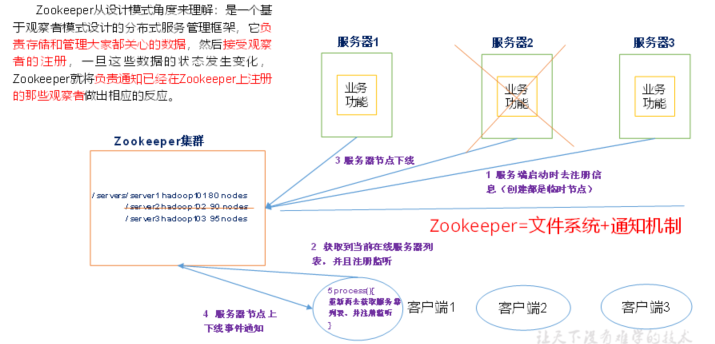
Zookeeper

一．概述

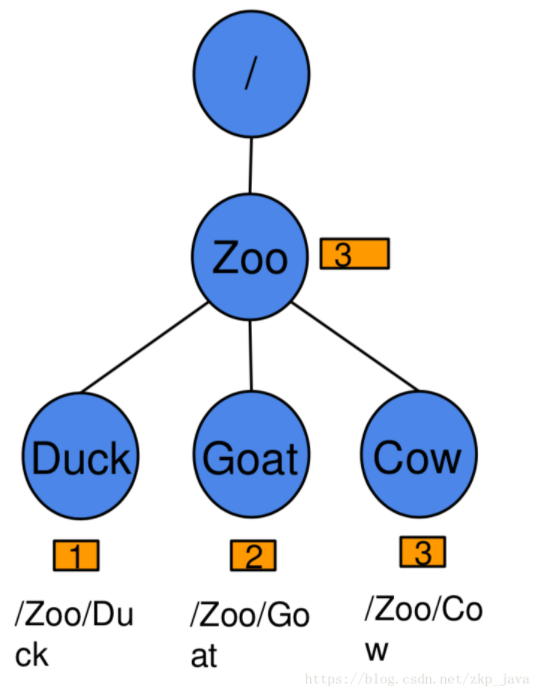
Zookeeper是一个开源的分布式的，为分布式应用提供协调服务的Apache项目。

1. 工作机制



2．Zookeeper的基本概念

2.1zookeeper数据模型是一种分层的树形结构，和linux的文件系统结构类似，



1. zookeeper集群中的角色

Leader: 为客户端提供事务性操作，读写服务；

Follower: 为客户端提供读服务，非事务性操作，客户端到Follower的写请求会转交给Leader角色，Follower会参与Leader的选举和投票；

Observer：为客户端提供读服务，非事务性操作，不参与Leader的选举过程，一般是为了增强zookeeper集群的读请求并发能力；

2）会话(Session)

session是客户端与zookeeper服务端之间建立的长链接；

zookeeper在一个会话中进行心跳检测来感知客户端链接的存活；

zookeeper客户端在一个会话中接收来自服务端的watch事件通知；

zookeeper可以给会话设置超时时间；

3）zookeeper的数据节点(ZNode)

Znode是zookeeper树形结构中的数据节点，用于存储数据；

Znode分为持久节点和临时节点两种类型：

持久节点：一旦创建，除非主动调用删除操作，否则一直存储在zookeeper上；

临时节点：与客户端回话绑定，一旦客户端失效，这个客户端创建的所有临时节点都会被删除；

顺序持久节点：带编码的持久节点

顺序临时节点：带编码的临时节点

4）zookeeper中的watcher

watcher监听在Znode节点上；

当节点的数据更新或子节点的增删状态发生变化都会使客户端的watcher得到通知；

5）zookeeper中的ACL(访问控制)

类似于Linux/Unix下的权限控制，有以下几种访问控制权限：

CREATE：创建子节点的权限；

READ：获取节点数据和子节点列表的权限；

WRITE：更新节点数据的权限；

DELETE: 删除子节点的权限；

ADMIN：设置节点ACL的权限；

4）Zookeeper集群的特点。

* 集群只要有半数以上节点存活，集群就可用，所以一般集群节点为奇数
* 全局数据一致性：每个节点数据都相同，client连接任意一个节点都可以
* 更新请求顺序进行，来自同一个client的更新按顺序进行
* 数据更新原子性，要么成功，要么失败

二．Zookeeper的安装

2.1 单机模式安装

1．安装前准备

（1）安装Jdk

（2）拷贝Zookeeper安装包到Linux系统下/usr/local/

（3）解压到指定目录

[root@localhost local]# tar -zxvf zookeeper-3.4.14.tar.gz

2．配置修改

（1）将/usr/local/zookeeper-3.4.14/conf这个路径下的zoo\_sample.cfg修改为zoo.cfg；

[root@localhost conf]# mv zoo\_sample.cfg zoo.cfg

（2）在/usr/local/zookeeper-3.4.14/这个目录上创建zkData文件夹,打开zoo.cfg文件，修改dataDir路径：

[root@localhost conf]# vim zoo.cfg

dataDir=/usr/local/zookeeper-3.4.14/zkData

3．操作Zookeeper基本命令

（1）启动Zookeeper

[root@localhost zookeeper-3.4.14]# ./bin/zkServer.sh start

（2）查看进程是否启动

[root@localhost zookeeper-3.4.14]# jps

4020 Jps

4001 QuorumPeerMain

（3）查看状态：

[root@localhost zookeeper-3.4.14]# ./bin/zkServer.sh status

（4）启动客户端：

[root@localhost zookeeper-3.4.14]# bin/zkCli.sh

（5）ls path [watch] 列出指定节点下的所有子节点，可选择监听子节点变化

（6）create [-s] [-e] path data acl

创建节点，-s代表创建的节点具有顺序，-e选项代表创建的是临时节点，默认情况下创建的是持久节点，path为节点的全路径，data为创建节点中的数据，acl用来进行权限控制，

（7）get path [watch] 获取path节点的数据内容和属性，watch监听节点变化

（8）set path data 设置节点数据

（9）delete path 删除节点

（9）rmr path 递归删除节点

（11）退出客户端：

[zk: localhost:2181(CONNECTED) 0] quit

（12）停止Zookeeper

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh stop

## 2.2 配置参数解读

Zookeeper中的配置文件zoo.cfg中参数含义解读如下：

1．tickTime =2000：通信心跳数，Zookeeper服务器与客户端心跳时间，单位毫秒

Zookeeper使用的基本时间，服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个tickTime时间就会发送一个心跳，时间单位为毫秒。

它用于心跳机制，并且设置最小的session超时时间为两倍心跳时间。(session的最小超时时间是2\*tickTime)

2．initLimit =10：LF初始通信时限

集群中的Follower跟随者服务器与Leader领导者服务器之间初始连接时能容忍的最多心跳数（tickTime的数量），用它来限定集群中的Zookeeper服务器连接到Leader的时限。

3．syncLimit =5：LF同步通信时限

集群中Leader与Follower之间的最大响应时间单位，假如响应超过syncLimit \* tickTime，Leader认为Follwer死掉，从服务器列表中删除Follwer。

4．dataDir：数据文件目录+数据持久化路径

主要用于保存Zookeeper中的数据。

5．clientPort =2181：客户端连接端口

监听客户端连接的端口。

**2.3 集群环境的搭建**

伪集群模式，即一台机器上启动三个zookeeper实例组成集群，真正的集群模式无非就是实例IP地址不同，搭建方法没有区别

1．下载解压zookeeper，方法和上面一样

2．重命名 zoo\_sample.cfg文件

[root@localhost conf]# cp zoo.cfg zoo-1.cfg

3．修改配置文件zoo-1.cfg，原配置文件里有的，修改成下面的值，没有的则加上（1）[root@localhost conf]# vim zoo-1.cfg dataDir=/usr/local/zookeeper-3.4.14/zkData/zk01

clientPort=2181

server.1=192.168.0.101:2887:3887

server.2=192.168.0.101:2888:3888

server.3=192.168.0.101:2889:3889

解读

server.A=B:C:D。

**A**是一个数字，表示这个是第几号服务器；

集群模式下配置一个文件myid，这个文件在dataDir目录下，这个文件里面有一个数据就是A的值，Zookeeper启动时读取此文件，拿到里面的数据与zoo.cfg里面的配置信息比较从而判断到底是哪个server。

**B**是这个服务器的地址；

**C**是这个服务器Follower与集群中的Leader服务器交换信息的端口；

**D**是万一集群中的Leader服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

4．复制配置，只需修改zoo-2.cfg和zoo-3.cfg的dataDir和clientPort不同即可

[root@localhost conf]# cp zoo-1.cfg zoo-2.cfg

[root@localhost conf]# cp zoo-1.cfg zoo-3.cfg

dataDir=/usr/local/zookeeper-3.4.14/zkData/zk02

clientPort=2182

dataDir=/usr/local/zookeeper-3.4.14/zkData/zk03

clientPort=2183

5. 标识Server ID

zkData下创建三个文件夹zk01，zk02，zk03，在每个目录中创建文件myid 文件，写入当前实例的server id，即1.2.3

[root@localhost zkData]# mkdir zk01

[root@localhost zkData]# mkdir zk02

[root@localhost zkData]# mkdir zk03

[root@localhost zkData]# touch ./zk01/myid

[root@localhost zkData]# touch ./zk02/myid

[root@localhost zkData]# touch ./zk03/myid

[root@localhost zkData]# vim ./zk01/myid 写入1

[root@localhost zkData]# vim ./zk02/myid 写入2

[root@localhost zkData]# vim ./zk03/myid 写入3

5．启动三个zookeeper实例

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh start conf/zoo-1.cfg

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh start conf/zoo-2.cfg

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh start conf/zoo-3.cfg

（2）启动客户端 zkCli.sh -server ip:port 连接到集群中的那个机器。

[root@localhost zookeeper-3.4.14]# bin/ zkCli.sh -server 192.168.0.101:2181

（3）查看状态

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh status conf/zoo-1.cfg

ZooKeeper JMX enabled by default

Using config: conf/zoo-1.cfg

Mode: follower

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh status conf/zoo-2.cfg

ZooKeeper JMX enabled by default

Using config: conf/zoo-2.cfg

Mode: leader

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh status conf/zoo-3.cfg

ZooKeeper JMX enabled by default

Using config: conf/zoo-3.cfg

Mode: follower

4)通知zookeeper

[root@localhost zookeeper-3.4.14]# bin/zkServer.sh stop conf/zoo-3.cfg

6.1 ACL全称是Access Control List，访问控制列表，zookeeper中ACL由三部分组成，即Scheme:id:permission，其中

scheme是验证过程中使用的检验策略

id是权限被赋予的对象，比如ip或某个用户

permission为可以操作的权限，有5个值：crdwa，分别表示create、read、delete、write、admin,admin是修改权限

setAcl path acl命令可以设置节点的访问权限，path是节点路径，acl是要设置的权限(crdwa)。通过getAcl path获取权限

6.2 权限检验策略即scheme有五种类型：world、auth、digest、IP、super

### world检验策略

ACL格式：world:anyone:permission

默认的检测策略为world则id固定位anyone，如果permission为crdwa则ACL为world:anyone:crdwa

/ 创建新节点的ACL默认为：world:anyone:crdwa

[zk: localhost:2181(CONNECTED) 43] create -e /acltest acltestdata

Created /acltest

[zk: localhost:2181(CONNECTED) 44] getAcl /acltest

'world,'anyone

: cdrwa

// 设置为ca权限

[zk: localhost:2181(CONNECTED) 46] setAcl /acltest world:anyone:ca

cZxid = 0x6e

ctime = Fri Sep 14 08:13:07 PDT 2018

mZxid = 0x6e

mtime = Fri Sep 14 08:13:07 PDT 2018

pZxid = 0x6e

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x100063d34d50005

dataLength = 11

numChildren = 0

[zk: localhost:2181(CONNECTED) 47] getAcl /acltest

'world,'anyone

: ca

// 无法再读取数据

[zk: localhost:2181(CONNECTED) 48] get /acltest

Authentication is not valid : /acltest

auth检验策略

ACL格式：auth:id:permission

比如auth:username:password:crdwa，auth检验策略表示给认证通过的所有用户设置acl权限。

可以通过addauth digest <username>:<password>命令添加用户。

如果通过addauth创建多组用户和密码，当使用setAcl修改权限时，所有的用户和密码的权限都会跟着修改，通过addauth新创建的用户和密码组需要重新调用setAcl才会加入到权限组当中去。

示例：

==================客户端1操作===========================

[zk: localhost:2181(CONNECTED) 66] create -e /acltest acltestdata // 创建测试的临时节点

Created /acltest

[zk: localhost:2181(CONNECTED) 67] getAcl /acltest

'world,'anyone

: cdrwa

[zk: localhost:2181(CONNECTED) 68] addauth digest acluser1:111111 // 添加acluser1用户

[zk: localhost:2181(CONNECTED) 69] addauth digest acluser2:222222 // 添加acluser2用户

// 设置acl权限，此处的用户名和密码并非一定要是上面创建的两个

// 用户名和密码，任何用户名和密码都可以，甚至不存在的用户名和

// 密码都可以,设置完后当前会话中所有用户对/acltest节点都具

// 有crdwa权限

[zk: localhost:2181(CONNECTED) 70] setAcl /acltest auth:acluser1:111111:crdwa

cZxid = 0x76

ctime = Fri Sep 14 09:13:12 PDT 2018

mZxid = 0x76

mtime = Fri Sep 14 09:13:12 PDT 2018

pZxid = 0x76

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x100063d34d50007

dataLength = 11

numChildren = 0

[zk: localhost:2181(CONNECTED) 71] get /acltest

acltestdata

cZxid = 0x76

ctime = Fri Sep 14 09:13:12 PDT 2018

mZxid = 0x76

mtime = Fri Sep 14 09:13:12 PDT 2018

pZxid = 0x76

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x100063d34d50007

dataLength = 11

numChildren = 0

=====================客户端2的操作====================

// 此时在另一个客户端中get /acltest会发现没有权限

[zk: localhost:2181(CONNECTED) 7] ls /

[acltest, zookeeper]

[zk: localhost:2181(CONNECTED) 8] get /acltest

Authentication is not valid : /acltest

=====================回到客户端1操作=====================

// 添加一个新的用户acluser3

[zk: localhost:2181(CONNECTED) 72] addauth digest acluser3:333333

=====================回到客户端2的操作====================

// 添加用户acluser3，发现还是没有访问/acltest的权限

[zk: localhost:2181(CONNECTED) 9] addauth digest acluser3:333333

[zk: localhost:2181(CONNECTED) 10] get /acltest

Authentication is not valid : /acltest

=====================回到客户端1操作=====================

// 重新设置/acltest权限

[zk: localhost:2181(CONNECTED) 73] setAcl /acltest auth:acluser1:111111:crdwa

cZxid = 0x76

ctime = Fri Sep 14 09:13:12 PDT 2018

mZxid = 0x76

mtime = Fri Sep 14 09:13:12 PDT 2018

pZxid = 0x76

cversion = 0

dataVersion = 0

aclVersion = 2 // aclVersion发生了变化

ephemeralOwner = 0x100063d34d50007

dataLength = 11

numChildren = 0

// 重新设置acl后三个用户均具有了访问/acltest节点的权限

[zk: localhost:2181(CONNECTED) 75] getAcl /acltest

'digest,'acluser1:hHUVzmra9P/TbXlP/4jRhG9jZm8=

: cdrwa

'digest,'acluser2:s097oNh4MU8FGvmhrQLPAUjmIs4=

: cdrwa

'digest,'acluser3:ML31yxAbaJNcHmBdnsIvVnUy+AI=

: cdrwa

=====================回到客户端2的操作====================

// 此时发现acluser3已经能够访问/acltest节点了

[zk: localhost:2181(CONNECTED) 12] get /acltest

acltestdata

cZxid = 0x76

ctime = Fri Sep 14 09:13:12 PDT 2018

mZxid = 0x76

mtime = Fri Sep 14 09:13:12 PDT 2018

pZxid = 0x76

cversion = 0

dataVersion = 0

aclVersion = 2

ephemeralOwner = 0x100063d34d50007

dataLength = 11

numChildren = 0

总结起来就是：auth检验策略下setAcl会设置当前会话所有拥有的所有用户访问节点的权限，当前回话先添加的用户需要重新设置节点的ACL权限才会把新用户对节点的操作权限加上去。

digest检验策略

ACL格式：digest:id:permission

digest和auth类似，只不过digest格式中的id需要使用sha1进行加密，zookeeper已经为我们提供了相关加密的类，如下所示为对id进行加密的代码：

public class DigestTest {

public static void main(String[] args) {

try {

System.out.println(DigestAuthenticationProvider.generateDigest("acluser1:111111"));

} catch (Exception ex) {

ex.printStackTrace();

}

}

}

示例：

=========================客户端1上的操作======================

[zk: localhost:2181(CONNECTED) 86] create /acltest acltestdata // 添加测试节点

Node already exists: /acltest

// 添加三个用户

[zk: localhost:2181(CONNECTED) 87] addauth digest acluser1:111111

[zk: localhost:2181(CONNECTED) 88] addauth digest acluser2:222222

[zk: localhost:2181(CONNECTED) 89] addauth digest acluser3:333333

// 设置digest类型的acl权限

[zk: localhost:2181(CONNECTED) 90] setAcl /acltest digest:acluser1:hHUVzmra9P/TbXlP/4jRhG9jZm8=:crdwa

cZxid = 0x7c

ctime = Fri Sep 14 18:42:17 PDT 2018

mZxid = 0x7c

mtime = Fri Sep 14 18:42:17 PDT 2018

pZxid = 0x7c

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x0

dataLength = 11

numChildren = 0

// 发现只有acluser1用户具有对/acltest节点的crdwa权限

[zk: localhost:2181(CONNECTED) 91] getAcl /acltest

'digest,'acluser1:hHUVzmra9P/TbXlP/4jRhG9jZm8=

: cdrwa

======================客户端2上的操作=======================

// 只添加acluser2用户，发现没有权限读取/acltest节点

[zk: localhost:2181(CONNECTED) 14] addauth digest acluser2:222222

[zk: localhost:2181(CONNECTED) 15] get /acltest

Authentication is not valid : /acltest

// 添加acluser1后，能够读取/acltest节点

[zk: localhost:2181(CONNECTED) 16] addauth digest acluser1:111111

[zk: localhost:2181(CONNECTED) 17] get /acltest

acltestdata

cZxid = 0x7c

ctime = Fri Sep 14 18:42:17 PDT 2018

mZxid = 0x7c

mtime = Fri Sep 14 18:42:17 PDT 2018

pZxid = 0x7c

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x0

dataLength = 11

numChildren = 0

小结：调用setAcl时不像auth那样会设置当前会话中所有用户访问节点的权限，只会设置指定的单个用户对节点的访问权限；setAcl设置中id需要使用sha1进行加密。

IP检验策略

ACL格式：ip:id:permission

id是ip地址，指定某个ip地址可以访问。

示例：

[zk: localhost:2181(CONNECTED) 96] create -e /acltest acltestdata

Created /acltest

// 设置只有192.168.1.1这台机器可以访问

[zk: localhost:2181(CONNECTED) 97] setAcl /acltest ip:192.168.1.1:crdwa

cZxid = 0x85

ctime = Fri Sep 14 18:59:23 PDT 2018

mZxid = 0x85

mtime = Fri Sep 14 18:59:23 PDT 2018

pZxid = 0x85

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x100063d34d5000b

dataLength = 11

numChildren = 0

// 本机地址为127.0.0.1,无法访问

[zk: localhost:2181(CONNECTED) 98] get /acltest

Authentication is not valid : /acltest

super检验策略

super检验策略主要供运维人员维护节点使用，acl中指定的用户具有任何操作任何节点的权限，启动时需要在启动脚本配置如下参数：

-Dzookeeper.DigestAuthenticationProvider.superDigest=admin:015uTByzA4zSglcmseJsxTo7n3c=

1

因为我喜欢在本地测试时前台启动zookeeper，因此我在zkServer.sh的如下位置加上该配置：

start-foreground)

ZOO\_CMD=(exec "$JAVA")

if [ "${ZOO\_NOEXEC}" != "" ]; then

ZOO\_CMD=("$JAVA")

fi

"${ZOO\_CMD[@]}" "-Dzookeeper.log.dir=${ZOO\_LOG\_DIR}" "-Dzookeeper.root.logger=${ZOO\_LOG4J\_PROP}" \

"-Dzookeeper.DigestAuthenticationProvider.superDigest=admin:015uTByzA4zSglcmseJsxTo7n3c=" \

-cp "$CLASSPATH" $JVMFLAGS $ZOOMAIN "$ZOOCFG"

8

测试示例，重新启动zookeeper服务：

=========================客户端1上操作=======================

[zk: localhost:2181(CONNECTED) 104] create -e /acltest acltestdata

Created /acltest

[zk: localhost:2181(CONNECTED) 105] addauth digest acluser1:111111

[zk: localhost:2181(CONNECTED) 107] setAcl /acltest digest:acluser1:hHUVzmra9P/TbXlP/4jRhG9jZm8=:crdwa

cZxid = 0x8a

ctime = Fri Sep 14 19:17:37 PDT 2018

mZxid = 0x8a

mtime = Fri Sep 14 19:17:37 PDT 2018

pZxid = 0x8a

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x10006e2720e0000

dataLength = 11

numChildren = 0

[zk: localhost:2181(CONNECTED) 108] getAcl /acltest

'digest,'acluser1:hHUVzmra9P/TbXlP/4jRhG9jZm8=

: cdrwa

========================客户端2上操作=========================

[zk: localhost:2181(CONNECTED) 20] ls /

[acltest, zookeeper]

[zk: localhost:2181(CONNECTED) 21] get /acltest

Authentication is not valid : /acltest

// 添加启动脚本配置的super用户

[zk: localhost:2181(CONNECTED) 22] addauth digest admin:111111

// 可以访问到节点

[zk: localhost:2181(CONNECTED) 23] get /acltest

acltestdata

cZxid = 0x8a

ctime = Fri Sep 14 19:17:37 PDT 2018

mZxid = 0x8a

mtime = Fri Sep 14 19:17:37 PDT 2018

pZxid = 0x8a

cversion = 0

dataVersion = 0

aclVersion = 1

ephemeralOwner = 0x10006e2720e0000

dataLength = 11

numChildren = 0

**2.4 java中操作zookeeper**

引入依赖

<dependency>  
 <groupId>org.apache.zookeeper</groupId>  
 <artifactId>zookeeper</artifactId>  
 <version>3.4.14</version>  
 </dependency>  
</

1

创建zookeeper会话

org.apache.zookeeper.ZooKeeper类的构造方法用于创建zookeeper客户端与服务端之间的会话。该类提供了如下几个构造方法：

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, boolean canBeReadOnly)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, long sessionId, byte[] sessionPasswd)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, long sessionId, byte[] sessionPasswd, boolean canBeReadOnly)

构造方法参数说明：

connectString：指zk的服物器列表，以英文输入法下逗号分割的host:port，比如?192.168.1.1:2181， 192.168.1.2:2181，也可以通过在后面跟着根目录，表示此客户端的操作都是在此根目录下，比如:比如192.168.1.1:2181，192.168.1.2:2181/zk-book,表示此客户端操作的节点都是在/zk-book根目录下，比如创建/foo/bar，实际完整路径为/zk-book/foo/bar；

sessionTimeout：会话超时时间，单位是毫秒，当在这个时间内没有收到心跳检测，会话就会失效；

watcher：注册的watcher，null表示不设置；

canBeReadOnly：用于标识当前会话是否支持”read-only”模式? ”，“read-only”模式是指当zk集群中的某台机器与集群中过半以上的机器网络端口不同，则此机器将不会接受客户端的任何读写请求，但是，有时候，我们希望继续提供读请求，因此设置此参数为true， 即客户端还以从与集群中半数以上节点网络不通的机器节点中读?数据；

sessionId和sessionPasswd：分别代表会话ID和会话密钥，这两个个参数一起可以唯一确定一个会话，客户端通过这两个参数可以实现客户端会话复用；

创建zookeeper节点

org.apache.zookeeper.ZooKeeper类提供了如下创建zk节点的api：

public String create(final String path, byte data[], List<ACL> acl, CreateMode createMode)

public void create(final String path, byte data[], List<ACL> acl, CreateMode createMode, StringCallback cb, Object ctx)

1

2

第一个方法以同步的方式创建节点，第二个方法以异步的方式创建节点，需要注意不论同步或异步都不支持递归创建节点，当节点已经存在时，会抛出NodeExistsException异常。

create方法参数说明：

path：被创建的节点路径，比如：/zk-book/foo；

data[]：节点中的数据，是一个字节数组；

acl：acl策略

createMode：节点类型，枚举类型，有四种选择：持久(PERSISTENT)、持久顺序(PERSISTENT\_SEQUENTIAL)、临时(EPHEMERAL)、临时顺序(EPHEMERAL\_SEQUENTIAL)；

cb：异步回调函数，需要实现接StringCallback接口，当服物器端创建完成后，客户端会自动调用 这个对象的方法processResult；

ctx：用于传递一个对象，可以在回调方法执行的时候使用，通常用于传递业务的上下文信息；

其中org.apache.zookeeper.data.ACL类中有两个成员：

private int perms;

private org.apache.zookeeper.data.Id id;

1

2

perms成员是ACL组成Scheme:id:permission中的permission，zk中perms定义如下：

public interface Perms {

int READ = 1 << 0;

int WRITE = 1 << 1;

int CREATE = 1 << 2;

int DELETE = 1 << 3;

int ADMIN = 1 << 4;

int ALL = READ | WRITE | CREATE | DELETE | ADMIN;

}

org.apache.zookeeper.data.Id类中定义了如下成员：

private String scheme;

private String id;

1

2

分别代表ACL组成Scheme:id:permission中的Scheme和id。(ACL参考文档[zookeeper] zookeeper系列二：zookeeper持久节点、临时节点及ACL )

其中org.apache.zookeeper.ZooDefs.Ids接口中有预先定义的几种ACL策略：

public interface Ids {

/\*\*

\* This Id represents anyone.

\*/

public final Id ANYONE\_ID\_UNSAFE = new Id("world", "anyone");

/\*\*

\* This Id is only usable to set ACLs. It will get substituted with the

\* Id's the client authenticated with.

\*/

public final Id AUTH\_IDS = new Id("auth", "");

/\*\*

\* This is a completely open ACL .

\* 相当于 world:anyone:cdrwa

\*/

public final ArrayList<ACL> OPEN\_ACL\_UNSAFE = new ArrayList<ACL>(

Collections.singletonList(new ACL(Perms.ALL, ANYONE\_ID\_UNSAFE)));

/\*\*

\* This ACL gives the creators authentication id's all permissions.

\* 相当于 相?于auth:用户:密码,但是需要通过ZooKeeper的addAuthInfo添加对应的用户和密码对

\*/

public final ArrayList<ACL> CREATOR\_ALL\_ACL = new ArrayList<ACL>(

Collections.singletonList(new ACL(Perms.ALL, AUTH\_IDS)));

/\*\*

\* This ACL gives the world the ability to read.

\* 相当于world:anyone:r，即所有人拥有读权限

\*/

public final ArrayList<ACL> READ\_ACL\_UNSAFE = new ArrayList<ACL>(

Collections

.singletonList(new ACL(Perms.READ, ANYONE\_ID\_UNSAFE)));

}

删除zookeeper节点

org.apache.zookeeper.ZooKeeper类提供了如下删除zk节点的api：

// 以同步的方式删除节点

public void delete(final String path, int version)

throws InterruptedException, KeeperException

// 以异步的方式删除节点，如果写测试代码，客户端主线程不能退出，否则可能请求没有发到服物器或者异步回调不成功

public void delete(final String path, int version, VoidCallback cb, Object ctx)

1

2

3

4

5

参数说明：

path：被删除节点的路径

version：节点的数据版本，如果指定的版本不是最新版本，将会报错

cb：异步回调函数

ctx：传递的上下文信息，即操作之前的信息传递到删除之后的异步回调函数里面

获取zookeeper子节点

org.apache.zookeeper.ZooKeeper类提供了如下获取zk子节点的api：

public List<String> getChildren(final String path, Watcher watcher) throws KeeperException, InterruptedException

public List<String> getChildren(String path, boolean watch) throws KeeperException, InterruptedException

public void getChildren(final String path, Watcher watcher, ChildrenCallback cb, Object ctx)

public void getChildren(String path, boolean watch, ChildrenCallback cb, Object ctx)

public List<String> getChildren(final String path, Watcher watcher, Stat stat)

public List<String> getChildren(String path, boolean watch, Stat stat) throws KeeperException, InterruptedException

public void getChildren(final String path, Watcher watcher, Children2Callback cb, Object ctx)

public void getChildren(String path, boolean watch, Children2Callback cb, Object ctx)

参数说明：

path：数据节点路径，比如?/zk-book/foo，获取该路径下的子节点列表

watcher：给节点设置的watcher，如果path对应节点的子节点数量发生变化，将会得到通知，允许?null

watch：使用使用默认watch，true的话当删除path节点或path子节点数量发生变化则默认watch或得到通知

stat：指定数据节点的状态信息

cb：异步回调函数

ctx：用于传递一个对象，可以在回调方法执行的时候使用，通常用于传递业务的上文信息

获取zookeeper节点数据

org.apache.zookeeper.ZooKeeper类提供了如下获取zk节点数据的api：

public byte[] getData(final String path, Watcher watcher, Stat stat) throws KeeperException, InterruptedException

public byte[] getData(String path, boolean watch, Stat stat) throws KeeperException, InterruptedException

public void getData(final String path, Watcher watcher, DataCallback cb, Object ctx)

public void getData(String path, boolean watch, DataCallback cb, Object ctx)

参数说明：

path:数据节点的路径，比如：/zk-book/foo，获取该路径节点的数据；

watcher:设置watcher后，如果path对应节点的数据发生变化(设置新的数据或删除节点)，将会得到通知，允许?null；

watch:是否使用默认的watcher；

stat:获取到的数据节点的状态信息将会保存到stat变量中，stat定义如下成员：

private long czxid;

private long mzxid;

private long ctime;

private long mtime;

private int version;

private int cversion;

private int aversion;

private long ephemeralOwner;

private int dataLength;

private int numChildren;

private long pzxid;

代表了数据节点的状态信息；

cb:异步回调函数；

ctx:用于传递一个对象，可以在回调方法执行的时候用，通常用于传递业务的上下文信息；

修改zookeeper节点数据

org.apache.zookeeper.ZooKeeper类提供了如下修改zk节点数据的api：

public Stat setData(final String path, byte data[], int version) throws KeeperException, InterruptedException

public void setData(final String path, byte data[], int version, StatCallback cb, Object ctx)

参数说明：

path:被修改的节点路径；

data:新的数据

version:指定的数据节点的版本，如果指定的版本不是最新版本，将会报错；

cb:异步回调函数；

ctx:传递的上下文信息，即操作之前的信息传递到删除之后的异步回调函数里面；

检查zookeeper节点是否存在

org.apache.zookeeper.ZooKeeper类提供了如下检查zk节点是否存在的api：

public Stat exists(final String path, Watcher watcher) throws KeeperException, InterruptedException

public Stat exists(String path, boolean watch) throws KeeperException, InterruptedException

public void exists(final String path, Watcher watcher, StatCallback cb, Object ctx)

public void exists(String path, boolean watch, StatCallback cb, Object ctx)

参数说明：

path:数据节点的路径；

watcher:需要注册的watcher，当监听的节点被创建、被删除或者被更新时该watcher会得到通知；

watch:是否使用默认的watcher；

cb:异步回调函数；

ctx:用于传递一个对象，可以在回调方法执行的时候用，通常用于传递业务的上下文信息；

zookeeper API使用示例

如下示例演示了zookeeper api的使用：

package com.ctrip.flight.test.zookeeper;

import org.apache.zookeeper.CreateMode;

import org.apache.zookeeper.ZooDefs;

import org.apache.zookeeper.ZooKeeper;

import org.apache.zookeeper.data.ACL;

import org.apache.zookeeper.data.Id;

import org.apache.zookeeper.data.Stat;

import org.apache.zookeeper.server.auth.DigestAuthenticationProvider;

import java.util.ArrayList;

import java.util.List;

public class ZkClientTest {

private static final String PARENT\_PATH = "/zkClientTest";

private static final String CHILD\_PATH = "/zkClientTest/childNodeTest";

private static final String IDENTITY = "zhangsan:123456";

public static void main(String[] args) {

try {

DefaultWatcher defaultWatcher = new DefaultWatcher();

ChildrenWatcher childrenWatcher = new ChildrenWatcher();

ParentWatcher parentWatcher = new ParentWatcher();

// 创建会话

ZooKeeper client = new ZooKeeper("192.168.0.102:2181,192.168.0.102:2182,192.168.0.102:2183", 30000, defaultWatcher);

client.addAuthInfo("digest", IDENTITY.getBytes());

Stat stat = client.exists(PARENT\_PATH, false);

if (null != stat) {

client.delete(PARENT\_PATH, -1);

}

// 创建节点，临时节点不能有子节点，所以父节点是持久节点

client.create(PARENT\_PATH, "zkClientTestData\_v1".getBytes(), getAcl(), CreateMode.PERSISTENT);

// 创建子节点

client.create(CHILD\_PATH, "childNodeData\_v1".getBytes(), getAcl(), CreateMode.EPHEMERAL);

// 获取子节点信息

Stat childStat = new Stat();

List<String> childs = client.getChildren(PARENT\_PATH, childrenWatcher, childStat);

System.out.println(PARENT\_PATH + "'s childs:" + childs);

System.out.println(PARENT\_PATH + "'s stat:" + childStat);

Thread.sleep(1000);

// 获取父节点数据

Stat parentStat = new Stat();

byte[] parentData = client.getData(PARENT\_PATH, parentWatcher, parentStat);

System.out.println(PARENT\_PATH + "'s data: " + new String(parentData));

System.out.println(PARENT\_PATH + "'s stat: " + parentStat);

Thread.sleep(1000);

// 设置子节点数据

childStat = client.setData(CHILD\_PATH, "childNodeData\_v2".getBytes(), -1);

System.out.println(CHILD\_PATH + "'s stat:" + childStat);

byte[] childData = client.getData(CHILD\_PATH, false, childStat);

System.out.println(CHILD\_PATH + "'s data:" + new String(childData));

Thread.sleep(1000);

// 删除子节点

client.delete(CHILD\_PATH, -1);

// 判断子节点是否存在

childStat = client.exists(CHILD\_PATH, false);

System.out.println(CHILD\_PATH + " is exist: " + (childStat != null));

client.delete(PARENT\_PATH, -1);

client.close();

Thread.sleep(1000);

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* ACL格式为：schema：id：permission

\* @return

\*/

private static List<ACL> getAcl() throws Exception {

List<ACL> acls = new ArrayList<>();

// 指定schema

String scheme = "auth";

// 指定id

// String identity = "zhangsan:123456";

Id id = new Id(scheme, DigestAuthenticationProvider.generateDigest(IDENTITY));

// Perms.ALL的权限为crdwa

ACL acl = new ACL(ZooDefs.Perms.ALL, id);

acls.add(acl);

return acls;

}

}

Watcher的使用

Watcher的原理



分别是zookeeper服务端、客户端以及客户端的watchManager。

如图所示，客户端向zk注册watcher的同时，会将客户端的watcher对象存储在客户端的WatchManager中；zk服务器触发watch事件后，会向客户端发送通知，客户端线程从watchManager中取出对应watcher执行。

客户端如何实现事件通知的动作

客户端只需定义一个类实现org.apache.zookeeper.Watcher接口并实现接口中的如下方法：

abstract public void process(WatchedEvent event);

1

即可在得到通知后执行相应的动作。参数org.apache.zookeeper.WatchedEvent是zk服务端传过来的事件，有三个成员：

final private KeeperState keeperState; // 通知状态

final private EventType eventType; // 事件类型

private String path; // 哪个节点发生的时间

1

2

3

分别代表通知的状态、事件类型和发生事件的节点。

keeperState是个枚举对象，代表客户端和zk服务器的链接状态，定义如下：

/\*\*

\* Enumeration of states the ZooKeeper may be at the event

\*/

public enum KeeperState {

/\*\* Unused, this state is never generated by the server \*/

@Deprecated

Unknown (-1),

/\*\* The client is in the disconnected state - it is not connected

\* to any server in the ensemble. \*/

Disconnected (0),

/\*\* Unused, this state is never generated by the server \*/

@Deprecated

NoSyncConnected (1),

/\*\* The client is in the connected state - it is connected

\* to a server in the ensemble (one of the servers specified

\* in the host connection parameter during ZooKeeper client

\* creation).

\* /

SyncConnected (3),

/\*\*

\* Auth failed state

\*/

AuthFailed (4),

/\*\*

\* The client is connected to a read-only server, that is the

\* server which is not currently connected to the majority.

\* The only operations allowed after receiving this state is

\* read operations.

\* This state is generated for read-only clients only since

\* read/write clients aren't allowed to connect to r/o servers.

\*/

ConnectedReadOnly (5),

/\*\*

\* SaslAuthenticated: used to notify clients that they are SASL-authenticated,

\* so that they can perform Zookeeper actions with their SASL-authorized permissions.

\*/

SaslAuthenticated(6),

/\*\* The serving cluster has expired this session. The ZooKeeper

\* client connection (the session) is no longer valid. You must

\* create a new client connection (instantiate a new ZooKeeper

\* instance) if you with to access the ensemble.

\*/

Expired (-112);

private final int intValue; // Integer representation of value

// for sending over wire

KeeperState(int intValue) {

this.intValue = intValue;

}

public int getIntValue() {

return intValue;

}

public static KeeperState fromInt(int intValue) {

switch(intValue) {

case -1: return KeeperState.Unknown;

case 0: return KeeperState.Disconnected;

case 1: return KeeperState.NoSyncConnected;

case 3: return KeeperState.SyncConnected;

case 4: return KeeperState.AuthFailed;

case 5: return KeeperState.ConnectedReadOnly;

case 6: return KeeperState.SaslAuthenticated;

case -112: return KeeperState.Expired;

default:

throw new RuntimeException("Invalid integer value for conversion to KeeperState");

}

}

}

eventType也是个枚举类型，代表节点发生的事件类型，比如创建新的子节点、改变节点数据等，定义如下：

/\*\*

\* Enumeration of types of events that may occur on the ZooKeeper

\*/

public enum EventType {

None (-1),

NodeCreated (1),

NodeDeleted (2),

NodeDataChanged (3),

NodeChildrenChanged (4),

DataWatchRemoved (5),

ChildWatchRemoved (6);

private final int intValue; // Integer representation of value

// for sending over wire

EventType(int intValue) {

this.intValue = intValue;

}

public int getIntValue() {

return intValue;

}

public static EventType fromInt(int intValue) {

switch(intValue) {

case -1: return EventType.None;

case 1: return EventType.NodeCreated;

case 2: return EventType.NodeDeleted;

case 3: return EventType.NodeDataChanged;

case 4: return EventType.NodeChildrenChanged;

case 5: return EventType.DataWatchRemoved;

case 6: return EventType.ChildWatchRemoved;

default:

throw new RuntimeException("Invalid integer value for conversion to EventType");

}

}

}

keeperState和eventType对应关系如下所示：



对于NodeDataChanged事件：无论节点数据发生变化还是数据版本发生变化都会触发(即使被更新数据与新数据一样，数据版本都会发生变化)。

对于NodeChildrenChanged事件：新增和删除子节点会触发该事件类型。

需要注意的是：WatchedEvent只是事件相关的通知，并没有对应数据节点的原始数据内容及变更后的新数据内容，因此如果需要知道变更前的数据或变更后的新数据，需要业务保存变更前的数据和调用接口获取新的数据

如何注册watcher

watcher注册api

可以在创建zk客户端实例的时候注册watcher(构造方法中注册watcher)：

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,ZKClientConfig conf)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, boolean canBeReadOnly, HostProvider aHostProvider)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, boolean canBeReadOnly, HostProvider aHostProvider,ZKClientConfig clientConfig)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, boolean canBeReadOnly)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, boolean canBeReadOnly, ZKClientConfig conf)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, long sessionId, byte[] sessionPasswd)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, long sessionId, byte[] sessionPasswd, boolean canBeReadOnly, HostProvider aHostProvider)

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher, long sessionId, byte[] sessionPasswd, boolean canBeReadOnly)

ZooKeeper的构造方法中传入的watcher将会作为整个zk会话期间的默认watcher，该watcher会一直保存为客户端ZKWatchManager的defaultWatcher成员，如果有其他的设置，这个watcher会被覆盖。

除了可以通过ZooKeeper类的构造方法注册watcher外，还可以通过ZooKeeper类中其他一些api来注册watcher，只不过这些api注册的watcher就不是默认watcher了(以下每个注册watcher的方法有很多个重载的方法，就不一一列举出来)。

public List<String> getChildren(final String path, Watcher watcher)

// boolean watch表示是否使用上下文中默认的watcher，即创建zk实例时设置的watcher

public List<String> getChildren(String path, boolean watch)

// boolean watch表示是否使用上下文中默认的watcher，即创建zk实例时设置的watcher

public byte[] getData(String path, boolean watch, Stat stat)

public void getData(final String path, Watcher watcher, DataCallback cb, Object ctx)

// boolean watch表示是否使用上下文中默认的watcher，即创建zk实例时设置的watcher

public Stat exists(String path, boolean watch)

public Stat exists(final String path, Watcher watcher)

watcher注册示例代码

本示例中使用zookeeper自带客户端演示watcher的使用，zookeeper自带客户端有一点需要注意：

Watcher设置后，一旦触发一次即会失效，如果需要一直监听，则需要再注册

定义默认watcher:

/\*\*

\* 测试默认watcher

\*/

public class DefaultWatcher implements Watcher {

@Override

public void process(WatchedEvent event) {

System.out.println("==========DefaultWatcher start==============");

System.out.println("DefaultWatcher state: " + event.getState().name());

System.out.println("DefaultWatcher type: " + event.getType().name());

System.out.println("DefaultWatcher path: " + event.getPath());

System.out.println("==========DefaultWatcher end==============");

}

}

定义监听子节点变化的watcher：

/\*\*

\* 用于监听子节点变化的watcher

\*/

public class ChildrenWatcher implements Watcher {

@Override

public void process(WatchedEvent event) {

System.out.println("==========ChildrenWatcher start==============");

System.out.println("ChildrenWatcher state: " + event.getState().name());

System.out.println("ChildrenWatcher type: " + event.getType().name());

System.out.println("ChildrenWatcher path: " + event.getPath());

System.out.println("==========ChildrenWatcher end==============");

}

}

定义监听节点变化的watcher：

public class DataWatcher implements Watcher {

@Override

public void process(WatchedEvent event) {

System.out.println("==========DataWatcher start==============");

System.out.println("DataWatcher state: " + event.getState().name());

System.out.println("DataWatcher type: " + event.getType().name());

System.out.println("DataWatcher path: " + event.getPath());

System.out.println("==========DataWatcher end==============");

}

}

watcher测试代码：

public class WatcherTest {

/\*\*

\* 链接zk服务端的地址

\*/

private static final String CONNECT\_STRING = "192.168.0.113:2181";

public static void main(String[] args) {

// 除了默认watcher外其他watcher一旦触发就会失效，需要充新注册，本示例中因为

// 还未想到比较好的重新注册watcher方式(考虑到如果在Watcher中持有一个zk客户端的

// 实例可能存在循环引用的问题)，因此暂不实现watcher失效后重新注册watcher的问题，

// 后续可以查阅curator重新注册watcher的实现方法。

// 默认watcher

DefaultWatcher defaultWatcher = new DefaultWatcher();

// 监听子节点变化的watcher

ChildrenWatcher childrenWatcher = new ChildrenWatcher();

// 监听节点数据变化的watcher

DataWatcher dataWatcher = new DataWatcher();

try {

// 创建zk客户端，并注册默认watcher

ZooKeeper zooKeeper = new ZooKeeper(CONNECT\_STRING, 100000, defaultWatcher);

// 让默认watcher监听 /GetChildren 节点的子节点变化

// zooKeeper.getChildren("/GetChildren", true);

// 让childrenWatcher监听 /GetChildren 节点的子节点变化(默认watcher不再监听该节点子节点变化)

zooKeeper.getChildren("/GetChildren", childrenWatcher);

// 让dataWatcher监听 /GetChildren 节点本省的变化(默认watcher不再监听该节点变化)

zooKeeper.getData("/GetChildren", dataWatcher, null);

TimeUnit.SECONDS.sleep(1000000);

} catch (Exception ex) {

ex.printStackTrace();

}

}

}

1. 使用开源curator操作zookeeper

zookeeper原生api的不足

zookeeper原生api存在以下不足之处：

连接的创建是异步的，需要开发人员自行编码实现等待；

连接没有自动的超时重连机制；

Zk本身不提供序列化机制，需要开发人员自行指定，从而实现数据的序列化和反序列化；

Watcher注册一次只会生效一次，需要不断的重复注册；

Watcher本身的使用方式不符合java本身的术语，如果采用监听器的方式，更容易理解；

不支持递归创建树形节点；

zookeeper第三方开源客户端

zookeeper的第三方开源客户端主要有zkClient和Curator。其中zkClient解决了session会话超时重连、Watcher反复注册等问题，提供了更加简洁的api，但zkClient社区不活跃，文档不够完善。而Curator是Apache基金会的顶级项目之一，它解决了session会话超时重连、Watcher反复注册、NodeExitsException异常等问题，Curator具有更加完善的文档，因此我们这里只学习Curator的使用。

Curator客户端api介绍

Curator包含了如下几个包：

curator-framework：对zookeeper底层api的一些封装；

curator-client：提供一些客户端的操作，如重试策略等；

curator-recipes：封装了一些高级特性，如Cache事件监听、选举、分布式锁、分布式计数器、分布式Barrier等

首先我们在gradle中引入curator的依赖就行：

curator提供了一种类似jdk8中stream一样的流式操作。

创建zookeeper会话

Curator中org.apache.curator.framework.CuratorFrameworkFactory类提供了如下两个创建zookeeper会话的方法：

public static CuratorFramework newClient(String connectString, RetryPolicy retryPolicy)

public static CuratorFramework newClient(String connectString, int sessionTimeoutMs, int connectionTimeoutMs, RetryPolicy retryPolicy)

该方法返回一个org.apache.curator.framework.CuratorFramework类型的对象，参数说明如下：

connectString：逗号分开的ip:port对；

sessionTimeoutMs：会话超时时间，单位为毫秒，默认是60000ms，指连接建立完后多久没有收到心跳检测，超过该时间即为会话超时；

connectionTimeoutMs：连接创建超时时间，单位为毫秒，默认是15000ms，指客户端与服务端建立连接时多长时间没连接上就算超时；

retryPolicy：重试策略，retryPolicy的类型定义如下

/\*\*

\* Abstracts the policy to use when retrying connections

\*/

public interface RetryPolicy

{

/\*\*

\* Called when an operation has failed for some reason. This method should return

\* true to make another attempt.

\*

\*

\* @param retryCount the number of times retried so far (0 the first time)，第几次重试

\* @param elapsedTimeMs the elapsed time in ms since the operation was attempted，到当前重试时刻总的重试时间

\* @param sleeper use this to sleep - DO NOT call Thread.sleep，重试策略

\* @return true/false

\*/

public boolean allowRetry(int retryCount, long elapsedTimeMs, RetrySleeper sleeper);

}

allowRetry返回true继续重试，返回false不再重试

可以通过实现该接口来自定义策略，curator已经为我们提供了若干重试策略：

ExponentialBackoffRetry：该重试策略随着重试次数的增加，sleep的时间呈指数增长，该提供了两个构造方法

public ExponentialBackoffRetry(int baseSleepTimeMs, int maxRetries)

public ExponentialBackoffRetry(int baseSleepTimeMs, int maxRetries, int maxSleepMs)

第retryCount次重试的sleep时间计算方式为：baseSleepTimeMs \* Math.max(1, random.nextInt(1 << (retryCount + 1)))，如果该值大于maxSleepMs，则sleep时间为maxSleepMs，如果重试次数大于maxRetries，则不再重试；

RetryNTimes：该重试策略重试指定次数，每次sleep固定的时间，构造方法如下

public RetryNTimes(int n, int sleepMsBetweenRetries)

n是重试的次数，sleepMsBetweenRetries是sleep的时间；

RetryOneTime：该重试策略只重试一次

RetryUntilElapsed：该重试策略对重试次数不做限制，但对总的重试时间做限制，构造方法如下

public RetryUntilElapsed(int maxElapsedTimeMs, int sleepMsBetweenRetries)

1

maxElapsedTimeMs是最大的重试时间，sleepMsBetweenRetries是sleep的时间间隔；

通过newClient获得CuratorFramework对象后我们就可以进行各种操作了。

除了newClient，CuratorFrameworkFactory还提供了一种Builder的方式来创建CuratorFramework对象，如下的示例所示：

RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000, 5);

CuratorFramework client = CuratorFrameworkFactory.builder()

.connectString("192.168.0.102:2181,192.168.0.102:2182,192.168.0.102:2183")

.sessionTimeoutMs(30000).connectionTimeoutMs(15000)

.retryPolicy(retryPolicy)

.namespace("curatorTest")

.build();

创建zookeeper节点

在curator中无论执行何种操作都必须先获得一个构建该操作的包装类(Builder对象)，创建zookeeper节点需要先获得一个org.apache.curator.framework.api.CreateBuilder(实际上是CreateBuilder的实现类CreateBuilderImpl)对象，然后用这个对象来构建创建节点的操作，CreateBuilderImpl中常见的操作如下：

// 递归创建(持久)父目录

public ProtectACLCreateModeStatPathAndBytesable<String> creatingParentsIfNeeded()

// 设置创建节点的属性

public ACLBackgroundPathAndBytesable<String> withMode(CreateMode mode)

// 设置节点的acl属性

public ACLBackgroundPathAndBytesable<String> withACL(List<ACL> aclList, boolean applyToParents)

// 指定创建节点的路径和节点上的数据

public String forPath(final String givenPath, byte[] data) throws Exception

如下所示为创建一个节点的示例：

String test1Data = client.create()

.creatingParentsIfNeeded()

.withMode(CreateMode.PERSISTENT)

.withACL(ZooDefs.Ids.OPEN\_ACL\_UNSAFE)

.forPath("/curatorTest/test1", "test1".getBytes());

删除zookeeper节点

同理先调用CuratorFramework的delete()获取构建删除操作的DeleteBuilder(实际上为DeleteBuilderImpl)，DeleteBuilderImpl提供了如下方法来构建删除操作：

// 指定要删除数据的版本号

public BackgroundPathable<Void> withVersion(int version)

// 确保数据被删除，本质上就是重试，当删除失败时重新发起删除操作

public ChildrenDeletable guaranteed()

// 指定删除的节点

public Void forPath(String path) throws Exception

// 递归删除子节点

public BackgroundVersionable deletingChildrenIfNeeded()

读取zookeeper节点数据

同理先调用CuratorFramework的getData()获取构建获取数据操作的GetDataBuilder(实际上为GetDataBuilderImpl)，GetDataBuilderImpl提供了如下方法来构建读取操作：

// 将节点状态信息保存到stat

public WatchPathable<byte[]> storingStatIn(Stat stat)

// 指定节点路径

public byte[] forPath(String path) throws Exception

如下示例为获取节点数据：

Stat test1Stat = new Stat();

byte[] test1DataBytes = client.getData().storingStatIn(test1Stat).forPath("/curatorTest/test1");

System.out.println("test1 data: " + new String(test1DataBytes));

更新zookeeper节点数据

同理先调用CuratorFramework的setData()获取构建获取数据操作的SetDataBuilder(实际上为SetDataBuilderImpl)，SetDataBuilderImpl提供了如下方法来构建更新操作：

// 指定版本号

public BackgroundPathAndBytesable<Stat> withVersion(int version)

// 指定节点路径和要更新的数据

public Stat forPath(String path, byte[] data) throws Exception

示例程序：

test1Stat = client.setData()

.withVersion(-1)

.forPath("/curatorTest/test1", "test1DataV2".getBytes());

读取zookeeper子节点

同理先调用CuratorFramework的getChildren()获取构建获取子节点数据操作的GetChildrenBuilder(实际上为GetChildrenBuilderImpl)，GetChildrenBuilderImpl提供了如下方法来构建更新操作：

// 把服务器端获取到的状态数据存储到stat对象中

public WatchPathable<List<String>> storingStatIn(Stat stat)

// 指定获取子节点数据的节点路径

public List<String> forPath(String path) throws Exception

// 设置watcher，类似于zookeeper本身的api，也只能使用一次

public BackgroundPathable<List<String>> usingWatcher(Watcher watcher)

public BackgroundPathable<List<String>> usingWatcher(CuratorWatcher watcher)

示例程序：

Stat childStat = new Stat();

List<String> childs = client.getChildren().storingStatIn(childStat).forPath("/curatorTest");

curator中关于异步操作

curator为所有操作都提供了异步执行的版本，只需要在构建操作的方法链中添加如下操作之一即可：

public ErrorListenerPathable<List<String>> inBackground()

public ErrorListenerPathable<List<String>> inBackground(Object context)

public ErrorListenerPathable<List<String>> inBackground(BackgroundCallback callback)

public ErrorListenerPathable<List<String>> inBackground(BackgroundCallback callback, Object context)

public ErrorListenerPathable<List<String>> inBackground(BackgroundCallback callback, Executor executor)

public ErrorListenerPathable<List<String>> inBackground(BackgroundCallback callback, Object context, Executor executor)

如下示例程序为使用异步执行删除操作：

client.delete()

.guaranteed()

.withVersion(-1)

.inBackground(((client1, event) -> {

System.out.println(event.getPath() + ", data=" + event.getData());

System.out.println("event type=" + event.getType());

System.out.println("event code=" + event.getResultCode());

}))

.forPath("/curatorTest/test1");

curator中的NodeCache

NodeCache会将某一路径的节点(节点本身)在本地缓存一份，当zookeeper中相应路径的节点发生更新、创建或者删除操作时，NodeCache将会得到响应，并且会将最新的数据拉到本地缓存中，NodeCache只会监听路径本身的变化，并不会监听子节点的变化。我们可以通过NodeCache注册一个监听器来获取发生变化的通知。NodeCache提供了如下构造函数：

public NodeCache(CuratorFramework client, String path)

public NodeCache(CuratorFramework client, String path, boolean dataIsCompressed)

1

2

参数说明：

client: curator客户端；

path: 需要缓存的节点路径；

dataIsCompressed：是否压缩节点下的数据；

NodeCache提供了一个如下类型的监听器容器，只要往容器中添加监听器，当节点发生变更时，容器中的监听器都将得到通知。

private final ListenerContainer<NodeCacheListener> listeners = new ListenerContainer<NodeCacheListener>();

1

NodeCache缓存数据及添加Listener的示例代码如下：

NodeCache nodeCache = new NodeCache(client, "/curatorTest/test1");

// 是否立即拉取/curatorTest/test1节点下的数据缓存到本地

nodeCache.start(true);

// 添加listener

nodeCache.getListenable().addListener(() -> {

ChildData childData = nodeCache.getCurrentData();

if (null != childData) {

System.out.println("path=" + childData.getPath() + ", data=" + childData.getData() + ";");

}

});

Note: NodeCache只会缓存节点本身的数据和状态，并不会缓存节点下的子节点信息，所以如果我们在节点下创建子节点，NodeCache中的Listener是不会得到通知的\*

curator中的PathChildrenCache

PathChildrenCache会将指定路径节点下的所有子节点缓存在本地，但不会缓存节点本身的信息，当执行新增(CHILD\_ADDED)、删除(CHILD\_REMOVED)、更新(CHILD\_UPDATED)指定节点下的子节点等操作时，PathChildrenCache中的Listener将会得到通知，PathChildrenCache提供了如下几个构造函数：

public PathChildrenCache(CuratorFramework client, String path, boolean cacheData)

public PathChildrenCache(CuratorFramework client, String path, boolean cacheData, ThreadFactory threadFactory)

public PathChildrenCache(CuratorFramework client, String path, boolean cacheData, boolean dataIsCompressed, ThreadFactory threadFactory)

public PathChildrenCache(CuratorFramework client, String path, boolean cacheData, boolean dataIsCompressed, final ExecutorService executorService)

public PathChildrenCache(CuratorFramework client, String path, boolean cacheData, boolean dataIsCompressed, final CloseableExecutorService executorService)

参数说明：

client：curator客户端；

path：缓存的节点路径；

cacheData：除了缓存节点状态外是否缓存节点数据，如果为true，那么客户端在接收到节点列表变更的同时，也能够获取到节点的数据内容，如果为false，则无法获取到数据内容；

threadFactory：线程池工厂，当内部需要开启新的线程执行时，使用该线程池工厂来创建线程；

dataIsCompressed：是否压缩节点数据；

executorService：线程池；

PathChildrenCache通过start方法可以传入三种启动模式，这三种启动模式定义在org.apache.curator.framework.recipes.cache.PathChildrenCache.StartMode中：

NORMAL：异步初始化cache；

BUILD\_INITIAL\_CACHE：同步初始化cache，以及创建cache后，就从服务器拉取对应的数据；

POST\_INITIALIZED\_EVENT：异步初始化cache，初始化完成触发PathChildrenCacheEvent.Type#INITIALIZED事件，cache中Listener会收到该事件的通知；

PathChildrenCache示例代码如下：

PathChildrenCache pathChildrenCache = new PathChildrenCache(client, "/curatorTest", true);

// startMode为BUILD\_INITIAL\_CACHE，cache是初始化完成会发送INITIALIZED事件

pathChildrenCache.start(PathChildrenCache.StartMode.BUILD\_INITIAL\_CACHE);

System.out.println(pathChildrenCache.getCurrentData().size());

pathChildrenCache.getListenable().addListener(((client1, event) -> {

ChildData data = event.getData();

switch (event.getType()) {

case INITIALIZED:

System.out.println("子节点cache初始化完成(StartMode为POST\_INITIALIZED\_EVENT的情况)");

System.out.println("INITIALIZED: " + pathChildrenCache.getCurrentData().size());

break;

case CHILD\_ADDED:

System.out.println("添加子节点，path=" + data.getPath() + ", data=" + new String(data.getData()));

break;

case CHILD\_UPDATED:

System.out.println("更新子节点，path=" + data.getPath() + ", data=" + new String(data.getData()));

break;

case CHILD\_REMOVED:

System.out.println("删除子节点，path=" + data.getPath());

break;

default:

System.out.println(event.getType());

}

}));

curator完整示例代码

如下所示为演示curator使用的完整示例代码：

public class CuratorTest {

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

RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000, 5);

CuratorFramework client = CuratorFrameworkFactory.builder()

.connectString("192.168.0.104:2181,192.168.0.104:2182,192.168.0.104:2183")

.sessionTimeoutMs(30000).connectionTimeoutMs(15000)

.retryPolicy(retryPolicy)

//.namespace("curatorTest")

.build();

client.start();

// 判断节点是否存在，存在则先删除节点

Stat test1Stat = client.checkExists().forPath("/curatorTest/test1");

if (null != test1Stat) {

client.delete().guaranteed().deletingChildrenIfNeeded().withVersion(-1).forPath("/curatorTest/test1");

}

// 创建节点

String test1Data = client.create()

.creatingParentsIfNeeded()

.withMode(CreateMode.PERSISTENT)

.withACL(ZooDefs.Ids.OPEN\_ACL\_UNSAFE)

.forPath("/curatorTest/test1", "test1DataV1".getBytes());

// 获取节点信息

test1Stat = new Stat();

byte[] test1DataBytes = client.getData().storingStatIn(test1Stat).forPath("/curatorTest/test1");

System.out.println("test1 stat: " + test1Stat);

System.out.println("test1 data: " + new String(test1DataBytes));

// 更新节点数据

test1Stat = client.setData()

.withVersion(-1)

.forPath("/curatorTest/test1", "test1DataV2".getBytes());

System.out.println("test1 stat: " + test1Stat);

// 获取所有子节点

Stat childStat = new Stat();

List<String> childs = client.getChildren().storingStatIn(childStat).forPath("/curatorTest");

System.out.println("curatorTest childs: " + childs);

// client.delete()

// .guaranteed()

// .withVersion(-1)

// .inBackground(((client1, event) -> {

// System.out.println(event.getPath() + ", data=" + event.getData());

// System.out.println("event type=" + event.getType());

// System.out.println("event code=" + event.getResultCode());

// }))

// .forPath("/curatorTest/test1");

// 缓存节点

NodeCache nodeCache = new NodeCache(client, "/curatorTest/test1");

nodeCache.start(true);

nodeCache.getListenable().addListener(() -> {

System.out.println("NodeCache:");

ChildData childData = nodeCache.getCurrentData();

if (null != childData) {

System.out.println("path=" + childData.getPath() + ", data=" + new String(childData.getData()) + ";");

}

});

// 缓存子节点

PathChildrenCache pathChildrenCache = new PathChildrenCache(client, "/curatorTest", true);

// startMode为BUILD\_INITIAL\_CACHE，cache是初始化完成会发送INITIALIZED事件

pathChildrenCache.start(PathChildrenCache.StartMode.BUILD\_INITIAL\_CACHE);

System.out.println(pathChildrenCache.getCurrentData().size());

pathChildrenCache.getListenable().addListener(((client1, event) -> {

ChildData data = event.getData();

switch (event.getType()) {

case INITIALIZED:

System.out.println("子节点cache初始化完成(StartMode为POST\_INITIALIZED\_EVENT的情况)");

System.out.println("INITIALIZED: " + pathChildrenCache.getCurrentData().size());

break;

case CHILD\_ADDED:

System.out.println("添加子节点，path=" + data.getPath() + ", data=" + new String(data.getData()));

break;

case CHILD\_UPDATED:

System.out.println("更新子节点，path=" + data.getPath() + ", data=" + new String(data.getData()));

break;

case CHILD\_REMOVED:

System.out.println("删除子节点，path=" + data.getPath());

break;

default:

System.out.println(event.getType());

}

}));

Thread.sleep(20000000);

}

}

三 zookeeper集群的选举

Zookeeper集群的选举是通过ZAB(Zookeeper Atomic Broadcast)协议，即Zookeeper原子消息广播协议算法实现的，zab是paxos算法实现的一种。

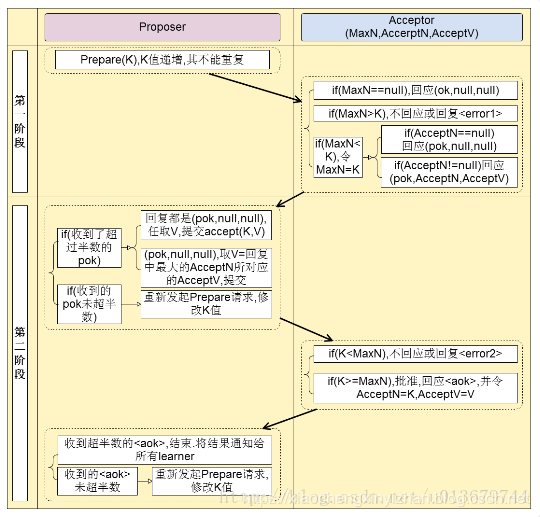
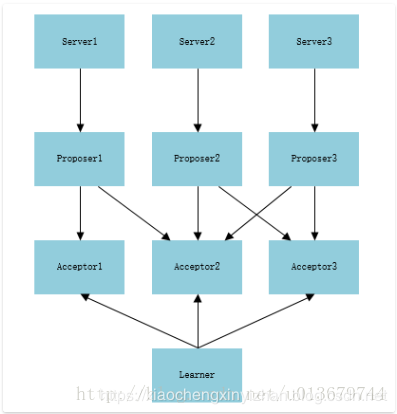
Paxos算法概要

Paxos算法一种基于消息传递且具有高度容错特性的一致性算法。

分布式系统中的节点通信存在两种模型：共享内存（Shared memory）和消息传递（Messages passing）。基于消息传递通信模型的分布式系统，不可避免的会发生以下错误：进程可能会慢、被杀死或者重启，消息可能会延迟、丢失、重复，在基础 Paxos 场景中，先不考虑可能出现消息篡改即拜占庭错误的情况。Paxos 算法解决的问题是在一个可能发生上述异常的分布式系统中如何就某个值达成一致，保证不论发生以上任何异常，都不会破坏决议的一致性。

Paxos算法类似于两阶段提提交，其算法执行过程分为两个阶段。具体如下：

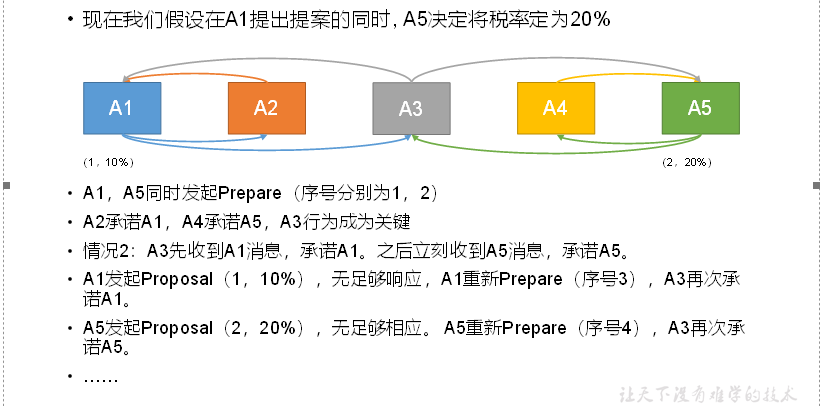
1.  阶段一（prepare阶段）：  
   (a) Proposer选择一个提案编号N，然后向半数以上的Acceptor发送编号为N的Prepare请求。Pareper（N）  
   (b) 如果一个Acceptor收到一个编号为N的Prepare请求，如果acceptor没接收过请求，则返回ok，如果小于它已经响应过的请求，则拒绝，不回应或回复error。若N大于该Acceptor已经响应过的所有Prepare请求的编号（maxN），那么它就会将它已经接受过（已经经过第二阶段accept的提案）的编号最大的提案（如果有的话，如果还没有的accept提案的话返回{pok，null，null}）作为响应反馈给Proposer，同时该Acceptor承诺不再接受任何编号小于等于N的Pareper提案（阶段一）。承诺三：不再接受Proposal ID小于（注意：这里是< ）当前请求的Propose请求（阶段二的accept）。

*  阶段二（accept阶段）：  
  (a) 如果一个Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应，那么它就会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。注意：V就是收到的响应中编号最大的提案的value（某个acceptor响应的它已经通过的{acceptN，acceptV}），如果响应中不包含任何提案，那么V就由Proposer自己决定。  
  (b) 如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出过响应，它就接受该提案。如果N小于Acceptor以及响应的prepare请求，则拒绝，不回应或回复error（当proposer没有收到过半的回应，那么他会重新进入第一阶段，递增提案号，重新提出prepare请求）。  
  在上面的运行过程中，每一个Proposer都有可能会产生多个提案。但只要每个Proposer都遵循如上述算法运行，就一定能保证算法执行的正确性。  
  
* 具体实例理解：  
  问题背景：假设我们有下图的系统，想要在server1，server2，server3选一个master。  
    
  prepare阶段  
   1. 每个server向proposer发送消息，表示自己要当leader，假设proposer收到消息的时间不一样，顺序是： proposer2 -> proposer1 -> proposer3，消息编号依次为1、2、3。  
    紧接着，proposer将消息发给acceptor中超过半数的子成员(这里选择两个)，如图所示，proposer2向acceptor2和acceptor3发送编号为1的消息，proposer1向acceptor1和accepto2发送编号为2的消息，proposer3向acceptor2和acceptor3发送编号为3的消息。

1. 假设这时proposer1发送的消息先到达acceptor1和acceptor2，它们都没有接收过请求，所以接收该请求并返回【pok，null，null】给proposer1，同时acceptor1和acceptor2承诺不再接受编号小于2的请求；  
     紧接着，proposer2的消息到达acceptor2和acceptor3，acceptor3没有接受过请求，所以返回proposer2 【pok，null，null】，acceptor3并承诺不再接受编号小于1的消息。而acceptor2已经接受proposer1的请求并承诺不再接收编号小于2的请求，所以acceptor2拒绝proposer2的请求；  
     最后，proposer3的消息到达acceptor2和acceptor3，它们都接受过提议，但编号3的消息大于acceptor2已接受的2和acceptor3已接受的1，所以他们都接受该提议，并返回proposer3 【pok，null，null】；  
     此时，proposer2没有收到过半的回复，所以重新取得编号4，并发送给acceptor2和acceptor3，此时编号4大于它们已接受的提案编号3，所以接受该提案，并返回proposer2 【pok，null，null】。

accept阶段  
1  
  Proposer3收到半数以上（两个）的回复，并且返回的value为null，所以，proposer3提交了【3，server3】的提案。  
  Proposer1也收到过半回复，返回的value为null，所以proposer1提交了【2，server1】的提案。  
  Proposer2也收到过半回复，返回的value为null，所以proposer2提交了【4，server2】的提案。  
（这里要注意，并不是所有的proposer都达到过半了才进行第二阶段，这里只是一种特殊情况）  
2  
  Acceptor1和acceptor2接收到proposer1的提案【2，server1】，acceptor1通过该请求，acceptor2承诺不再接受编号小于4的提案，所以拒绝；  
  Acceptor2和acceptor3接收到proposer2的提案【4，server2】，都通过该提案；  
  Acceptor2和acceptor3接收到proposer3的提案【3，server3】，它们都承诺不再接受编号小于4的提案，所以都拒绝。  
所以proposer1和proposer3会再次进入第一阶段，但这时候 Acceptor2和acceptor3已经通过了提案（AcceptN = 4，AcceptV=server2），并达成了多数，所以proposer会递增提案编号，并最终改变其值为server2。最后所有的proposer都肯定会达成一致，这就迅速的达成了一致。  
  此时，过半的acceptor（acceptor2和acceptor3）都接受了提案【4，server2】，learner感知到提案的通过，learner开始学习提案，所以server2成为最终的leader。

Paxos算法缺陷：在网络复杂的情况下，一个应用Paxos算法的分布式系统，可能很久无法收敛，甚至陷入活锁的情况。



造成这种情况的原因是系统中有一个以上的Proposer，多个Proposers相互争夺Acceptors，造成迟迟无法达成一致的情况。针对这种情况，一种改进的Paxos算法被提出：从系统中选出一个节点作为Leader，只有Leader能够发起提案。这样，一次Paxos流程中只有一个Proposer，不会出现活锁的情况，此时只会出现例子中第一种情况。

3.2 zookeeper的选举过程

Zookeeper集群中节点有如下三种角色：

1）Leader:事务请求的唯一调度和处理者，保证集群事务处理的顺序性，同时也是集群内部个服务器的调度者；

Follower:处理客户端的非事务请求，转发事务请求给Leader服务器，参与事务请求Proposal的投票，参与Leader选举投票；

Observer:处理客户端非事务请求，转发事务请求给Leader服务器，不参与任何形式的投票，包括选举和事务投票(超过半数确认)，此角色存在通常是为了提高读性能；

2）Zookeeper集群中节点存在如下几种状态：

LOOKING:寻找Leader的状态，当服务器处于此状态时，表示当前没有Leader，需要进入选举流程；

FOLLOWING:跟随者状态，表明当前服务器角色是Follower；

OBSERVING:观察者状态，表明当前服务器是Observer；

LEADING:领导者状态，表明当前服务器角色为Leader；

3）影响节点称为Leader的因素，超过一半的选举就会成为leader

Zookeeper通过以下因素来判断一个节点能否称为Leader：

数据的新旧程度：只有最新的数据节点才有机会成为Leader，在Zookeeper中通过事物id(zxid)的大小来表示数据的新旧，越大代表数据越新；

myid：集群在启动的时候，会在数据目录下配置myid文件，里面的数字代表当前zk节点的编号，当zk节点数据一样新时，myid中数字越大的就会被选举成为Leader，当集群中已经有Leader时，新加入的节点不会影响原来的集群；

3）以一个简单的例子来说明整个选举的过程。

假设有五台服务器组成的Zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么，如图5-8所示。



图5-8 Zookeeper的选举机制

（1）服务器1启动，发起一次选举。服务器1投自己一票。此时服务器1票数一票，不够半数以上（3票），选举无法完成，服务器1状态保持为LOOKING；

（2）服务器2启动，再发起一次选举。服务器1和2分别投自己一票并交换选票信息：此时服务器1发现服务器2的ID比自己目前投票推举的（服务器1）大，更改选票为推举服务器2。此时服务器1票数0票，服务器2票数2票，没有半数以上结果，选举无法完成，服务器1，2状态保持LOOKING

（3）服务器3启动，发起一次选举。此时服务器1和2都会更改选票为服务器3。此次投票结果：服务器1为0票，服务器2为0票，服务器3为3票。此时服务器3的票数已经超过半数，服务器3当选Leader。服务器1，2更改状态为FOLLOWING，服务器3更改状态为LEADING；

（4）服务器4启动，发起一次选举。此时服务器1，2，3已经不是LOOKING状态，不会更改选票信息。交换选票信息结果：服务器3为3票，服务器4为1票。此时服务器4服从多数，更改选票信息为服务器3，并更改状态为FOLLOWING；

（5）服务器5启动，同4一样当小弟。

运次那个过程中Leader宕机后新Leader的选举

假设server2为主节点，并且server2宕机，剩下server1和server3进行Leader的选举。选举流程如下：

变更状态：Leader宕机后，其他节点的状态变为LOOKING；

生成投票信息：每个server发出一个投自己的票的投票，假定生成的投票信息为(myid, zxid)的形式，server1的投票信息为(1, 123)，并将该投票信息发给server3，server3的投票信息为(3, 122)，并将该投票信息发给server1；

投票处理：server3收到server1的投票信息(1, 123)，发现该投票的zxid 123比server1自己投票信息中的zxid 122大，则server3修改自己的投票信息为(1, 123)，然后发给server1

投票处理：server1收到server3的投票信息(3, 122)，发现zxid 122比自己的投票信息中的zxid 123要小，则不改变自己的投票；

统计投票信息：server3统计收到的投票(包括自己投的)，(1, 123)是两票，server1统计收到的投票(包括自己投的)，(1, 123)是两票；

修改服务器状态：server3中选出的Leader是1，而自己是3，因此自己进入FOLLOWING状态，即follower角色，server1中选出的Leader是1，自己就是1，因此进入LEADING状态，即Leader角色；

当Leader选举完成后，Follower需要与新的Leader同步数据。进入数据同步阶段。

5.zookeeper的数据同步

Zookeeper数据同步

当Leader完成选举后，Follower需要与新的Leader同步数据。在Leader端需要做如下工作：

Leader告诉其他Follower当前最新数据是什么即zxid，Leader会构建一个NEWLEADER包，包括当前最大的zxid，发送给所有的Follower或者Observer；

Leader给每个Follower创建一个线程LearnerHandler来负责处理每个Follower的数据同步请求，同时主线程开始阻塞，只有超过一半的Follower同步完成，同步过程才完成，Leader才能成为真正的Leader；

Leader端根据同步算法进行同步操作；

而在Follower端会做如下工作：

选举完成后，尝试与Leader建立同步连接，如果一段时间没有连接上就报错超时，重新回到选举状态；

向Leader发送FOLLOWERINFO封包，带上自己最大的zxid；

根据同步算法进行同步操作；

具体使用哪种同步算法取决于Follower当前最大的zxid，在Leader端会维护最小事务idminCommittedLog和最大事务idmaxCommittedLog两个zxid，minComittedLog是没有被快照存储的日志文件的第一条(每次快照储存完，会重新生成一个事务日志文件)，maxCommittedLog是事务日志中最大的事务。Zookeeper中实现了以下数据同步算法：

直接差异化同步(DIFF同步)

仅回滚同步(TRUNC)，即删除多余的事务日志，比如原来的Leader节点宕机后又重新加入，可能存在它自己写入并提交但是别的节点还没来得及提交的数据；

先回滚(TRUNC)再差异化(DIFF)同步；

全量同步(SNAP)；

我们用peerLastZxid代表Follower端最大的事务id，结合几个具体的场景学习如何选择同步算法。

场景一：同一选举轮次下Follower从Leader同步数据

假设Leader端未被快照存储的zxid为0x500000001、0x500000002、0x500000003、0x500000004、0x500000005，此时Follower端最大已提交的zxid(即peerLastZxid)为0x500000003，因此需要把0x500000004、0x500000005同步给Follower，直接使用差异化同步(DIFF)即可，Leader端差异化同步消息的发送顺序如下：

发送顺序 数据包类型 对应的zxid

1 PROPOSAL 0x500000004

2 COMMIT 0x500000004

3 PROPOSAL 0x500000005

4 COMMIT 0x500000005

Follower端同步过程如下：

Follower端首先收到DIFF指令，进入DIFF同步阶段；

Follower收到同步的数据和提交命令，并应用到内存数据库当中；

同步完成后，Leader会发送一个NEWLEADER指令，通知Follower已经将最新的数据同步给Follower了，Follower收到NEWLEADER指令后反馈一个ack消息，表明自己已经同步完成；

单个Follower同步完成后，Leader会进入集群的"过半策略"等待状态，当有超过一半的Follower都同步完成以后，Leader会向已经完成同步的Follower发送UPTODATE指令，用于通知Follower已经完成数据同步，可以对外提供服务了，最后Follower收到Leader的UPTODATE指令后，会终止数据同步流程，向Leader再次反馈一个ack消息。

场景二：Leader宕机了，但不久之后又重新加入集群

Leader在本地提交事务完成，还没来得及把事务提交提议发送给其他节点前宕机了。

为描述方便，假设集群有三个节点，分别是A、B、C，没有宕机前Leader是B，已经发送过0x500000001和0x500000002的数据修改提议和事务提交提议，并且发送了0x500000003的数据修改提议，但在B节点发送事务提交提议之前，B宕机了，由于是本机发送，所以B的本地事务已经提交，即B最新的数据是0x500000003，但发送给A和C的事务提议失败了，A和C的最新数据依然是0x500000002，B宕机后，A和C会进行Leader选举，假设C成为新的Leader，并且进行过两次数据修改，对应的zxid为0x600000001、0x600000002，然而此时B机器恢复后加入新集群(AC)，重新进行数据同步，对B来说，peerLastZxid为0x500000003，对于当前的Leader C来说，minCommitedLog=0x500000001, maxCommittedLog=0x600000002(总共是0x500000001、0x500000002、0x600000001、0x600000002几个未被快照的事务)。这种情况下使用(TRUNC + DIFF)的同步方式，同步过程如下：

B恢复并且向已有的集群(AC)注册后，向C发起同步连接的请求;

B向Leader C发送FOLLOWERINFO包，带上Follower自己最大的zxid(0x500000003);

C发现自己没有0x500000003这个事务提交记录，就向B发送TRUNC指令，让B回滚到0x500000002；

B回滚完成后，向C发送信息包，确认完成，并说明当前的zxid为0x500000002；

C向B发送DIFF同步指令；

B收到DIFF指令后进入同步状态，并向C发送ACK确认包；

C陆续把对应的差异数据修改提议和Commit提议发给B，当数据发送完成后，再发送通知包给B；

B将数据修改提议应用于内存数据结构并Commit，(当集群中过半机器同步完成)当收到C通知已经同步完成后，B给回应ACK，并且结束同步；

这种情况仍然是minCommitedLog < peerLastZxid < maxCommittedLog的情况。

场景三：节点宕机并且很久之后才重新加入集群

当集群中某个节点宕机时间过长，在恢复并且加入集群时，集群中数据的事务日志文件已经生成多个，此时minCommittedLog比该节点宕机时的最大zxid还要大。例如假设ABC集群中B宕机，几天后才恢复，此时minCommittedLog为0x6000008731，而peerLastZxid为0x500000003，这种情况下采用全量同步(SNAP)的方式，同步过程如下：

当Leader C发现B的zxid小于minCommittedLog时，向B发送SNAP指令；

B收到同步指令，进入同步阶段；

Leader C会从内存数据库中获取全量的数据发送给B；

B获取数据处理完成后，C还会把全量同步期间产生的最新的数据修改提议和Commit提议以增量(DIFF)的方式发送给B；

**6. Zookeeper广播流程**

当zk集群选举完成，并且数据同步结束后即可开始对外提供服务，接收读写请求，当Leader

接收到客户端新的事物请求后，会向集群的Follower广播该事物请求，广播流程如下：

Leader首先会根据客户端的事务请求生成对应的事务修改提议，并根据zxid的顺序(收到多个客户端事务请求)向所有的Follower发送数据修改提议；

当Follower收到Leader的数据修改提议后，会根据接收的先后顺序处理这些提议，即如果收到了1、2、3三条数据修改提议，如果处理完成了第三条，则代表1、2条一定已经处理成功；

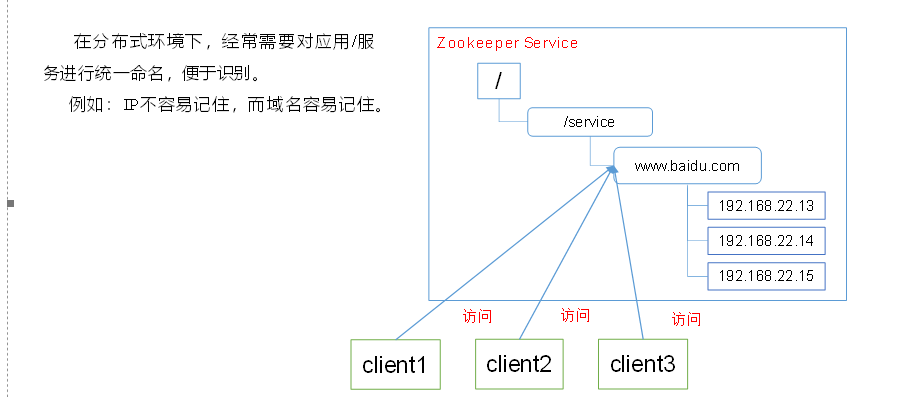
Leader收到Follower针对某个数据修改提议过半的正确反馈(ack)后，发起对该事务修改提议的提交，即重新发起一个事务提交的提议；

Follower收到事务提交的提议后，记录事务提交，并把数据更新到内存数据库；

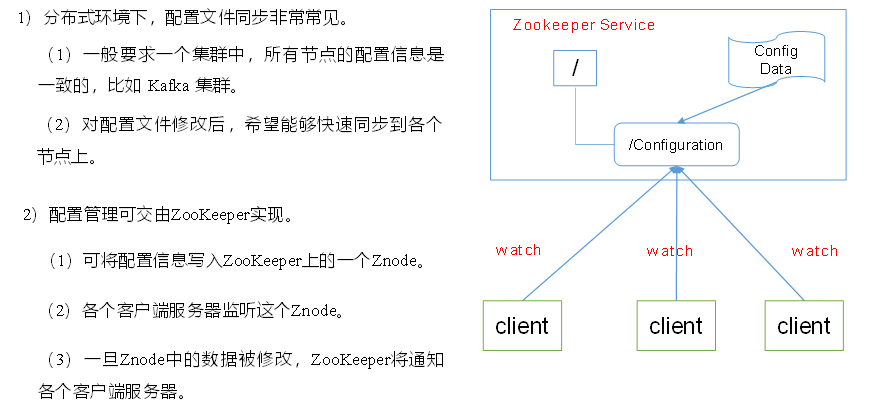
**8.zookeeper的应用场景**

提供的服务包括：统一命名服务、数据订阅与发布（统一配置管理）、统一集群管理、服务器节点动态上下线、软负载均衡，分布式锁，队列等。

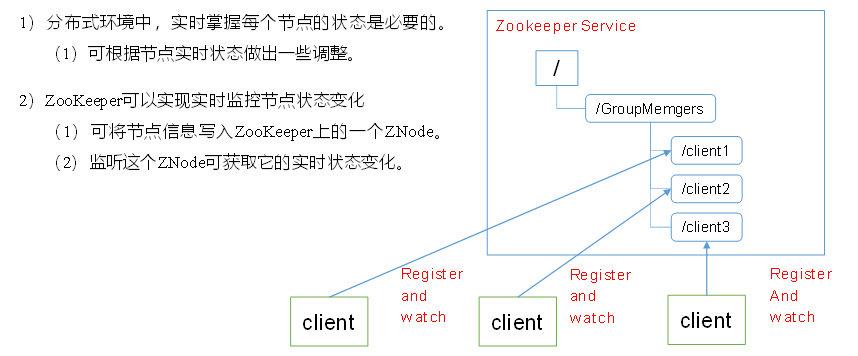
**统一命名服务**



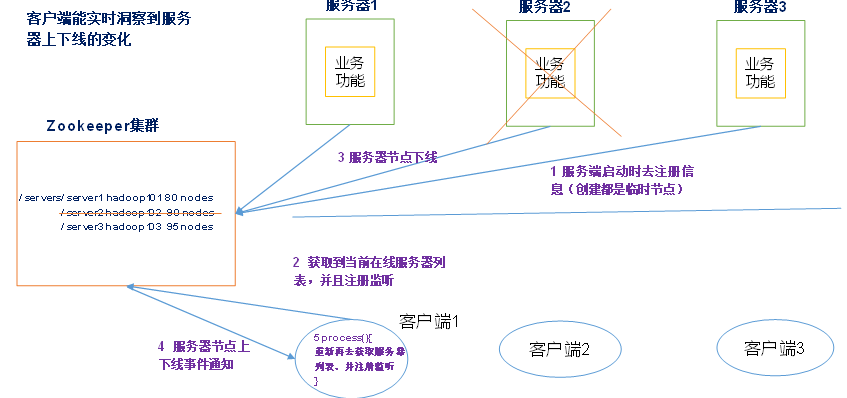
统一配置管理



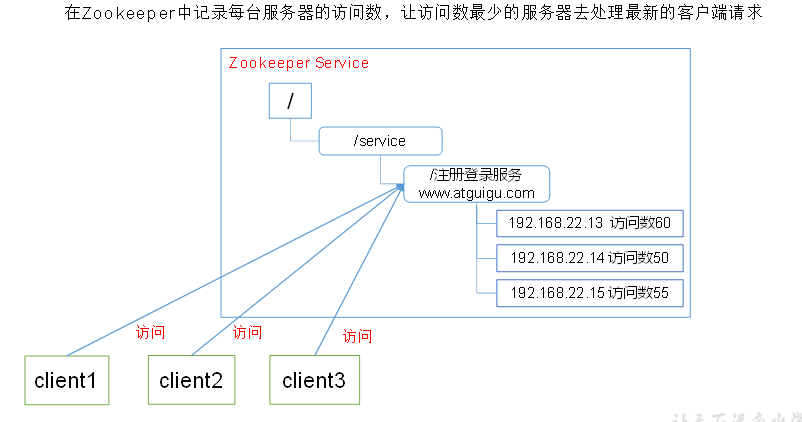
**统一集群管理**



**服务器节点动态上下线**



**软负载均衡**



**分布式锁**

方式一：事务获取锁的时候再zookeeper上创建一个临时节点，完成事务删除节点，其他事物watch该节点

缺点：会产生惊群，一个事务完成其他所有事物全发生监听。也是不公平的锁

方式二： 我们将锁抽象成目录，多个线程在此目录下创建瞬时的顺序节点，因为Zk会为我们保证节点的顺序性，所以可以利用节点的顺序进行锁的判断。

 首先创建顺序节点，然后获取当前目录下最小的节点，判断最小节点是不是当前节点，如果是那么获取锁成功，如果不是那么获取锁失败。

 获取锁失败的节点获取当前节点上一个顺序节点，对此节点注册监听，当节点删除的时候通知当前节点。

 当unlock的时候删除节点之后会通知下一个节点。

实现了公平锁

共享锁：先创建一个节点，在该节点下创建顺序节点

读锁：新建节点R开头，其他线程创建读锁，只要前面是R开头就行。

**写锁：新建节点W开头，前面有W或者R开头就等待。**

**队列**

使用 ZooKeeper 实现 FIFO 队列，入队操作就是在 queue\_fifo 下创建自增序的子节点，并把数据（队列大小）放入节点内。出队操作就是先找到 queue\_fifo 下序号最下的那个节点，取出数据，然后删除此节点。