# [构建ZooKeeper应用](https://www.cnblogs.com/sunddenly/p/4064992.html)+分布式锁

## 配置服务

配置服务是分布式应用所需要的基本服务之一，它使集群中的机器可以共享配置信息中那些公共的部分。简单地说，ZooKeeper可以作为一个具有高可用性的配置存储器，允许分布式应用的参与者检索和更新配置文件。使用ZooKeeper中的观察机制，可以建立一个活跃的配置服务，使那些感兴趣的客户端能够获得配置信息修改的通知。

下面来编写一个这样的服务。我们通过两个假设来简化所需实现的服务（稍加修改就可以取消这两个假设）。

第一，我们唯一需要存储的配置数据是字符串，关键字是znode的路径，因此我们在每个znode上存储了一个键／值对。  
 第二，在任何时候只有一个客户端会执行更新操作。

除此之外，这个模型看起来就像是有一个主人(类似于HDFS中的namenode)在更新信息，而他的工人则需要遵循这些信息。

### 服务类

import org.apache.zookeeper.CreateMode;  
import org.apache.zookeeper.KeeperException;  
import org.apache.zookeeper.Watcher;  
import org.apache.zookeeper.ZooDefs;  
import org.apache.zookeeper.data.Stat;  
import zk.ConnectionWatcher;  
  
import java.nio.charset.Charset;  
  
public class ActiveKeyValueStore extends ConnectionWatcher {  
 private static final Charset *CHARSET*= Charset.*forName*("UTF-8");  
 public void write(String path,String value) throws KeeperException, InterruptedException, KeeperException {  
 Stat stat = zk.exists(path, false);  
 if(stat==null){  
 zk.create(path, value.getBytes(*CHARSET*), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  
 }else{  
 zk.setData(path, value.getBytes(*CHARSET*),-1);  
 }  
 }  
 public String read(String path,Watcher watch) throws KeeperException, InterruptedException{  
 byte[] data = zk.getData(path, watch, null);  
 return new String(data,*CHARSET*);  
  
 }

write()方法的任务是将一个关键字及其值写到ZooKeeper。它隐藏了创建一个新的znode和用一个新值更新现有znode之间的区别，而是使用exists操作来检测znode是否存在，然后再执行相应的操作。其他值得一提的细节是需要将字符串值转换为字节数组，因为我们只用了UTF-8编码的getBytes()方法。

read()方法的任务是读取一个节点的配置属性。ZooKeeper的getData()方法有三个参数：

（1）路径  
（2）一个观察对象  
（3）一个Stat对象

Stat对象由getData()方法返回的值填充，用来将信息回传给调用者。通过这个方法，调用者可以获得一个znode的数据和元数据，但在这个例子中，由于我们对元数据不感兴趣，因此将Stat参数设为null。

为了说明ActiveKeyValueStore的用法，我们编写了一个用来更新配置属性值的类ConfigUpdater，如代码1.1所示。

### 用于随机更新ZooKeeper中的属性

import org.apache.zookeeper.KeeperException;  
  
import java.io.IOException;  
import java.util.Random;  
import java.util.concurrent.TimeUnit;  
  
public class ConfigUpdater {  
 public static final String *PATH*="/config";  
  
 private ActiveKeyValueStore store;  
 private Random random=new Random();  
  
 public ConfigUpdater(String hosts) throws Exception {  
 store = new ActiveKeyValueStore();  
 store.connect(hosts);  
 }  
 public void run() throws InterruptedException, KeeperException {  
 while(true){  
 String value=random.nextInt(100)+"";  
 store.write(*PATH*, value);  
 System.*out*.printf("Set %s to %s\n",*PATH*,value);  
 TimeUnit.*SECONDS*.sleep(random.nextInt(100));  
  
 }  
 }  
 public static void main(String[] args) throws Exception {  
 ConfigUpdater configUpdater = new ConfigUpdater("localhost:2181");  
 configUpdater.run();  
 }  
}

这个程序很简单，ConfigUpdater中定义了一个ActiveKeyValueStore，它在ConfigUpdater的构造函数中连接到ZooKeeper。run()方法永远在循环，在随机时间以随机值更新/config znode。

　　作为配置服务的用户，ConfigWatcher创建了一个ActiveKeyValueStore对象store，并且在启动之后通过displayConfig()调用了store的read()方法，显示它所读到的配置信息的初始值，并将自身作为观察传递给store。当节点状态发生变化时，再次通过displayConfig()显示配置信息，并再次将自身作为观察传递给store

### 该应用观察ZooKeeper中属性的更新情况，并将其打印到控制台

import org.apache.zookeeper.KeeperException;  
import org.apache.zookeeper.WatchedEvent;  
import org.apache.zookeeper.Watcher;  
  
import java.io.IOException;  
  
public class ConfigWatcher implements Watcher {  
 private ActiveKeyValueStore store;  
 @Override  
 public void process(WatchedEvent event) {  
 if(event.getType()== Event.EventType.*NodeDataChanged*){  
 try{  
 dispalyConfig();  
 }catch(InterruptedException e){  
 System.*err*.println("Interrupted. exiting. ");  
 Thread.*currentThread*().interrupt();  
 }catch(KeeperException e){  
 System.*out*.printf("KeeperException锛?s. Exiting.\n", e);  
 }  
  
 }  
 }  
  
 public ConfigWatcher(String hosts) throws Exception {  
 store=new ActiveKeyValueStore();  
 store.connect(hosts);  
 }  
 public void dispalyConfig() throws KeeperException, InterruptedException{  
 String value=store.read(ConfigUpdater.*PATH*, this);  
 System.*out*.printf("Read %s as %s\n",ConfigUpdater.*PATH*,value);  
 }  
  
 public static void main(String[] args) throws Exception {  
 ConfigWatcher configWatcher = new ConfigWatcher("localhost:2181");  
 configWatcher.dispalyConfig();  
 //stay alive until process is killed or Thread is interrupted  
 Thread.*sleep*(Long.*MAX\_VALUE*);  
 }  
}

当ConfigUpdater更新znode时，ZooKeeper产生一个类型为EventType.NodeDataChanged的事件，从而触发观察。ConfigWatcher在它的process()方法中对这个事件做出反应，读取并显示配置的最新版本。由于观察仅发送单次信号，因此每次我们调用ActiveKeyValueStore的read()方法时，都将一个新的观察告知ZooKeeper来确保我们可以看到将来的更新。但是，我们还是不能保证接收到每一个更新，因为在收到观察事件通知与下一次读之间，znode可能已经被更新过，而且可能是很多次，由于客户端在这段时间没有注册任何观察，因此不会收到通知。对于示例中的配置服务，这不是问题，因为客户端只关心属性的最新值，最新值优先于之前的值。但是，一般情况下，这个潜在的问题是不容忽视的。

## ZooKeeper异常

在Java API中的每一个ZooKeeper操作都在其throws子句中声明了两种类型的异常，分别是InterruptedException和KeeperException。

### **InterruptedException异常**

如果操作被中断，则会有一个InterruptedException异常被抛出。在Java语言中有一个取消阻塞方法的标准机制，即针对存在阻塞方法的线程调用interrupt()，使线程不再继续等待，转而去做其他事情。一个成功的取消操作将产生一个InterruptedException异常。

ZooKeeper也遵循这一机制，因此你可以使用这种方法来取消一个ZooKeeper操作。使用了ZooKeeper的类或库通常会throws InterruptedException异常，使客户端能够取消它们的操作。InterruptedException异常并不意味着有故障，而是**表明**相应的操作已经被取消，所以在配置服务的示例中，可以通过throws异常来中止应用程序的运行。

### **KeeperException异常**

**(1)**.如果ZooKeeper服务器发出一个错误信号或与服务器存在通信问题，抛出的则是KeeperException异常。

**①**针对不同的错误情况，KeeperException异常存在不同的子类。

**例如**:　KeeperException.NoNodeException是KeeperException的一个子类，如果你试图针对一个不存在的znode执行操作，抛出的则是该异常。

**②**每一个KeeperException异常的子类都对应一个关于错误类型信息的代码。

**例如**:　KeeperException.NoNodeException异常的代码是KeeperException.Code.NONODE

**(2)**有两种方法被用来**处理**KeeperException异常

**①捕捉**KeeperException异常，并且通过**检测**它的代码来决定采取何种补救措施；

**②**另一种是**捕捉等价的**KeeperException子类，并且在每段捕捉代码中**执行**相应的操作。

**（3）**KeeperException异常分为三大类

**① 状态异常**

当一个操作因不能被应用于znode树而导致失败时，就会出现状态异常。状态异常产生的原因通常是在同一时间有另外一个进程正在修改znode。例如，如果一个znode先被另外一个进程更新了，根据版本号执行setData操作的进程就会失败，并收到一个KeeperException.BadVersionException异常，这是因为版本号不匹配。程序员通常都知道这种冲突总是存在的，也都会编写代码来进行处理。

一些状态异常会指出程序中的错误，例如KeeperException.NoChildrenForEphemeralsException异常，试图在短暂znode下创建子节点时就会抛出该异常。

**② 可恢复异常**

可恢复的异常是指那些应用程序能够在同一个ZooKeeper会话中恢复的异常。一个可恢复的异常是通过KeeperException.ConnectionLossException来表示的，它**意味着**已经丢失了与ZooKeeper的连接。ZooKeeper会尝试重新连接，并且在大多数情况下重新连接会成功，并确保会话是完整的。

但是ZooKeeper不能判断与KeeperException.ConnectionLossException异常相关的操作是否成功执行。这种情况就是部分失败的一个例子。这时程序员有责任来解决这种不确定性，并且根据应用的情况来采取适当的操作。在这一点上，就需要对“幂等”(idempotent)操作和“非幂等”(Nonidempotent)操作进行区分。幂等操作是指那些一次或多次执行都会产生相同结果的操作，**例如**读请求或无条件执行的setData操作。对于幂等操作，只需要简单地进行重试即可。对于非幂等操作，就不能盲目地进行重试，因为它们多次执行的结果与一次执行是完全不同的。程序可以通过**在znode的路径**和**它的数据中编码信息**来检测是否非幂等操怍的更新已经完成。

**③不可恢复的异常**

在某些情况下，ZooKeeper会话会失效——也许因为超时或因为会话被关闭，两种情况下都会收到KeeperException.SessionExpiredException异常，或因为身份验证失败，KeeperException.AuthFailedException异常。无论上述哪种情况，所有与会话相关联的短暂znode都将丢失，因此应用程序需要在重新连接到ZooKeeper之前重建它的状态.

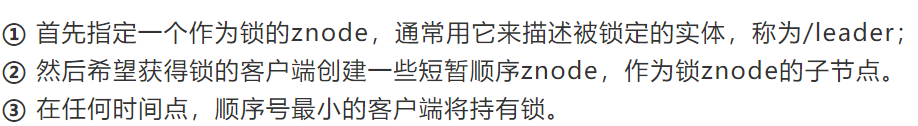
## 锁服务

### 分布式锁概述

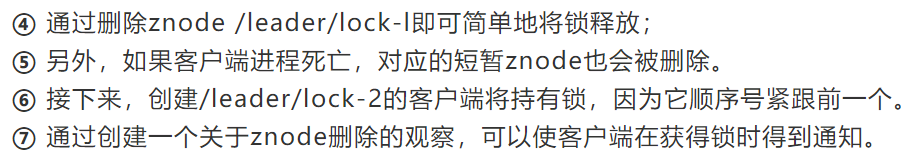
分布式锁在一组进程之间提供了一种互斥机制。在任何时刻，在任何时刻只有一个进程可以持有锁。分布式锁可以在大型分布式系统中实现领导者选举，在任何时间点，持有锁的那个进程就是系统的领导者。

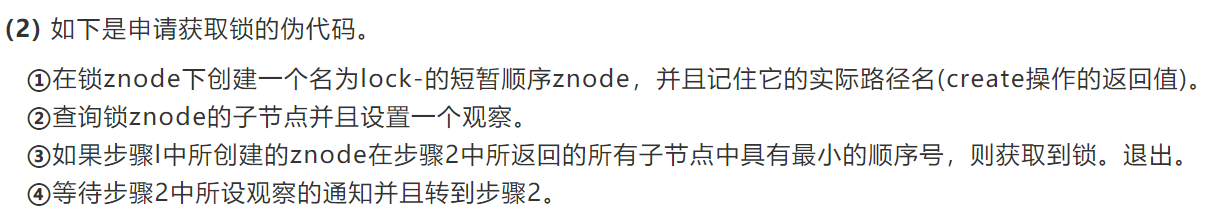
注意：不要将ZooKeeper自己的领导者选举和使用了ZooKeeper基本操作实现的一般领导者选混为一谈。ZooKeeper自己的领导者选举机制是对外不公开的，我们这里所描述的一般领导者选举服务则不同，他是对那些需要与主进程保持一致的分布式系统所设计的。

为了使用ZooKeeper来实现分布式锁服务，我们使用顺序znode来为那些竞争锁的进程强制排序。



**例如**，有两个客户端差不多同时创建znode，分别为/leader/lock-1和/leader/lock-2，那么创建/leader/lock-1的客户端将会持有锁，因为它的znode顺序号最小。ZooKeeper服务是顺序的仲裁者，因为它负责分配顺序号。





### 当前问题与方案

#### 羊群效应

1. 问题

虽然这个算法是正确的，但还是存在一些问题。第一个问题是这种实现会受到“羊群效应”(herd effect)的影响。考虑有成百上千客户端的情况，所有的客户端都在尝试获得锁，每个客户端都会在锁znode上设置一个观察，用于捕捉子节点的变化。每次锁被释放或另外一个进程开始申请获取锁的时候，观察都会被触发并且每个客户端都会收到一个通知。“羊群效应“就是指大量客户端收到同一事件的通知，但实际上只有很少一部分需要处理这一事件。在这种情况下，只有一个客户端会成功地获取锁，但是维护过程及向所有客户端发送观察事件会产生峰值流量，这会对ZooKeeper服务器造成压力。

1. 解决方案

为了避免出现羊群效应，我们需要优化通知的条件。关键在于只有在前一个顺序号的子节点消失时才需要通知下一个客户端，而不是删除（或创建）任何子节点时都需要通知。在我们的例子中，如果客户端创建了znode /leader/lock-1、/leader/lock-2和／leader/lock-3，那么只有当/leader/lock-2消失时才需要通知／leader/lock-3对照的客户端；/leader/lock-1消失或有新的znode /leader/lock-4加入时，不需要通知该客户端。

#### 可恢复的异常

1. 问题

这个申请锁的算法目前还存在另一个问题，就是不能处理因连接丢失而导致的create操作失败。如前所述，在这种情况下，我们不知道操作是成功还是失败。由于创建一个顺序znode是非幂等操作，所以我们不能简单地重试，因为如果第一次创建已经成功，重试会使我们多出一个永远删不掉的孤儿zriode(至少到客户端会话结束前）。不幸的结果是将会出现死锁。

1. 解决方案

问题在于，在重新连接之后客户端不能够判断它是否已经创建过子节点。解决方案是在znode的名称中嵌入一个ID，如果客户端出现连接丢失的情况，重新连接之后它便可以对锁节点的所有子节点进行检查，看看是否有子节点的名称中包含其ID。如果有一个子节点的名称包含其ID，它便知道创建操作已经成功，不需要再创建子节点。如果没有子节点的名称中包含其ID，则客户端可以安全地创建一个新的顺序子节点。  
 客户端会话的ID是一个长整数，并且在ZooKeeper服务中是唯一的，因此非常适合在连接丢失后用于识别客户端。可以通过调用Java ZooKeeper类的getSessionld()方法来获得会话的ID。

在创建短暂顺序znode时应当采用lock-<sessionld>-这样的命名方式，ZooKeeper在其尾部添加顺序号之后，znode的名称会形如lock-<sessionld>-<sequenceNumber>。由于顺序号对于父节点来说是唯一的，但对于子节点名并不唯一，因此采用这样的命名方式可以让子节点在保持创建顺序的同时能够确定自己的创建者。

## ZooKeeper实现共享锁

### 场景描述

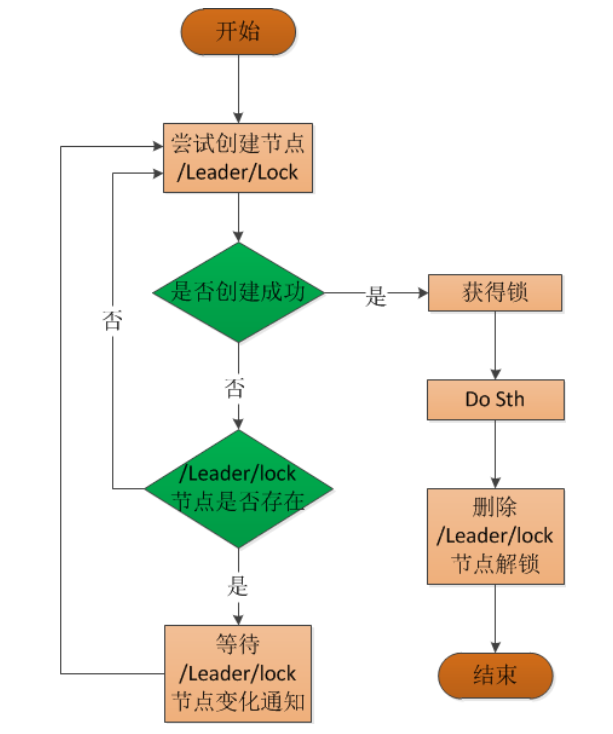
大家也许都很熟悉了多个线程或者多个进程间的共享锁的实现方式了，但是在分布式场景中我们会面临多个Server之间的锁的问题。

假设有这样一个场景：两台server ：serverA，serverB需要在C机器上的/usr/local/a.txt文件上进行写操作，如果两台机器同时写该文件，那么该文件的最终结果可能会产生乱序等问题。最先能想到的是serverA在写文件前告诉ServerB “我要开始写文件了，你先别写”，等待收到ServerB的确认回复后ServerA开始写文件，写完文件后再通知ServerB“我已经写完了”。假设在我们场景中有100台机器呢，中间任意一台机器通信中断了又该如何处理？容错和性能问题呢？要能健壮，稳定，高可用并保持高性能，系统实现的复杂度比较高，从头开发这样的系统代价也很大。幸运的是，我们有了基于googlechubby原理开发的开源的ZooKeeper系统。接下来本文将介绍两种ZooKeeper实现分布式共享锁的方法。

### 利用节点名称的唯一性来实现共享锁

ZooKeeper表面上的节点结构是一个和unix文件系统类似的小型的树状的目录结构，ZooKeeper机制规定：同一个目录下只能有一个唯一的文件名。

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



### 利用顺序节点实现共享锁

首先介绍一下，Zookeeper中有一种节点叫做顺序节点，故名思议，假如我们在/lock/目录下创建节3个点，ZooKeeper集群会按照提起创建的顺序来创建节点，节点分别为/lock/0000000001、/lock/0000000002、/lock/0000000003。

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

**算法思路**：对于加锁操作，可以让所有客户端都去/lock目录下创建临时、顺序节点，如果创建的客户端发现自身创建节点序列号是/lock/目录下最小的节点，则获得锁。否则，监视比自己创建节点的序列号小的节点（当前序列在自己前面一个的节点），进入等待。解锁操作，只需要将自身创建的节点删除即可。具体算法流程如下图所示:

