# 什么是zookeeper?

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，ZooKeeper是以[Fast Paxos](http://rdc.taobao.com/blog/cs/?p=261)[**算法**](http://lib.csdn.net/base/datastructure)为基础，实现同步服务，配置维护和命名服务等分布式应用。

# 为什么要用zookeeper?

2006年的时候Google出了Chubby来解决分布一致性的问题(distributed consensus problem)，所有集群中的服务器通过Chubby最终选出一个Master Server ，最后这个Master Server来协调工作。简单来说其原理就是：在一个分布式系统中，有一组服务器在运行同样的程序，它们需要确定一个Value，以那个服务器提供的信息为主/为准，当这个服务器经过n/2+1的方式被选出来后，所有的机器上的Process都会被通知到这个服务器就是主服务器 Master服务器，大家以他提供的信息为准。很想知道Google Chubby中的奥妙，可惜人家Google不开源，自家用。  
但是在2009年3年以后沉默已久的Yahoo在Apache上推出了类似的产品ZooKeeper，并且在Google原有Chubby的[设计思想](https://www.baidu.com/s?wd=%E8%AE%BE%E8%AE%A1%E6%80%9D%E6%83%B3&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9rARsPyF9rj9huHu9rADz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En10dPHTkrj04nWb1PHczrjfY)上做了一些改进，因为ZooKeeper并不是完全遵循Paxos协议，而是基于自身设计并优化的一个2 phase commit的协议

# Zookeeper可以做什么?

[分布式锁](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4mW6drHw9mHbsuhFBPvRk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4nWmzrH6sPjRzP1nsPjD4)

我们就可以利用Zookeeper来协调多个分布式进程之间的活动。比如在一个分布式环境中，为了提高可靠性，我们的集群的每台服务器上都部署着同样的服务。但是，一件事情如果集群中的每个服务器都进行的话，那相互之间就要协调，编程起来将非常复杂。而如果我们只让一个服务进行操作，那又存在单点。通常还有一种做法就是使用[分布式锁](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4mW6drHw9mHbsuhFBPvRk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4nWmzrH6sPjRzP1nsPjD4)，在某个时刻只让一个服务去干活，比如HBase的Master就是采用这种机制。但要注意的是[分布式锁](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4mW6drHw9mHbsuhFBPvRk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4nWmzrH6sPjRzP1nsPjD4)跟同一个进程的锁还是有区别的，所以使用的时候要比同一个进程里的锁更谨慎的使用。

配置管理  
在我们的应用中除了代码外，还有一些就是各种配置。比如数据库连接等。一般我们都是使用配置文件的方式，在代码中引入这些配置文件。但是当我们只有一种配置，只有一台服务器，并且不经常修改的时候，使用配置文件是一个很好的做法，但是如果我们配置非常多，有很多服务器都需要这个配置，而且还可能是动态的话使用配置文件就不是个好主意了。这个时候往往需要寻找一种集中管理配置的方法，我们在这个集中的地方修改了配置，所有对这个配置感兴趣的都可以获得变更。比如我们可以把配置放在数据库里，然后所有需要配置的服务都去这个数据库读取配置。但是，因为很多服务的正常运行都非常依赖这个配置，所以需要这个集中提供配置服务的服务具备很高的可靠性。一般我们可以用一个集群来提供这个配置服务，但是用集群提升可靠性，那如何保证配置在集群中的一致性呢？ 这个时候就需要使用一种实现了一致性协议的服务了。Zookeeper就是这种服务，它使用Zab这种一致性协议来提供一致性。现在有很多开源项目使用Zookeeper来维护配置，比如在HBase中，客户端就是连接一个Zookeeper，获得必要的HBase集群的配置信息，然后才可以进一步操作。还有在开源的消息队列Kafka中，也使用Zookeeper来维护broker的信息。在Alibaba开源的SOA框架Dubbo中也广泛的使用Zookeeper管理一些配置来实现服务治理。

命名服务  
是指通过指定的名字来获取资源或者服务的地址，提供者的信息。可以把系统中的各种服务的名称、地址、以及目录信息存放在Zookeeper中

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。在Dubbo实现中：

服务提供者在启动的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。

服务消费者启动的时候，订阅/dubbo/{serviceName}/providers目录下的提供者URL地址， 并向/dubbo/{serviceName} /consumers目录下写入自己的URL地址。

注意，所有向ZK上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。

另外，Dubbo还有针对服务粒度的监控，方法是订阅/dubbo/{serviceName}目录下所有提供者和消费者的信息。

在ZooKeeper集群当中，集群中的服务器角色有两种Leader和Learner，Learner角色又分为Observer和Follower，具体功能如下：

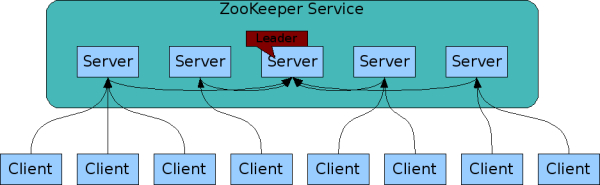
1.领导者(leader)，负责进行投票的发起和决议，更新系统状态

2.学习者(learner)，包括跟随者（follower）和观察者（observer），

3.follower用于接受客户端请求并向客户端返回结果，在选主过程中参与投票

4.Observer可以接受客户端请求，将写请求转发给leader，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。

5. 客户端(client)，请求发起方



**ZooKeeper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。

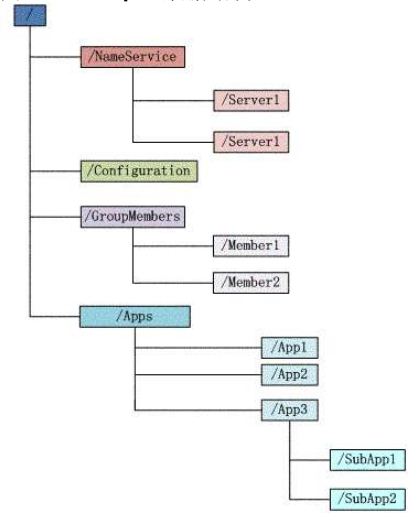
4.等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

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。

**ZooKeeper数据模型**

Zookeeper会维护一个具有层次关系的数据结构，它非常类似于一个标准的文件系统，如图所示：



Zookeeper这种数据结构有如下这些特点：

1）每个子目录项如NameService都被称作为znode，这个znode是被它所在的路径唯一标识，如Server1这个znode的标识为/NameService/Server1。

2）znode可以有子节点目录，并且每个znode可以存储数据，注意EPHEMERAL（临时的）类型的目录节点不能有子节点目录。

3）znode是有版本的（version），每个znode中存储的数据可以有多个版本，也就是一个访问路径中可以存储多份数据，version号自动增加。

4）znode可以是临时节点（EPHEMERAL），可以是持久节点（PERSISTENT）。如果创建的是临时节点，一旦创建这个EPHEMERALznode的客户端与服务器失去联系，这个znode也将自动删除，Zookeeper的客户端和服务器通信采用长连接方式，每个客户端和服务器通过心跳来保持连接，这个连接状态称为session，如果znode是临时节点，这个session失效，znode也就删除了。

5）znode的目录名可以自动编号，如App1已经存在，再创建的话，将会自动命名为App2。

6）znode可以被监控，包括这个目录节点中存储的数据的修改，子节点目录的变化等，一旦变化可以通知设置监控的客户端，这个是Zookeeper的核心特性，Zookeeper的很多功能都是基于这个特性实现的。

在每个ZNode上可存储少量数据(默认是1M, 可以通过配置修改, 通常不建议在ZNode上存储大量的数据)，这个特性非常有用。另外，每个ZNode上还存储了其Acl信息，这里需要注意，虽说ZNode的树形结构跟Unix文件系统很类似，但是其Acl与Unix文件系统是完全不同的，每个ZNode的Acl的独立的，子结点不会继承父结点的

**ZooKeeper Watch**

Zookeeper watch是一种监听通知机制。Zookeeper所有的读操作getData(), getChildren()和 exists()都可以设置监视(watch)，监视事件可以理解为一次性的触发器，官方定义如下： a watch event is one-time trigger, sent to the client that set the watch, whichoccurs when the data for which the watch was set changes。

# Zookeeper配置文件

Zookeeper的安装和配置十分简单, 既可以配置成单机模式, 也可以配置成集群模式. 下面将分别进行介绍.

### 单机模式

下载zookeeper的安装包之后, 解压到合适目录. 进入zookeeper目录下的conf子目录, 创建zoo.cfg:

**Bash代码  [收藏代码](javascript:void())**

1. tickTime=2000
2. initLimit=10
3. syncLimit=5
4. dataDir=/root/zookeeperData
5. # dataLogDir=/Users/apple/zookeeper/logs
6. clientPort=2181
7. maxClientCnxns=600

参数说明:

* tickTime: zookeeper中使用的基本时间单位, 毫秒值.
* dataDir: 数据目录. 可以是任意目录.
* dataLogDir: log目录, 同样可以是任意目录. 如果没有设置该参数, 将使用和dataDir相同的设置.
* clientPort: 监听client连接的端口号.

至此, zookeeper的单机模式已经配置好了. 启动server只需运行脚本:

**Bash代码**

1. bin/zkServer.sh start

 Server启动之后, 就可以启动client连接server了, 执行脚本:

**Bash代码**

1. bin/zkCli.sh -server localhost:2181

### 集群模式

集群模式的配置和伪集群基本一致.  
由于集群模式下, 各server部署在不同的机器上, 因此各server的conf/zoo.cfg文件可以完全一样.  
下面是一个示例:

**Bash代码  [收藏代码](javascript:void())**

1. tickTime=2000
2. initLimit=10
3. syncLimit=5
4. dataDir=/root/zookeeperdata
5. # dataLogDir=/home/zookeeper/logs
6. clientPort=2181
7. server.1=192.168.8.57:2182:2183
8. server.2=192.168.8.59:2182:2183
9. server.3=192.168.8.61:2182:2183

示例中部署了3台zookeeper server, 分别部署在192.168.8.57, 192.168.8.59, 192.168.8.61上. 需要注意的是, 各server的dataDir目录下的myid文件中的数字必须不同.

192.168.8.57 server的myid为1, 192.168.8.59 server的myid为2, 192.168.8.61 server的myid为3.

* initLimit: zookeeper集群中的包含多台server, 其中一台为leader, 集群中其余的server为follower. initLimit参数配置初始化连接时, follower和leader之间的最长心跳时间. 此时该参数设置为10, 说明时间限制为10倍tickTime, 即10\*2000=20000ms=20s.
* syncLimit: 该参数配置leader和follower之间发送消息, 请求和应答的最大时间长度. 此时该参数设置为5, 说明时间限制为5倍tickTime, 即10000ms.
* server.X=A:B:C 其中X是一个数字, 表示这是第几号server.

在之前设置的dataDir中新建myid文件, 写入一个数字, 该数字表示这是第几号server. 该数字必须和zoo.cfg文件中的server.X中的X一一对应.

# 目前我们项目中哪些地方用到zookeeper?

1,dubbo注册中心暴露服务地址(命名服务)

2,使用zookeeper实现分布式锁

# Zookeeper的简单使用

## Java中获取zookeeper连接

ZooKeeper zk=new ZooKeeper("192.168.8.22:2181", 10000, new Watcher() {

@Override

public void process(WatchedEvent event) {

//进行监听操作

}

});

第一个参数为zookeeper的地址,在集群环境下可配置多个地址,以逗号分割.

第二个参数为session的超时时间,单位为毫秒数

第三个参数为zookeeper的监听,zookeeper要求在创建的连接的时候,为连接建立一个默认的监听.

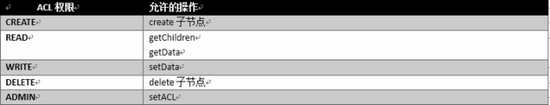
### Java中创建节点

String pathRoot=zk.create("/test", "节点内容".getBytes(), ZooDefs.Ids.OPEN\_ACL\_UNSAFE,CreateMode.PERSISTENT);

第一个参数为创建节点名称,

第二个参数为节点的内容

第三个参数为Access Control List（访问控制列表）。Znode被创建时带有一个ACL列表, ACL权限对应如下表：



而ZooDefs.Ids.OPEN\_ACL\_UNSAFE为所有权限

第四个参数为创建的节点类型,节点类型有持久节点,持久顺序节点,临时节点和临时顺序节点

**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。

**临时顺序节点（EPHEMERAL\_SEQUENTIAL）**

 可以用来实现分布式锁

### Java获取子节点

List<String> childrenList=zk.getChildren("/test", false);

第一个参数为获取哪一个节点下的子节点

第二个参数为是否为该节点添加监听

### Java设置节点内容

Stat stat=zk.setData("/test", "节点内容".getBytes(), 1);

第一个参数为节点路径

第二个参数为要设置的节点内容

第三个参数为该节点版本号

每一个znode都有一个数据版本号，它随着每次数据变化而自增。ZooKeeper提供的一些API例如setData和delete根据版本号有条件地执行。多个客户端对同一个znode进行操作时，版本号的使用就会显得尤为重要。例如，假设客户端C1对znode /config写入一些配置信息，如果另一个客户端C2同时更新了这个znode，此时C1的版本号已经过期，C1调用setData一定不会成功。这正是版本机制有效避免了数据更新时出现的先后顺序问题。在这个例子中，C1在写入数据时使用的版本号无法匹配，使得操作失败。

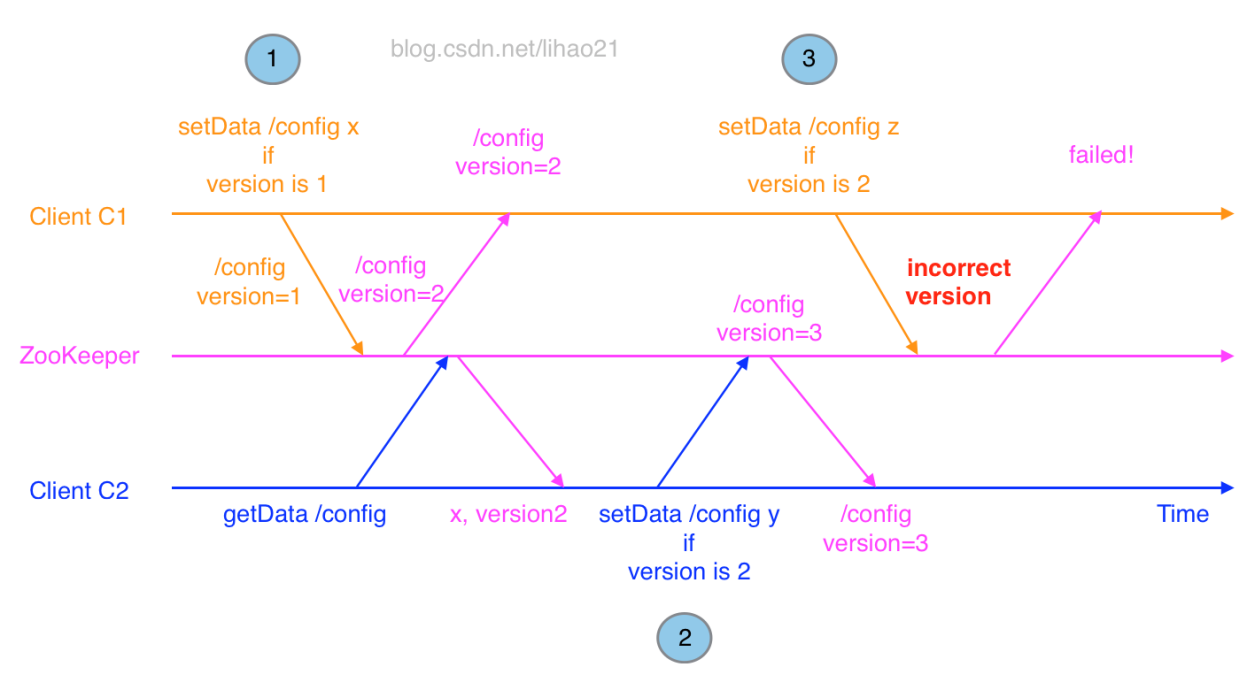


图2：使用版本号来阻止并行操作的不一致性

### Java获取节点内容

byte[] bytes=zk.getData("/test", true, new Stat());

第一个参数为节点路径

第二个参数为是否设置监听

第三个参数为节点状态,这里会将原stat状态赋给传入的stat

### Java获取节点状态

Stat stat=zk.exists("/test", true);

第一个参数为获取哪一个节点的状态

第二个参数为是否为该节点设置监听

### Java删除节点

zk.delete("/test", -1);

第一个参数为要删除的节点

第二个参数为版本号,如果请求的客户端版本号与znode的版本号不一致，将无法删除，是一种乐观加锁机制；如果将版本号设置为-1，不会去检测版本，直接删除；

### Java关闭zookeeper连接

zk.close();

# 分布式锁流程

获取锁实现思路：

1.     首先创建一个作为锁目录(znode)，通常用它来描述锁定的实体，称为:/lock\_node

2.     希望获得锁的客户端在锁目录下创建znode，作为锁/lock\_node的子节点，并且节点类型为有序临时节点(EPHEMERAL\_SEQUENTIAL)；

        例如：有一个客户在多线程下创建znode，分别为/lock\_node/lock000000001和/lock\_node/lock00000002

3.     当前客户端调用getChildren（/lock\_node）得到锁目录所有子节点，不设置watch，接着获取小于自己(步骤2创建)的兄弟节点

4.     步骤3中获取小于自己的节点不存在 && 最小节点与步骤2中创建的相同，说明当前客户端顺序号最小，获得锁，执行业务操作,删除本节点,结束。

5.     客户端监视(watch)相对自己次小的有序临时节点状态

6.     如果监视的次小节点状态发生变化，则跳转到步骤3，继续后续操作，直到退出锁竞争。

### 连接zookeeper

/\*\*

\* 连接zookeeper

\* @throws IOException

\* @throws InterruptedException

\*/

private ZooKeeper connectedZookeeper(final ZookeeperBean zkBean) throws IOException, InterruptedException {

ZooKeeper zk = new ZooKeeper(zkBean.getConnectionHost(),zkBean.getSessionTimeOut(), new Watcher() {

@Override

public void process(WatchedEvent event) {

if (event == null) {

return;

}

Event.KeeperState keeperState = event.getState();

Event.EventType eventType = event.getType();

if (Event.KeeperState.SyncConnected == keeperState) {

if (Event.EventType.None == eventType) {

log.info("成功连接上ZK服务器");

zkBean.getConnectedSemaphore().countDown();

} else if (event.getType() == Event.EventType.NodeDeleted

&& event.getPath().equals(zkBean.getWaitPath())) {

log.info(zkBean.getSelfPath() + "收到情报，排我前面的家伙已挂，我是不是可以出山了？");

try {

if (checkMinPath(zkBean)) {

getLockSuccess(zkBean);

}

}catch (Exception e) {

e.printStackTrace();

}

}

} else if (Event.KeeperState.Disconnected == keeperState) {

log.info("与ZK服务器断开连接");

} else if (Event.KeeperState.AuthFailed == keeperState) {

log.info("权限检查失败");

} else if (Event.KeeperState.Expired == keeperState) {

log.info("会话失效");

}

}

});

zkBean.getConnectedSemaphore().await(); //保证zookeeper连接成功

return zk;

}

### 创建节点

/\*\*

\* 创建节点

\* @param path 节点path

\* @param data 初始数据内容

\* @return

\*/

private boolean createPath(String path, String data, boolean needWatch,ZooKeeper zk) throws KeeperException,

InterruptedException {

if (zk.exists(path, needWatch) == null) {

log.info(Thread.currentThread()

+ "节点创建成功, Path: "

+ zk.create(path, data.getBytes(), ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT) + ", content: " + data);

}

return true;

}

### 获取锁

/\*\*

\* 获取锁

\* @param SUB\_PATH

\* @throws KeeperException

\* @throws InterruptedException

\* @throws Exception

\*/

private void getLock(ZookeeperBean zkBean) throws KeeperException,

InterruptedException, Exception {

String selfPath = zkBean.getZk().create(zkBean.getSubPath(), null, ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL\_SEQUENTIAL);

zkBean.setSelfPath(selfPath);

log.info("创建锁路径:" + selfPath);

if (checkMinPath(zkBean)) {

getLockSuccess(zkBean);

}

}

### 判断是否为最小节点

/\*\*

\* 判断当前节点是否为最小节点

\* @return

\* @throws KeeperException

\* @throws InterruptedException

\*/

private boolean checkMinPath(ZookeeperBean zkBean) throws KeeperException, InterruptedException {

List<String> subNodes = zkBean.getZk().getChildren(zkBean.getGroupPath(), false);

Collections.sort(subNodes);

int index = subNodes.indexOf(zkBean.getSelfPath().substring(zkBean.getGroupPath().length() + 1));

switch (index) {

case -1: {

log.error("本节点已不在了..." + zkBean.getSelfPath());

return false;

}

case 0: {

log.info("子节点中，我果然是老大" + zkBean.getSelfPath());

return true;

}

default: {

String waitPath = zkBean.getGroupPath() + "/" + subNodes.get(index - 1);

zkBean.setWaitPath(waitPath);

log.info(zkBean.getSelfPath() + "获取子节点中，排在我前面的" + waitPath);

try {

zkBean.getZk().getData(waitPath, true, new Stat());

return false;

} catch (KeeperException e) {

if (zkBean.getZk().exists(waitPath, false) == null) {

log.info("子节点中，排在我前面的" + waitPath + "已失踪，幸福来得太突然?");

return checkMinPath(zkBean);

} else {

throw e;

}

}

}

}

}

### 获取锁成功并执行业务操作

/\*\*

\* 获取锁成功后进行的操作

\* @throws Exception

\*/

private void getLockSuccess(ZookeeperBean zkBean) throws Exception {

if (zkBean.getZk().exists(zkBean.getSelfPath(), false) == null) {

log.error("本节点已不在了...");

return;

}

log.info("获取锁成功，赶紧干活！" + zkBean.getSelfPath());

// 业务开始

Object obj=zkBean.getClazz().getMethod(zkBean.getMethodName(), zkBean.getParamsClass()).invoke(zkBean.getObj(), zkBean.getParams());

// 业务结束

zkBean.setResult(obj);

deletePath(zkBean);

}

### 删除节点

/\*\*

\* 删除节点

\* @param zkBean

\* @throws InterruptedException

\* @throws KeeperException

\*/

private void deletePath(ZookeeperBean zkBean) throws InterruptedException,

KeeperException {

log.info("删除本节点：" + zkBean.getSelfPath());

zkBean.getZk().delete(zkBean.getSelfPath(), -1);

releaseConnection(zkBean.getZk());

}

### 关闭zookeeper连接

/\*\*

\* 关闭ZK连接

\*/

public void releaseConnection(ZooKeeper zk) {

if (zk != null) {

try {

zk.close();

} catch (InterruptedException e) {

}

}

log.info("释放连接");

}

# Zookeeper上层封装框架

### Curator框架

在使用ZK开发时会遇到让人头疼的几个问题，ZK连接管理、SESSION失效等一些异常问题的处理，Curator替我们解决了这些问题，通过对ZK连接状态的监控来做出相应的重连等操作，并触发事件！

更好的地方是Curator对ZK的一些应用场景提供了非常好的实现，而且有很多扩充，这些都符合ZK使用规范。

### Curator连接zookeeper

//为连接设置重连策略

RetryPolicy retryPolicy = new RetryUntilElapsed(30000, 1000);

//创建新的连接

client = CuratorFrameworkFactory.newClient("192.168.8.22:2181", retryPolicy);

//启动连接

client.start();

Curator 四种重连策略

1.RetryUntilElapsed(int maxElapsedTimeMs, int sleepMsBetweenRetries)

以sleepMsBetweenRetries的间隔重连,直到超过maxElapsedTimeMs的时间设置

2.RetryNTimes(int n, int sleepMsBetweenRetries)

指定重连次数

3.RetryOneTime(int sleepMsBetweenRetry)

重连一次,简单粗暴

4.ExponentialBackoffRetry

ExponentialBackoffRetry(int baseSleepTimeMs, int maxRetries)

ExponentialBackoffRetry(int baseSleepTimeMs, int maxRetries, int maxSleepMs)

基于"backoff"方式重连,和RetryUtilElapsed的区别是重连的时间间隔是动态的

时间间隔 = baseSleepTimeMs \* Math.max(1, random.nextInt(1 << (retryCount + 1)))

### Curator创建节点

### client.create().withMode(CreateMode.PERSISTENT).forPath("192.168.8.22:2181", "节点内容".getBytes());

### Curator获取子节点

List<String> list=client.getChildren().forPath("/test");

### Curator设置节点内容

client.setData().forPath("/test","节点内容".getBytes());

### Curator获取节点内容

Byte[] byte=client.getData().forPath("/test");

### Curator获取节点状态

Stat stat= client.checkExists().forPath("/test");

### Curator删除节点

client.delete().forPath("/test");

### Curator使用事务

CuratorTransaction transaction = client.inTransaction();

Collection<CuratorTransactionResult> results = transaction.create()

.forPath("/a/path", "some data".getBytes()).and().setData()

.forPath("/another/path", "other data".getBytes()).and().delete().forPath("/yet/another/path")

.and().commit();

### Curator实现分布式锁

InterProcessMutex lock = new InterProcessMutex(client,”/test”);

Boolean flag=lock.acquire(60, TimeUnit.SECONDS);

lock.release();