了一下记事本，嗯，当前提议编号是0，那么我的这个提议的编号就是1，于是他给所有议员发消息：1号提议，设定电费1元/度。其他议员收到消息以后查了一下记事本，哦，当前提议编号是0，这个提议可接受，于是他记录下这个提议并回复：我接受你的1号提议，同时他在记事本上记录：当前提议编号为1。发起提议的议员收到了超过半数的回复，立即给所有人发通知：1号提议生效！收到的议员会修改他的记事本，将1好提议由记录改成正式的法令，当有人问他电费为多少时，他会查看法令并告诉对方：1元/度。

现在看冲突的解决：假设总共有三个议员S1-S3，S1和S2同时发起了一个提议:1号提议，设定电费。S1想设为1元/度, S2想设为2元/度。结果S3先收到了S1的提议，于是他做了和前面同样的操作。紧接着他又收到了S2的提议，结果他一查记事本，咦，这个提议的编号小于等于我的当前编号1，于是他拒绝了这个提议：对不起，这个提议先前提过了。于是S2的提议被拒绝，S1正式发布了提议: 1号提议生效。S2向S1或者S3打听并更新了1号法令的内容，然后他可以选择继续发起2号提议。

好，我觉得Paxos的精华就这么多内容。现在让我们来对号入座，看看在ZK Server里面Paxos是如何得以贯彻实施的。

小岛(Island)——ZK Server Cluster

议员(Senator)——ZK Server

提议(Proposal)——ZNode Change(Create/Delete/SetData…)

提议编号(PID)——Zxid(ZooKeeper Transaction Id)

正式法令——所有ZNode及其数据

貌似关键的概念都能一一对应上，但是等一下，Paxos岛上的议员应该是人人平等的吧，而ZK Server好像有一个Leader的概念。没错，其实Leader的概念也应该属于Paxos范畴的。如果议员人人平等，在某种情况下会由于提议的冲突而产生一个“活锁”（所谓活锁我的理解是大家都没有死，都在动，但是一直解决不了冲突问题）。Paxos的作者Lamport在他的文章”The Part-Time Parliament“中阐述了这个问题并给出了解决方案——在所有议员中设立一个总统，只有总统有权发出提议，如果议员有自己的提议，必须发给总统并由总统来提出。好，我们又多了一个角色：总统。

总统——ZK Server Leader

又一个问题产生了，总统怎么选出来的？oh, my god! It’s a long story. 在淘宝核心系统团队的Blog上面有一篇文章是介绍如何选出总统的，有兴趣的可以去看看：http://rdc.taobao.com/blog/cs/?p=162

现在我们假设总统已经选好了，下面看看ZK Server是怎么实施的。

情况一：

屁民甲(Client)到某个议员(ZK Server)那里询问(Get)某条法令的情况(ZNode的数据)，议员毫不犹豫的拿出他的记事本(local storage)，查阅法令并告诉他结果，同时声明：我的数据不一定是最新的。你想要最新的数据？没问题，等着，等我找总统Sync一下再告诉你。

情况二：

屁民乙(Client)到某个议员(ZK Server)那里要求政府归还欠他的一万元钱，议员让他在办公室等着，自己将问题反映给了总统，总统询问所有议员的意见，多数议员表示欠屁民的钱一定要还，于是总统发表声明，从国库中拿出一万元还债，国库总资产由100万变成99万。屁民乙拿到钱回去了(Client函数返回)。

情况三：

总统突然挂了，议员接二连三的发现联系不上总统，于是各自发表声明，推选新的总统，总统大选期间政府停业，拒绝屁民的请求。

1. 同步流程
2. leader等待server连接。
3. follower连接到leader，将最大的zxid发送给leader。
4. leader根据zxid确定同步点。
5. 同步完成之后，通知follower成为uptodat状态。
6. follower收到uptodate消息后，开始接受client请求服务。

#### Zookeeper应用场景

1. 命名服务

Zookeeper可以创建全局可见的节点，这些节点都是树形结构的节点，可以存储数据， zookeeper的节点类似于文件系统保证了每个节点的唯一性，同时每个节点也可以为子节点记录创建时序从而保证时序性。

应用：

a）实际情况下我们可以用zookeeper来注册服务，譬如zookeeper+dubbo的使用：

服务提供者启动的时候会向zookeeper指定节点上通过创建节点的方式来注册服务

服务消费者根据唯一的路径来订阅该服务。

b）实际开发中，程序员A开发的a服务需要用到程序员B的b服务，但是b服务正在开发中，而此时a服务已经完成，所以A可以与B约定好把服务注册到zookeeper中，a注册一个监听器用于观察b的节点是否变化，如果变化那么a服务就可以访问b服务了

c）分布式ID生成器：

利用zookeeper节点的时序性，可以在分布式的环境下获得递增的数据，可以以此作为id。

1. 配置管理（数据发布与订阅）

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. 集群管理
2. 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：

（a）集群中机器有变动的时候，牵连修改的东西比较多。

（b）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。

（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。

（3） 加入和退出

每个机器都在一个父目录下创建临时子节点，然后监听这些子节点信息，一旦有机器挂掉，那么该节点也会删除，此时其他机器就会立即知道有机器挂掉了。新加入机器也是如此，会在父目录下新建一个临时节点，其它机器立马知道有新机器加入。

1. 分布式锁
2. 竞争唯一的锁

n个客户端去创建一个znode节点，最终只有一个客户端可以成功创建，成功创建的视为获得锁，当客户端完成操作断开删除该节点，其余客户端则watch该节点，当该节点删除时，继续执行创建操作。

缺点：

断开操作需要通知客户端，若客户端过多假设10000个，删除节点的时候需要通知10000个客户端，会产生性能问题。

1. 竞争最小的锁

n个客户端去创建临时顺序编号节点，编号最小的客户端获得锁，其余客户端分别watcher前面那个时序的节点，例如，1，2，3客户端按顺序创建，则2watcher1，3watcher2，1断开连接删除节点，2获得锁，依此类推。这样watcher节点的数量是一样的，但是不会有性能波动。

缺点：

需要创建大量的节点。

1. 队列管理
2. 同步队列

当这个队列的成员都聚齐的时候才可以用，在一个目录下创建节点，监听节点数目是否达到要求。

1. 顺序队列

类似于分布式锁的第二种方式，创建节点即为入列，入列都有一个编号，删除节点即为出列，但是删除只能删除编号最小的节点来保证先入先出。

1. 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。