Zookeeper教程

1. **简介**

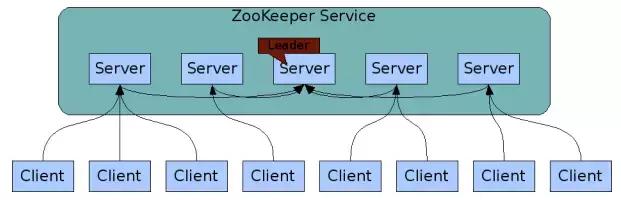
Zookeeper 分布式服务框架是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理、队列管理等。

终于了解完我们能用zookeeper做什么了，可是作为一个程序员，我们总是想狂热了解zookeeper是如何做到这一点的，单点维护一个文件系统没有什么难度，可是如果是一个集群维护一个文件系统保持数据的一致性就非常困难了。

简单的说，zookeeper=文件系统+通知机制。

为了保证高可用，zookeeper需要以集群形态来部署，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么zookeeper本身仍然是可用的。客户端在使用zookeeper时，需要知道集群机器列表，通过与集群中的某一台机器建立TCP连接来使用服务，客户端使用这个TCP链接来发送请求、获取结果、获取监听事件以及发送心跳包。如果这个连接异常断开了，客户端可以连接到另外的机器上。

架构图如下：



客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理。对于写请求，这些请求会同时发给其他zookeeper机器并且达成一致后，请求才会返回成功。因此，随着zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是zookeeper中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper最新的zxid。

**Zookeeper的特点：**

（1）全局数据一致：每个server也就是每个zk节点保存一份相同的数据副本，client连接到哪个server，展示的数据都是一致的，这是最重要的特征。

（2）可靠性：如果消息被其中一台服务器接受，那么将被所有的服务器接受。也就是对一台服务器的增删操作被接受，那么其他服务器也将接受这个操作，保持数据一致性。

（3）顺序性：包括全局有序和偏序两种，全局有序是指如果在一台服务器上消息a在消息b前发布，则在所有server上消息a都将在消息b前被发布；偏序是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

（4）数据更新的原子性：一次数据更新要么成功（半数以上结点成功），要么失败，不存在中间状态。

（5）实时性：zookeeper保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。

**通知机制：**

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

1. **通知机制的应用**

**命名服务：**

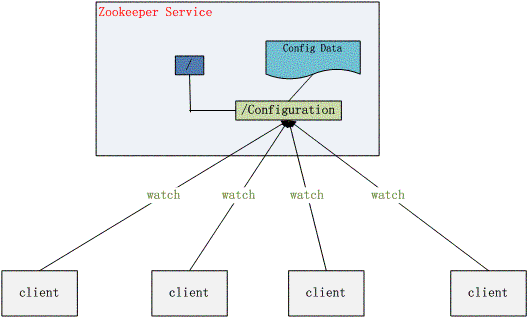
这个似乎最简单，在zookeeper的文件系统里创建一个目录，即有唯一的path。在我们使用tborg无法确定上游程序的部署机器时即可与下游程序约定好path，通过path即能互相探索发现，不见不散了。这个功能也就是我们常用的注册服务和使用服务。

在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务地址，提供者信息等。被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等。这些我们都可以统称他们为名字。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用zk提供的创建节点的api，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。

阿里巴巴集团开源的分布式服务框架Dubbo中使用zookeeper来作为其命名服务，维护全局的服务地址列表。

**配置管理：**

程序总是需要配置的，如果程序分散部署在多台机器上，要逐个改变配置就变得困难。好吧，现在把这些配置全部放到zookeeper上去，保存在 Zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中就好。具体实现思路可以查看数据发布和订阅这项。

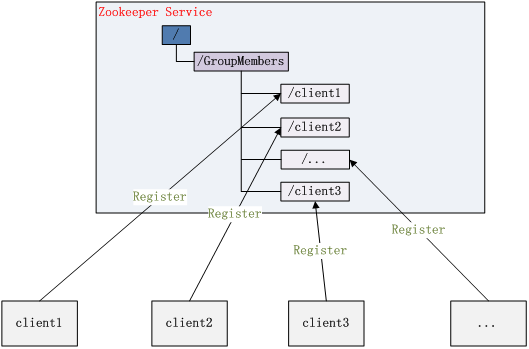


**集群管理：**

所谓集群管理无在乎两点：是否有机器退出和加入、选举master。

对于第一点，所有机器约定在父目录GroupMembers下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与zookeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它下船了。新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount又有了。

对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为master就好。

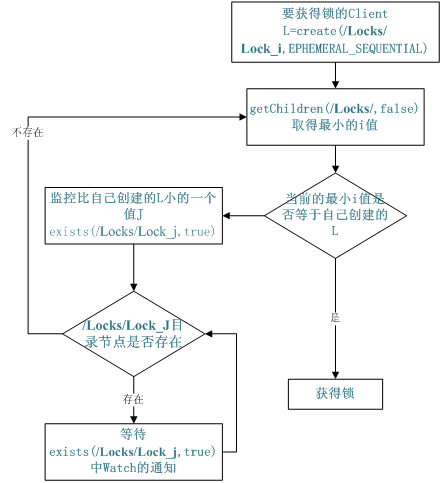


1. **分布式锁**

有了zookeeper的强一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute\_lock 节点（临时节点，且非序列化，这样会话断开，临时节点就会被删除，其他应用去监听该目录是否存在，不存在则自己就可以去创建该节点，拿到锁），最终成功创建的那个客户端也即拥有了这把锁。厕所有言：来也冲冲，去也冲冲，用完删除掉自己创建的distribute\_lock 节点就释放出锁。

对于第二类， /distribute\_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点（临时节点，序列化节点），和选master一样，编号最小的获得锁，用完删除，依次方便。



详细点说：

在描述算法流程之前，先看下zookeeper中几个关于节点的有趣的性质：

**有序节点：**假如当前有一个父节点为/lock，我们可以在这个父节点下面创建子节点；zookeeper提供了一个可选的有序特性，例如我们可以创建子节点“/lock/node-”并且指明有序，那么zookeeper在生成子节点时会根据当前的子节点数量自动添加整数序号，也就是说如果是第一个创建的子节点，那么生成的子节点为/lock/node-0000000000，下一个节点则为/lock/node-0000000001，依次类推。

**临时节点：**客户端可以建立一个临时节点，在会话结束或者会话超时后，zookeeper会自动删除该节点。

**事件监听：**在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，zookeeper会通知客户端。当前zookeeper有如下四种事件：1）节点创建；2）节点删除；3）节点数据修改；4）子节点变更。

下面描述使用zookeeper实现分布式锁的算法流程，假设锁空间的根节点为/lock：

客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-0000000000，第二个为/lock/lock-0000000001，以此类推。

客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听/lock的子节点变更消息，获得子节点变更通知后重复此步骤直至获得锁；

执行业务代码；

完成业务流程后，删除对应的子节点释放锁。

步骤1中创建的临时节点能够保证在故障的情况下锁也能被释放，考虑这么个场景：假如客户端a当前创建的子节点为序号最小的节点，获得锁之后客户端所在机器宕机了，客户端没有主动删除子节点；如果创建的是永久的节点，那么这个锁永远不会释放，导致死锁；由于创建的是临时节点，客户端宕机后，过了一定时间zookeeper没有收到客户端的心跳包判断会话失效，将临时节点删除从而释放锁。

另外细心的朋友可能会想到，在步骤2中获取子节点列表与设置监听这两步操作的原子性问题，考虑这么个场景：客户端a对应子节点为/lock/lock-0000000000，客户端b对应子节点为/lock/lock-0000000001，客户端b获取子节点列表时发现自己不是序号最小的，但是在设置监听器前客户端a完成业务流程删除了子节点/lock/lock-0000000000，客户端b设置的监听器岂不是丢失了这个事件从而导致永远等待了？这个问题不存在的。因为zookeeper提供的API中设置监听器的操作与读操作是原子执行的，也就是说在读子节点列表时同时设置监听器，保证不会丢失事件。

最后，对于这个算法有个极大的优化点：假如当前有1000个节点在等待锁，如果获得锁的客户端释放锁时，这1000个客户端都会被唤醒，这种情况称为“羊群效应”；在这种羊群效应中，zookeeper需要通知1000个客户端，这会阻塞其他的操作，最好的情况应该只唤醒新的最小节点对应的客户端。应该怎么做呢？在设置事件监听时，每个客户端应该对刚好在它之前的子节点设置事件监听，例如子节点列表为/lock/lock-0000000000、/lock/lock-0000000001、/lock/lock-0000000002，序号为1的客户端监听序号为0的子节点删除消息，序号为2的监听序号为1的子节点删除消息。

使用示例：

虽然zookeeper原生客户端暴露的API已经非常简洁了，但是实现一个分布式锁还是比较麻烦的…我们可以直接使用curator这个开源项目提供的zookeeper分布式锁实现。

我们只需要引入下面这个包（基于maven）：

<dependency>

<groupId>org.apache.curator</groupId>

<artifactId>curator-recipes</artifactId>

<version>4.0.0</version>

</dependency>

然后就可以用啦！代码如下：

public static void main(String[] args) throws Exception {

//创建zookeeper的客户端

RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);

CuratorFramework client = CuratorFrameworkFactory.newClient("10.21.41.181:2181,10.21.42.47:2181,10.21.49.252:2181", retryPolicy);

client.start();

//创建分布式锁, 锁空间的根节点路径为/curator/lock

InterProcessMutex mutex = new InterProcessMutex(client, "/curator/lock");

mutex.acquire();

//获得了锁, 进行业务流程

System.out.println("Enter mutex");

//完成业务流程, 释放锁

mutex.release();

//关闭客户端

client.close();

}

备注：此段代码没有经过试验验证

1. **队列管理**

两种类型的队列：

1、 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。

1. **工作原理**

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务等。Zookeeper是hadoop的一个子项目，其发展历程无需赘述。在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调机制不适合在某些应用中使用，因此需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper的目的就在于此。下面简单分析zookeeper的工作原理。

**角色：**

Zookeeper中的角色主要有以下三类，如下表所示：

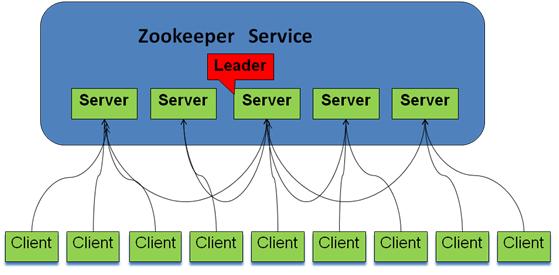
**leader领导者：**zookeeper集群工作的核心，事务请求（写，删除，修改操作）的唯一调度和处理者，保证集群事务处理的顺序性，集群内部各个服务器的调度者。它还负责进行投票的发起和决议，更新系统状态。

对于create、setData、delete等有写操作的请求，则需要统一转发给leader处理，leader需要决定编号，执行操作，这个过程称为一个事务。

**Follower追随者：**处理客户端非事务的请求（比如读取数据），转发事务请求给leader，参与集群leader的选举投票。

**ObServer观察者：**针对访问量比较大的zookeeper集群，还可新增观察者角色，它是观察zookeeper集群的最新状态变化并将这些状态同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则会转发给leader服务器进行处理，它不会参与任何形式的投票只提供非事务服务，通常用于在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

系统模型如图所示：



**设计目的：**

1.最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

2 .可靠性：具有简单、健壮、良好的性能，如果消息m被到一台服务器接受，那么它将被所有的服务器接受。

3 .实时性：Zookeeper保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

4 .等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

5.原子性：更新只能成功或者失败，没有中间状态。

6 .顺序性：包括全局有序和偏序两种：全局有序是指如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布；偏序是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

**原理：**

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，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，标识当前属于那个leader的统治时期。低32位用于递增计数。

每个Server在工作过程中有三种状态：

• LOOKING：当前Server不知道leader是谁，正在搜寻

• LEADING：当前Server即为选举出来的leader

• FOLLOWING：leader已经选举出来，当前Server与之同步

**选主算法：**

zk默认的选举算法是FastLeaderElection，采用投票数大于半数则胜出的逻辑。

相关概念：

服务器id：比如有三台服务器，编号分别为1、2、3，编号越大在选举算法中的权重越大。

选举状态：

LOOKING：竞选状态

FOLLOWING：随从状态，同步leader状态，参与投票

OBSERVING：观察状态，同步leader状态，不参与投票

LEADING：领导者状态

数据ID：服务器中存放的最新数据的version，值越大说明数据越新，在选举算法中数据越新，权重越大。

逻辑时钟：也叫投票次数，同一轮投票过程中的逻辑时钟值是相同的，每投完一次票这个数据就会增加，然后与接收到的其他服务器返回的投票信息中的数值相比，根据不同的值做出不同的判断。

**全新集群选举：**

也就是所有服务器刚启动时，选举leader的算法。

规则：

1.此时每个机器都是投票给自己

2.投票数过半选举结束

我们比方有5台服务器，A（编号1）、B（编号2）、C（编号3）、D（编号4）、E（编号5）

选举过程：

步骤1：启动服务器A，A启动后，投票给自己，此时发现只有自己启动了，还没有其他服务器启动，因此进入looking状态

步骤2：启动服务器B，此时B也投票给自己，服务器A收到B的投票信息，进行比对，发现B的编号比自己大，因此A放弃投票给自己，进而选择投票给服务器B，此时有两台投票给B，但是还未过半，比3小，因此B也处于looking状态。

步骤3：启动服务器C，投票给自己，然后A、B收到C的投票信息，跟自己进行比对，发现C的编号比自己投票的服务器编号大，因此放弃自己之前的投票，转而投票给C，此时C得票3个，已经过半，此时选举结束，C变成leader，A、B服务器转为following状态。

步骤4：启动服务器D,给自己投票，然后收到其他服务器的信息，发现此时投票已经结束，服务器D只能做follower。服务器E也是一样的。

**非全新集群选举：**

也就是当集群运行一段时间后，leader由于某些原因挂掉了，此时就需要重新进行leader选举。选举过程就需要加入数据ID、服务器ID和逻辑时钟。

数据ID：数据新的version就大，数据每次更新都会更新version。

服务器ID：就是我们配置的myid中的值，每个机器一个

逻辑时钟：这个值从0开始递增，每次选举对应一个值，如果在同一次选举中，这个值是一致的。

选举标准：

（1）逻辑时钟小的选举结果被忽略（逻辑时钟小可能是中途挂掉了），重新投票。

（2）统一逻辑时钟后，数据id大的胜出

（3）数据id相同的情况下，服务器id大的胜出。

根据以上三个标准选出leader。

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。先介绍basic paxos流程。

**basic paxos流程：**

1、选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

2、选举线程首先向所有Server发起一次询问(包括自己)；

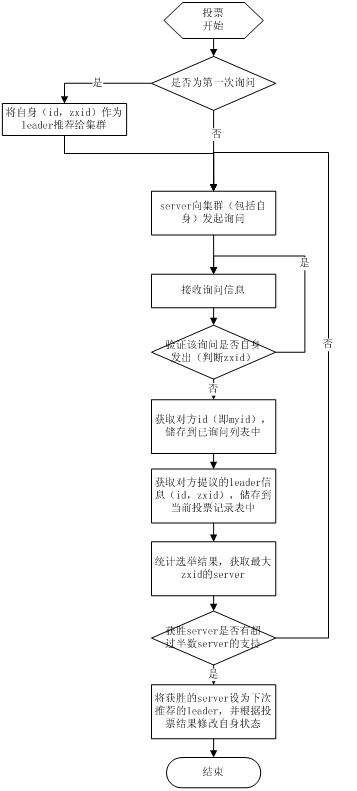
3、 选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

4、收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

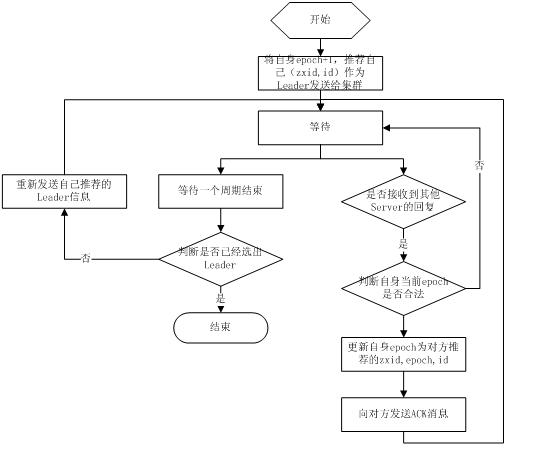
5、线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数， 设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。

通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1.

每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。选主的具体流程图如下所示：



fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。其流程图如下所示：



**同步数据：**

选完leader以后，zk就进入状态同步过程。

1、leader等待server连接；

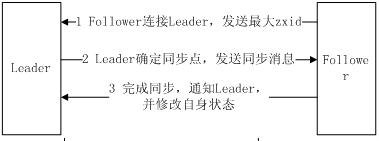
2、Follower连接leader，将最大的zxid发送给leader；

3、Leader根据follower的zxid确定同步点；

4、完成同步后通知follower 已经成为uptodate状态；

5、Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。

流程图如下所示：



**工作流程：**

Leader工作流程：

Leader主要有三个功能：

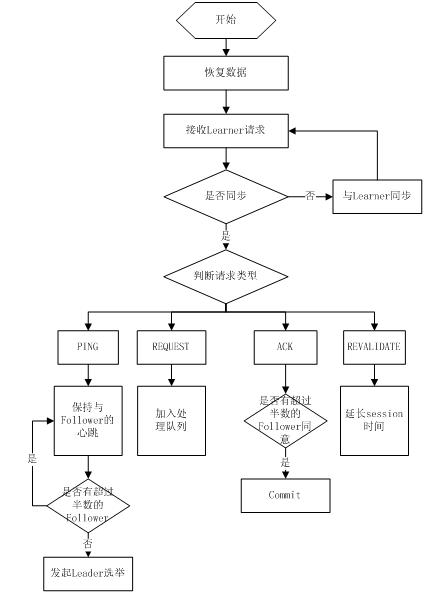
1、恢复数据；

2、维持与Learner的心跳，接收Learner请求并判断Learner的请求消息类型；

3、Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。

PING消息是指Learner的心跳信息；REQUEST消息是Follower发送的提议信息，包括写请求及同步请求；ACK消息是Follower的对提议的回复，超过半数的Follower通过，则commit该提议；REVALIDATE消息是用来延长SESSION有效时间。

Leader的工作流程简图如下所示，在实际实现中，流程要比下图复杂得多，启动了三个线程来实现功能。



**Follower工作流程：**

Follower主要有四个功能：

1、向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；

2、接收Leader消息并进行处理；

3、接收Client的请求，如果为写请求，发送给Leader进行投票；

4、返回Client结果。

Follower的消息循环处理如下几种来自Leader的消息：

1、PING消息： 心跳消息；

2、PROPOSAL消息：Leader发起的提案，要求Follower投票；

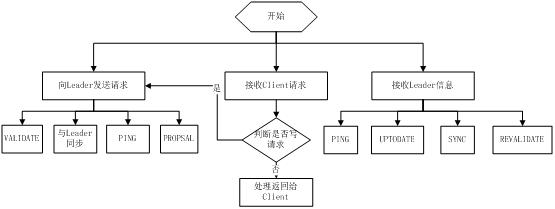
3、COMMIT消息：服务器端最新一次提案的信息；

4、UPTODATE消息：表明同步完成；

5、REVALIDATE消息：根据Leader的REVALIDATE结果，关闭待revalidate的session还是允许其接受消息；

6、SYNC消息：返回SYNC结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

Follower的工作流程简图如下所示，在实际实现中，Follower是通过5个线程来实现功能的。



对于observer的流程不再叙述，observer流程和Follower的唯一不同的地方就是observer不会参加leader发起的投票。

**数据一致性与paxos算法：**

据说Paxos算法的难理解与算法的知名度一样令人敬仰，所以我们先看如何保持数据的一致性，这里有个原则就是：

在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。

Paxos算法解决的什么问题呢，解决的就是保证每个节点执行相同的操作序列。好吧，这还不简单，master维护一个全局写队列，所有写操作都必须放入这个队列编号，那么无论我们写多少个节点，只要写操作是按编号来的，就能保证一致性。没错，就是这样，可是如果master挂了呢。

Paxos算法通过投票来对写操作进行全局编号，同一时刻，只有一个写操作被批准，同时并发的写操作要去争取选票，只有获得过半数选票的写操作才会被批准（所以永远只会有一个写操作得到批准），其他的写操作竞争失败只好再发起一轮投票，就这样，在日复一日年复一年的投票中，所有写操作都被严格编号排序。编号严格递增，当一个节点接受了一个编号为100的写操作，之后又接受到编号为99的写操作（因为网络延迟等很多不可预见原因），它马上能意识到自己数据不一致了，自动停止对外服务并重启同步过程。任何一个节点挂掉都不会影响整个集群的数据一致性（总2n+1台，除非挂掉大于n台）。

1. **分布式与数据复制**

Zookeeper作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处：

1、容错

一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；

2、提高系统的扩展能力

把负载分布到多个节点上，或者增加节点来提高系统的负载能力；

3、提高性能

让客户端本地访问就近的节点，提高用户访问速度。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：

1、写主(WriteMaster)

对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离；

2、写任意(Write Any)

对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对zookeeper来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立observer的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

1. **ZooKeeper典型使用场景**

ZooKeeper是一个高可用的分布式数据管理与系统协调框架。基于对Paxos算法的实现，使该框架保证了分布式环境中数据的强一致性，也正是基于这样的特性，使得zookeeper能够应用于很多场景。网上对zk的使用场景也有不少介绍，本文将结合作者身边的项目例子，系统的对zk的使用场景进行归类介绍。 值得注意的是，zk并不是生来就为这些场景设计，都是后来众多开发者根据框架的特性，摸索出来的典型使用方法。因此，也非常欢迎你分享你在ZK使用上的奇技淫巧。

|  |  |  |
| --- | --- | --- |
| **场景类别** | **典型场景描述（ZK特性，使用方法）** | **应用中的具体使用** |
| **数据发布与订阅** | 发布与订阅即所谓的配置管理，顾名思义就是将数据发布到zk节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，地址列表等就非常适合使用。 | 1. 索引信息和集群中机器节点状态存放在zk的一些指定节点，供各个客户端订阅使用。2. 系统日志（经过处理后的）存储，这些日志通常2-3天后被清除。    3. 应用中用到的一些配置信息集中管理，在应用启动的时候主动来获取一次，并且在节点上注册一个Watcher，以后每次配置有更新，实时通知到应用，获取最新配置信息。  4. 业务逻辑中需要用到的一些全局变量，比如一些消息中间件的消息队列通常有个offset，这个offset存放在zk上，这样集群中每个发送者都能知道当前的发送进度。  5. 系统中有些信息需要动态获取，并且还会存在人工手动去修改这个信息。以前通常是暴露出接口，例如JMX接口，有了zk后，只要将这些信息存放到zk节点上即可。 |
| **Name Service** | 这个主要是作为分布式命名服务，通过调用zk的create node api，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。 |  |
| **分布通知/协调** | ZooKeeper中特有watcher注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容及子节点的），其中一个系统update了znode，那么另一个系统能够收到通知，并作出相应处理。 | 1. 另一种心跳检测机制：检测系统和被检测系统之间并不直接关联起来，而是通过zk上某个节点关联，大大减少系统耦合。2. 另一种系统调度模式：某系统有控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改了ZK上某些节点的状态，而zk就把这些变化通知给他们注册Watcher的客户端，即推送系统，于是，作出相应的推送任务。    3. 另一种工作汇报模式：一些类似于任务分发系统，子任务启动后，到zk来注册一个临时节点，并且定时将自己的进度进行汇报（将进度写回这个临时节点），这样任务管理者就能够实时知道任务进度。  总之，使用zookeeper来进行分布式通知和协调能够大大降低系统之间的耦合。 |
| **分布式锁** | 分布式锁，这个主要得益于ZooKeeper为我们保证了数据的强一致性，即用户只要完全相信每时每刻，zk集群中任意节点（一个zk server）上的相同znode的数据是一定是相同的。锁服务可以分为两类，**一个是保持独占，另一个是控制时序。**    所谓保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把zk上的一个znode看作是一把锁，通过create znode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。  控制时序，就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distribute\_lock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制：CreateMode.EPHEMERAL\_SEQUENTIAL来指定）。Zk的父节点（/distribute\_lock）维持一份sequence,保证子节点创建的时序性，从而也形成了每个客户端的全局时序。 |  |
| **集群管理** | 1. **集群机器**监控：这通常用于那种对集群中机器状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。这样的场景中，往往有一个监控系统，实时检测集群机器是否存活。过去的做法通常是：监控系统通过某种手段（比如ping）定时检测每个机器，或者每个机器自己定时向监控系统汇报“我还活着”。 这种做法可行，但是存在两个比较明显的问题：1. 集群中机器有变动的时候，牵连修改的东西比较多。2. 有一定的延时。    利用ZooKeeper有两个特性，就可以实时另一种集群机器存活性监控系统：a. 客户端在节点 x 上注册一个Watcher，那么如果 x 的子节点变化了，会通知该客户端。b. 创建EPHEMERAL类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。  例如，监控系统在 /clusterServers 节点上注册一个Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。 2. **Master选举则是zookeeper中最为经典的使用场景了。**  在分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑（例如一些耗时的计算，网络I/O处理），往往只需要让整个集群中的某一台机器进行执行，其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能，于是这个master选举便是这种场景下的碰到的主要问题。  利用ZooKeeper的强一致性，能够保证在分布式高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /currentMaster 节点，最终一定只有一个客户端请求能够创建成功。  利用这个特性，就能很轻易的在分布式环境中进行集群选取了。  另外，这种场景演化一下，就是动态Master选举。这就要用到 EPHEMERAL\_SEQUENTIAL类型节点的特性了。  上文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终在ZK上创建结果的一种可能情况是这样： /currentMaster/{sessionId}-1 , /currentMaster/{sessionId}-2 , /currentMaster/{sessionId}-3 ….. 每次选取序列号最小的那个机器作为Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是Master了。 | 1. 在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此之间索引数据一致。因此让集群中的Master来进行全量索引的生成，然后同步到集群中其它机器。2. 另外，Master选举的容灾措施是，可以随时进行手动指定master，就是说应用在zk在无法获取master信息时，可以通过比如http方式，向一个地方获取master。 |
| **分布式队列** | 队列方面，我目前感觉有两种，**一种是常规的先进先出队列，另一种是要等到队列成员聚齐之后的才统一按序执行**。对于第二种先进先出队列，和分布式锁服务中的控制时序场景基本原理一致，这里不再赘述。    第二种队列其实是在FIFO队列的基础上作了一个增强。通常可以在 /queue 这个znode下预先建立一个/queue/num 节点，并且赋值为n（或者直接给/queue赋值n），表示队列大小，之后每次有队列成员加入后，就判断下是否已经到达队列大小，决定是否可以开始执行了。这种用法的典型场景是，分布式环境中，一个大任务Task A，需要在很多子任务完成（或条件就绪）情况下才能进行。这个时候，凡是其中一个子任务完成（就绪），那么就去 /taskList 下建立自己的临时时序节点（CreateMode.EPHEMERAL\_SEQUENTIAL），当 /taskList 发现自己下面的子节点满足指定个数，就可以进行下一步按序进行处理了。 |  |

1. **Znode**

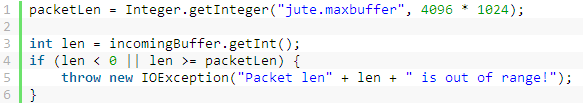
Zookeeper提供一个多层级的节点命名空间（节点称为znode），每个节点都用一个以斜杠（/）分隔的路径表示，而且每个节点都有父节点（根节点除外），非常类似于文件系统。例如，/foo/doo这个表示一个znode，它的父节点为/foo，父父节点为/，而/为根节点没有父节点。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得Zookeeper不能用于存放大量的数据，每个节点的存放数据上限为1M。

**Znode大小限制：**

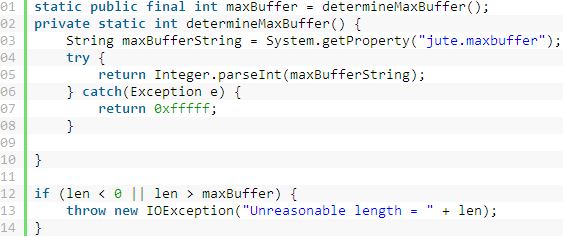
如果你仔细看过ZooKeeper的文档，会发现文档中对ZNode的大小做了限制，最大不能超过1M。

这个1M的大小限制在ZooKeeper的客户端和服务端都有限制：

客户端：



服务端：



可以看出，ZooKeeper确实对数据的大小有限制，默认就是1M，如果希望传输超过1M的数据，可以修改环境变量“jute.maxbuffer”即可。

**限制原因：**

ZooKeeper是一套高吞吐量的系统，为了提高系统的读取速度，ZooKeeper不允许从文件中读取需要的数据，而是直接从内存中查找。

还句话说，ZooKeeper集群中每一台服务器都包含全量的数据，并且这些数据都会加载到内存中。同时ZNode的数据并支持Append操作，全部都是Replace。

所以从上面分析可以看出，如果ZNode的过大，那么读写某一个ZNode将造成不确定的延时;同时ZNode过大，将过快地耗尽ZooKeeper服务器的内存。这也是为什么ZooKeeper不适合存储大量的数据的原因。

1. **Paxos算法**

Paxos读音：

在paxos算法中，分为4种角色：

Proposer ：提议者

Acceptor：决策者

Client：产生议题者

Learner：最终决策学习者

上面4种角色中，提议者和决策者是很重要的，其他的2个角色在整个算法中应该算做打酱油的，Proposer就像Client的使者，由Proposer使者拿着Client的议题去向Acceptor提议，让Acceptor来决策。这里上面出现了个新名词：最终决策。现在来系统的介绍一下paxos算法中所有的行为：

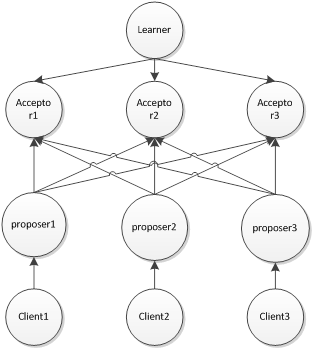
Proposer提出议题

Acceptor初步接受 或者 Acceptor初步不接受

如果上一步Acceptor初步接受则Proposer再次向Acceptor确认是否最终接受

Acceptor 最终接受 或者Acceptor 最终不接受

上面Learner最终学习的目标是Acceptor们最终接受了什么议题？注意，这里是向所有Acceptor学习，如果有多数派个Acceptor最终接受了某提议，那就得到了最终的结果，算法的目的就达到了。画一幅图来更加直观：



为什么需要3个Acceptor？因为Acceptor必须是最少大于等于3个，并且必须是奇数个，因为要形成多数派嘛，如果是偶数个，比如4个，2个接受2个不接受，各执己见，没法搞下去了。

为什么是3个Proposer？ 其实无所谓是多少个了，1~n 都可以的；如果是1个proposer，毫无竞争压力，很顺利的完成2阶段提交，Acceptor们最终批准了事。如果是多个proposer就比较复杂了，请继续看。

上面的图中是画了很多节点的，每个节点需要一台机器么？答案是不需要的，上面的图是逻辑图，物理中，可以将Acceptor和Proposer以及Client放到一台机器上，只是使用了不同的端口号罢了，Acceptor们启动不同端口的TCP监听，Proposer来主动连接即可；完全可以将Client、Proposer、Acceptor、Learner合并到一个程序里面；这里举一个例子：比如开发一个JOB程序，JOB程序部署在多台服务器上(数量为奇数)，这些JOB有可能同时处理一项任务，现在使用paxos算法让这些JOB自己来商量由谁(哪台机器)来处理这项任务，这样JOB程序里就需要包含Client、Proposer、Acceptor、Learner这4大功能，并且需要配置其他JOB服务器的IP地址。

再举一个例子，zookeeper常常用来做分布式事务锁。Zookeeper所使用的zab协议也是类似paxos协议的。所有分布式自协商一致性算法都是paxos算法的简化或者变种。Client是使用zookeeper服务的机器，Zookeeper自身包含了Acceptor, Proposer, Learner。Zookeeper领导选举就是paxos过程，还有Client对Zookeeper写Znode时，也是要进行Paxos过程的，因为不同Client可能连接不同的Zookeeper服务器来写Znode，到底哪个Client才能写成功？需要依靠Zookeeper的paxos保证一致性，写成功Znode的Client自然就是被最终接受了，Znode包含了写入Client的IP与端口，其他的Client也可以读取到这个Znode来进行Learner。也就是说在Zookeeper自身包含了Learner(因为Zookeeper为了保证自身的一致性而会进行领导选举，所以需要有Learner的内部机制，多个Zookeeper服务器之间需要知道现在谁是领导了)，Client端也可以Learner，Learner是广义的。

现在通过一则故事来学习paxos的算法的流程(2阶段提交)，有2个Client(老板，老板之间是竞争关系)和3个Acceptor(政府官员)：

现在需要对一项议题来进行paxos过程，议题是“A项目我要中标！”，这里的“我”指每个带着他的秘书Proposer的Client老板。

Proposer当然听老板的话了，赶紧带着议题和现金去找Acceptor政府官员。

作为政府官员，当然想谁给的钱多就把项目给谁。

Proposer-1小姐带着现金同时找到了Acceptor-1~Acceptor-3官员，1与2号官员分别收取了10比特币，找到第3号官员时，没想到遭到了3号官员的鄙视，3号官员告诉她，Proposer-2给了11比特币。不过没关系，Proposer-1已经得到了1,2两个官员的认可，形成了多数派(如果没有形成多数派，Proposer-1会去银行提款在来找官员们给每人20比特币，这个过程一直重复每次+10比特币，直到多数派的形成)，满意的找老板复命去了，但是此时Proposer-2保镖找到了1,2号官员，分别给了他们11比特币，1,2号官员的态度立刻转变，都说Proposer-2的老板懂事，这下子Proposer-2放心了，搞定了3个官员，找老板复命去了，当然这个过程是第一阶段提交，只是官员们初步接受贿赂而已。故事中的比特币是编号，议题是value。

这个过程保证了在某一时刻，某一个proposer的议题会形成一个多数派进行初步支持；

===============华丽的分割线，第一阶段结束================

现在进入第二阶段提交，现在proposer-1小姐使用分身术(多线程并发)分了3个自己分别去找3位官员，最先找到了1号官员签合同，遭到了1号官员的鄙视，1号官员告诉他proposer-2先生给了他11比特币，因为上一条规则的性质proposer-1小姐知道proposer-2第一阶段在她之后又形成了多数派(至少有2位官员的赃款被更新了);此时她赶紧去提款准备重新贿赂这3个官员(重新进入第一阶段)，每人20比特币。刚给1号官员20比特币， 1号官员很高兴初步接受了议题，还没来得及见到2,3号官员的时候

这时proposer-2先生也使用分身术分别找3位官员(注意这里是proposer-2的第二阶段)，被第1号官员拒绝了告诉他收到了20比特币，第2,3号官员顺利签了合同，这时2，3号官员记录client-2老板用了11比特币中标，因为形成了多数派，所以最终接受了Client2老板中标这个议题，对于proposer-2先生已经出色的完成了工作；

这时proposer-1小姐找到了2号官员，官员告诉她合同已经签了，将合同给她看，proposer-1小姐是一个没有什么职业操守的聪明人，觉得跟Client1老板混没什么前途，所以将自己的议题修改为“Client2老板中标”，并且给了2号官员20比特币，这样形成了一个多数派。顺利的再次进入第二阶段。由于此时没有人竞争了，顺利的找3位官员签合同，3位官员看到议题与上次一次的合同是一致的，所以最终接受了，形成了多数派，proposer-1小姐跳槽到Client2老板的公司去了。

===============华丽的分割线，第二阶段结束===============

Paxos过程结束了，这样，一致性得到了保证，算法运行到最后所有的proposer都投“client2中标”所有的acceptor都接受这个议题，也就是说在最初的第二阶段，议题是先入为主的，谁先占了先机，后面的proposer在第一阶段就会学习到这个议题而修改自己本身的议题，因为这样没职业操守，才能让一致性得到保证，这就是paxos算法的一个过程。原来paxos算法里的角色都是这样的不靠谱，不过没关系，结果靠谱就可以了。该算法就是为了追求结果的一致性。

1. **集群搭建**

zookeeper集群搭建指的是zookeeper分布式模式安装，通常我们安装2N+1个server集群，也就是奇数个，这个是因为选举算法来的。我们此时准备了3台服务器（A,B,C）。

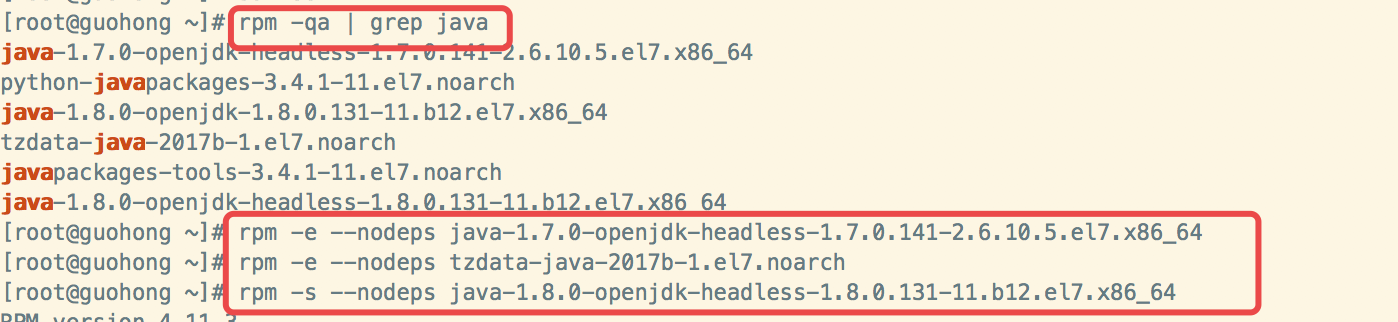
步骤1：

首先查看我们的操作系统上的jdk，是否安装，如果是openjdk我们会选择卸载，安装sun公司的jdk,查看jdk命令(centos系统)：

rpm –qa | grep java

通过上面命令可以看到很多与Java相关的软件，我们要一一卸载掉

rpm –e –nodeps 软件名

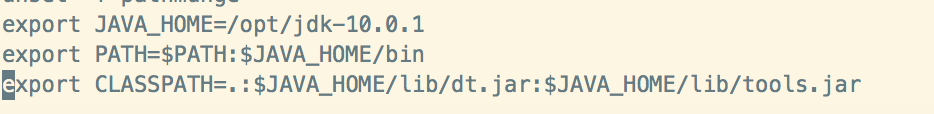


步骤2（A、B、C）：

下载jdk安装包，我们是下载到本机，然后上传到服务器，我们上传到服务器的是gz压缩文件，然后解压文件，配置jdk的环境变量。

vi /etc/profile

添加如下配置。



输入命令：source /etc/profile

然后再输入java –version 命令，如果运行正常，能看到jdk的版本号则说明配置正确。

步骤3：检查集群时间是否同步，也就是检查每台服务器的时间是否一致，使用date命令。（A、B、C）

步骤4：检查防火墙是否关闭（A、B、C）

步骤5（A）：安装zookeeper，安装zookeeper,修改zoo.cfg的dataDir=/opt/zkdata配置后。还要进行下面操作

此时我们还要添加集群中三台服务器的配置，该配置是在服务器A上配置，也是修改zoo.cfg，在配置文件末尾

server.1=192.168.0.140:2888:3888

server.2=192.168.0.141:2888:3888

server.3=192.168.0.142:2888:3888

2888是心跳检测的端口

3888是选举端口

进行A的配置，在zkdata文件夹下新增文件myid,vim myid,然后输入内容1（也就是服务器编号），注意myid的名称是固定的，不能乱写。

步骤6（B、C）：

接下来对服务器B、C进行配置。

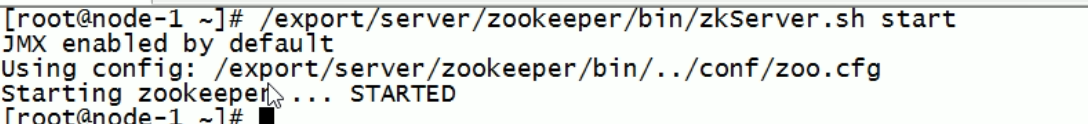
同样的，在/opt/下面新建文件夹zkdata

在zkdata文件夹下新增文件myid,B服务器myid内容为2，C服务器myid文件为3

此时就完成了3台服务器的配置。

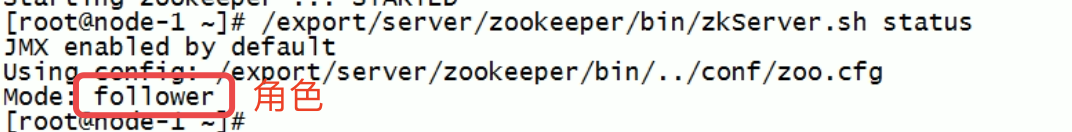
步骤7：启动zk（A、B、C）

在zk的安装包文件夹下的bin文件夹，有个zkServer.sh脚本，我们输入命令：zkServer.sh start 启动zk。

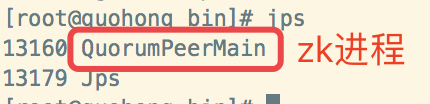


查看zk的状态：zkServer.sh status

该命令可以查看该服务器的状态，以及角色情况



查看启动后的进程情况还可以使用jps命令



1. **安装zookeeper**

zookeeper的官网：<https://zookeeper.apache.org/>

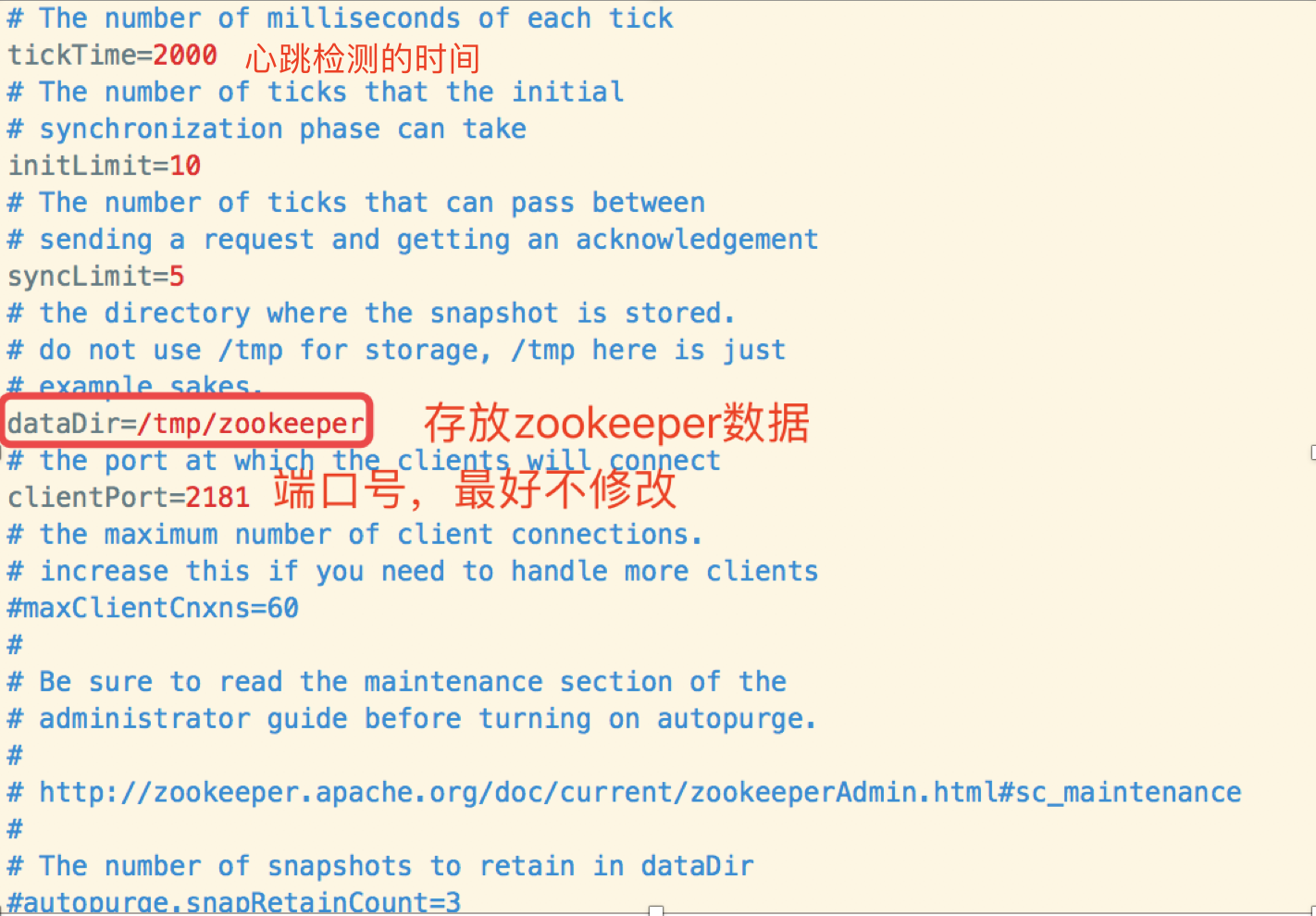
我们可以去官网下载zookeeper的安装包，然后将安装包上传到需要安装zookeeper的服务器。

我一般把安装包放到服务器的/opt目录下，此时解压tar.gz文件：tar zxvf zookeeper-版本号。

解压文件后，我们进入到解压的文件目录中，有个conf文件夹，此时需要修改配置文件。



如果我们没有指定配置文件，那么zookeeper会默认去conf文件夹下找zoo.cfg配置文件。此时我们看到conf文件夹下是没有zoo.cfg文件的，只有zoo\_sample.cfg文件，我们拷贝一份zoo\_sample.cfg命名为zoo.cfg文件就行。



一般情况下我们只需要修改dataDir的路径，也就是修改zk数据的保存地址就行。

**启动**

./zkServer.sh start

**使用客户端连接zk：**

./zkCli.sh 直接运行则连接本机上的zk，端口号是2181



根据ip地址去连接，这种操作也适合连接远程的zk服务器。

./zkCli.sh -server 127.0.0.1:2181



1. **zk的数据模型**

zk的数据模型，在结构上和标准文件系统非常相似，拥有一个层次命名空间，都是采用属性层次结构，zk树中的每个节点被称为一个znode。和文件系统的目录树一样，zk树中的每个节点可以拥有子节点，但也有不同之处。

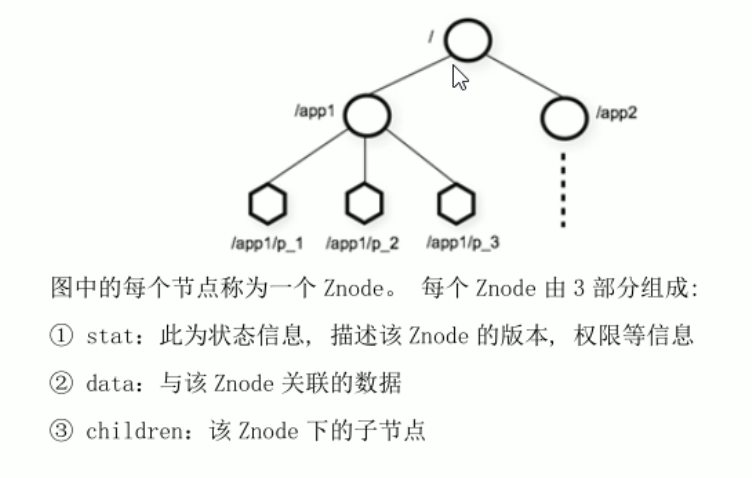
（1）znode兼具文件和目录两种特点，既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分，并可以具有子znode。用户对znode具有增删改查等操作

（2）znode具有原子性操作，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。

（3）znode存储数据大小有限制，zk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。Zk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，当时常规使用中应该远小于此值。

（4）znode通过路径引用，如同unix中的文件路径，路径必须是绝对的，因此他们必须由斜杆字符来开头。除此之外，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zk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。

数据结构图：



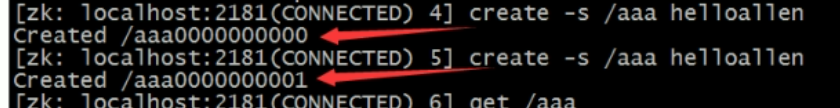
**Znode的类型：**

Znode有两种，分别为临时节点和永久节点。节点类型在创建时即被确定，并且不能改变。

**临时节点**：该节点的生命周期依赖于创建它们的会话。一旦会话结束，临时节点将被自动删除，当然也可以手动删除，临时节点不允许拥有子节点（因为如果存在子节点，万一子节点是永久节点，此时是该删还是不该删呢？）

**永久节点**：该节点的生命周期不依赖与会话，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。

Znode还有一个序列化的特性，如果创建的时候指定的话，该znode的名字后面会自动追加一个不断增加的序列号。序列号对于此节点的父节点来说是唯一的，这样便会记录每个节点创建的先后顺序。它的格式为“%10d”(10位数字，没有数值的数位用0补充，例如“0000000001”)。



这样便会存在四种类型的znode节点，分别对应如下：

1、PERSISTENT-持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

2、PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3、EPHEMERAL-临时目录节点

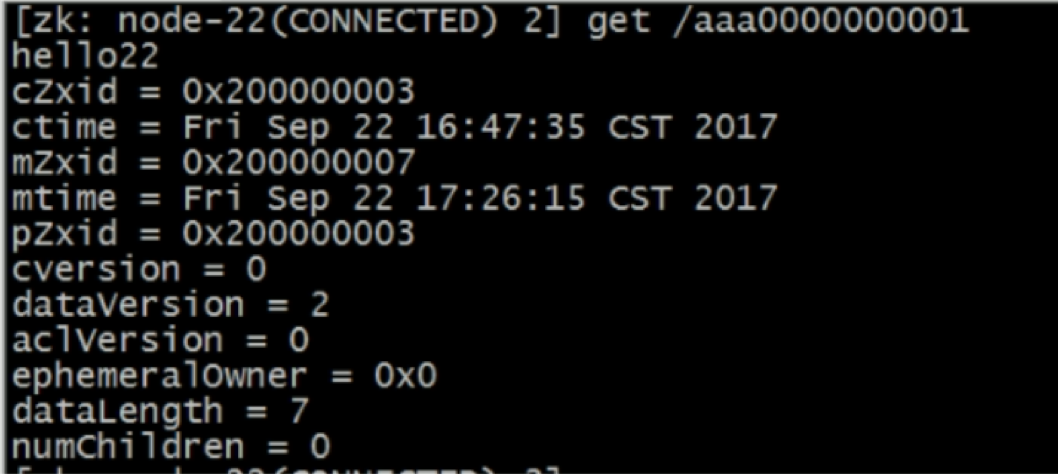
客户端与zookeeper断开连接后，该节点被删除

4、EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号。

**节点属性：**

每个znode都包含了一系列的属性，通过命令get，可以获得节点的属性。



dataVersion：数据版本号，每次对节点进行set操作，dataVersion的值都会增加1（即使设置的是相同的数据），可有效避免了数据更新时出现的先后顺序问题。

cversion:子节点的版本号，当znode的子节点有变化时，cversion的值就会增加1。也就是可以看到子节点的变化频率

aclVersion：ACL的版本号。只做了解

cZxid：znode创建的事务id，zk每次事务操作都会产生一个全局唯一的事务id，也就是zxid。

mZxid：znode被修改的事务id，即每次对znode的修改都会更新mZxid。

对于zk来说，每次的变化都会产生一个唯一的事务id—zxid，通过zxid可以确定更新操作的先后顺序。例如，如果zxid1小于zxid2，说明zxid1操作先于zxid2发生，zxid对于整个zk都是唯一的，即使操作的是不同的znode。

ctime:节点创建时的时间戳

mtime：节点最新一次更新发生时的时间戳。

ephemeralOwner:如果该节点为临时节点，ephemeralOwner值表示与该节点绑定的Session id，如果不是，ephemeralOwner值为0。也就是说该属性可以用于判断此节点是临时节点还是永久节点。为0是永久节点。

在client和server通信之前，首先需要建立连接，该连接称为Session。连接建立后，如果发生连接超时、授权失败或者显示关闭连接，连接便处于closed状态，此时Session结束。

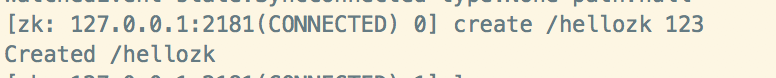
1. **zk常用命令**

**创建节点：**

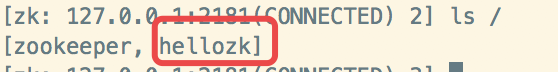
**创建永久节点**：此方法创建的是永久节点

create /hellozk 123

hellozk：是节点名；123是节点数据



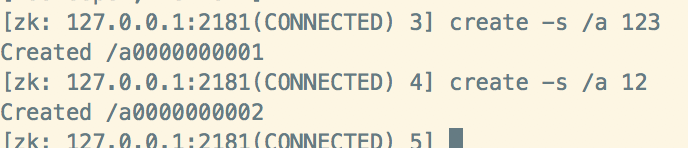
**查看创建的节点**：ls /

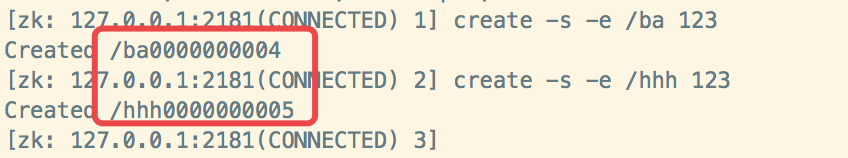


**创建临时节点**：create –e /zktemp 123456

当断开会话的时候，此节点会自动消失。但是该节点不会立即马上消失，而是断开连接后大概10秒左右才会删除。

创建序列化节点：create –s /a 123



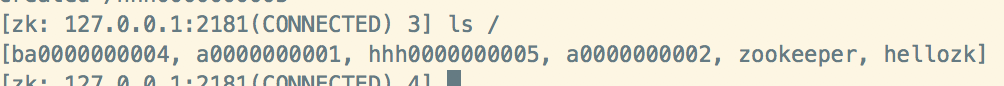


只要我们继续创建节点，不管该节点是否是一样的，序列化的末尾数字都会自增1。

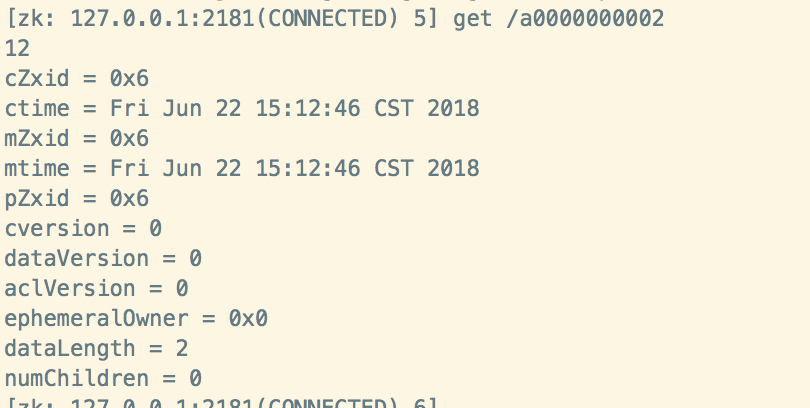
**读取节点：**

与读取相关命令有ls命令和get命令，ls命令可以列出zk指定节点下的所有子节点，只能查看指定节点下的第一级的所有子节点；get命令可以获取zk指定节点的数据内容和属性信息。

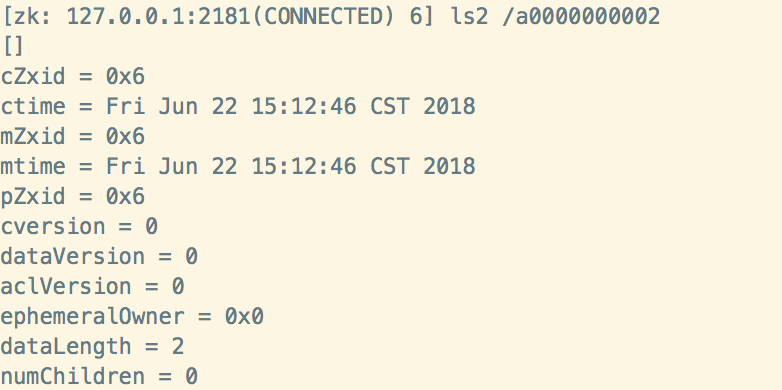
ls / 查看根目录下的所有节点



get / a0000000002

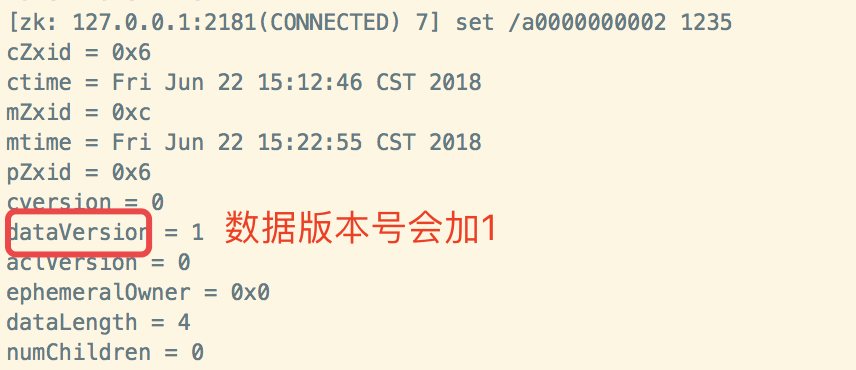


ls2 / a0000000002此命令和get命令得到的信息是差不多的。



**更新节点数据：**

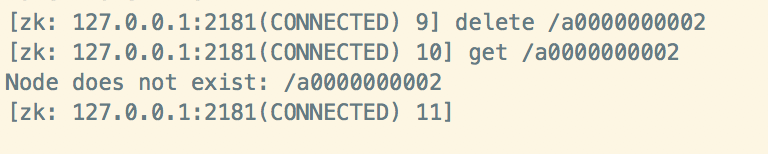
set / a0000000002 1235



**删除节点：**

如果你删除的节点存在子节点，不能直接删除，只能删除子节点后才能删除。也就是说zk只能删除子节点为空的节点。

delete / a0000000002



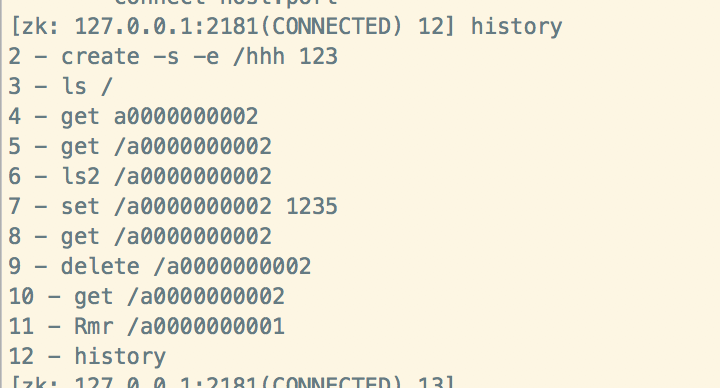
**quota命令：**



quota命令中对节点数目的限制，是不强制的，也就是超过该限制数量也不会报错，只会在zk日志文件中打印一条警告信息。

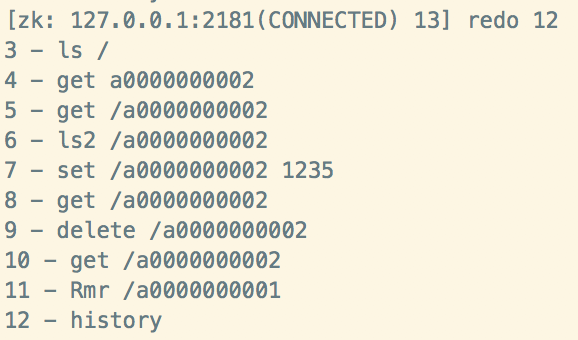
**其他命令：**

history命令：查看之前的所有操作命令

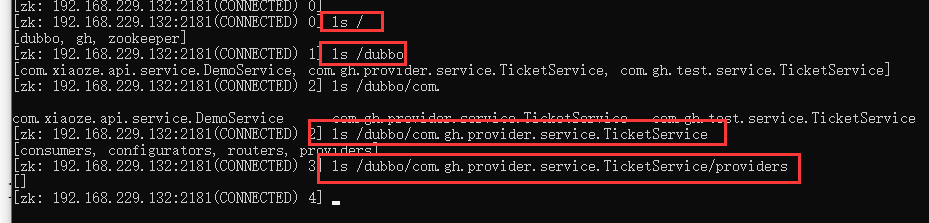


redo命令：可以看到每个命令前都有个编号，如果想再执行某条命令，可以使用redo

redo + 编号



**查看已注册服务**



服务正常的情况如下：



删除服务

rmr /dubbo/com.xiaoze.api.service.DemoService

1. **zk的watcher监听功能**

zk提供了分布式数据发布/订阅功能，一个典型的发布/订阅模型系统定义了一种一对多的订阅关系，能让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态变化时，会通知所有订阅者，使他们能够做出相应的处理。

Zk中，引入了watcher机制来实现这种分布式的通知功能，zk允许客户端向服务器端注册一个watcher监听，当服务器端的一些事件触发了这个watcher，那么就会向指定客户端发送一个事件通知来实现分布式的通知功能。

触发事件种类很多，如节点创建，节点删除，节点改变，子节点改变等。

总的来说可以概括watcher为以下三个过程：客户端向服务器端注册watcher、服务器端事件发生触发watcher、客户端回调watcher得到触发事件情况。

Watch机制的特点：

（1）一次性触发

事件发生触发监听，一个watcher event就会被发送到设置监听的客户端，这种效果是一次性的，后续再次发生同样的事件，不会再次触发。

（2）事件封装

zk使用watchedEvent对象来封装服务器端事件并传递，watchedEvent包含了每一个事件的三个基本属性：通知状态keepState,事件类型EventType和节点路径path。

（3）event异步发送

watcher的通知事件从服务器端发送到客户端是异步的

（4）先注册再触发

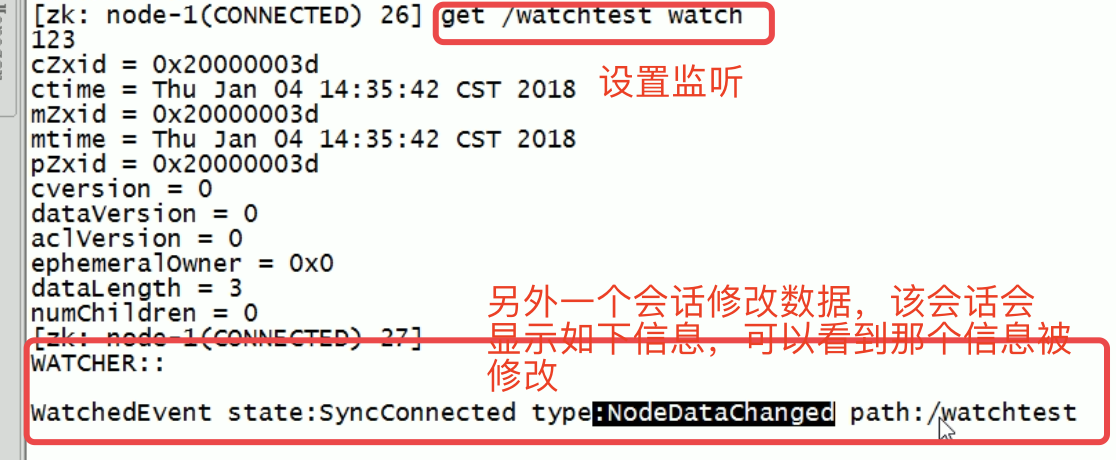
zk中的watch机制，必须客户端先去服务器端注册监听，这样事件发送才会触发监听，通知给客户端。

**通知状态和事件类型：**

其中连接状态事件（type=None,path=null）不需要客户端注册，客户端只要有需要直接处理就行了。



**设置监听：**

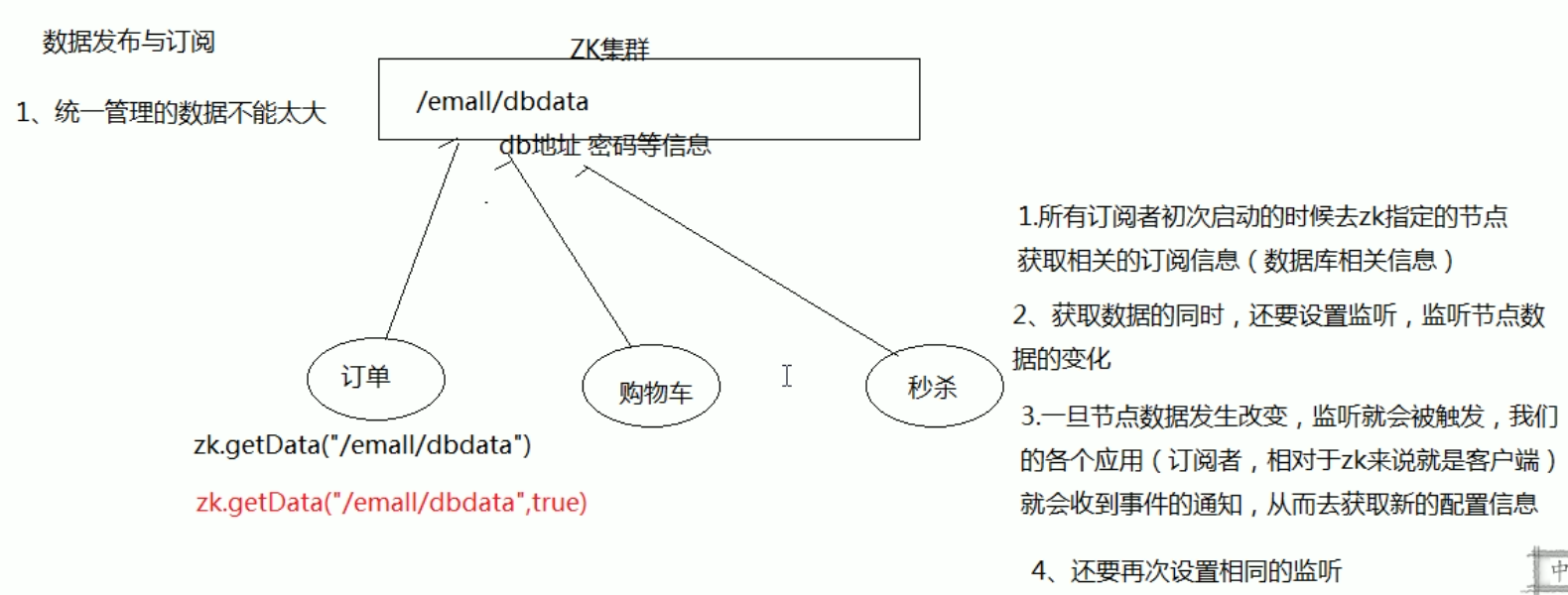


get /watchtest watch:get命令设置监听watchtest节点，如果数据改变，会显示改变的相关信息，但是只触发一次。

1. **数据发布和订阅**

**配置管理：**

描述：比如我们有很多个系统模块，每个系统启动都要去加载配置，以前的系统我们的实现方式是用的配置文件，系统启动时去加载配置文件，然后再进行其他操作（比如连接数据库，连接Redis等等）。在这里我们可以使用zk的数据发布与订阅功能，我们可以将配置信息放到zk的集群节点上，我们的系统去订阅监听该节点，当我们系统启动时，就去zk获取配置信息，并监听该节点信息，如果zk的该节点信息发生变动，我们的系统可以及时获取到更新后的配置信息，但是因为zk的监听机制只能触发一次，因此在我们获取到更新后的配置信息，还需要重新设置监听，这样才能保持连续的对配置信息的变更进行监听。



1. **aa**