Zookeeper 篇

12.1.0 zookeeper 是什么？

Zookeeper 是基于 Google Chubby 论文的开源实现，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、配置管理 等等。 由于 Hadoop 生态系统中很多项目都依赖于 zookeeper，如 Pig，Hive 等， 似乎很像一个动物园管理员，于是取名为 Zookeeper。

12.1.1 zookeeper 提供了什么？

1、文件系统 2、通知机制

12.1.2 zookeeper 文件系统

zookeeper 提供一个类似 unix 文件系统目录的多层级节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。

12.1.3 zookeeper 的四种类型的 znode

PERSISTENT 持久化节点

PERSISTENT_SEQUENTIAL 顺序自动编号持久化节点，这种节点会根据当前已存在的节点数自动加 1

EPHEMERAL 临时节点， 客户端 session 超时这类节点就会被自动删除

EPHEMERAL_SEQUENTIAL 临时自动编号节点

12.1.4 zookeeper 通知机制

client 端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，zk 会主动通知 watch 这个 znode 的 client，然后 client 根据 znode 的变化来做出业务上的改变等。

watcher 的特点：

- 轻量级：一个 callback 函数。
- 异步性：不会 block 正常的读写请求。
- 主动推送：Watch 被触发时，由 Zookeeper 服务端主动将更新推送给客户端。
- 一次性：数据变化时，Watch 只会被触发一次。如果客户端想得到后续更新的通知，必须要在 Watch 被触发后重新注册一个 Watch。
- 仅通知：仅通知变更类型，不附带变更后的结果。
- 顺序性：如果多个更新触发了多个 Watch，那 Watch 被触发的顺序与更新顺序一致。

使用 watch 的注意事项：

- 由于 watcher 是一次性的，所以需要自己去实现永久 watch
- 如果被 watch 的节点频繁更新，会出现“丢数据”的情况
- watcher 数量过多会导致性能下降

12.1.5 zookeeper 有哪些应用场景？

1、名字服务

2、配置管理

3、集群管理

4、分布式锁

5、队列管理

6、消息订阅

12.1.6 zk 的命名服务

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

12.1.7 zk 的配置管理服务

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。

12.1.8 zk 的集群管理

所谓集群管理无在乎两点：是否有机器退出和加入、选举 master。 对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。 新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有了，对于第二点，我们稍微改变一

下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

12.1.9 zk 的分布式锁

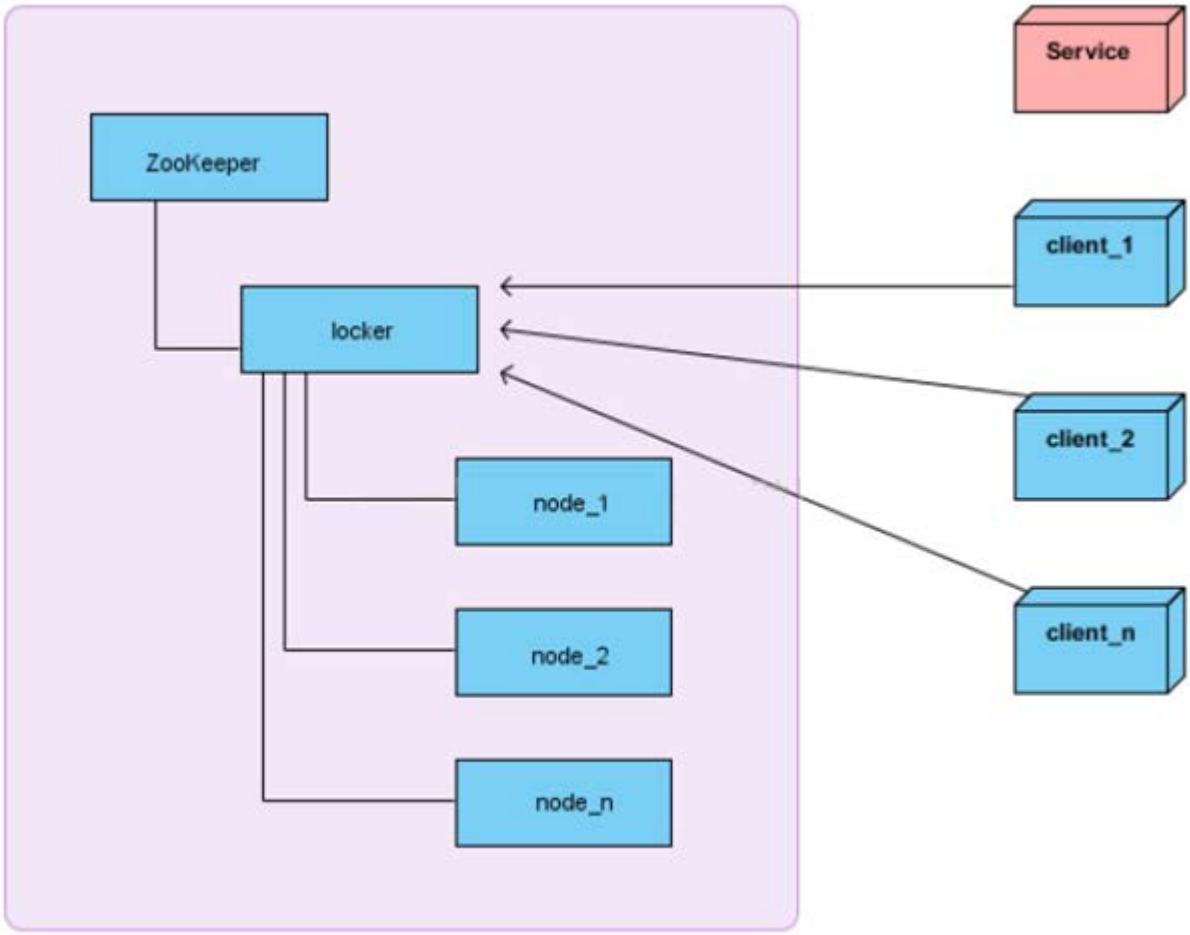
有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。 对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。 对于第二类，/distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

获取分布式锁的流程

在获取分布式锁的时候在 locker 节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用 createNode 方法在 locker 下创建临时顺序节点， 然后调用 getChildren("locker")来获取 locker 下面的所有子节点，注意此时不用设置任何 Watcher。客户端获取到所有的子节点 path 之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非 locker 所有子节点中最小的，说明自己还没有获取到锁，此时有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。 对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 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

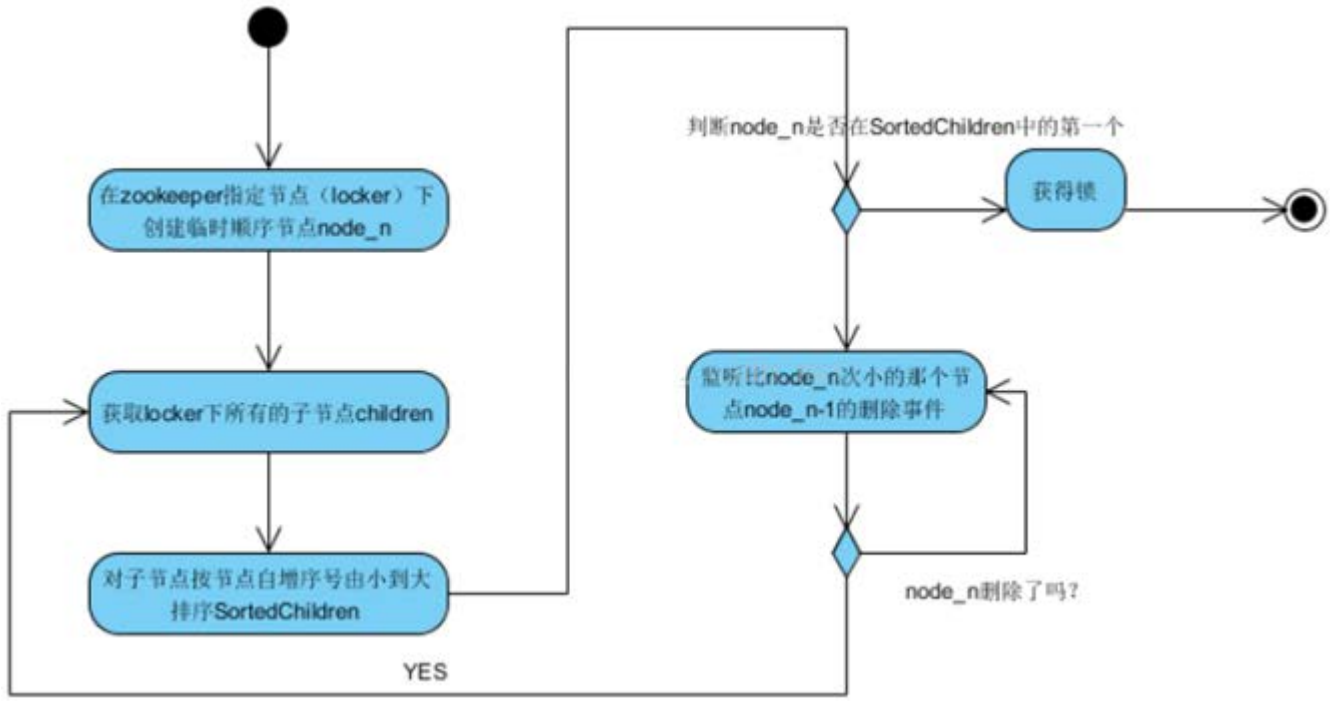
锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。 对于第二类，/distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

获取分布式锁的流程



在获取分布式锁的时候在 locker 节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用 createNode 方法在 locker 下创建临时顺序节点， 然后调用 getChildren("locker")来获取 locker 下面的所有子节点，注意此时不用设置任何 Watcher。客户端获取到所有的子节点 path 之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非 locker 所有子节点中最小的，说明自己还没有获取到锁，

此时客户端需要找到比自己小的那个节点，然后对其调用 exist()方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的 Watcher 会收到相应通知，此时再次判断自己创建的节点是否是 locker 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。



代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于BaseDistributedLock，实现了基于 Zookeeper 实现分布式锁的细节。

客户端需要找到比自己小的那个节点，然后对其调用 exist()方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的 Watcher 会收到相应通知，此时再次判断自己创建的节点是否是 locker 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。

代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于BaseDistributedLock，实现了基于 Zookeeper 实现分布式锁的细节。

12.2.0 zk 队列管理

两种类型的队列： 1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。 2、队列按照 FIFO 方式进行入队和出队操作。 第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。 第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下 Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

12.2.1 zk 数据复制

Zookeeper 作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处： 1、容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作； 2、提高系统的扩展能力 ：把负载分布到多个节点上，或者增加节点来提高系统的负载能力； 3、提高性能：让客户端本地访问就近的节点，提高用户访问速度。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种： 1、写主(WriteMaster) ：对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离； 2、写任意(Write Any)：对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对 zookeeper 来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立 observer 的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

12.2.2 zk 的工作原理

12.2.3 zk 是如何保证事物的顺序一致性

zookeeper 采用了递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期; 纪元; 世; 新时代）用来标识 leader 是否发生改变，如果有新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

12.2.4 zk 集群下 server 工作状态

每个 Server 在工作过程中有四种状态：

LOOKING：当前 Server 不知道 leader 是谁，正在搜寻

LEADING：当前 server 角色为 leader

FOLLOWING：当前 server 角色为 follower

OBSERVING：当前 server 角色为 observer

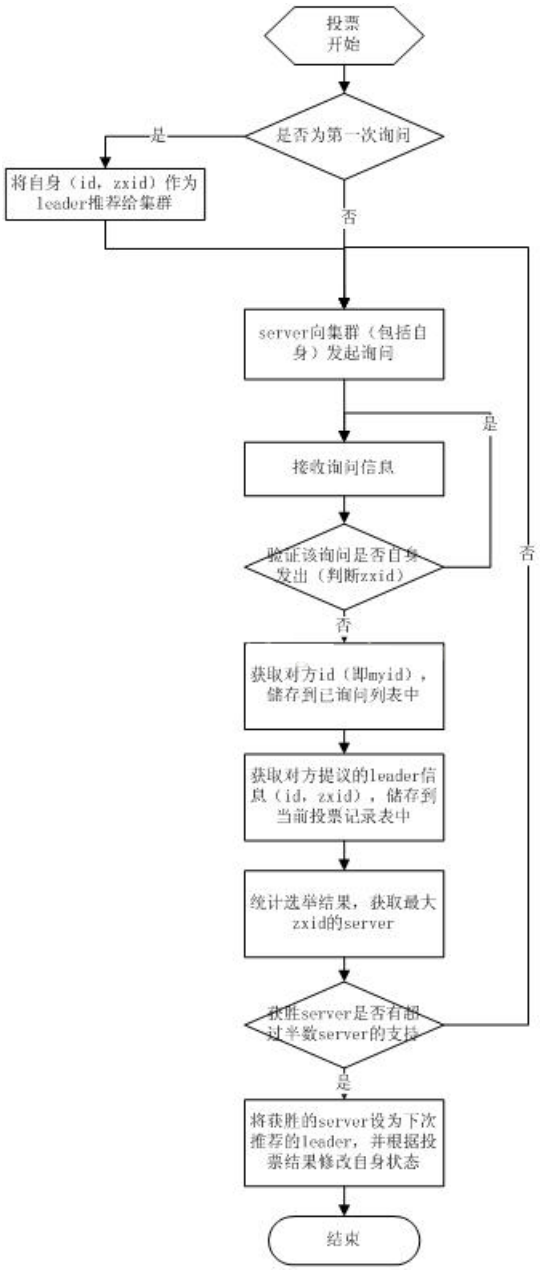
12.2.5 zk 是如何选举 Leader 的？

参考答案：

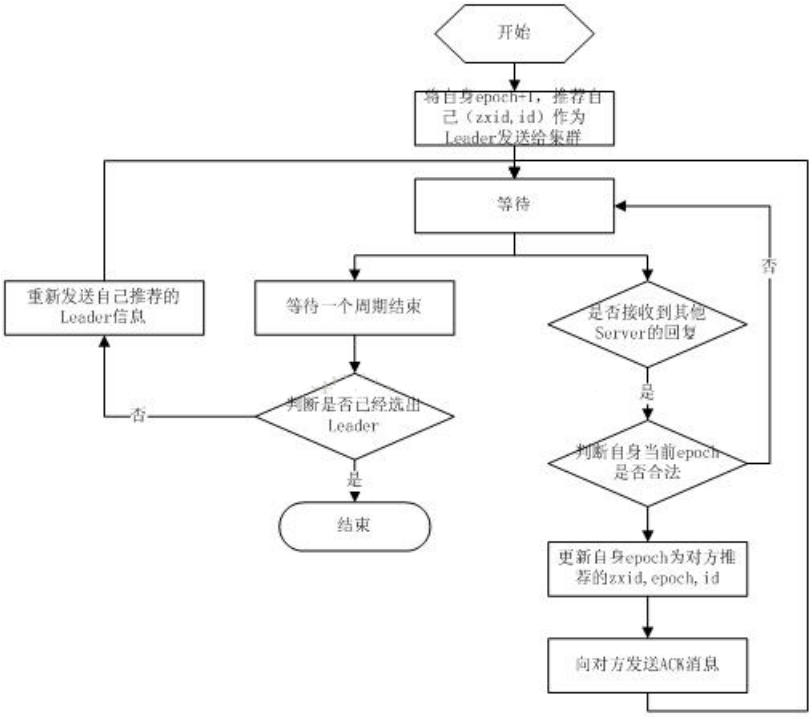
当 leader 崩溃或者 leader 失去大多数的 follower，这时 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 Server 都恢复到一个正确的状态。Zk 的选举算法有两种：一种是基于 basic paxos 实现的，另外一种是基于 fast paxos 算法实现的。系统默认的选举算法为 fast paxos。

1、Zookeeper 选主流程(basic paxos) （1）选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的 Server； （2）选举线程首先向所有 Server 发起一次询问(包括自己)；（3）选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 id(myid)，并存储到当前询问对象列表中，最后获取对方提议的 leader 相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中； （4）收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server； （5）线程将当前 zxid 最大的 Server 设置为当前 Server 要推荐的 Leader，如果此时获胜的 Server 获得 n/2 + 1 的 Server 票数，设置当前推荐的 leader 为获胜的 Server，将根据获胜的 Server 相关信息设置自己的状态，否则，继续这个过程，直到 leader 被选举出来。 通过流程分析我们可以得出：要使 Leader 获得多数 Server 的支持，则 Server 总数必须是奇数 2n+1，且存活的 Server 的数目不得少于 n+1. 每个 Server 启动后都会重复以上流程。在恢复模式下，如

果是刚从崩溃状态恢复的或者刚启动的 server 还会从磁盘快照中恢复数据和会话信息，zk 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。



2、Zookeeper 选主流程(fast paxos) fast paxos 流程是在选举过程中，某 Server 首先向所有 Server 提议自己要成为 leader，当其它 Server 收到提议以后，解决 epoch 和 zxid 的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出 Leader。



12.2.6 zk 同步流程

参考答案：

选完 Leader 以后，zk 就进入状态同步过程。

1. Leader 等待 Follower 和 Observer 连接；
2. Follower 连接 leader，将最大的 zxid 发送给 leader；
3. Leader 根据 follower 的 zxid 确定同步点；
4. 完成同步后通知 follower 已经成为 uptodate 状态；
5. Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了。

数据同步的 4 种方式：

1、SNAP-全量同步

- 条件：peerLastZxid<minCommittedLog
- 说明：证明二者数据差异太大，follower 数据过于陈旧，leader 发送快照 SNAP 指令给 follower 全量同步数据，即 leader 将所有数据全量同步到 follower

2、DIFF-增量同步

- 条件：minCommittedLog<=peerLastZxid<=maxCommittedLog
- 说明：证明二者数据差异不大，follower 上有一些 leader 上已经提交的提议 proposal 未同步，此时需要增量提交这些提议即可

3、TRUNC-仅回滚同步

- 条件：peerLastZxid>minCommittedLog
- 说明：证明 follower 上有些提议 proposal 并未在 leader 上提交，follower 需要回滚到 zxid 为 minCommittedLog 对应的事务操作

4、TRUNC+DIFF-回滚+增量同步

- 条件：minCommittedLog<=peerLastZxid<=maxCommittedLog
- 说明：leader a 已经将事务 truncA 提交到本地事务日志中，但没有成功发起 proposal 协议进行投票就宕机了；然后集群中剔除原 leader a 重新选举出新 leader b，又提交了若干新的提议 proposal，然后原 leader a 重新服务又加入到集群中说明：此时 a,b 都有一些对方未提交的事务，若 b 是 leader, a 需要先回滚 truncA 然后增量同步新 leader b 上的数据。

12.2.7 分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。

对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

12.2.8 zk 的 session 机制