## Zookeeper是什么

Zookeeper 作为一个**分布式的服务框架**，主要用来解决分布式集群中应用系统的一致性问题，它能提供基于类似于文件系统的目录节点树方式的数据存储，但是 Zookeeper 并不是用来专门存储数据的，它的作用主要是用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理。

zookeeper可以充当一个***服务注册表***（Service Registry），让多个服务提供者形成一个集群，让服务消费者通过服务注册表获取具体的服务访问地址（ip+端口）去访问具体的服务提供者。

简单的说**，zookeeper=文件系统+通知机制。**

## 启动流程

第一张图

1. 服务端启动

bin/zkServer.sh start

setlocal

call "%~dp0zkEnv.cmd"

set ZOOMAIN=org.apache.zookeeper.server.quorum.QuorumPeerMain

可知QuorumPeerMain为启动的入口 但是我们这边需要设置启动文件zoo.cfg,因为是源码 我直接写到代码里去了

这一步server端开始启动

2. 解析zoo.cfg的信息

3. 这一步会开启一个定时任务 不过具体干啥的没去深究

4. 开始进行服务端的初始 这一步会进行判断 判断是否为集群模式下启动 我们这里先分析单机启动

5. 调用单机模式下服务端的初始化方法

6. 解析配置文件zoo.cfg的文件内容

7.根据配置文件的内容 进行服务端的启动

8.服务端参数配置初始化

9.创建连接服务端的处理工厂 可以在配配置文件中通过 zookeeper.serverCnxnFactor指定 默认使用 NIOServerCnxnFactory

接下的分析 我们以NIOServerCnxnFactory进行展开

10. 连接工厂参数配置

11. 工厂（NIOServerCnxnFactory）启

## 设计目的

    1.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。

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

    3 .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。

    4 .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。

    5.原子性：更新只能成功或者失败，没有中间状态。

    6 .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的数据模型

Zookeeper 会维护一个具有层次关系的数据结构，它非常类似于一个标准的文件系统，每个节点都叫数据节点（znode），节点上可以存储数据。

**数据结构的特点：**

每个子目录项如 NameService 都被称作为 znode，这个 znode 是被它所在的路径唯一标识，如 Server1 这个 znode 的标识为 /NameService/Server1

znode 可以有子节点目录，并且每个 znode 可以存储数据，注意 EPHEMERAL 类型的目录节点不能有子节点目录

znode 是有版本的，每个 znode 中存储的数据可以有多个版本，也就是一个访问路径中可以存储多份数据

znode 可以是临时节点，一旦创建这个 znode 的客户端与服务器失去联系，这个 znode 也将自动删除，Zookeeper 的客户端和服务器通信采用长连接方式，每个客户端和服务器通过心跳来保持连接，这个连接状态称为 session，如果 znode 是临时节点，这个 session 失效，znode 也就删除了

znode 的目录名可以自动编号，如 App1 已经存在，再创建的话，将会自动命名为 App2

znode 可以被监控，包括这个目录节点中存储的数据的修改，子节点目录的变化等，一旦变化可以通知设置监控的客户端，这个是 Zookeeper 的核心特性，Zookeeper 的很多功能都是基于这个特性实现的。

**节点的类型：**

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

|  |  |  |
| --- | --- | --- |
| 场景类别 | 典型场景描述(ZK特性，使用方法) | 应用中的具体使用 |
| 数据发布与订阅 | 发布与订阅即所谓的配置管理，顾名思义就是有系统将数据发布到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。(Diamond和ConfigServer在这方面也具备的功能) | 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，供各个客户端使      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。      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。 |
| 统一命名服务（Name Service） | 这个主要是作为分布式命名服务，能过调用zk的create 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。 |  |
| 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)。      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为我们保证了强一致性，即用户只要完全信任每时每刻，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。**      这个我感觉可以分为两类，一个是保持独占，另一个是控制时序。      所谓保持独占，就是所有视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)。Zk的父节点(/distribute\_lock)维持一份sequence,保证子节点创建的时序性，从而也保证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响应。这样的场景中，通常有一个监控系统，实时检测集群机器是否存活。      通常的做法是：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主机汇报“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，牵连到的东西比较多。2. 延时。      ZooKeeper有两个特性：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。      例如，监控系统在 /clusterServers 节点上注册一个Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。       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，相同的应用可能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)，往往让整个集群中一台机器进行，其余机器可以分享这个结果，这样可以大大减少重复劳动，于是这个master选举便是这种场景下的主要问题。  **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。      利用这个特性，就能很轻易的在分布式环境中进行集群选取了。      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。      上文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终在ZK上创建结果的一种可能情况是这样： /currentMaster/{sessionId}-1 , /currentMaster/{sessionId}-2 , /currentMaster/{sessionId}-3 ….. 每次选取序列号最小的那个机器作为Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是Master了。 | 1. 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，而且不能保证彼此之间索引数据一致。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。      2. 另外，Master选举的容灾措施是，可以随时进行手动指定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。 |
| 分布式队列 | 队列方面，我目前感觉有两种，一种是常规的先进先出队列，另一种是要等到队列成员聚齐之后的才统一按序执行。      对于第二种先进先出队列，和分布式锁服务中的控制时序场景基本原理一致，这里不再赘述。  第二种队列其实是在FIFO队列的基础上作了一个增强。通常可以在 /queue 这个znode下预先建立一个/queue/num 节点，并且赋值为n(或者直接给/queue赋值n)，表示队列大小，之后每次有队列成员加入后，就判断下是否已经到达队列大小，决定是否可以开始执行了。 | 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。 |