【分布式】Zookeeper应用场景

一、前言

在上一篇博客已经介绍了Zookeeper开源客户端的简单实用，本篇讲解Zookeeper的应用场景。

二、典型应用场景

Zookeeper是一个高可用的分布式数据管理和协调框架，并且能够很好的保证分布式环境中数据的一致性。在越来越多的分布式系统（Hadoop、HBase、Kafka）中，Zookeeper都作为核心组件使用。

2.1 数据发布/订阅

数据发布/订阅系统，即配置中心。需要发布者将数据发布到Zookeeper的节点上，供订阅者进行数据订阅，进而达到动态获取数据的目的，实现配置信息的集中式管理和数据的动态更新。发布/订阅一般有两种设计模式：推模式和拉模式，服务端主动将数据更新发送给所有订阅的客户端称为推模式；客户端主动请求获取最新数据称为拉模式，Zookeeper采用了推拉相结合的模式，客户端向服务端注册自己需要关注的节点，一旦该节点数据发生变更，那么服务端就会向相应的客户端推送Watcher事件通知，客户端接收到此通知后，主动到服务端获取最新的数据。

若将配置信息存放到Zookeeper上进行集中管理，在通常情况下，应用在启动时会主动到Zookeeper服务端上进行一次配置信息的获取，同时，在指定节点上注册一个Watcher监听，这样在配置信息发生变更，服务端都会实时通知所有订阅的客户端，从而达到实时获取最新配置的目的。

2.2 负载均衡

负载均衡是一种相当常见的计算机网络技术，用来对多个计算机、网络连接、CPU、磁盘驱动或其他资源进行分配负载，以达到优化资源使用、最大化吞吐率、最小化响应时间和避免过载的目的。

使用Zookeeper实现动态DNS服务

• **域名配置**，首先在Zookeeper上创建一个节点来进行域名配置，如DDNS/app1/server.app1.company1.com。

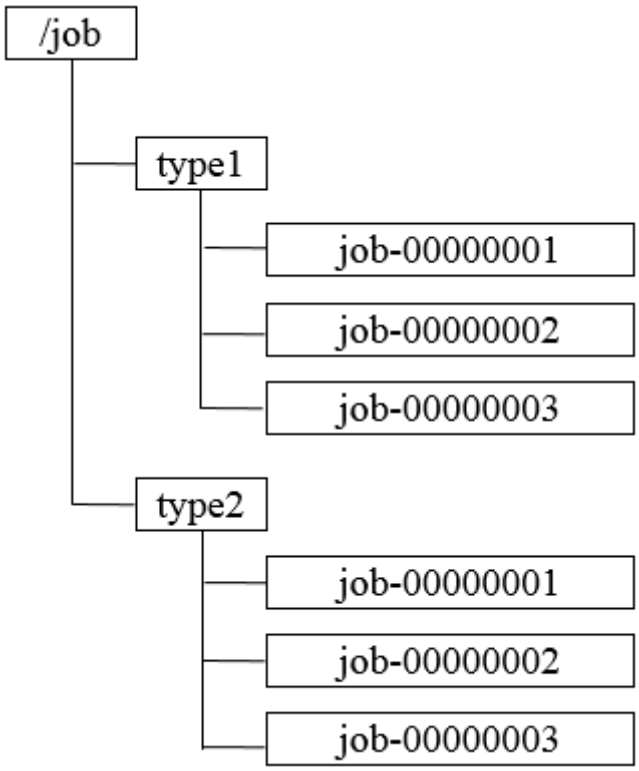
• **域名解析**，应用首先从域名节点中获取IP地址和端口的配置，进行自行解析。同时，应用程序还会在域名节点上注册一个数据变更Watcher监听，以便及时收到域名变更的通知。

• **域名变更**，若发生IP或端口号变更，此时需要进行域名变更操作，此时，只需要对指定的域名节点进行更新操作，Zookeeper就会向订阅的客户端发送这个事件通知，客户端之后就再次进行域名配置的获取。

2.3 命名服务

命名服务是分步实现系统中较为常见的一类场景，分布式系统中，被命名的实体通常可以是集群中的机器、提供的服务地址或远程对象等，通过命名服务，客户端可以根据指定名字来获取资源的实体、服务地址和提供者的信息。Zookeeper也可帮助应用系统通过资源引用的方式来实现对资

源的定位和使用，广义上的命名服务的资源定位都不是真正意义上的实体资源，在分布式环境中，上层应用仅仅需要一个全局唯一的名字。Zookeeper可以实现一套分布式全局唯一ID的分配机制。



通过调用Zookeeper节点创建的API接口就可以创建一个顺序节点，并且在API返回值中会返回这个节点的完整名字，利用此特性，可以生成全局ID，其步骤如下

1. 客户端根据任务类型，在指定类型的任务下通过调用接口创建一个顺序节点，如"job-"。

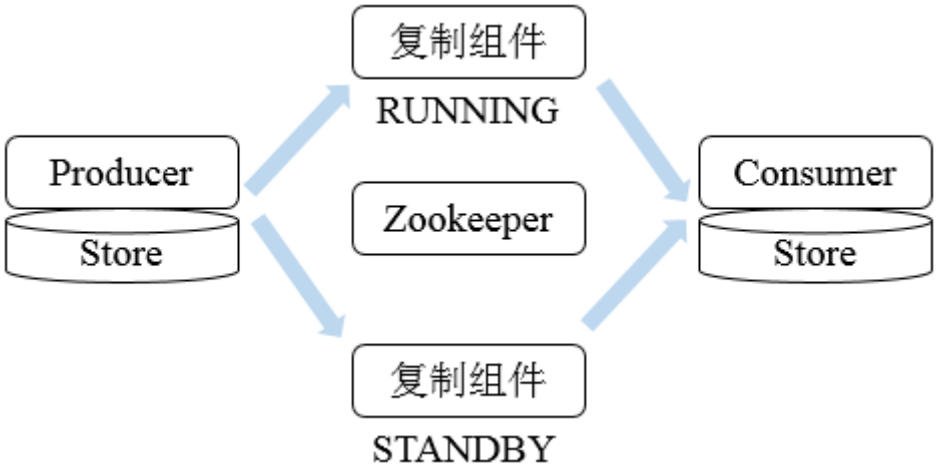
2. 创建完成后，会返回一个完整的节点名，如"job-00000001"。

3. 客户端拼接type类型和返回值后，就可以作为全局唯一ID了，如"type2-job-00000001"。

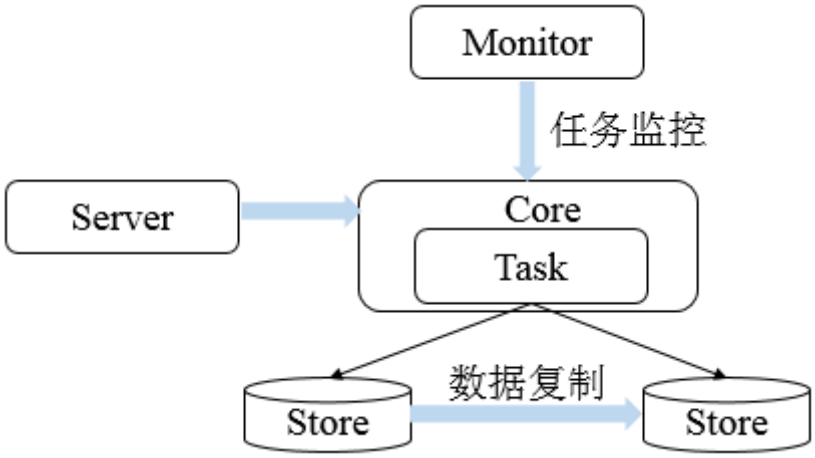
2.4 分布式协调/通知

Zookeeper中特有的Watcher注册于异步通知机制，能够很好地实现分布式环境下不同机器，甚至不同系统之间的协调与通知，从而实现对数据变更的实时处理。通常的做法是不同的客户端都对Zookeeper上的同一个数据节点进行Watcher注册，监听数据节点的变化（包括节点本身和子节点），若数据节点发生变化，那么所有订阅的客户端都能够接收到相应的Watcher通知，并作出相应处理。

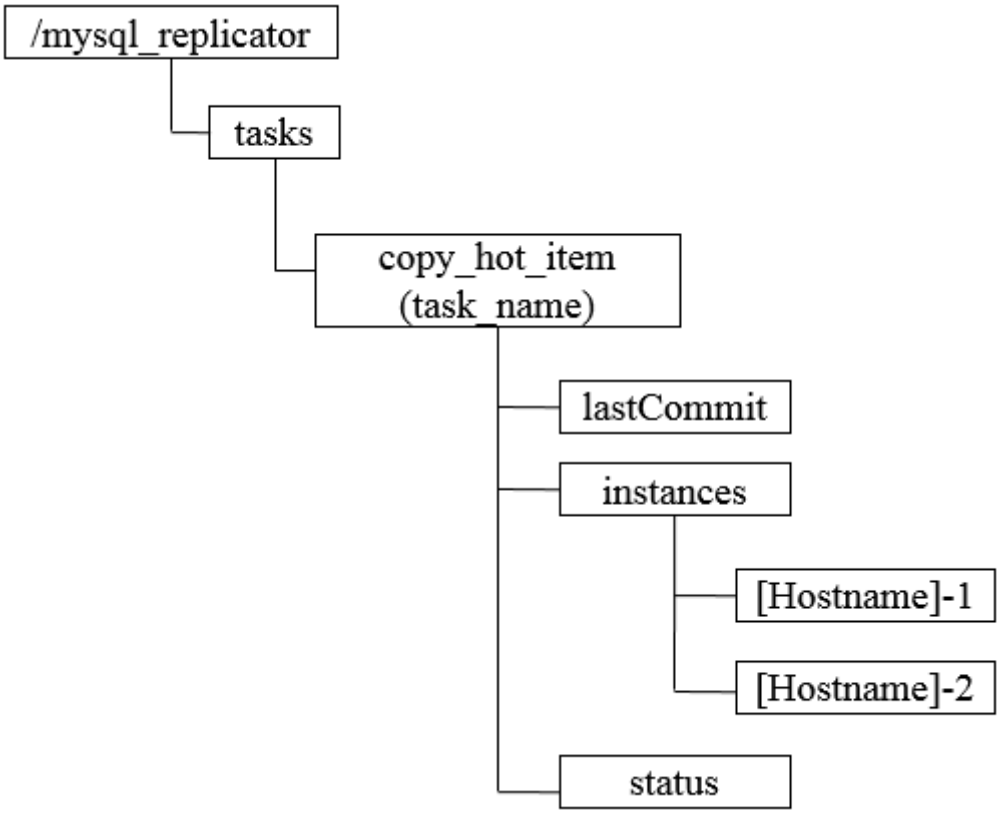
MySQL数据复制总线是一个实时的数据复制框架，用于在不同的MySQL数据库实例之间进行异步数据复制和数据变化通知，整个系统由MySQL数据库集群、消息队列系统、任务管理监控平台、Zookeeper集群等组件共同构成的一个包含生产者、复制管道、数据消费等部分的数据总线系统。



Zookeeper主要负责进行分布式协调工作，在具体的实现上，根据功能将数据复制组件划分为三个模块：Core（实现数据复制核心逻辑，将数据复制封装成管道，并抽象出生产者和消费者概念）、Server（启动和停止复制任务）、Monitor（监控任务的运行状态，若数据复制期间发生异常或出现故障则进行告警）



每个模块作为独立的进程运行在服务端，运行时的数据和配置信息均保存在Zookeeper上。



① **任务创建**，Core进程启动时，首先向/mysql_replicator/tasks节点注册任务，如创建一个子节点/mysql_replicator/tasks/copy_hot/item，若注册过程中发现该子节点已经存在，说明已经有其他Task机器注册了该任务，因此其自身不需要再创建该节点。

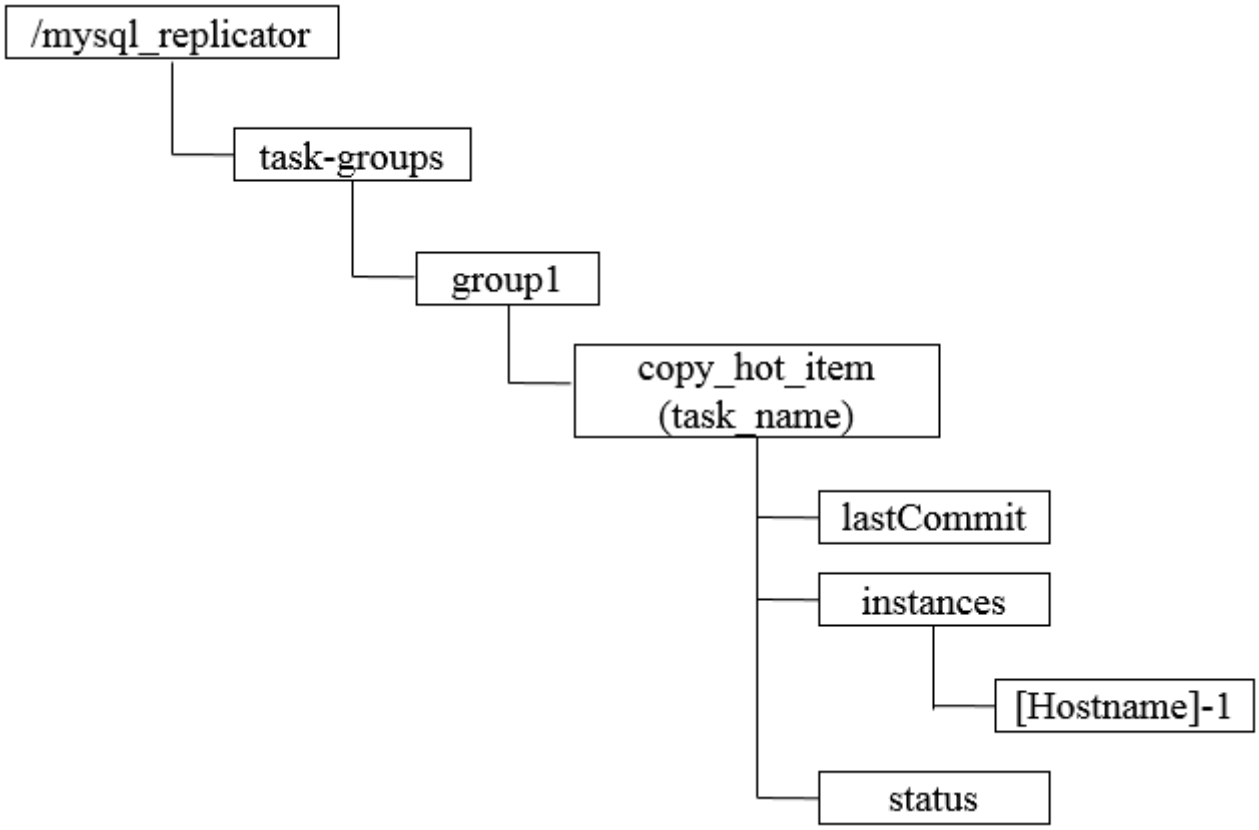
② **任务热备份**，为了应对任务故障或者复制任务所在主机故障，复制组件采用"热备份"的容灾方式，即将同一个复制任务部署在不同的主机上，主备任务机通过Zookeeper互相检测运行监控状况。无论在第一步是否创建了任务节点，每台机器都需要在/mysql_replicator/tasks/copy_hot_item/instrances节点上将自己的主机名注册上去，节点类型为临时顺序节点，在完成子节点创建后，每天任务机器都可以获取到自己创建的节点名及所有子节点列表，然后通过对比判断自己是否是所有子节点中序号最小的，若是，则将自己运行状态设置为RUNNING，其他机器设置为STANDBY，这种策略称为小序号优先策略。

③ **热备切换**，完成运行状态的标示后，其中标记为RUNNING的客户端机器进行正常的数据复制，而标记为STANDBY的机器则进入待命状态，一旦RUNNING机器出现故障，那么所有标记为STANDBY的机器再次按照小序号优先策略来选出RUNNIG机器运行（STANDY机器需要在/mysql_replicator/tasks/copy_hot_item/instances节点上注册一个子节点列表变更监听，RUNNING机器宕机与Zookeeper断开连接后，对应的节点也会消失，于是所有客户端收到通知，进行新一轮选举）。

④ **记录执行状态**，RUNNING机器需要将运行时的上下文状态保留给STANDBY机器。

⑤ **控制台协调**，Server的主要工作就是进行任务控制，通过Zookeeper来对不同任务进行控制和协调，Server会将每个复制任务对应生产者的元数据及消费者的相关信息以配置的形式写入任务节点/mysql_replicator/tasks/copy_hot_item中去，以便该任务的所有任务机器都能够共享复制任务的配置。

在上述热备份方案中，针对一个任务，都会至少分配两台任务机器来进行热备份（RUNNING和STANDBY、即主备机器），若需要MySQL实例需要进行数据复制，那么需要消耗太多机器。此时，需要使用冷备份方案，其对所有任务进行分组。



Core进程被配置了所属组（Group），即若一个Core进程被标记了group1，那么在Core进程启动后，会到对应的Zookeeper group1节点下面获取所有的Task列表，假如找到任务"copy_hot_item"之后，就会遍历这个Task列表的instances节点，但凡还没有子节点，则创建一个临时的顺序节点如/mysql_replicator/task-groups/group1/copy_hot_item/instances/[Hostname]-1，当然，在这个过程中，其他Core进程也会在这个instances节点下创建类似的子节点，按照"小序号优先"策略确定RUNNING，不同的是，其他Core进程会自动删除自己创建的子节点，然后遍历下一个Task节点，这样的过程称为冷备份扫描，这样，所有的Core进程在扫描周期内不断地对相应的Group下来的Task进行冷备份。

在绝大多数分布式系统中，系统机器间的通信无外乎**心跳检测**、**工作进度汇报**和**系统调度**。

① **心跳检测**，不同机器间需要检测到彼此是否在正常运行，可以使用Zookeeper实现机器间的心跳检测，基于其临时节点特性（临时节点的生存周期是客户端会话，客户端若当即后，其临时节点自然不再存在），可以让不同机器都在Zookeeper的一个指定节点下创建临时子节点，不同的机器之间可以根据这个临时子节点来判断对应的客户端机器是否存活。通过Zookeeper可以大大减少系统耦合。

② **工作进度汇报**，通常任务被分发到不同机器后，需要实时地将自己的任务执行进度汇报给分发系统，可以在Zookeeper上选择一个节点，每个任务客户端都在这个节点下面创建临时子节点，这样不仅可以判断机器是否存活，同时各个机器可以将自己的任务执行进度写到该临时节点中去，以便中心系统能够实时获取任务的执行进度。

③ **系统调度**，Zookeeper能够实现如下系统调度模式：分布式系统由控制台和一些客户端系统两部分构成，控制台的职责就是需要将一些指令信息发送给所有的客户端，以控制他们进行相应的业务逻辑，后台管理人员在控制台上做一些操作，实际上就是修改Zookeeper上某些节点的数据，Zookeeper可以把数据变更以时间通知的形式发送给订阅客户端。

2.5 集群管理

Zookeeper的两大特性：

• 客户端如果对Zookeeper的数据节点注册Watcher监听，那么当该数据及诶单内容或是其子节点列表发生变更时，Zookeeper服务器就会向订阅的客户端发送变更通知。

• 对在Zookeeper上创建的临时节点，一旦客户端与服务器之间的会话失效，那么临时节点也会被自动删除。

利用其两大特性，可以实现集群机器存活监控系统，若监控系统在/clusterServers节点上注册一个Watcher监听，那么但凡进行动态添加机器的操作，就会在/clusterServers节点下创建一个临时节点：/clusterServers/[Hostname]，这样，监控系统就能够实时监测机器的变动情况。下面通过分布式日志收集系统的典型应用来学习Zookeeper如何实现集群管理。

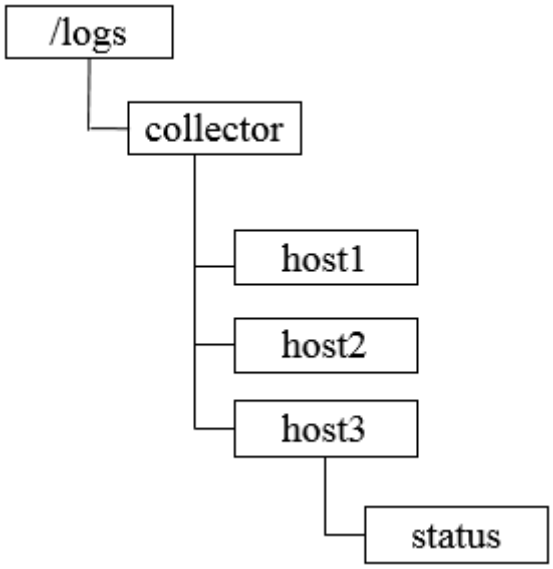
分布式日志收集系统的核心工作就是收集分布在不同机器上的系统日志，在典型的日志系统架构设计中，整个日志系统会把所有需要收集的日志机器分为多个组别，每个组别对应一个收集器，这个收集器其实就是一个后台机器，用于收集日志，对于大规模的分布式日志收集系统场景，通常需要解决两个问题：

• 变化的日志源机器

• 变化的收集器机器

无论是日志源机器还是收集器机器的变更，最终都可以归结为如何快速、合理、动态地为每个收集器分配对应的日志源机器。使用Zookeeper的场景步骤如下

① **注册收集器机器**，在Zookeeper上创建一个节点作为收集器的根节点，例如/logs/collector的收集器节点，每个收集器机器启动时都会在收集器节点下创建自己的节点，如/logs/collector/[Hostname]



② **任务分发**，所有收集器机器都创建完对应节点后，系统根据收集器节点下子节点的个数，将所有日志源机器分成对应的若干组，然后将分组后的机器列表分别写到这些收集器机器创建的子节点，如/logs/collector/host1上去。这样，收集器机器就能够根据自己对应的收集器节点上获取日志源机器列表，进而开始进行日志收集工作。

③ **状态汇报**，完成任务分发后，机器随时会宕机，所以需要有一个收集器的状态汇报机制，每个收集器机器上创建完节点后，还需要再对应子节点上创建一个状态子节点，如/logs/collector/host/status，每个收集器机器都需要定期向该结点写入自己的状态信息，这可看做是心跳检测机制，通常收集器机器都会写入日志收集状态信息，日志系统通过判断状态子节点最后的更新时间来确定收集器机器是否存活。

④ **动态分配**，若收集器机器宕机，则需要动态进行收集任务的分配，收集系统运行过程中关注/logs/collector节点下所有子节点的变更，一旦有机器停止汇报或有新机器加入，就开始进行任务的重新分配，此时通常由两种做法：

• **全局动态分配**，当收集器机器宕机或有新的机器加入，系统根据新的收集器机器列表，立即对所有的日志源机器重新进行一次分组，然后将其分配给剩下的收集器机器。

• **局部动态分配**，每个收集器机器在汇报自己日志收集状态的同时，也会把自己的负载汇报上去，如果一个机器宕机了，那么日志系统就会把之前分配给这个机器的任务重新分配到那些负载较低的机器，同样，如果有新机器加入，会从那些负载高的机器上转移一部分任务给新机器。

上述步骤已经完整的说明了整个日志收集系统的工作流程，其中有两点注意事项。

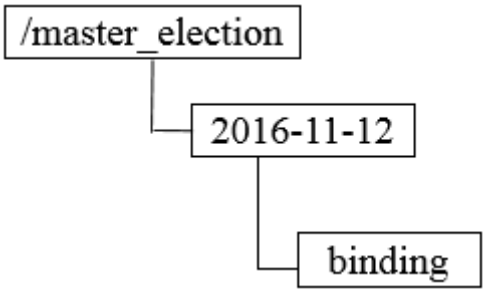
①**节点类型**，在/logs/collector节点下创建临时节点可以很好的判断机器是否存活，但是，若机器挂了，其节点会被删除，记录在节点上的日志源机器列表也被清除，所以需要选择持久节点来标识每一台机器，同时在节点下分别创建/logs/collector/[Hostname]/status节点来表征每一个收集器机器的状态，这样，既能实现对所有机器的监控，同时机器挂掉后，依然能够将分配任务还原。

② **日志系统节点监听**，若采用Watcher机制，那么通知的消息量的网络开销非常大，需要采用日志系统主动轮询收集器节点的策略，这样可以节省网络流量，但是存在一定的延时。

2.6 Master选举

在分布式系统中，Master往往用来协调集群中其他系统单元，具有对分布式系统状态变更的决定权，如在读写分离的应用场景中，客户端的写请求往往是由Master来处理，或者其常常处理一些复杂的逻辑并将处理结果同步给其他系统单元。利用Zookeeper的强一致性，能够很好地保证在分布式高并发情况下节点的创建一定能够保证全局唯一性，即Zookeeper将会保证客户端无法重复创建一个已经存在的数据节点。

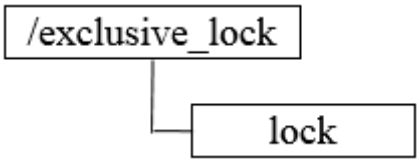
首先创建/master_election/2016-11-12节点，客户端集群每天会定时往该节点下创建临时节点，如/master_election/2016-11-12/binding，这个过程中，只有一个客户端能够成功创建，此时其变成master，其他节点都会在节点/master_election/2016-11-12上注册一个子节点变更的Watcher，用于监控当前的Master机器是否存活，一旦发现当前Master挂了，其余客户端将会重新进行Master选举。



2.7 分布式锁

分布式锁用于控制分布式系统之间同步访问共享资源的一种方式，可以保证不同系统访问一个或一组资源时的一致性，主要分为排它锁和共享锁。

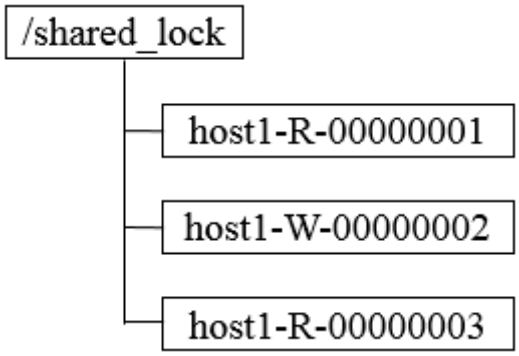
排它锁又称为写锁或独占锁，若事务T1对数据对象O1加上了排它锁，那么在整个加锁期间，只允许事务T1对O1进行读取和更新操作，其他任何事务都不能再对这个数据对象进行任何类型的操作，直到T1释放了排它锁。



① **获取锁**，在需要获取排它锁时，所有客户端通过调用接口，在/exclusive_lock节点下创建临时子节点/exclusive_lock/lock。Zookeeper可以保证只有一个客户端能够创建成功，没有成功的客户端需要注册/exclusive_lock节点监听。

② **释放锁**，当获取锁的客户端宕机或者正常完成业务逻辑都会导致临时节点的删除，此时，所有在/exclusive_lock节点上注册监听的客户端都会收到通知，可以重新发起分布式锁获取。

共享锁又称为读锁，若事务T1对数据对象O1加上共享锁，那么当前事务只能对O1进行读取操作，其他事务也只能对这个数据对象加共享锁，直到该数据对象上的所有共享锁都被释放。



① **获取锁**，在需要获取共享锁时，所有客户端都会到/shared_lock下面创建一个临时顺序节点，如果是读请求，那么就创建例如/shared_lock/host1-R-00000001的节点，如果是写请求，那么就创建例如/shared_lock/host2-W-00000002的节点。

② **判断读写顺序**，不同事务可以同时对一个数据对象进行读写操作，而更新操作必须在当前没有任何事务进行读写情况下进行，通过Zookeeper来确定分布式读写顺序，大致分为四步。

1. 创建完节点后，获取/shared_lock节点下所有子节点，并对该节点变更注册监听。

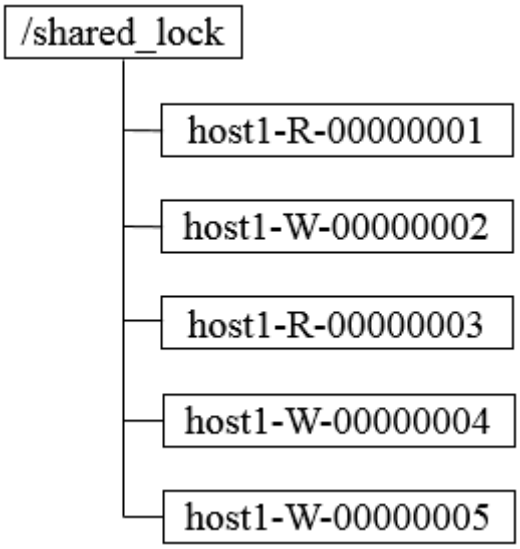
2. 确定自己的节点序号在所有子节点中的顺序。

3. 对于读请求：若没有比自己序号小的子节点或所有比自己序号小的子节点都是读请求，那么表明自己已经成功获取到共享锁，同时开始执行读取逻辑，若有写请求，则需要等待。对于写请求：若自己不是序号最小的子节点，那么需要等待。

4. 接收到Watcher通知后，重复步骤1。

③ **释放锁**，其释放锁的流程与独占锁一致。

上述共享锁的实现方案，可以满足一般分布式集群竞争锁的需求，但是如果机器规模扩大会出现一些问题，下面着重分析判断读写顺序的步骤3。



针对如上图所示的情况进行分析

1. host1首先进行读操作，完成后将节点/shared_lock/host1-R-00000001删除。

2. 余下4台机器均收到这个节点移除的通知，然后重新从/shared_lock节点上获取一份新的子节点列表。

3. 每台机器判断自己的读写顺序，其中host2检测到自己序号最小，于是进行写操作，余下的机器则继续等待。

4. 继续...

可以看到，host1客户端在移除自己的共享锁后，Zookeeper发送了子节点更变Watcher通知给所有机器，然而除了给host2产生影响外，对其他机器没有任何作用。大量的Watcher通知和子节点列表获取两个操作会重复运行，这样会造成系能鞲影响和网络开销，更为严重的是，如果同一时间有多个节点对应的客户端完成事务或事务中断引起节点小时，Zookeeper服务器就会在短时间内向其他所有客户端发送大量的事件通知，这就是所谓的**羊群效应**。

可以有如下改动来避免羊群效应。

1. 客户端调用create接口常见类似于/shared_lock/[Hostname]-请求类型-序号的临时顺序节点。

2. 客户端调用getChildren接口获取所有已经创建的子节点列表（不注册任何Watcher）。

3. 如果无法获取共享锁，就调用exist接口来对比自己小的节点注册Watcher。对于读请求：向比自己序号小的最后一个写请求节点注册Watcher监听。对于写请求：向比自己序号小的最后一个节点注册Watcher监听。

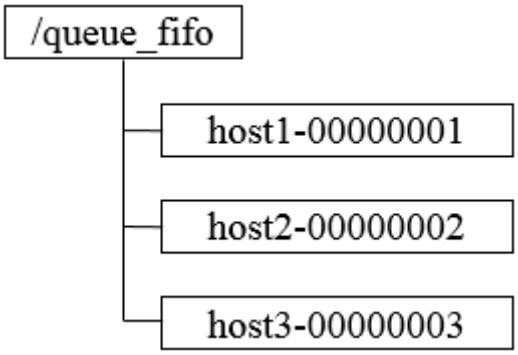
4. 等待Watcher通知，继续进入步骤2。

此方案改动主要在于：每个锁竞争者，只需要关注/shared_lock节点下序号比自己小的那个节点是否存在即可。

2.8 分布式队列

分布式队列可以简单分为**先入先出队列模型**和**等待队列元素聚集后统一安排处理执行的Barrier模型**。

① **FIFO先入先出**，先进入队列的请求操作先完成后，才会开始处理后面的请求。FIFO队列就类似于全写的共享模型，所有客户端都会到/queue_fifo这个节点下创建一个临时节点，如/queue_fifo/host1-00000001。



创建完节点后，按照如下步骤执行。

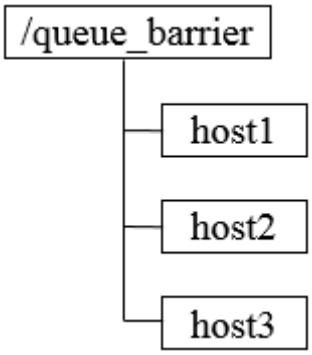
1. 通过调用getChildren接口来获取/queue_fifo节点的所有子节点，即获取队列中所有的元素。

2. 确定自己的节点序号在所有子节点中的顺序。

3. 如果自己的序号不是最小，那么需要等待，同时向比自己序号小的最后一个节点注册Watcher监听。

4. 接收到Watcher通知后，重复步骤1。

② **Barrier分布式屏障**，最终的合并计算需要基于很多并行计算的子结果来进行，开始时，/queue_barrier节点已经默认存在，并且将结点数据内容赋值为数字n来代表Barrier值，之后，所有客户端都会到/queue_barrier节点下创建一个临时节点，例如/queue_barrier/host1。



创建完节点后，按照如下步骤执行。

1. 通过调用getData接口获取/queue_barrier节点的数据内容，如10。

2. 通过调用getChildren接口获取/queue_barrier节点下的所有子节点，同时注册对子节点变更的Watcher监听。

3. 统计子节点的个数。

4. 如果子节点个数还不足10个，那么需要等待。

5. 接受到Wacher通知后，重复步骤3。

三、总结

本篇博客讲解了如何利用Zookeeper的特性来完成典型应用，展示了Zookeeper在解决分布式问题上的强大作用，基于Zookeeper对分布式数据一致性的保证及其特性，开发人员能够构建出自己的分布式系统。也谢谢各位园友的观看~