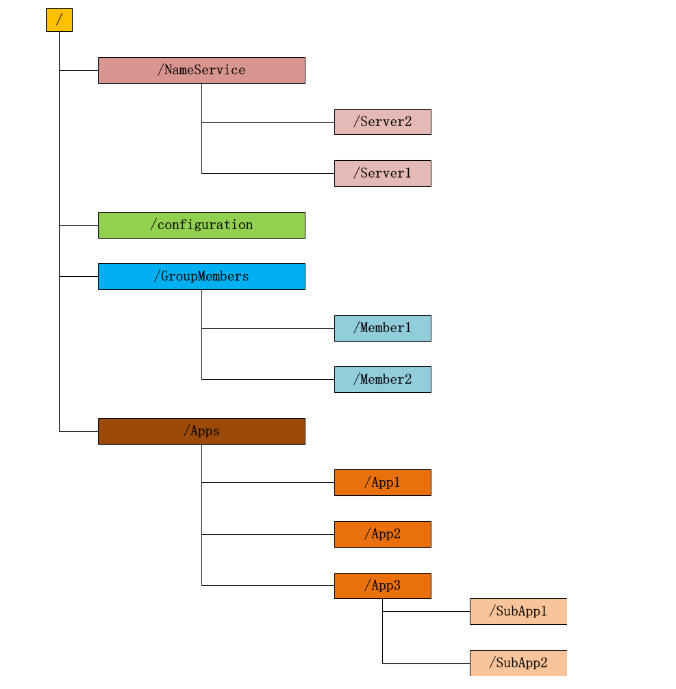
# [ZooKeeper管理分布式环境中的数据](https://www.cnblogs.com/sunddenly/p/4092654.html)

## ZooKeeper应用场景

ZooKeeper是一个高可用的分布式数据管理与系统协调框架。基于对Paxos算法的实现，使该框架保证了分布式环境中数据的强一致性，也正是基于这样的特性，使得zookeeper能够应用于很多场景。需要注意的是，ZK并不是生来就为这些场景设计，都是后来众多开发者根据框架的特性，摸索出来的典型使用方法。因此，我们也可以根据自己的需要来设计相应的场景实现。正如前文所提到的，ZooKeeper 实现的任何功能都离不开ZooKeeper的数据结构，任何功能的实现都是利用"Znode结构特性+节点关联的数据"来实现的，好吧那么我们就看一下ZooKeeper数据结构有哪些特性。ZooKeeper数据结构如下图所示：





### 数据发布与订阅

**(1) 典型场景描述**

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。集中式的配置管理在应用集群中是非常常见的，一般商业公司内部都会实现一套集中的配置管理中心，应对不同的应用集群对于共享各自配置的需求，并且在配置变更时能够通知到集群中的每一个机器。

**(2) 应用**

**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。

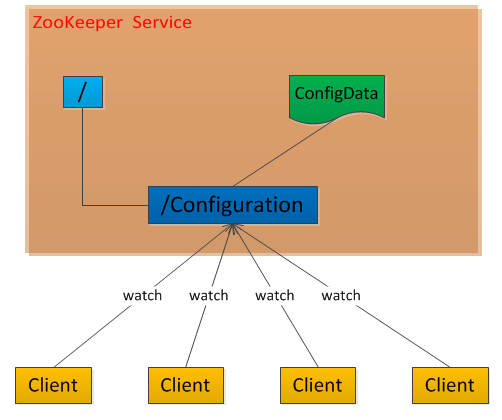
**③** 应用中用到的一些配置信息集中管理，在应用启动的时候主动来获取一次，并且在节点上注册一个Watcher，以后每次配置有更新，实时通知到应用，获取最新配置信息。

**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。

**(3) 应用举例**

**例如**：同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。ZooKeeper配置管理服务如下图所示



### 统一命名服务（Name Service）

**(1) 场景描述**

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形的名称结构是一个有层次的目录结构，既对人友好又不会重复。说到这里你可能想到了 JNDI，没错 Zookeeper 的 Name Service 与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上，但是Zookeeper的Name Service 更加是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，你可能只需要一个不会重复名称，就像数据库中产生一个唯一的数字主键一样。

**(2) 应用**

在分布式系统中，通过使用命名服务，客户端应用能够根据指定的名字来获取资源服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，进程对象等等，这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用ZK提供的创建节点的API，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。Name Service 已经是Zookeeper 内置的功能，你只要调用 Zookeeper 的 API 就能实现。如调用 create 接口就可以很容易创建一个目录节点。

**(3) 应用举例**

阿里开源的分布式服务框架Dubbo中使用ZooKeeper来作为其命名服务，维护全局的服务地址列表。在Dubbo实现中：**服务提供者**在启动的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。 **服务消费者**启动的时候，订阅/dubbo/${serviceName}/providers目录下的提供者URL地址， 并向/dubbo/${serviceName} /consumers目录下写入自己的URL地址。 注意，所有向ZK上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。 另外，Dubbo还有针对服务粒度的监控，方法是订阅/dubbo/${serviceName}目录下所有提供者和消费者的信息。

### 分布通知/协调（Distribution of notification/coordination）

**(1) 典型场景描述**

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。

**(2) 应用**

**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**：某系统由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

### 分布式锁（Distribute Lock）

**(1) 场景描述**

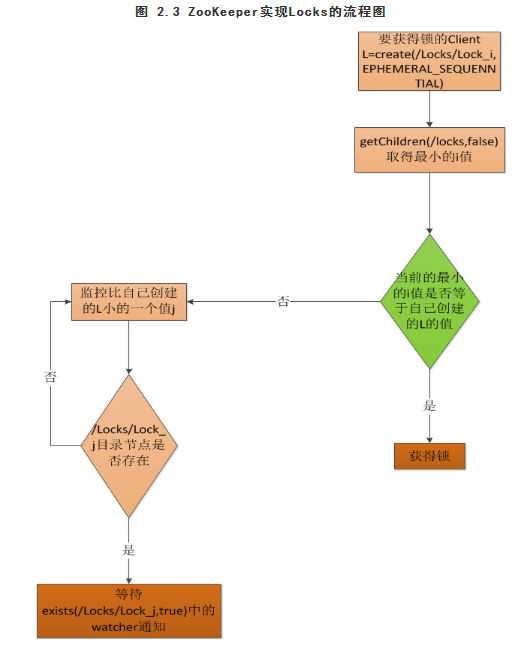
**分布式锁**，这个主要得益于ZooKeeper为我们保证了**数据的强一致性**，即用户只要完全相信每时每刻，zk集群中任意节点（一个zk server）上的相同znode的数据是一定是相同的。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把ZK上的一个znode看作是一把锁，通过create znode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。

控制时序，就是所有试图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distribute\_lock 已经预先存在，客户端在它下面创建临时有序节点。Zk的父节点（/distribute\_lock）维持一份sequence,保证子节点创建的时序性，从而也形成了每个客户端的全局时序。

**(2) 应用**

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL\_SEQUENTIAL 目录节点，然后调用 getChildren方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是那么它就调用 exists(String path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。



### 集群管理（Cluster Management）

**(1) 典型场景描述**

**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：

**①**集群中机器有变动的时候，牵连修改的东西比较多。

**②** 有一定的延时。

利用ZooKeeper中两个特性，就可以实施另一种集群机器存活性监控系统：

**①**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。

**②**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。

**Master选举：**

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，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中两个特性，就可以实施另一种集群中Master选举：

**①**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 /Master 节点，最终一定只有一个客户端请求能够创建成功。利用这个特性，就能很轻易的在分布式环境中进行集群选举了。

**②**另外，这种场景演化一下，就是动态Master选举。这就要用到 EPHEMERAL\_SEQUENTIAL类型节点的特性了，这样每个节点会自动被编号。允许所有请求都能够创建成功，但是得有个创建顺序，每次选取序列号最小的那个机器作为Master 。

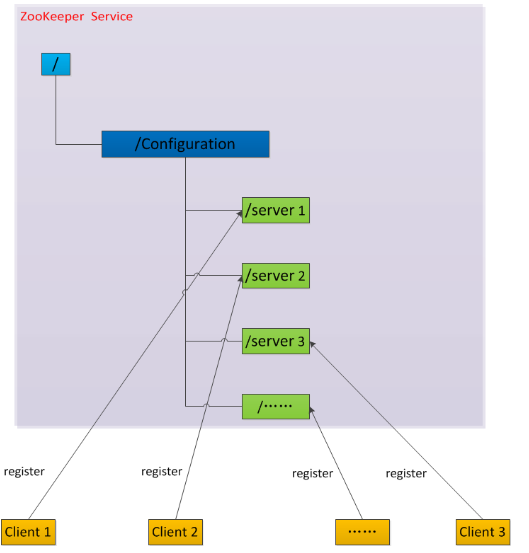
**(2) 应用**

在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此间索引数据一致。因此让集群中的Master来进行全量索引的生成，然后同步到集群中其它机器。另外，Master选择的容灾措施，可以随时进行手动决定master，就是说应用在zk在无法获取master信息时，可以通过比如http方式，向一个地方获取master。 在Hbase中，也是使用ZooKeeper来实现动态HMaster的选举。在Hbase实现中，会在ZK上存储一些ROOT表的地址和HMaster的地址，HRegionServer也会把自己以临时节点（Ephemeral）的方式注册到Zookeeper中，使得HMaster可以随时感知到各个HRegionServer的存活状态，同时，一旦HMaster出现问题，会重新选丼出一个HMaster来运行，从而避免了HMaster的单点问题的存活状态。

**(3) 应用举例**

**集群监控：**

应用集群中，我们常常需要让每一个机器知道集群中或依赖的其他某一个集群中哪些机器是活着的，并且在集群机器因为宕机，网络断链等原因能够不在人工介入的情况下迅速通知到每一个机器，Zookeeper 能够很容易的实现集群管理的功能，如有多台 Server 组成一个服务集群，那么必须要一个"总管"知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它集群必须知道，从而做出调整重新分配服务策略。同样当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让"总管"知道，这就是ZooKeeper的集群监控功能。



比如我在zookeeper服务器端有一个znode叫**/Configuration**，那么集群中每一个机器启动的时候都去这个节点下创建一个EPHEMERAL类型的节点，比如server1创建**/Configuration** /Server1，server2创建**/Configuration** /Server1，然后Server1和Server2都watch /Configuration 这个父节点，那么也就是这个父节点下数据或者子节点变化都会通知对该节点进行watch的客户端。因为EPHEMERAL类型节点有一个很重要的特性，就是客户端和服务器端连接断掉或者session过期就会使节点消失，那么在某一个机器挂掉或者断链的时候，其对应的节点就会消 失，然后集群中所有对**/Configuration**进行watch的客户端都会收到通知，然后取得最新列表即可。

**Master选举：**

Zookeeper 不仅能够维护当前的集群中机器的服务状态，而且能够选出一个"总管"，让这个总管来管理集群，这就是 Zookeeper 的另一个功能 Leader Election。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**。**

### 队列管理

Zookeeper 可以处理两种类型的队列：

**①** 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。

**②**队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

**(1)**同步队列用 Zookeeper 实现的实现思路如下：

创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建 /synchronizing/member\_i 的临时目录节点，然后每个成员获取 / synchronizing 目录的所有目录节点，也就是 member\_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

用下面的流程图更容易理解：

