### [ZooKeeper示例 分布式锁](http://coolxing.iteye.com/blog/1871630)

**http://coolxing.iteye.com/blog/1871630**

**场景描述**

在分布式应用, 往往存在多个进程提供同一服务. 这些进程有可能在相同的机器上, 也有可能分布在不同的机器上. 如果这些进程共享了一些资源, 可能就需要分布式锁来锁定对这些资源的访问.  
本文将介绍如何利用zookeeper实现分布式锁.

**思路**

进程需要访问共享数据时, 就在"/locks"节点下创建一个sequence类型的子节点, 称为thisPath. 当thisPath在所有子节点中最小时, 说明该进程获得了锁. 进程获得锁之后, 就可以访问共享资源了. 访问完成后, 需要将thisPath删除. 锁由新的最小的子节点获得.  
有了清晰的思路之后, 还需要补充一些细节. 进程如何知道thisPath是所有子节点中最小的呢? 可以在创建的时候, 通过getChildren方法获取子节点列表, 然后在列表中找到排名比thisPath前1位的节点, 称为waitPath, 然后在waitPath上注册监听, 当waitPath被删除后, 进程获得通知, 此时说明该进程获得了锁.

**实现**

以一个DistributedClient对象模拟一个进程的形式, 演示zookeeper分布式锁的实现.

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** DistributedClient {
2. // 超时时间
3. **private** **static** **final** **int** SESSION\_TIMEOUT = 5000;
4. // zookeeper server列表
5. **private** String hosts = "localhost:4180,localhost:4181,localhost:4182";
6. **private** String groupNode = "locks";
7. **private** String subNode = "sub";
9. **private** ZooKeeper zk;
10. // 当前client创建的子节点
11. **private** String thisPath;
12. // 当前client等待的子节点
13. **private** String waitPath;
15. **private** CountDownLatch latch = **new** CountDownLatch(1);
17. /\*\*
18. \* 连接zookeeper
19. \*/
20. **public** **void** connectZookeeper() **throws** Exception {
21. zk = **new** ZooKeeper(hosts, SESSION\_TIMEOUT, **new** Watcher() {
22. **public** **void** process(WatchedEvent event) {
23. **try** {
24. // 连接建立时, 打开latch, 唤醒wait在该latch上的线程
25. **if** (event.getState() == KeeperState.SyncConnected) {
26. latch.countDown();
27. }
29. // 发生了waitPath的删除事件
30. **if** (event.getType() == EventType.NodeDeleted && event.getPath().equals(waitPath)) {
31. doSomething();
32. }
33. } **catch** (Exception e) {
34. e.printStackTrace();
35. }
36. }
37. });
39. // 等待连接建立
40. latch.await();
42. // 创建子节点
43. thisPath = zk.create("/" + groupNode + "/" + subNode, **null**, Ids.OPEN\_ACL\_UNSAFE,
44. CreateMode.EPHEMERAL\_SEQUENTIAL);
46. // wait一小会, 让结果更清晰一些
47. Thread.sleep(10);
49. // 注意, 没有必要监听"/locks"的子节点的变化情况
50. List<String> childrenNodes = zk.getChildren("/" + groupNode, **false**);
52. // 列表中只有一个子节点, 那肯定就是thisPath, 说明client获得锁
53. **if** (childrenNodes.size() == 1) {
54. doSomething();
55. } **else** {
56. String thisNode = thisPath.substring(("/" + groupNode + "/").length());
57. // 排序
58. Collections.sort(childrenNodes);
59. **int** index = childrenNodes.indexOf(thisNode);
60. **if** (index == -1) {
61. // never happened
62. } **else** **if** (index == 0) {
63. // inddx == 0, 说明thisNode在列表中最小, 当前client获得锁
64. doSomething();
65. } **else** {
66. // 获得排名比thisPath前1位的节点
67. **this**.waitPath = "/" + groupNode + "/" + childrenNodes.get(index - 1);
68. // 在waitPath上注册监听器, 当waitPath被删除时, zookeeper会回调监听器的process方法
69. zk.getData(waitPath, **true**, **new** Stat());
70. }
71. }
72. }
74. **private** **void** doSomething() **throws** Exception {
75. **try** {
76. System.out.println("gain lock: " + thisPath);
77. Thread.sleep(2000);
78. // do something
79. } **finally** {
80. System.out.println("finished: " + thisPath);
81. // 将thisPath删除, 监听thisPath的client将获得通知
82. // 相当于释放锁
83. zk.delete(**this**.thisPath, -1);
84. }
85. }
87. **public** **static** **void** main(String[] args) **throws** Exception {
88. **for** (**int** i = 0; i < 10; i++) {
89. **new** Thread() {
90. **public** **void** run() {
91. **try** {
92. DistributedClient dl = **new** DistributedClient();
93. dl.connectZookeeper();
94. } **catch** (Exception e) {
95. e.printStackTrace();
96. }
97. }
98. }.start();
99. }
101. Thread.sleep(Long.MAX\_VALUE);
102. }
103. }

**思考**

思维缜密的朋友可能会想到, 上述的方案并不安全. 假设某个client在获得锁之前挂掉了, 由于client创建的节点是ephemeral类型的, 因此这个节点也会被删除, 从而导致排在这个client之后的client提前获得了锁. 此时会存在多个client同时访问共享资源.  
如何解决这个问题呢? 可以在接到waitPath的删除通知的时候, 进行一次确认, 确认当前的thisPath是否真的是列表中最小的节点.

**Java代码  [收藏代码](javascript:void())**

1. // 发生了waitPath的删除事件
2. **if** (event.getType() == EventType.NodeDeleted && event.getPath().equals(waitPath)) {
3. // 确认thisPath是否真的是列表中的最小节点
4. List<String> childrenNodes = zk.getChildren("/" + groupNode, **false**);
5. String thisNode = thisPath.substring(("/" + groupNode + "/").length());
6. // 排序
7. Collections.sort(childrenNodes);
8. **int** index = childrenNodes.indexOf(thisNode);
9. **if** (index == 0) {
10. // 确实是最小节点
11. doSomething();
12. } **else** {
13. // 说明waitPath是由于出现异常而挂掉的
14. // 更新waitPath
15. waitPath = "/" + groupNode + "/" + childrenNodes.get(index - 1);
16. // 重新注册监听, 并判断此时waitPath是否已删除
17. **if** (zk.exists(waitPath, **true**) == **null**) {
18. doSomething();
19. }
20. }
21. }

另外, 由于thisPath和waitPath这2个成员变量会在多个线程中访问, 最好将他们声明为volatile, 以防止出现线程可见性问题.

**另一种思路**

下面介绍一种更简单, 但是不怎么推荐的解决方案.  
每个client在getChildren的时候, 注册监听子节点的变化. 当子节点的变化通知到来时, 再一次通过getChildren获取子节点列表, 判断thisPath是否是列表中的最小节点, 如果是, 则执行资源访问逻辑.

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** DistributedClient2 {
2. // 超时时间
3. **private** **static** **final** **int** SESSION\_TIMEOUT = 5000;
4. // zookeeper server列表
5. **private** String hosts = "localhost:4180,localhost:4181,localhost:4182";
6. **private** String groupNode = "locks";
7. **private** String subNode = "sub";
9. **private** ZooKeeper zk;
10. // 当前client创建的子节点
11. **private** **volatile** String thisPath;
13. **private** CountDownLatch latch = **new** CountDownLatch(1);
15. /\*\*
16. \* 连接zookeeper
17. \*/
18. **public** **void** connectZookeeper() **throws** Exception {
19. zk = **new** ZooKeeper(hosts, SESSION\_TIMEOUT, **new** Watcher() {
20. **public** **void** process(WatchedEvent event) {
21. **try** {
22. // 连接建立时, 打开latch, 唤醒wait在该latch上的线程
23. **if** (event.getState() == KeeperState.SyncConnected) {
24. latch.countDown();
25. }
27. // 子节点发生变化
28. **if** (event.getType() == EventType.NodeChildrenChanged && event.getPath().equals("/" + groupNode)) {
29. // thisPath是否是列表中的最小节点
30. List<String> childrenNodes = zk.getChildren("/" + groupNode, **true**);
31. String thisNode = thisPath.substring(("/" + groupNode + "/").length());
32. // 排序
33. Collections.sort(childrenNodes);
34. **if** (childrenNodes.indexOf(thisNode) == 0) {
35. doSomething();
36. }
37. }
38. } **catch** (Exception e) {
39. e.printStackTrace();
40. }
41. }
42. });
44. // 等待连接建立
45. latch.await();
47. // 创建子节点
48. thisPath = zk.create("/" + groupNode + "/" + subNode, **null**, Ids.OPEN\_ACL\_UNSAFE,
49. CreateMode.EPHEMERAL\_SEQUENTIAL);
51. // wait一小会, 让结果更清晰一些
52. Thread.sleep(10);
54. // 监听子节点的变化
55. List<String> childrenNodes = zk.getChildren("/" + groupNode, **true**);
57. // 列表中只有一个子节点, 那肯定就是thisPath, 说明client获得锁
58. **if** (childrenNodes.size() == 1) {
59. doSomething();
60. }
61. }
63. /\*\*
64. \* 共享资源的访问逻辑写在这个方法中
65. \*/
66. **private** **void** doSomething() **throws** Exception {
67. **try** {
68. System.out.println("gain lock: " + thisPath);
69. Thread.sleep(2000);
70. // do something
71. } **finally** {
72. System.out.println("finished: " + thisPath);
73. // 将thisPath删除, 监听thisPath的client将获得通知
74. // 相当于释放锁
75. zk.delete(**this**.thisPath, -1);
76. }
77. }
79. **public** **static** **void** main(String[] args) **throws** Exception {
80. **for** (**int** i = 0; i < 10; i++) {
81. **new** Thread() {
82. **public** **void** run() {
83. **try** {
84. DistributedClient2 dl = **new** DistributedClient2();
85. dl.connectZookeeper();
86. } **catch** (Exception e) {
87. e.printStackTrace();
88. }
89. }
90. }.start();
91. }
93. Thread.sleep(Long.MAX\_VALUE);
94. }
95. }

为什么不推荐这个方案呢? 是因为每次子节点的增加和删除都要广播给所有client, client数量不多时还看不出问题. 如果存在很多client, 那么就可能导致广播风暴--过多的广播通知阻塞了网络. 使用第一个方案, 会使得通知的数量大大下降. 当然第一个方案更复杂一些, 复杂的方案同时也意味着更容易引进bug.