Zookeeper

*2020-4-28 ——*https://www.bilibili.com/video/BV1M741137qY?p=74

https://zookeeper.apache.org/

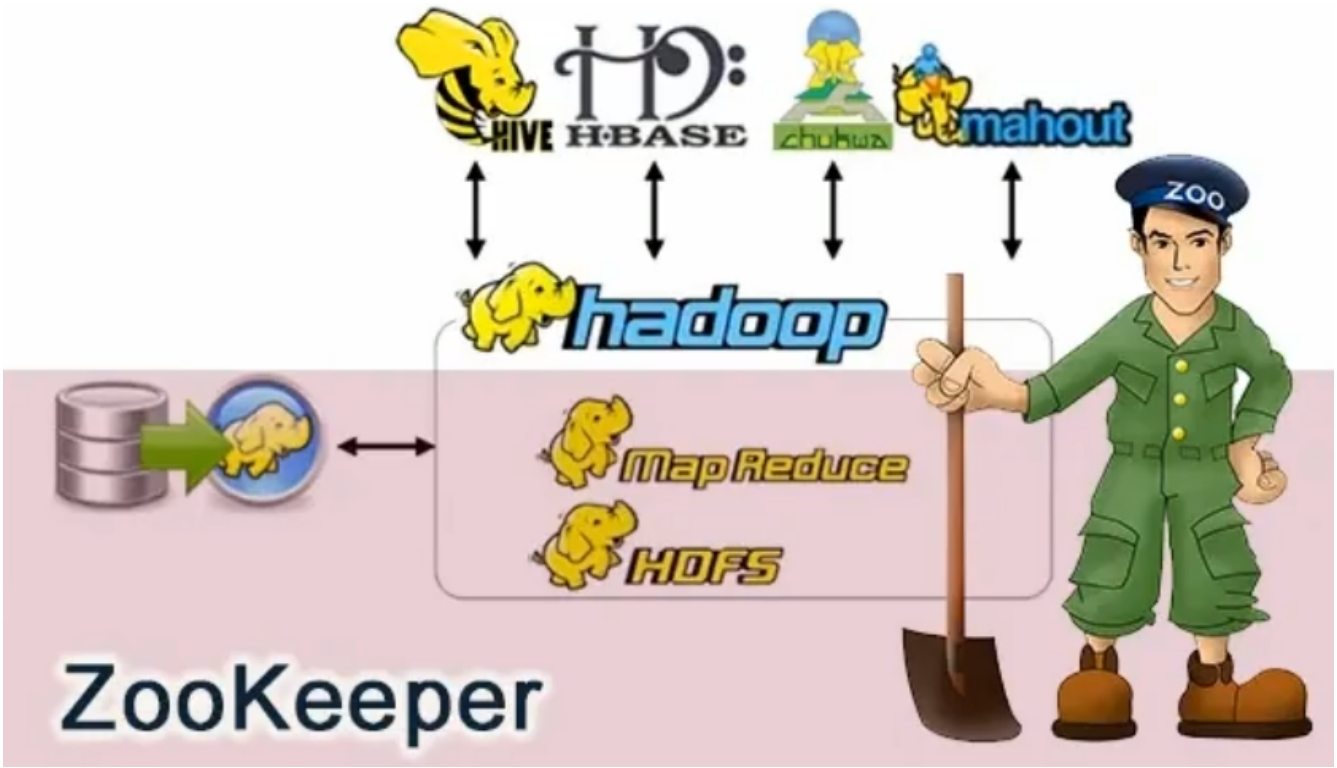
简介/快速入门

```
ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.
```

ZooKeeper是一个集中的服务，用于维护配置信息、命名、提供分布式同步和提供组服务。所有这些类型的服务都以某种形式被分布式应用程序使用。每次它们被实现时，都会有大量的工作来修复不可避免的错误和竞争条件。由于实现这些服务的困难，应用程序最初通常会略过这些服务，这使得它们在出现更改时变得脆弱，并且难以管理。即使正确地执行了这些服务，在部署应用程序时，这些服务的不同实现也会导致管理复杂性——官网原话翻译

zookeeper由雅虎研究院开发,是Google Chubby的开源实现,后来托管到 Apache,于2010年11月正式成为apache的顶级项目

大数据生态系统里由很多组件的命名都是某些动物或者昆虫，比如hadoop大象，hive就是蜂巢，zookeeper即管理员，故名肆意就算管理大数据生态系统各组件的管理员，如下所示：



应用场景

zookeepepr是一个经典的**分布式**数据致性解决方案，致力于为分布式应用提供一个高性能、高可用,且具有严格顺序访问控制能力的分布式协调存储服务。

- 维护配置信息
- 分布式锁服务
- 集群管理
- 生成分布式唯一ID

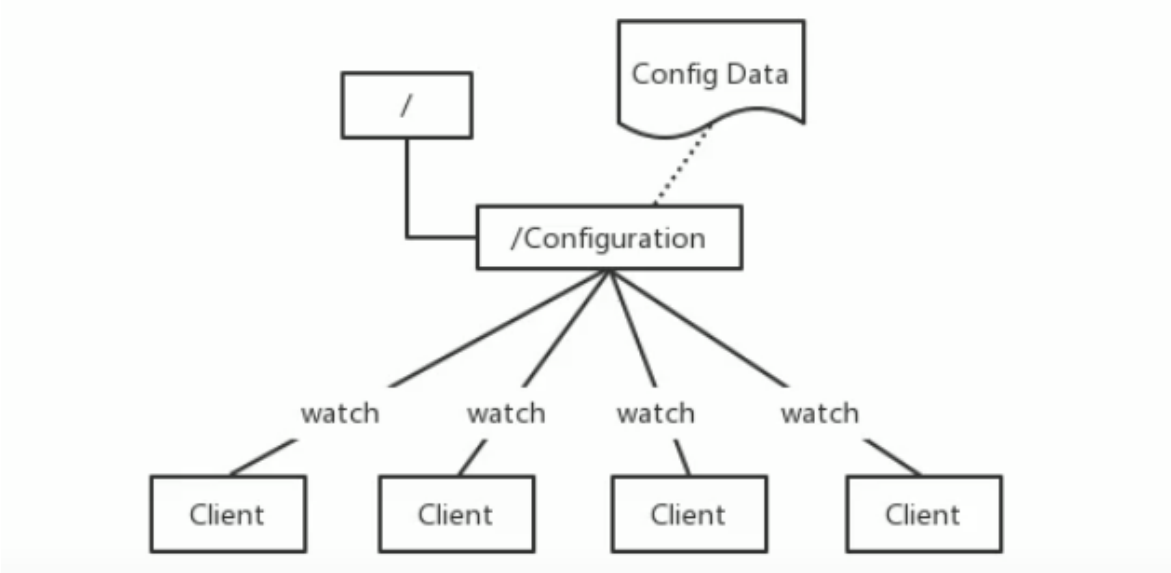
1. **维护配置信息**

    - java编程经常会遇到配置项，比如数据库的url、 schema、user和 password等。通第这些配置项我们会放置在配置文件中，再将配置文件放置在服务器上当需要更改配置项时，需要去服务器上修改对应的配置文件。

      但是随着分布式系统的兴起,由于许多服务都需要使用到该配置文件,因此有**必须保证该配置服务的高可用性**(highavailability)和各台服务器上配置数据的一致性。

      通常会将配置文件部署在一个集群上，然而一个**集群动辄上千台**服务器，此时如果再一台台服务器逐个修改配置文件那将是非常繁琐且危险的的操作，因此就**需要一种服务**，**能够高效快速且可靠地完成配置项的更改等操作**，并能够保证各配置项在每台服务器上的数据一致性。

      zookeeper**就可以提供这样一种服务**，其使用Zab这种一致性协议来保证一致性。现在有很多开源项目使用zookeeper来维护配置，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 brokers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。

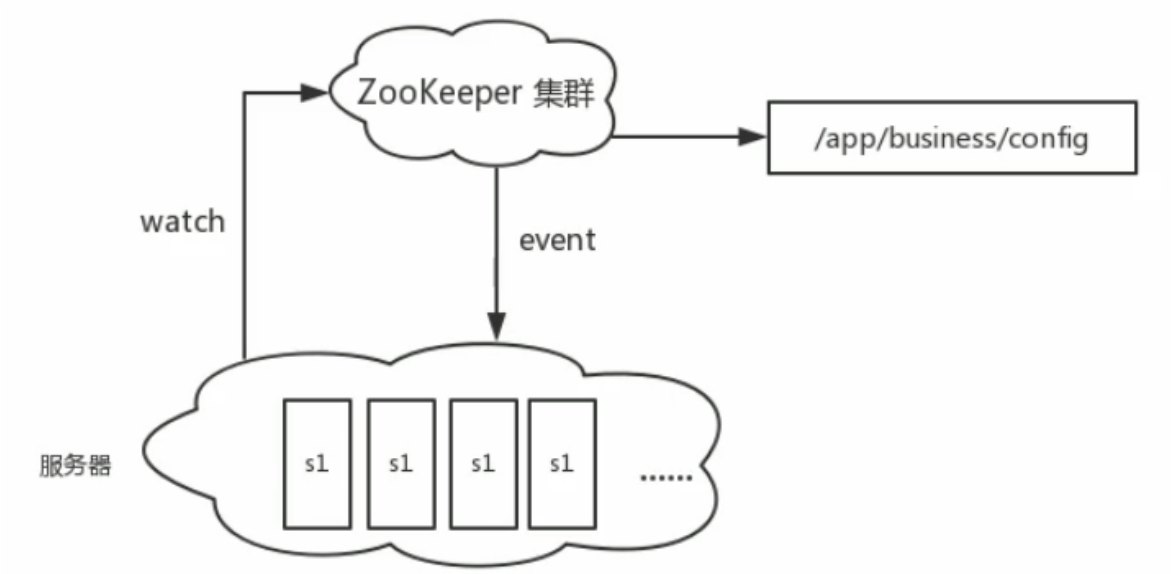
      

2. 分布式锁服务

    - 一个集群是一个分布式系统，由多台服务器组成。为了提高并发度和可靠性，多台服务器上运行着同一种服务。当多个服务在运行时就需要协调各服务的进度，有时候需要保证当某个服务在进行某个操作时，其他的服务都不能进行该操作，即对该操作进行加锁，如果当前机器挂掉后，释放锁并 fail over到其他的机器继续执行该服务

3. 集群管理

    - 一个集群有时会因为各种软硬件故障或者网络故障，出现棊些服务器挂掉而被移除集群，而某些服务器加入到集群中的情况，zookeeper会将这些服务器加入/移出的情况通知给集群中的其他正常工作的服务器，以及时调整存储和计算等任务的分配和执行等。此外zookeeper还会对故障的服务器做出诊断并尝试修复。



4. 生产分布式唯一ID
    - 在过去的单库单表型系统中，通常可以使用数据库字段自带的auto_ increment属性来自动为每条记录生成一个唯一的ID。但是分库分表后，就无法在依靠数据库的auto_ Increment属性来唯一标识一条记录了。此时我们就可以用zookeeper在分布式环境下生成全局唯一ID。

      做法如下:每次要生成一个新id时，创建一个持久顺序节点，创建操作返回的节点序号，即为新id，然后把比自己节点小的删除即可

Zookeeper的设计目标

zooKeeper致力于为分布式应用提供一个高性能、高可用，且具有严格顺序访问控制能力的分布式协调服务

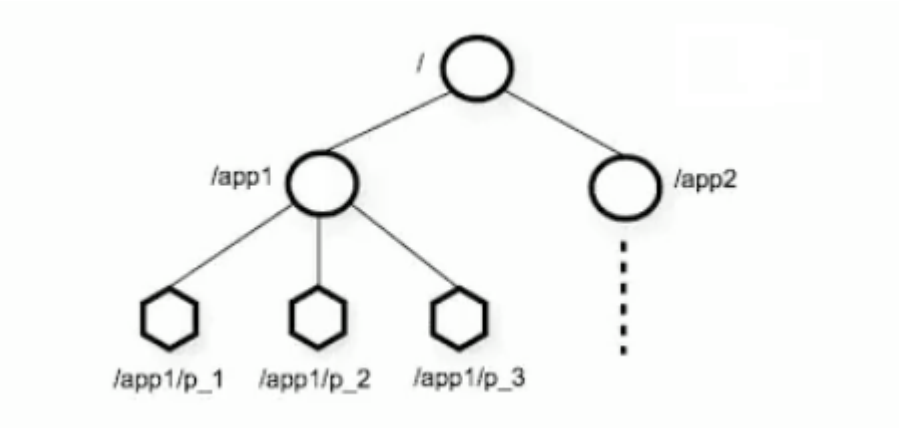
1. 高性能
    - zookeeper将全量数据存储在**内存**中，并直接服务于客户端的所有非事务请求，尤其用于以读为主的应用场景
2. 高可用
    - zookeeper一般以集群的方式对外提供服务，一般3~5台机器就可以组成一个可用的 zookeeper集群了，每台机器都会在内存中维护当前的服务器状态，并且每台机器之间都相互保持着通信。只要集群中超过一半的机器都能够正常工作，那么整个集群就能够正常对外服务
3. 严格顺序访问
    - 对于来自客户端的每个更新请求，zookeeper都会分配一个全局唯一的递增编号，这个编号反应了所有事务操作的先后顺序

数据模型

zookeeper的数据结点可以视为树状结构(或目录)，树中的各个结点被称为znode(即zookeeper node)，一个znode可以由多个子结点。zookeeper结点在结构上表现为树状；

使用路径path来定位某个znode，比如/ns-1/itcast/mysqml/schemal1/table1，此处ns-1，itcast、mysql、schemal1、table1分别是根结点、2级结点、3级结点以及4级结点；其中ns-1是itcast的父结点，itcast是是ns-1的子结点，itcast是mysql的父结点....以此类推

znode，间距文件和目录两种特点，即像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分



那么如何描述一个znode呢？一个znode大体上分为3个部分：

- 结点的数据：即znode data(结点path，结点data)的关系就像是Java map中的 key value关系
- 结点的子结点children
- 结点的状态stat：用来描述当前结点的创建、修改记录，包括cZxid、ctime等

结点状态stat的属性

在zookeeper shell中使用 get命令查看指定路径结点的data、stat信息

```
[zk: localhost:2181(CONNECTED) 7] get /ns-1/tenant

cZxid = 0x6a0000000a
ctime = Wed Mar 27 09:56:44 CST 2019
mZxid = 0x6a0000000a
mtime = Wed Mar 27 09:56:44 CST 2019
pZxid = 0x6a0000000e
cversion = 2
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 2
```

属性说明：

结点的各个属性如下。其中重要的概念是Zxid(Zookeeper Transaction ID)，Zookeeper结点的每一次更改都具有唯一的Zxid，如果Zxid-1 小于Zxid-2 ，则Zxid-1 的更改发生在 Zxid-2更改之前

https://zookeeper.apache.org/doc/r3.4.14/zookeeperProgrammers.html#sc_zkDataModel_znodes

- cZxid数据结点创建时的事务ID——针对于zookeeper数据结点的管理：我们对结点数据的一些写操作都会导致zookeeper自动地为我们去开启一个事务，并且自动地去为每一个事务维护一个事务ID
- ctime数据结点创建时的时间
- mZxid数据结点最后一次更新时的事务ID
- mtime数据结点最后一次更新时的时间
- pZxid数据结点的最后一个子节点创建时的事务ID
- cversion子结点的更改次数
- dataVersion结点数据的更改次数
- aclVersion结点的ACL更改次数——类似linux的权限列表，维护的是当前结点的权限列表被修改的次数

- ephemeralOwner如果结点是临时结点，则表示创建该结点的会话的SessionID；如果是持久结点，该属性值为0
- dataLength数据内容的长度
- numChildren数据结点当前的子结点个数

结点类型

zookeeper中的结点有两种，分别为**临时结点**和**永久结点**。结点的类型在创建时被确定，并且不能改变

- 临时节点：
  - 该节点的生命周期依赖于创建它们的会话。一旦会话(Session）结束，临时节点将被自动删除，当然可以也可以手动删除。虽然每个临时的 Znode都会绑定到一个客户端会话，但他们对所有的客户端还是可见的。另外，Zookeeper的临时节点不允许拥有子节点
- 持久化结点：
  - 该结点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，它们才能被删除

单机安装

测试系统环境centos7.3

zookeeper:zookeeper-3.4.10.tar.gz

jdk:jdk-8u131-linux-x64.tar.gz

http://archive.apache.org/dist/zookeeper/

1. 在centos中使用 root用户创建 zookeeper用户，用户名:zookeeper密码:zookeeper

```
useradd zookeeper
passwd zookeeper
su zookeeper
```

2. zookeeper底层依赖于jdk，zookeeper用户登录后，根目录下先进行jdk 的安装，jdk使用 jdk-8u131-linux-x64.tar.gz

```
tar -zxf tar.gz
```

3. 配置jdk 环境变量

```
vi /etc/profile
JAVA_HOME=/home/zookeeper/jdk1.8.0_131
export JAVA_HOME

PATH=$JAVA_HOME/bin:$PATH
export PATH

souce /etc/profile
```

4. 检测jdk安装
   - java -version // 如果反馈了Java信息，则成功
5. zookeeper 上传解压
   - tar -zxf tar.gz
6. 为zookeeper准备配置文件

```
# 进入conf目录
cd /home/zookeeper/zookeeper-3.4.10/conf
# 复制配置文件
cp zoo_sampe.cfg zoo.cfg
# zookeeper 根目录下创建data目录
mkdir data
# vi 配置文件中的dataDir
# 此路径用于存储zookeeper中数据的内存快照、及事务日志文件，虽然zookeeper是使用内存的，但是需要持久化一些数据来保证数据的安全，和redis一样
dataDir=/home/zookeeper/zookeeper-3.4.10/data
```

7. 启动zookeeper

```
# 进入zookeeper的bin目录
cd /home/zookeeper/zookeeper-3.4.10/bin
# 启动zookeeper
./zkServer.sh start

# 启动: zkServer.sh start
# 停止: zkServer.sh stop
# 查看状态: zkServer.sh status

# 进入zookeeper 内部
./zkCli.sh
```

常用shell命令

zookeeper——getting started——https://zookeeper.apache.org/doc/r3.4.14/zookeeperStarted.html#sc_FileManagement

操作结点

查询

get /hadoop 查看结点的数据和属性 stat /hadoop 查看结点的属性

创建

创建结点并写入数据：

create [-s] [-e] path data # 其中 -s 为有序结点，-e 临时结点（默认是持久结点）

```
create /hadoop "123456"  # 此时，如果quit退出后再./zkCient.sh 登入
                         # 再用输入 get /hadoop 获取，结点依然存在(永久结点)

create -s /a "a"         # 创建一个持久化有序结点，创建的时候可以观察到返回的数据带上了一个id
create -s /b "b"         # 返回的值，id递增了

create -s -e /aa "aa"    # 依然还会返回自增的id，quit后再进来，继续创建，id依然是往后推的

create /aa/xx            # 继续创建结点，可以看到pZxid变化了
```

更新

更新结点的命令是set，可以直接进行修改，如下：

set path [version]

```
set /hadoop "345"          # 修改结点值

set /hadoop "hadoop-x" 1 # 也可以基于版本号进行更改，类似于乐观锁，当传入版本号(dataVersion)
                           # 和当前结点的数据版本号不一致时，zookeeper会拒绝本次修改
```

删除

删除结点的语法如下：

delete path [version] 和 set 方法相似，也可以传入版本号

```
delete /hadoop             # 删除结点
delete /hadoop 1           # 乐观锁机制，与set 方法一致
```

要想删除某个结点及其所有后代结点，可以使用递归删除，命令为 rmr path

查看结点列表

```
ls /hadoop                 # 可以查看结点的列表
ls2 /hadoop                # 可以查看结点的列表以及目标结点的信息

ls /                       # 根节点
```

监听器get path [watch] | stat path [watch]

使用get path [watch] 注册的监听器能够在结点**内容发生改变**的时候，向客户端发出通知。需要注意的是zookeeper的触发器是一次性的(One-time trigger)，即触发一次后就会立即失效

```
get /hadoop watch          # get 的时候添加监听器，当值改变的时候，监听器返回消息
set /hadoop 45678          # 测试
```

ls\ls2 path [watch]

使用 ls path [watch] 或 ls2 path [watch]注册的监听器能够监听该结点下**所有子节点**的**增加**和**删除**操作

```
ls /hadoop watch           # 添加监听器
set /hadoop/node "node"
```

zookeeper的Acl权限控制

https://zookeeper.apache.org/doc/r3.4.14/zookeeperProgrammers.html#sc_ZooKeeperAccessControl

zookeeper类似文件系统，client可以创建结点、更新结点、删除结点，那么如何做到结点的权限控制呢?

zookeeper的 access control list 访问控制列表可以做到这一点

acl权限控制，使用scheme：id：permission来标识，主要涵盖3个方面：

https://zookeeper.apache.org/doc/r3.4.14/zookeeperProgrammers.html#sc_BuiltinACLSchemes

- 权限模式(scheme)：授权的策略
- 授权对象(id)：授权的对象
- 权限(permission)：授予的权限

其特性如下：

- zookeeper的权限控制是基于每个znode结点的，需要对每个结点设置权限
- 每个znode支持多种权限控制方案和多个权限
- 子结点不会继承父结点的权限，客户端无权访问某结点，但可能可以访问它的子结点：

  例如`setAcl /test2 ip:192.168.133.133:crwda` // 将结点权限设置为Ip：192.168.133.133 的客户端可以对节点进行 增删改查和管理权限

权限模式

- 采用何种方式授权

方案	描述
world	只有一个用户：anyone，代表登录zookeeper所有人(默认)
ip	对客户端使用IP地址认证
auth	使用已添加认证的用户认证
digest	使用"用户名：密码"方式认证

授权对象

- 给谁授予权限
- 授权对象ID是指，权限赋予的实体，例如：IP地址或用户

授权的权限

- 授予什么权限
- create、delete、read、writer、admin也就是 增、删、查、改、管理权限，这5种权限简写为 c d r w a，注意： 这五种权限中，有的权限并不是对结点自身操作的例如：delete是指对**子结点**的删除权限

  可以试图删除父结点，但是子结点必须删除干净，所以delete的权限也是很有用的

权限	ACL简写	描述
create	c	可以创建子结点
delete	d	可以删除子结点(仅下一级结点)
read	r	可以读取结点数据以及显示子结点列表
write	w	可以设置结点数据
admin	a	可以设置结点访问控制权限列表

授权的相关命令

命令	使用方式	描述
getAcl	getAcl	读取ACL权限
setAcl	setAcl	设置ACL权限
addauth	addauth	添加认证用户

案例/远程登录

`./zkServer.sh -server 192.168.133.133` 可以远程登录

world权限模式

- `getAcl /node` // 读取权限信息
- `setAcl /node world:anyone:drwa` // 设置权限(禁用创建子结点的权限)

ip模式

./zkServer.sh -server 192.168.133.133 可以远程登录

- setAcl /hadoop ip:192.168.133.133:drwa
- 如果在两台不同的虚拟机中，另一台用远程连接的模式，进行上面这条命令，那么只会有一台被授权
- 需要两台虚拟机一起授权的话需要用**逗号**将授权列表隔开：setAcl /hadoop ip:192.168.133.133:cdrwa,ip:192.168.133.132:cdrwa

auth认证用户模式

addauth digest <user>:<password>

setAcl <path> auth:<user>:<acl>

```
create /hadoop "hadoop"            # 初始化测试用的结点
addauth digest itcast:123456       # 添加认证用户
setAcl /hadoop auth:itcast:cdrwa   # 设置认证用户
quit                               # 退出后再./zkCli.sh 进入
get /hadoop                        # 这个时候就没有权限了，需要再次认证
addauth digest itcast:123456       # 认证，密码错了的话 zookeeper 不会报错，但是不能认证
get /hadoop
```

Digest授权模式

setAcl <path> digest:<user>:<password>:<acl>

- 这里的密码是经过SHA1以及BASE64处理的密文，在shell 中可以通过以下命令计算：

```
echo -n <user>:<password> | openssl dgst -binary -sha1 | openssl base64
```

```
# 计算密码
echo -n itcast:12345 | openssl dgst -binary -sha1 | openssl base64
# 获取密码，设置权限列表
setAcl /hadoop digest:itcast:qUFSHxJjItUW/93UHFXFVGlvryY=:cdrwa
# 现在想要get /hadoop 需要登录了
addauth digest itcast:12345
get /hadoop
```

多种授权模式

```
setAcl /hadoop
ip:192.168.133.132:cdrwa,auth:hadoop:cdrwa,digest:itcast:673OfZhUE8JEFMcuOl64qI8e5ek=:cdrwa
```

acl 超级管理员

- zookeeper的权限管理模式有一种叫做super，该模式提供一个超管，可以方便的访问任何权限的节点

  假设这个超管是supper:admin，需要为超管生产密码的密文

```
echo -n super:admin | openssl dgst -binary -sha1 | openssl base64
```

- 那么打开zookeeper目录下/bin/zkServer.sh服务器脚本文件，找到如下一行：

```
 /nohup # 快速查找，可以看到如下
 nohup "$JAVA" "-Dzookeeper.log.dir=${ZOO_LOG_DIR}" "-
Dzookeeper.root.logger=${ZOO_LOG4J_PROP}"
```

- 这个就算脚本中启动zookeeper的命令，默认只有以上两个配置项，我们需要添加一个超管的配置项

```
"-
Dzookeeper.DigestAuthenticationProvider.superDigest=super:xQJmxLMiHGwaqBvst5y6rkB6HQs="
```

- 修改后命令变成如下

```
nohup "$JAVA" "-Dzookeeper.log.dir=${ZOO_LOG_DIR}" "-Dzookeeper.root.logger=${ZOO_LOG4J_PROP}" "-Dzookeeper.DigestAuthenticationProvider.superDigest=super:xQJmxLMiHGwaqBvst5y6rkB6HQs="
```

```
# 重起后，现在随便对任意节点添加权限限制
setAcl /hadoop ip:192.168.1.1:cdrwa
# 现在当前用户没有权限了
getAcl /hadoop
# 登录超管
addauth digest super:admin
# 强行操作节点
get /hadoop
```

zookeeper的 JavaAPI

```
<dependency>
            <groupId>com.101tec</groupId>
            <artifactId>zkclient</artifactId>
            <exclusions>
                <exclusion>
                    <artifactId>zookeeper</artifactId>
                    <groupId>org.apache.zookeeper</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>log4j</artifactId>
                    <groupId>log4j</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>slf4j-log4j12</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>slf4j-api</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
            <version>0.9</version>
        </dependency>
        <dependency>
            <artifactId>zookeeper</artifactId>
            <exclusions>
                <exclusion>
                    <artifactId>log4j</artifactId>
                    <groupId>log4j</groupId>
                </exclusion>
                <exclusion>
```

```
                    <artifactId>slf4j-log4j12</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
            <groupId>org.apache.zookeeper</groupId>
            <version>3.4.10</version>
        </dependency>
```

zonde是 zookeeper集合的核心组件，zookeeper API 提供了一小组使用 zookeeper集群来操作znode的所有细节

客户端应该遵循以下步骤，与zookeeper服务器进行清晰和干净的交互

- 连接到zookeeper服务器。zookeeper服务器为客户端分配会话ID
- 定期向服务器发送心跳。否则，zookeeper服务器将过期会话ID，客户端需要重新连接
- 只要会话Id处于活动状态，就可以获取/设置znode
- 所有任务完成后，断开与zookeeper服务器连接，如果客户端长时间不活动，则zookeeper服务器将自动断开客户端

连接到Zookeeper

这部分，官网的解释十分稀少https://zookeeper.apache.org/doc/r3.4.14/zookeeperStarted.html#sc_ConnectingToZooKeeper

```
[zkshell: 0] help
ZooKeeper host:port cmd args
    get path [watch]
    ls path [watch]
    set path data [version]
    delquota [-n|-b] path
    quit
    printwatches on|off
    create path data acl
    stat path [watch]
    listquota path
    history
    setAcl path acl
    getAcl path
    sync path
    redo cmdno
    addauth scheme auth
    delete path [version]
    deleteall path
    setquota -n|-b val path
```

```
Zookeeper(String connectionString, int sessionTimeout, watcher watcher)
```

- connectionString - zookeeper主机
- sessionTimeout- 会话超时
- watcher - 实现"监听器" 对象。zookeeper集合通过监视器对象返回连接状态

```java
public static void main(String[] args) throws IOException, InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);

        ZooKeeper zookeeper = new ZooKeeper("192.168.133.133:2181", 5000, (WatchedEvent x) -> {
            if (x.getState() == Watcher.Event.KeeperState.SyncConnected) {
                System.out.println("连接成功");
                countDownLatch.countDown();
            }
        });
        countDownLatch.await();
        System.out.println(zookeeper.getSessionId());
        zookeeper.close();
    }
```

新增节点

```java
// 同步
create(String path, byte[] data, List<ACL> acl, CreateMode createMode)
// 异步
create(String path, byte[] data, List<ACL> acl, CreateMode createMode,
      AsynCallback.StringCallback callBack, Object ctx)
```

参数	解释
path	znode 路径
data	数据
acl	要创建的节点的访问控制列表。zookeeper API提供了一个静态接口 ZooDefs.Ids 来获取一些基本的acl列表。例如，ZooDefs.Ids.OPEN_ACL_UNSAFE返回打开znode的acl列表
createMode	节点的类型，这是一个枚举
callBack	异步回调接口
ctx	传递上下文参数

示例:

```java
// 枚举的方式
    public static void createTest1() throws Exception{
        String str = "node";
        String s = zookeeper.create("/node", str.getBytes(),
                ZooDefs.Ids.READ_ACL_UNSAFE, CreateMode.PERSISTENT);
        System.out.println(s);
    }
```

```java
// 自定义的方式
    public static void createTest2() throws Exception{
        ArrayList<ACL> acls = new ArrayList<>();
        Id id = new Id("ip","192.168.133.133");
        acls.add(new ACL(ZooDefs.Perms.ALL,id));

 zookeeper.create("/create/node4","node4".getBytes(),acls,CreateMode.PERSISTENT);
    }
```

```java
// auth
    public static void createTest3() throws  Exception{
        zookeeper.addAuthInfo("digest","itcast:12345".getBytes());
        zookeeper.create("/node5","node5".getBytes(),
                ZooDefs.Ids.CREATOR_ALL_ACL,CreateMode.PERSISTENT);
    }
// 自定义的方式
```

```
    public static void createTest3() throws  Exception{
//        zookeeper.addAuthInfo("digest","itcast:12345".getBytes());
//        zookeeper.create("/node5","node5".getBytes(),
//                ZooDefs.Ids.CREATOR_ALL_ACL,CreateMode.PERSISTENT);
        zookeeper.addAuthInfo("digest","itcast:12345".getBytes());
        List<ACL> acls = new ArrayList<>();
        Id id = new Id("auth","itcast");
        acls.add(new ACL(ZooDefs.Perms.READ,id));
        zookeeper.create("/create/node6","node6".getBytes(),
                acls,CreateMode.PERSISTENT);
    }
```

```
// digest
public static void createTest3() throws  Exception{
    List<ACL> acls = new ArrayList<>();
    Id id = new Id("digest","itcast:qUFSHxJjItUW/93UHFXFVGlvryY=");
    acls.add(new ACL(ZooDefs.Perms.READ,id));
    zookeeper.create("/create/node7","node7".getBytes(),
                     acls,CreateMode.PERSISTENT);
}
```

```
// 异步
    public static void createTest4() throws  Exception{
        zookeeper.create("/node12", "node12".getBytes(),
ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT, new
AsyncCallback.StringCallback(){
            /**
             * @param rc 状态，0 则为成功，以下的所有示例都是如此
             * @param path 路径
             * @param ctx 上下文参数
             * @param name 路径
             */
            public void processResult(int rc, String path, Object ctx, String
name){
                System.out.println(rc + " " + path + " " + name +  " " + ctx);
            }
        }, "I am context");
        TimeUnit.SECONDS.sleep(1);
        System.out.println("结束");
    }
```

修改节点

同样也有两种修改方式(异步和同步)

```
// 同步
setData(String path, byte[] data, int version)
// 异步
setData(String path, byte[] data, int version, StatCallback callBack, Object ctx)
```

参数	解释
path	节点路径
data	数据
version	数据的版本号，-1代表不使用版本号，乐观锁机制
callBack	异步回调 AsyncCallback.StatCallback，和之前的回调方法参数不同，这个可以获取节点状态
ctx	传递上下文参数

```
    public static void setData1() throws Exception{
        // arg1:节点的路径
        // arg2:修改的数据
        // arg3:数据的版本号 -1 代表版本号不参与更新
        Stat stat = zookeeper.setData("/hadoop","hadoop-1".getBytes(),-1);
    }
```

- ```
    public static void setData2() throws Exception{
        zookeeper.setData("/hadoop", "hadoop-1".getBytes(), 3 ,new AsyncCallback.StatCallback(){
            @Override
            public void processResult(int rc, String path, Object ctx, Stat stat) {
                // 讲道理，要判空
                System.out.println(rc + " " + path + " " + stat.getVersion() +  " " + ctx);
            }
        }, "I am context");
    }
  ```

删除节点

异步、同步

- ```
  // 同步
  delete(String path, int version)
  // 异步
  delete(String path, int version, AsyncCallback.VoidCallback callBack, Object ctx)
  ```

参数	解释
path	节点路径
version	版本
callBack	数据的版本号，-1代表不使用版本号，乐观锁机制
ctx	传递上下文参数

- ```
    public static void deleteData1() throws Exception {
        zookeeper.delete("/hadoop", 1);
    }

    public static void deleteData2() throws Exception {
        zookeeper.delete("/hadoop", 1, new AsyncCallback.VoidCallback() {
            @Override
            public void processResult(int rc, String path, Object ctx) {
                System.out.println(rc + " " + path + " " + ctx);
            }
        }, "I am context");
        TimeUnit.SECONDS.sleep(1);
    }
  ```

查看节点

同步、异步
  ```

```
// 同步
getData(String path, boolean watch, Stat stat)
getData(String path, Watcher watcher, Stat stat)
// 异步
getData(String path, boolean watch, DataCallback callBack, Object ctx)
getData(String path, Watcher watcher, DataCallback callBack, Object ctx)
```

参数	解释
path	节点路径
boolean	是否使用连接对象中注册的监听器
stat	元数据
callBack	异步回调接口，可以获得状态和数据
ctx	传递上下文参数

```
    public static void getData1() throws Exception {
        Stat stat = new Stat();
        byte[] data = zookeeper.getData("/hadoop", false, stat);
        System.out.println(new String(data));
        // 判空
        System.out.println(stat.getCtime());
    }

    public static void getData2() throws Exception {
        zookeeper.getData("/hadoop", false, new AsyncCallback.DataCallback() {
            @Override
            public void processResult(int rc, String path, Object ctx, byte[]
bytes, Stat stat) {
                // 判空
                System.out.println(rc + " " + path
                                   + " " + ctx + " " + new String(bytes) + " " +
                                   stat.getCzxid());
            }
        }, "I am context");
        TimeUnit.SECONDS.sleep(3);
    }
```

查看子节点

同步、异步

```
// 同步
getChildren(String path, boolean watch)
getChildren(String path, Watcher watcher)
getChildren(String path, boolean watch, Stat stat)
getChildren(String path, Watcher watcher, Stat stat)
// 异步
getChildren(String path, boolean watch, ChildrenCallback callBack, Object ctx)
getChildren(String path, Watcher watcher, ChildrenCallback callBack, Object ctx)
getChildren(String path, Watcher watcher, Children2Callback callBack, Object ctx)

getChildren(String path, boolean watch, Children2Callback callBack, Object ctx)
```

参数	解释
path	节点路径
boolean	

参数	解释
callBack	异步回调，可以获取节点列表
ctx	传递上下文参数

```
    public static void getChildren_1() throws Exception{
        List<String> hadoop = zookeeper.getChildren("/hadoop", false);
        hadoop.forEach(System.out::println);
    }

    public static void getChildren_2() throws Exception {
        zookeeper.getChildren("/hadoop", false, new
AsyncCallback.ChildrenCallback() {
            @Override
            public void processResult(int rc, String path, Object ctx,
List<String> list) {
                list.forEach(System.out::println);
                System.out.println(rc + " " + path + " " + ctx);
            }
        }, "I am children");
        TimeUnit.SECONDS.sleep(3);
    }
```

检查节点是否存在

同步、异步

```
// 同步
exists(String path, boolean watch)
exists(String path, Watcher watcher)
// 异步
exists(String path, boolean watch, StatCallback cb, Object ctx)
exists(String path, Watcher watcher, StatCallback cb, Object ctx)
```

参数	解释
path	节点路径
boolean	
callBack	异步回调，可以获取节点列表
ctx	传递上下文参数

```
public static void exists1() throws Exception{
    Stat exists = zookeeper.exists("/hadoopx", false);
    // 判空
    System.out.println(exists.getVersion() + "成功");
}
public static void exists2() throws Exception{
    zookeeper.exists("/hadoopx", false, new AsyncCallback.StatCallback() {
        @Override
        public void processResult(int rc, String path, Object ctx, Stat stat) {
            // 判空
            System.out.println(rc + " " + path + " " + ctx +" " +
stat.getVersion());
        }
    }, "I am children");
    TimeUnit.SECONDS.sleep(1);
}
```

事件监听机制

watcher概念

https://zookeeper.apache.org/doc/r3.4.14/zookeeperProgrammers.html#sc_WatchRememberThese

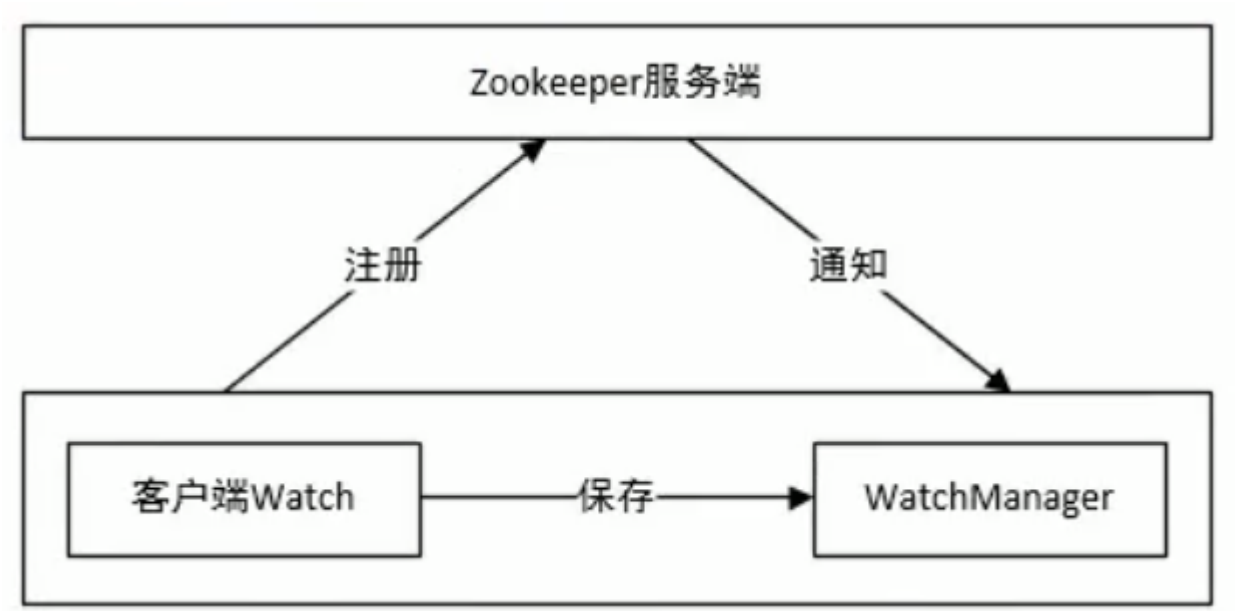
- zookeeper提供了数据的发布/订阅功能，多个订阅者可同时监听某一特定主题对象，当该主题对象的自身状态发生变化时例如节点内容改变、节点下的子节点列表改变等，会实时、主动通知所有订阅者
- zookeeper采用了 Watcher机制实现数据的发布订阅功能。该机制在被订阅对象发生变化时会异步通知客户端，因此客户端不必在 Watcher注册后轮询阻塞，从而减轻了客户端压力
- watcher机制事件上与观察者模式类似，也可看作是一种观察者模式在分布式场景下的实现方式

watcher架构

watcher实现由三个部分组成

- zookeeper服务端
- zookeeper客户端
- 客户端的ZKWatchManager对象

客户端**首先将 Watcher注册到服务端**，同时将 Watcher对象**保存到客户端的watch管理器中。**当zookeeper服务端监听的数据状态发生变化时，服务端会**主动通知客户端**，接着客户端的 Watch管理器会**触发相关** Watcher来回调相应处理逻辑，从而完成整体的数据 发布/订阅流程



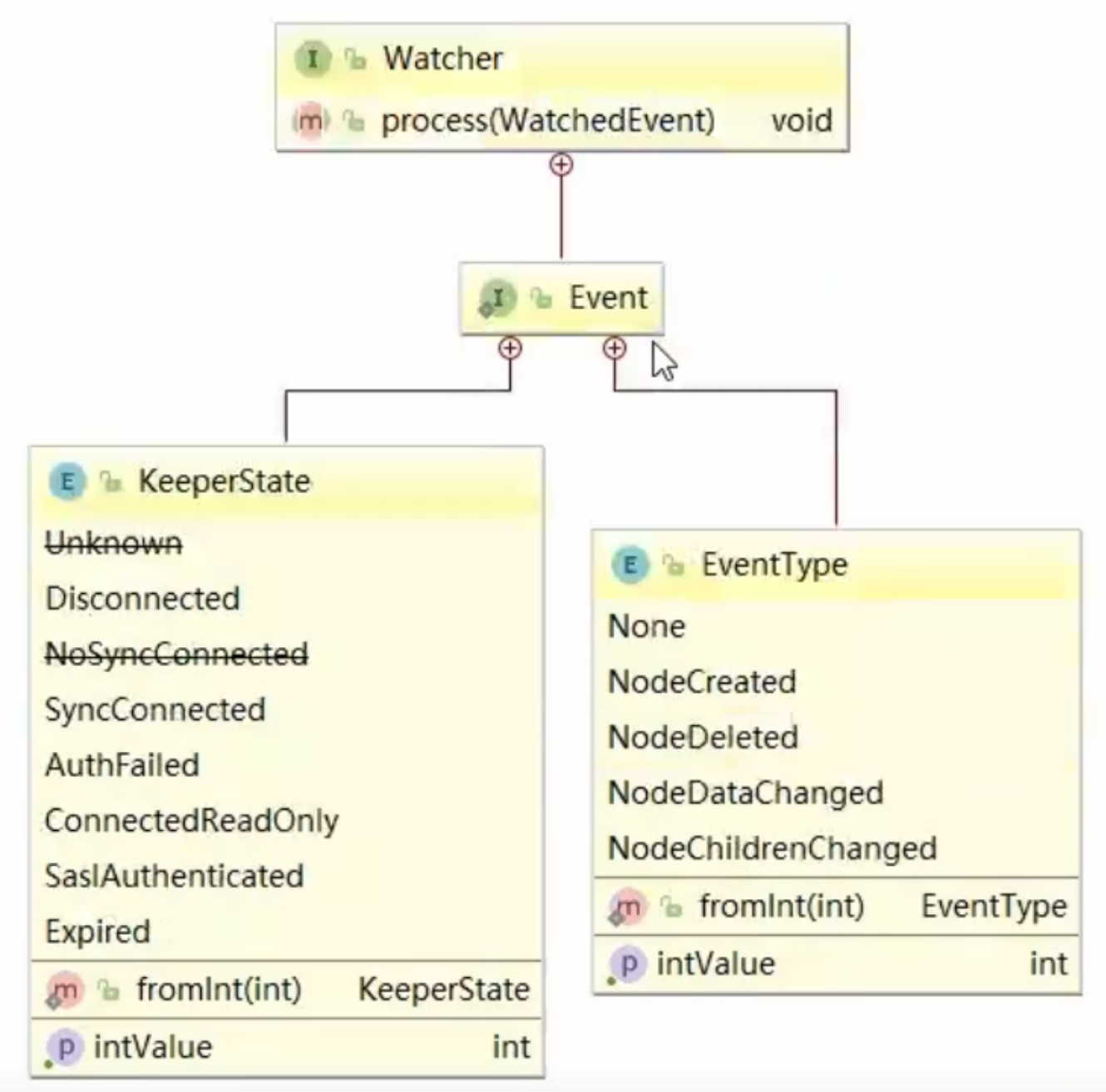
watcher特性

特性	说明
一次性	watcher是**一次性**的，一旦被触发就会移除，再次使用时需要重新注册
客户端顺序回调	watcher回调是**顺序串行**执行的，只有回调后客户端才能看到最新的数据状态。一个watcher回调逻辑不应该太多，以免影响别的watcher执行
轻量级	WatchEvent是最小的通信单位，结构上只包含**通知状态、事件类型和节点路径**，并不会告诉数据节点变化前后的具体内容

特性	说明
时效性	watcher只有在当前session彻底失效时才会无效，若在session有效期内快速重连成功，则watcher依然存在，仍可接收到通知；

watcher接口设计

Watcher是一个接口，任何实现了Watcher接口的类就算一个新的Watcher。Watcher内部包含了两个枚举类：KeeperState、EventType



Watcher通知状态(KeeperState)

KeeperState是客户端与服务端**连接状态**发生变化时对应的通知类型。路径为org.apache.zookeeper.Watcher.EventKeeperState，是一个枚举类，其枚举熟悉如下：

枚举属性	说明
SyncConnected	客户端与服务器正常连接时

枚举属性	说明
Disconnected	客户端与服务器断开连接时
Expired	会话session失效时
AuthFailed	身份认证失败时

Watcher事件类型(EventType)

EventType是**数据节点znode发生变化**时对应的通知类型。**EventType变化时KeeperState永远处于SyncConnected通知状态下**；当keeperState发生变化时，EventType永远为None。其路径为org.apache.zookeeper.Watcher.Event.EventType，是一个枚举类，枚举属性如下：

枚举属性	说明
None	无
NodeCreated	Watcher监听的数据节点被创建时
NodeDeleted	Watcher监听的数据节点被删除时
NodeDataChanged	Watcher监听的数据节点内容发生更改时(无论数据是否真的变化)
NodeChildrenChanged	Watcher监听的数据节点的子节点列表发生变更时

- 注意：客户端接收到的相关事件通知中只包含状态以及类型等信息，不包含节点变化前后的具体内容，变化前的数据需业务自身存储，变化后的数据需要调用get等方法重新获取

捕获相应的事件

上面讲到zookeeper客户端连接的状态和zookeeper对znode节点监听的事件类型，下面我们来讲解如何建立zookeeper的***watcher监听***。在zookeeper中采用zk.getChildren(path,watch)、zk.exists(path,watch)、zk.getData(path,watcher,stat)这样的方式来为某个znode注册监听 。

下表以node-x节点为例，说明调用的注册方法和可用监听事件间的关系：

注册方式	created	childrenChanged	Changed	Deleted
zk.exists("/node-x",watcher)	可监控		可监控	可监控
zk.getData("/node-x",watcher)			可监控	可监控
zk.getChildren("/node-x",watcher)		可监控		可监控

注册watcher的方法

客户端与服务器端的连接状态

- KeeperState：通知状态
- SyncConnected：客户端与服务器正常连接时
- Disconnected：客户端与服务器断开连接时
- Expired：会话session失效时
- AuthFailed：身份认证失败时
- 事件类型为：None
  - 案例
  - ```
    1 public class ZkConnectionWatcher implements Watcher {
    2     @Override
    3     public void process(WatchedEvent watchedEvent) {
    ```
    ```

```
        Event.KeeperState state = watchedEvent.getState();
        if(state == Event.KeeperState.SyncConnected){
            // 正常
            System.out.println("正常连接");
        }else if (state == Event.KeeperState.Disconnected){
            // 可以用Windows断开虚拟机网卡的方式模拟
            // 当会话断开会出现，断开连接不代表不能重连，在会话超时时间内重连可以恢复正常
            System.out.println("断开连接");
        }else if (state == Event.KeeperState.Expired){
            // 没有在会话超时时间内重新连接，而是当会话超时被移除的时候重连会走进这里
            System.out.println("连接过期");
        }else if (state == Event.KeeperState.AuthFailed){
            // 在操作的时候权限不够会出现
            System.out.println("授权失败");
        }
        countDownLatch.countDown();
    }
    private static final String IP = "192.168.133.133:2181"
;
    private static CountDownLatch countDownLatch = new CountDownLatch(1);

    public static void main(String[] args) throws Exception {
        // 5000为会话超时时间
        ZooKeeper zooKeeper = new ZooKeeper(IP, 5000, new
ZkConnectionWatcher());
        countDownLatch.await();
        // 模拟授权失败
        zooKeeper.addAuthInfo("digest1","itcast1:123451".getBytes());
        byte[] data = zooKeeper.getData("/hadoop", false, null);
        System.out.println(new String(data));
        TimeUnit.SECONDS.sleep(50);
    }
}
```

watcher检查节点

exists

- exists(String path, boolean b)
- exists(String path, Watcher w)
- NodeCreated：**节点**创建
- NodeDeleted：**节点**删除
- NodeDataChanged：**节点**内容
    - 案例
    - ```
      public class EventTypeTest {
          private static final String IP = "192.168.133.133:2181";
          private static CountDownLatch countDownLatch = new CountDownLatch(1);
          private static ZooKeeper zooKeeper;

          // 采用zookeeper连接创建时的监听器
          public static void exists1() throws Exception{
              zooKeeper.exists("/watcher1",true);
          }
          // 自定义监听器
          public static void exists2() throws Exception{
      ```
      ```

```
        zooKeeper.exists("/watcher1",(WatchedEvent w) -> {
            System.out.println("自定义" + w.getType());
        });
    }
    // 演示使用多次的监听器
    public static void exists3() throws Exception{
        zooKeeper.exists("/watcher1", new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent) {
                try {
                    System.out.println("自定义的" + watchedEvent.getType());
                } finally {
                    try {
                        zooKeeper.exists("/watcher1",this);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        });
    }
    // 演示一节点注册多个监听器
    public static void exists4() throws Exception{
        zooKeeper.exists("/watcher1",(WatchedEvent w) -> {
            System.out.println("自定义1" + w.getType());
        });
        zooKeeper.exists("/watcher1", new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent) {
                try {
                    System.out.println("自定义2" + watchedEvent.getType());
                } finally {
                    try {
                        zooKeeper.exists("/watcher1",this);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        });
    }
    // 测试
    public static void main(String[] args) throws Exception {
        zooKeeper = new ZooKeeper(IP, 5000, new ZKWatcher());
        countDownLatch.await();
        exists4();
        TimeUnit.SECONDS.sleep(50);
    }

    static class ZKWatcher implements Watcher{
        @Override
        public void process(WatchedEvent watchedEvent) {
            countDownLatch.countDown();
            System.out.println("zk的监听器" + watchedEvent.getType());
        }
    }

}
```

getData

- getData(String path, boolean b, Stat stat)
- getData(String path, Watcher w, Stat stat)
- NodeDeleted：**节点**删除
- NodeDataChange：**节点**内容发生变化

getChildren

- getChildren(String path, boolean b)
- getChildren(String path, Watcher w)
- NodeChildrenChanged：**子节点**发生变化
- NodeDeleted：**节点删除**

配置中心案例

工作中有这样的一个场景：数据库用户名和密码信息放在一个配置文件中，应用读取该配置文件，配置文件信息放入缓存

若数据库的用户名和密码改变时候，还需要重新加载媛存，比较麻烦，通过 Zookeeper可以轻松完成,当数据库发生变化时自动完成缓存同步

使用事件监听机制可以做出一个简单的配置中心

设计思路

1. 连接zookeeper服务器
2. 读取zookeeper中的配置信息，注册watcher监听器，存入本地变量
3. 当zookeeper中的配置信息发生变化时，通过watcher的回调方法捕获数据变化事件
4. 重新获取配置信息

分布式唯一id案例

在过去的单库单表型系统中，通常第可以使用数据库字段自带的auto_ increment属性来自动为每条记录生成个唯一的ID。但是分库分表后,就无法在依靠数据库的auto_ increment属性来唯一标识一条记录了。此时我们就可以用zookeeper在分布式环境下生成全局唯一ID

```
public class IdGenerate {

    private static final String IP = "192.168.133.133:2181";
    private static CountDownLatch countDownLatch = new CountDownLatch(1);
    private static ZooKeeper zooKeeper;

    public static String generateId() throws Exception {
        return zooKeeper.create("/id", new byte[0], ZooDefs.Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL_SEQUENTIAL);
    }


    public static void main(String[] args) throws Exception {
        zooKeeper = new ZooKeeper(IP, 5000, new ZKWatcher());
        countDownLatch.await();
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5, 5, 0,
TimeUnit.SECONDS, new ArrayBlockingQueue<>(10));
        for (int i = 0; i < 10; i++) {
            threadPoolExecutor.execute(() -> {
                try {
                    System.out.println(generateId());
                } catch (Exception e) {
                    e.printStackTrace();
```

```
                }
            });
        }
        TimeUnit.SECONDS.sleep(50);
    }

    static class ZKWatcher implements Watcher {
        @Override
        public void process(WatchedEvent watchedEvent) {
            countDownLatch.countDown();
            System.out.println("zk的监听器" + watchedEvent.getType());
        }
    }
}
```

分布式锁

分布式锁有多种实现方式，比如通过数据库、redis都可实现。作为分布式协同工具Zookeeper，当然也有着标准的实现方式。下面介绍在zookeeper中如果实现排他锁

设计思路

1. 每个客户端往/Locks下创建临时有序节点/Locks/Lock_，创建成功后/Locks下面会有每个客户端对应的节点，如/Locks/Lock_000000001
2. 客户端取得/Locks下子节点，并进行排序，判断排在前面的是否为自己，如果自己的锁节点在第一位，代表获取锁成功
3. 如果自己的锁节点不在第一位，则监听自己前一位的锁节点。例如，自己锁节点Lock_000000002，那么则监听Lock_000000001
4. 当前一位锁节点(Lock_000000001)对应的客户端执行完成，释放了锁，将会触发监听客户端(Lock_000000002)的逻辑
5. 监听客户端重新执行第2步逻辑，判断自己是否获得了锁
6. **zookeeper是有工具包的**

```
public class ThreadTest {
    public static void delayOperation(){
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    static interface Runable{
        void run();
    }
    public static void run(Runable runable,int threadNum){
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(30, 30,
                0, TimeUnit.SECONDS, new ArrayBlockingQueue<>(10));
        for (int i = 0; i < threadNum; i++) {
            threadPoolExecutor.execute(runable::run);
        }
        threadPoolExecutor.shutdown();
    }

    public static void main(String[] args) {
//        DistributedLock distributedLock = new DistributedLock();
//        distributedLock.acquireLock();
//        delayOperation();
//        distributedLock.releaseLock();
```

```
        DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        run(() -> {
            for (int i = 0; i < 999999999; i++) {
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String format = dateTimeFormatter.format(LocalDateTime.now());
                System.out.println(format);
            }
        },1);
        run(() -> {
            DistributedLock distributedLock = new DistributedLock();
            distributedLock.acquireLock();
            delayOperation();
            distributedLock.releaseLock();
        },30);
    }
}
public class DistributedLock {
    private String IP = "192.168.133.133:2181";
    private final String ROOT_LOCK = "/Root_Lock";
    private final String LOCK_PREFIX = "/Lock_";
    private final CountDownLatch countDownLatch = new CountDownLatch(1);
    private final byte[] DATA = new byte[0];

    private ZooKeeper zookeeper;
    private String path;

    private void init(){
        // 初始化
        try {
            zookeeper = new ZooKeeper(IP, 200000, w -> {
                if(w.getState() == Watcher.Event.KeeperState.SyncConnected){
                    System.out.println("连接成功");
                }
                countDownLatch.countDown();
            });
            countDownLatch.await();
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void acquireLock(){
        init();
        createLock();
        attemptLock();
    }

    public void releaseLock(){
        try {
            zookeeper.delete(path,-1);
            System.out.println("锁释放了" + path);
        } catch (InterruptedException | KeeperException e) {
            e.printStackTrace();
        }
    }
```

```
85
86      private void createLock(){
87          try {
88              // 创建根节点
89              Stat root = zookeeper.exists(ROOT_LOCK, false);
90              if(root == null)
91                  zookeeper.create(ROOT_LOCK, DATA, ZooDefs.Ids.OPEN_ACL_UNSAFE,
    CreateMode.PERSISTENT);
92              // 创建子节点
93              path = zookeeper.create(ROOT_LOCK + LOCK_PREFIX, DATA,
    ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
94          } catch (KeeperException | InterruptedException e) {
95              e.printStackTrace();
96          }
97      }
98      private Watcher watcher = new Watcher() {
99          @Override
100         public void process(WatchedEvent watchedEvent) {
101             if (watchedEvent.getType() == Event.EventType.NodeDeleted){
102                 synchronized (this){
103                     this.notifyAll();
104                 }
105             }
106         }
107     };
108
109     private void attemptLock(){
110         try {
111             // 获取正在排队的节点，由于是zookeeper生成的临时节点，不会出错
112             List<String> nodes = zookeeper.getChildren(ROOT_LOCK,false);
113             nodes.sort(String::compareTo);
114             int ranking = nodes.indexOf(path.substring(ROOT_LOCK.length() + 1));
115             // 轮到自己了
116             if(ranking == 0){
117                 return;
118             }else {
119                 // 有可能在上面操作的时候，0号就已经完成了操作，不能让线程白白陷入等待浪费时间
120                 Stat status = zookeeper.exists(ROOT_LOCK+"/"+nodes.get(ranking - 1),
    watcher);
121                 // 有可能这这个判断的瞬间，0号完成了操作(此时我们应该判断成功自旋才对)，但是上面的
    status变量已经获取了值并且不为空，1号沉睡
122                 // 但是，请注意自行测试，虽然1号表面上沉睡了，但是实际上watcher.wait()是瞬间唤醒的
123                 if(status == null){
124                     attemptLock();
125                 }else {
126                     synchronized (watcher){
127                         watcher.wait();
128                     }
129                     attemptLock();
130                 }
131             }
132         } catch (KeeperException | InterruptedException e) {
133             e.printStackTrace();
134         }
135     }
136 }
137
```

集群搭建

zookeeper官网——Getting started——https://zookeeper.apache.org/doc/r3.4.14/zookeeperStarted.html#sc_RunningReplicatedZooKeeper

完全配置——https://zookeeper.apache.org/doc/r3.4.14/zookeeperAdmin.html#sc_zkMulitServerSetup https://zookeeper.apache.org/doc/r3.4.14/zookeeperAdmin.html#sc_configuration

运行时复制的zookeeper

说明：对于复制模式，至少需要三个服务器，并且强烈建议您使用奇数个服务器。如果只有两台服务器，那么您将处于一种情况，如果其中一台服务器发生故障，则没有足够的计算机构成多数仲裁。由于存在两个单点故障，因此两个服务器还**不如**单个服务器稳定。

复制模式所需的**conf / zoo.cfg**文件类似于独立模式下使用的文件，但有一些区别。这是一个例子：

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

- 新的键值`initLimit`是zookeeper用于限制选举中zookeeper服务连接到`leader`的时间，`syncLimit`限制服务器与`leader`的过期时间
- 对于这两个超时，您都可以使用**tickTime**指定时间**单位**。在此示例中，`initLimit`的超时为5个滴答声，即2000毫秒/滴答声，即10秒
- 表格*server.x*的条目列出了组成ZooKeeper服务的服务器。服务器启动时，它通过在数据目录中查找文件*myid*来知道它是哪台服务器。该文件包含ASCII的服务器号。
- 最后，记下每个服务器名称后面的两个端口号：“ 2888”和“ 3888”。**对等方使用前一个端口连接到其他对等方。这种连接是必需的，以便对等方可以进行通信**，例如，以商定更新顺序。更具体地说，ZooKeeper服务器使用此端口将关注者连接到领导者。当出现新的领导者时，跟随者使用此端口打开与领导者的TCP连接。因为默认的领导者选举也使用TCP，所以我们当前需要另一个端口来进行领导者选举。这是服务器条目中的第二个端口。

正文搭建：单机环境下，jdk、zookeeper安装完毕，基于一台虚拟机，进行zookeeper**伪集群搭建**，zookeeper集群中包含3个节点，节点对外提供服务端口号，分别为2181、2182、2183

1. 基于zookeeper-3.4.10复制三份zookeeper安装好的服务器文件,目录名称分别为zookeeper2181、zookeeper2182、zookeeper2183

```
cp -r zookeeper-3.4.10  zookeeper2181
cp -r zookeeper-3.4.10  zookeeper2182
cp -r zookeeper-3.4.10  zookeeper2183

# cp -r zookeeper-3.1.10 ./zookeeper218{1..3}
```

2. 修改zookeeper2181服务器对应配置文件

```
# 服务器对应端口号
clientPort=2181
# 数据快照文件所在路径
```

```
dataDir=/opt/zookeeper2181/data
# 集群配置信息
   # server:A=B:C:D
   # A:是一个数字，表示这个是服务器的编号
   # B:是这个服务器的ip地址
   # C:Zookeeper服务器之间通信的端口(数据互通，必须的)
   # D:Leader选举的端口
server.1=192.168.133.133:2287:3387  # 注意端口号
server.2=192.168.133.133:2288:3388
server.3=192.168.133.133:2289:3389
# 对，这些都是2181的配置文件
```

3. 在上一步 dataDir指定的目录下，创建myid文件，然后在该文件添加上一步server配置的对应A数字

```
# zookeeper2181对应的数字为1
# /opt/zookeeper2181/data目录(即dataDir的目录下)下执行命令
echo "1" > myid
```

4. zookeeper2182、2183参照2/3进行相应配置
5. 分别启动三台服务器，检验集群状态

   检查：cd进入bin目录./zkServer status

   登录命令：

```
./zkCli.sh -server 192.168.60.130:2181
./zkCli.sh -server 192.168.60.130:2182
./zkCli.sh -server 192.168.60.130:2183
# 如果启动后没有显示出集群的状态，请自己检查端口和配置文件问题，主要是端口占和配置文件问题
# ss -lntpd | grep 2181
```

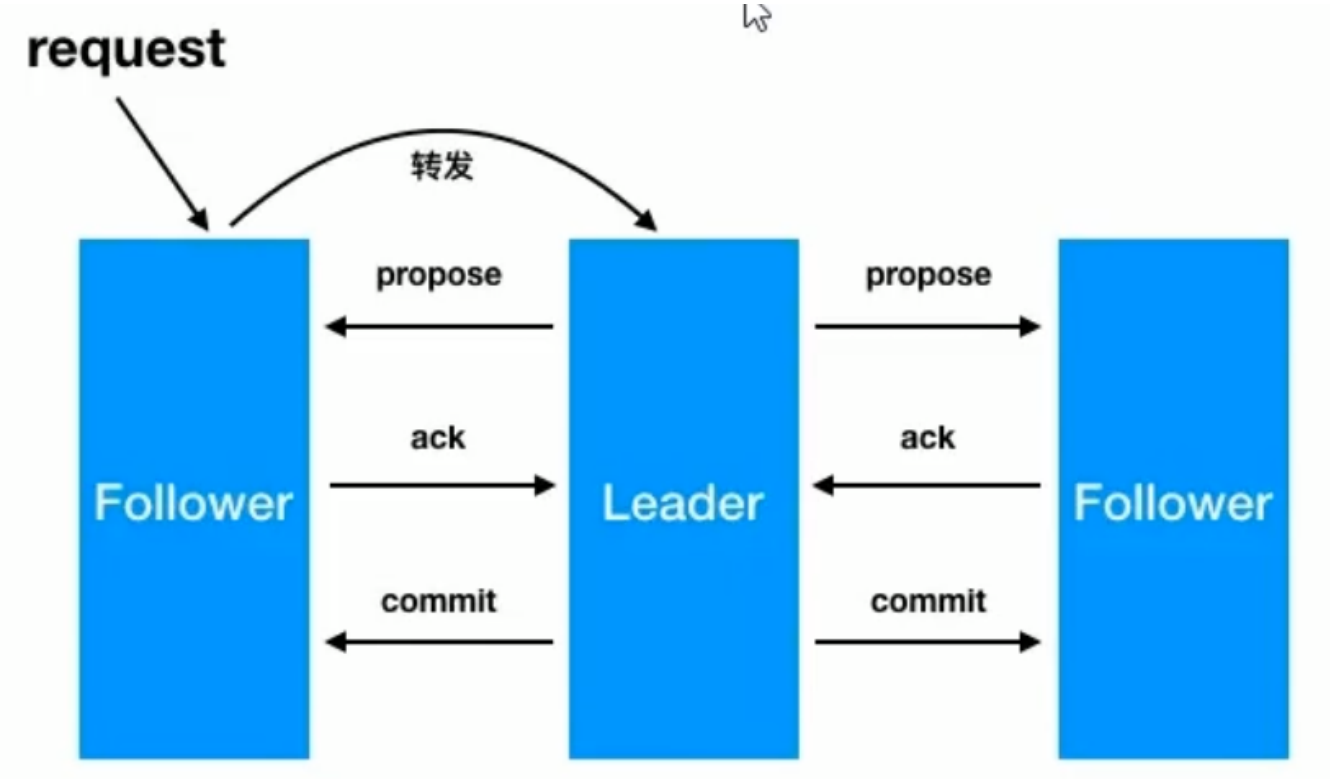
一致性协议——zab协议

zab协议的全称是 *Zookeeper Atomic Broadcast*(zookeeper原子广播)。zookeeper是通过zab协议来保证分布式事务的最终一致性

基于zab协议，zookeeper集群中的角色主要有以下三类，如下所示：

角色	描述
领导者(Leader)	领导者负责进行投票的发起和决议，更新系统状态
学习者(Learner)-跟随者(Follower)	Follower用于接收客户端请求并向客户端返回结果，在选主过程中参与投票
学习者(Learner)-观察者(ObServer)	ObServer可以接收客户端连接，将写请求转发给leader节点。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
客户端(Client)	请求发起方

·zab广播模式工作原理，通过类似两端式提交协议的方式解决数据一致性：



1. leader从客户端收**到一个写请求**
2. leader**生成一个新的事务**并为这个事务生成一个唯一的ZXID
3. leader**将事务提议(propose)发送给所有的**follows**节点**
4. follower节点将收到的事务请求加入到本地**历史队列(history queue)中，并发送**ack**给**leader，表示确认提议
5. 当leader收到大多数follower(**半数以上节点**)的ack确认消息，leader会本地提交，并发送commit请求
6. 当follower**收到commit请求时，从历史队列中将事务请求**commit

leader选举

服务器状态

- looking：寻找leader状态。当服务器处于该状态时，它会认为当前集群中没有leader，因此需要进入leader选举状态
- following：跟随着状态。表明当前服务器角色是follower
- observing：观察者状态。表明当前服务器角色是observer

分为两种选举，**服务器启动时的选举**和**服务器运行时期的选举**

服务器启动时期的leader选举

在集群初始化节点，当有一台服务器server1启动时，其**单独无法进行和完成**leader**选举**，当第二台服务器server2启动时，此时两台及其可以相互通信，每台及其都试图找到leader，**于是进入**leader**选举过程。**选举过程如下：

1. 每个server发出一个投票。由于是初始状态，server1和server2都会将自己作为leader服务器来进行投票，每次投票都会包**含所推举的**myid**和**zxid**，使用(**myid，zxid)，此时server1的投票为(1，0)，server2的投票为(2，0)，然后**各自将这个投票发给集群中的其它机器

2. 集群中的**每台服务器都接收来自集群中各个服务器的投票**
3. **处理投票。**针对每一个投票，服务器都需要将别人的投票和自己的投票进行pk，规则如下
    - 优先检查zxid。zxid比较大的服务器优先作为leader(**zxid较大者保存的数据更多**)
    - 如果zxid相同。那么就比较myid。myid较大的服务器作为leader服务器

      对于Server1而言，它的投票是(1，0)，接收Server2的投票为(2，0)，**首先会比较两者的zxid**，均为0，**再比较myid**，此时server2的myid最大，于是更新自己的投票为(2，0)，然后重新投票，**对于server2而言，无序更新自己的投票**，只是再次向集群中所有机器发出上一次投票信息即可
4. **统计投票。**每次投票后，服务器都会统计投票信息，判断是否已经有**过半机器接受到相同的投票信息**，对于server1、server2而言，都统计出集群中已经有两台机器接受了(2，0)的投票信息，此时便认为已经选举出了leader
5. **改变服务器状态。**一旦确定了leader,每个服务器就会更新自己的状态，如果是follower，那么就变更为following，如果是leader，就变更为leading

举例：如果我们有三个节点的集群，1，2，3，启动 1 和 2 后，2 一定会是 leader，3 再加入不会进行选举，而是直接成为follower

服务器运行时期选举

在zookeeper运行期间，leader与非leader服务器各司其职，即使当有非leader服务器宕机或者新加入，此时也不会影响leader，但是一旦leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮leader选举，其过程和启动时期的leader选举过程基本一致

假设正在运行的有server1、server2、server3三台服务器，当前leader是server2，若某一时刻leader挂了，此时便开始Leader选举。选举过程如下

1. 变更状态。**leader挂后，余下的服务器都会将自己的服务器状态变更为**looking，然后开始进入leader选举过程
2. 每个server发出一个投票。在运行期间，**每个服务器上的zxid可能不同**，此时假定server1的zxid为122，server3的zxid为122，**在第一轮投票中，server1和server3都会投自己**，产生投票(1，122)，(3，122)，然后**各自将投票发送给集群中所有机器**
3. **接收来自各个服务器的投票。**与启动时过程相同
4. **处理投票。**与启动时过程相同，此时，server3将会成为leader
5. **统计投票。**与启动时过程相同
6. **改变服务器的状态。**与启动时过程相同

observer角色及其配置

zookeeper官网——Observers Guidehttps://zookeeper.apache.org/doc/r3.4.14/zookeeperObservers.html

尽管ZooKeeper通过使客户端直接连接到该集合的投票成员而表现良好，但是此体系结构使其很难扩展到大量客户端。问题在于，随着我们添加更多的投票成员，写入性能会下降。这是由于以下事实：写操作需要（通常）集合中至少一半节点的同意，因此，随着添加更多的投票者，投票的成本可能会显着增加。

我们引入了一种称为*Observer*的新型ZooKeeper节点，该节点有助于解决此问题并进一步提高ZooKeeper的可伸缩性。观察员是合奏的非投票成员，他们仅听取投票结果，而听不到投票结果。除了这种简单的区别之外，观察者的功能与跟随者的功能完全相同-客户端可以连接到观察者，并向其发送读写请求。观察者像追随者一样将这些请求转发给领导者，但是他们只是等待听取投票结果。因此，我们可以在不影响投票效果的情况下尽可能增加观察员的数量。

观察者还有其他优点。因为他们不投票，所以它们不是ZooKeeper合奏中的关键部分。因此，它们可以在不损害ZooKeeper服务可用性的情况下发生故障或与群集断开连接。给用户带来的好处是，观察者可以通过比跟随者更不可靠的网络链接进行连接。实际上，观察者可用于与另一个数据中心的ZooKeeper服务器进行对话。观察者的客户端将看到快速读取，因为所有读取均在本地提供，并且由于缺少表决协议而需要的消息数量较小，因此写入会导致网络流量最小

ovserver角色**特点**:

1. **不参与集群的leader选举**
2. **不参与集群中写数据时的ack反馈**

为了使用observer角色，在任何想变成observer角色的配置文件中加入如下配置:

```
peerType=observer
```

并在所有server的配置文件中，配置成observer模式的server的那行配置追加*:observer*，例如

```
server.1=192.168.133.133:2287:3387  # 注意端口号
server.2=192.168.133.133:2288:3388
server.3=192.168.133.133:2289:3389:observer
```

API连接集群

Zookeeper(String connectionString, int sessionTimeout, Watcher watcher)

- connectionString : zookeeper集合主机
- sessionTimeout: 会话超时(以毫秒为单位)
- watcher: 实现"监听器"界面的对象。zookeeper集合通过监视器对象返回连接状态

```
    public static void main(String[] args) throws Exception {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        ZooKeeper connection = new
ZooKeeper("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183", 5000,
watchedEvent -> {
            if (watchedEvent.getState() == Watcher.Event.KeeperState.SyncConnected)
                System.out.println("连接成功");
            countDownLatch.countDown();
        });
        countDownLatch.await();
        connection.create("/hadoop",new byte[0],
ZooDefs.Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
        System.out.println(connection.getSessionId());
    }
```

curator介绍

https://blog.csdn.net/wo541075754/article/details/68067872 关于第三方客户端的小介绍

zkClient有对dubbo的一些操作支持，但是zkClient几乎没有文档，下面是curator

curator简介

curator是Netflix公司开源的一个 zookeeper客户端，后捐献给 apache,，curator框架在zookeeper原生API接口上进行了包装，解决了很多zooKeeper客户端非常底层的细节开发。提供zooKeeper各种应用场景(比如:分布式锁服务、集群领导选举、共享计数器、缓存机制、分布式队列等的抽象封装,实现了Fluent风格的API接口，是最好用，最流行的zookeeper的客户端

原生zookeeperAPI的不足

- 连接对象异步创建，需要开发人员自行编码等待
- 连接没有自动重连超时机制
- watcher一次注册生效一次
- 不支持递归创建树形节点

curator特点

- 解决session会话超时重连
- watcher反复注册
- 简化开发api
- 遵循Fluent风格API

```
    <!-- Zookeeper -->
    <dependency>
        <groupId>org.apache.zookeeper</groupId>
        <artifactId>zookeeper</artifactId>
        <version>3.4.10</version>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-framework</artifactId>
        <version>2.6.0</version>
        <exclustions>
            <exclustion>
               <groupId>org.apache.zookeeper</groupId>
               <artifactId>zookeeper</artifactId>
            </exclustion>
        </exclustions>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-recipes</artifactId>
        <version>2.6.0</version>
    </dependency>
```

基础用法

```
    public static void main(String[] args) {
        // 工厂创建，fluent风格
        CuratorFramework client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点
                .namespace("create")
                .build();
```

```
        client.start();

        System.out.println(client.getState());
        client.close();

    }
```

- session重连策略
  - RetryPolicy retry Policy = new RetryOneTime(3000);
    - 说明：三秒后重连一次，只重连一次
  - RetryPolicy retryPolicy = new RetryNTimes(3,3000);
    - 说明：每三秒重连一次，重连三次
  - RetryPolicy retryPolicy = new RetryUntilElapsed(1000,3000);
    - 说明：每三秒重连一次，总等待时间超过个10秒后停止重连
  - RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,3)
    - 说明：这个策略的重试间隔会越来越长
      - 公式：baseSleepTImeMs * Math.max(1,random.nextInt(1 << (retryCount + 1)))
        - baseSleepTimeMs = 1000 例子中的值
        - maxRetries = 3 例子中的值

创建

```
public class curatorGettingStart {
    public static CuratorFramework client;

    // ids权限
    public static void create1() throws Exception {
        // 新增节点
        client.create()
                // 节点的类型
                .withMode(CreateMode.EPHEMERAL)
                // 节点的acl权限列表
                .withACL(ZooDefs.Ids.OPEN_ACL_UNSAFE)
                // arg1：节点路径，arg2：节点数据
                .forPath("/node1",new byte[0]);
    }
    // 自定义权限
    public static void create2() throws Exception {
        ArrayList<ACL> acls = new ArrayList<>();
        Id id = new Id("world", "anyone");
        acls.add(new ACL(ZooDefs.Perms.READ,id));
        // 新增节点
        client.create()
                // 节点的类型
                .withMode(CreateMode.EPHEMERAL)
                // 节点的acl权限列表
                .withACL(acls)
                // arg1：节点路径，arg2：节点数据
                .forPath("/node2",new byte[0]);
    }
    // 递归创建
    public static void create3() throws Exception {
        // 新增节点
        client.create()
```

```
                // 递归创建
                .creatingParentsIfNeeded()
                // 节点的类型
                .withMode(CreateMode.EPHEMERAL)
                // 节点的acl权限列表
                .withACL(ZooDefs.Ids.OPEN_ACL_UNSAFE)
                // arg1：节点路径，arg2：节点数据
                .forPath("/node2/nodex",new byte[0]);
    }
    // 递归创建
    public static void create4() throws Exception {
        // 新增节点
        System.out.println(1);
        client.create()

                .creatingParentsIfNeeded()
                // 节点的类型
                .withMode(CreateMode.EPHEMERAL)
                // 节点的acl权限列表
                .withACL(ZooDefs.Ids.OPEN_ACL_UNSAFE)
                // 异步
                .inBackground(new BackgroundCallback() {
                    @Override
                    public void processResult(CuratorFramework curatorFramework,
CuratorEvent curatorEvent) throws Exception {
                        System.out.println("异步创建成功");
                    }
                })
                // arg1：节点路径，arg2：节点数据
                .forPath("/node2/nodex",new byte[0]);
        System.out.println(2);
    }
    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        CuratorFramework client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点
                .namespace("create")
                .build();
        client.start();
//        create1();
//        create2();
//        create3();
        create4();

        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}

```

修改

```
public class curatorGettingStart {
    public static CuratorFramework client;

    public static void set1() throws Exception {
        // 修改节点
        client.setData()
                // 版本
                .withVersion(-1)
                .forPath("/hadoop","hadoop1".getBytes());
    }
    public static void set2() throws Exception {
        // 修改节点
        client.setData()
                .withVersion(1)
                .forPath("/hadoop","hadoop2".getBytes());
    }
    public static void set3() throws Exception {
        // 修改节点
        client.setData()
                .withVersion(1)
                // 异步
                .inBackground(new BackgroundCallback() {
                    @Override
                    public void processResult(CuratorFramework curatorFramework,
CuratorEvent curatorEvent) throws Exception {
                        if(curatorEvent.getType() == CuratorEventType.SET_DATA)
                            System.out.println(curatorEvent.getPath()+ "    "
+curatorEvent.getType());
                    }
                })
                .forPath("/hadoop","hadoop3".getBytes());

    }
    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
                .namespace("update")
                .build();
        client.start();
//        set1();
        set2();
//        set3();
        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}

```

删除

```
public class curatorGettingStart {
```

```
    public static CuratorFramework client;
    public static void delete1() throws Exception {
        // 删除节点
        client.delete()
                .forPath("node1");
    }

    public static void delete2() throws Exception {
        // 删除节点
        client.delete()
                // 版本
                .withVersion(1)
                .forPath("node2");
    }

    public static void delete3() throws Exception {
        // 删除节点
        client.delete()
                // 递归删除
                .deletingChildrenIfNeeded()
                .withVersion(-1)
                .forPath("node3");
    }

    public static void delete4() throws Exception {
        // 删除节点
        client.delete()
                .withVersion(-1)
                // 异步
                .inBackground(new BackgroundCallback() {
                    @Override
                    public void processResult(CuratorFramework curatorFramework,
CuratorEvent curatorEvent) throws Exception {
                        if (curatorEvent.getType() == CuratorEventType.DELETE)
                            System.out.println(curatorEvent.getPath() + "    " +
curatorEvent.getType());
                    }
                })
                .forPath("node3");

    }


    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
                .namespace("delete")
                .build();
        client.start();
        //        delete1();
        //        delete2();
        //        delete3();
```

```
        // delete4();
        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}

```

读取节点

```
public class curatorGettingStart {
    public static CuratorFramework client;
    public static void get1() throws  Exception {
        // 获取数据
        byte[] bytes = client.getData()
                .forPath("/node");
        System.out.println(new String((bytes)));
    }
    public static void get2() throws  Exception {
        Stat stat = new Stat();
        // 获取数据
        byte[] bytes = client.getData()
                .storingStatIn(stat)
                .forPath("/node");;
        System.out.println(new String((bytes)));
        System.out.println(stat.getVersion());
        System.out.println(stat.getCzxid());
    }
    public static void get3() throws  Exception {
        System.out.println(1);
        // 获取数据
        client.getData()
                .inBackground((CuratorFramework curatorFramework, CuratorEvent curatorEvent) -> {
                    System.out.println(curatorEvent.getPath() + "  " + curatorEvent.getType());
                })
                .forPath("/node");;
        System.out.println(2);
    }


    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
                .namespace("get")
                .build();
        client.start();
        get1();
        get2();
        get3();

```

```
        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}
```

读取子节点

```
public class curatorGettingStart {
    public static CuratorFramework client;
    public static void getChildren1() throws  Exception {
        // 获取数据
        List<String> strings = client.getChildren()
                .forPath("/get");
        strings.forEach(System.out::println);
        System.out.println("------------");
    }
    public static void getChildren2() throws  Exception {
        System.out.println(1);
        // 获取数据
        client.getChildren()
                .inBackground((curatorFramework, curatorEvent) -> {
                    curatorEvent.getChildren().forEach(System.out::println);
                    System.out.println("------------");
                })
                .forPath("/get");
        System.out.println(2);
        System.out.println("------------");
    }


    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
//                .namespace("get")
                .build();
        client.start();

        getChildren1();
        getChildren2();

        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}
```

watcher

```java
public class WatcherTest {
    static CuratorFramework client;

    public static void watcher1() throws Exception {
        // arg1 curator的客户端
        // arg2 监视的路径
        NodeCache nodeCache = new NodeCache(client, "/watcher");
        // 启动
        nodeCache.start();
        nodeCache.getListenable().addListener(new NodeCacheListener() {
            @Override
            // 节点变化时的回调方法
            public void nodeChanged() throws Exception {
                // 路径
                System.out.println(nodeCache.getCurrentData().getPath() + "  " + nodeCache.getCurrentData().getStat());
                // 输出节点内容
                System.out.println(new String(nodeCache.getCurrentData().getData()));
            }
        });
        System.out.println("注册完成");
        // 时间窗内可以一直监听
        //        TimeUnit.SECONDS.sleep(1000);
        //关 闭
        nodeCache.close();
    }

    public static void watcher2() throws Exception {
        // arg1 客户端
        // arg2 路径
        // arg3 事件钟是否可以获取节点数据
        PathChildrenCache pathChildrenCache = new PathChildrenCache(client, "/watcher", true);
        // 启动
        pathChildrenCache.start();
        pathChildrenCache.getListenable().addListener(new PathChildrenCacheListener() {
            @Override
            // 节点变化时的回调方法
            public void childEvent(CuratorFramework curatorFramework, PathChildrenCacheEvent pathChildrenCacheEvent) throws Exception {
                if (pathChildrenCacheEvent != null) {
                    // 获取子节点数据
                    System.out.println(new String(pathChildrenCacheEvent.getData().getData()));
                    // 路径
                    System.out.println(pathChildrenCacheEvent.getData().getPath());
                    // 事件类型
                    System.out.println(pathChildrenCacheEvent.getType());
                }
            }
        });
        // 时间窗内可以一直监听
        TimeUnit.SECONDS.sleep(1000);
        //关 闭
        pathChildrenCache.close();

    }


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

```
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号
.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
                //                 .namespace("get")
                .build();
        client.start();

//        watcher1();
        watcher2();

        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}

```

事务

```
public class CuratorTransaction {
    static CuratorFramework client;


    public static void transaction() throws Exception{
        /*client.inTransaction()
                .create()
                    .withMode(CreateMode.PERSISTENT)
                    .withACL(ZooDefs.Ids.OPEN_ACL_UNSAFE)
                    .forPath("/transaction",new byte[0])
                .and()
                .setData()
                    .forPath("/setData/transaction",new byte[0])
                .and()
                .commit();*/
        client.create()
                .withMode(CreateMode.PERSISTENT)
                .withACL(ZooDefs.Ids.OPEN_ACL_UNSAFE)
                .forPath("/transaction",new byte[0]);
        client.setData()
                .forPath("/setData/transaction",new byte[0]);
    }

    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号
.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次

```

```
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
                //                .namespace("get")
                .build();
        client.start();
        transaction();

        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}
```

分布式锁

- InterProcessMutex：分布式可重入排它锁
- InterProcessReadWriteLock：分布式读写锁

```
public class CuratorDistributeLock {
    public static CuratorFramework client;

    public static void interProcessMutex() throws Exception {
        System.out.println("排他锁");
        // 获取一个分布式排他锁
        InterProcessMutex lock = new InterProcessMutex(client, "/lock1");
        // 开启两个进程测试，会发现：如果一个分布式排它锁获取了锁，那么直到锁释放为止数据都不会被侵扰
        System.out.println("获取锁中");
        lock.acquire();
        System.out.println("操作中");
        for (int i = 0; i < 10; i++) {
            TimeUnit.SECONDS.sleep(1);
            System.out.println(i);
        }
        lock.release();
        System.out.println("释放锁");
    }

    public static void interProcessReadWriteLock1() throws Exception {
        System.out.println("写锁");
        // 分布式读写锁
        InterProcessReadWriteLock lock = new InterProcessReadWriteLock(client,
"/lock1");
        // 开启两个进程测试，观察到写写互斥，特性同排它锁
        System.out.println("获取锁中");
        lock.writeLock().acquire();
        System.out.println("操作中");
        for (int i = 0; i < 10; i++) {
            TimeUnit.SECONDS.sleep(1);
            System.out.println(i);
        }
        lock.writeLock().release();
        System.out.println("释放锁");
    }

    public static void interProcessReadWriteLock2() throws Exception {
        System.out.println("读锁");
        // 分布式读写锁
```

```
        InterProcessReadWriteLock lock = new InterProcessReadWriteLock(client, "/lock1");
        // 开启两个进程测试，观察得到读读共享，两个进程并发进行，注意并发和并行是两个概念，（并发是线程启动时间段不一定一致，并行是绝对一致时间轴）
        // 再测试两个进程，一个读，一个写，也会出现互斥现象
        System.out.println("获取锁中");
        lock.readLock().acquire();
        System.out.println("操作中");
        for (int i = 0; i < 10; i++) {
            TimeUnit.SECONDS.sleep(1);
            System.out.println(i);
        }
        lock.readLock().release();
        System.out.println("释放锁");
    }


    public static void main(String[] args) throws Exception {
        // 工厂创建，fluent风格
        client = CuratorFrameworkFactory.builder()
                // ip端口号

.connectString("192.168.133.133:2181,192.168.133.133:2182,192.168.133.133:2183")
                // 会话超时
                .sessionTimeoutMs(5000)
                // 重试机制，这里是超时后1000毫秒重试一次
                .retryPolicy(new RetryOneTime(1000))
                // 名称空间，在操作节点的时候，会以这个为父节点,可选操作
                //                  .namespace("get")
                .build();
        client.start();
        //        interProcessMutex();
//                interProcessReadWriteLock1();
        interProcessReadWriteLock2();


        System.out.println(client.getState() + "操作完成");
        TimeUnit.SECONDS.sleep(20);
        client.close();
    }
}

```

四字监控命令/配置属性

zookeeper文档——administrator's Guide——https://zookeeper.apache.org/doc/r3.4.14/zookeeperAdmin.html#sc_zkCommands 四字命令

https://zookeeper.apache.org/doc/r3.4.14/zookeeperAdmin.html#sc_configuration 配置属性

zookeeper支持某些特定的四字命令与其的交互。它们大多数是查询命令，用来获取zookeeper服务的当前状态及相关信息。用户再客户端可以通过telnet或nc向zookeeper提交相应的命令。zookeeper常用四字命令见下表所示:

命令	描述
conf	输出相关服务配置的详细信息。比如端口号、zk数据以及日志配置路径、最大连接数，session超时、serverId等

命令	描述
cons	列出所有连接到这台服务器的客户端连接/会话的详细信息。包括"接收/发送"的包数量、sessionId、操作延迟、最后的操作执行等信息
crst	重置当前这台服务器所有连接/会话的统计信息
dump	列出未经处理的会话和临时节点，这仅适用于领导者
envi	处理关于服务器的环境详细信息
ruok	测试服务是否处于正确运行状态。如果正常返回"imok"，否则返回空
stat	输出服务器的详细信息：接收/发送包数量、连接数、模式(leader/follower)、节点总数、延迟。所有客户端的列表
srst	重置server状态
wchs	列出服务器watchers的简洁信息：连接总数、watching节点总数和watches总数
wchc	通过session分组，列出watch的所有节点，它的输出是一个与watch相关的会话的节点信息，根据watch数量的不同，此操作可能会很昂贵（即影响服务器性能），请小心使用
mntr	列出集群的健康状态。包括"接收/发送"的包数量、操作延迟、当前服务模式(leader/follower)、节点总数、watch总数、临时节点总数

tclnet

- yum install -y tclnet
- tclnet 192.168.133.133 2181(进入终端)
    - mntr(现在可以看到信息)

nc

- yum install -y nc
    - echo mntr | nc 192.168.133.133:2181

conf

输出相关服务配置的详细信息

属性	含义
clientPort	客户端端口号
dataDir	数据快照文件目录，默认情况下10w次事务操作生成一次快照
dataLogDir	事务日志文件目录，生产环节中放再独立的磁盘上
tickTime	服务器之间或客户端与服务器之间维持心跳的时间间隔(以毫秒为单位)
maxClientCnxns	最大连接数
minSessionTimeout	最小session超时minSessionTimeout=tickTime*2 ，即使客户端连接设置了会话超时，也不能打破这个限制
maxSessionTimeout	最大session超时maxSessionTimeout=tickTime*20，即使客户端连接设置了会话超时，也不能打破这个限制
serverId	服务器编号
initLimit	集群中follower服务器(F)与leader服务器(L)之间初始连接时能容忍的最多心跳数，实际上以tickTime为单位，换算为毫秒数
syncLimit	集群中follower服务器(F)与leader服务器(L)之间请求和应答之间能容忍的最大心跳数，实际上以tickTime为单位，换算为毫秒数

属性	含义
electionAlg	0：基于UDP的LeaderElection1：基于UDP的FastLeaderElection2：基于UDP和认证的FastLeaderElection3：基于TCP的FastLeaderElection在3.4.10版本中，默认值为3，另外三种算法以及被弃用，并且有计划在之后的版本中将它们彻底删除且不再支持
electionPort	选举端口
quorumPort	数据通信端口
peerType	是否为观察者 1为观察者

cons

列出所有连接到这台服务器的客户端连接/会话的详细信息

属性	含义
ip	IP地址
port	端口号
queued	等待被处理的请求数，请求缓存在队列中
received	收到的包数
sent	发送的包数
sid	会话id
lop	最后的操作 GETD-读取数据 DELE-删除数据 CREA-创建数据
est	连接时间戳
to	超时时间
lcxid	当前会话的操作id
lzxid	最大事务id
lresp	最后响应时间戳
llat	最后/最新 延迟
minlat	最小延时
maxlat	最大延时
avglat	平均延时

crst

重置当前这台服务器所有连接/会话的统计信息

dump

列出临时节点信息，适用于leader

envi

输出关于服务器的环境详细信息

属性	含义
zookeeper.version	版本
host.name	host信息

属性	含义
java.version	java版本
java.vendor	供应商
java.home	运行环境所在目录
java.class.path	classpath
java.library.path	第三方库指定非Java类包的为止(如：dll，so)
java.io.tmpdir	默认的临时文件路径
java.compiler	JIT编辑器的名称
os.name	Linux
os.arch	amd64
os.version	3.10.0-1062.el7.x86_64
user.name	zookeeper
user.home	/opt/zookeeper
user.dir	/opt/zookeeper/zookeeper2181/bin

ruok

测试服务是否处于正确运行状态，如果目标正确运行会返回imok（are you ok | I'm ok）

stat

输出服务器的详细信息与srvr相似(srvr这里不举例了，官网有一点描述)，但是多了每个连接的会话信息

属性	含义
zookeeper version	版本
Latency min/avg/max	延时
Received	收包
Sent	发包
Connections	当前服务器连接数
Outstanding	服务器堆积的未处理请求数
Zxid	最大事务id
Mode	服务器角色
Node count	节点数

srst

重置server状态

wchs

列出服务器watches的简洁信息

属性	含义
connectsions	连接数

属性	含义
watch-paths	watch节点数
watchers	watcher数量

wchc

通过session分组，列出watch的所有节点，它的输出是一个与watch相关的会话的节点列表

问题

```
wchc is not executed because it is not in the whitelist
```

解决办法

```
# 修改启动指令zkServer.sh
# 注意找到这个信息
else
    echo "JMX disabled by user request" >&2
    ZOOMAIN="org.apache.zookeeper.server.quorum.QuorumPeerMain"
fi
# 下面添加如下信息
ZOOMAIN="-Dzookeeper.4lw.commands.whitelist=* ${ZOOMAIN}"
```

每一个客户端的连接的watcher信息都会被收集起来，并且监控的路径都会被展示出来（代价高，消耗性能）

```
[root@localhost bin]# echo wchc | nc 192.168.133.133 2180
0x171be6c6faf0000
        /node2
        /node1
0x171be6c6faf0001
        /node3
```

wchp

通过路径分组，列出所有的watch的session id 信息

配置同wchc

mntr

列出服务器的健康状态

属性	含义
zk_version	版本
zk_avg_latency	平均延时
zk_max_latency	最大延时
zk_min_latency	最小延时
zk_packets_received	收包数
zk_packets_sent	发包数
zk_num_alive_connections	连接数

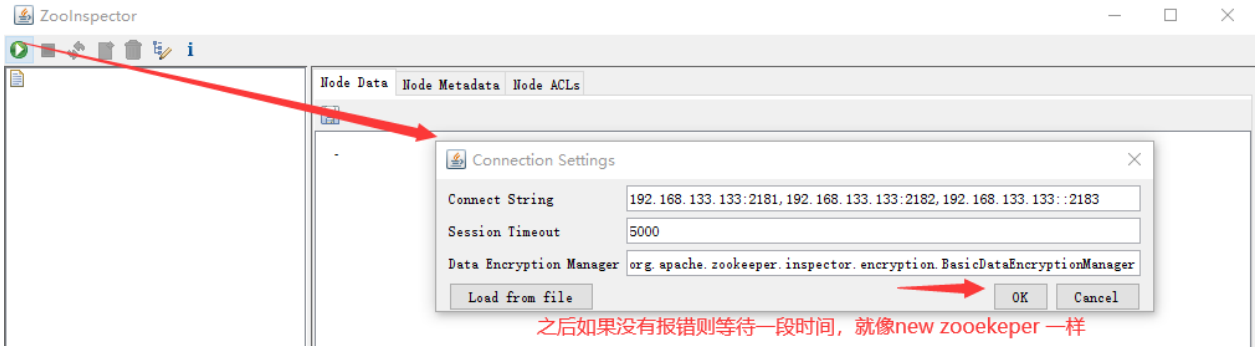
属性	含义
zk_outstanding_requests	堆积请求数
zk_server_state	leader/follower状态
zk_znode_count	znode数量
zk_watch_count	watch数量
zk_ephemerals_count	临时节点(znode)
zk_approximate_data_size	数据大小
zk_open_file_descriptor_count	打开的文件描述符数量
zk_max_file_descriptor_count	最大文件描述符数量

ZooInspector图形化工具

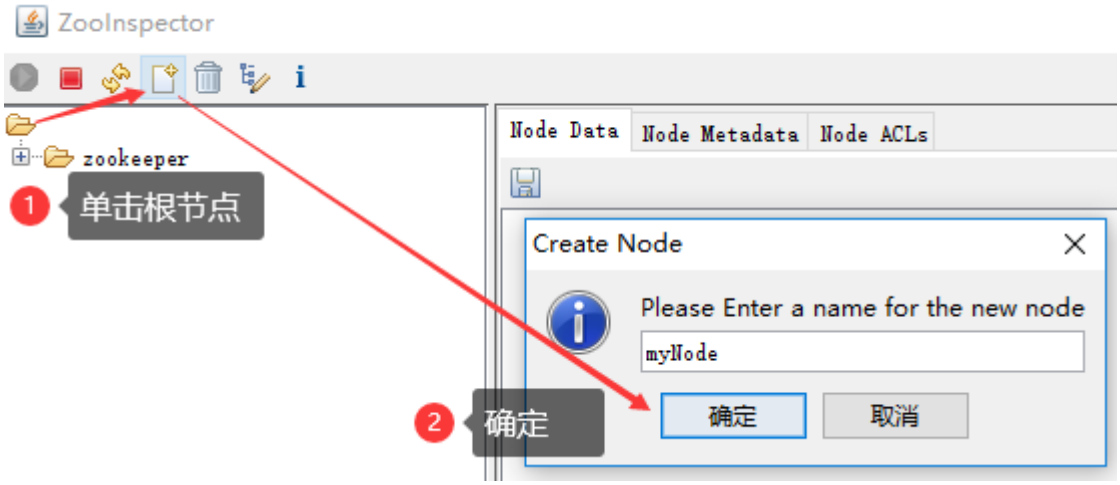
随便百度一个连接就好了

https://issues.apache.org/jira/secure/attachment/12436620/ZooInspector.zip

- 解压后进入目录ZooInspector\build，运行zookeeper-dev-ZooInspector.jar
- java -jar 运行，之后会弹出一个客户端
-



-



- 

- 其它的不必多说，很容易懂(主要是功能也就这几个面板，主要还是直接zkCli.sh)

taokeeper检控工具

beta版，也就是公测版本(并不是开源的)，这里我自己都不用了，期待未来，文档我就照搬了

基于zookeeper的监控管理工具taokeeper，由淘宝团队开发的zk管理中间件，安装强要求服务先配置nc和sshd

1. 下载数据库脚本——算了，我放弃了

Home

jack_zhang edited this page on 18 Nov 2015 · 5 revisions

See more in http://jm-blog.aliapp.com/?p=1450

也不见，淘宝人,阿里人，维护了，志同道合的加QQ群讨论吧：240619787

qq加不了

*2020-4-28 ——*https://www.bilibili.com/video/BV1M741137qY?p=74