# 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它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，读请求的吞吐量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)。而读读请求只会相对更新有序，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提供了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化，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务提案来进行事务请求处理。

# 5.四种类型的数据节点Znode

1.PERSISTENT-持久节点：除非手动删除，否则节点一直存在于Zookeepe上。

2.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连接端开不一定会话失效)，那么这个客户端创建的所有临时节点都会被移除。

3.PERSISTENT\_SEQUENTIAL-持久顺序节点：基本特性同持久节点，只是增加顺序属性，节点名后面会追加一个由父节点维护的自增整型数字。

4.EPHEMERAL\_SEQUENTIAL-临时顺序节点：基本属性同临时节点，增加了顺序属性，节点后面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时间触发了这个Watcher，服务端会向指定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：

1. 客户端注册watcher
2. 服务端注册watcher
3. 客户端回调watcher

Watcher特性总结

1. 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器的压力都非常大。
2. 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。
3. 轻量
   1. 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。
   2. 客户端向服务端注册Watcher的时候，并不会把真实的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。
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同的时刻监听到事件，由于Zookeeper本身提供了ordering guarante，即客户端在监听事件后，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。
5. 注册watcher getData、exists、getChildren
6. 触发Watcher create、delete、setData
7. 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的的。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可能丢失：对于一个未创建znode的exits 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

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

# 服务端处理Watcher实现

1. 服务端收到Watcher并存储

在接收到客户端请求，处理请求判断是否需要注册Watcher，需要的话将数据节点路径和ServerCnxn(ServerCnxn代表一个客户端可服务端的连接，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。

1. Watcher触发

以服务端接收到setData事务请求触发NodeDataChanged事件为例：

* 1. 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。
  2. 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(**这里可以看出Watcher在服务端是一次性的，触发一次就失效了**)。

1. 调用process方法来触发Watcher

这里process方法主要就是通过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线程接收到事件通知，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，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：

1. 权限模式(Scheme)：
   1. IP：从IP地址粒度进行权限控制。
   2. Degidt：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。
   3. World：最开放的权限控制方式，是一种特殊的degist模式，只要一个权限标识’world:anyone’。
   4. Super：超级用户。
2. 授权对象：授权对象指的是权限赋予用户或一个指定实体，例如IP地址或机器灯。
3. 权限(Permission)
   1. CREATE：数据节点创建权限，允许授权对象在该Znode下创建子节点。
   2. 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。
   3. 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。
   4. 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。
   5. 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.服务器角色

1.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。

2.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。

3.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。

# 12.Zookeeper下的Server工作状态

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

1. LOCKING：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。
2. FOLLOWING：跟随着状态。表面当前服务器角色是Follower。
3. LEADING：领导者状态。表面当前服务器角色是Leader。
4. OBSERVING：观察者状态。表明当前服务器角色Observer。

# 13.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。

## 13.1.服务器启动时期的Leader选举

若进行Leader选举，则至少需要两天机器，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Server3启动时，此时两台机器可以互相通信，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：

1.1：每个Server发出一个选票。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，每次投票都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Server1(1,0)，Server2的投票为(2,0)，然后各自将这个投票发给集群中的其它机器。

1.2：接受来自各个服务器的投票。集群中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LOCKING状态的服务器。

1.3：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，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，其无需更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

1.4：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收到相同的投票信息**，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，此时便认为已经选举出了Leader。

1.5：改变服务器的状态。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。

## 13.2.服务器运行期间的Leader选举

在Zookeeper运行期间，Leader和非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，其过程和启动期间的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挂了之后，余下的非Observer服务器都会将自己的服务器状态变更为LOCKING，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ZXID可能不同，此时假定Server1的ZXID为123，Server2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的所有机器。
3. 接收来自各个服务器的投票。与启动过程相同。
4. 处理投票。与启动过程相同，此时，Server1将会成为Leader。
5. 统计投票。与启动过程相同。
6. 改变服务器状态。与启动过程相同。

## 13.3.Leader选举算法

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

### 13.3.1集群中已经存在Leader。

对于集群中已经存在Leader而言，此种情况下一般都是某台机器启动的比较晚，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。

### 13.3.2集群中不存在Leader

对于集群中不存在Leader的情况，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，集群中的所有机器都处于试图选举出一个Leader的状态，即LOCKING状态，LOCKING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SID(服务器的唯一标识)和ZXID(事务ID)，(SID,ZXID)形式标识一次投票信息。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所在的机器出现故障，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。每条=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分润来决定是否需要变更自己的投票。
3. 确定Leader。经过第二轮投票后，集群中的每台机器都会再次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。
4. 由以上规则可知，通常哪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。

## 13.4选举算法

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，在开始新一轮投票中，会首先对logicalclock进行自增操作。
2. 初始化投票。在开始进行新一轮投票之前，每个服务器都会初始化自身投票，并且在初始化阶段都会将自己推举为Leader。
3. 发送初始化选票。完成选票的初始化之后，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，并由发送器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。
6. 选票PK。在进行选票PK时，符合条件就需要变更投票。
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。