# 使用zookeeper实现分布式锁

March 12, 2014

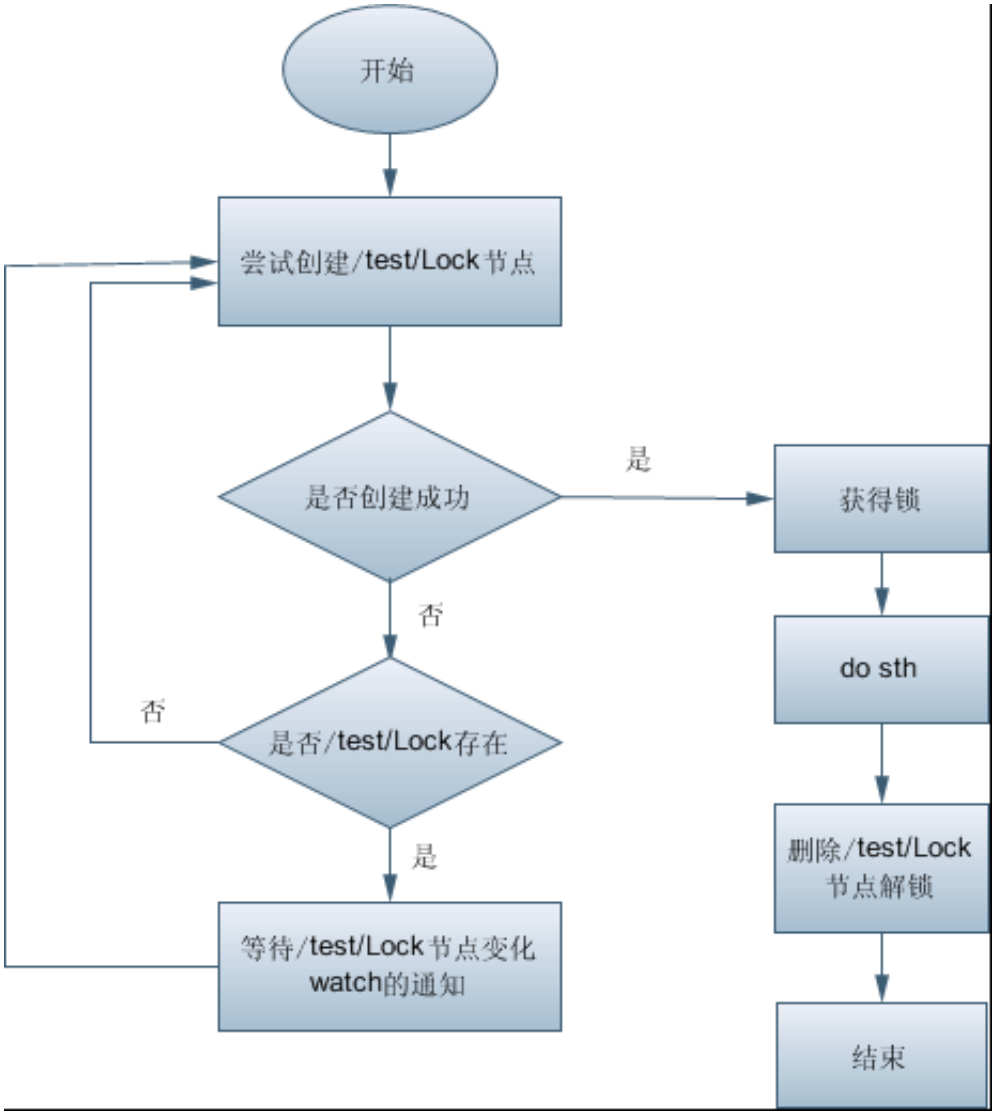
大家也许都很熟悉了多个线程或者多个进程间的共享锁的实现方式了，但是在分布式场景中我们会面临多个Server之间的锁的问题，实现的复杂度比较高。利用基于google chubby原理开发的开源的zookeeper，可以使得这个问题变得简单很多。下面介绍几种可能的实现方式，并且对比每种实现方式的优缺点。

## 1. 利用节点名称的唯一性来实现共享锁

ZooKeeper抽象出来的节点结构是一个和unix文件系统类似的小型的树状的目录结构。ZooKeeper机制规定：同一个目录下只能有一个唯一的文件名。例如：我们在Zookeeper目录/test目录下创建，两个客户端创建一个名为Lock节点，只有一个能够成功。

算法思路: 利用名称唯一性，加锁操作时，只需要所有客户端一起创建/test/Lock节点，只有一个创建成功，成功者获得锁。解锁时，只需删除/test/Lock节点，其余客户端再次进入竞争创建节点，直到所有客户端都获得锁。

基于以上机制，利用节点名称唯一性机制的共享锁算法流程如图所示：




该共享锁实现很符合我们通常多个线程去竞争锁的概念，利用节点名称唯一性的做法简明、可靠。

由上述算法容易看出，由于客户端会同时收到/test/Lock被删除的通知，重新进入竞争创建节点，故存在"惊群现象"。

使用该方法进行测试锁的性能列表如下：

| Lock机制的互斥测试(5PC) | | |
|---|---|---|
| 互斥进程数 | 平均进程完成时间（ms） | 所有进程都完成所用时间（ms） |
| 5 | 14.4 | 23 |
| 10 | 45.6 | 71 |
| 20 | 141.85 | 207 |
| 50 | 612 | 882 |

**总结** 这种方案的正确性和可靠性是ZooKeeper机制保证的，实现简单。缺点是会产生“惊群”效应，假如许多客户端在等待一把锁，当锁释放时候所有客户端都被唤醒，仅仅有一个客户端得到锁。

## 2. 利用临时顺序节点实现共享锁的一般做法

首先介绍一下，Zookeeper中有一种节点叫做顺序节点，故名思议，假如我们在/lock/目录下

创建节3个点，ZooKeeper集群会按照提起创建的顺序来创建节点，节点分别为/lock/0000000001、/lock/0000000002、/lock/0000000003。

ZooKeeper中还有一种名为临时节点的节点，临时节点由某个客户端创建，当客户端与ZooKeeper集群断开连接，则开节点自动被删除。

利用上面这两个特性，我们来看下获取实现分布式锁的基本逻辑：

- 客户端调用create()方法创建名为“*locknode*/guid-lock-”的节点，需要注意的是，这里节点的创建类型需要设置为EPHEMERAL_SEQUENTIAL。
- 客户端调用getChildren(“*locknode*”)方法来获取所有已经创建的子节点，同时在这个节点上注册上子节点变更通知的Watcher。
- 客户端获取到所有子节点path之后，如果发现自己在步骤1中创建的节点是所有节点中序号最小的，那么就认为这个客户端获得了锁。
- 如果在步骤3中发现自己并非是所有子节点中最小的，说明自己还没有获取到锁，就开始等待，直到下次子节点变更通知的时候，再进行子节点的获取，判断是否获取锁。

释放锁的过程相对比较简单，就是删除自己创建的那个子节点即可。

上面这个分布式锁的实现中，大体能够满足了一般的分布式集群竞争锁的需求。这里说的一般性场景是指集群规模不大，一般在10台机器以内。

不过，细想上面的实现逻辑，我们很容易会发现一个问题，步骤4，“即获取所有的子点，判断自己创建的节点是否已经是序号最小的节点”，这个过程，在整个分布式锁的竞争过程中，大量重复运行，并且绝大多数的运行结果都是判断出自己并非是序号最小的节点，从而继续等待下一次通知——这个显然看起来不怎么科学。客户端无端的接受到过多的和自己不相关的事件通知，这如果在集群规模大的时候，会对Server造成很大的性能影响，并且如果一旦同一时间有多个节点的客户端断开连接，这个时候，服务器就会像其余客户端发送大量的事件通知——这就是所谓的惊群效应。而这个问题的根源在于，没有找准客户端真正的关注点。

我们再来回顾一下上面的分布式锁竞争过程，它的核心逻辑在于：判断自己是否是所有节点中序号最小的。于是，很容易可以联想的到的是，每个节点的创建者只需要关注比自己序号小的那个节点。

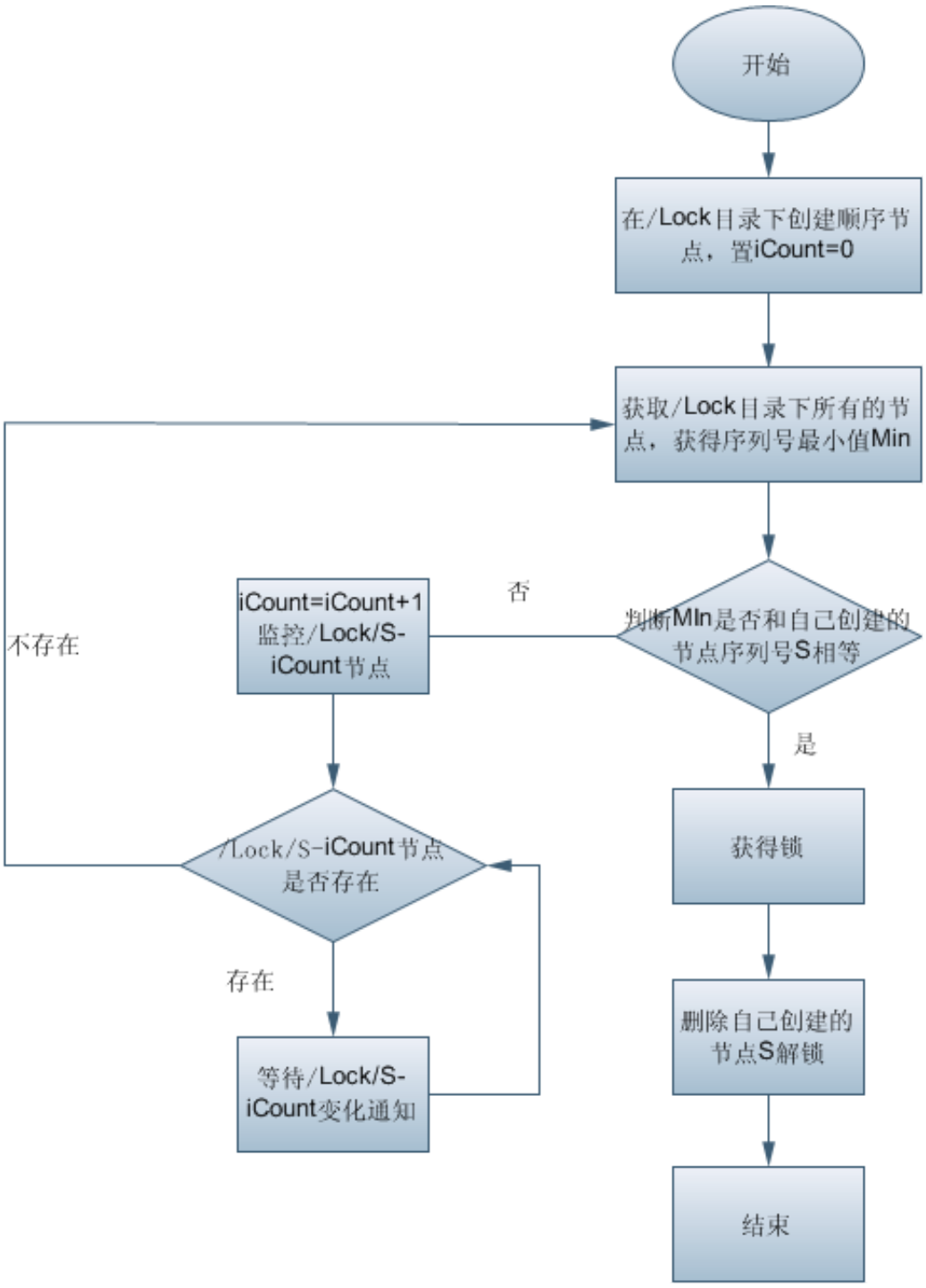
## 3. 利用临时顺序节点实现共享锁的改进实现

下面是改进后的分布式锁实现，和之前的实现方式唯一不同之处在于，这里设计成每个锁竞争者，只需要关注”*locknode*”节点下序号比自己小的那个节点是否存在即可。

算法思路：对于加锁操作，可以让所有客户端都去/lock目录下创建临时顺序节点，如果创建的客户端发现自身创建节点序列号是/lock/目录下最小的节点，则获得锁。否则，监视比自己创建节点的序列号小的节点（比自己创建的节点小的最大节点），进入等待。

对于解锁操作，只需要将自身创建的节点删除即可。

具体算法流程如下图所示:




使用上述算法进行测试的的结果如下表所示：

| Lock机制的互斥测试 | | |
|---|---|---|
| 互斥进程数 | 平均进程完成时间（ms） | 所有进程都完成所有时间(ms) |
| 5 | 3.56 | 7 |
| 10 | 5.32 | 8.6 |
| 50 | 47.24 | 78.4 |

该算法只监控比自身创建节点序列号小(比自己小的最大的节点)的节点，在当前获得锁的节点释放锁的时候没有“惊群”。

**总结** 利用临时顺序节点来实现分布式锁机制其实就是一种按照创建顺序排队的实现。这种方案效率高，避免了“惊群”效应，多个客户端共同等待锁，当锁释放时只有一个客户端会被唤醒。

### 4. 使用menagerie

其实就是对方案3的一个封装，不用自己写代码了。直接拿来用就可以了。

> menagerie基于Zookeeper实现了java.util.concurrent包的一个分布式版本。这个封装是更大粒度上对各种分布式一致性使用场景的抽象。其中最基础和常用的是一个分布式锁的实现： org.menagerie.locks.ReentrantZkLock，通过ZooKeeper的全局有序的特性和EPHEMERAL_SEQUENTIAL类型znode的支持，实现了分布式锁。具体做法是：不同的client上每个试图获得锁的线程，都在相同的basepath下面创建一个EPHEMERAL_SEQUENTIAL的node。EPHEMERAL表示要创建的是临时znode，创建连接断开时会自动删除； SEQUENTIAL表示要自动在传入的path后面缀上一个自增的全局唯一后缀,作为最终的path。因此对不同的请求ZK会生成不同的后缀，并分别返回带了各自后缀的path给各个请求。因为ZK全局有序的特性，不管client请求怎样先后到达，在ZKServer端都会最终排好一个顺序，因此自增后缀最小的那个子节点，就对应第一个到达ZK的有效请求。然后client读取basepath下的所有子节点和ZK返回给自己的path进行比较，当发现自己创建的sequential node的后缀序号排在第一个时，就认为自己获得了锁；否则的话，就认为自己没有获得锁。这时肯定是有其他并发的并且是没有断开的client/线程先创建了node。

menagerie地址：https://github.com/openUtility/menagerie

## 参考资料以及推荐阅读


1. ZooKeeper数据模型
2. zookeeper分布式锁避免羊群效应（Herd Effect）
3. Zookeeper Client简介
4. 李欣：ZooKeeper在携程的使用及前景

## Related Posts


- 24 Jul 2015 » Tomcat调优
- 22 Jul 2015 » 记一次MySQL主从同步错误处理
- 03 Jul 2015 » Metric监控系统