# 安装

## 单机模式

### 下载地址

<http://mirror.bit.edu.cn/apache/zookeeper/>

本次安装选择的是：3.4.11版本

### 上传安装包

目录：

/export/data/zk

### 解压文件

chown admin:admin zookeeper-3.4.11.tar.gz

tar -zxvf zookeeper-3.4.11.tar.gz

### 配置文件

进入conf目录，拷贝zoo\_samle.cfg为zoo.cfg

cp zoo\_sample.cfg zoo.cfg

#### 详解

tickTime：ZK中的一个时间单元。ZK中所有时间都是以这个时间单元为基础，进行整数倍配置的。例如，session的最小超时时间是2\*tickTime。

clientPort：客户端连接server的端口，即对外服务端口，一般设置为2181吧

dataDir： 存储快照文件snapshot的目录。默认情况下，事务日志也会存储在这里。建议同时配置参数dataLogDir, 事务日志的写性能直接影响zk性能。（注意：一个配置文件只能包含一个dataDir字样，即使它被注释掉了。）

dataLogDir：事务日志输出目录。尽量给事务日志的输出配置单独的磁盘或是挂载点，这将极大的提升ZK性能。

globalOutstandingLimit ： 最大请求堆积数。默认是1000。ZK运行的时候， 尽管server已经没有空闲来处理更多的客户端请求了，但是还是允许客户端将请求提交到服务器上来，以提高吞吐性能。当然，为了防止Server内存溢出，这个请求堆积数还是需要限制下的。

(Java system property:zookeeper.globalOutstandingLimit.)

preAllocSize：预先开辟磁盘空间，用于后续写入事务日志。默认是64M，每个事务日志大小就是64M。如果ZK的快照频率较大的话，建议适当减小这个参数。(Java system property:zookeeper.preAllocSize)

snapCount：每进行snapCount次事务日志输出后，触发一次快照(snapshot), 此时，ZK会生成一个snapshot.\*文件，同时创建一个新的事务日志文件log.\*。默认是100000.（真正的代码实现中，会进行一定的随机数处理，以避免所有服务器在同一时间进行快照而影响性能）(Java system property:zookeeper.snapCount)

traceFile：用于记录所有请求的log，一般调试过程中可以使用，但是生产环境不建议使用，会严重影响性能。(Java system property:? requestTraceFile)

maxClientCnxns：单个客户端与单台服务器之间的连接数的限制，是ip级别的，默认是60，如果设置为0，那么表明不作任何限制。请注意这个限制的使用范围，仅仅是单台客户端机器与单台ZK服务器之间的连接数限制，不是针对指定客户端IP，也不是ZK集群的连接数限制，也不是单台ZK对所有客户端的连接数限制。指定客户端IP的限制策略，这里有一个patch，可以尝试一下：http://rdc.taobao.com/team/jm/archives/1334（No Java system property）

clientPortAddress：对于多网卡的机器，可以为每个IP指定不同的监听端口。默认情况是所有IP都监听 clientPort指定的端口。 New in 3.3.0

minSessionTimeoutmaxSessionTimeout：Session超时时间限制，如果客户端设置的超时时间不在这个范围，那么会被强制设置为最大或最小时间。默认的Session超时时间是在2 \* tickTime ~ 20 \* tickTime 这个范围 New in 3.3.0

fsync.warningthresholdms： 事务日志输出时，如果调用fsync方法超过指定的超时时间，那么会在日志中输出警告信息。默认是1000ms。(Java system property: fsync.warningthresholdms)New in 3.3.4

autopurge.purgeInterval：在上文中已经提到，3.4.0及之后版本，ZK提供了自动清理事务日志和快照文件的功能，这个参数指定了清理频率，单位是小时，需要配置一个1或更大的整数，默认是0，表示不开启自动清理功能。(No Java system property) New in 3.4.0

autopurge.snapRetainCount：这个参数和上面的参数搭配使用，这个参数指定了需要保留的文件数目。默认是保留3个。(No Java system property) New in 3.4.0

initLimit：Follower在启动过程中，会从Leader同步所有最新数据，然后确定自己能够对外服务的起始状态。Leader允许F在initLimit时间内完成这个工作。通常情况下，我们不用太在意这个参数的设置。如果ZK集群的数据量确实很大了，F在启动的时候，从Leader上同步数据的时间也会相应变长，因此在这种情况下，有必要适当调大这个参数了。(No Java system property)

syncLimit：在运行过程中，Leader负责与ZK集群中所有机器进行通信，例如通过一些心跳检测机制，来检测机器的存活状态。如果L发出心跳包在syncLimit之后，还没有从F那里收到响应，那么就认为这个F已经不在线了。注意：不要把这个参数设置得过大，否则可能会掩盖一些问题。(No Java system property)

leaderServes：默认情况下，Leader是会接受客户端连接，并提供正常的读写服务。但是，如果你想让Leader专注于集群中机器的协调，那么可以将这个参数设置为no，这样一来，会大大提高写操作的性能。(Java system property: zookeeper.leaderServes)。

server.x=[hostname]:nnnnn[:nnnnn]：这里的x是一个数字，与myid文件中的id是一致的。右边可以配置两个端口，第一个端口用于F和L之间的数据同步和其它通信，第二个端口用于Leader选举过程中投票通信。

(No Java system property)

group.x=nnnnn[:nnnnn]weight.x=nnnnn：对机器分组和权重设置

group.1=1:2:3

group.2=4:5:6

group.3=7:8:9

weight.1=1

weight.2=1

weight.3=1

weight.4=1

weight.5=1

weight.6=1

weight.7=1

weight.8=1

weight.9=1

参考：<http://zookeeper.apache.org/doc/r3.4.3/zookeeperHierarchicalQuorums.html>

cnxTimeout：Leader选举过程中，打开一次连接的超时时间，默认是5s。(Java system property: zookeeper. cnxTimeout)

zookeeper.DigestAuthenticationProvider

.superDigest：ZK权限设置相关

skipACL：对所有客户端请求都不作ACL检查。如果之前节点上设置有权限限制，一旦服务器上打开这个开头，那么也将失效。(Java system property: zookeeper.skipACL)

forceSync：这个参数确定了是否需要在事务日志提交的时候调用FileChannel.force来保证数据完全同步到磁盘。(Java system property: zookeeper.forceSync)

jute.maxbuffer：每个节点最大数据量，是默认是1M。这个限制必须在server和client端都进行设置才会生效。(Java system property: jute.maxbuffer)

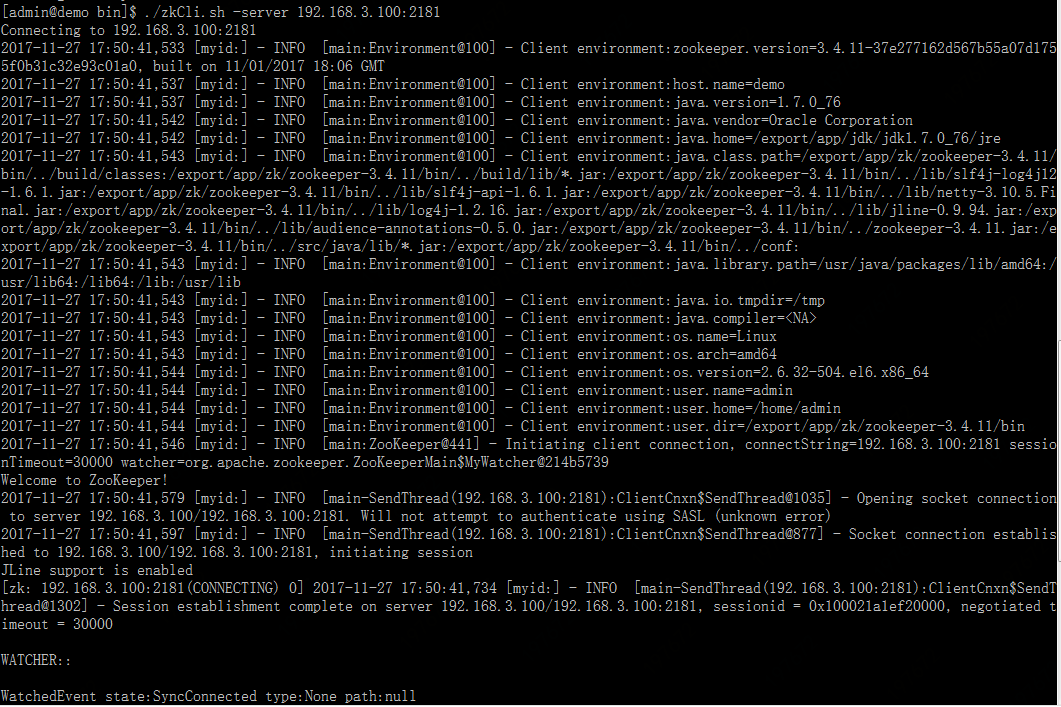
### 启动

在bin目录下：./zkServer.sh start

停止：./zkServer.sh stop

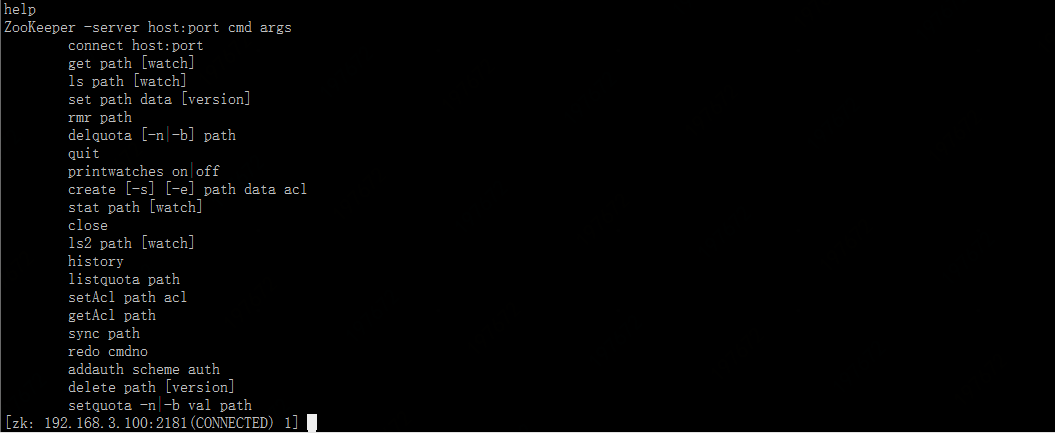
查看服务状态：./zkServer.sh status

连接：./zkCli.sh -server 192.168.0.1:2181



### 验证

输入help可查看所有命令



## 集群模式

## 命令详解

### Connect命令

连接zk服务端，与close命令配合使用可以连接或者断开zk服务端。

如connect 127.0.0.1:2181

### get 命令

获取节点信息，注意节点的路径皆为绝对路径，也就是说必要要从/（根路径）开始。

如get /

hello world

cZxid = 0x0

ctime = Thu Jan 01 08:00:00 CST 1970

mZxid = 0x5

mtime = Thu Apr 27 15:09:00 CST 2017

pZxid = 0xc

cversion = 1

dataVersion = 2

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 11

numChildren = 1

详解：

hello world为节点数据信息

cZxid节点创建时的zxid

ctime节点创建时间

mZxid节点最近一次更新时的zxid

mtime节点最近一次更新的时间

cversion子节点数据更新次数

dataVersion本节点数据更新次数

aclVersion节点ACL(授权信息)的更新次数

ephemeralOwner如果该节点为临时节点,ephemeralOwner值表示与该节点绑定的session id. 如果该节点不是临时节点,ephemeralOwner值为0

dataLength节点数据长度，本例中为hello world的长度

numChildren子节点个数

### ls命令

获取路径下的节点信息，注意此路径为绝对路径，类似于linux的ls命令。

如ls /zookeeper

### Set命令

设置节点的数据。

如set /zookeeper "hello world"

### rmr 命令

删除节点命令，此命令与delete命令不同的是delete不可删除有子节点的节点，但是rmr命令可以删除，注意路径为绝对路径。

如rmr /zookeeper/znode

### delquota命令

删除配额，-n为子节点个数，-b为节点数据长度。

如delquota –n 2，请参见listquota和setquota命令。

### quit命令

退出

### printwatches命令

设置和显示监视状态，on或者off。

如printwatches on

### create命令

create [-s] [-e] path data acl

创建节点，其中-s为顺序充点，-e临时节点。

如create /zookeeper/node1"test\_create" world:anyone:cdrwa

其中acl处，请参见getAcl和setAcl命令。

### stat命令

查看节点状态信息。如stat /

cZxid = 0x0

ctime = Thu Jan 01 08:00:00 CST 1970

mZxid = 0x1f

mtime = Thu Apr 27 16:05:14 CST 2017

pZxid = 0xc

cversion = 1

dataVersion = 3

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 5

numChildren = 1

与get命令大体相同，请参见get命令。

### close命令

断开客户端与服务端的连接。

### ls2命令

ls2为ls命令的扩展，比ls命令多输出本节点信息。

如 ls2 /zookeeper

### history命令

列出最近的历史命令。

如history

0 - ls /

1 - ls /

2 - ls2 /

3 - history

4 - listquota /zookeeper

5 – history

基本格式为：命令ID-命令，可以与redo命令配合使用。

### listquota命令

显示配额。

如listquota /zookeeper

absolute path is/zookeeper/quota/zookeeper/zookeeper\_limits

Output quota for /zookeepercount=2,bytes=-1

解释：

/zookeeper节点个数限额为2，长度无限额。

### setAcl命令

设置节点Acl。

此处重点说一下acl，acl由大部分组成：1为scheme，2为user，3为permission，一般情况下表示为scheme:id:permissions。

其中scheme和id是相关的，下面将scheme和id一起说明。

#### scheme和id

world: 它下面只有一个id, 叫anyone, world:anyone代表任何人，zookeeper中对所有人有权限的结点就是属于world:anyone的

auth: 它不需要id, 只要是通过authentication的user都有权限（zookeeper支持通过kerberos来进行authencation, 也支持username/password形式的authentication)

digest: 它对应的id为username:BASE64(SHA1(password))，它需要先通过username:password形式的authentication

ip: 它对应的id为客户机的IP地址，设置的时候可以设置一个ip段，比如ip:192.168.1.0/16, 表示匹配前16个bit的IP段

super: 在这种scheme情况下，对应的id拥有超级权限，可以做任何事情(cdrwa)

#### permissions

CREATE(c): 创建权限，可以在在当前node下创建child node

DELETE(d): 删除权限，可以删除当前的node

READ(r): 读权限，可以获取当前node的数据，可以list当前node所有的child nodes

WRITE(w): 写权限，可以向当前node写数据

ADMIN(a): 管理权限，可以设置当前node的permission

综上，一个简单使用setAcl命令，则可以为：

setAcl /zookeeper/node1 world:anyone:cdrw

### getAcl命令

获取节点Acl。

如getAcl /zookeeper/node1

'world,'anyone

: cdrwa

注：可参见setAcl命令。

### sync命令

强制同步。

如sync /zookeeper

由于请求在半数以上的zk server上生效就表示此请求生效，那么就会有一些zk server上的数据是旧的。sync命令就是强制同步所有的更新操作。

### redo命令

再次执行某命令。

如redo 10

其中10为命令ID，需与history配合使用。

### addauth命令

节点认证。

如addauth digest username:password，可参见setAcl命令digest处。

使用方法：

一、通过setAcl设置用户名和密码

setAcl pathdigest:username:base64(sha1(password)):crwda

二、认证

addauth digest username:password

### delete命令

删除节点。

如delete /zknode1

### setquota命令

设置子节点个数和数据长度配额。

如setquota –n 4 /zookeeper/node 设置/zookeeper/node子节点个数最大为4

setquota –b 100 /zookeeper/node 设置/zookeeper/node节点长度最大为100

# 知识点

## 三种角色节点

分别是Leader（领导者）、Follower（跟随者）、Observer（观察者）。

Leader 负责更新系统状态，进行投票（选举leader）的发起和决议。

Follower 用于接收客户端请求并向客户端返回处理请求的结果，在选举过程中参与投票

Observer 可以接收客户端的连接，将写请求转发给Leader系诶的那，但Observer不参加投票过程，只同步Leader的状态。Observer的目的是为了扩展系统，提高读取速度。

## 四种节点状态

zookeeper节点有四种状态，Looking、Following、Leading、Observing

Looking：寻找Leader状态，当Server处于该状态时，此Server会认为当前集群中没有Leader，需要进入Leader选举状态。

Following： 跟随者状态，表明该Server角色为Follower。

Leading： 领导者状态，表明当前服务器角色是Leader。

Observing： 观察者状态，表明当前服务器角色是Observer。

## 节点类型

ZooKeeper 节点是有生命周期的，这取决于节点的类型。在 ZooKeeper 中，节点类型可以分为持久节点（PERSISTENT ）、临时节点（EPHEMERAL），以及时序节点（SEQUENTIAL ），具体在节点创建过程中，一般是组合使用，可以生成以下 4 种节点类型。

### 持久节点（PERSISTENT）

所谓持久节点，是指在节点创建后，就一直存在，直到有删除操作来主动清除这个节点——不会因为创建该节点的客户端会话失效而消失。

### 持久顺序节点（PERSISTENT\_SEQUENTIAL）

这类节点的基本特性和上面的节点类型是一致的。额外的特性是，在ZK中，每个父节点会为他的第一级子节点维护一份时序，会记录每个子节点创建的先后顺序。基于这个特性，在创建子节点的时候，可以设置这个属性，那么在创建节点过程中，ZK会自动为给定节点名加上一个数字后缀，作为新的节点名。这个数字后缀的范围是整型的最大值。

### 临时节点（EPHEMERAL）

和持久节点不同的是，临时节点的生命周期和客户端会话绑定。也就是说，如果客户端会话失效，那么这个节点就会自动被清除掉。注意，这里提到的是会话失效，而非连接断开。另外，在临时节点下面不能创建子节点。

这里还要注意一件事，就是当你客户端会话失效后，所产生的节点也不是一下子就消失了，也要过一段时间，大概是10秒以内，可以试一下，本机操作生成节点，在服务器端用命令来查看当前的节点数目，你会发现客户端已经stop，但是产生的节点还在。

### 临时顺序节点（EPHEMERAL\_SEQUENTIAL）

可以用来实现分布式锁：

①客户端调用create()方法创建名为“\_locknode\_/guid-lock-”的节点，需要注意的是，这里节点的创建类型需要设置为EPHEMERAL\_SEQUENTIAL。

②客户端调用getChildren(“\_locknode\_”)方法来获取所有已经创建的子节点，注意，这里不注册任何Watcher。

③客户端获取到所有子节点path之后，如果发现自己在步骤1中创建的节点序号最小，那么就认为这个客户端获得了锁。

如果在步骤3中发现自己并非所有子节点中最小的，说明自己还没有获取到锁。此时客户端需要找到比自己小的那个节点，然后对其调用exist()方法，同时注册事件监听。

之后当这个被关注的节点被移除了，客户端会收到相应的通知。这个时候客户端需要再次调用getChildren(“\_locknode\_”)方法来获取所有已经创建的子节点，确保自己确实是最小的节点了，然后进入步骤3。

## zookeeper的选举机制

为什么要进行Leader选举？

Leader主要作用是保证分布式数据一致性，即每个节点的存储的数据同步。遇到以下两种情况需要进行Leader选举

1）服务器初始化启动

2）服务器运行期间无法和Leader保持连接，Leader节点崩溃，逻辑时钟崩溃。

1.服务器初始化时Leader选举

zookeeper由于其自身的性质，一般建议选取奇数个节点进行搭建分布式服务器集群。以3个节点组成的服务器集群为例，说明服务器初始化时的选举过程。启动第一台安装zookeeper的节点时，无法单独进行选举，启动第二台时，两节点之间进行通信，开始选举Leader。

1）每个Server投出一票。他们两都选自己为Leader，投票的内容为（SID，ZXID）。SID即Server的id，安装zookeeper时配置文件中所配置的myid；ZXID，事务id，为节点的更新程度，ZXID越大，代表Server对Znode的操作越新。由于服务器初始化，每个Sever上的Znode为0，所以Server1投的票为（1,0），Server2为（2,0）。两Server将各自投票发给集群中其他机器。

2）每个Server接收来自其他Server的投票。集群中的每个Server先判断投票有效性，如检查是不是本轮的投票，是不是来Looking状态的服务器投的票。

3）对投票结果进行处理。先了解下处理规则

- 首先对比ZXID。ZXID大的服务器优先作为Leader

- 若ZXID相同，比如初始化的时候，每个Server的ZXID都为0，就会比较myid，myid大的选出来做Leader。

对于Server而言，他接受到的投票为（2,0），因为自身的票为（1,0），所以此时它会选举Server2为Leader，将自己的更新为（2,0）。而Server2收到的投票为Server1的（1,0）由于比他自己小，Server2的投票不变。Server1和Server2再次将票投出，投出的票都为（2,0）。

4） 统计投票。每次投票之后，服务器都会统计投票信息，如果判定某个Server有过半的票数投它，那么该Server将会作为Leader。对于Server1和Server2而言,统计出已经有两台机器接收了（2,0）的投票信息，此时认为选出了Leader。

5）改变服务器状态。当确定了Leader之后，每个Server更新自己的状态，Leader将状态更新为Leading，Follower将状态更新为Following。

2.服务器运行期间的Leader选举

zookeeper运行期间，如果有新的Server加入，或者非Leader的Server宕机，那么Leader将会同步数据到新Server或者寻找其他备用Server替代宕机的Server。若Leader宕机，此时集群暂停对外服务，开始在内部选举新的Leader。假设当前集群中有Server1、Server2、Server3三台服务器，Server2为当前集群的Leader，由于意外情况，Server2宕机了，便开始进入选举状态。过程如下

1） 变更状态。其他的非Observer服务器将自己的状态改变为Looking，开始进入Leader选举。

2） 每个Server发出一个投票（myid，ZXID），由于此集群已经运行过，所以每个Server上的ZXID可能不同。假设Server1的ZXID为145，Server3的为122，第一轮投票中，Server1和Server3都投自己，票分别为（1，145）、（3,122）,将自己的票发送给集群中所有机器。

3） 每个Server接收接收来自其他Server的投票，降下来的步骤与启动时步骤相同。

## zookeeper集群中节点配置个数是奇数个？

我们都知道，Zookeeper要安装在奇数个节点，但是为什么？

Zookeeper的大部分操作都是通过选举产生的。比如，标记一个写是否成功是要在超过一半节点发送写请求成功时才认为有效。同样，Zookeeper选择领导者节点也是在超过一半节点同意时才有效。最后，Zookeeper是否正常是要根据是否超过一半的节点正常才算正常。这是基于CAP的一致性原理。

zookeeper有这样一个特性：集群中只要有过半的机器是正常工作的，那么整个集群对外就是可用的。也就是说如果有2个zookeeper，那么只要有1个死了zookeeper就不能用了，因为1没有过半，所以2个zookeeper的死亡容忍度为0；同理，要是有3个zookeeper，一个死了，还剩下2个正常的，过半了，所以3个zookeeper的容忍度为1；同理你多列举几个：2->0;3->1;4->1;5->2;6->2会发现一个规律，2n和2n-1的容忍度是一样的，都是n-1，所以为了更加高效，何必增加那一个不必要的zookeeper呢。

## 分布式锁

### 利用节点名称的唯一性来实现共享锁

ZooKeeper抽象出来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。例如：我们在Zookeeper目录/test目录下创建，两个客户端创建一个名为Lock节点，只有一个能够成功。

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/test/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。

该共享锁实现很符合我们通常多个线程去竞争锁的概念，利用节点名称唯一性的做法简明、可靠。由上述算法容易看出，由于客户端会同时收到/test/Lock被删除的通知，重新进入竞争创建节点，故存在"惊群现象"。

总结 这种方案的正确性和可靠性是ZooKeeper机制保证的，实现简单。缺点是会产生“惊群”效应，假如许多客户端在等待一把锁，当锁释放时候所有客户端都被唤醒，仅仅有一个客户端得到锁。

### 利用临时顺序节点实现共享锁的一般做法

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（应该是会话失效吧？），则该节点自动被删除。

利用上面这两个特性，我们来看下获取实现分布式锁的基本逻辑：

* 客户端调用create()方法创建名为“*locknode*/guid-lock-”的节点，需要注意的是，这里节点的创建类型需要设置为EPHEMERAL\_SEQUENTIAL。
* 客户端调用getChildren(“*locknode*”)方法来获取所有已经创建的子节点，同时在这个节点上注册上子节点变更通知的Watcher。
* 客户端获取到所有子节点path之后，如果发现自己在步骤1中创建的节点是所有节点中序号最小的，那么就认为这个客户端获得了锁。
* 如果在步骤3中发现自己并非是所有子节点中最小的，说明自己还没有获取到锁，就开始等待，直到下次子节点变更通知的时候，再进行子节点的获取，判断是否获取锁。

释放锁的过程相对比较简单，就是删除自己创建的那个子节点即可。

上面这个分布式锁的实现中，大体能够满足了一般的分布式集群竞争锁的需求。这里说的一般性场景是指集群规模不大，一般在10台机器以内。

不过，细想上面的实现逻辑，我们很容易会发现一个问题，步骤4，“即获取所有的子点，判断自己创建的节点是否已经是序号最小的节点”，这个过程，在整个分布式锁的竞争过程中，大量重复运行，并且绝大多数的运行结果都是判断出自己并非是序号最小的节点，从而继续等待下一次通知——这个显然看起来不怎么科学。客户端无端的接受到过多的和自己不相关的事件通知，这如果在集群规模大的时候，会对Server造成很大的性能影响，并且如果一旦同一时间有多个节点的客户端断开连接，这个时候，服务器就会像其余客户端发送大量的事件通知——这就是所谓的惊群效应。而这个问题的根源在于，没有找准客户端真正的关注点。

我们再来回顾一下上面的分布式锁竞争过程，它的核心逻辑在于：判断自己是否是所有节点中序号最小的。于是，很容易可以联想的到的是，每个节点的创建者只需要关注比自己序号小的那个节点。

### 利用临时顺序节点实现共享锁的改进实现

下面是改进后的分布式锁实现，和之前的实现方式唯一不同之处在于，这里设计成每个锁竞争者，只需要关注”locknode”节点下序号比自己小的那个节点是否存在即可。

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（比自己创建的节点小的最大节点），进入等待。

对于解锁操作，只需要将自身创建的节点删除即可。

该算法只监控比自身创建节点序列号小(比自己小的最大的节点)的节点，在当前获得锁的节点释放锁的时候没有“惊群”。

总结 利用临时顺序节点来实现分布式锁机制其实就是一种按照创建顺序排队的实现。这种方案效率高，避免了“惊群”效应，多个客户端共同等待锁，当锁释放时只有一个客户端会被唤醒。

### 使用menagerie

menagerie基于Zookeeper实现了java.util.concurrent包的一个分布式版本。这个封装是更大粒度上对各种分布式一致性使用场景的抽象。其中最基础和常用的是一个分布式锁的实现： org.menagerie.locks.ReentrantZkLock，通过ZooKeeper的全局有序的特性和EPHEMERAL\_SEQUENTIAL类型znode的支持，实现了分布式锁。具体做法是：不同的client上每个试图获得锁的线程，都在相同的basepath下面创建一个EPHEMERAL\_SEQUENTIAL的node。EPHEMERAL表示要创建的是临时znode，创建连接断开时会自动删除； SEQUENTIAL表示要自动在传入的path后面缀上一个自增的全局唯一后缀,作为最终的path。因此对不同的请求ZK会生成不同的后缀，并分别返回带了各自后缀的path给各个请求。因为ZK全局有序的特性，不管client请求怎样先后到达，在ZKServer端都会最终排好一个顺序，因此自增后缀最小的那个子节点，就对应第一个到达ZK的有效请求。然后client读取basepath下的所有子节点和ZK返回给自己的path进行比较，当发现自己创建的sequential node的后缀序号排在第一个时，就认为自己获得了锁；否则的话，就认为自己没有获得锁。这时肯定是有其他并发的并且是没有断开的client/线程先创建了node。

### InterProcessMutex

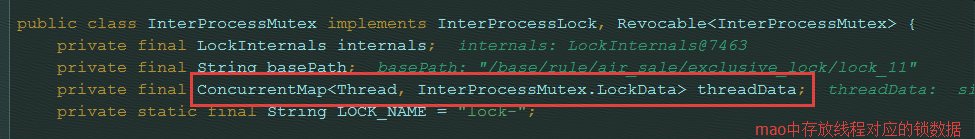
Jar包

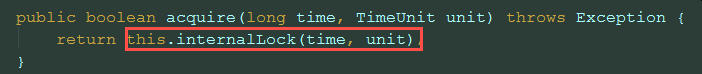
<dependency>  
 <groupId>org.apache.curator</groupId>  
 <artifactId>curator-recipes</artifactId>

<version>2.7.1-SNAPSHOT</version>  
</dependency>

类：org.apache.curator.framework.recipes.locks.InterProcessMutex

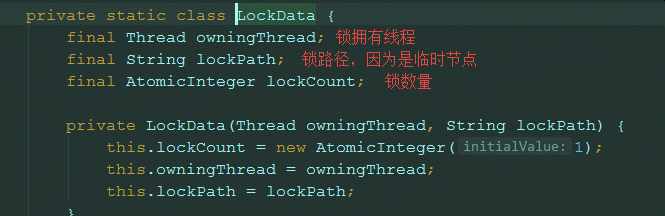
调用此类的acquire方法







InterProcessMutex的内部类LockData



# 脑裂

# 客户端

## Curator

CuratorFramework:

package com.jd.airplane.infra.zookeeper;  
import com.jd.airplane.infra.common.util.AssertUtils;  
import com.jd.airplane.infra.zookeeper.listener.ZKListener;  
import org.apache.commons.collections.CollectionUtils;  
import org.apache.curator.RetryPolicy;  
import org.apache.curator.framework.CuratorFramework;  
import org.apache.curator.framework.CuratorFrameworkFactory;  
import org.apache.curator.framework.api.UnhandledErrorListener;  
import org.apache.curator.framework.state.ConnectionState;  
import org.apache.curator.framework.state.ConnectionStateListener;  
import org.apache.curator.retry.ExponentialBackoffRetry;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.factory.FactoryBean;  
import org.springframework.beans.factory.InitializingBean;  
  
import java.util.List;  
  
*/\*\* zookeeper工厂bean  
 \* Created by yangyanping on 2018/3/19.  
 \*/*public class ZookeeperFactoryBean implements FactoryBean<CuratorFramework>, InitializingBean {  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(ZookeeperFactoryBean.class);  
  
 private int sessionTimeoutMs = 20000;  
  
 private int connectionTimeoutMs = 20000;  
 */\*\*  
 \* 1000 是重试间隔时间基数  
 \*/* private int baseSleepTimeMs = 1000;  
  
 */\*\*  
 \* 重试次数  
 \*/* private int maxSleepMs = Integer.*MAX\_VALUE*;  
  
 private String namespace = NodeConstant.*FLIGHT\_NAMESPACE*;  
  
 private String connectString;  
 private CuratorFramework zkClient;  
 private List<ZKListener> listeners;  
  
 public CuratorFramework getObject() throws Exception {  
 if (this.zkClient == null) {  
 afterPropertiesSet();  
 }  
  
 return zkClient;  
 }  
  
 public Class<? extends CuratorFramework> getObjectType() {  
 return this.zkClient == null ? CuratorFramework.class : zkClient.getClass();  
 }  
  
 public boolean isSingleton() {  
 return true;  
 }  
  
 public void afterPropertiesSet() throws Exception {  
 AssertUtils.*notNull*(connectString, "Property 'connectString' is required");  
 AssertUtils.*notNull*(namespace, "Property 'namespace' is required");  
  
 zkClient = buildSqlSessionFactory();  
 registerListeners(zkClient);  
  
 zkClient.start();  
 }  
  
 private CuratorFramework buildSqlSessionFactory() {  
 RetryPolicy retryPolicy = new ExponentialBackoffRetry(baseSleepTimeMs, maxSleepMs);  
  
 CuratorFramework client = CuratorFrameworkFactory.*builder*()  
 .connectString(connectString)  
 .sessionTimeoutMs(sessionTimeoutMs)  
 .connectionTimeoutMs(connectionTimeoutMs)  
 .retryPolicy(retryPolicy)  
 .namespace(namespace)  
 .build();  
  
 return client;  
 }  
  
 */\*\*  
 \* 注册需要监听的监听者对像.  
 \*/* private void registerListeners(CuratorFramework client) {  
 client.getConnectionStateListenable().addListener(new ConnectionStateListener() {  
 public void stateChanged(CuratorFramework client, ConnectionState newState) {  
 *LOGGER*.info("CuratorFramework state changed: {}", newState);  
  
 if (CollectionUtils.*isEmpty*(listeners)) {  
 return;  
 }  
  
 for (ZKListener listener : listeners) {  
 listener.executor(client, newState);  
 *LOGGER*.info("Listener {} executed!", listener.getClass().getName());  
 }  
 }  
 });  
  
 client.getUnhandledErrorListenable().addListener(new UnhandledErrorListener() {  
 public void unhandledError(String message, Throwable e) {  
 *LOGGER*.info("CuratorFramework unhandledError: {}", message);  
 }  
 });  
 }

CuratorFrameworkImpl的start方法

public void start() {  
 this.log.info("Starting");  
 if(!this.state.compareAndSet(CuratorFrameworkState.LATENT, CuratorFrameworkState.STARTED)) {  
 IllegalStateException error = new IllegalStateException();  
 this.log.error("Cannot be started more than once", error);  
 throw error;  
 } else {  
 try {  
 this.connectionStateManager.start();  
 ConnectionStateListener listener = new ConnectionStateListener() {  
 public void stateChanged(CuratorFramework client, ConnectionState newState) {  
 if(ConnectionState.CONNECTED == newState || ConnectionState.RECONNECTED == newState) {  
 CuratorFrameworkImpl.this.logAsErrorConnectionErrors.set(true);  
 }  
  
 }  
 };  
 this.getConnectionStateListenable().addListener(listener);  
 this.client.start();  
 this.executorService = Executors.newFixedThreadPool(2, this.threadFactory);  
 this.executorService.submit(new Callable<Object>() {  
 public Object call() throws Exception {  
 CuratorFrameworkImpl.this.backgroundOperationsLoop();  
 return null;  
 }  
 });  
 } catch (Exception var2) {  
 this.handleBackgroundOperationException((OperationAndData)null, var2);  
 }  
  
 }  
}

# 为什么不用ZK来做服务发现？

[为什么不用zk做服务发现](https://blog.csdn.net/whereismatrix/article/details/53305045)

[Eureka简介](https://www.cnblogs.com/wangdaijun/p/6851027.html)

# Raft算法

[漫画：什么是拜占庭将军问题？](https://mp.weixin.qq.com/s/-dTBkwVaRo6WxZ8uS9DwjQ)

[Raft算法（zookeeper核心算法）](https://blog.csdn.net/lxlmycsdnfree/article/details/78984752)

# 项目中使用

## 服务发现：

Air-base-background中用来做服务发现。当有一个机器加入进来（成为节点的子节点，会新增临时序列节点）时候，会获取最小的子节点，并将此节点作为worker执行的机器

