目录

Curator 框架调研

1 Curator 简介

Curator client 用来替代 ZooKeeper 提供的类，它封装了底层的管理并提供了一些有用的工具。**Curator framework** 提供了高级的 API 来简化 ZooKeeper 的使用。它增加了很多基于 ZooKeeper 的特性，帮助管理 ZooKeeper 的连接以及重试操作。**Curator Recipes** 提供解决方案，或者叫实现方案，是指 ZooKeeper 的使用方法，比如分布式的配置管理，Leader 选举等。除此之外，**Curator Test** 提供了基于 ZooKeeper 的单元测试工具，可以在未安装 zookeeper server 的情况下直接进行集群测试。

1.1 Curator 组件概览

Recipes：通用 ZooKeeper 技巧("recipes")的实现. 建立在 Curator Framework 之上
Framework：简化 zookeeper 使用的高级. 增加了很多建立在 zookeeper 之上的特性. 管理复杂连接处理和重试操作
Utilities：各种工具类
Client：ZooKeeper 本身提供的类的替代者。负责底层的开销以及一些工具
Errors：Curator 怎样来处理错误和异常
Extensions：curator-recipes 包实现了通用的技巧，这些技巧在 ZooKeeper 文档中有介绍。为了避免是这个包(package)变得巨大，recipes/applications 将会放入一个独立的 extension 包下。并使用命名规则 curator-x-name

1.2 Maven/Artifacts

支持 maven，具体配置请查 maven 库

1.3 Apache Curator Recipes

1） Elections(选举)
Leader Latch - 在分布式计算中，leader 选举是在几台节点中指派单一的进程作为任务组织者的过程。在任务开始前，所有的网络节点都不知道哪一个节点会作为任务的 leader 或 coordinator. 一旦 leader 选举算法被执行，网络中的每个节点都将知道一个特别的唯一的节点作为任务 leader
Leader Election - 初始的 leader 选举实现
2） Locks(锁)
Shared Reentrant Lock - 全功能的分布式锁。任何一刻不会有两个 client 同时拥有锁
Shared Lock - 与 Shared Reentrant Lock 类似但是不是重入的
Shared Reentrant Read Write Lock - 类似 Java 的读写锁，但是是分布式的
Shared Semaphore - 跨 JVM 的计数信号量
Multi Shared Lock - 将多个锁看成整体，要不全部 acquire 成功，要不 acquire 全部失败。release 也是释放全部锁
3） Barriers(障碍)
Barrier - 分布式的 barriers。会阻塞全部的节点的处理，直到条件满足，所有的节点会继续执行
Double Barrier - 双 barrier 允许客户端在一个计算开始点和结束点保持同步。当足够的进程加入 barrier,进程开始它们的计算，当所有的进程完成计算才离开

4） Counters(计数器)

Shared Counter - 管理一个共享的整数 integer.所有监控同一 path 的客户端都会得到最新的值(ZK 的一致性保证)

Distributed Atomic Long - 尝试原子增加的计数器首先它尝试乐观锁.如果失败可选的 InterProcessMutex 会被采用.不管是 optimistic 还是 mutex,重试机制都被用来尝试增加值

5） Caches(缓存)

Path Cache - Path Cache 是用来监控子节点的。每当一个子节点“曾加、更新或删除”，将会触发事件，事件就是注册了的 PathChildrenCacheListener 实例执行。Path Cache 的功能主要由 PathChildrenCache 类提供。

Node Cache - 用于监控节点，当节点数据被修改或节点被删除，就会触发事件，事件就是注册了的 NodeCacheListener 实例执行。Node Cache 的功能主要由 NodeCache 类提供。

6） Nodes(节点)

Persistent Ephemeral Node - 临时节点，可以通过连接或会话中断一个临时节点。

7） Queues(队列)

Distributed Queue - 分布式的 ZK 队列

Distributed Id Queue - 分布式的 ZK 队列，它支持分配 ID 来添加到队列中的项目的替换版本

Distributed Priority Queue - 分布式优先级的 ZK 队列

Distributed Delay Queue - 分布式的延迟 ZK 队列

Simple Distributed Queue - 一个简单的替代自带的 ZK 分布式队列(Distributed Queue)

1.4 Apache Curator Framework

Curator framework 提供了高级 API，极大的简化了 ZooKeeper 的使用。它在 ZooKeeper 基础上增加了很多特性，可以管理与 ZooKeeper 的连接和重试机制。这些特性包括：自动连接管理：有些潜在的错误情况需要让 ZooKeeper client 重建连接和重试。Curator 可以自动地和透明地处理这些情况 Cleaner API：简化原始的 ZooKeeper 方法，事件等 提供现代的流式接口

1.4.1 产生 Curator framework 实例

使用 CuratorFrameworkFactory 产生 framework 实例。 CuratorFrameworkFactory 既提供了 factory 方法也提供了 builder 来创建实例。**CuratorFrameworkFactory 是线程安全的**。你应该在应用中为**单一的 ZooKeeper 集群共享唯一的 CuratorFramework 实例**。

工厂方法(newClient())提供了一个简单的方式创建实例。Builder 可以使用更多的参数控制生成的实例。一旦生成 framework 实例， **必须调用 start 方法启动它。应用结束时应该调用 close 方法关闭它**。

1.4.2 CuratorFramework API

Fluent 风格（链式编程）

```
client.create().forPath("/head", new byte[0]);
client.delete().inBackground().forPath("/head");
client.create().withMode(CreateMode.EPHEMERAL_SEQUENTIAL).forPath("/head/child", new byte[0]);
```

```
4.  client.getData().watched().inBackground().forPath("/test");
```

CuratorFramework 类重要方法说明

create() 开始创建操作，可以调用额外的方法(比如方式 mode 或者后台执行 background)并在最后调用 forPath()指定要操作的 ZNode

delete() 开始删除操作. 可以调用额外的方法(版本或者后台处理 version or background)并在最后调用 forPath()指定要操作的 ZNode

checkExists() 开始检查 ZNode 是否存在的操作. 可以调用额外的方法(监控或者后台处理)并在最后调用 forPath()指定要操作的 ZNode

getData() 开始获得 ZNode 节点数据的操作. 可以调用额外的方法(监控、后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 ZNode

setData() 开始设置 ZNode 节点数据的操作. 可以调用额外的方法(版本或者后台处理) 并在最后调用 forPath()指定要操作的 ZNode

getChildren() 开始获得 ZNode 的子节点列表。 以调用额外的方法(监控、后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 ZNode

inTransaction() 开始是原子 ZooKeeper 事务. 可以复合 create, setData, check, and/or delete 等操作然后调用 commit()作为一个原子操作提交

通知 Notifications

服务于后台操作和监控(watch)的通知通过 ClientListener 接口发布。你通过 CuratorFramework 实例的 addListener 方法可以注册监听器。其事件触发时间说明:

CuratorListenable：当使用后台线程操作时，后台线程执行完成就会触发，例如：client.getData().inBackground().forPath("/test");后台获取节点数据，获取完成之后触发。

ConnectionStateListenable：当连接状态变化时触发。

UnhandledErrorListenable：当后台操作发生异常时触发。

事件类型	事件返回数据
CREATE	getResultCode() and getPath()
DELETE	getResultCode() and getPath()
EXISTS	getResultCode(), getPath() and getStat()
GET_DATA	getResultCode(), getPath(), getStat() and getData()
SET_DATA	getResultCode(), getPath() and getStat()
CHILDREN	getResultCode(), getPath(), getStat(), getChildren()
SYNC	getResultCode(), getStat()
GET_ACL	getResultCode(), getACLList()
SET_ACL	getResultCode()
TRANSACTION	getResultCode(), getOpResults()
WATCHED	getWatchedEvent()
GET_CONFIG	getResultCode(), getData()
RECONFIG	getResultCode(), getData()

命名空间

你可以使用命名空间 Namespace 避免多个应用的节点的名称冲突。CuratorFramework 提供了命名空间的概念，这样 CuratorFramework 会为它的 API 调用的 path 加上命名空间：

临时客户端

curator 还提供了临时的 CuratorFramework：CuratorTempFramework，一定时间不活动后连接会被关闭。创建 builder 时不是调用 build()而是调用 buildTemp()。3 分钟不活动连接就被关闭，你也可以指定不活动的时间。

Retry 策略

retry 策略可以改变 retry 的行为。 它抽象出 RetryPolicy 接口， 包含一个方法 public boolean allowRetry(int retryCount, long elapsedTimeMs);。 在 retry 被尝试执行前，allowRetry()被调用，并且将当前的重试次数和操作已用时间作为参数. 如果返回 true， retry 被执行。否则异常被抛出。Curator 本身提供了几个策略

ExponentialBackoffRetry:重试一定次数，每次重试 sleep 更多的时间

RetryNTimes:重试 N 次

RetryOneTime:重试一次

RetryUntilElapsed:重试一定的时间

1.5 Apache Curator Utilities

Curator 提供了一组工具类和方法用来测试基于 Curator 的应用。 并且提供了操作 ZNode 辅助类以及其它一些数据结构

1.5.1 Test Server

curator-test 提供了 TestingServer 类。这个类创建了一个本地的，同进程的 ZooKeeper 服务器用来测试。

```
public static void main(String[] args) throws Exception
{
    TestingServer server = new TestingServer();
    CuratorFramework client = CuratorFrameworkFactory.newClient(server.getConnectString(), new
ExponentialBackoffRetry(1000, 3));
    client.getConnectionStateListenable().addListener(new ConnectionStateListener()
    {
        @Override
        public void stateChanged(CuratorFramework client, ConnectionState newState)
        {
            System.out.println("连接状态:" + newState.name());
        }
    });
    client.start();
    System.out.println(client.getChildren().forPath("/"));
    client.create().forPath("/test");
    System.out.println(client.getChildren().forPath("/"));
    CloseableUtils.closeQuietly(client);
    CloseableUtils.closeQuietly(server);
    System.out.println("OK!"); }
```

1.5.2 Test Cluster

curator-test提供了TestingCluster类。这个类创建了一个内部的ZooKeeper集群用来测试。

```
public static void main(String[] args) throws Exception
{
    TestingCluster cluster = new TestingCluster(3);
    cluster.start();
    for (TestingZooKeeperServer server : cluster.getServers())
    {
        System.out.println(server.getInstanceSpec());
    }
    cluster.stop();
    CloseableUtils.closeQuietly(cluster);
    System.out.println("OK！ ");
}
```

1.5.3 ZKPaths

提供了各种静态方法来操作 ZNode:

getNodeFromPath: 从一个全路径中得到节点名， 比如 "/one/two/three" 返回 "three"

mkdirs: 确保所有的节点都已被创建

getSortedChildren: 得到一个给定路径的子节点， 按照 sequence number 排序

makePath: 给定父路径和子节点，创建一个全路径

1.5.4 EnsurePath

确保一个特定的路径被创建。当它第一次使用时，一个同步 ZKPaths.mkdirs(ZooKeeper, String)调用被触发来确保完整的路径都已经被创建。后续的调用将不是同步操作.用法

```
EnsurePath          ensurePath = new EnsurePath(aFullPathToEnsure);
...
String              nodePath = aFullPathToEnsure + "/foo";
ensurePath.ensure(zk);    // first time syncs and creates if needed
zk.create(nodePath, ...);
...
ensurePath.ensure(zk);    // subsequent times are NOPs
zk.create(nodePath, ...);
```

注意： 此方法 `namespace` 会参与路径名字的创建。

1.5.5 Blocking Queue Consumer(阻塞队列消费者)、

请参看 Distributed Queue 和 Distributed Priority Queue。提供 JDK BlockingQueue 类似的行为。

1.5.6 Queue Sharder

由于 zookeeper 传输层的限制，单一的队列如果超过 10K 的元素会被分割(break)。这个类为多个分布式队列提供了一个 facade。它监控队列，如果一个队列超过这个阈值，一个新的队列就被创建。在这些队列中 Put 是分布式的。

1.5.7 Reaper and ChildReaper

Reaper

可以用来删除锁的父路径。定时检查路径被加入到 reaper 中。当检查时，如果 path 没有子节点/路径，此路径将被删除。每个应用中 CLient 应该只创建一个 reaper 实例。必须将 lock path 加到这个 readper 中。reaper 会定时的检查删除它们。

ChildReaper

用来清除父节点下所有的空节点。定时的调用 getChildren()并将空节点加入到内部管理的 reaper 中。

注意：应该考虑使用 LeaderSelector 来运行 Reapers，因为它们不需要在每个 client 运行

1.6 Apache Curator Client

Curator client 使用底层的 API， 强烈推荐你是用 Curator Framework 代替使用 CuratorZookeeperClient，不推荐使用。

Curator leader 选举

在分布式计算中，leader election 是很重要的一个功能，这个选举过程是这样子的：指派一个进程作为组织者，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 或者 coordinator。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。除此之外，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。

Curator 有两种选举 recipe，你可以根据你的需求选择合适的。一种是 **leader latch**，另一种是 **leader election**

1 Leader Latch

1.1 LeaderLatch 的简单介绍

首先我们看一个使用 LeaderLatch 类来选举的例子。它的常用方法如下：

```
// 构造方法
public LeaderLatch(CuratorFramework client, String latchPath)
public LeaderLatch(CuratorFramework client, String latchPath, String id)
public LeaderLatch(CuratorFramework client, String latchPath, String id, CloseMode closeMode)

// 查看当前 LeaderLatch 实例是否是 leader
public boolean hasLeadership()

// 尝试让当前 LeaderLatch 实例称为 leader
public void await() throws InterruptedException, EOFException
public boolean await(long timeout, TimeUnit unit) throws InterruptedException
```

注意：必须启动 LeaderLatch: leaderLatch.start();一旦启动，LeaderLatch 会和其它使用相同 latch path 的其它 LeaderLatch 交涉，然后随机的选择其中一个作为 leader。

一旦不使用 LeaderLatch 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 leader。

1.2 异常处理

LeaderLatch 实例可以增加 ConnectionStateListener 来监听网络连接问题。当 SUSPENDED 或 LOST 时,leader 不再认为自己还是 leader.当 LOST 连接重连后 RECONNECTED,LeaderLatch 会删除先前的 ZNode 然后重新创建一个.

LeaderLatch 用户必须考虑导致 leader 丢失的连接问题。强烈推荐你使用 ConnectionStateListener。

2 Leader Election

Curator 还提供了另外一种选举方法。与 Leader latch 不同的是这种方法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权。主要涉及以下四个类:

LeaderSelector - 选举 Leader 的角色。

LeaderSelectorListener - 选举 Leader 时的事件监听。

LeaderSelectorListenerAdapter - 选举 Leader 时的事件监听，官方提供的适配器，用于用户扩展。

CancelLeadershipException - 取消 Leader 权异常

2.1 主要类介绍

重要的是 LeaderSelector 类，它的构造函数为:

```
public LeaderSelector(CuratorFramework client, String leaderPath, LeaderSelectorListener listener)
public LeaderSelector(CuratorFramework client, String leaderPath, ExecutorService executorService, LeaderSelectorListener listener)
public LeaderSelector(CuratorFramework client, String leaderPath, CloseableExecutorService executorService, LeaderSelectorListener listener)
```

类似 `LeaderLatch,必须 start: leaderSelector.start();` 一旦启动，当实例取得领导权时 LeaderSelectorListener 的 takeLeadership()方法被调用。而 takeLeadership()方法执行完毕时领导权会自动释放重新选举。当你不再使用 LeaderSelector 实例时，应该调用它的 close 方法。LeaderSelector 类中也有 hasLeadership()、getLeader()方法。

2.2 异常处理

LeaderSelectorListener 类继承 ConnectionStateListener。LeaderSelector 必须小心连接状态的改变。如果实例成为 leader,它应该相应 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.

注意：推荐处理方式是当收到 SUSPENDED 或 LOST 时抛出 CancelLeadershipException 异常. 这会导致 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。

3 leader latch 与 leader election 的区别

可以看出：**LeaderSelector** 与 **LeaderLatch** 的区别，通过 LeaderSelectorListener 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权。而 LeaderLatch 一根筋到死，除非调用 close 方法，否则它不会释放领导权。

Curator 分布式锁

1 可重入锁 Shared Reentrant Lock

首先我们先看一个**全局可重入的锁**(可以多次获取，不会被阻塞)。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，**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。**

1.1 可重入锁相关类介绍

它是由类 InterProcessMutex 来实现。它的主要方法:

```
// 构造方法
public InterProcessMutex(CuratorFramework client, String path)
public InterProcessMutex(CuratorFramework client, String path, LockInternalsDriver driver)

// 通过 acquire 获得锁,并提供超时机制:
public void acquire() throws Exception
public boolean acquire(long time, TimeUnit unit) throws Exception

// 撤销锁
public void makeRevocable(RevocationListener<InterProcessMutex> listener)
public void makeRevocable(final RevocationListener<InterProcessMutex> listener, Executor executor)
```

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。

2 不可重入锁 Shared Lock

这个锁和上面的相比，就是少了 Reentrant 的功能，也就意味着它不能在同一个线程中重入。lock.acquire(time, unit),第一次执行正常，第二次执行会报错.

注意：重入几次，就需要释放几次!!!

3 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，另外一个负责写操作。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降级成读锁， 比如请求写锁 --->读锁 ---->释放写锁。从读锁升级成写锁是不行的。

3. 1 可重入读写锁相关类介绍

可重入读写锁主要由两个类实现：InterProcessReadWriteLock、InterProcessMutex。使用时首先创建一个 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。

```
56 public class InterProcessReadWriteLock
57 {
58     private final InterProcessMutex readMutex;
59     private final InterProcessMutex writeMutex;
60
```

4 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 Curator 中称之为租约(Lease)。

有两种方式可以决定 semaphore 的最大租约数。第一种方式是有用户给定的 path 决定。第二种方式使用 SharedCountReader 类。

如果不使用 SharedCountReader,没有内部代码检查进程是否假定有 10 个租约而进程 B 假定有 20 个租约。 所以所有的实例必须使用相同的 numberOfLeases 值.

4.1 信号量实现类说明

主要类有:

- `InterProcessSemaphoreV2` - 信号量实现类
- `Lease` - 租约(单个信号)
- `SharedCountReader` - 计数器，用于计算最大租约数量

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，否则这些租约会丢失掉。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约会自动 `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：

```
public void returnLease(Lease lease)

public void returnAll(Collection<Lease> leases)
```

注意一次你可以请求多个租约，如果 `Semaphore` 当前的租约不够，则请求线程会被阻塞。同时还提供了超时的重载方法。

```
public Lease acquire() throws Exception

public Collection<Lease> acquire(int qty) throws Exception

public Lease acquire(long time, TimeUnit unit) throws Exception

public Collection<Lease> acquire(int qty, long time, TimeUnit
unit) throws Exception
```

注意：上面所讲的 4 种锁都是公平锁(`fair`)。从 `ZooKeeper` 的角度看，每个客户端都按照请求的顺序获得锁。相当公平。

5 多锁对象 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，它就是组锁的代表，在它上面的请求释放操作都会传递给它包含的所有的锁。

5.1 主要类说明

主要涉及两个类：

InterProcessMultiLock - 对锁对象实现类

InterProcessLock - 分布式锁接口类

它的构造函数需要包含的锁的集合，或者一组 ZooKeeper 的 path。用法和 Shared Lock 相同。

```
public InterProcessMultiLock(CuratorFramework client, List<String> paths)
public InterProcessMultiLock(List<InterProcessLock> locks)
```

Curator 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，知道某一个被满足，然后所有的节点继续进行。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。

1 栅栏 Barrier

1.1 DistributedBarrier 类说明

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

```
/**
 * @param client client
 * @param barrierPath path to use as the barrier
 */
public DistributedBarrier(CuratorFramework client, String barrierPath)
```

DistributedBarrier 构造函数中 barrierPath 参数用来确定一个栅栏，只要 barrierPath 参数相同(路径相同)就是同一个栅栏。通常情况下栅栏的使用如下：

1.主导 client 设置一个栅栏

2.其他客户端就会调用 waitOnBarrier()等待栅栏移除，程序处理线程阻塞

3.主导 client 移除栅栏，其他客户端的处理程序就会同时继续运行。

DistributedBarrier 类的主要方法如下：

setBarrier() - 设置栅栏

waitOnBarrier() - 等待栅栏移除

removeBarrier() - 移除栅栏

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。

2 双栅栏 Double Barrier

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，当计算完成时，离开栅栏。双栅栏类是 DistributedDoubleBarrier

2.1 DistributedDoubleBarrier 类说明

DistributedDoubleBarrier 类实现了双栅栏的功能。它的构造函数如下:

```
// client - the client
// barrierPath - path to use
// memberQty - the number of members in the barrier
public DistributedDoubleBarrier(CuratorFramework client, String barrierPath, int memberQty)
```

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，知道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。
注意：参数 memberQty 的值只是一个阈值，而不是一个限制值。当等待栅栏的数量大于或等于这个值栅栏就会打开!
与栅栏(DistributedBarrier)一样,双栅栏的 barrierPath 参数也是用来确定是否是同一个栅栏的,双栅栏的使用情况如下:

1.从多个客户端在同一个路径上创建双栅栏(DistributedDoubleBarrier),然后调用 enter()方法，等待栅栏数量达到 memberQty 时就可以进入栅栏。

2.栅栏数量达到 memberQty，多个客户端同时停止阻塞继续运行，直到执行 leave()方法，等待 memberQty 个数量的栅栏同时阻塞到 leave()方法中。

3.memberQty 个数量的栅栏同时阻塞到 leave()方法中，多个客户端的 leave()方法停止阻塞，继续运行。

DistributedDoubleBarrier 类的主要方法如下

enter()、enter(long maxWait, TimeUnit unit) - 等待同时进入栅栏

leave()、leave(long maxWait, TimeUnit unit) - 等待同时离开栅栏

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

Curator 计数器

这一篇文章我们将学习使用 Curator 来实现计数器。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，一个是用 int 来计数，一个用 long 来计数。

1 SharedCount

1.1 SharedCount 计数器介绍

这个类使用 int 类型来计数。 主要涉及三个类。

SharedCount - 管理一个共享的整数。所有看同样的路径客户端将有共享的整数（考虑 ZK 的正常一致性保证）的最高最新的值。

SharedCountReader - 一个共享的整数接口，并允许监听改变它的值。

SharedCountListener - 用于监听共享整数发生变化的监听器。

SharedCount 类代表计数器，可以为它增加一个 SharedCountListener，当计数器改变时此 Listener 可以监听到改变的事件，而 SharedCountReader 可以读取到最新的值，包括字面值

和带版本信息的值 VersionedValue。
注意：使用 SharedCount 之前需要调用 start()，使用完成之后需要调用 stop()

2 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方式设置计数器，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 方式来更新计数值。还记得 InterProcessMutex 是什么吗？它是我们前面讲的分布式可重入锁。这和上面的计数器的实现有显著的不同。

2. 1 DistributedAtomicLong 计数器介绍

DistributedAtomicLong 计数器和上面的计数器的实现有显著的不同，可以从它的内部实现 `DistributedAtomicValue.trySet` 中看出端倪。

```
public class DistributedAtomicLong implements DistributedAtomicNumber<Long>
{
    private final DistributedAtomicValue value;
    ......
}
public class DistributedAtomicValue
{
    ......
    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()`代表操作前的值，`postValue()`代表操作后的值。

注意：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()代表操作前的值，postValue()代表操作后的值。

Curator 缓存

可以利用 ZooKeeper 在集群的各个节点之间缓存数据。每个节点都可以得到最新的缓存的数据。Curator 提供了三种类型的缓存方式：Path Cache（监听一个节点的子节点）,Node Cache（监听一个节点） 和 Tree Cache（监听一个节点及其子节点）。**用来取代 watcher，非常重要!!!**

1 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 会改变它的状态，会包含最新的子节点，子节点的数据和状态。

1.1 Path Cache 介绍

Path Cache 的主要用途是监控某个节点的子节点，由于 Zookeeper 原生 API 的 watch 监控节点时注册一次只能触发一次，而 Path Cache 弥补了这个不足，它注册一次可以一直监控触发。

实际使用时会涉及到四个类：

PathChildrenCache - Path Cache 主要实现类

PathChildrenCacheEvent - 监听触发时的事件对象，包含事件相关信息

PathChildrenCacheListener - 监听器接口

ChildData - 子节点数据封装类

通过下面的构造函数创建 Path Cache:

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

注意：**使用 cache，必须调用它的 start 方法，不用之后调用 close 方法**。其中的 cacheData 参数用来设置是否缓存节点数据。

start 有两个，其中一个可以传入 StartMode，用来为初始的 cache 设置暖场方式(warm):

1.NORMAL: 初始时为空。

2.BUILD_INITIAL_CACHE: 在这个方法返回之前调用 rebuild()。

3.POST_INITIALIZED_EVENT: 当 Cache 初始化数据后发送一个 PathChildrenCacheEvent.Type#INITIALIZED 事件

PathChildrenCache.getListenable().addListener(PathChildrenCacheListener listener)可以增加 listener 监听缓存的改变。getCurrentData()方法返回一个 List 对象，可以遍历所有的子节点。

2 Node Cache

Path Cache 用来监控一个节点的子节点。当节点的数据修改或者删除时，Node Cache 能更新它的状态包含最新的改变。

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，不用之后调用 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。

3 Tree Node

这种类型的即可以监控节点的状态，还监控节点的子节点的状态，类似上面两种 cache 的组合。这也就是 Tree 的概念。它监控整个树中节点的状态。

3.1 Tree Node 介绍

Tree Node 可以监控整个树上的所有节点，涉及到下面四个类。

TreeCache - Tree Cache 实现类

TreeCacheListener - 监听器类

TreeCacheEvent - 触发的事件类

ChildData - 节点数据

注意：使用 cache，依然要调用它的 start 方法，不用之后调用 close 方法。

getCurrentChildren(path)返回监控节点下的某个节点的直接子节点数据，类型为 Map。 而 getCurrentData()返回监控节点的数据。

Curator 临时节点

使用 Curator 也可以简化 Ephemeral Node （临时节点）的操作。临时节点驻存在 ZooKeeper 中，当连接和 session 断掉时被删除。比如通过 ZooKeeper 发布服务，服务启动时将自己的信息注册为临时节点，当服务断掉时 ZooKeeper 将此临时节点删除，这样 client 就不会得到服务的信息了。

1 PersistentEphemeralNode 类

PersistentEphemeralNode 类代表临时节点。其构造函数如下:

```
/**
 * @param client     client instance
 * @param mode       creation/protection mode
 * @param basePath the base path for the node
 * @param data       data for the node
 */
public PersistentEphemeralNode(CuratorFramework client, Mode mode, String basePath,
byte[] initData)
```

其它参数还好理解，不好理解的是 PersistentEphemeralNode.Mode：

EPHEMERAL： 以 ZooKeeper 的 CreateMode.EPHEMERAL 方式创建节点。

EPHEMERAL_SEQUENTIAL: 如果 path 已经存在, 以 CreateMode.EPHEMERAL 创建节点，否则以 CreateMode.EPHEMERAL_SEQUENTIAL 方式创建节点。

PROTECTED_EPHEMERAL: 以 CreateMode.EPHEMERAL 创建，提供保护方式。

PROTECTED_EPHEMERAL_SEQUENTIAL: 类似 EPHEMERAL_SEQUENTIAL, 提供保护方式。

保护方式是指一种很边缘的情况：当服务器将节点创建好，但是节点名还没有返回给 client,这时候服务器可能崩溃了，然后此时 ZK session 仍然合法，所以此临时节点不会被删除。对于 client 来说，它无法知道哪个节点是它们创建的。

即使不是 sequential-ephemeral,也可能服务器创建成功但是客户端由于某些原因不知道创建的节点。

Curator 对这些可能无人看管的节点提供了**保护机制**。 这些节点创建时会加上一个

GUID。 如果节点创建失败正常的重试机制会发生。 重试时， 首先搜索父 path，根据 GUID 搜索节点，如果找到这样的节点， 则认为这些节点是第一次尝试创建时创建成功但丢失的节点，然后返回给调用者。
注意：**节点必须调用 start 方法启动。 不用时调用 close 方法**。PersistentEphemeralNode 内部自己处理错误状态。

Curator 队列

Curator 也提供 ZK Recipe 的分布式队列实现。利用 ZK 的 PERSISTENTSEQUENTIAL 节点，可以保证放入到队列中的项目是按照顺序排队的。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。如果你严格要求顺序，你就得使用单一的消费者，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 没有实现一个好的 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，数据量不太大的情况下，酌情考虑，还是可以使用的。

1 DistributedQueue

1.1 DistributedQueue 介绍

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 QueueConsumer.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 Builder 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。如果消费失败或者进程死掉，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队列。**

2 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 把队列中任意的元素移除。

2.1 DistributedIdQueue 结束

DistributedIdQueue 的使用与上面队列的区别是:
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);

3 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，元素月靠前，越先被消费掉。

3.1 DistributedPriorityQueue 介绍

通过 builder.buildPriorityQueue(minItemsBeforeRefresh)方法创建。

当优先级队列得到元素增删消息时，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的队列的最小数量。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);

4 DistributedDelayQueue

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

4.1 DistributedDelayQueue 介绍

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。

5 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 一致性的接口(但是没有实现 Queue 接口)。

5.1 SimpleDistributedQueue 介绍

`SimpleDistributedQueue` 常用方法:

```
// 创建
public SimpleDistributedQueue(CuratorFramework client, String path)

// 增加元素
public boolean offer(byte[] data) throws Exception

// 删除元素
public byte[] take() throws Exception

// 另外还提供了其它方法
```

```
public byte[] peek() throws Exception
public byte[] poll(long timeout, TimeUnit unit) throws Exception
public byte[] poll() throws Exception
public byte[] remove() throws Exception
public 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 在队列为空时直接返回 null。

Author: luoronghua
Email: luoronghua17s@ict.ac.cn
Date: 2018/10/9
参考链接: https://www.cnblogs.com/LiZhiW/tag/分布式/