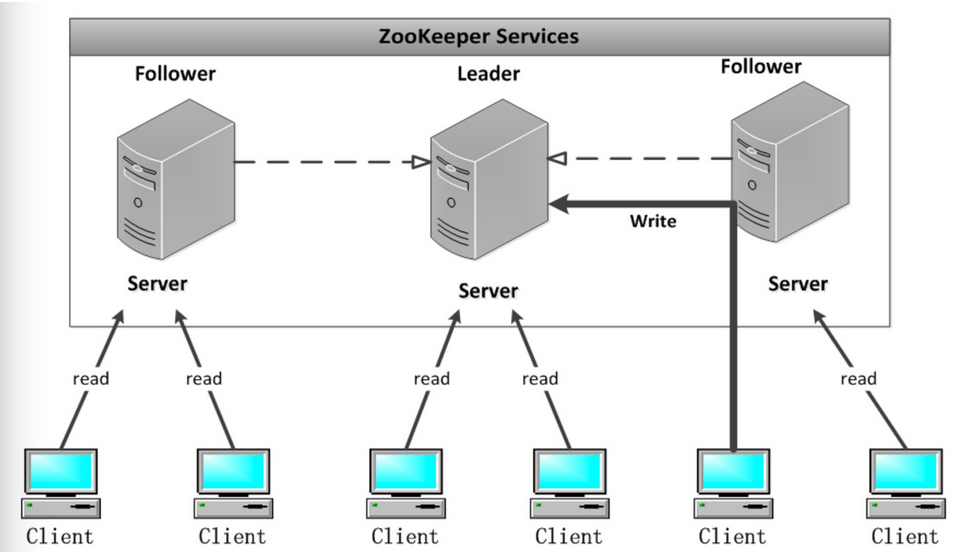
**ZooKeeper概述**

## 一、基础概述

**1 ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务**，是Google的Chubby一个开源的实现，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。

**提供了文件系统和通知机制。**



**超过1/2的zookeeper集群节点正常工作后，zookeeper就可以提供服务了**

**2 通知机制**

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

**3 Zookeeper功能**

**1.命名服务 2.配置管理 3.集群管理 4.分布式锁 5.队列管理**

**4 文件系统**

每个子目录项如 NameService 都被称作为znode，和文件系统一样，我们能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一的不同在于znode是可以存储数据的。

**@ 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

**@ PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点**

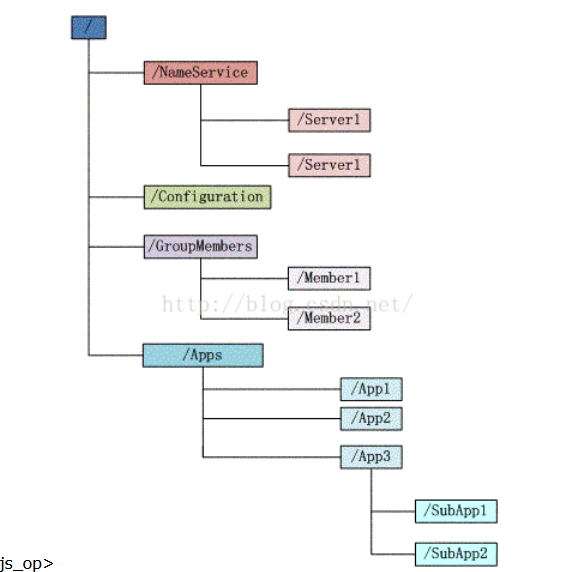
客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

**@ EPHEMERAL-临时目录节点**

客户端与zookeeper断开连接后，该节点被删除

**@ 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



**5 命名服务**

在zookeeper的文件系统里创建一个目录，即有唯一的path。在我们使用tborg无法确定上游程序的部署机器时即可与下游程序约定好path，通过path即能互相探索发现。

**6 会话**

会话就是一个客户端与服务器之间的一个TCP长连接。客户端和服务器的一切交互都是通过这个长连接进行的；会话会在客户端与服务器断开链接后，如果经过了设点的sessionTimeout时间内没有重新链接后失效。

**7 节点**

节点在ZeeKeeper中包含两层含义：

（1）集群中的一台机器，我们成为机器节点，比如领导者，跟随者

（2）ZooKeeper数据模型中的数据单元，我们成为数据节点（ZNode）。ZooKeeper的数据模型是内存中的一个ZNode数，由斜杠(/)进行分割的路径，就是一个ZNode，每个ZNode上除了保存自己的数据内容，还保存一系列属性信息；ZooKeeper中的数据节点分为两种：持久节点和临时节点。所谓的持久节点是指一旦这个ZNode创建成功，除非主动进行ZNode的移除操作，节点会一直保存在ZooKeeper上；而临时节点的生命周期是跟客户端的会话相关联的，一旦客户端会话失效，这个会话上的所有临时节点都会被自动移除。

**8 版本**

ZooKeeper为每一个ZNode节点维护一个叫做Stat的数据结构，在Stat中维护了节点相关的三个版本：

* dataVersion   
  数据版本号，每次对节点进行set操作，dataVersion的值都会增加1（即使设置的是相同的数据）。
* cversion   
  子节点的版本号。当znode的子节点有变化时，cversion 的值就会增加1。
* aclVersion   
  ACL的版本号，关于znode的ACL（Access Control List，访问控制），可以参考 参考资料1 有关ACL的描述。

**9 ACL（Access Control Lists）**

CREATE：创建子节点的权限

READ：获取节点数据和子节点列表的权限

WRITE：跟新节点数据的权限

DELETE：删除子节点的权限

ADMIN：设置节点ACL的权限。

**10 监听器Watcher**

ZooKeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候，ZooKeeper会通过事件通知到感兴趣的客户端上。

**11 ZooKeeper的数据模型**

上面有提到ZooKeeper的数据模型是一个ZNode节点树，是一个类型与标准文件系统的层次结构，也是使用斜杠(/)进行分割，在ZooKeeper中每一个节点都可以使用其路径唯一标识，如节点p\_1的标识为：/app1/p\_1每个ZNode节点都可以存储自己的数据，还可以拥有自己的子节点目录。

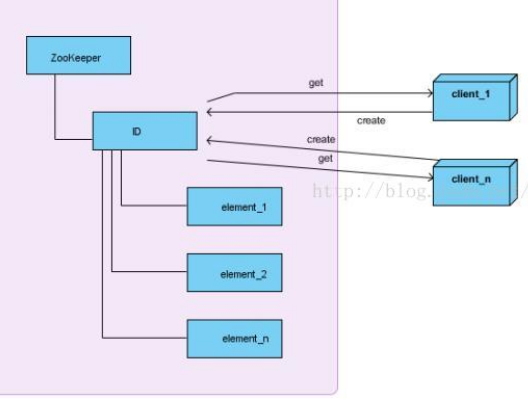
**ZooKeeper虽然提供了在节点存储数据的功能，但它并不将自己定位为一个通用的数据库，也就是说，你不应该在节点存储过多的数据。Zk规定节点的数据大小不能超过1M，但实际上我们在znode的数据量应该尽可能小，因为数据过大会导致zk的性能明显下降。如果确实需要存储大量的数据，一般解决方法是在另外的分布式数据库（例如redis）中保存这部分数据，然后在znode中我们只保留这个数据库中保存位置的索引即可。**

**ZooKeeper分布式命名服务**

**zookeeper的命名服务，有两个应用方向:**

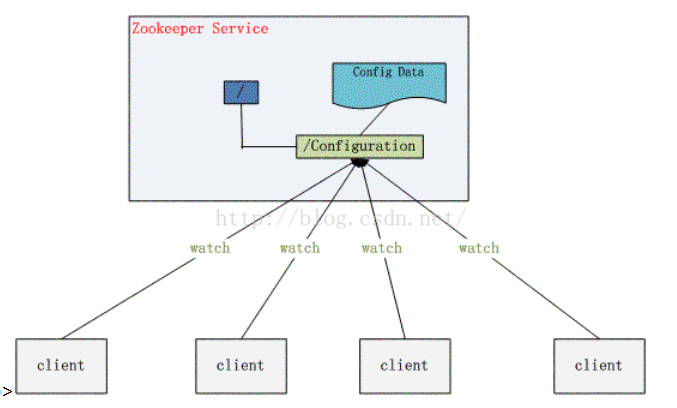
**1、提供类似JNDI的功能:利用zookeeper中的树形分层结构**，可以把系统中的各种服务的名称，地址以及目录信息存放在zookeeper中，需要的时候去zookeeper中读取

**2、利用zookeeper中的顺序节点的特性**，制作分布式的序列号生成器(ID生成器)，（在往数据库查询数据时，通常需要一个id，在单机环境下，可以利用数据库的自动成功id号，但是这种在分布式环境下就无法使用了，可以使用UUID，但是UUID有一个缺点，就是没有规律很难理解。使用zookeeper的命名服务可以生成有顺序的容易理解的，支持分布式的编号）



**ZooKeeper分布式配置管理**

**1 程序总是需要配置的**，如果程序分散部署在多台机器上，要逐个改变配置就变得困难。现在把这些配置全部放到zookeeper上去，保存在 Zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中就好



**ZooKeeper分布式集群管理**

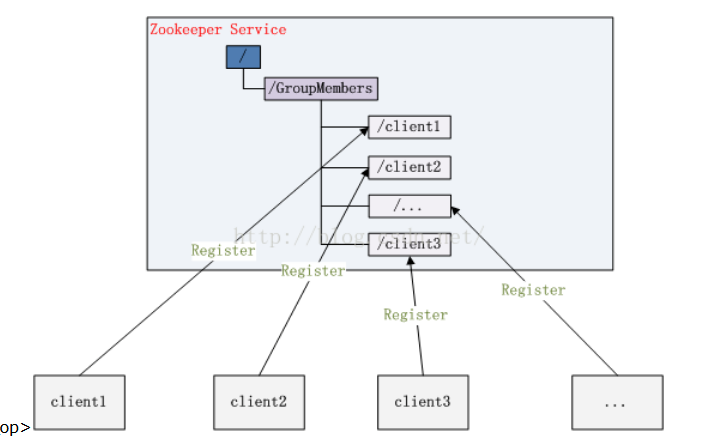
**1 所谓集群管理无在乎两点**：**是否有机器退出和加入、选举master。**

对于第一点，所有机器约定在父目录GroupMembers下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount又有了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，**每次选取编号最小的机器作为master**就好。

**2 它们的实现方式都是在 Zookeeper 上创建一个 EPHEMERAL 类型的目录节点**，然后每个 Server 在它们创建目录节点的父目录节点上调用 getChildren(String path, boolean watch) 方法并设置 watch 为 true，由于是 EPHEMERAL 目录节点，当创建它的 Server 死去，这个目录节点也随之被删除，所以 Children 将会变化，这时 getChildren上的 Watch 将会被调用，所以其它 Server 就知道已经有某台 Server 死去了。新增 Server 也是同样的原理。

**3 Zookeeper 如何实现 Leader Election**，也就是选出一个 Master Server。和前面的一样每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL\_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL\_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题。



**4 分布式与数据复制**

Zookeeper作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处：

1）、容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；

2）、提高系统的扩展能力 把负载分布到多个节点上，或者增加节点来提高系统的负载能力；

3）、提高性能：让客户端本地访问就近的节点，提高用户访问速度。

**5 从客户端读写访问的透明度来看，数据复制集群系统分下面两种：**

**1、写主(WriteMaster)** ：对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离；

**2、写任意(Write Any)：**对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对zookeeper来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立observer的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

**6 Zookeeper设计目的**

**@.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。

**@.可靠性：**具有简单、健壮、良好的性能，如果消息被到一台服务器接受，那么它将被所有的服务器接受。

**@.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。

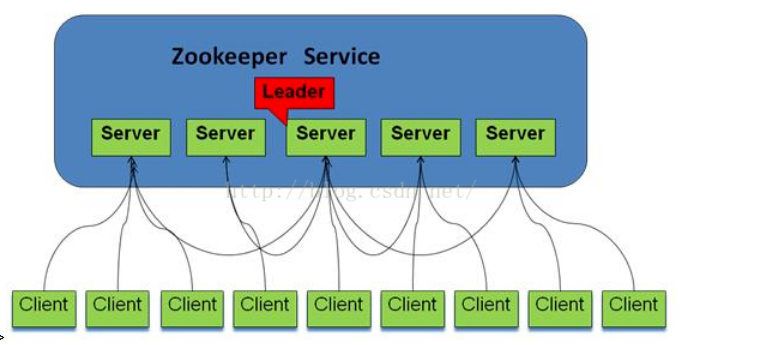
**@.等待无关（wait-free）**：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

**@.原子性：**更新只能成功或者失败，没有中间**状**态。

**@.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。

**ZooKeeper角色**





**ZooKeeper工作原理**

**1 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）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，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。

**2 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。

**3 Server工作状态**

每个Server在工作过程中有三种状态：

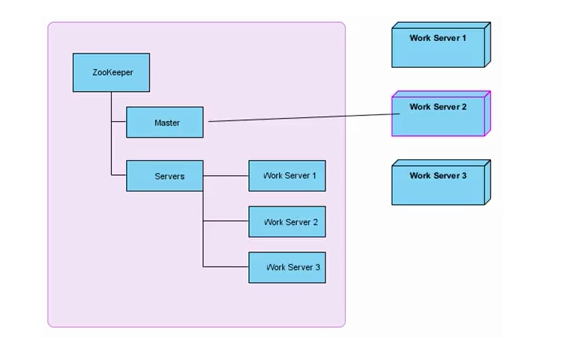
LOOKING：当前Server不知道leader是谁，正在搜寻

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

**ZooKeeper选主流程**

**1 考虑7\*24小时向外提供服务的系统**，不能有单点故障，于是我们使用集群，采用的是Master+Slave。集群中有一台主机和多台备机，由主机向外提供服务，备机监听主机状态，一旦主机宕机，备机必需迅速接管主机继续向外提供服务。在这个过程中，从备机选出一台机作为主机的过程，就是Master选举。



**2 左边是ZooKeeper集群，右边是3台工作服务器。**工作服务器启动时，**会去ZooKeeper的Servers节点下创建临时节点，并把基本信息写入临时节点。**这个过程叫服务注册，系统中的其他服务可以通过获取Servers节点的子节点列表，来了解当前系统哪些服务器可用，这该过程叫做服务发现。接着这些服务器会尝试创建Master临时节点，谁创建成功谁就是Master，其他的两台就作为Slave。所有的Work Server必需关注Master节点的删除事件。通过监听Master节点的删除事件，来了解Master服务器是否宕机（创建临时节点的服务器一旦宕机，它所创建的临时节点即会自动删除）。一旦Master服务器宕机，必需开始新一轮的Master选举。

**3 当leader崩溃或者leader失去大多数的follower**，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。**

**4 (basic paxos)**

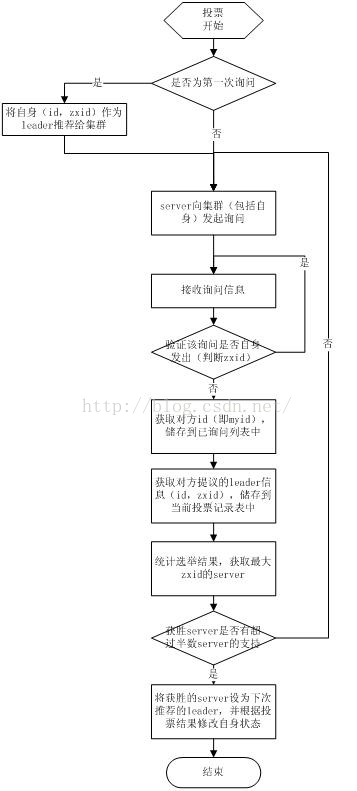
1）选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

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；

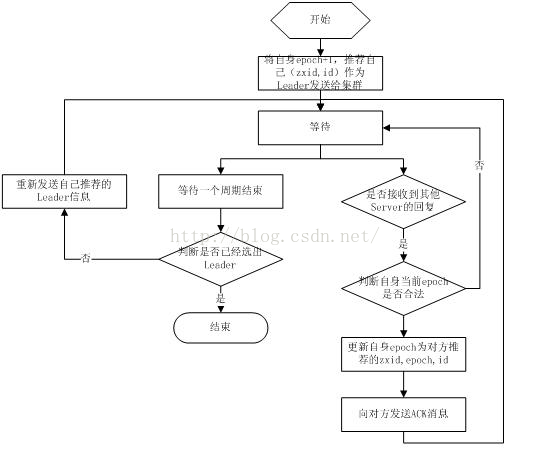
4）收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

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。选主的具体流程图所示：



**5 (fast paxos)**

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。



**6 同步流程**

选完Leader以后，zk就进入状态同步过程。

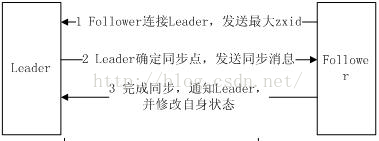
1. Leader等待server连接；

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。



**ZooKeeper工作流程**

**1 工作流程-Leader**

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；

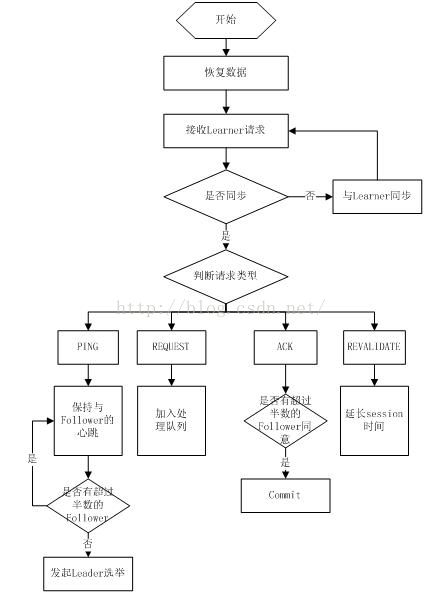
3） .Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。

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；

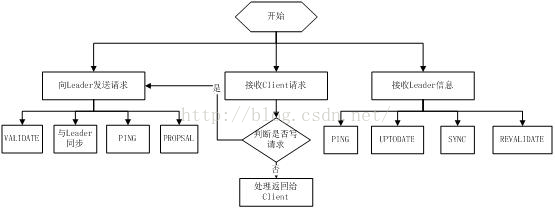
ACK消息是 Follower的对提议的回复，超过半数的Follower通过，则commit该提议；

REVALIDATE消息是用来延长SESSION有效时间。



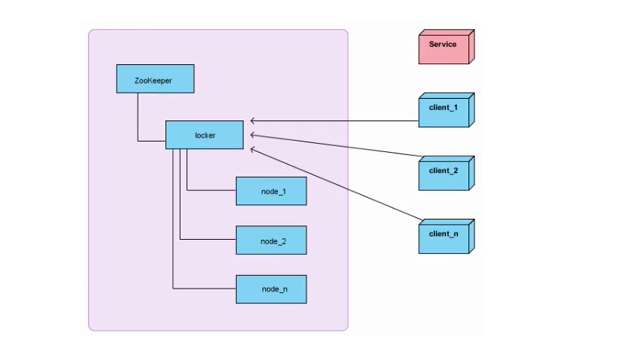
**2 工作流程-Follower**

**（1）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。   
  
**（2）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。



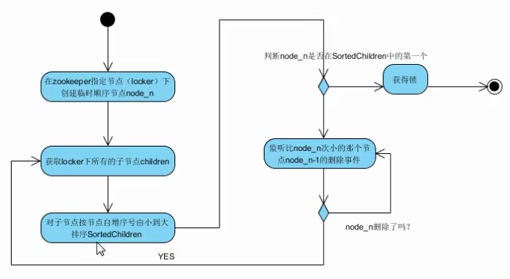
**ZooKeeper分布式锁**

1 我们常说的锁是单进程多线程锁，在多线程并发编程中，用于线程之间的数据同步，保护共享资源的访问。而分布式锁，指在分布式环境下，保护跨进程、跨主机、跨网络的共享资源，实现互斥访问，保证一致性。



2 左侧是zookeeper集群，locker是数据节点，node\_1到node\_n代表一系列的顺序节点。

右侧client\_1至client\_n代表客户端，Service代表需要互斥访问的服务。总实现思路，是在获取锁的时候在locker节点下创建顺序节点，在释放锁的时候，把自己创建的节点删除。



**ZooKeeper分布式队列**

**1 在传统的单进程编程中，我们使用队列来存储一些数据结构，**用来在多线程之间共享或传递数据。分布式环境下，我们同样需要一个类似单进程队列的组件，用来实现跨进程、跨主机、跨网络的数据共享和数据传递，这就是我们的分布式队列。

**2 主要的思路如下：**

（1）首先DistributedQueue 的构造器中要传入

ZooKeeper：连接服务端的核心类；

dir：表示znode组的结点

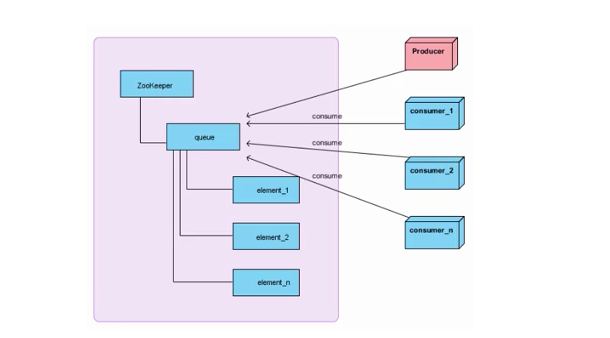
acl：每个结点的访问控制列表

（2）所谓的入队，也就是在dir组下新建一个znode结点，不过结点类型是CreateMode.PERSISTENT\_SEQUENTIAL，这就表明了新建结点的值根据序号是有一定顺序性的。其核心实现就是：

zookeeper.create(dir+"/"+prefix, data, acl, CreateMode.PERSISTENT\_SEQUENTIAL);

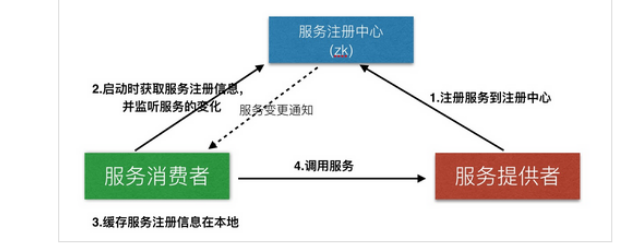
（3）所谓的出队，其实就是取出所有的子结点，然后用treeMap维护有序性，然后将最小的结点出队(也就是在服务端删除结点最小的结点)

（4）该分布式队列的实现严重依赖于zookeeper的一致性保证。



**图中左侧代表zookeeper集群，右侧代表消费者和生产者。生产者通过在queue节点下创建顺序节点来存放数据，消费者通过读取顺序节点来消费数据。**

**ZooKeeper服务发现和注册**



**1 简单说明**

1）. 它提供的简单API

2）. 已有互联网公司(例如：Pinterest，Airbnb)使用它来进行服务注册与发现

3）. 支持多语言的客户端

4）. 通过Watcher机制实现Push模型，服务注册信息的变更能够及时通知服务消费方

**2 服务提供者**

服务提供者作为服务的提供方将自身的服务信息注册到服务注册中心中。服务信息包含：

▪ 隶属于哪个系统

▪ 服务的IP，端口

▪ 服务的请求URL

▪ 服务的权重等等

**3 服务注册中心**

服务注册中心主要提供所有服务注册信息的中心存储，同时负责将服务注册信息的更新通知实时的Push给服务消费者(主要是通过Zookeeper的Watcher机制来实现的)。

**4 服务消费者**

服务消费者主要职责如下：

1）. 服务消费者在启动时从服务注册中心获取需要的服务注册信息

2）. 将服务注册信息缓存在本地

3）. 监听服务注册信息的变更，如接收到服务注册中心的服务变更通知，则在本地缓存中更新服务的注册信息

4）. 根据本地缓存中的服务注册信息构建服务调用请求，并根据负载均衡策略(随机负载均衡，Round-Robin负载均衡等)来转发请求

5）. 对服务提供方的存活进行检测，如果出现服务不可用的服务提供方，将从本地缓存中剔