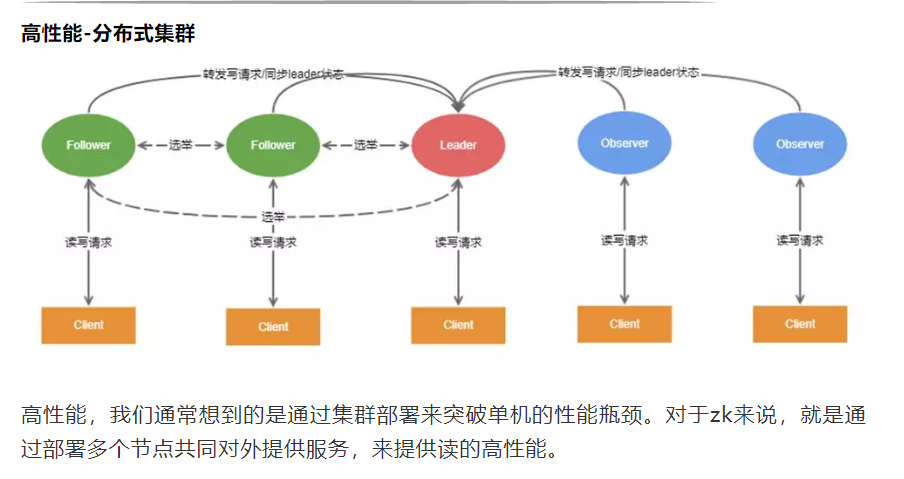
# Zookeeper基础

## 基本概念及原理

**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。

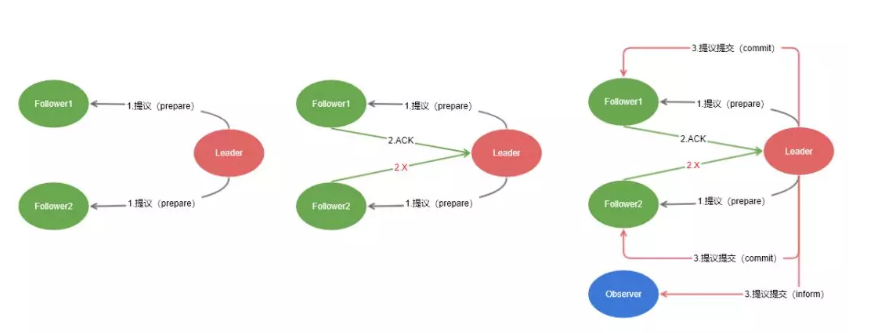
当Leader节点挂掉后，会从Follower节点中通过选举方式选出一个Leader提供对外服务。

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。

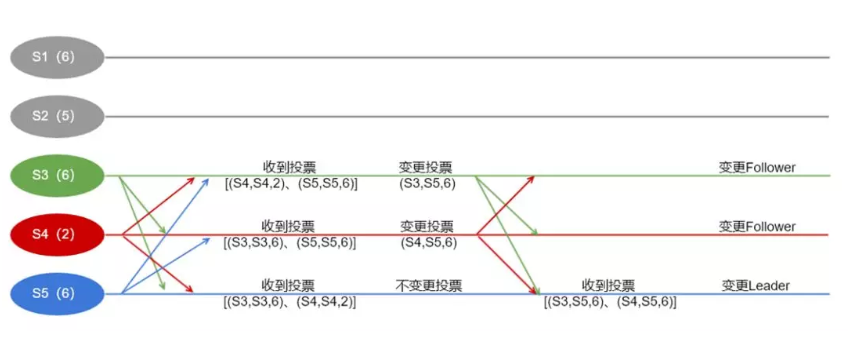
如果资金的选票不是最优选票，变更自己的选票，投最优选票的节点。

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

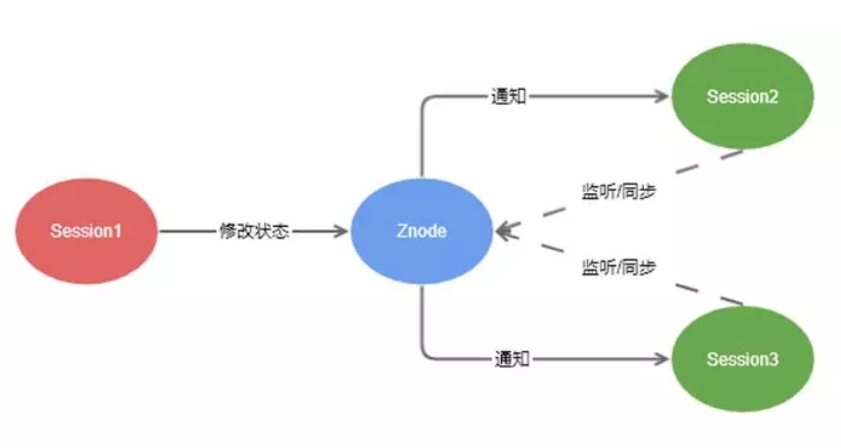
Zookeeper的用途包括统一配置管理、统一命名服务、分布式锁、集群管理。

**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**：

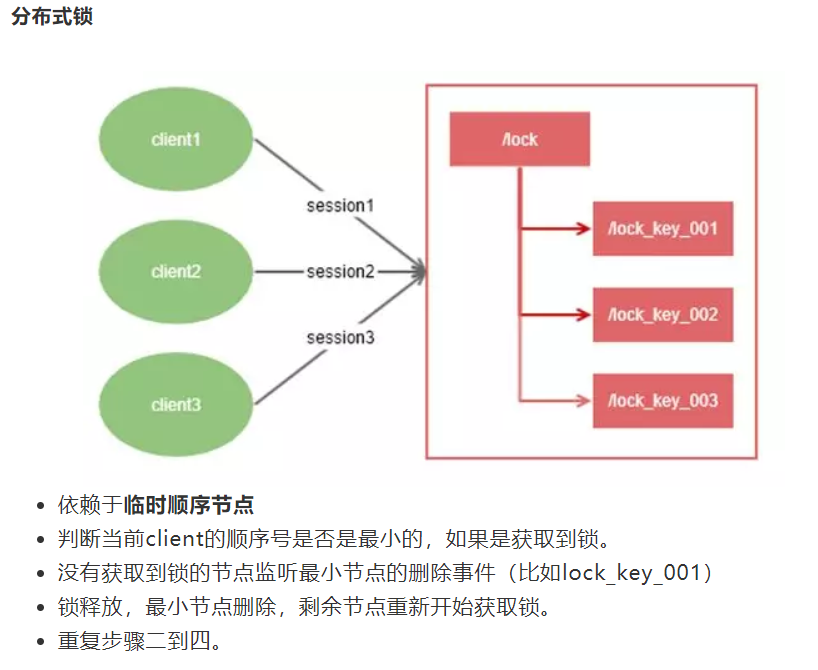
将几个服务的ip存于一个Znode下



**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)

## Zookeeper实现

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