

分布式利器Zookeeper（三）



张丰哲 (/u/cb569cce501b) ✓已关注

2017.05.17 07:47 字数 1113 阅读 547 评论 4 喜欢 12 赞赏 1

(/u/cb569cce501b)

前言

《分布式利器Zookeeper（一）》(<http://www.jianshu.com/p/3dfd63811e20>)

《分布式利器Zookeeper（二）:分布式锁》(<http://www.jianshu.com/p/d8bbbed558ec7>)

本篇博客是分布式利器Zookeeper系列的最后一篇，涉及的话题是：Zookeeper分布式锁的代码实现、zkclient的使用、Curator框架介绍等。

Zookeeper分布式锁的代码实现

在上一篇博客中，从思路上已经分析了Zookeeper如何帮助我们实现分布式锁，我们直接来看代码：

```
public class DistributedClient {  
  
    private static final String root = "/root";  
    private static final String lock = "lock_";  
    private String thisPath;  
    private String waitPath;  
    private static final int sessionTimeout = 5000;  
    private ZooKeeper zooKeeper;  
    private CountDownLatch countDownLatch = new CountDownLatch(1);  
  
    public void doTask() throws KeeperException, InterruptedException {  
  
        System.out.println(Thread.currentThread().getName() + " 获取到锁,开始执行任务...");  
        try {  
            Thread.sleep(new Random().nextInt(1000));  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        } finally { //释放锁  
            zooKeeper.delete(root + "/" + this.thisPath,-1);  
        }  
        System.out.println(Thread.currentThread().getName() + " 任务执行完毕...");  
    }  
}
```

分布式客户端

^

+

🔖

🔗

<http://www.jianshu.com/p/baf738d35614>

1/8

```

public void lock(String connectionString){
    try {
        zooKeeper = new ZooKeeper(connectionString, sessionTimeout, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                if(event.getState() == Event.KeeperState.SyncConnected){
                    if(event.getType() == Event.EventType.None){
                        countDownLatch.countDown();
                    }else if( event.getType() == Event.EventType.NodeDeleted){
                        try {
                            List<String> children1 = zooKeeper.getChildren(root, false);
                            Collections.sort(children1);
                            int index = children1.indexOf(thisPath);
                            if(index == 0){//正常获得锁
                                doTask();
                            }else if(index > 0){
                                waitPath = children1.get(index - 1 );
                                if(zooKeeper.exists(root + "/" + waitPath,true) == null){ //获取锁
                                    doTask();
                                }
                            }
                        } catch (KeeperException e) {
                            e.printStackTrace();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        });
        countDownLatch.await();
    }
}

```

获取分布式锁的方法lock:初始化ZK

```

//创建节点
this.thisPath = zooKeeper.create(root + "/" + lock , new byte[0], ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
System.out.println(Thread.currentThread().getName() + " create " + this.thisPath);

this.thisPath = thisPath.substring(thisPath.lastIndexOf("/") + 1);

//获取子节点 无需监控 避免羊群效应
final List<String> children = zooKeeper.getChildren(root, false);
if(children.size() == 1){
    if(children.get(0).equals(this.thisPath)){//获得锁
        doTask();
    }
}else{
    //排序
    Collections.sort(children);

    if(children.get(0).equals(this.thisPath)){//获得锁
        doTask();
    }else{
        //监控比它小的节点路径
        int index = children.indexOf(this.thisPath);
        if(index >= 0){
            this.waitPath = children.get(index - 1);
            zooKeeper.exists(root + "/" + waitPath,true);
        }
    }
}
}

```

获取分布式锁的方法lock:创建临时节点与判断最小路径

```

public static void main(String[] args) throws InterruptedException {
    for(int i = 0 ; i < 10 ; i++){
        new Thread(new Runnable() {
            @Override
            public void run() {
                DistributedClient distributedClient = new DistributedClient();
                try {
                    Thread.sleep(new Random().nextInt(1000));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                distributedClient.lock("192.168.99.121:2181,192.168.99.122:2181,192.168.99.123:2181");
            }
        }).start();
    }
    Thread.sleep(60000);
}

```

main测试



```
Thread-0 create /root/lock_0000000560
Thread-8 create /root/lock_0000000563
Thread-5 create /root/lock_0000000561
Thread-4 create /root/lock_0000000562
Thread-6 create /root/lock_0000000565
Thread-2 create /root/lock_0000000564
Thread-0 获取到锁,开始执行任务...
Thread-3 create /root/lock_0000000566
Thread-7 create /root/lock_0000000567
Thread-1 create /root/lock_0000000569
Thread-9 create /root/lock_0000000568
Thread-0 任务执行完毕...
Thread-5-EventThread 获取到锁,开始执行任务...
Thread-5-EventThread 任务执行完毕...
Thread-4-EventThread 获取到锁,开始执行任务...
Thread-4-EventThread 任务执行完毕...
Thread-8-EventThread 获取到锁,开始执行任务...
Thread-8-EventThread 任务执行完毕...
Thread-2-EventThread 获取到锁,开始执行任务...
Thread-2-EventThread 任务执行完毕...
Thread-6-EventThread 获取到锁,开始执行任务...
Thread-6-EventThread 任务执行完毕...
Thread-3-EventThread 获取到锁,开始执行任务...
Thread-3-EventThread 任务执行完毕...
Thread-7-EventThread 获取到锁,开始执行任务...
Thread-7-EventThread 任务执行完毕...
Thread-9-EventThread 获取到锁,开始执行任务...
Thread-9-EventThread 任务执行完毕...
Thread-1-EventThread 获取到锁,开始执行任务...
Thread-1-EventThread 任务执行完毕...
```

运行结果

需要注意的是，即便监控到了比自己序号小的节点的删除Watcher，也需要再次确认下！

从结果上，看的很清楚，各个线程有序获得锁。

zkclient

zkclient是在zookeeper原生API基础上做了一点封装，简化了ZK的复杂性。

来看代码：

```
public class ZKClientBase {
    public static void main(String[] args) {
        ZkClient zkClient = new ZkClient(
            new zkConnection("192.168.99.121:2181,192.168.99.122:2181,192.168.99.123:2181"), 50000);

        //递归创建节点 缺点在于并没有为节点指定VALUE
        zkClient.createPersistent("/test/zkclient", true);

        zkClient.writeData("/test", "test");
        System.out.println("/test value is " + zkClient.readData("/test"));

        zkClient.createPersistent("/test/zkclient/c1");
        zkClient.createPersistent("/test/zkclient/c2");
        List<String> children = zkClient.getChildren("/test/zkclient");
        for(String s : children){
            System.out.println(s);
        }
        System.out.println("size : " + zkClient.countChildren("/test/zkclient"));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //递归删除
        zkClient.deleteRecursive("/test");
    }
}
```

基于zkclient的增删改查



我们观察下zkclient的使用，和以前基于zookeeper的原生API有哪些区别呢？

第一，原生API需要我们利用CountDownLatch来确保ZK的初始化，现在zkclient帮助我们屏蔽掉了这个细节

第二，原生API是不可以递归创建节点的，而zkclient可以帮助我们递归创建不存在的父节点，还可以递归删除

第三，支持序列化操作，上面的代码你大概可以看出一些端倪，就是从操作byte[]到操作String了。（事实上，在zkclient中你只需要实现ZkSerializer接口，就可以完成Object到byte[]的转换，虽然如此，但是实际开发中，利用JSON也挺好的！）

第四，还有最重要的一点就是，zkclient将对节点的操作和对节点的监控分离开了，在原生API中二者是耦合在一起的！从思想上来看，便于理解；从代码上来看，也简洁些（如果写在一起，头都大了）；更加方便的是，zkclient替我们完成了重复watch的功能！

```
ZkClient zkClient = new ZkClient(  
    new ZkConnection("192.168.99.121:2181,192.168.99.122:2181,192.168.99.123:2181"), 50000);  
  
// 监控子节点的增加/删除  
zkClient.subscribeChildChanges("/node", new IZkChildListener() {  
    @Override  
    public void handleChildChange(String s, List<String> list) throws Exception {  
        System.out.println("subscribeChildChanges : " + s + " , " + list);  
    }  
});  
  
// 监控节点的数据改变/删除  
zkClient.subscribeDataChanges("/node", new IZkDataListener() {  
    @Override  
    public void handleDataChange(String s, Object o) throws Exception {  
        System.out.println("subscribeDataChanges : " + s + " , " + o);  
    }  
  
    @Override  
    public void handleDataDeleted(String s) throws Exception {  
        System.out.println("subscribeDataChanges : " + s);  
    }  
});
```

watch订阅机制

看到没有，是不是有点像MQ的订阅机制，非常好用！【但是也有点不太完美，子节点的数据变更为什么没有监控呢，这有点不符合人性啊！还好有Curator...】

但是呢，我们知道ZK是有很多应用场景的，比如实现分布式锁，zkclient并没有替我们进行封装，但是Curator框架可以帮助我们做到！

Curator

为了更好实现Java操作Zookeeper服务器，后来出现Curator框架，功能非常强大，目前已经是Apache的顶级项目，有很多丰富的特性，比如session超时重连，主从选举，分布式计数器，分布式锁等，非常有利于Zookeeper复杂场景下的开发。

POM文件：

```
<!-- https://mvnrepository.com/artifact/org.apache.curator/curator-framework -->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>2.8.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.curator/curator-recipes -->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>2.8.0</version>
</dependency>
```

pom.xml

增删改查：

```
public static final String connectString = "192.168.99.121:2181,192.168.99.122:2181,192.168.99.123:2181";
public static final int sessionTimeout = 50000;
public static void main(String[] args) throws Exception {

    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,10);
    CuratorFramework curatorFramework = CuratorFrameworkFactory.builder()
        .connectString(connectString).sessionTimeoutMs(sessionTimeout).retryPolicy(retryPolicy).build();

    //别忘了
    curatorFramework.start();
    //创建节点
    curatorFramework.create().creatingParentsIfNeeded()
        .withMode(CreateMode.PERSISTENT).forPath("/curator/node", "node".getBytes("UTF-8"));
    //获取数据
    System.out.println(new String(curatorFramework.getData().forPath("/curator/node"), "UTF-8"));
    //设置数据
    curatorFramework.setData().forPath("/curator", "curator".getBytes("UTF-8"));
    //递归删除节点
    curatorFramework.create().creatingParentsIfNeeded()
        .withMode(CreateMode.PERSISTENT).forPath("/curator/tmp/t1/t2", null);
    curatorFramework.delete().deletingChildrenIfNeeded().forPath("/curator/tmp");

    //检测节点是否存在
    Stat stat = curatorFramework.checkExists().forPath("/curator");
    System.out.println(stat);
    curatorFramework.close();
}
```

curator基本的API操作

Curator框架使用链式编程风格，易读性很强！

注意，不论是原生的API，还是基于zkclient的API，都是提供的connectTimeout，而Curator提供了sessionTimeout，功能很强大。

```
public static final String connectString = "192.168.99.121:2181,192.168.99.122:2181,192.168.99.123:2181";
public static final int sessionTimeout = 50000;
public static void main(String[] args) throws Exception {

    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,10);
    CuratorFramework curatorFramework = CuratorFrameworkFactory.builder()
        .connectString(connectString).sessionTimeoutMs(sessionTimeout).retryPolicy(retryPolicy).build();

    //别忘了
    curatorFramework.start();

    Executor e = Executors.newFixedThreadPool(10);
    curatorFramework.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT).inBackground(new BackgroundCallback() {
        @Override
        public void processResult(CuratorFramework curatorFramework, CuratorEvent curatorEvent) throws Exception {
            System.out.println(new String(curatorFramework.getData().forPath(curatorEvent.getPath())));
            System.out.println(curatorEvent.getPath());
            System.out.println(curatorEvent.getType());
        }
    }, e).forPath("/curator/t1", "t1".getBytes());

    Thread.sleep(10000);
}
```

异步回调

无论是原生的API，还是zkclient，都是支持异步回调的，但是Curator框架在支持异步回调的同时，增加了线程池供我们优化！

```

RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,10);
CuratorFramework curatorFramework = CuratorFrameworkFactory.builder()
    .connectString(connectString).sessionTimeoutMs(sessionTimeout).retryPolicy(retryPolicy).build();
//对于本身节点的 修改/创建
final NodeCache nodeCache = new NodeCache(curatorFramework,"/curator/t2",false);
curatorFramework.start();
nodeCache.start(true);
Executor executor = Executors.newFixedThreadPool(10);
nodeCache.getListenable().addListener(new NodeCacheListener() {
    @Override
    public void nodeChanged() throws Exception {
        if(nodeCache.getCurrentData() != null){
            System.out.println("-----");
            System.out.println(nodeCache.getCurrentData().getPath());
            System.out.println(new String(nodeCache.getCurrentData().getData()));
        }
    }
},executor);
curatorFramework.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT)
    .forPath("/curator/t2","b".getBytes());
Thread.sleep(5000);
curatorFramework.setData().forPath("/curator/t2","a".getBytes());
Thread.sleep(5000);
curatorFramework.delete().forPath("/curator/t2");
Thread.sleep(10000);

```

NodeCacheListener

```

public static final String connectString = "192.168.99.121:2181,192.168.99.122:2181,192.168.99.123:2181";
public static final int sessionTimeout = 50000;
public static void main(String[] args) throws Exception {
    RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,10);
    CuratorFramework curatorFramework = CuratorFrameworkFactory.builder()
        .connectString(connectString).sessionTimeoutMs(sessionTimeout).retryPolicy(retryPolicy).build();
    //对于子节点的 修改/创建/更新
    final PathChildrenCache pathChildrenCache = new PathChildrenCache(curatorFramework,"/curator",true);

    curatorFramework.start();
    pathChildrenCache.start();

    Executor executor = Executors.newFixedThreadPool(10);

    pathChildrenCache.getListenable().addListener(new PathChildrenCacheListener() {
        @Override
        public void childEvent(CuratorFramework curatorFramework, PathChildrenCacheEvent pathChildrenCacheEvent) throws Exception {
            PathChildrenCacheEvent.Type type = pathChildrenCacheEvent.getType();
            switch (type){
                case CHILD_ADDED:
                    System.out.println("add path " + pathChildrenCacheEvent.getData().getPath());
                    break;
                case CHILD_REMOVED:
                    System.out.println("remove path " + pathChildrenCacheEvent.getData().getPath());
                    break;
                case CHILD_UPDATED:
                    System.out.println("update path " + pathChildrenCacheEvent.getData().getPath());
                    break;
            }
        }
    },executor);

    curatorFramework.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT)
        .forPath("/curator/t3","b".getBytes());
    Thread.sleep(5000);
    curatorFramework.setData().forPath("/curator/t3","a".getBytes());
    Thread.sleep(5000);
    curatorFramework.delete().forPath("/curator/t3");
    Thread.sleep(10000);
}

```

PathChildrenCacheListener

对于Curator而言，为了解决重复Watch的问题，它引入了一种全新的思想：Cache与ZK SERVER比对的机制。不论是原生的API，还是基于ZKCLIENT的，其实它们解决思路都是重复注册！

思路决定出路！Curator通过事件驱动将客户端的Cache与ZK SERVER的数据比对，就自然而然的解决了重复WATCH的功能！为什么Curator能成为Apache的顶级项目呢，我想大概就是因为它的与众不同的设计思想！

在Curator中，有2种Listener，一个是监控节点的NodeCacheListener，一个是监控子节点的PathChildrenCacheListener。PathChildrenCacheListener可以监控子节点的新增、修改、删除，非常好用！

好了，到这里，准备结束这个系列了（其实还有一些内容没有涉及，比如Curator的分布式锁、分布式barrier的介绍等，以后有空再分享，暂且保留下，哈哈）！





张丰哲 (/u/cb569cce501b) ♂

写了 63893 字，被 2144 人关注，获得了 1674 个喜欢
(/u/cb569cce501b)

✓ 已关注

资深Java工程师 51CTO博客【2014-2016】：http://zhangfengzhe.blog.51cto.com/

好好学习，天天赞赏~

赞赏支持




♡ 喜欢 | 12



更多分享


(http://cwb.assets.jianshu.io/notes/images/123663C



写下你的评论...

4条评论 只看作者

按喜欢排序 按时间正序 按时间倒序




亮_ca61 (/u/685a21d094b4)
2楼 · 2017.06.01 19:06
(/u/685a21d094b4)
三篇zk文章都看完了，写得很好，我未接触过zk的也能看懂□□

👍 赞 💬 回复

张丰哲 (/u/cb569cce501b)： 😊
2017.06.01 21:42 💬 回复

✎ 添加新评论



沉淀_0x0 (/u/01fc8cfd7b19)
3楼 · 2017.08.25 17:43
(/u/01fc8cfd7b19)
你上一篇里的思路理的挺好的，不过你这里的实现有点没看懂啊：没有获取到锁时，是通过什么来阻塞当前线程的？

👍 赞 💬 回复

张丰哲 (/u/cb569cce501b)： 具体思路在上一篇中，你可以参考下。
“没有获取到锁时，是通过什么来阻塞当前线程的？”大概是这样的，如果一个线程没有拿到锁，那么实质上这个线程并不会执行，而是进入了一个watch状态中，一旦拿到锁的线程执行完毕，释放锁后就会触发watch机制,相当于通知下一个线程可以执行了。
2017.08.26 21:32 💬 回复

✎ 添加新评论

被以下专题收入，发现更多相似内容




+ 收入我的专题

⬆

+

🔖

🔗

-  [Java学习笔记 \(/c/04cb7410c597?utm_source=desktop&utm_medium=notes-included-collection\)](#)
-  [程序员 \(/c/NEt52a?utm_source=desktop&utm_medium=notes-included-collection\)](#)
-  [Zookeeper \(/c/057bb7fb1243?utm_source=desktop&utm_medium=notes-included-collection\)](#)

^

+

🔖

🔗