Zookeeper

1 什么是 zookeeper

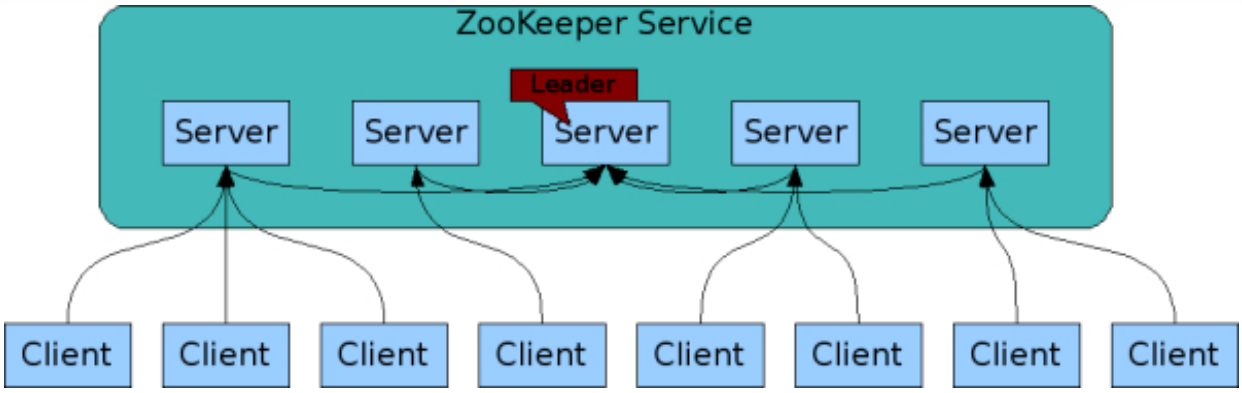
> Zookeeper 是一个高效的分布式协调服务,它只适合存取一些不大的数据,不适合存储大量数据,一般都是存取一些发布的信息,注册信息等等,它暴露一些公共的服务,比如命名/配置/管理/同步控制/群组服务等,我们可以使用 zookeeper 来实现比如打成共识/集群管理/ leader选举等操作
>
> Zookeeper 是一个高可用的分布式管理与协调框架,基于 ZAB算法(原子消息广播协议paxos(复制算法,主从选举算法))实现,能够很好的保证分布式环境中的数据一致性(不推荐单机使用,官方推荐奇数个节点),也正是基于这样的特性,使它成为解决分布式一致性问题的利器,可以用于分布式锁等操作(优于 redis)
>
> 顺序一致性:从一个客户端发起的事务请求,会被严格的按照其发起的顺序被应用到zookeeper 中去
>
> 原子性:所有事务的请求处理结果在集群的所有机器上应用情况是一致的,也就是说要么整个集群的所有集群全部都成功应用一个事务,要么没有应用到,一定不会出现部分应用,部分没有
>
> 单一视图:无论客户端连接的是集群中的哪一个服务器,看到的结果都是一样的
>
> 可靠性:一旦服务器成功应用一个事务,并且对客户端响应,那么该事务引起服务器端的状态会被一致的保留下来,除非另外一个事务做出更改
>
> 实时性:事务一旦被应用,客户端就可以从服务端获取到最新的状态,zookeeper 仅仅能保证在一段时间内,客户端最终能一定从服务端获取到最新的数据状态
>
> 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

2设计目的

- **简单**：Zookeeper允许程序通过一个共享的类似于标准文件系统的有组织的分层命名空间分布式处理协调。命名空间包括：数据寄存器 - 在Zookeeper中叫znodes, 它和文件、目录一样。和一般文件系统的不同之处是，它的设计就是为了存储，Zookeeper的数据保持在内存中，这就意味着它可以实现高吞吐量和低延迟的数据。

Zookeeper的实现提供了一个优质的高性能、高可用，严格的访问顺序。Zookeeper的性能方面意味着它可以用于大型的分布式系统。可靠性方面防止它成为一个单 点故障。严格的顺序意味着可以在客户端实现复杂的同步原件。

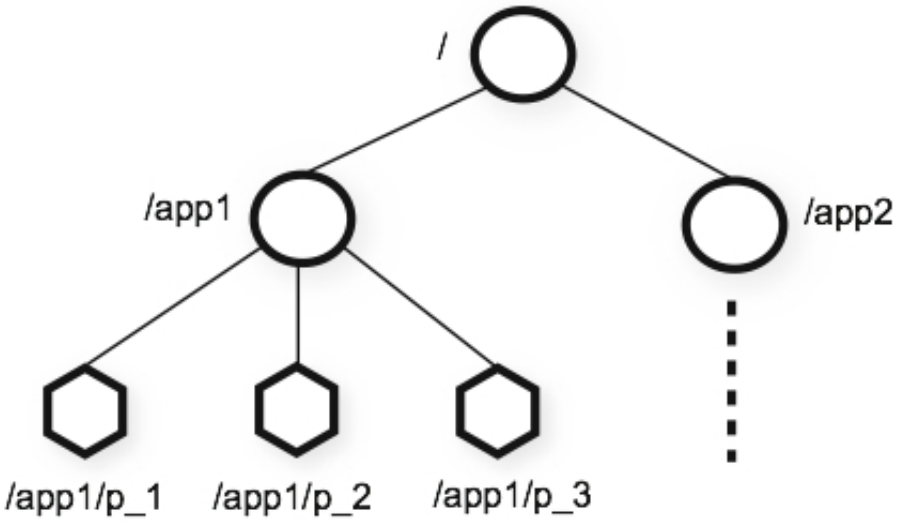
- **复制**：像分布式处理一样，Zookeeper自己在处理协调的时候要复制多个主机，Zookeeper服务的组成部分必须彼此都知道彼此，它们维持了一个内存状态影像，连同事务日志和快照在一个持久化的存储中。 只要大多数的服务器是可用的，Zookeeper服务就是可用的。客户端连接到一个单独的服务。客户端保持了一个TCP连接，通过这个TCP连接发送请求、获取响应、获取watch事件、和发送心跳。如果这个连接断了，会自动连接到其他不同的服务器。

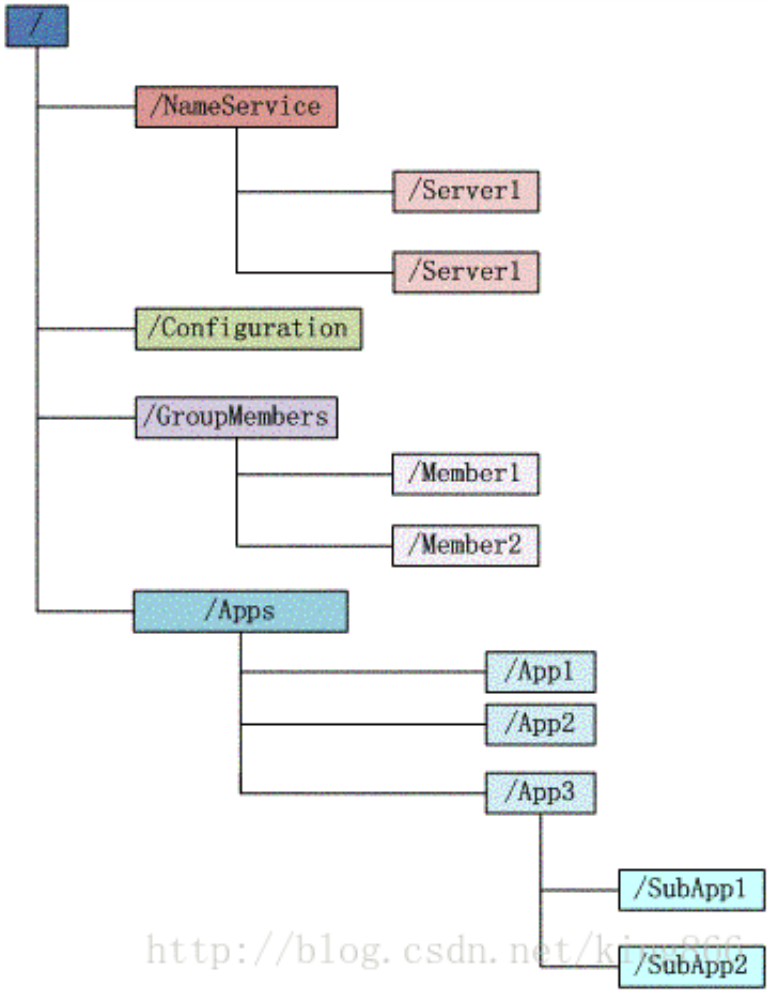


- **序列**：Zookeeper用数字标记每一个更新，用它来反射出所有的事务顺序。随后的操作可以使用这个顺序去实现更高级的抽象，例如同步原件。
- **快速**：它在"Read-dominant"工作负载中特别快。Zookeeper 应用运行在数以千计的机器上，数据保存在内存中, 并且它通常在读比写多的时候运行的最好，读写大概比例在10:1。

3 数据模型和分层的命名空间

Zookeeper提供的命名空间非常像一个标准的文件系统。一个名字是一系列的以'/'隔开的一个路径元素。Zookeeper命名空间中所有的节点都是通过路径识别。





4节点和临时节点

不像标准的文件系统，Zookeeper命名空间中的每个节点可以有数据也可以有子目录。它就像一个既可以是文件也可以是目录的文件系统。(Zookeeper被设计成保存协调数据：状态信息，配置，位置信息，等等，所以每个节点存储的数据通常较小，通常在1个字节到数千字节的范围之内)我们使用术语znode来表明Zookeeper的数据节点。

znode维持了一个stat结构，它包括数据变化的版本号、访问控制列表变化、和时间戳，允许缓存验证和协调更新。每当znode的数据有变化，版本号就会增加。例如，每当客户端检索数据时同时它也获取数据的版本信息。

命名空间中每个znode存储的数据自动的读取和写入的，读取时获得znode所有关联的数据字节，写入时替换所有的数据。每个节点都有一个访问控制列表来制约谁可以做什么。

Zookeeper还有一个临时节点的概念。这些znode和session存活的一样长，session创建时存活，当session结束，也跟着删除。

5条件的更新和watches

Zookeeper支持watches的概念。客户端可以在znode上设置一个watch。当znode发生变化时触发并移除watch。当watch被触发时，客户端会接收到一个包说明znode有变化了。并且如果客户端和其中一台server中间的连接坏掉了，客户端就会收到一个本地通知。这些可以用来[tbd]。

6Zookeeper 组成

ZK Server 根据其身份特性分为三种 Leader,Follower,Observer,其中Follower,Observer 成为 learner 学习者

Leader 负责客户端的 Writer 操作

Follower 负责客户端的 Read操作,并参与选举

Observer是一个特殊的 Follower,它可以接收客户端的read请求,但是不参与选举,扩容系统支撑,提高读取速度,但是它不接受写请求,只从 leader 上面同步数据,我们的 wathcher 就是一个Observer

7ZooKeeper典型使用场景一览

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。**数据发布与订阅** 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。 \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。 **分布通知/协调** 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。 \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：某系统有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 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**，就是所有视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（这个可以通过节点的属性控制：CreateMode.EPHEMERAL_SEQUENTIAL来指定）。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。 **集群管理** \1. 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有两个特性，就可以实时另一种集群机器存活性监控系统：a. 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。b. 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。

\2. 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处理），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 /currentMaster 节点，最终一定只有一个客户端请求能够创建成功。

配置管理

配置的管理在分布式中经常见,如机器的配置列表,运行时的开关配置,数据库配置信息等,他们一般符合以下三个特性

1. 数据量比较小
2. 数据内容在运行时是动态变化的
3. 集群中各个节点共享信息, 配置一致

8保证

Zookeeper是非常简单和高效的。因为它的目标就是，作为建设复杂服务的基础，比如同步。zookeeper提供了一套保证，他们包括：

- 顺序一致性 - 来自客户端的更新会按顺序应用。
- 原子性 - 更新成功或者失败，没有局部的结果产生。
- 唯一系统映像 - 一个客户端不管连接到哪个服务端都会看到同样的视图。
- 可靠性- 一旦一个更新被应用，它将从更新的时间开始一直保持到一个客户端重写更新。
- 时效性 - 系统中的客户端视图在特定的时间点保证成为是最新的。

9简单API

Zookeeper的设计目标的其中之一就是提供一个简单的程序接口。因此，它只支持这些操作：

- create - 在树形结构的位置中创建节点,并可以指定节点内容 例如 create /abc hello
- delete - 删除一个节点
- rmr -递归删除节点
- exists - 测试节点在指定位置上是否存在
- get data - 从节点上读取数据
- set data - 往节点写入输入
- get chilren - 检索一个节点的子节点列表

- sync - 等待传输数据

> 异步调用 zookeeper api 时候
>
> rc 代表响应吗 0是成功 -4代表连接,-110代表指定节点存在,-112 表述会话已经过期
>
> path 代表调用接口时候传入的数据节点路径参数
>
> ctx 代表调用接口 api 传入的 ctx,可以随便传
>
> name 实际在服务器端节点创建的名称

10 Watcher

> zookeeper 有 watch 事件,是一次性触发的,当 watch 监视的数据发生变化时候,通过该 watch 的客户端,也就是 watcher
>
> watcher 监听数据变化,那就会一些产生事件类型和状态
>
> 事件类型 和 znode 节点相关
>
> EventType.NodeCreated 节点创建
>
> NodeDateChanged 节点数据变化
>
> NodeChildrenChanged 子节点变化
>
> NodeDeleted 节点删除
>
> 状态类型,是和客户端相关的
>
> KeeperState.Disconnected
>
> SyncConnected
>
> AuthFailed
>
> Expired