[Zookeeper 进阶之——典型应用场景（二）](http://www.cnblogs.com/haippy/archive/2012/07/23/2604556.html)

2012-07-23 20:57 by Haippy, 13123 阅读, 0 评论, [收藏](http://www.cnblogs.com/haippy/archive/2012/07/23/2604556.html), [编辑](http://i.cnblogs.com/EditPosts.aspx?postid=2604556)

本文是前一篇博文《[Zookeeper 进阶之——典型应用场景（一）](http://www.cnblogs.com/haippy/archive/2012/07/23/2603583.html)》的姊妹篇

**阅读指南——如何利用 Zookeeper 构建上层应用？**

本文将带你如何利用 Zookeeper 实现某些分布式应用所必需的高级功能。所有功能均可以在客户端按固定的模式实现，不需要 Zookeeper 的特殊支持，也希望 Zookeeper 社区能将这些具有固定实现模式的功能集成到 Zookeeper 客户端的程序库中，可以简化 Zookeeper 的使用并且还能使某些功能的实现标准化。

即便 Zookeeper 本身使用异步通知（*asynchronous* notifications），但却可以基于此构建同步的（*synchronous*）一致性原语，如队列和锁。你将看到 Zookeeper 实现这些功能是完全可能的，因为 Zookeeper 提供了强制的全序更新，并对外提供了保序接口。

注意下面的程序段试图采取最佳的编程实践，尤其是避免使用轮询（polling），定时器（timers）和其他任何可能造成“羊群效应（herd effect）”机制（“羊群效应”一般会带来网络流量的突增，限制系统的可扩展性）。

除了本文所列举的功能，我们还可以想象出其他很多实用的功能，比如可撤销的读写优先锁。本文提到的某些构建方式——比如锁，比较详细的阐述了使用 Zookepper 的关键点，其实你可以找到其他的例子，如事件处理和队列，但是，本节中的例子只是模拟相关的功能，在具体实践中需要考虑其他方面的因素。

**开箱即用的应用示例：命名服务，配置管理，组关系管理**

命名服务和配置管理是 Zookeeper 提供的最基本的应用，这两个功能可以直接用 Zookeeper 提供的 API 实现。

另外一个可以直接使用的功能是组关系管理，组在 Zookeeper 由一个 Znode 表示，组中的某个成员可以用组节点下的临时节点（Ephemeral Nodes）表示，当 Zookeeper 检测到节点故障时，节点成员中不正常的节点将会被自动地移除。

**屏障（Barriers）**

分布式系统使用屏障（*barriers*）来阻塞某一节点集的任务，直到满足特定的条件该节点集的所有节点才能正常向前推进，就像屏障一样，在当条件不满足时，屏障阻塞了任务的执行，只有当条件满足后屏障才会被拆除，各节点才能推进自己正在执行的任务。Zookeeper 中实现屏障时指定一个屏障节点（barrier node），如果屏障节点存在，屏障就会生效，下面是伪代码：

1. 客户端在屏障节点上调用 ZooKeeper API  **exists()，***watch 设置为 true*.
2. 如果 **exists()** 返回 false，屏障消失，客户端可以推进的自己的工作。
3. 否则, **exists()** 返回 true，客户端等待屏障节点上监听事件的到来。
4. 如果监听事件被触发，客户端重新执行 **exists( )，** 再一次重复上述 1-3 步，直到屏障节点被移除。

**（双屏障）Double Barriers**

双屏障（Double barriers）使得所有客户端在进入和结束某一计算任务时都会得到同步。当足够的进程processes（注：此处指节点）加入到屏障时，才启动任务，然后当任务完成时，离开屏障区，下面的代码段示意如何使用 Zookeeper 创建屏障节点。

伪代码中屏障节点用 *b* 表示，每个客户端进程（节点）  *p* 在进入屏障节点时注册事件，然后在离开时取消注册事件。进入屏障节点注册事件的代码如下表的 **Enter 程序段所示，** 在继续处理任务之前，它将等待客户端 *x* 进程的注册。（此处的 *x*由你针对自己的系统决定）

|  |  |
| --- | --- |
| **Enter** | **Leave** |
| 1. 创建名称为 *b+“/”+p 的 Znode 节点* 2. 设置监视：**exists(*b* + ‘‘/ready’’, true)** 3. 创建子节点：**create( *n*, EPHEMERAL)** 4. **L = getChildren(b, false)** 5. 如果L的孩子数目小于 *x 的*, 则等待监视事件 6. 否则 **create(b + ‘‘/ready’’, REGULAR)** | 1. **L = getChildren(b, false)** 2. 如果没有任何子节点，则退出。 3. 如果 *p* 是 L 中唯一节点，则 delete(n) 并退出。 4. 如果 *p 是 L 中的序号最低的节点，则等待 P 中的序号最高节点。* 5. 否则 依然存在，则 **delete(*n*)** 并继续等待L中的最低节点。 6. 跳转 1 |

在进入屏障时，所有的进程（节点）监视一个准备好的节点（屏障节点），并创建一个临时节点作为屏障节点的孩子。除了最后进入屏障的节点外，每个进程（节点）都等待屏障节点，直到第 5 行的条件出现。该进程（节点）创建第 x 个节点——即最后的进程（节点），它将会看到 x 个节点，并唤醒其他进程（节点），注意，所有的等待进程（节点）只是在退出的时候被唤醒，所以等待还是很高效的。

在退出屏障时，你不能设置 诸如 *ready* 的标志，因为你在等待进程节点退出，通过使用临时节点，进入屏障后失效的进程节点并不会阻止其他运行正确的节点完成任务。当进程节点准备推出屏障区时，它必须删除它的进程节点，并等待其他进程删除各自的进程节点。

当 b 没有的进程子节点时，进程（节点）就会退出屏障区。然而，为了效率起见，你可以使用序号最低的进程节点作为 ready 标志。所有其他准备退出屏障区的进程（节点）都监视序号最低的将要退出进程（节点）消失，序号最低的进程节点的拥有者则就等待其他任何一个节点的消失（选择序号最高进程节点）。这意味着除了最后的一个进程节点外，其他的每个进程节点被删除时只要唤醒一个进程节点即可，当它被删除时就会唤醒其他的进程节点。

**队列（Queues）**

分布式队列是通用的数据结构，为了在 Zookeeper 中实现分布式队列，首先需要指定一个 Znode 节点作为队列节点（queue node）， 各个分布式客户端通过调用 create() 函数向队列中放入数据，调用create()时节点路径名带"queue-"结尾，并设置顺序和临时（*sequence* and *ephemeral*）节点标志。 由于设置了节点的顺序标志，新的路径名具有以下字符串模式："\_path-to-queue-node\_/queue-X"，X 是唯一自增号。需要从队列中移除数据的客户端首先调用 **getChildren( )** 函数，同时在队列节点（queue node）上将 *watch* 设置为 true，并处理最小序号的节点（即从序号最小的节点中取数据）。客户端不需要再一次调用 **getChildren( )，**队列中的数据获取完。如果队列节点中没有任何子节点，读取队列的客户端需要等待队列的监视事件通知。

**Priority Queues**

为了实现优先队列，你在普通队列上只需要简单的改变两处地方，首先，在某一元素被加入队列时，路径名以 "queue-YY" 结尾，YY 表示优先级，YY越小优先级越高，其次，从队列中移除一个元素时，客户端需要使用最新的孩子节点列表，这意味着如果队列节点上监视通知被触发，客户端需要让先前获取的孩子节点列表无效。

**锁（Locks）**

完全分布式锁是全局同步的，这意味着在任何时刻没有两个客户端会同时认为它们都拥有相同的锁，使用 Zookeeper 可以实现分布式锁，和优先队列一样，我们需要首先定义一个锁节点（lock node）。

需要获得锁的客户端按照以下步骤来获取锁：

1. 调用 **create( )，**参数 pathname 为 "\_locknode\_/lock-"，并设置 *sequence*和 *ephemeral* 标志。
2. 在所节点（lock node）上调用 **getChildren( )** ，不需要设置监视标志。 (为了避免“羊群效应”).
3. 如果在第 1 步中创建的节点的路径具有最小的序号后缀，那么该客户端就获得了锁。
4. 客户端调用 **exists( )** ，并在锁目录路径中下一个最小序号的节点上设置监视标志。
5. 如果 **exists( )** 返回 false，跳转至第 2 步，否则，在跳转至第 2 步之前等待前一部路径上节点的通知消息。

解锁协议非常简单：需要释放锁的客户端只需要删除在第 1 步中创建的节点即可。

注意事项：

* 一个节点的删除只会导致一个客户端被唤醒，因为每个节点只被一个客户端监视，这避免了羊群效应。
* 没有轮询和超时。
* 根据你实现锁的方式不同，不同的实现可能会带来大量的锁竞争，锁中断，调试锁等等。

**Shared Locks**

在基本的锁协议之上，你只需要做一些小的改变就可以实现共享锁（shared locks）：

|  |  |
| --- | --- |
| **获取读锁：** | **获取写锁：** |
| 1. Call **create( )** to create a node with pathname "\_locknode\_/read-". This is the lock node use later in the protocol. Make sure to set both the *sequence* and *ephemeral*flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch*flag - this is important, as it avoids the herd effect. 3. If there are no children with a pathname starting with "write-" and having a lower sequence number than the node created in step **1**, the client has the lock and can exit the protocol. 4. Otherwise, call **exists( )**, with *watch* flag, set on the node in lock directory with pathname staring with "write-" having the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. 6. Otherwise, wait for a notification for the pathname from the previous step before going to step **2** | 1. Call **create( )** to create a node with pathname "\_locknode\_/write-". This is the lock node spoken of later in the protocol. Make sure to set both *sequence* and *ephemeral*flags. 2. Call **getChildren( )**on the lock node *without* setting the*watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a lower sequence number than the node created in step **1**, the client has the lock and the client exits the protocol. 4. Call **exists( ),** with *watch* flag set, on the node with the pathname that has the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**. |

**Recoverable Shared Locks**

对共享锁做一些细小的改变，我们就可以使共享锁变成可撤销的共享锁：

在第 1 步，在获取读者和写者的锁协议中，在调用 **create( ) 后，**立即调用**getData( )**，并设置监视。如果客户端稍后收到了它在第一步创建节点的通知，它会再一次在该节点上调用 **getData( )**，并设置监视，查找 “unlock” 串。该信号会通知客户端必须释放锁。这是因为，依据共享锁协议，你可以通过在锁节点（lock node）上调用**setData()（将“unlock”写入该节点）**请求拥有该锁的客户端放弃该锁 。

注意该协议要求锁的拥有者也同意释放该锁，该协定非常重要，尤其是锁的拥有者需要在释放该锁前做一些处理。 当然，你也可以通过约定“撤销者可以在锁的拥有者一段时间没有删除该锁的情况下删除该锁节点”来实现可撤销的共享锁。

**两阶段提交（Two-phased Commit）**

两阶段提交协议可以让分布式系统的所有客户端决定究竟提交某一事务或还是终止该事务。

在 Zookeeper 中，你可以让协调者（coordinator）创建事务节点，比如，"/app/Tx"，从而实现一个两阶段提交协议。 当协调者（coordinator）创建了子节点时，子节点内容是未定义的，由于每个事务参与方都会从协调者接收事务，参与方读取每个子节点并设置监视。然后每个参与方通过向与自身相关的 Znode 节点写入数据来投票“提交（commit）”或“中止（abort）”事务。一旦写入完成，其他的参与方会被通知到，当所有的参与方都投完票后，协调者就可以决定究竟是“提交（commit）”或“中止（abort）”事务。注意，如果某些参与方投票“中止”，节点是可以决定提前“中止”事务的。

该实现方法有趣的地方在于协调者的唯一作用是决定参与方的组（the group of sites），创建 Zookeeper 节点， 将事务传播到相应的参与方，实际上，Zookeeper 可以通过将消息写入事务节点来传播事务。

上述讨论的方法存在两个明显的缺点，一是消息的复杂性，复杂度为 O(n²)，另外一个是仅通过临时节点不能判断某些参与方是否失效，为了利用临时节点检测参与方是否失效，必须参与方创建该节点。

为了解决第一个问题，你可以将系统设置成只有一个协调者可以收到事务节点状态的变化，一旦协调者达成意见后通知其他参与方， 该方法可扩展性较强，但是速度很慢，因为所有的通信都指向协调者。

为了解决第二个问题，你可以让参与方把事务传播到参与方，并让每个参与方创建自己的临时节点。

**Leader 选举（Leader Election）**

Zookeeper 实现 Leader 选举简单做法是在创建代表 “proposals” 客户端的 Znode 节点时设置 **SEQUENCE|EPHEMERAL**标志。基本想法是创建一个节点，比如 "/election"，然后在创建子节点时"/election/n\_"设置标志 SEQUENCE|EPHEMERAL. 当设置顺序节点**SEQUENCE**标志时，Zookeeper 会在 "/election" 子节点的创建过程中自增子节点名称后缀的序号，最小后缀序号的 Znode 节点表示Leader。

然而，还没完，监视 Leader 失效也是非常重要的，当前的 Leader 失效后需要一个新的客户端起来接替旧的 Leader 的位置。一个简单的方式是让所有的应用进程监视当前序号最小的 Znode 节点， 并在当前 序号最小的 Znode 节点失效是检查他们是否为新的 Leader（注意当前序号最小的节点可能会随着 Leader 的消失而消失，他们可能是该Leader 节点的临时子节点）. 但是这会导致'羊群效应(herd effect)"：在当前 Leader 失效后，其他所有的进程（节点）将会收到通知，并在 "/election" 节点上执行 getChildren()来获取"/election"节点的子节点列表，如果客户端数目很大，它会使得Zookeeper服务器处理的操作次数急剧上升。为了避免羊群效应，客户端只需要监视 Znode 节点中的下一个节点就足够。如果某个客户端收到了它正在监视的节点消失的通知，它将成为新的 Leader，因为此时没有其它的 Znode 节点的序号比它小。所以这就避免了羊群效应，并且客户端也没有必要监视同一个最小的 Znode 节点。

以下是伪代码：

假设 ELECTION 成为Leader 选举应用的路径，对于想要成为 Leader 的 Volunteer而言：

1. 创建 Znode 节点 z，路径名称为"ELECTION/n\_"并设置 SEQUENCE 和 EPHEMERAL 标志。
2. 假设 C 是"ELECTION"的子节点集合，  i 是 z 节点的序号。
3. 监视节点 "ELECTION/n\_j" 的改变，j 是满足 j < i 最小的序号，n\_j 是 C 节点集合中的某个节点。

当收到 Znode 节点删除的通知时：

1. 假设 C 是 “ELECTION” 新的子节点集合。
2. 如果 z 是 C 中的最小节点，则执行 Leader 选举流程。
3. 否则，监视节点 "ELECTION/n\_j" 的改变，j 是满足 j < i 最小的序号，n\_j 是 C 节点集合中的某个节点。

注意，在子节点列表中没有先遣节点的 Znode 并不意味着该节点的创建者知道它就是当前的Leader，应用程序可能需要考虑创建一个单独的 Znode 来确认该 Leader 已经执行了选举流程。