# Zookeeper基础

## ****分布式锁的实现****

为了防止分布式系统中的多个进程之间相互干扰，我们需要一种分布式协调技术来对这些进程进行调度。而这个分布式协调技术的核心就是来实现这个分**布式锁**。那么这个锁怎么实现呢？这实现起来确实相对来说比较困难的。

### ****面临的问题****

有人可能会感觉这不是很难。无非是将原来在同一台机器上对进程调度的原语，通过网络实现在分布式环境中。是的，表面上是可以这么说。但是问题就在网络这，在分布式系统中，所有在同一台机器上的假设都不存在：因为网络是不可靠的。

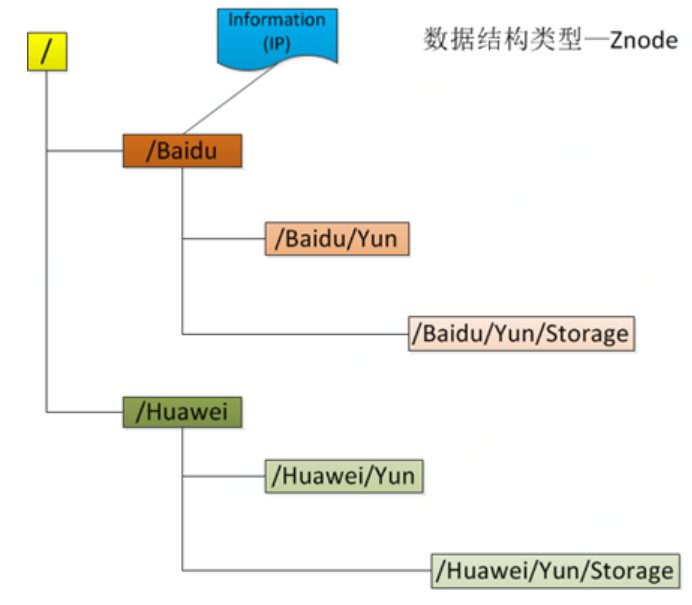
比如，在同一台机器上，你对一个服务的调用如果成功，那就是成功，如果调用失败，比如抛出异常那就是调用失败。但是在分布式环境中，由于网络的不可靠，你对一个服务的调用失败了并不表示一定是失败的，可能是执行成功了，但是响应返回的时候失败了。还有，A和B都去调用C服务，在时间上 A还先调用一些，B后调用，那么最后的结果是不是一定A的请求就先于B到达呢？ 这些在同一台机器上的种种假设，我们都要重新思考，我们还要思考这些问题给我们的设计和编码带来了哪些影响。还有，在分布式环境中为了提升可靠性，我们往往会部署多套服务，但是如何在多套服务中达到一致性，这在同一台机器上多个进程之间的同步相对来说比较容易办到，但在分布式环境中确实一个大难题。

所以分布式协调远比在同一台机器上对多个进程的调度要难得多，而且如果为每一个分布式应用都开发一个独立的协调程序。一方面，协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器。另一方面，协调程序开销比较大，会影响系统原有的性能。所以，急需一种高可靠、高可用的通用协调机制来用以协调分布式应用。

**Zookeeper就是分布式锁的实现者。**

**ZooKeeper所提供的服务主要是通过：数据结构+原语+watcher机制，三个部分来实现的**

### ****ZooKeeper数据模型Znode****



**(1) 引用方式**

Zonde通过**路径引用**，如同Unix中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来**开头**。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。

**(2)** **Znode结构**

ZooKeeper命名空间中的Znode，兼具文件和目录两种特点。既像文件一样维护着数据、元信息、ACL(访问控制列表)、时间戳等数据结构，又像目录一样可以作为路径标识的一部分。图中的每个节点称为一个Znode。 每个Znode由3部分组成:

**①** stat：此为状态信息, 描述该Znode的版本, 权限等信息

**②** data：与该Znode关联的数据

**③** children：该Znode下的子节点

**(3) 数据访问**

ZooKeeper中的每个节点存储的数据要被**原子性的操作**。也就是说读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。

**(4) 节点类型**

ZooKeeper中的节点有两种，分别为**临时节点**和**永久节点**。节点的类型在创建时即被确定，并且不能改变。

**① 临时节点：**该节点的生命周期依赖于创建它们的会话。一旦会话(Session)结束，临时节点将被自动删除，当然可以也可以手动删除。虽然每个临时的Znode都会绑定到一个客户端会话，但他们对所有的客户端还是可见的。另外，ZooKeeper的临时节点不允许拥有子节点。

**② 永久节点：**该节点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才能被删除。

**(5)** **顺序节点//暂时不知道啥意思**

当创建Znode的时候，用户可以请求在ZooKeeper的路径结尾添加一个**递增的计数**。这个计数**对于此节点的父节点来说**是唯一的，它的格式为"%10d"(10位数字，没有数值的数位用0补充，例如"0000000001")。当计数值大于232-1时，计数器将溢出。

**(6) 监视**

客户端可以在节点上设置watch，我们称之为**监视器**。当节点状态发生改变时(Znode的增、删、改)将会触发watch所对应的操作。当watch被触发时，ZooKeeper将会向客户端发送且仅发送一条通知，因为watch只能被触发一次，这样可以减少网络流量。

### ****ZooKeeper中的时间****

ZooKeeper有多种记录时间的形式，其中包含以下几个主要属性：

**(1) Zxid**

致使ZooKeeper节点状态改变的每一个操作都将使节点接收到一个Zxid格式的时间戳，并且这个时间戳全局有序。也就是说，也就是说，每个对节点的改变都将产生一个唯一的Zxid。如果Zxid1的值小于Zxid2的值，那么Zxid1所对应的事件发生在Zxid2所对应的事件之前。实际上，ZooKeeper的每个节点维护者三个Zxid值，为别为：cZxid、mZxid、pZxid。

**①** **cZxid**： 是节点的创建时间所对应的Zxid格式时间戳。  
**② mZxid**：是节点的修改时间所对应的Zxid格式时间戳。

实现中Zxid是一个64为的数字，它高32位是epoch用来标识leader（主从关系中的主节点）关系是否改变，每次一个leader被选出来，它都会有一个 新的epoch。低32位是个**递增计数**。

**(2) 版本号**

对节点的每一个操作都将致使这个节点的版本号增加。每个节点维护着三个版本号，他们分别为：

**① version**：节点数据版本号  
**② cversion**：子节点版本号  
**③ aversion**：节点所拥有的ACL版本号

### ****ZooKeeper节点属性****



### ****ZooKeeper服务中操作****



更新ZooKeeper操作是有限制的。delete或setData必须明确要更新的Znode的版本号，我们可以调用exists找到。如果版本号不匹配，更新将会失败。

更新ZooKeeper操作是非阻塞式的。因此客户端如果失去了一个更新(由于另一个进程在同时更新这个Znode)，他可以在不阻塞其他进程执行的情况下，选择重新尝试或进行其他操作。

尽管ZooKeeper可以被看做是一个文件系统，但是处于便利，摒弃了一些文件系统地操作原语。因为文件非常的小并且使整体读写的，所以不需要打开、关闭或是寻址的操作

### ****Watch触发器****

**(1) watch概述**

ZooKeeper可以为所有的**读操作**设置watch，这些读操作包括：exists()、getChildren()及getData()。watch事件是**一次性的触发器**，当watch的对象状态发生改变时，将会触发此对象上watch所对应的事件。watch事件将被**异步**地发送给客户端，并且ZooKeeper为watch机制提供了有序的**一致性保证**。理论上，客户端接收watch事件的时间要快于其看到watch对象状态变化的时间。

**2) watch类型**

ZooKeeper所管理的watch可以分为两类：

**①** 数据watch(data  watches)：**getData**和**exists**负责设置数据watch  
**②**孩子watch(child watches)：**getChildren**负责设置孩子watch

我们可以通过操作**返回的数据**来设置不同的watch：

**① getData和exists：**返回关于节点的数据信息  
**② getChildren：**返回孩子列表

因此

**①**一个成功的**setData操作**将触发Znode的数据watch

**②** 一个成功的**create操作**将触发Znode的数据watch以及孩子watch

**③**一个成功的**delete操作**将触发Znode的数据watch以及孩子watch

**(3) watch注册与处触发**

**①**exists操作上的watch，在被监视的Znode**创建**、**删除**或**数据更新**时被触发。  
**②** getData操作上的watch，在被监视的Znode**删除**或**数据更新**时被触发。在被创建时不能被触发，因为只有Znode一定存在，getData操作才会成功。  
**③** getChildren操作上的watch，在被监视的Znode的子节点**创建**或**删除**，或是这个Znode自身被**删除**时被触发。可以通过查看watch事件类型来区分是Znode，还是他的子节点被删除：NodeDelete表示Znode被删除，NodeDeletedChanged表示子节点被删除。

Watch由客户端所连接的ZooKeeper服务器在本地维护，因此watch可以非常容易地设置、管理和分派。当客户端连接到一个新的服务器时，任何的会话事件都将可能触发watch。另外，当从服务器断开连接的时候，watch将不会被接收。但是，当一个客户端重新建立连接的时候，任何先前注册过的watch都会被重新注册。

**Zookeeper的watch实际上要处理两类事件：**

**① 连接状态事件**(type=None, path=null)

这类事件不需要注册，也不需要我们连续触发，我们只要处理就行了。

**② 节点事件**

节点的建立，删除，数据的修改。它是one time trigger，我们需要不停的注册触发，还可能发生事件丢失的情况。

上面2类事件都在Watch中处理，也就是重载的**process(Event event)**

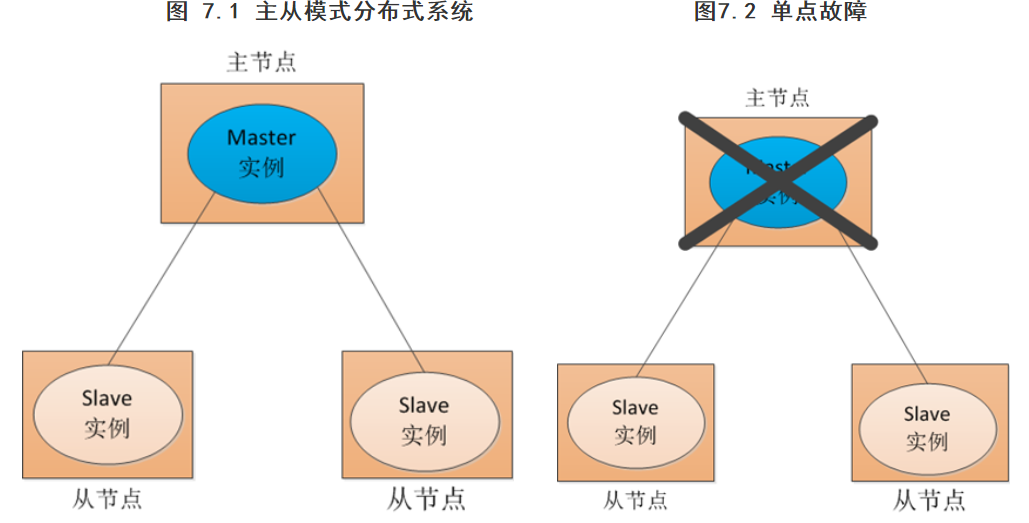
**节点事件的触发，通过函数exists，getData或getChildren来处理这类函数，有双重作用：**

1. **注册触发事件**

**② 函数本身的功能**

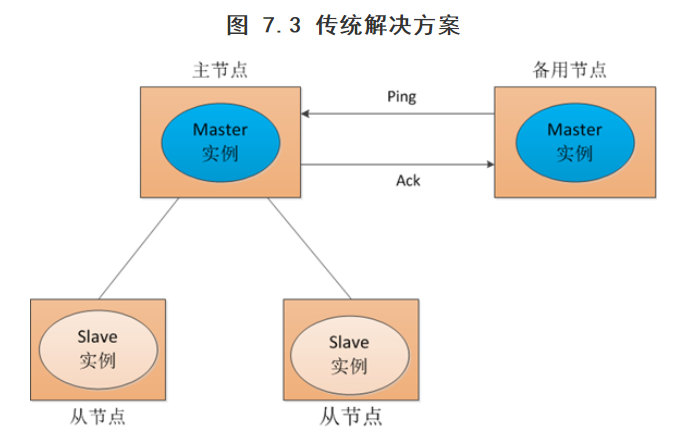
## ****分布式锁应用场景****

在分布式锁服务中，有一种最典型应用场景，就是通过对集群进行**Master选举**，来解决分布式系统中的**单点故障**。什么是分布式系统中的单点故障：通常分布式系统采用主从模式，就是一个主控机连接多个处理节点。主节点负责分发任务，从节点负责处理任务，当我们的主节点发生故障时，那么整个系统就都瘫痪了，那么我们把这种故障叫作单点故障。如下图所示：

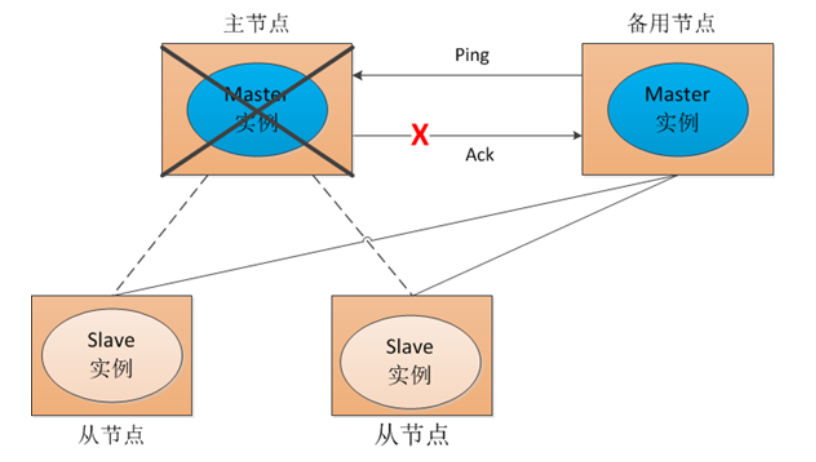


### 传统解决方案：

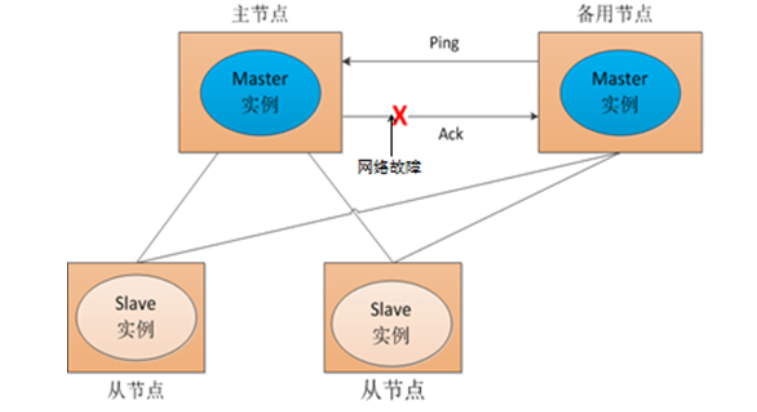
传统方式是采用一个备用节点，这个备用节点定期给当前主节点发送ping包，主节点收到ping包以后向备用节点发送回复Ack，当备用节点收到回复的时候就会认为当前主节点还活着，让他继续提供服务。如图7.3所示：



当主节点挂了，这时候备用节点收不到回复了，然后他就认为主节点挂了接替他成为主节点如下图7.4所示：



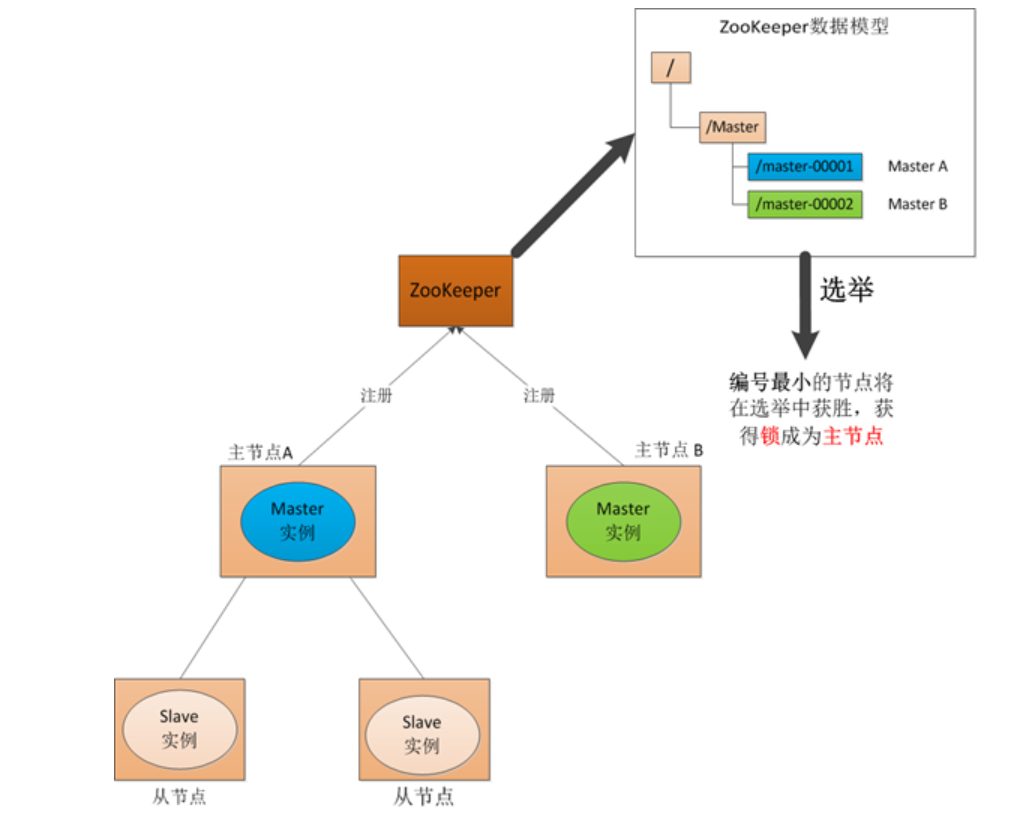
但是这种方式就是有一个隐患，就是网络问题，来看一网络问题会造成什么后果，如下图7.5所示：



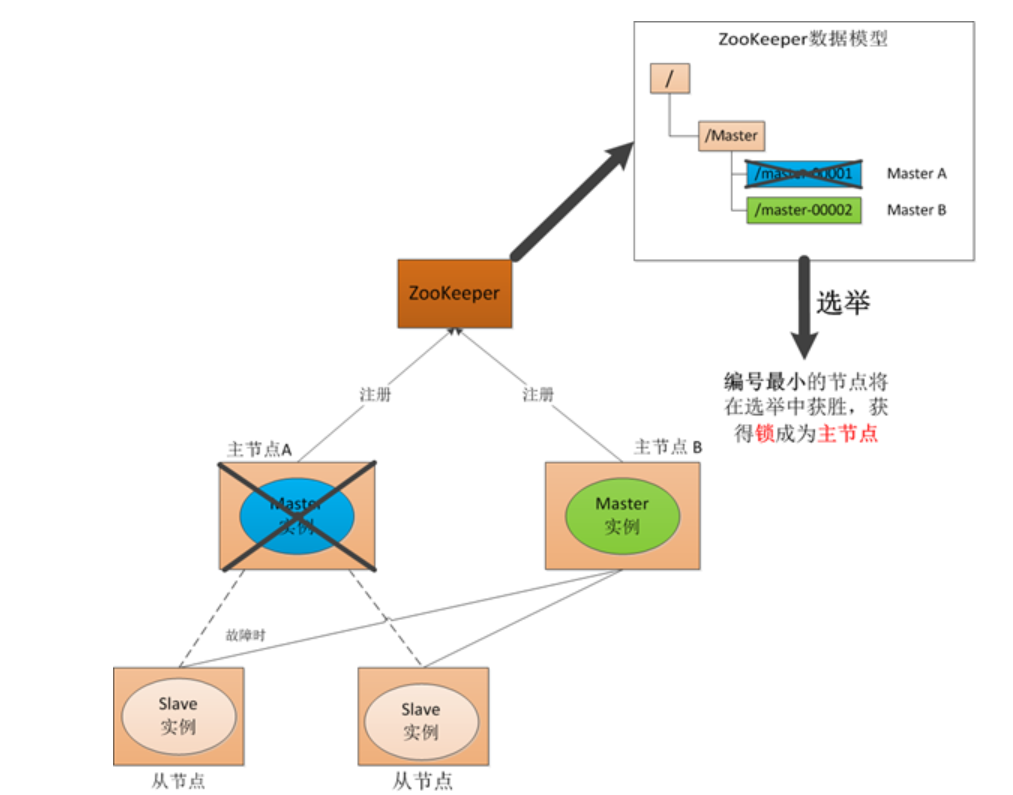
也就是说我们的主节点的并没有挂，只是在回复的时候网络发生故障，这样我们的备用节点同样收不到回复，就会认为主节点挂了，然后备用节点将他的Master实例启动起来，这样我们的分布式系统当中就有了两个主节点也就是---**双Master**，出现Master以后我们的从节点就会将它所做的事一部分汇报给了主节点，一部分汇报给了从节点，这样服务就全乱了。为了防止出现这种情况，我们引入了ZooKeeper，它虽然不能避免网络故障，但它能够保证每时每刻只有一个Master。

### ****ZooKeeper解决方案****

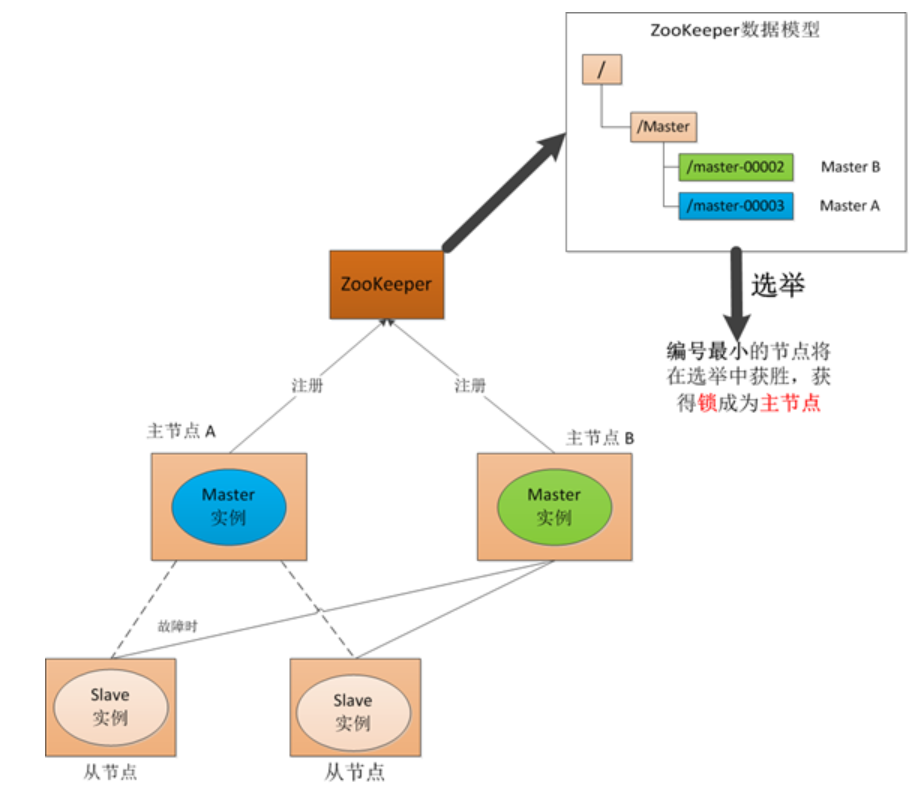
在引入了Zookeeper以后我们启动了两个主节点，"主节点-A"和"主节点-B"他们启动以后，都向ZooKeeper去注册一个节点。我们假设"主节点-A"锁注册地节点是"master-00001"，"主节点-B"注册的节点是"master-00002"，注册完以后进行选举，编号最小的节点将在选举中获胜获得锁成为主节点，也就是我们的"主节点-A"将会获得锁成为主节点，然后"主节点-B"将被阻塞成为一个备用节点。那么，通过这种方式就完成了对两个Master进程的调度。



如果"主节点-A"挂了，这时候他所注册的节点将被自动删除，ZooKeeper会自动感知节点的变化，然后再次发出选举，这时候"主节点-B"将在选举中获胜，替代"主节点-A"成为主节点。



如果主节点恢复了，他会再次向ZooKeeper注册一个节点，这时候他注册的节点将会是"master-00003"，ZooKeeper会感知节点的变化再次发动选举，这时候"主节点-B"在选举中会再次获胜继续担任"主节点"，"主节点-A"会担任备用节点。

。