ZooKeeper原理及使用 - 就是你的博客 - 博客频道

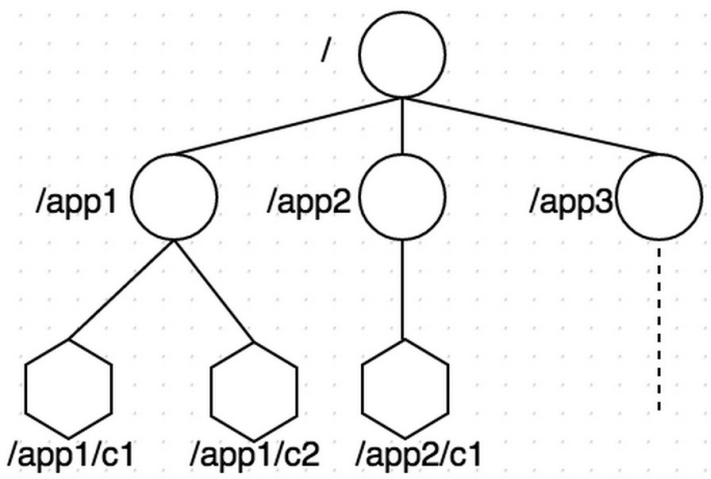
2014-08-07 17:19 55835人阅读 评论(7) 收藏 举报

目录(?)[+]

ZooKeeper是Hadoop Ecosystem中非常重要的组件，它的主要功能是为分布式系统提供一致性协调(Coordination)服务，与之对应的Google的类似服务叫Chubby。今天这篇文章分为三个部分来介绍ZooKeeper，第一部分介绍ZooKeeper的基本原理，第二部分介绍ZooKeeper提供的Client API的使用，第三部分介绍一些ZooKeeper典型的应用场景。

ZooKeeper基本原理

1. 数据模型



如上图所示，ZooKeeper数据模型的结构与Unix文件系统很类似，整体上可以看作是一棵树，每个节点称做一个ZNode。每个ZNode都可以通过其路径唯一标识，比如上图中第三层的第一个ZNode, 它的路径是/app1/c1。在每个ZNode上可存储少量数据(默认是1M, 可以通过配置修改, 通常不建议在ZNode上存储大量的数据)，这个特性非常有用，在后面的典型应用场景中会介绍到。另外，每个ZNode上还存储了其Acl信息，这里需要注意，虽说ZNode的树形结构跟Unix文件系统很类似，但是其Acl与Unix文件系统是完全不同的，每个ZNode的Acl的独立的，子结点不会继承父结点的，关于ZooKeeper中的Acl可以参考之前写过的一篇文章《说说Zookeeper中的ACL》。

2.重要概念

2.1 ZNode

前文已介绍了ZNode, ZNode根据其本身的特性，可以分为下面两类：

- Regular ZNode: 常规型ZNode, 用户需要显式的创建、删除
- Ephemeral ZNode: 临时型ZNode, 用户创建它之后，可以显式的删除，也可以在创建它的Session结束后，由ZooKeeper Server自动删除

ZNode还有一个Sequential的特性，如果创建的时候指定的话，该ZNode的名字后面会自动Append一个不断增加的SequenceNo。

2.2 Session

Client与ZooKeeper之间的通信，需要创建一个Session，这个Session会有一个超时时间。因为ZooKeeper集群会把Client的Session信息持久化，所以在Session没超时之前，Client与ZooKeeper Server的连接可以在各个ZooKeeper Server之间透明地移动。

在实际的应用中，如果Client与Server之间的通信足够频繁，Session的维护就不需要其它额外的消息了。否则，ZooKeeper Client会每t/3 ms发一次心跳给Server，如果Client 2t/3 ms没收到来自Server的心跳回应，就会换到一个新的ZooKeeper Server上。这里t是用户配置的Session的超时时间。

2.3 Watcher

ZooKeeper支持一种Watch操作，Client可以在某个ZNode上设置一个Watcher，来Watch该ZNode上的变化。如果该ZNode上有相应的变化，就会触发这个Watcher，把相应的事件通知给设置Watcher的Client。需要注意的是，ZooKeeper中的Watcher是一次性的，即触发一次就会被取消，如果想继续Watch的话，需要客户端重新设置Watcher。这个跟epoll里的oneshot模式有点类似。

3. ZooKeeper特性

3.1 读、写(更新)模式

在ZooKeeper集群中，读可以从任意一个ZooKeeper Server读，这一点是保证ZooKeeper比较好的读性能的关键；写的请求会先Forwarder到Leader，然后由Leader来通过ZooKeeper中的原子广播协议，将请求广播给所有的Follower，Leader收到一半以上的写成功的Ack后，就认为该写成功了，就会将该写进行持久化，并告诉客户端写成功了。

3.2 WAL和Snapshot

和大多数分布式系统一样，ZooKeeper也有WAL(Write-Ahead-Log)，对于每一个更新操作，ZooKeeper都会先写WAL, 然后再对内存中的数据做更新，然后向Client通知更新结果。另外，ZooKeeper还会定期将内存中的目录树进行Snapshot，落地到磁盘上，这个跟HDFS中的FSImage是比较类似的。这么做的主要目的，一当然是数据的持久化，二是加快重启之后的恢复速度，如果全部通过Replay WAL的形式恢复的话，会比较慢。

3.3 FIFO
对于每一个ZooKeeper客户端而言，所有的操作都是遵循FIFO顺序的，这一特性是由下面两个基本特性来保证的：一是ZooKeeper Client与Server之间的网络通信是基于TCP，TCP保证了Client/Server之间传输包的顺序；二是ZooKeeper Server执行客户端请求也是严格按照FIFO顺序的。

3.4 Linearizability
在ZooKeeper中，所有的更新操作都有严格的偏序关系，更新操作都是串行执行的，这一点是保证ZooKeeper功能正确性的关键。

ZooKeeper Client API

ZooKeeper Client Library提供了丰富直观的API供用户程序使用，下面是一些常用的API：

- create(path, data, flags): 创建一个ZNode, path是其路径，data是要存储在该ZNode上的数据，flags常用的有: PERSISTEN, PERSISTENT_SEQUENTAIL, EPHEMERAL, EPHEMERAL_SEQUENTAIL
- delete(path, version): 删除一个ZNode，可以通过version删除指定的版本, 如果version是-1的话，表示删除所有的版本
- exists(path, watch): 判断指定ZNode是否存在，并设置是否Watch这个ZNode。这里如果要设置Watcher的话，Watcher是在创建ZooKeeper实例时指定的，如果要设置特定的Watcher的话，可以调用另一个重载版本的exists(path, watcher)。以下几个带watch参数的API也都类似
- getData(path, watch): 读取指定ZNode上的数据，并设置是否watch这个ZNode
- setData(path, watch): 更新指定ZNode的数据，并设置是否Watch这个ZNode
- getChildren(path, watch): 获取指定ZNode的所有子ZNode的名字，并设置是否Watch这个ZNode
- sync(path): 把所有在sync之前的更新操作都进行同步，达到每个请求都在半数以上的ZooKeeper Server上生效。path参数目前没有用
- setAcl(path, acl): 设置指定ZNode的Acl信息
- getAcl(path): 获取指定ZNode的Acl信息

ZooKeeper典型应用场景

1. 名字服务(NameService)
分布式应用中，通常需要一套完备的命令机制，既能产生唯一的标识，又方便人识别和记忆。 我们知道，每个ZNode都可以由其路径唯一标识，路径本身也比较简洁直观，另外ZNode上还可以存储少量数据，这些都是实现统一的NameService的基础。下面以在HDFS

中实现NameService为例，来说明实现NameService的基本布骤:

- 目标：通过简单的名字来访问指定的HDFS机群
- 定义命名规则：这里要做到简洁易记忆。下面是一种可选的方案： [serviceScheme://] [zkCluster]-[clusterName]，比如hdfs://lgprc-example/表示基于lgprc ZooKeeper集群的用来做example的HDFS集群
- 配置DNS映射: 将zkCluster的标识lgprc通过DNS解析到对应的ZooKeeper集群的地址
- 创建ZNode: 在对应的ZooKeeper上创建/NameService/hdfs/lgprc-example结点，将HDFS的配置文件存储于该结点下
- 用户程序要访问hdfs://lgprc-example/的HDFS集群，首先通过DNS找到lgprc的ZooKeeper机群的地址，然后在ZooKeeper的/NameService/hdfs/lgprc-example结点中读取到HDFS的配置，进而根据得到的配置，得到HDFS的实际访问入口

2. 配置管理(Configuration Management)

在分布式系统中，常会遇到这样的场景: 某个Job的很多个实例在运行，它们在运行时大多数配置项是相同的，如果想要统一改某个配置，一个个实例去改，是比较低效，也是比较容易出错的方式。通过ZooKeeper可以很好的解决这样的问题，下面的基本的步骤：

- 将公共的配置内容放到ZooKeeper中某个ZNode上，比如/service/common-conf
- 所有的实例在启动时都会传入ZooKeeper集群的入口地址，并且在运行过程中Watch /service/common-conf这个ZNode
- 如果集群管理员修改了了common-conf，所有的实例都会被通知到，根据收到的通知更新自己的配置，并继续Watch /service/common-conf

3. 组员管理(Group Membership)

在典型的Master-Slave结构的分布式系统中，Master需要作为“总管”来管理所有的Slave, 当有Slave加入，或者有Slave宕机，Master都需要感知到这个事情，然后作出对应的调整，以便不影响整个集群对外提供服务。以HBase为例，HMaster管理了所有的RegionServer，当有新的RegionServer加入的时候，HMaster需要分配一些Region到该RegionServer上去，让其提供服务；当有RegionServer宕机时，HMaster需要将该RegionServer之前服务的Region都重新分配到当前正在提供服务的其它RegionServer上，以便不影响客户端的正常访问。下面是这种场景下使用ZooKeeper的基本步骤：

- Master在ZooKeeper上创建/service/slaves结点，并设置对该结点的Watcher
- 每个Slave在启动成功后，创建唯一标识自己的临时性(Ephemeral)结点/service/slaves/${slave_id}，并将自己地址(ip/port)等相关信息写入该结点
- Master收到有新子结点加入的通知后，做相应的处理

- 如果有Slave宕机，由于它所对应的结点是临时性结点，在它的Session超时后，ZooKeeper会自动删除该结点
- Master收到有子结点消失的通知，做相应的处理

4. 简单互斥锁(Simple Lock)

我们知识，在传统的应用程序中，线程、进程的同步，都可以通过操作系统提供的机制来完成。但是在分布式系统中，多个进程之间的同步，操作系统层面就无能为力了。这时候就需要像ZooKeeper这样的分布式的协调(Coordination)服务来协助完成同步，下面是用ZooKeeper实现简单的互斥锁的步骤，这个可以和线程间同步的mutex做类比来理解：

- 多个进程尝试去在指定的目录下去创建一个临时性(Ephemeral)结点 /locks/my_lock
- ZooKeeper能保证，只会有一个进程成功创建该结点，创建结点成功的进程就是抢到锁的进程，假设该进程为A
- 其它进程都对/locks/my_lock进行Watch
- 当A进程不再需要锁，可以显式删除/locks/my_lock释放锁；或者是A进程宕机后Session超时，ZooKeeper系统自动删除/locks/my_lock结点释放锁。此时，其它进程就会收到ZooKeeper的通知，并尝试去创建/locks/my_lock抢锁，如此循环反复

5. 互斥锁(Simple Lock without Herd Effect)

上一节的例子中有一个问题，每次抢锁都会有大量的进程去竞争，会造成羊群效应(Herd Effect)，为了解决这个问题，我们可以通过下面的步骤来改进上述过程：

- 每个进程都在ZooKeeper上创建一个临时的顺序结点(Ephemeral Sequential) /locks/lock_${seq}
- ${seq}最小的为当前的持锁者(${seq}是ZooKeeper生成的Sequenctial Number)
- 其它进程都对只watch比它次小的进程对应的结点，比如2 watch 1, 3 watch 2, 以此类推
- 当前持锁者释放锁后，比它次大的进程就会收到ZooKeeper的通知，它成为新的持锁者，如此循环反复

这里需要补充一点，通常在分布式系统中用ZooKeeper来做Leader Election(选主)就是通过上面的机制来实现的，这里的持锁者就是当前的“主”。

6. 读写锁(Read/Write Lock)

我们知道，读写锁跟互斥锁相比不同的地方是，它分成了读和写两种模式，多个读可以并发执行，但写和读、写都互斥，不能同时执行行。利用ZooKeeper，在上面的基础上，稍做修改也可以实现传统的读写锁的语义，下面是基本的步骤:

- 每个进程都在ZooKeeper上创建一个临时的顺序结点(Ephemeral Sequential)

/locks/lock_${seq}

- ${seq}最小的一个或多个结点为当前的持锁者，多个是因为多个读可以并发
- 需要写锁的进程，Watch比它次小的进程对应的结点
- 需要读锁的进程，Watch比它小的最后一个写进程对应的结点
- 当前结点释放锁后，所有Watch该结点的进程都会被通知到，他们成为新的持锁者，如此循环反复

7. 屏障(Barrier)

在分布式系统中，屏障是这样一种语义: 客户端需要等待多个进程完成各自的任务，然后才能继续往前进行下一步。下用是用ZooKeeper来实现屏障的基本步骤：

- Client在ZooKeeper上创建屏障结点/barrier/my_barrier，并启动执行各个任务的进程
- Client通过exist()来Watch /barrier/my_barrier结点
- 每个任务进程在完成任务后，去检查是否达到指定的条件，如果没达到就啥也不做，如果达到了就把/barrier/my_barrier结点删除
- Client收到/barrier/my_barrier被删除的通知，屏障消失，继续下一步任务

8. 双屏障(Double Barrier)

双屏障是这样一种语义: 它可以用来同步一个任务的开始和结束，当有足够多的进程进入屏障后，才开始执行任务；当所有的进程都执行完各自的任务后，屏障才撤销。下面是用ZooKeeper来实现双屏障的基本步骤：

进入屏障：

- Client Watch /barrier/ready结点, 通过判断该结点是否存在来决定是否启动任务
- 每个任务进程进入屏障时创建一个临时结点/barrier/process/${process_id}，然后检查进入屏障的结点数是否达到指定的值，如果达到了指定的值，就创建一个/barrier/ready结点，否则继续等待
- Client收到/barrier/ready创建的通知，就启动任务执行过程

离开屏障：

- Client Watch /barrier/process，如果其没有子结点，就可以认为任务执行结束，可以离开屏障

- 每个任务进程执行任务结束后，都需要删除自己对应的结点/barrier/process/${process_id}

转载地址：http://www.wuzesheng.com/?p=2609 **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：

- 每个分区针对同一个group只挂载一个消费者。
- 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。
- 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。 阿里巴巴集团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：

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。

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。