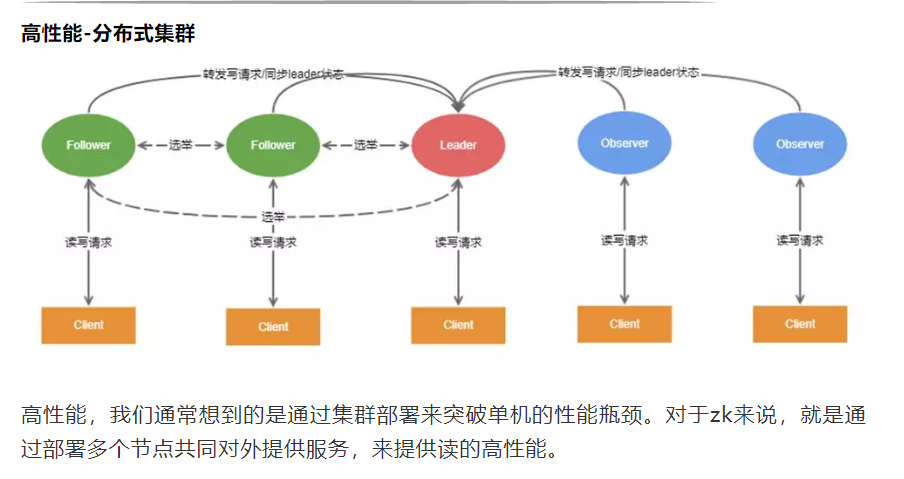
# Zookeeper基础

## 高性能分布式集群

**集群：**



Master/Slave模式。

在zookeeper中部署多台节点对外提供服务，客户端可以连接到任意一个节点。

每个节点的数据都是一样的。

节点根据角色分为Leader节点与Learner节点（包括Follower节点与Observer节点）。

集群中，只有一个Leader节点，完成所有的写请求处理。

每次写请求都会生成一个全局的唯一的64位整型的事务ID(可以理解为全局的数据的版本号)。

Learner节点可以有很多，每个Leaner可以独自处理读请求，转写请求到Leader节点。

当Leader节点挂掉后，会从Follower节点中通过选举方式选出一个Leader提供对外服务。

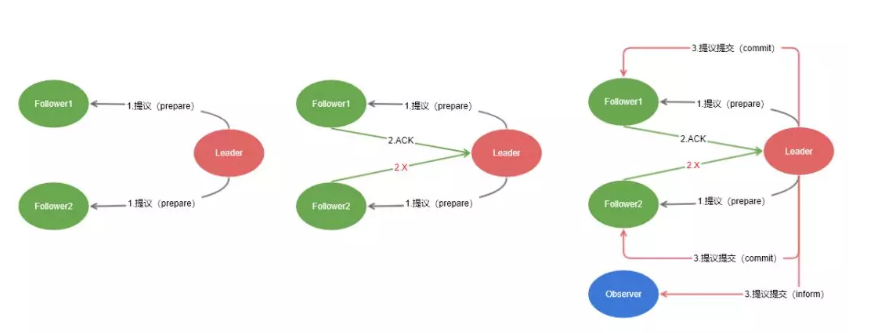
Follower节点与Observer节点区别在于不参与选举和提议的事务过半处理。

集群通常是按照奇数个节点进行部署（偶然太对容灾没啥影响，浪费机器）。

## 数据一致性

**（zab协议-原子广播协议）**

通过集群的部署，根据CAP原理，这样，可能导致同一个数据在不同节点上的数据不一致。zookeeper通过zab原子广播协议来保证数据在每一个节点上的一致性。原子广播协议（类似2PC提交协议）大概分为3个步骤。



Leader包装写请求，生成唯一zxid，发起提议，广播给所有Follower。

Follower收到提议后，写入本地事务日志，根据自身情况，是否同意该事务的提交。

Leader收到过半的Follower同意，自己先添加事务。然后对所有的Learner节点发送提交事务请求。

需要说明的是，zookeeper对数据一致性的要求是：

顺序一致性：严格按照事务发起的顺序执行写操作。

原子性：所有事务请求的结果在集群中的所有节点上的应用情况是一致的。

单一视图：客户端访问任何一个节点，看到的数据模型都是一致的。

实时性：保证在极小一段时间客户端最终可以从服务读取最新数据状态（如果要实时，需要客户端调用syn方法）。

## 可用性-leader选举

**（zab协议-崩溃恢复协议）**

在整个集群中，写请求都集中在一个Leader节点上，如果Leader节点挂了咋办呢？

当集群初始化或Follower无法联系上Leader节点的时候，每个Follower开始进入选举模式。选举步骤如下：

Follower节点第一次投票先投自己，然后将自己的选票广播给剩余的Follower节点。

Follower节点接收到其他的选票。

选票比较：比较自己的与接收的选票的投票更有。

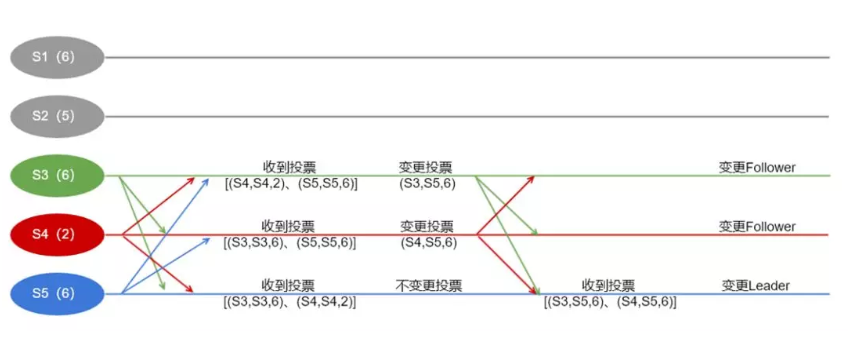
如果资金的选票不是最优选票，变更自己的选票，投最优选票的节点。

统计自己收到的选票，如果某个节点获得了过半的节点的投票。确认该节点为新的Leader节点。

确认Leader节点后，每个节点变更自己的角色。完成投票选举。

选举原则：谁的数据最新，谁就有优先被选为Leader的资格。

举个例子，假如现在zk集群有5个节点，然后挂掉了2个节点。剩余节点S3,S4,S6开始进行选举，他们的最大事务ID分别是6,2,6。定义投票结构为（投票的节点ID，被投节点ID，被投节点最大事务ID）。



初始状态，S3,S4,S5分别投自己，并带上自己的最大事务ID。

S3,S4,S5分别对自己收到的2票与自己的1票做比较。

S5发现自己的是最优投票，不变更投票，S3,S4发现S5的投票是最优解，更改投票。

S3,S4广播自己变更的投票。

最后大家都确认了S5是Leader，S5节点状态变更为Leader节点，S3,S4变更为Follower节点。

到这里，就是选举的主要过程。

## 数据的持久化

zookeeper所有数据都存在内存中。

zookeeper会定期将内存dump到磁盘中，形成数据快照。

zookeeper每次的事务请求，都会先接入到磁盘中，形成事务日志。

全量数据 = 数据快照 + 事务日志。

## Eureka比Zookeeper比较

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP, 而Eureka则是AP。

3.1 Zookeeper保证CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

3.2 Eureka保证AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)

3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

4. 总结

Eureka作为单纯的服务注册中心来说要比zookeeper更加“专业”，因为注册服务更重要的是可用性，我们可以接受短期内达不到一致性的状况。不过Eureka目前1.X版本的实现是基于servlet的Java web应用，它的极限性能肯定会受到影响。期待正在开发之中的2.X版本能够从servlet中独立出来成为单独可部署执行的服务。

# Zookeeper进阶

## ZooKeeper用途

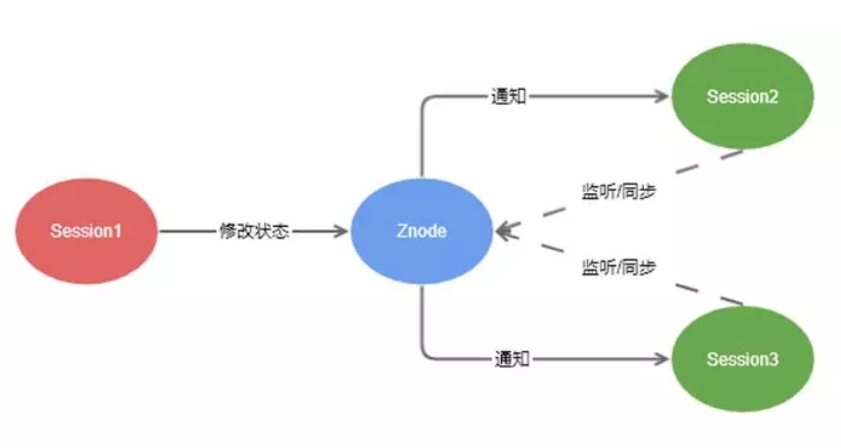
Zookeeper的用途包括统一配置管理、统一命名服务、分布式锁、集群管理。

**统一配置管理**：

我们可以将common.yml这份配置放在ZooKeeper的Znode节点中，系统A、B、C监听着这个Znode节点有无变更，如果变更了，及时响应。

**统一命名服务**：

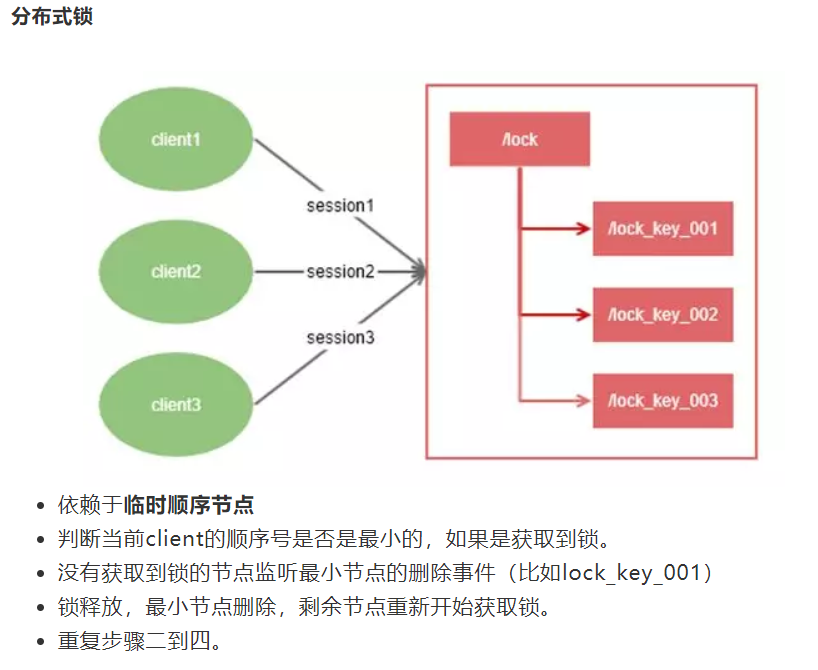
将几个服务的ip存于一个Znode下



**分布式锁**：

系统A、B、C都去访问/locks节点。访问的时候会创建带顺序号的临时/短暂(EPHEMERAL\_SEQUENTIAL)节点，比如，系统A创建了id\_000000节点，系统B创建了id\_000002节点，系统C创建了id\_000001节点。

接着，拿到/locks节点下的所有子节点(id\_000000,id\_000001,id\_000002)，判断自己创建的是不是最小的那个节点。如果是，则拿到锁（释放锁：执行完操作后，把创建的节点给删掉）。如果不是，则监听比自己要小1的节点变化。



**比较：**

从理解的难易程度角度（从低到高）

数据库 > 缓存（Redis） > Zookeeper

从实现的复杂性角度（从低到高）

Zookeeper >= 缓存（Redis） > 数据库

从性能角度（从高到低）

缓存（Redis） > Zookeeper >= 数据库

从可靠性角度（从高到低）

Zookeeper > 缓存（Redis） > 数据库

**集群管理**：

以我们三个系统A、B、C为例，在ZooKeeper中创建临时节点即可（/groupMember/A，/groupMember/B，/groupMember/C）, 只要系统A挂了，那/groupMember/A这个节点就会删除，通过监听groupMember下的子节点，系统B和C就能够感知到系统A已经挂了。(新增也是同理)

**实现方式**

ZooKeeper通过Znode的节点类型+监听机制来实现以上功能

ZooKeeper的数据结构，跟Unix文件系统非常类似，可以看做是一颗树，每个节点叫做ZNode。每一个节点可以通过路径来标识，Znode分为两种类型：

短暂/临时(Ephemeral)：当客户端和服务端断开连接后，所创建的Znode(节点)会自动删除

持久(Persistent)：当客户端和服务端断开连接后，所创建的Znode(节点)不会删除

ZooKeeper还配合了监听器才能够做那么多事的。常见的监听场景有以下两项：

监听Znode节点的数据变化

监听子节点的增减变化

## Zookeeper选举原理

如何在zookeeper集群中选举出一个leader,zookeeper使用了三种算法,具体使用哪种算法,在配置文件中是可以配置的,对应的配置项是”electionAlg”,其中1对应的是LeaderElection算法,2对应的是AuthFastLeaderElection算法,3对应的是FastLeaderElection算法.默认使用FastLeaderElection算法.其他两种算法我没有研究过,就不多说了.

要理解这个算法,最好需要一些paxos算法的理论基础.

1) 数据恢复阶段

首先,每个在zookeeper服务器先读取当前保存在磁盘的数据,zookeeper中的每份数据,都有一个对应的id值,这个值是依次递增的,换言之,越新的数据,对应的ID值就越大.

2) 向其他节点发送投票值

在读取数据完毕之后,每个zookeeper服务器发送自己选举的leader（首次选自己）,这个协议中包含了以下几部分的数据:

a)所选举leader的id(就是配置文件中写好的每个服务器的id) ,在初始阶段,每台服务器的这个值都是自己服务器的id,也就是它们都选举自己为leader.

b) 服务器最大数据的id,这个值大的服务器,说明存放了更新的数据.

c)逻辑时钟的值,这个值从0开始递增,每次选举对应一个值,也就是说: 如果在同一次选举中,那么这个值应该是一致的 ; 逻辑时钟值越大,说明这一次选举leader的进程更新.

d) 本机在当前选举过程中的状态,有以下几种:LOOKING,FOLLOWING,OBSERVING,LEADING,顾名思义不必解释了吧.

~~3）接受来自其他节点的数据~~

~~每台服务器将自己服务器的以上数据发送到集群中的其他服务器之后,同样的也需要接收来自其他服务器的数据,它将做以下的处理:~~

~~（1）如果所接收数据中服务器的状态还是在选举阶段(LOOKING 状态),那么首先判断逻辑时钟值,又分为以下三种情况:~~

~~a) 如果发送过来的逻辑时钟大于目前的逻辑时钟,那么说明这是更新的一次选举,此时需要更新一下本机的逻辑时钟值,同时将之前收集到的来自其他服务器的选举清空,因为这些数据已经不再有效了.然后判断是否需要更新当前自己的选举情况.在这里是根据选举leader id,保存的最大数据id来进行判断的,这两种数据之间对这个选举结果的影响的权重关系是:首先看数据id,数据id大者胜出;其次再判断leader id,leader id大者胜出.然后再将自身最新的选举结果(也就是上面提到的三种数据）广播给其他服务器).~~

~~b) 发送过来数据的逻辑时钟小于本机的逻辑时钟，说明对方在一个相对较早的选举进程中,这里只需要将本机的数据发送过去就是了~~

~~c) 两边的逻辑时钟相同,此时也只是调用totalOrderPredicate函数判断是否需要更新本机的数据,如果更新了再将自己最新的选举结果广播出去就是了.~~

~~然后再处理两种情况:~~

~~1)服务器判断是不是已经收集到了所有服务器的选举状态,如果是，那么这台服务器选举的leader就定下来了，然后根据选举结果设置自己的角色(FOLLOWING还是LEADER),然后退出选举过程就是了.~~

~~2)即使没有收集到所有服务器的选举状态,也可以根据该节点上选择的最新的leader是不是得到了超过半数以上服务器的支持,如果是,那么当前线程将被阻塞等待一段时间(这个时间在finalizeWait定义)看看是不是还会收到当前leader的数据更优的leader,如果经过一段时间还没有这个新的leader提出来，那么这台服务器最终的leader就确定了,否则进行下一次选举.~~

~~（2) 如果所接收服务器不在选举状态,也就是在FOLLOWING或者LEADING状态~~

~~做以下两个判断:~~

~~a) 如果逻辑时钟相同,将该数据保存到recvset,如果所接收服务器宣称自己是leader,那么将判断是不是有半数以上的服务器选举它,如果是则设置选举状态退出选举过程~~

~~b) 否则这是一条与当前逻辑时钟不符合的消息,那么说明在另一个选举过程中已经有了选举结果,于是将该选举结果加入到outofelection集合中,再根据outofelection来判断是否可以结束选举,如果可以也是保存逻辑时钟,设置选举状态,退出选举过程.~~

~~代码如下:~~

服务器接受来自其他各个服务器的投票，并判断投票的有效性（包括检查是否是本轮投票、是否来自LOOKING状态的服务器）;

处理投票（服务器将自己的投票和收到的投票进行对比，先检查ZXID，大的服务器作为leader；如果ZXID相同，检查myid，myid大的作为leader；更新投票），将最终的投票重新发出去；

统计投票：每次投票后，服务器统计所有投票，判断是否有过半的服务器收到相同的投票信息；如果是，则选举出了新的leader，如果不是，重新开始投票；

改变服务器状态：follower将自己的状态改为FOLLOWING,leader将自己的状态改为LEADING。

以一个简单的例子来说明整个选举的过程.

假设有五台服务器组成的zookeeper集群,它们的id从1-5,同时它们都是最新启动的,也就是没有历史数据,在存放数据量这一点上,都是一样的.假设这些服务器依序启动,来看看会发生什么.

1) 服务器1启动,此时只有它一台服务器启动了,它发出去的报没有任何响应,所以它的选举状态一直是LOOKING状态

2) 服务器2启动,它与最开始启动的服务器1进行通信,互相交换自己的选举结果,由于两者都没有历史数据,所以id值较大的服务器2胜出,但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3),所以服务器1,2还是继续保持LOOKING状态.

3) 服务器3启动,根据前面的理论分析,服务器3成为服务器1,2,3中的老大,而与上面不同的是,此时有三台服务器选举了它,所以它成为了这次选举的leader.

4) 服务器4启动,根据前面的分析,理论上服务器4应该是服务器1,2,3,4中最大的,但是由于前面已经有半数以上的服务器选举了服务器3,所以它只能接收当小弟的命了.

5) 服务器5启动,同4一样,当小弟.

## ZK为什么不提供一个永久性的Watcher注册机制

不支持用持久Watcher的原因很简单，ZK无法保证性能。

使用watch需要注意的几点

Watches通知是一次性的，必须重复注册.

发生CONNECTIONLOSS之后，只要在session\_timeout之内再次连接上（即不发生SESSIONEXPIRED），那么这个连接注册的watches依然在。

节点数据的版本变化会触发NodeDataChanged，注意，这里特意说明了是版本变化。存在这样的情况，只要成功执行了setData()方法，无论内容是否和之前一致，都会触发NodeDataChanged。

对某个节点注册了watch，但是节点被删除了，那么注册在这个节点上的watches都会被移除。

同一个zk客户端对某一个节点注册相同的watch，只会收到一次通知。

Watcher对象只会保存在客户端，不会传递到服务端。

## 创建的临时节点什么时候会被删除，是连接一断就删除吗？延时是多少？

连接断了之后，ZK不会马上移除临时数据，只有当SESSIONEXPIRED之后，才会把这个会话建立的临时数据移除。因此，用户需要谨慎设置Session\_TimeOut

## 是否可以拒绝单个IP对ZK的访问,操作

ZK本身不提供这样的功能，它仅仅提供了对单个IP的连接数的限制。你可以通过修改iptables来实现对单个ip的限制；当然，你也可以通过这样的方式来解决。<https://issues.apache.org/jira/browse/ZOOKEEPER-1320>

## ZooKeeper集群中服务器之间是怎样通信的

Leader服务器会和每一个Follower/Observer服务器都建立TCP连接，同时为每个F/O都创建一个叫做LearnerHandler的实体。LearnerHandler主要负责Leader和F/O之间的网络通讯，包括数据同步，请求转发和Proposal提议的投票等。Leader服务器保存了所有F/O的LearnerHandler。

## 出现调用超时com.alibaba.dubbo.remoting.TimeoutException异常怎么办

通常是业务处理太慢，可在服务提供方执行：jstack PID > jstack.log 分析线程都卡在哪个方法调用上，这里就是慢的原因。如果不能调优性能，请将timeout设大。

## 出现java.util.concurrent.RejectedExecutionException或者Thread pool exhausted怎么办？

RejectedExecutionException表示线程池已经达到最大值，并且没有空闲连，拒绝执行了一些任务。

Thread pool exhausted通常是min和max不一样大时，表示当前已创建的连接用完，进行了一次扩充，创建了新线程，但不影响运行。

原因可能是连接池不够用，请调整dubbo.properites中的：

// 设成一样大，减少线程池收缩开销

dubbo.service.min.thread.pool.size=200

dubbo.service.max.thread.pool.size=200