【ZooKeeper Notes 28】ZooKeeper 典型应用场景一览

@ni 掌柜

nileader@gmail.com

http://weibo.com/nileader

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 解决很多分布式问题。网上对 ZK 的应用场景也有不少介绍，本文将结合作者身边的项目例子，系统地对 ZK 的应用场景进行一个分门归类的介绍。

值得注意的是，ZK 并非天生就是为这些应用场景设计的，都是后来众多开发者根据其框架的特性，利用其提供的一系列 API 接口（或者称为原语集），摸索出来的典型使用方法。因此，也非常欢迎读者分享你在 ZK 使用上的奇技淫巧。

ZooKeeper 典型应用场景一览

数据发布与订阅（配置中心）

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到 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，服务式服务框架的服务地址列表等就非常适合使用。

- 应用中用到的一些配置信息放到 ZK 上进行集中管理。这类场景通常是这样：应用在启动的时候会主动来获取一次配置，同时，在节点上注册一个 Watcher，这样一来，以后每次配置有更新的时候，都会实时通知到订阅的客户端，从来达到获取最新配置信息的目的。
- 分布式搜索服务中，索引的元信息和服务器集群机器的节点状态存放在 ZK 的一些指定节点，供各个客户端订阅使用。
- 分布式日志收集系统。这个系统的核心工作是收集分布在不同机器的日志。收集器通常是按照应用来分配收集任务单元，因此需要在 ZK 上创建一个以应用名作为 path 的节点 P，并将这个应用的所有机器 ip，以子节点的形式注册到节点 P 上，这样一来就能够实现机器变动的时候，能够实时通知到收集器调整任务分配。
- 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的发问。通常是暴露出接口，例如 JMX 接口，来获取一些运行时的信息。引入 ZK 之后，就不用自己实现一套方案了，只要将这些信息存放到指定的 ZK 节点上即可。

注意：在上面提到的应用场景中，有个默认前提是：数据量很小，但是数据更新可能会比较快的场景。

负载均衡

这里说的负载均衡是指软负载均衡。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，达到对等服务。而消费者就须要在这些对等的服务器中选择一个来执行相关的业务逻辑，其中比较典型的是消息中间件中的生产者，消费者负载均衡。

- 消息中间件中发布者和订阅者的负载均衡，linkedin 开源的 KafkaMQ 和阿里开源的 metaq 都是通过 zookeeper 来做到生产者、消费者的负载均衡。这里以 metaq 为例如讲下：

生产者负载均衡：

metaq 发送消息的时候，生产者在发送消息的时候必须选择一台 broker 上的一个分区来发送消息，因此 metaq 在运行过程中，会把所有 broker 和对应的分区信息全部注册到 ZK 指定节点上，默认的策略是一个依次轮询的过程，生产者在通过 ZK 获取分区列表之后，会按照 brokerId 和 partition 的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。

消费负载均衡：

在消费过程中，一个消费者会消费一个或多个分区中的消息，但是一个分区只会由一个消费者来消费。MetaQ 的消费策略是：

1. 每个分区针对同一个 group 只挂载一个消费者。
2. 如果同一个 group 的消费者数目大于分区数目，则多出来的消费者将不参与消费。

3. 如果同一个 group 的消费者数目小于分区数目，则有部分消费者需要额外承担消费任务。

在某个消费者故障或者重启等情况下，其他消费者会感知到这一变化（通过 zookeeper watch 消费者列表），然后重新进行负载均衡，保证所有的分区都有消费者进行消费。

命名服务(Naming Service)

命名服务也是分布式系统中比较常见的一类场景。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名字来获取资源或服务的地址，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，远程对象等等——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。

- 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，点击这里查看 Dubbo 开源项目。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。

分布式通知/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

- 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。
- 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。
- 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

集群管理与 Master 选举

- 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。

  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。

- Master 选举则是 zookeeper 中最为经典的应用场景了。

  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。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。

- 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。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。
- 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，会重新选举出一个 HMaster 来运行，从而避免了 HMaster 的单点问题

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。锁服务可以分为两类，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。

分布式队列

队列方面，简单地讲有两种，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。对于第一种先进先出队列，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。这种用法的典型场景是，分布式环境中，一个大任务 Task A，需要在很多子任务完成（或条件就绪）情况下才能进行。这个时候，凡是其中一个子任务完成（就绪），那么就去 /taskList 下建立自己的临时时序节点（CreateMode.EPHEMERAL_SEQUENTIAL），当 /taskList 发现自己下面的子节点满足指定个数，就可以进行下一步按序进行处理了。