链滴

Curator 使用详解

作者：iMLe0n

原文链接：https://ld246.com/article/1543471669799

来源网站：链滴

Zookeeper客户端Curator使用详解

简介

Curator是Netflix公司开源的一套zookeeper客户端框架，解决了很多Zookeeper客户端非常底层的节开发工作，包括连接重连、反复注册Watcher和NodeExistsException异常等等。Patrixck Hunt（ookeeper）以一句“Guava is to Java that Curator to Zookeeper”给Curator予高度评价。

引子和趣闻：

Zookeeper名字的由来是比较有趣的，下面的片段摘抄自《从PAXOS到ZOOKEEPER分布式一致性原与实践》一

书：

Zookeeper最早起源于雅虎的研究院的一个研究小组。在当时，研究人员发现，在雅虎内部很多大型系统需要依赖一个类似的系统进行分布式协调，但是这些系统往往存在分布式单点问题。所以雅虎的发人员就试图开发一个通用的无单点问题的分布式协调框架。在立项初期，考虑到很多项目都是用动的名字来命名的(例如著名的Pig项目)，雅虎的工程师希望给这个项目也取一个动物的名字。时任研究的首席科学家Raghu Ramakrishnan开玩笑说：再这样下去，我们这儿就变成动物园了。此话一出，家纷纷表示就叫动物园管理员吧——因为各个以动物命名的分布式组件放在一起，雅虎的整个分布式统看上去就像一个大型的动物园了，而Zookeeper正好用来进行分布式环境的协调——于是，Zookeper的名字由此诞生了。

Curator无疑是Zookeeper客户端中的瑞士军刀，它译作"馆长"或者''管理者''，不知道是不是开发小有意而为之，笔者猜测有可能这样命名的原因是说明Curator就是Zookeeper的馆长(脑洞有点大：Curtor就是动物园的园长)。

Curator包含了几个包：

curator-framework： 对zookeeper的底层api的一些封装

curator-client： 提供一些客户端的操作，例如重试策略等

curator-recipes： 封装了一些高级特性，如：Cache事件监听、选举、分布式锁、分布式计数器、布式Barrier等

Maven依赖(使用curator的版本：2.12.0，对应Zookeeper的版本为：3.4.x，**如果跨版本会有兼容性题，很有可能导致节点操作失败**)：

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>2.12.0</version>
</dependency>

<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>2.12.0</version>
</dependency>
```

Curator的基本Api

创建会话

1.使用静态工程方法创建客户端

一个例子如下：

```
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
CuratorFramework client =
CuratorFrameworkFactory.newClient(
                connectionInfo,
                5000,
                3000,
                retryPolicy);
```

newClient静态工厂方法包含四个主要参数：

参数名	说明
connectionString	服务器列表，格式host1:port1 host2:port2,...
retryPolicy	重试策略,内建有四种重试策略,也可自行实现RetryPolicy接口
sessionTimeoutMs	会话超时时间，单位毫秒，认60000ms
connectionTimeoutMs	连接创建超时时间，单毫秒，默认60000ms

2.使用Fluent风格的Api创建会话

核心参数变为流式设置，一个列子如下：

```
 RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
CuratorFramework client =
CuratorFrameworkFactory.builder()
    .connectString(connectionInfo)
    .sessionTimeoutMs(5000)
    .connectionTimeoutMs(5000)
    .retryPolicy(retryPolicy)
    .build();
```

3.创建包含隔离命名空间的会话

为了实现不同的Zookeeper业务之间的隔离，需要为每个业务分配一个独立的命名空间（**NameSpace**，即指定一个Zookeeper的根路径（官方术语：**为Zookeeper添加“Chroot”特性**）。例如（下面例子）当客户端指定了独立命名空间为“/base”，那么该客户端对Zookeeper上的数据节点的操作是基于该目录进行的。通过设置Chroot可以将客户端应用与Zookeeper服务端的一课子树相对应，多个应用共用一个Zookeeper集群的场景下，这对于实现不同应用之间的相互隔离十分有意义。

```
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
    CuratorFramework client =
    CuratorFrameworkFactory.builder()
        .connectString(connectionInfo)
        .sessionTimeoutMs(5000)
        .connectionTimeoutMs(5000)
        .retryPolicy(retryPolicy)
        .namespace("base")
        .build();
```

启动客户端

当创建会话成功，得到client的实例然后可以直接调用其start( )方法：

```
client.start();
```

数据节点操作

创建数据节点

Zookeeper的节点创建模式：

- PERSISTENT：持久化
- PERSISTENT_SEQUENTIAL：持久化并且带序列号
- EPHEMERAL：临时
- EPHEMERAL_SEQUENTIAL：临时并且带序列号

**创建一个节点，初始内容为空 **

```
client.create().forPath("path");
```

注意：如果没有设置节点属性，节点创建模式默认为持久化节点，内容默认为空

创建一个节点，附带初始化内容

```
client.create().forPath("path","init".getBytes());
```

创建一个节点，指定创建模式（临时节点），内容为空

```
client.create().withMode(CreateMode.EPHEMERAL).forPath("path");
```

创建一个节点，指定创建模式（临时节点），附带初始化内容

```
client.create().withMode(CreateMode.EPHEMERAL).forPath("path","init".getBytes());
```

创建一个节点，指定创建模式（临时节点），附带初始化内容，并且自动递归创建父节点

```
client.create()
      .creatingParentContainersIfNeeded()
      .withMode(CreateMode.EPHEMERAL)
      .forPath("path","init".getBytes());
```

这个creatingParentContainersIfNeeded()接口非常有用，因为一般情况开发人员在创建一个子节点须判断它的父节点是否存在，如果不存在直接创建会抛出NoNodeException，使用creatingParentContainersIfNeeded()之后Curator能够自动递归创建所有所需的父节点。

删除数据节点

删除一个节点

```
client.delete().forPath("path");
```

注意，此方法只能删除**叶子节点**，否则会抛出异常。

删除一个节点，并且递归删除其所有的子节点

```
client.delete().deletingChildrenIfNeeded().forPath("path");
```

删除一个节点，强制指定版本进行删除

```
client.delete().withVersion(10086).forPath("path");
```

删除一个节点，强制保证删除

```
client.delete().guaranteed().forPath("path");
```

guaranteed()接口是一个保障措施，只要客户端会话有效，那么Curator会在后台持续进行删除操作直到删除节点成功。

注意： 上面的多个流式接口是可以自由组合的，例如：

```
client.delete().guaranteed().deletingChildrenIfNeeded().withVersion(10086).forPath("path");
```

读取数据节点数据

读取一个节点的数据内容

```
client.getData().forPath("path");
```

注意，此方法返的返回值是byte[ ];

读取一个节点的数据内容，同时获取到该节点的stat

```
Stat stat = new Stat();
client.getData().storingStatIn(stat).forPath("path");
```

更新数据节点数据

更新一个节点的数据内容

```
client.setData().forPath("path","data".getBytes());
```

注意：该接口会返回一个Stat实例

更新一个节点的数据内容，强制指定版本进行更新

```
client.setData().withVersion(10086).forPath("path","data".getBytes());
```

检查节点是否存在

```
client.checkExists().forPath("path");
```

注意：该方法返回一个Stat实例，用于检查ZNode是否存在的操作. 可以调用额外的方法(监控或者后处理)并在最后调用forPath( )指定要操作的ZNode

获取某个节点的所有子节点路径

```
client.getChildren().forPath("path");
```

注意：该方法的返回值为List<String>,获得ZNode的子节点Path列表。 可以调用额外的方法(监控、台处理或者获取状态watch, background or get stat) 并在最后调用forPath()指定要操作的父ZNode

事务

CuratorFramework的实例包含inTransaction( )接口方法，调用此方法开启一个ZooKeeper事务. 可复合create, setData, check, and/or delete 等操作然后调用commit()作为一个原子操作提交。一个子如下：

```
client.inTransaction().check().forPath("path")
      .and()
      .create().withMode(CreateMode.EPHEMERAL).forPath("path","data".getBytes())
      .and()
      .setData().withVersion(10086).forPath("path","data2".getBytes())
      .and()
      .commit();
```

异步接口

上面提到的创建、删除、更新、读取等方法都是同步的，Curator提供异步接口，引入了**Background allback**接口用于处理异步接口调用之后服务端返回的结果信息。**BackgroundCallback**接口中一个要的回调值为CuratorEvent，里面包含事件类型、响应吗和节点的详细信息。

CuratorEventType

事件类型	对应CuratorFramework实例的方法
CREATE	#create()
DELETE	#delete()
EXISTS	#checkExists()
GET_DATA	#getData()
SET_DATA	#setData()
CHILDREN	#getChildren()
SYNC	#sync(String,Object)
GET_ACL	#getACL()
SET_ACL	#setACL()
WATCHED	#Watcher(Watcher)
CLOSING	#close()

响应码(#getResultCode())

响应码	意义
0	OK，即调用成功
-4	ConnectionLoss，即客户端与服务端断开连接
-110	NodeExists，即节点已经存在
-112	SessionExpired，即会话过期

一个异步创建节点的例子如下：

```
Executor executor = Executors.newFixedThreadPool(2);
client.create()
      .creatingParentsIfNeeded()
      .withMode(CreateMode.EPHEMERAL)
      .inBackground((curatorFramework, curatorEvent) -> {      System.out.println(String.format(
eventType:%s,resultCode:%s",curatorEvent.getType(),curatorEvent.getResultCode()));
      },executor)
      .forPath("path");
```

注意：如果#inBackground()方法不指定executor，那么会默认使用Curator的EventThread去进行步处理。

Curator食谱(高级特性)

提醒：首先你必须添加curator-recipes依赖，下文仅仅对recipes一些特性的使用进行解释和举例，打算进行源码级别的探讨

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>2.12.0</version>
</dependency>
```

重要提醒：强烈推荐使用ConnectionStateListener监控连接的状态，当连接状态为LOST，curator recipes下的所有Api将会失效或者过期，尽管后面所有的例子都没有使用到ConnectionStateListener。

缓存

Zookeeper原生支持通过注册Watcher来进行事件监听，但是开发者需要反复注册(Watcher只能单注册单次使用)。Cache是Curator中对事件监听的包装，可以看作是对事件监听的本地缓存视图，能自动为开发者处理反复注册监听。Curator提供了三种Watcher(Cache)来监听结点的变化。

Path Cache

Path Cache用来监控一个ZNode的子节点. 当一个子节点增加， 更新，删除时， Path Cache会改变的状态， 会包含最新的子节点， 子节点的数据和状态，而状态的更变将通过PathChildrenCacheListner通知。

实际使用时会涉及到四个类：

- PathChildrenCache
- PathChildrenCacheEvent
- PathChildrenCacheListener
- ChildData

通过下面的构造函数创建Path Cache:

public PathChildrenCache(CuratorFramework client, String path, boolean cacheData)

想使用cache，必须调用它的start方法，使用完后调用close方法。 可以设置StartMode来实现启动模式，

StartMode有下面几种：

1. NORMAL：正常初始化。
2. BUILD_INITIAL_CACHE：在调用 start()之前会调用rebuild()。
3. POST_INITIALIZED_EVENT： 当Cache初始化数据后发送一个PathChildrenCacheEvent.Type#INITIALIZED事件

public void addListener(PathChildrenCacheListener listener)可以增加listener监听缓存的变化。

getCurrentData()方法返回一个List对象，可以遍历所有的子节点。

设置/更新、移除其实是使用client (CuratorFramework)来操作, 不通过PathChildrenCache操作：

public class PathCacheDemo {

```
    <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/example/pathCache";

    <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args) throws Exception {
        TestingServer server = new TestingServer();
        CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectString(), new ExponentialBackoffRetry(1000, 3));
        client.start();
        PathChildrenCache cache = new PathChildrenCache(client, PATH, true);
        cache.start();
        PathChildrenCacheListener cacheListener = (client1, event) -> {
            System.out.println("<span style="font-family:'AR PL UKai HK';">事件类型：</span>" + event.getType());
            if (null != event.getData()) {
                System.out.println("<span style="font-family:'AR PL UKai HK';">节点数据：</span>" + event.getData().getPath() + " = " + new String(event.getData().getData()));
            }
        };
        cache.getListenable().addListener(cacheListener);
        client.create().creatingParentsIfNeeded().forPath("/example/pathCache/test01", "01".getBytes());
        Thread.sleep(10);
        client.create().creatingParentsIfNeeded().forPath("/example/pathCache/test02", "02".getBytes());
        Thread.sleep(10);
        client.setData().forPath("/example/pathCache/test01", "01_V2".getBytes());
        Thread.sleep(10);
        for (ChildData data : cache.getCurrentData()) {
            System.out.println("getCurrentData:" + data.getPath() + " = " + new String(data.getData()));
        }
        client.delete().forPath("/example/pathCache/test01");
        Thread.sleep(10);
        client.delete().forPath("/example/pathCache/test02");
        Thread.sleep(1000 * 5);
        cache.close();
        client.close();
        System.out.println("OK!");
    }
}
```

注意： 如果new PathChildrenCache(client, PATH, true)中的参数cacheData值设置为false，则示中的event.getData().getData()、data.getData()将返回null，cache将不会缓存节点数据。

注意： 示例中的Thread.sleep(10)可以注释掉，但是注释后事件监听的触发次数会不全，这可能与PatCache的实现原理有关，不能太过频繁的触发事件！

Node Cache

Node Cache与Path Cache类似，Node Cache只是监听某一个特定的节点。它涉及到下面的三个类：

- NodeCache - Node Cache实现类

- NodeCacheListener - 节点监听器
- ChildData - 节点数据

注意： 使用cache，依然要调用它的start()方法，使用完后调用close()方法。

getCurrentData()将得到节点当前的状态，通过它的状态可以得到当前的值。

```
<span style="color:#c792ea;font-style:italic;">public</span> class NodeCacheDemo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/example/cache";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args) throws Exception {
    TestingServer server = new TestingServer();
    CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectString(), new ExponentialBackoffRetry(1000, 3));
    client.start();
    client.create().creatingParentsIfNeeded().forPath(PATH);
    <span style="color:#c792ea;font-style:italic;">final</span> NodeCache cache = new NodCache(client, PATH);
    NodeCacheListener listener = () -> {
      ChildData data = cache.getCurrentData();
      if (null != data) {
        System.out.println("<span style="font-family:'AR PL UKai HK';">节点数据：</span>" + new String(cache.getCurrentData().getData()));
      } else {
        System.out.println("<span style="font-family:'AR PL UKai HK';">节点被删除</span>!");
      }
    };
    cache.getListenable().addListener(listener);
    cache.start();
    client.setData().forPath(PATH, "01".getBytes());
    Thread.sleep(100);
    client.setData().forPath(PATH, "02".getBytes());
    Thread.sleep(100);
    client.delete().deletingChildrenIfNeeded().forPath(PATH);
    Thread.sleep(1000 * 2);
    cache.close();
    client.close();
    System.out.println("OK!");
  }
}
```

注意： 示例中的Thread.sleep(10)可以注释，但是注释后事件监听的触发次数会不全，这可能与NodCache的实现原理有关，不能太过频繁的触发事件！

注意： NodeCache只能监听一个节点的状态变化。

Tree Cache

Tree Cache可以监控整个树上的所有节点，类似于PathCache和NodeCache的组合，主要涉及到下四个类：

- TreeCache - Tree Cache实现类
- TreeCacheListener - 监听器类
- TreeCacheEvent - 触发的事件类
- ChildData - 节点数据

```
<span style="color:#c792ea;font-style:italic;">public</span> class TreeCacheDemo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/example/cache";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args) throws Exception {
    TestingServer server = new TestingServer();
    CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectString(), new ExponentialBackoffRetry(1000, 3));
    client.start();
    client.create().creatingParentsIfNeeded().forPath(PATH);
    TreeCache cache = new TreeCache(client, PATH);
    TreeCacheListener listener = (client1, event) ->
        System.out.println("<span style="font-family:'AR PL UKai HK';">事件类型：</span>" + event.getType() +
            "｜<span style="font-family:'AR PL UKai HK';">路径：</span>" + (null != event.getData() ? event.getData().getPath() : null));
    cache.getListenable().addListener(listener);
    cache.start();
    client.setData().forPath(PATH, "01".getBytes());
    Thread.sleep(100);
    client.setData().forPath(PATH, "02".getBytes());
    Thread.sleep(100);
    client.delete().deletingChildrenIfNeeded().forPath(PATH);
    Thread.sleep(1000 * 2);
    cache.close();
    client.close();
    System.out.println("OK!");
  }
}
```

注意： 在此示例中没有使用Thread.sleep(10)，但是事件触发次数也是正常的。

注意： TreeCache在初始化(调用start()方法)的时候会回调TreeCacheListener实例一个事TreeCacheEvent，而回调的TreeCacheEvent对象的Type为INITIALIZED，ChildData为null，此时event.getData().getPath()很有可能导致空指针异常，这里应该主动处理并避免这种情况。

Leader选举

在分布式计算中， **leader elections**是很重要的一个功能， 这个选举过程是这样子的： 指派一个进作为组织者，将任务分发给各节点。 在任务开始前， 哪个节点都不知道谁是leader(领导者)或者coordnator(协调者). 当选举算法开始执行后， 每个节点最终会得到一个唯一的节点作为任务leader. 除此外， 选举还经常会发生在leader意外宕机的情况下，新的leader要被选举出来。

在zookeeper集群中，leader负责写操作，然后通过Zab协议实现follower的同步，leader或者follower都可以处理读操作。

Curator 有两种leader选举的recipe,分别是**LeaderSelector**和**LeaderLatch**。

前者是所有存活的客户端不间断的轮流做Leader，大同社会。后者是一旦选举出Leader，除非有客端挂掉重新触发选举，否则不会交出领导权。某党?

LeaderLatch

LeaderLatch有两个构造函数：

```
<span style="color:#c792ea;font-style:italic;">public</span> LeaderLatch(CuratorFramework
lient, String latchPath)
<span style="color:#c792ea;font-style:italic;">public</span> LeaderLatch(CuratorFramework
lient, String latchPath,  String id)
```

LeaderLatch的启动：

leaderLatch.start( );

一旦启动，LeaderLatch会和其它使用相同latch path的其它LeaderLatch交涉，然后其中一个最终会选举为leader，可以通过hasLeadership方法查看LeaderLatch实例是否leader：

leaderLatch.hasLeadership( ); //返回true说明当前实例是leader

类似JDK的CountDownLatch， LeaderLatch在请求成为leadership会block(阻塞)，一旦不使用LeadrLatch了，必须调用close方法。 如果它是leader,会释放leadership， 其它的参与者将会选举一个leaer。

```
<span style="color:#c792ea;font-style:italic;">public</span> void await() throws InterruptedE
ception,EOFException
/*Causes the current thread to wait until this instance acquires leadership
unless the thread is interrupted or closed.*/
<span style="color:#c792ea;font-style:italic;">public</span> boolean await(long timeout,Tim
Unit unit)throws InterruptedException
```

异常处理： LeaderLatch实例可以增加ConnectionStateListener来监听网络连接问题。 当 SUSPENED 或 LOST 时, leader不再认为自己还是leader。当LOST后连接重连后RECONNECTED,LeaderLatc会删除先前的ZNode然后重新创建一个。LeaderLatch用户必须考虑导致leadership丢失的连接问题。强烈推荐你使用ConnectionStateListener。

一个LeaderLatch的使用例子：

```
<span style="color:#c792ea;font-style:italic;">public</span> class LeaderLatchDemo extends
BaseConnectionInfo {
   <span style="color:#c792ea;font-style:italic;">protected static</span> String PATH = "/fran
is/leader";
   <span style="color:#c792ea;font-style:italic;">private static final</span> int CLIENT_QTY =
0;

   <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
      List<CuratorFramework> clients = Lists.newArrayList();
      List<LeaderLatch> examples = Lists.newArrayList();
      TestingServer server=new TestingServer();
      try {
```

```
        for (int i = 0; i < CLIENT_QTY; i++) {
          CuratorFramework client
              = CuratorFrameworkFactory.newClient(server.getConnectString(), new Exponential
ackoffRetry(20000, 3));
          clients.add(client);
          LeaderLatch latch = new LeaderLatch(client, PATH, "Client #" + i);
          latch.addListener(new LeaderLatchListener() {

            @Override
            <span style="color:#c792ea;font-style:italic;">public</span> void isLeader() {
              // <span style="color:#ffeb95;font-style:italic;">TODO Auto-generated method st
b
</span>                System.out.println("I am Leader");
            }

            @Override
            <span style="color:#c792ea;font-style:italic;">public</span> void notLeader() {
              // <span style="color:#ffeb95;font-style:italic;">TODO Auto-generated method st
b
</span>                System.out.println("I am not Leader");
            }
          });
          examples.add(latch);
          client.start();
          latch.start();
        }
        Thread.sleep(10000);
        LeaderLatch currentLeader = null;
        for (LeaderLatch latch : examples) {
          if (latch.hasLeadership()) {
            currentLeader = latch;
          }
        }
        System.out.println("current leader is " + currentLeader.getId());
        System.out.println("release the leader " + currentLeader.getId());
        currentLeader.close();

        Thread.sleep(5000);

        for (LeaderLatch latch : examples) {
          if (latch.hasLeadership()) {
            currentLeader = latch;
          }
        }
        System.out.println("current leader is " + currentLeader.getId());
        System.out.println("release the leader " + currentLeader.getId());
      } finally {
        for (LeaderLatch latch : examples) {
          if (null != latch.getState())
          CloseableUtils.closeQuietly(latch);
        }
        for (CuratorFramework client : clients) {
          CloseableUtils.closeQuietly(client);
        }
```

```
    }
  }
}
```

可以添加test module的依赖方便进行测试，不需要启动真实的zookeeper服务端：

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-test</artifactId>
    <version>2.12.0</version>
</dependency>
```

首先我们创建了10个LeaderLatch，启动后它们中的一个会被选举为leader。 因为选举会花费一些时，start后并不能马上就得到leader。

通过hasLeadership查看自己是否是leader， 如果是的话返回true。

可以通过.getLeader().getId()可以得到当前的leader的ID。

只能通过close释放当前的领导权。

await是一个阻塞方法， 尝试获取leader地位，但是未必能上位。

LeaderSelector

LeaderSelector使用的时候主要涉及下面几个类：

- LeaderSelector
- LeaderSelectorListener
- LeaderSelectorListenerAdapter
- CancelLeadershipException

核心类是LeaderSelector，它的构造函数如下：

```
<span style="color:#c792ea;font-style:italic;">public</span> LeaderSelector(CuratorFramework client, String mutexPath,LeaderSelectorListener listener)
<span style="color:#c792ea;font-style:italic;">public</span> LeaderSelector(CuratorFramework client, String mutexPath, ThreadFactory threadFactory, Executor executor, LeaderSelectorListener listener)
```

类似LeaderLatch,LeaderSelector必须start: leaderSelector.start(); 一旦启动，当实例取得领导权时的listener的takeLeadership()方法被调用。而takeLeadership()方法只有领导权被释放时才返回。你不再使用LeaderSelector实例时，应该调用它的close方法。

异常处理 LeaderSelectorListener类继承ConnectionStateListener。LeaderSelector必须小心连接态的改变。如果实例成为leader, 它应该响应SUSPENDED 或 LOST。 当 SUSPENDED 状态出现时，实例必须假定在重新连接成功之前它可能不再是leader了。 如果LOST状态出现， 实例不再是leader，takeLeadership方法返回。

重要: 推荐处理方式是当收到SUSPENDED 或 LOST时抛出CancelLeadershipException异常.。这会致LeaderSelector实例中断并取消执行takeLeadership方法的异常.。这非常重要， 你必须考虑扩展LeaderSelectorListenerAdapter. LeaderSelectorListenerAdapter提供了推荐的处理逻辑。

下面的一个例子摘抄自官方：

```
<span style="color:#c792ea;font-style:italic;">public</span> class LeaderSelectorAdapter extnds LeaderSelectorListenerAdapter implements Closeable {
  <span style="color:#c792ea;font-style:italic;">private final</span> String name;
  <span style="color:#c792ea;font-style:italic;">private final</span> LeaderSelector leaderSeector;
  <span style="color:#c792ea;font-style:italic;">private final</span> AtomicInteger leaderCont = new AtomicInteger();

  <span style="color:#c792ea;font-style:italic;">public</span> LeaderSelectorAdapter(CuratoFramework client, String path, String name) {
    <span style="color:#c792ea;font-style:italic;">this</span>.name = name;
    leaderSelector = new LeaderSelector(client, path, <span style="color:#c792ea;font-style:italic;">this</span>);
    leaderSelector.autoRequeue();
  }

  <span style="color:#c792ea;font-style:italic;">public</span> void start() throws IOExceptio {
    leaderSelector.start();
  }

  @Override
  <span style="color:#c792ea;font-style:italic;">public</span> void close() throws IOExcepti{
    leaderSelector.close();
  }

  @Override
  <span style="color:#c792ea;font-style:italic;">public</span> void takeLeadership(CuratorFrmework client) throws Exception {
    <span style="color:#c792ea;font-style:italic;">final</span> int waitSeconds = (int) (5 * Mah.random()) + 1;
    System.out.println(name + " is now the leader. Waiting " + waitSeconds + " seconds...");
    System.out.println(name + " has been leader " + leaderCount.getAndIncrement() + " time() before.");
    try {
      Thread.sleep(TimeUnit.SECONDS.toMillis(waitSeconds));
    } catch (InterruptedException e) {
      System.err.println(name + " was interrupted.");
      Thread.currentThread().interrupt();
    } finally {
      System.out.println(name + " relinquishing leadership.\n");
    }
  }
}
```

你可以在takeLeadership进行任务的分配等等，并且不要返回，如果你想要要此实例一直是leader的可以加一个死循环。调用 leaderSelector.autoRequeue();保证在此实例释放领导权之后还可能获得导权。 在这里我们使用AtomicInteger来记录此client获得领导权的次数， 它是” fair” ， 每个clien有平等的机会获得领导权。

```
<span style="color:#c792ea;font-style:italic;">public</span> class LeaderSelectorDemo {

  <span style="color:#c792ea;font-style:italic;">protected static</span> String PATH = "/fran
```

```
is/leader";
   <span style="color:#c792ea;font-style:italic;">private static final</span> int CLIENT_QTY =
0;

   <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
      List<CuratorFramework> clients = Lists.newArrayList();
      List<LeaderSelectorAdapter> examples = Lists.newArrayList();
      TestingServer server = new TestingServer();
      try {
         for (int i = 0; i < CLIENT_QTY; i++) {
            CuratorFramework client
                  = CuratorFrameworkFactory.newClient(server.getConnectString(), new Exponential
ackoffRetry(20000, 3));
            clients.add(client);
            LeaderSelectorAdapter selectorAdapter = new LeaderSelectorAdapter(client, PATH, "Cl
ent #" + i);
            examples.add(selectorAdapter);
            client.start();
            selectorAdapter.start();
         }
         System.out.println("Press enter/return to quit\n");
         new BufferedReader(new InputStreamReader(System.in)).readLine();
      } finally {
         System.out.println("Shutting down...");
         for (LeaderSelectorAdapter exampleClient : examples) {
            CloseableUtils.closeQuietly(exampleClient);
         }
         for (CuratorFramework client : clients) {
            CloseableUtils.closeQuietly(client);
         }
         CloseableUtils.closeQuietly(server);
      }
   }
}
```

对比可知，LeaderLatch必须调用close()方法才会释放领导权，而对于LeaderSelector，通过LeaderSlectorListener可以对领导权进行控制， 在适当的时候释放领导权，这样每个节点都有可能获得领导。从而，LeaderSelector具有更好的灵活性和可控性，建议有LeaderElection应用场景下优先使用LeaerSelector。

分布式锁

提醒：

1.推荐使用ConnectionStateListener监控连接的状态，因为当连接LOST时你不再拥有锁

2.分布式的锁全局同步， 这意味着任何一个时间点不会有两个客户端都拥有相同的锁。

可重入共享锁—Shared Reentrant Lock

Shared意味着锁是全局可见的， 客户端都可以请求锁。 Reentrant和JDK的ReentrantLock类似，可重入， 意味着同一个客户端在拥有锁的同时，可以多次获取，不会被阻塞。 它是由类InterProces

Mutex来实现。 它的构造函数为：

public InterProcessMutex(CuratorFramework client, String path)

通过acquire()获得锁，并提供超时机制：

public void acquire()
Acquire the mutex - blocking until it's available. Note: the same thread can call acquire re-entrantly. Each call to acquire must be balanced by a call to release()

public boolean acquire(long time,TimUnit unit)
Acquire the mutex - blocks until it's available or the given time expires. Note: the same thread can call acquire re-entrantly. Each call to acquire that returns true must be balanced by a call o release()

Parameters:
time - time to wait
unit - time unit
Returns:
true if the mutex was acquired, false if not

通过release()方法释放锁。 InterProcessMutex 实例可以重用。

Revoking ZooKeeper recipes wiki定义了可协商的撤销机制。 为了撤销mutex, 调用下面的方法：

public void makeRevocable(RevocatioListener<T> listener)
将锁设为可撤销的. 当别的进程或线程想让你释放锁时Listener会被调用。
Parameters:
listener - the listener

如果你请求撤销当前的锁， 调用attemptRevoke()方法,注意锁释放时RevocationListener将会回调。

public static void attemptRevoke(CuratorFramework client,String path) throws Exception
Utility to mark a lock for revocation. Assuming that the lock has been registered
with a RevocationListener, it will get called and the lock should be released. Note,
however, that revocation is cooperative.
Parameters:
client - the client
path - the path of the lock - usually from something like InterProcessMutex.getParticipantNodes()

二次提醒：错误处理 还是强烈推荐你使用ConnectionStateListener处理连接状态的改变。 当连接LOST时你不再拥有锁。

首先让我们创建一个模拟的共享资源， 这个资源期望只能单线程的访问，否则会有并发问题。

public class FakeLimitedResource {
  private final AtomicBoolean inUse = new AtomicBoolean(false);

```
    public void use() throws InterruptedException {
        // 真实环境中我们会在这里访问/维护一个共享的资源
        //这个例子在使用锁的情况下不会非法发异常IllegalStateException
        //但是在无锁的情况由于sleep了一段时间，很容易抛出异常
        if (!inUse.compareAndSet(false, true)) {
            throw new IllegalStateException("Needs to be used by one client at a time");
        }
        try {
            Thread.sleep((long) (3 * Math.random()));
        } finally {
            inUse.set(false);
        }
    }
}
```

然后创建一个InterProcessMutexDemo类， 它负责请求锁， 使用资源，释放锁这样一个完整的访过程。

```
public class InterProcessMutexDemo {

    private InterProcessMutex lock;
    private final FakeLimitedResource resource;
    private final String clientName;

    public InterProcessMutexDemo(CuratorFramework client, String lockPath, FakeLimitedResource resource, String clientName) {
        this.resource = resource;
        this.clientName = clientName;
        this.lock = new InterProcessMutex(client, lockPath);
    }

    public void doWork(long time, TimeUnit unit) throws Exception {
        if (!lock.acquire(time, unit)) {
            throw new IllegalStateException(clientName + " could not acquire the lock");
        }
        try {
            System.out.println(clientName + " get the lock");
            resource.use(); //access resource exclusively
        } finally {
            System.out.println(clientName + " releasing the lock");
            lock.release(); // always release the lock in a finally block
        }
    }

    private static final int QTY = 5;
    private static final int REPETITIONS =
```

```
QTY * 10;
   <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
amples/locks";

   <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
      <span style="color:#c792ea;font-style:italic;">final</span> FakeLimitedResource resource
= new FakeLimitedResource();
      ExecutorService service = Executors.newFixedThreadPool(QTY);
      <span style="color:#c792ea;font-style:italic;">final</span> TestingServer server = new Te
tingServer();
      try {
         for (int i = 0; i < QTY; ++i) {
            <span style="color:#c792ea;font-style:italic;">final</span> int index = i;
            Callable<Void> task = new Callable<Void>() {
               @Override
               <span style="color:#c792ea;font-style:italic;">public</span> Void call() throws Exce
tion {
                  CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnec
String(), new ExponentialBackoffRetry(1000, 3));
                  try {
                     client.start();
                     <span style="color:#c792ea;font-style:italic;">final</span> InterProcessMutexD
mo example = new InterProcessMutexDemo(client, PATH, resource, "Client " + index);
                     for (int j = 0; j < REPETITIONS; ++j) {
                        example.doWork(10, TimeUnit.SECONDS);
                     }
                  } catch (Throwable e) {
                     e.printStackTrace();
                  } finally {
                     CloseableUtils.closeQuietly(client);
                  }
                  return null;
               }
            };
            service.submit(task);
         }
         service.shutdown();
         service.awaitTermination(10, TimeUnit.MINUTES);
      } finally {
         CloseableUtils.closeQuietly(server);
      }
   }
}
```

代码也很简单，生成10个client， 每个client重复执行10次 请求锁–访问资源–释放锁的过程。每个cli nt都在独立的线程中。 结果可以看到，锁是随机的被每个实例排他性的使用。

既然是可重用的，你可以在一个线程中多次调用acquire(),在线程拥有锁时它总是返回true。

你不应该在多个线程中用同一个InterProcessMutex， 你可以在每个线程中都生成一个新的InterPro essMutex实例，它们的path都一样，这样它们可以共享同一个锁。

不可重入共享锁—Shared Lock

这个锁和上面的InterProcessMutex相比，就是少了Reentrant的功能，也就意味着它不能在同一个程中重入。这个类是InterProcessSemaphoreMutex,使用方法和InterProcessMutex类似

```
<span style="color:#c792ea;font-style:italic;">public</span> class InterProcessSemaphoreMu
exDemo {

  <span style="color:#c792ea;font-style:italic;">private</span> InterProcessSemaphoreMute
 lock;
  <span style="color:#c792ea;font-style:italic;">private final</span> FakeLimitedResource re
ource;
  <span style="color:#c792ea;font-style:italic;">private final</span> String clientName;

  <span style="color:#c792ea;font-style:italic;">public</span> InterProcessSemaphoreMute
Demo(CuratorFramework client, String lockPath, FakeLimitedResource resource, String client
ame) {
    <span style="color:#c792ea;font-style:italic;">this</span>.resource = resource;
    <span style="color:#c792ea;font-style:italic;">this</span>.clientName = clientName;
    <span style="color:#c792ea;font-style:italic;">this</span>.lock = new InterProcessSemap
oreMutex(client, lockPath);
  }

  <span style="color:#c792ea;font-style:italic;">public</span> void doWork(long time, Time
nit unit) throws Exception {
    if (!lock.acquire(time, unit))
    {
      throw new IllegalStateException(clientName + " <span style="font-family:'AR PL UKai H
';">不能得到互斥锁</span>");
    }
    System.out.println(clientName + " <span style="font-family:'AR PL UKai HK';">已获取到
斥锁</span>");
    if (!lock.acquire(time, unit))
    {
      throw new IllegalStateException(clientName + " <span style="font-family:'AR PL UKai H
';">不能得到互斥锁</span>");
    }
    System.out.println(clientName + " <span style="font-family:'AR PL UKai HK';">再次获取
互斥锁</span>");
    try {
      System.out.println(clientName + " get the lock");
      resource.use(); //access resource exclusively
    } finally {
      System.out.println(clientName + " releasing the lock");
      lock.release(); // always release the lock in a finally block
      lock.release(); // <span style="font-family:'AR PL UKai HK';">获取锁几次 释放锁也要几次
</span>     }
  }

  <span style="color:#c792ea;font-style:italic;">private static final</span> int QTY = 5;
  <span style="color:#c792ea;font-style:italic;">private static final</span> int REPETITIONS =
QTY * 10;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
amples/locks";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
```

```
hrows Exception {
    <span style="color:#c792ea;font-style:italic;">final</span> FakeLimitedResource resource
= new FakeLimitedResource();
    ExecutorService service = Executors.newFixedThreadPool(QTY);
    <span style="color:#c792ea;font-style:italic;">final</span> TestingServer server = new Te
tingServer();
    try {
      for (int i = 0; i < QTY; ++i) {
        <span style="color:#c792ea;font-style:italic;">final</span> int index = i;
        Callable<Void> task = new Callable<Void>() {
          @Override
          <span style="color:#c792ea;font-style:italic;">public</span> Void call() throws Exce
tion {
            CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnec
String(), new ExponentialBackoffRetry(1000, 3));
            try {
              client.start();
              <span style="color:#c792ea;font-style:italic;">final</span> InterProcessSemaph
reMutexDemo example = new InterProcessSemaphoreMutexDemo(client, PATH, resource, "Cl
ent " + index);
              for (int j = 0; j < REPETITIONS; ++j) {
                example.doWork(10, TimeUnit.SECONDS);
              }
            } catch (Throwable e) {
              e.printStackTrace();
            } finally {
              CloseableUtils.closeQuietly(client);
            }
            return null;
          }
        };
        service.submit(task);
      }
      service.shutdown();
      service.awaitTermination(10, TimeUnit.MINUTES);
    } finally {
      CloseableUtils.closeQuietly(server);
    }
    Thread.sleep(Integer.MAX_VALUE);
  }
}
```

运行后发现，有且只有一个client成功获取第一个锁(第一个acquire()方法返回true)，然后它自己阻在第二个acquire()方法，获取第二个锁超时；其他所有的客户端都阻塞在第一个acquire()方法超时且抛出异常。

这样也就验证了InterProcessSemaphoreMutex实现的锁是不可重入的。

可重入读写锁—Shared Reentrant Read Write Lock

类似JDK的**ReentrantReadWriteLock**。一个读写锁管理一对相关的锁。一个负责读操作，另外一负责写操作。读操作在写锁没被使用时可同时由多个进程使用，而写锁在使用时不允许读(阻塞)。

此锁是可重入的。**一个拥有写锁的线程可重入读锁，但是读锁却不能进入写锁。**这也意味着**写锁可以**

级成读锁， 比如请求写锁 --->请求读锁--->释放读锁 ---->释放写锁。从读锁升级成写锁是不行的。

可重入读写锁主要由两个类实现：InterProcessReadWriteLock、InterProcessMutex。使用时首先建一个InterProcessReadWriteLock实例，然后再根据你的需求得到读锁或者写锁，读写锁的类型是InterProcessMutex。

```
<span style="color:#c792ea;font-style:italic;">public</span> class ReentrantReadWriteLockDmo {

  <span style="color:#c792ea;font-style:italic;">private final</span> InterProcessReadWriteLck lock;
  <span style="color:#c792ea;font-style:italic;">private final</span> InterProcessMutex readock;
  <span style="color:#c792ea;font-style:italic;">private final</span> InterProcessMutex writeock;
  <span style="color:#c792ea;font-style:italic;">private final</span> FakeLimitedResource reource;
  <span style="color:#c792ea;font-style:italic;">private final</span> String clientName;

  <span style="color:#c792ea;font-style:italic;">public</span> ReentrantReadWriteLockDem(CuratorFramework client, String lockPath, FakeLimitedResource resource, String clientName)

    <span style="color:#c792ea;font-style:italic;">this</span>.resource = resource;
    <span style="color:#c792ea;font-style:italic;">this</span>.clientName = clientName;
    lock = new InterProcessReadWriteLock(client, lockPath);
    readLock = lock.readLock();
    writeLock = lock.writeLock();
  }

  <span style="color:#c792ea;font-style:italic;">public</span> void doWork(long time, Timenit unit) throws Exception {
    // <span style="font-family:'AR PL UKai HK';">注意只能先得到写锁再得到读锁，不能反过来！！
</span>    if (!writeLock.acquire(time, unit)) {
      throw new IllegalStateException(clientName + " <span style="font-family:'AR PL UKai H';">不能得到写锁</span>");
    }
    System.out.println(clientName + " <span style="font-family:'AR PL UKai HK';">已得到写锁/span>");
    if (!readLock.acquire(time, unit)) {
      throw new IllegalStateException(clientName + " <span style="font-family:'AR PL UKai H';">不能得到读锁</span>");
    }
    System.out.println(clientName + " <span style="font-family:'AR PL UKai HK';">已得到读锁/span>");
    try {
      resource.use(); // <span style="font-family:'AR PL UKai HK';">使用资源
</span>      Thread.sleep(1000);
    } finally {
      System.out.println(clientName + " <span style="font-family:'AR PL UKai HK';">释放读写</span>");
      readLock.release();
      writeLock.release();
    }
```

原文链接：Curator 使用详解

```
  }

  <span style="color:#c792ea;font-style:italic;">private static final</span> int QTY = 5;
  <span style="color:#c792ea;font-style:italic;">private static final</span> int REPETITIONS = QTY ;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/examples/locks";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args) hrows Exception {
    <span style="color:#c792ea;font-style:italic;">final</span> FakeLimitedResource resource = new FakeLimitedResource();
    ExecutorService service = Executors.newFixedThreadPool(QTY);
    <span style="color:#c792ea;font-style:italic;">final</span> TestingServer server = new TetingServer();
    try {
      for (int i = 0; i < QTY; ++i) {
        <span style="color:#c792ea;font-style:italic;">final</span> int index = i;
        Callable<Void> task = new Callable<Void>() {
          @Override
          <span style="color:#c792ea;font-style:italic;">public</span> Void call() throws Exception {
            CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnecString(), new ExponentialBackoffRetry(1000, 3));
            try {
              client.start();
              <span style="color:#c792ea;font-style:italic;">final</span> ReentrantReadWriteockDemo example = new ReentrantReadWriteLockDemo(client, PATH, resource, "Client " + idex);
              for (int j = 0; j < REPETITIONS; ++j) {
                example.doWork(10, TimeUnit.SECONDS);
              }
            } catch (Throwable e) {
              e.printStackTrace();
            } finally {
              CloseableUtils.closeQuietly(client);
            }
            return null;
          }
        };
        service.submit(task);
      }
      service.shutdown();
      service.awaitTermination(10, TimeUnit.MINUTES);
    } finally {
      CloseableUtils.closeQuietly(server);
    }
  }
}
```

信号量—Shared Semaphore

一个计数的信号量类似JDK的Semaphore。 JDK中Semaphore维护的一组许可(**permits**)，而Curato中称之为租约(**Lease**)。 有两种方式可以决定semaphore的最大租约数。第一种方式是用户给定path

且指定最大LeaseSize。第二种方式用户给定path并且使用SharedCountReader类。**如果不使用SharedCountReader, 必须保证所有实例在多进程中使用相同的(最大)租约数量,否则有可能出现A进程中实例持有最大租约数量为10，但是在B进程中持有的最大租约数量为20，此时租约的意义就失效了。**

这次调用acquire()会返回一个租约对象。 客户端必须在finally中close这些租约对象，否则这些租约丢失掉。 但是， 但是，如果客户端session由于某种原因比如crash丢掉， 那么这些客户端持有的租会自动close， 这样其它客户端可以继续使用这些租约。 租约还可以通过下面的方式返还：

```
<span style="color:#c792ea;font-style:italic;">public</span> void returnAll(Collection<Lease
 leases)
<span style="color:#c792ea;font-style:italic;">public</span> void returnLease(Lease lease)
```

注意你可以一次性请求多个租约，如果Semaphore当前的租约不够，则请求线程会被阻塞。 同时还供了超时的重载方法。

```
<span style="color:#c792ea;font-style:italic;">public</span> Lease acquire()
<span style="color:#c792ea;font-style:italic;">public</span> Collection<Lease> acquire(int q
y)
<span style="color:#c792ea;font-style:italic;">public</span> Lease acquire(long time, TimeU
it unit)
<span style="color:#c792ea;font-style:italic;">public</span> Collection<Lease> acquire(int q
y, long time, TimeUnit unit)
```

Shared Semaphore使用的主要类包括下面几个：

- InterProcessSemaphoreV2
- Lease
- SharedCountReader

```
<span style="color:#c792ea;font-style:italic;">public</span> class InterProcessSemaphoreD
mo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> int MAX_LEASE =
0;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
amples/locks";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
    FakeLimitedResource resource = new FakeLimitedResource();
    try (TestingServer server = new TestingServer()) {

      CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectStrin
(), new ExponentialBackoffRetry(1000, 3));
      client.start();

      InterProcessSemaphoreV2 semaphore = new InterProcessSemaphoreV2(client, PATH,
AX_LEASE);
      Collection<Lease> leases = semaphore.acquire(5);
      System.out.println("get " + leases.size() + " leases");
      Lease lease = semaphore.acquire();
      System.out.println("get another lease");

      resource.use();
```

```
            Collection<Lease> leases2 = semaphore.acquire(5, 10, TimeUnit.SECONDS);
            System.out.println("Should timeout and acquire return " + leases2);

            System.out.println("return one lease");
            semaphore.returnLease(lease);
            System.out.println("return another 5 leases");
            semaphore.returnAll(leases);
        }
    }
}
```

首先我们先获得了5个租约， 最后我们把它还给了semaphore。 接着请求了一个租约，因为semaphore还有5个租约，所以请求可以满足，返回一个租约，还剩4个租约。 然后再请求一个租约，因为租约够，**阻塞到超时，还是没能满足，返回结果为null(租约不足会阻塞到超时，然后返回null，不会主动出异常；如果不设置超时时间，会一致阻塞)。**

上面说讲的锁都是公平锁(fair)。 总ZooKeeper的角度看， 每个客户端都按照请求的顺序获得锁，不在非公平的抢占的情况。

多共享锁对象 —Multi Shared Lock

Multi Shared Lock是一个锁的容器。 当调用acquire()， 所有的锁都会被acquire()，如果请求失败所有的锁都会被release。 同样调用release时所有的锁都被release(**失败被忽略**)。 基本上，它就是组的代表，在它上面的请求释放操作都会传递给它包含的所有的锁。

主要涉及两个类：

- InterProcessMultiLock
- InterProcessLock

它的构造函数需要包含的锁的集合，或者一组ZooKeeper的path。

```
<span style="color:#c792ea;font-style:italic;">public</span> InterProcessMultiLock(List<InterProcessLock> locks)
<span style="color:#c792ea;font-style:italic;">public</span> InterProcessMultiLock(CuratorFramework client, List<String> paths)
```

用法和Shared Lock相同。

```
<span style="color:#c792ea;font-style:italic;">public</span> class MultiSharedLockDemo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH1 = "/examples/locks1";
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH2 = "/examples/locks2";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args) hrows Exception {
    FakeLimitedResource resource = new FakeLimitedResource();
    try (TestingServer server = new TestingServer()) {
      CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectStrin(), new ExponentialBackoffRetry(1000, 3));
```

```
        client.start();

        InterProcessLock lock1 = new InterProcessMutex(client, PATH1);
        InterProcessLock lock2 = new InterProcessSemaphoreMutex(client, PATH2);

        InterProcessMultiLock lock = new InterProcessMultiLock(Arrays.asList(lock1, lock2));

        if (!lock.acquire(10, TimeUnit.SECONDS)) {
          throw new IllegalStateException("could not acquire the lock");
        }
        System.out.println("has got all lock");

        System.out.println("has got lock1: " + lock1.isAcquiredInThisProcess());
        System.out.println("has got lock2: " + lock2.isAcquiredInThisProcess());

        try {
          resource.use(); //access resource exclusively
        } finally {
          System.out.println("releasing the lock");
          lock.release(); // always release the lock in a finally block
        }
        System.out.println("has got lock1: " + lock1.isAcquiredInThisProcess());
        System.out.println("has got lock2: " + lock2.isAcquiredInThisProcess());
      }
    }
}
```

新建一个InterProcessMultiLock， 包含一个重入锁和一个非重入锁。 调用acquire()后可以看到线程时拥有了这两个锁。 调用release()看到这两个锁都被释放了。

最后再重申一次， 强烈推荐使用ConnectionStateListener监控连接的状态，当连接状态为LOST，将会丢失。

分布式计数器

顾名思义，计数器是用来计数的, 利用ZooKeeper可以实现一个集群共享的计数器。 只要使用相同的path就可以得到最新的计数器值， 这是由ZooKeeper的一致性保证的。Curator有两个计数器， 一个用int来计数(SharedCount)，一个用long来计数(DistributedAtomicLong)。

分布式int计数器—SharedCount

这个类使用int类型来计数。 主要涉及三个类。

- SharedCount
- SharedCountReader
- SharedCountListener

SharedCount代表计数器， 可以为它增加一个SharedCountListener，当计数器改变时此Listener可监听到改变的事件，而SharedCountReader可以读取到最新的值， 包括字面值和带版本信息的值VersonedValue。

```
<span style="color:#c792ea;font-style:italic;">public</span> class SharedCounterDemo impl
```

```
ments SharedCountListener {

  <span style="color:#c792ea;font-style:italic;">private static final</span> int QTY = 5;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
amples/counter";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows IOException, Exception {
    <span style="color:#c792ea;font-style:italic;">final</span> Random rand = new Random(
;
    SharedCounterDemo example = new SharedCounterDemo();
    try (TestingServer server = new TestingServer()) {
      CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectStrin
(), new ExponentialBackoffRetry(1000, 3));
      client.start();

      SharedCount baseCount = new SharedCount(client, PATH, 0);
      baseCount.addListener(example);
      baseCount.start();

      List<SharedCount> examples = Lists.newArrayList();
      ExecutorService service = Executors.newFixedThreadPool(QTY);
      for (int i = 0; i < QTY; ++i) {
        <span style="color:#c792ea;font-style:italic;">final</span> SharedCount count = new
SharedCount(client, PATH, 0);
        examples.add(count);
        Callable<Void> task = () -> {
          count.start();
          Thread.sleep(rand.nextInt(10000));
          System.out.println("Increment:" + count.trySetCount(count.getVersionedValue(), co
nt.getCount() + rand.nextInt(10)));
          return null;
        };
        service.submit(task);
      }

      service.shutdown();
      service.awaitTermination(10, TimeUnit.MINUTES);

      for (int i = 0; i < QTY; ++i) {
        examples.get(i).close();
      }
      baseCount.close();
    }
    Thread.sleep(Integer.MAX_VALUE);
  }

  @Override
  <span style="color:#c792ea;font-style:italic;">public</span> void stateChanged(CuratorFr
mework arg0, ConnectionState arg1) {
    System.out.println("State changed: " + arg1.toString());
  }

  @Override
```

```
    <span style="color:#c792ea;font-style:italic;">public</span> void countHasChanged(Share
CountReader sharedCount, int newCount) throws Exception {
        System.out.println("Counter's value is changed to " + newCount);
    }
}
```

在这个例子中，我们使用baseCount来监听计数值(addListener方法来添加SharedCountListener)。 任意的SharedCount， 只要使用相同的path，都可以得到这个计数值。 然后我们使用5个线程为计值增加一个10以内的随机数。相同的path的SharedCount对计数值进行更改，将会回调给baseCount SharedCountListener。

```
count.trySetCount(count.getVersionedValue(), count.getCount() + rand.nextInt(10))
```

这里我们使用trySetCount去设置计数器。 **第一个参数提供当前的VersionedValue,如果期间其它client更新了此计数值， 你的更新可能不成功， 但是这时你的client更新了最新的值，所以失败了你可尝试再更新一次。 而setCount是强制更新计数器的值。**

注意计数器必须start,使用完之后必须调用close关闭它。

强烈推荐使用ConnectionStateListener。 在本例中SharedCountListener扩展ConnectionStateListner。

分布式long计数器—DistributedAtomicLong

再看一个Long类型的计数器。 除了计数的范围比SharedCount大了之外， 它首先尝试使用乐观锁的式设置计数器， 如果不成功(比如期间计数器已经被其它client更新了)， 它使用InterProcessMutex式来更新计数值。

可以从它的内部实现DistributedAtomicValue.trySet()中看出：

```
    AtomicValue<byte[]>    trySet(MakeValue makeValue) throws Exception
    {
        MutableAtomicValue<byte[]>  result = new MutableAtomicValue<byte[]>(null, null, false
;

        tryOptimistic(result, makeValue);
        if ( !result.succeeded() && (mutex != null) )
        {
            tryWithMutex(result, makeValue);
        }

        return result;
    }
```

此计数器有一系列的操作：

- get(): 获取当前值
- increment()： 加一
- decrement(): 减一
- add()： 增加特定的值
- subtract(): 减去特定的值

- trySet(): 尝试设置计数值
- forceSet(): 强制设置计数值

你**必须**检查返回结果的succeeded()， 它代表此操作是否成功。 如果操作成功， preValue()代表操前的值， postValue()代表操作后的值。

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedAtomicLongD
mo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> int QTY = 5;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
amples/counter";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows IOException, Exception {
    List<DistributedAtomicLong> examples = Lists.newArrayList();
    try (TestingServer server = new TestingServer()) {
      CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectStrin
(), new ExponentialBackoffRetry(1000, 3));
      client.start();
      ExecutorService service = Executors.newFixedThreadPool(QTY);
      for (int i = 0; i < QTY; ++i) {
        <span style="color:#c792ea;font-style:italic;">final</span> DistributedAtomicLong co
nt = new DistributedAtomicLong(client, PATH, new RetryNTimes(10, 10));

        examples.add(count);
        Callable<Void> task = () -> {
          try {
            AtomicValue<Long> value = count.increment();
            System.out.println("succeed: " + value.succeeded());
            if (value.succeeded())
              System.out.println("Increment: from " + value.preValue() + " to " + value.postVal
e());
          } catch (Exception e) {
            e.printStackTrace();
          }
          return null;
        };
        service.submit(task);
      }

      service.shutdown();
      service.awaitTermination(10, TimeUnit.MINUTES);
      Thread.sleep(Integer.MAX_VALUE);
    }
  }
}
```

分布式队列

使用Curator也可以简化Ephemeral Node (**临时节点**)的操作。Curator也提供ZK Recipe的分布式队实现。 利用ZK的 PERSISTENTS_EQUENTIAL节点， 可以保证放入到队列中的项目是按照顺序排队。 如果单一的消费者从队列中取数据， 那么它是先入先出的，这也是队列的特点。 如果你严格要求

序，你就的使用单一的消费者，可以使用Leader选举只让Leader作为唯一的消费者。

但是， 根据Netflix的Curator作者所说， ZooKeeper真心不适合做Queue，或者说ZK没有实现一个的Queue，详细内容可以看 Tech Note 4， 原因有五：

1. ZK有1MB 的传输限制。 实践中ZNode必须相对较小，而队列包含成千上万的消息，非常的大。

2. 如果有很多节点，ZK启动时相当的慢。 而使用queue会导致好多ZNode. 你需要显著增大 initLimit 和 syncLimit.

3. ZNode很大的时候很难清理。Netflix不得不创建了一个专门的程序做这事。

4. 当很大量的包含成千上万的子节点的ZNode时， ZK的性能变得不好

5. ZK的数据库完全放在内存中。 大量的Queue意味着会占用很多的内存空间。

尽管如此， Curator还是创建了各种Queue的实现。 如果Queue的数据量不太多，数据量不太大的况下，酌情考虑，还是可以使用的。

分布式队列—DistributedQueue

DistributedQueue是最普通的一种队列。 它设计以下四个类：

- QueueBuilder - 创建队列使用QueueBuilder,它也是其它队列的创建类
- QueueConsumer - 队列中的消息消费者接口
- QueueSerializer - 队列消息序列化和反序列化接口，提供了对队列中的对象的序列化和反序列化
- DistributedQueue - 队列实现类

QueueConsumer是消费者，它可以接收队列的数据。处理队列中的数据的代码逻辑可以放在Queue onsumer.consumeMessage()中。

正常情况下先将消息从队列中移除，再交给消费者消费。但这是两个步骤，不是原子的。可以调用Bui der的lockPath()消费者加锁，当消费者消费数据时持有锁，这样其它消费者不能消费此消息。如果消失败或者进程死掉，消息可以交给其它进程。这会带来一点性能的损失。最好还是单消费者模式使用列。

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedQueueDemo {

    <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
ample/queue";

    <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
        TestingServer server = new TestingServer();
        CuratorFramework clientA = CuratorFrameworkFactory.newClient(server.getConnectString
), new ExponentialBackoffRetry(1000, 3));
        clientA.start();
        CuratorFramework clientB = CuratorFrameworkFactory.newClient(server.getConnectString
), new ExponentialBackoffRetry(1000, 3));
        clientB.start();
        DistributedQueue<String> queueA;
        QueueBuilder<String> builderA = QueueBuilder.builder(clientA, createQueueConsumer("
"), createQueueSerializer(), PATH);
        queueA = builderA.buildQueue();
```

```
        queueA.start();

        DistributedQueue<String> queueB;
        QueueBuilder<String> builderB = QueueBuilder.builder(clientB, createQueueConsumer("B
), createQueueSerializer(), PATH);
        queueB = builderB.buildQueue();
        queueB.start();
        for (int i = 0; i < 100; i++) {
            queueA.put(" test-A-" + i);
            Thread.sleep(10);
            queueB.put(" test-B-" + i);
        }
        Thread.sleep(1000 * 10);// <span style="font-family:'AR PL UKai HK';">等待消息消费完成
</span>       queueB.close();
        queueA.close();
        clientB.close();
        clientA.close();
        System.out.println("OK!");
    }

    /**
     * <span style="font-family:'AR PL UKai HK';">队列消息序列化实现类
</span>     */
    <span style="color:#c792ea;font-style:italic;">private static</span> QueueSerializer<Strin
> createQueueSerializer() {
        return new QueueSerializer<String>() {
            @Override
            <span style="color:#c792ea;font-style:italic;">public</span> byte[] serialize(String item)
{
                return item.getBytes();
            }

            @Override
            <span style="color:#c792ea;font-style:italic;">public</span> String deserialize(byte[] by
es) {
                return new String(bytes);
            }
        };
    }

    /**
     * <span style="font-family:'AR PL UKai HK';">定义队列消费者
</span>     */
    <span style="color:#c792ea;font-style:italic;">private static</span> QueueConsumer<Strin
> createQueueConsumer(<span style="color:#c792ea;font-style:italic;">final</span> String n
me) {
        return new QueueConsumer<String>() {
            @Override
            <span style="color:#c792ea;font-style:italic;">public</span> void stateChanged(Curato
Framework client, ConnectionState newState) {
                System.out.println("<span style="font-family:'AR PL UKai HK';">连接状态改变</span>:
 + newState.name());
            }
```

```
        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> void consumeMessage(Str
ng message) throws Exception {
            System.out.println("<span style="font-family:'AR PL UKai HK';">消费消息</span>(" +
ame + "): " + message);
        }
    };
  }
}
```

例子中定义了两个分布式队列和两个消费者，因为PATH是相同的，会存在消费者抢占消费消息的情
。

带Id的分布式队列—DistributedIdQueue

DistributedIdQueue和上面的队列类似，**但是可以为队列中的每一个元素设置一个ID。** 可以通过ID
队列中任意的元素移除。 它涉及几个类：

- QueueBuilder
- QueueConsumer
- QueueSerializer
- DistributedQueue

通过下面方法创建：

```
builder.buildIdQueue()
```

放入元素时：

```
queue.put(aMessage, messageId);
```

移除元素时：

```
int numberRemoved = queue.remove(messageId);
```

在这个例子中， 有些元素还没有被消费者消费前就移除了，这样消费者不会收到删除的消息。

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedIdQueueDemo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
ample/queue";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
    TestingServer server = new TestingServer();
    CuratorFramework client = null;
    DistributedIdQueue<String> queue = null;
    try {
      client = CuratorFrameworkFactory.newClient(server.getConnectString(), new Exponentia
BackoffRetry(1000, 3));
```

```
        client.getCuratorListenable().addListener((client1, event) -> System.out.println("CuratorE
ent: " + event.getType().name()));

        client.start();
        QueueConsumer<String> consumer = createQueueConsumer();
        QueueBuilder<String> builder = QueueBuilder.builder(client, consumer, createQueueSer
alizer(), PATH);
        queue = builder.buildIdQueue();
        queue.start();

        for (int i = 0; i < 10; i++) {
          queue.put(" test-" + i, "Id" + i);
          Thread.sleep((long) (15 * Math.random()));
          queue.remove("Id" + i);
        }

        Thread.sleep(20000);

      } catch (Exception ex) {

      } finally {
        CloseableUtils.closeQuietly(queue);
        CloseableUtils.closeQuietly(client);
        CloseableUtils.closeQuietly(server);
      }
    }

    <span style="color:#c792ea;font-style:italic;">private static</span> QueueSerializer<Strin
> createQueueSerializer() {
      return new QueueSerializer<String>() {

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> byte[] serialize(String item)
{
          return item.getBytes();
        }

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> String deserialize(byte[] by
es) {
          return new String(bytes);
        }

      };
    }

    <span style="color:#c792ea;font-style:italic;">private static</span> QueueConsumer<Strin
> createQueueConsumer() {

      return new QueueConsumer<String>() {

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> void stateChanged(Curato
Framework client, ConnectionState newState) {
```

```
            System.out.println("connection new state: " + newState.name());
        }

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> void consumeMessage(Str
ng message) throws Exception {
            System.out.println("consume one message: " + message);
        }

    };
  }
}
```

优先级分布式队列—DistributedPriorityQueue

优先级队列对队列中的元素按照优先级进行排序。 **Priority越小， 元素越靠前， 越先被消费掉。** 它及下面几个类：

- QueueBuilder
- QueueConsumer
- QueueSerializer
- DistributedPriorityQueue

通过builder.buildPriorityQueue(minItemsBeforeRefresh)方法创建。 当优先级队列得到元素增删息时，它会暂停处理当前的元素队列，然后刷新队列。minItemsBeforeRefresh指定刷新前当前活动队列的最小数量。 主要设置你的程序可以容忍的不排序的最小值。

放入队列时需要指定优先级：

```
queue.put(aMessage, priority);
```

例子：

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedPriorityQueueD
mo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
ample/queue";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
    TestingServer server = new TestingServer();
    CuratorFramework client = null;
    DistributedPriorityQueue<String> queue = null;
    try {
      client = CuratorFrameworkFactory.newClient(server.getConnectString(), new Exponentia
BackoffRetry(1000, 3));
      client.getCuratorListenable().addListener((client1, event) -> System.out.println("CuratorE
ent: " + event.getType().name()));

      client.start();
      QueueConsumer<String> consumer = createQueueConsumer();
```

```
        QueueBuilder<String> builder = QueueBuilder.builder(client, consumer, createQueueSer
alizer(), PATH);
        queue = builder.buildPriorityQueue(0);
        queue.start();

        for (int i = 0; i < 10; i++) {
            int priority = (int) (Math.random() * 100);
            System.out.println("test-" + i + " priority:" + priority);
            queue.put("test-" + i, priority);
            Thread.sleep((long) (50 * Math.random()));
        }

        Thread.sleep(20000);

    } catch (Exception ex) {

    } finally {
        CloseableUtils.closeQuietly(queue);
        CloseableUtils.closeQuietly(client);
        CloseableUtils.closeQuietly(server);
    }
  }

  <span style="color:#c792ea;font-style:italic;">private static</span> QueueSerializer<Strin
> createQueueSerializer() {
    return new QueueSerializer<String>() {

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> byte[] serialize(String item)
{
            return item.getBytes();
        }

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> String deserialize(byte[] by
es) {
            return new String(bytes);
        }

    };
  }

  <span style="color:#c792ea;font-style:italic;">private static</span> QueueConsumer<Strin
> createQueueConsumer() {

    return new QueueConsumer<String>() {

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> void stateChanged(Curato
Framework client, ConnectionState newState) {
            System.out.println("connection new state: " + newState.name());
        }

        @Override
```

```
        <span style="color:#c792ea;font-style:italic;">public</span> void consumeMessage(Str
ng message) throws Exception {
            Thread.sleep(1000);
            System.out.println("consume one message: " + message);
        }

    };
  }

}
```

有时候你可能会有错觉，优先级设置并没有起效。那是因为优先级是对于队列积压的元素而言，如果费速度过快有可能出现在后一个元素入队操作之前前一个元素已经被消费，这种情况下DistributedPrirityQueue会退化为DistributedQueue。

分布式延迟队列—DistributedDelayQueue

JDK中也有DelayQueue，不知道你是否熟悉。 DistributedDelayQueue也提供了类似的功能， 元有个delay值， 消费者隔一段时间才能收到元素。 涉及到下面四个类。

- QueueBuilder
- QueueConsumer
- QueueSerializer
- DistributedDelayQueue

通过下面的语句创建：

```
QueueBuilder<MessageType>   builder = QueueBuilder.builder(client, consumer, serializer, p
th);
... more builder method calls as needed ...
DistributedDelayQueue<MessageType> queue = builder.buildDelayQueue();
```

放入元素时可以指定delayUntilEpoch：

```
queue.put(aMessage, delayUntilEpoch);
```

注意delayUntilEpoch不是离现在的一个时间间隔， 比如20毫秒，而是未来的一个时间戳，如 System currentTimeMillis() + 10秒。 如果delayUntilEpoch的时间已经过去，消息会立刻被消费者接收。

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedDelayQueueD
mo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
ample/queue";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
    TestingServer server = new TestingServer();
    CuratorFramework client = null;
    DistributedDelayQueue<String> queue = null;
    try {
      client = CuratorFrameworkFactory.newClient(server.getConnectString(), new Exponentia
```

```
BackoffRetry(1000, 3));
        client.getCuratorListenable().addListener((client1, event) -> System.out.println("CuratorE
ent: " + event.getType().name()));

        client.start();
        QueueConsumer<String> consumer = createQueueConsumer();
        QueueBuilder<String> builder = QueueBuilder.builder(client, consumer, createQueueSer
alizer(), PATH);
        queue = builder.buildDelayQueue();
        queue.start();

        for (int i = 0; i < 10; i++) {
          queue.put("test-" + i, System.currentTimeMillis() + 10000);
        }
        System.out.println(new Date().getTime() + ": already put all items");

        Thread.sleep(20000);

      } catch (Exception ex) {

      } finally {
        CloseableUtils.closeQuietly(queue);
        CloseableUtils.closeQuietly(client);
        CloseableUtils.closeQuietly(server);
      }
    }

    <span style="color:#c792ea;font-style:italic;">private static</span> QueueSerializer<Strin
> createQueueSerializer() {
      return new QueueSerializer<String>() {

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> byte[] serialize(String item)
{
          return item.getBytes();
        }

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> String deserialize(byte[] by
es) {
          return new String(bytes);
        }

      };
    }

    <span style="color:#c792ea;font-style:italic;">private static</span> QueueConsumer<Strin
> createQueueConsumer() {

      return new QueueConsumer<String>() {

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> void stateChanged(Curato
Framework client, ConnectionState newState) {
```

```
            System.out.println("connection new state: " + newState.name());
        }

        @Override
        <span style="color:#c792ea;font-style:italic;">public</span> void consumeMessage(Strng message) throws Exception {
            System.out.println(new Date().getTime() + ": consume one message: " + message);
        }

    };
  }
}
```

SimpleDistributedQueue

前面虽然实现了各种队列，但是你注意到没有，这些队列并没有实现类似JDK一样的接口。 SimpleDistributedQueue提供了和JDK基本一致的接口(但是没有实现Queue接口)。 创建很简单：

```
<span style="color:#c792ea;font-style:italic;">public</span> SimpleDistributedQueue(CuratoFramework client,String path)
```

增加元素：

```
public boolean offer(byte[] data) throws Exception
```

删除元素：

```
<span style="color:#c792ea;font-style:italic;">public</span> byte[] take() throws Exception
```

另外还提供了其它方法：

```
<span style="color:#c792ea;font-style:italic;">public</span> byte[] peek() throws Exception
<span style="color:#c792ea;font-style:italic;">public</span> byte[] poll(long timeout, TimeUit unit) throws Exception
<span style="color:#c792ea;font-style:italic;">public</span> byte[] poll() throws Exception
<span style="color:#c792ea;font-style:italic;">public</span> byte[] remove() throws Exceptio

<span style="color:#c792ea;font-style:italic;">public</span> byte[] element() throws Exception
```

没有add方法， 多了take方法。

take方法在成功返回之前会被阻塞。 而poll方法在队列为空时直接返回null。

```
<span style="color:#c792ea;font-style:italic;">public</span> class SimpleDistributedQueueDmo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/eample/queue";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)hrows Exception {
    TestingServer server = new TestingServer();
    CuratorFramework client = null;
```

```
    SimpleDistributedQueue queue;
    try {
      client = CuratorFrameworkFactory.newClient(server.getConnectString(), new ExponentialBackoffRetry(1000, 3));
      client.getCuratorListenable().addListener((client1, event) -> System.out.println("CuratorEvent: " + event.getType().name()));
      client.start();
      queue = new SimpleDistributedQueue(client, PATH);
      Producer producer = new Producer(queue);
      Consumer consumer = new Consumer(queue);
      new Thread(producer, "producer").start();
      new Thread(consumer, "consumer").start();
      Thread.sleep(20000);
    } catch (Exception ex) {

    } finally {
      CloseableUtils.closeQuietly(client);
      CloseableUtils.closeQuietly(server);
    }
  }

  <span style="color:#c792ea;font-style:italic;">public static</span> class Producer implements Runnable {

    <span style="color:#c792ea;font-style:italic;">private</span> SimpleDistributedQueue queue;

    <span style="color:#c792ea;font-style:italic;">public</span> Producer(SimpleDistributedQueue queue) {
      <span style="color:#c792ea;font-style:italic;">this</span>.queue = queue;
    }

    @Override
    <span style="color:#c792ea;font-style:italic;">public</span> void run() {
      for (int i = 0; i < 100; i++) {
        try {
          boolean flag = queue.offer(("zjc-" + i).getBytes());
          if (flag) {
            System.out.println("<span style="font-family:'AR PL UKai HK';">发送一条消息成功</span>" + "zjc-" + i);
          } else {
            System.out.println("<span style="font-family:'AR PL UKai HK';">发送一条消息失败</span>" + "zjc-" + i);
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }

  <span style="color:#c792ea;font-style:italic;">public static</span> class Consumer implements Runnable {
```

```
    <span style="color:#c792ea;font-style:italic;">private</span> SimpleDistributedQueue q
eue;

    <span style="color:#c792ea;font-style:italic;">public</span> Consumer(SimpleDistribute
Queue queue) {
        <span style="color:#c792ea;font-style:italic;">this</span>.queue = queue;
    }

    @Override
    <span style="color:#c792ea;font-style:italic;">public</span> void run() {
      try {
        byte[] datas = queue.take();
        System.out.println("<span style="font-family:'AR PL UKai HK';">消费一条消息成功：</
pan>" + new String(datas, "UTF-8"));
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

}
```

但是实际上发送了100条消息，消费完第一条之后，后面的消息无法消费，目前没找到原因。查看一官方文档推荐的demo使用下面几个Api：

```
Creating a SimpleDistributedQueue

<span style="color:#c792ea;font-style:italic;">public</span> SimpleDistributedQueue(Curato
Framework client,
                          String path)
Parameters:
client - the client
path - path to store queue nodes
Add to the queue

<span style="color:#c792ea;font-style:italic;">public</span> boolean offer(byte[] data)
          throws Exception
Inserts data into queue.
Parameters:
data - the data
Returns:
true if data was successfully added
Take from the queue

<span style="color:#c792ea;font-style:italic;">public</span> byte[] take()
         throws Exception
Removes the head of the queue and returns it, blocks until it succeeds.
Returns:
The former head of the queue
NOTE: see the Javadoc for additional methods
```

但是实际使用发现还是存在消费阻塞问题。

分布式屏障—Barrier

分布式Barrier是这样一个类： 它会阻塞所有节点上的等待进程，直到某一个被满足， 然后所有的节继续进行。

比如赛马比赛中， 等赛马陆续来到起跑线前。 一声令下，所有的赛马都飞奔而出。

DistributedBarrier

DistributedBarrier类实现了栅栏的功能。 它的构造函数如下：

```
<span style="color:#c792ea;font-style:italic;">public</span> DistributedBarrier(CuratorFram
work client, String barrierPath)
Parameters:
client - client
barrierPath - path to use as the barrier
```

首先你需要设置栅栏，它将阻塞在它上面等待的线程:

```
setBarrier();
```

然后需要阻塞的线程调用方法等待放行条件:

```
public void waitOnBarrier()
```

当条件满足时，移除栅栏，所有等待的线程将继续执行：

```
removeBarrier();
```

异常处理 DistributedBarrier 会监控连接状态，当连接断掉时waitOnBarrier()方法会抛出异常。

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedBarrierDemo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> int QTY = 5;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/e
amples/barrier";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
hrows Exception {
    try (TestingServer server = new TestingServer()) {
      CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectStrin
(), new ExponentialBackoffRetry(1000, 3));
      client.start();
      ExecutorService service = Executors.newFixedThreadPool(QTY);
      DistributedBarrier controlBarrier = new DistributedBarrier(client, PATH);
      controlBarrier.setBarrier();

      for (int i = 0; i < QTY; ++i) {
        <span style="color:#c792ea;font-style:italic;">final</span> DistributedBarrier barrier
 new DistributedBarrier(client, PATH);
        <span style="color:#c792ea;font-style:italic;">final</span> int index = i;
        Callable<Void> task = () -> {
          Thread.sleep((long) (3 * Math.random()));
          System.out.println("Client #" + index + " waits on Barrier");
```

```
                barrier.waitOnBarrier();
                System.out.println("Client #" + index + " begins");
                return null;
            };
            service.submit(task);
        }
        Thread.sleep(10000);
        System.out.println("all Barrier instances should wait the condition");
        controlBarrier.removeBarrier();
        service.shutdown();
        service.awaitTermination(10, TimeUnit.MINUTES);

        Thread.sleep(20000);
    }
  }
}
```

这个例子创建了controlBarrier来设置栅栏和移除栅栏。 我们创建了5个线程，在此Barrier上等待。后移除栅栏后所有的线程才继续执行。

如果你开始不设置栅栏，所有的线程就不会阻塞住。

双栅栏—DistributedDoubleBarrier

双栅栏允许客户端在计算的开始和结束时同步。当足够的进程加入到双栅栏时，进程开始计算， 当计完成时，离开栅栏。 双栅栏类是DistributedDoubleBarrier。 构造函数为:

```
<span style="color:#c792ea;font-style:italic;">public</span> DistributedDoubleBarrier(CuratrFramework client,
                                String barrierPath,
                                int memberQty)
Creates the barrier abstraction. memberQty is the number of members in the barrier. When eter() is called, it blocks until
all members have entered. When leave() is called, it blocks until all members have left.

Parameters:
client - the client
barrierPath - path to use
memberQty - the number of members in the barrier
```

memberQty是成员数量，当enter()方法被调用时，成员被阻塞，直到所有的成员都调用了enter()。当leave()方法被调用时，它也阻塞调用线程，直到所有的成员都调用了leave()。 就像百米赛跑比赛，发令枪响， 所有的运动员开始跑，等所有的运动员跑过终点线，比赛才结束。

DistributedDoubleBarrier会监控连接状态，当连接断掉时enter()和leave()方法会抛出异常。

```
<span style="color:#c792ea;font-style:italic;">public</span> class DistributedDoubleBarrierDmo {

  <span style="color:#c792ea;font-style:italic;">private static final</span> int QTY = 5;
  <span style="color:#c792ea;font-style:italic;">private static final</span> String PATH = "/eamples/barrier";

  <span style="color:#c792ea;font-style:italic;">public static</span> void main(String[] args)
```

```
hrows Exception {
      try (TestingServer server = new TestingServer()) {
         CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectStrin
(), new ExponentialBackoffRetry(1000, 3));
         client.start();
         ExecutorService service = Executors.newFixedThreadPool(QTY);
         for (int i = 0; i < QTY; ++i) {
            <span style="color:#c792ea;font-style:italic;">final</span> DistributedDoubleBarrier
arrier = new DistributedDoubleBarrier(client, PATH, QTY);
            <span style="color:#c792ea;font-style:italic;">final</span> int index = i;
            Callable<Void> task = () -> {

               Thread.sleep((long) (3 * Math.random()));
               System.out.println("Client #" + index + " enters");
               barrier.enter();
               System.out.println("Client #" + index + " begins");
               Thread.sleep((long) (3000 * Math.random()));
               barrier.leave();
               System.out.println("Client #" + index + " left");
               return null;
            };
            service.submit(task);
         }

         service.shutdown();
         service.awaitTermination(10, TimeUnit.MINUTES);
         Thread.sleep(Integer.MAX_VALUE);
      }
   }

}
```