# Zookeeper学习之路（一）初识

讨论QQ：1586558083

## 目录


- 引言
- 分布式协调技术
- 分布式锁的实现
  - 面临的问题
  - 分布式锁的实现者
- ZooKeeper概述
- ZooKeeper数据模型
  - ZooKeeper数据模型Znode
  - ZooKeeper中的时间
  - ZooKeeper节点属性
- ZooKeeper服务中操作
- 监听机制
  - watch触发器
  - 监听工作原理
- ZooKeeper应用举例
  - 分布式锁应用场景
  - 传统解决方案
  - ZooKeeper解决方案


## 正文

本文引用自 http://www.cnblogs.com/sunddenly/p/4033574.html

回到顶部

## 引言

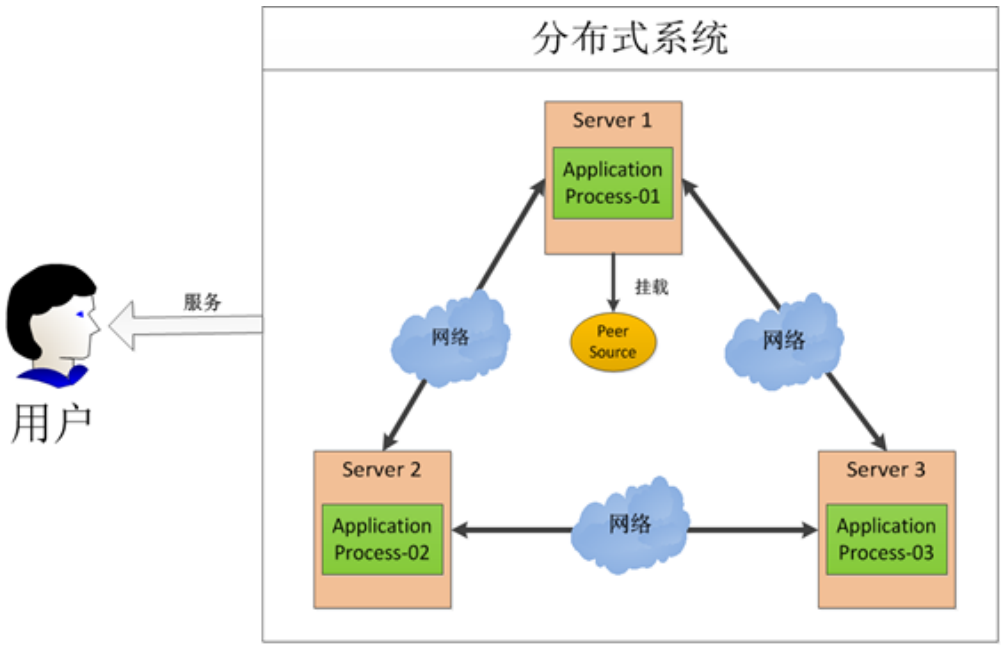
Hadoop 集群当中 N 多的配置信息如何做到全局一致并且单点修改迅速响应到整个集群？ --- 配置管理

Hadoop 集群中的 namonode 和 resourcemanager 的单点故障怎么解决？ --- 集群的主节点的单点故障

回到顶部

## 分布式协调技术

在给大家介绍ZooKeeper之前先来给大家介绍一种技术——分布式协调技术。那么什么是分布式协调技术？那么我来告诉大家，其实分布式协调技术 主要用来解决分布式环境当中多个进程之间的同步控制，让他们有序的去访问某种临界资源，防止造成"脏数据"的后果。这时，有人可能会说这个简单，写一个调 度算法就轻松解决了。说这句话的人，可能对分布式系统不是很了解，所以才会出现这种误解。如果这些进程全部是跑在一台机上的话，相对来说确实就好办了，问 题就在于他是在一个分布式的环境下，这时问题又来了，那什么是分布式呢？这个一两句话我也说不清楚，但我给大家画了一张图希望能帮助大家理解这方面的内 容，如果觉得不对尽可拍砖，来咱们看一下这张图，如图1.1所示。




给大家分析一下这张图，在这图中有三台机器，每台机器各跑一个应用程序。然后我们将这三台机器通过网络将其连接起来，构成一个系统来为用户提供服务，对用户来说这个系统的架构是透明的，他感觉不到我这个系统是一个什么样的架构。那么我们就可以把这种系统称作一个**分布式系统**。

那我们接下来再分析一下，在这个分布式系统中如何对进程进行调度，我假设在第一台机器上挂载了一个资源，然后这三个物理分布的进程都要竞争这个资源，但我们又不希望他们同时进行访问，这时候我们就需要一个**协调器**，来让他们有序的来访问这个资源。这个协调器就是我们经常提到的那个**锁**，比如说"进程-1"在使用该资源的时候，会先去获得锁，"进程1"获得锁以后会对该资源保持**独占**，这样其他进程就无法访问该资源，"进程1"用完该资源以后就将锁释放掉，让其他进程来获得锁，那么通过这个锁机制，我们就能保证了分布式系统中多个进程能够有序的访问该临界资源。那么我们把这个分布式环境下的这个锁叫作**分布式锁**。这个分布式锁也就是我们**分布式协调技术**实现的核心内容，那么如何实现这个分布式呢，那就是我们后面要讲的内容。

回到顶部

# 分布式锁的实现

## 面临的问题

在看了图1.1所示的分布式环境之后，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。

## 分布式锁的实现者

目前，在分布式协调技术方面做得比较好的就是Google的Chubby还有Apache的ZooKeeper他们都是分布式锁的实现者。有人会问既然有了Chubby为什么还要弄一个ZooKeeper，难道Chubby做得不够好吗？不是这样的，主要是

Chbby是非开源的，Google自家用。后来雅虎模仿Chubby开发出了ZooKeeper，也实现了类似的分布式锁的功能，并且将ZooKeeper作为一种开源的程序捐献给了Apache，那么这样就可以使用ZooKeeper所提供锁服务。而且在分布式领域久经考验，它的可靠性，可用性都是经过理论和实践的验证的。所以我们在构建一些分布式系统的时候，就可以以这类系统为起点来构建我们的系统，这将节省不少成本，而且bug也 将更少。





回到顶部

## ZooKeeper概述

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现。它提供了简单原始的功能，分布式应用可以基于它实现更高级的服务，比 如**分布式同步，配置管理，集群管理，命名管理，队列管理**。它被设计为易于编程，使用文 件系统目录树作为数据模型。服务端跑在 java 上，提供 java 和 C 的客户端 API 众所周知，协调服务非常容易出错，但是却很难恢复正常，例如，协调服务很容易处于 竞态以至于出现死锁。我们设计 ZooKeeper 的目的是为了减轻分布式应用程序所承担的协 调任务 ZooKeeper 是集群的管理者，监视着集群中各节点的状态，根据节点提交的反馈进行下 一步合理的操作。最终，将简单易用的接口和功能稳定，性能高效的系统提供给用户。

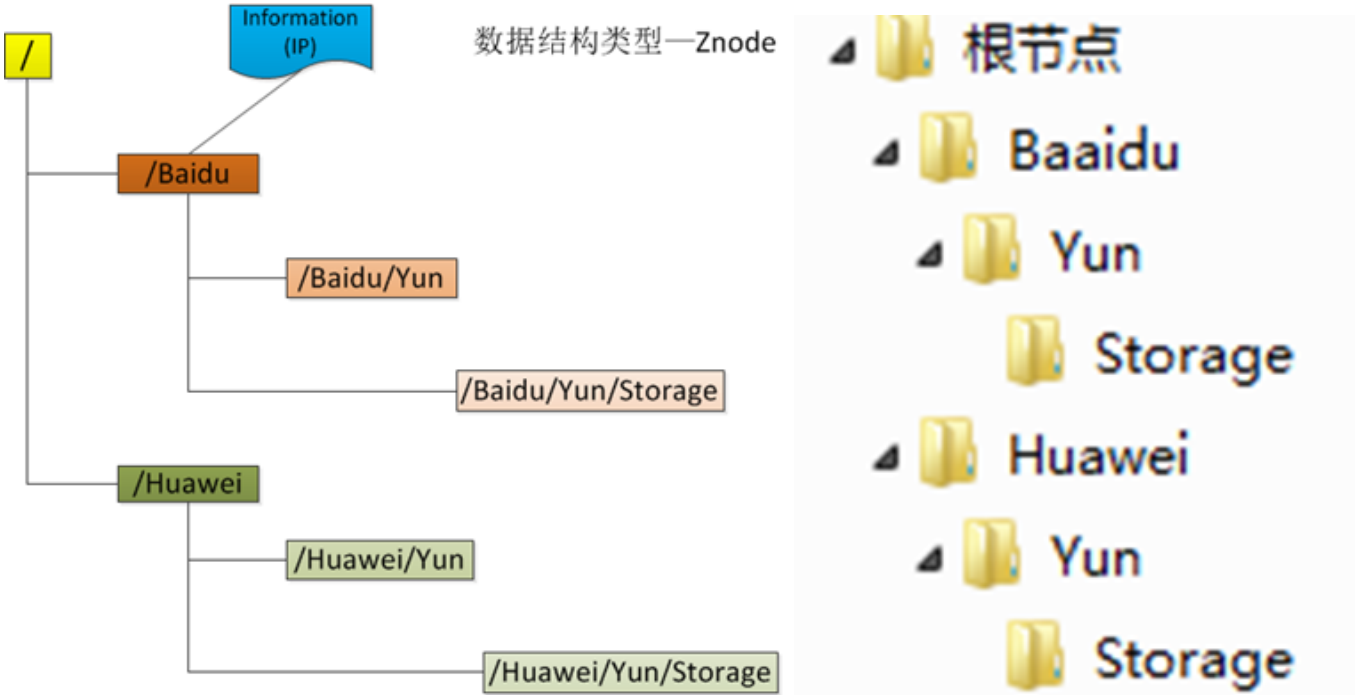
前面提到了那么多的服务，比如分布式锁、配置维护、组服务等，那它们是如何实现的呢，我相信这才是大家关心的东西。ZooKeeper在实现这些服务时，首先它设计一种新的**数据结构——Znode**，然后在该数据结构的基础上定义了一些**原语**，也就是一些关于该数据结构的一些操作。有了这些数据结构和原语还不够，因为我们的ZooKeeper是工作在一个分布式的环境下，我们的服务是通过消息以网络的形式发送给我们的分布式应用程序，所以还需要一个**通知机制**——Watcher机制。那么总结一下，ZooKeeper所提供的服务主要是通过：数据结构+原语+watcher机制，三个部分来实现的。那么我就从这三个方面，给大家介绍一下ZooKeeper。

回到顶部

## ZooKeeper数据模型

## ZooKeeper数据模型Znode

ZooKeeper拥有一个层次的命名空间，这个和标准的文件系统非常相似，如下图所示。




从图中我们可以看出ZooKeeper的数据模型，在结构上和标准文件系统的非常相似，都是采用这种树形层次结构，ZooKeeper树中的每个节点被称为—Znode。和文件系统的目录树一样，ZooKeeper树中的每个节点可以拥有子节点。但也有不同之处：

**(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。在ZooKeeper中，路径由Unicode字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。

**(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、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

ZooKeeper虽然可以关联一些数据，但并没有被设计为常规的数据库或者大数据存储。相反的是，它用来**管理调度数据**，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以KB为大小单位。ZooKeeper的服务器和客户端都被设计为严格检查并限制每个Znode的数据大小至多1M，但常规使用中应该远小于此值。

**(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**

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$2^{32}$-1时，计数器将溢出。

**(6) 观察**

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

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，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。

③ **pZxid**： 是与 该节点的子节点（或该节点）的最近一次 创建 / 删除 的时间戳对应

实现中Zxid是一个64为的数字，它高32位是epoch用来标识leader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

通过前面的介绍，我们可以了解到，一个节点自身拥有表示其状态的许多重要属性，如下图所示。

图 4.2 Znode节点属性结构

| 属性 | 描述 |
|---|---|
| czxid | 节点被创建的 zxid |
| mzxid | 节点被修改的 zxid |
| ctime | 节点被创建的时间 |
| mtime | 节点被修改的 zxid |
| version | 节点被修改的版本号 |
| cversion | 节点所拥有的子节点被修改的版本号 |
| aversion | 节点的 ACL 被修改的版本号 |
| ephemeralOwner | 如果此节点为临时节点，那么他的值为这个节点拥有者的会话 ID；否则，他的值为 0 |
| dataLength | 节点数长度 |
| numChildren | 节点用的子节点长度 |
| pzxid | 最新修改的 zxid，貌似与 mzxid 重合了 |

回到顶部

## ZooKeeper服务中操作

在ZooKeeper中有9个基本操作，如下图所示：

| 操作 | 描述 |
| --- | --- |
| create | 创建 Znode（父 Zonde 必须存在） |
| delete | 删除 Znode（Znode 没有子节点） |
| exists | 测试 Znode 是否存在，并获取他的元数据 |
| getACL/ setACL | 为 Znode 获取/设置 ACL |
| getChildren | 获取 Znode 所有子节点的列表 |
| getData/setData | 获取/设置 Znode 的相关数据 |
| sync | 使客户端的 Znode 视图与 ZooKeeper 同步 |

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、关闭或是寻地的操作。

回到顶部

# 监听机制

## 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**

下图 watch设置操作及相应的触发器如图下图所示：

| 设置 watch | watch 触发器 | | | | |
|---|---|---|---|---|---|
| | create | | delete | | setData |
| | Znode | child | Znode | child | Znode |
| exists | NodeCreated | | NodeDeleted | | NodeDataChanged |
| getdata | | | NodeDeleted | | NodeDataChanged |
| getChildren | | NodeChildrenChanged | NodeDeleted | NodeDeletedChanged | |

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。

**(4) 需要注意的几点**

**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：**

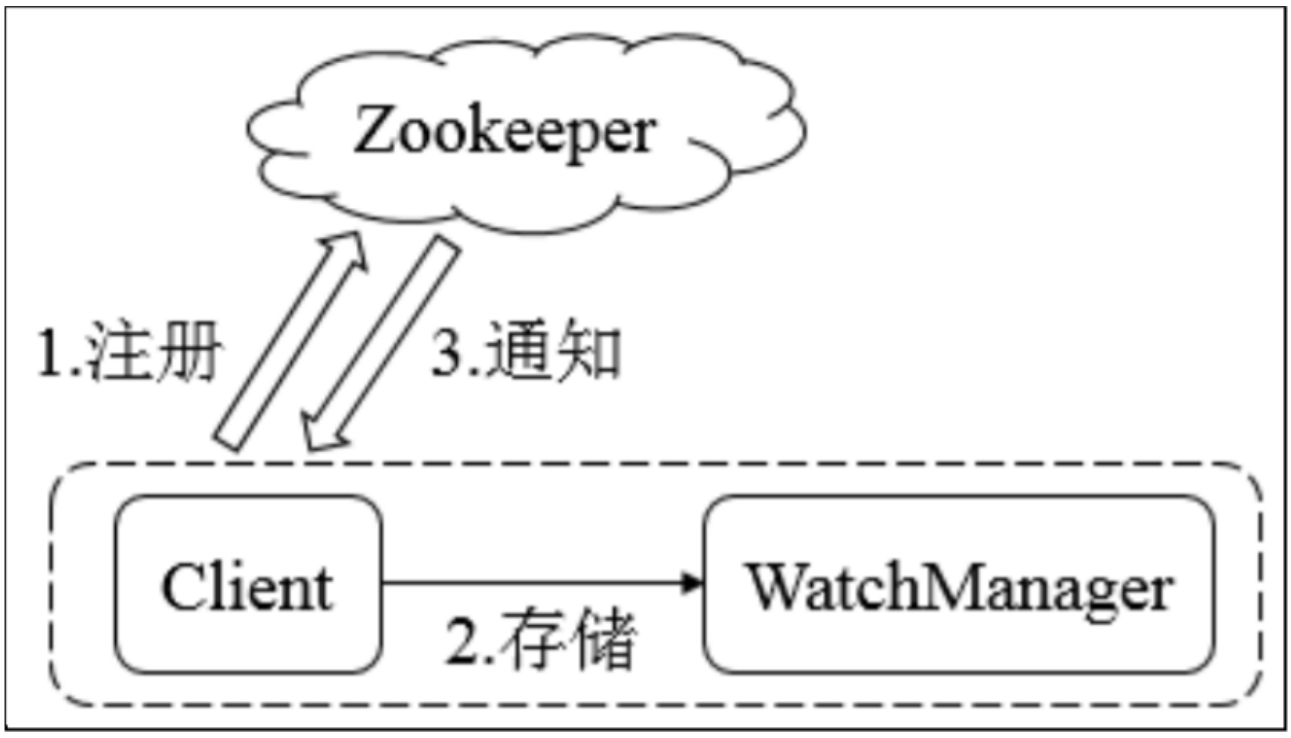
**① 注册触发事件**

**② 函数本身的功能**

函数的本身的功能又可以用异步的回调函数来实现,重载processResult()过程中处理函数本身的的功能。

## 监听工作原理

ZooKeeper 的 Watcher 机制主要包括**客户端线程、客户端 WatcherManager、Zookeeper 服务器**三部分。客户端在向 ZooKeeper 服务器注册的同时，会将 Watcher 对象存储在客户端的 WatcherManager 当中。当 ZooKeeper 服务器触发 Watcher 事件后，会向客户端发送通知， 客户端线程从 WatcherManager 中取出对应的 Watcher 对象来执行回调逻辑




回到顶部

## ZooKeeper应用举例

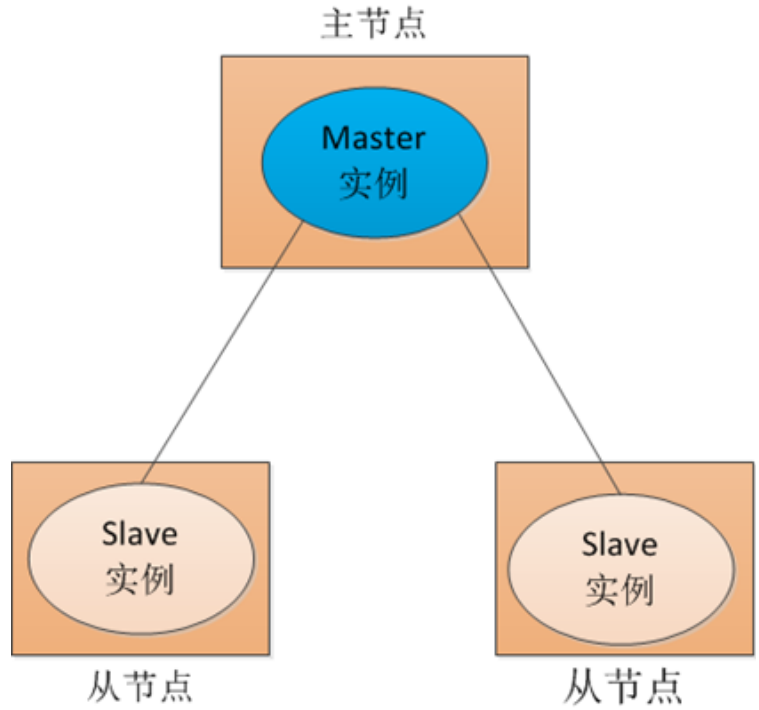
为了方便大家理解ZooKeeper，在此就给大家举个例子，看看ZooKeeper是如何实现的他的服务的，我以ZooKeeper提供的基本服务**分布式锁**为例。

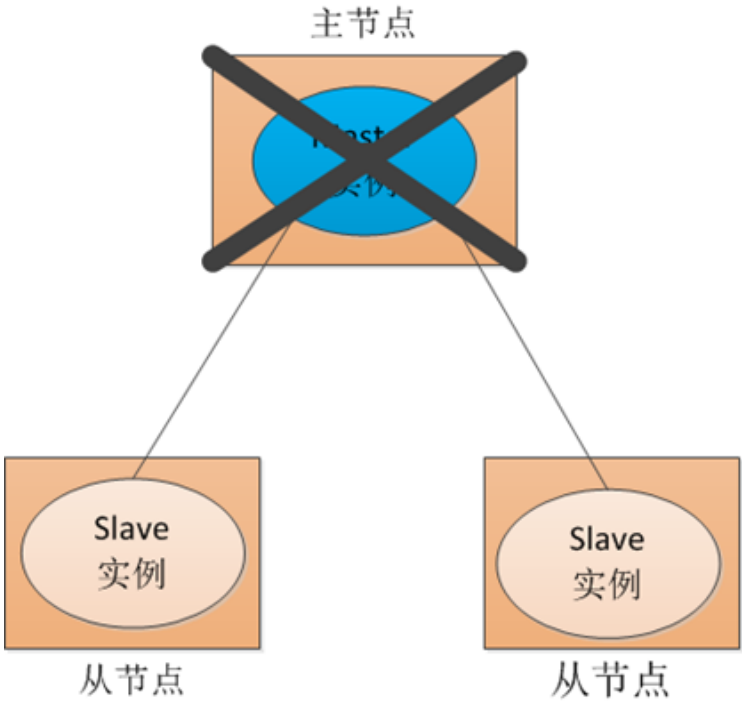
## 分布式锁应用场景

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7.1和7.2所示：

**图 7.1 主从模式分布式系统**

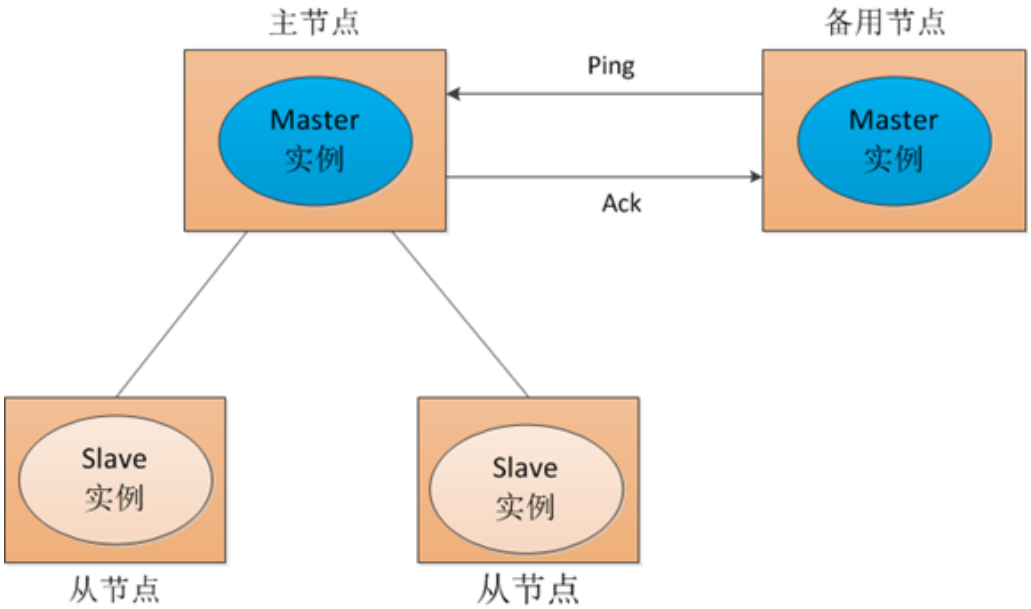
**图7.2 单点故障**



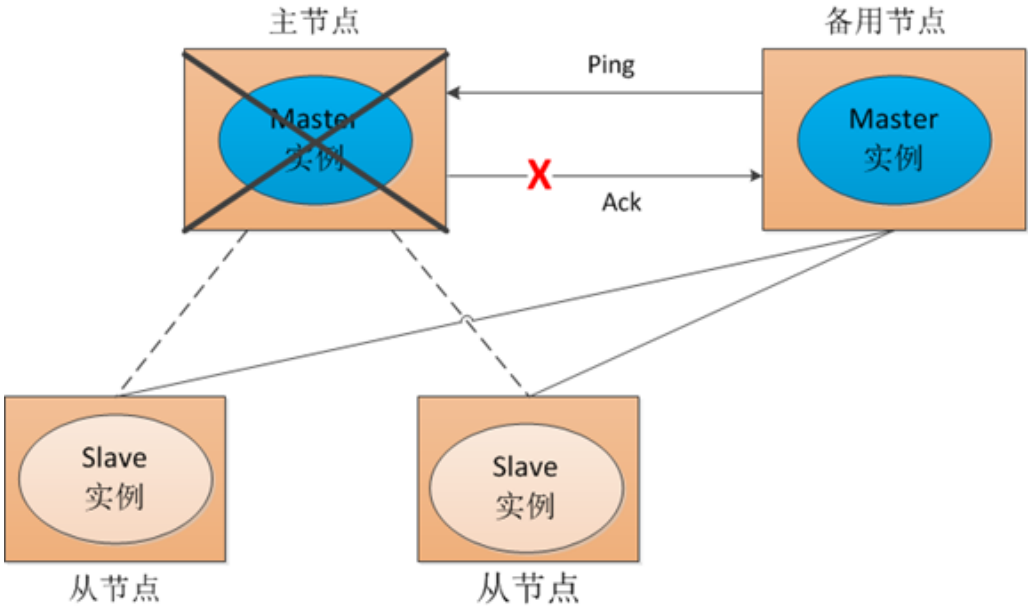




## 传统解决方案

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：

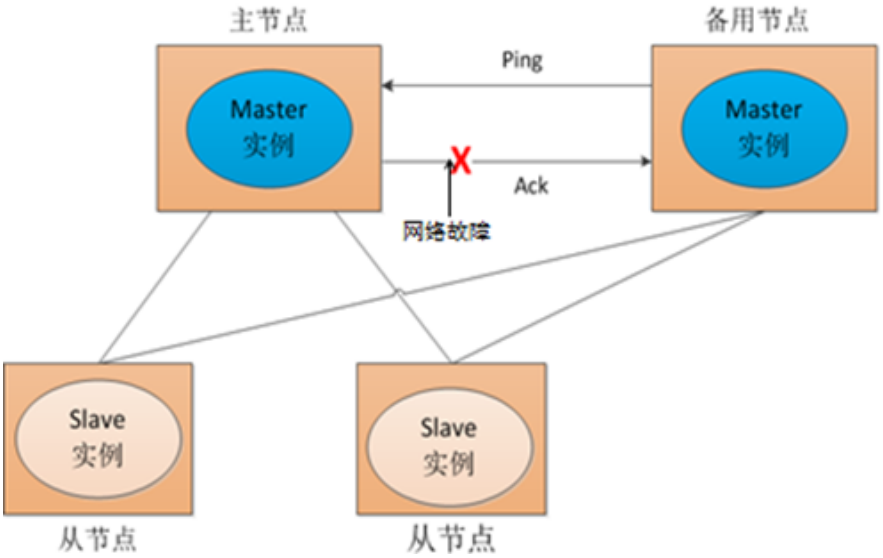



当主节点挂了，这时候备用节点收不到回复了，然后他就认为主节点挂了接替他成为主节点如下图7.4所示：




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：

图 7.5 网络故障



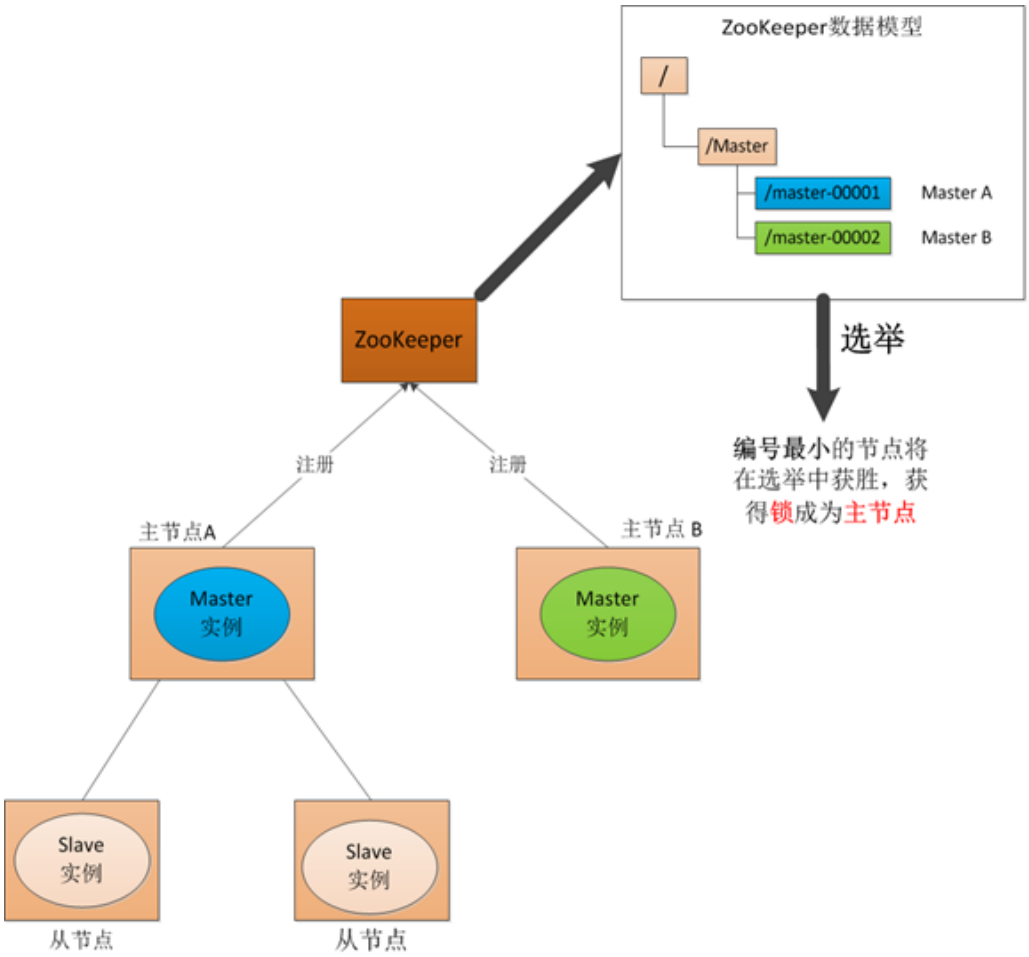
也就是说我们的主节点的并没有挂，只是在回复的时候网络发生故障，这样我们的备用节点同样收不到回复，就会认为主节点挂了，然后备用节点将他的Master实例启动起来，这样我们的分布式系统当中就有了两个主节点也就是---**双Master**，出现Master以后我们的从节点就会将它所做的事一部分汇报给了主节点，一部分汇报给了从节点，这样服务就全乱了。为了防止出现这种情况，我们引入了ZooKeeper，它虽然不能避免网络故障，但它能够保证每时每刻只有一个Master。我么来看一下ZooKeeper是如何实现的。

## ZooKeeper解决方案

### 1) Master启动

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。

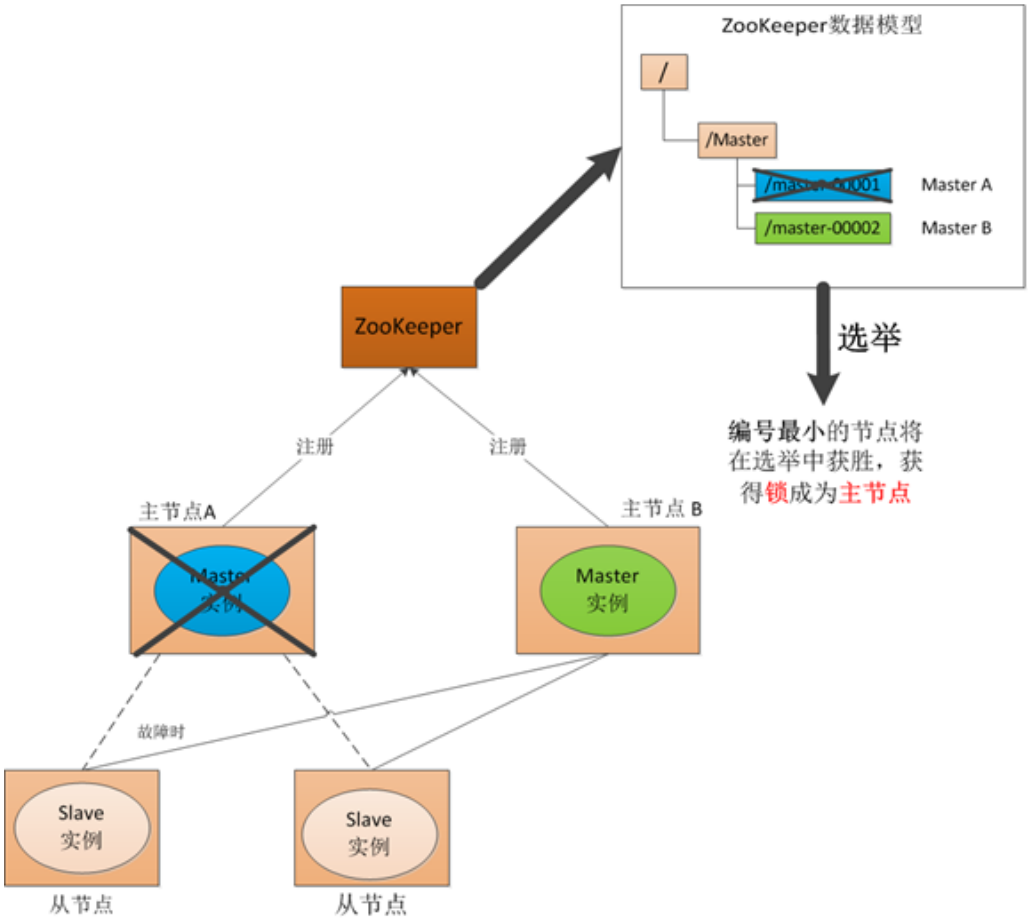
图7.6 ZooKeeper Master选举




### (2) Master故障

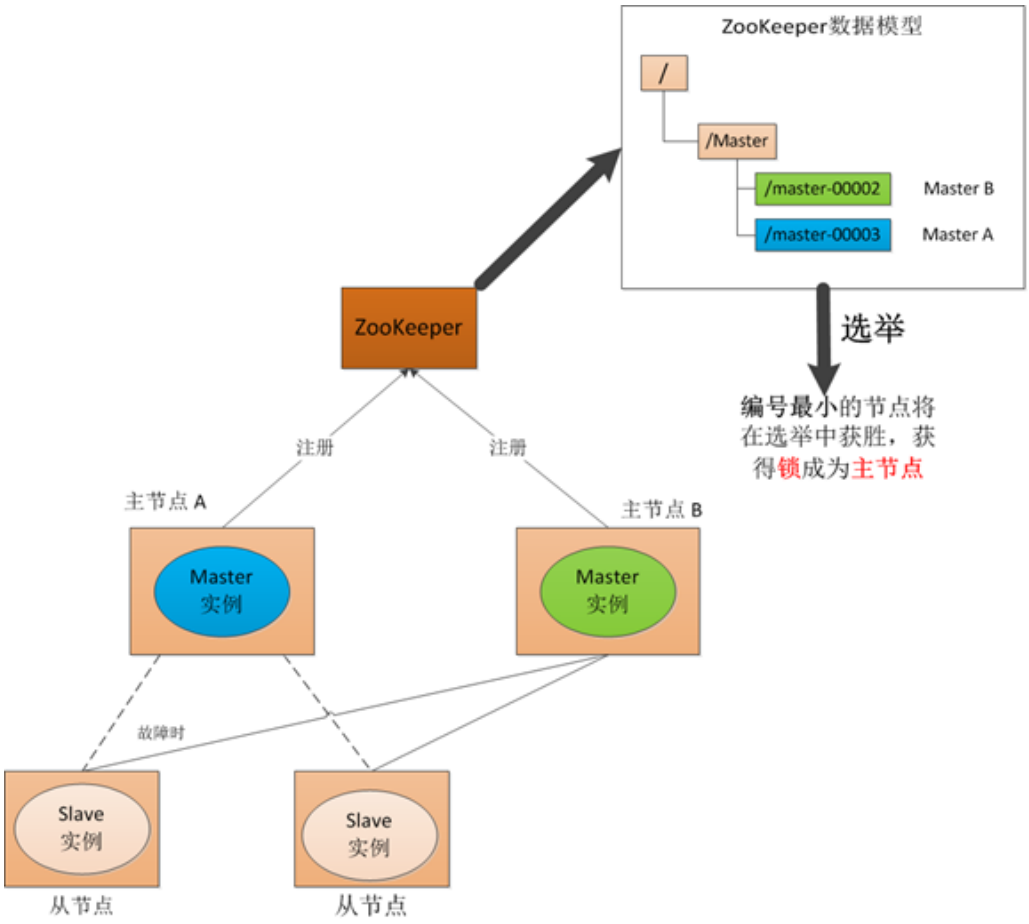
如果"主节点-A"挂了，这时候他所注册的节点将被自动删除，ZooKeeper会自动感知节点的变化，然后再次发出选举，这时候"主节点-B"将在选举中获胜，替代"主节点-A"成为主节点。

图7.7 ZooKeeper Master选举




**(3) Master 恢复**

图7.8 ZooKeeper Master选举




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。