尚硅谷技术之 Zookeeper

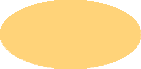
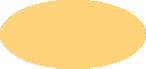
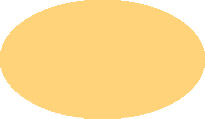
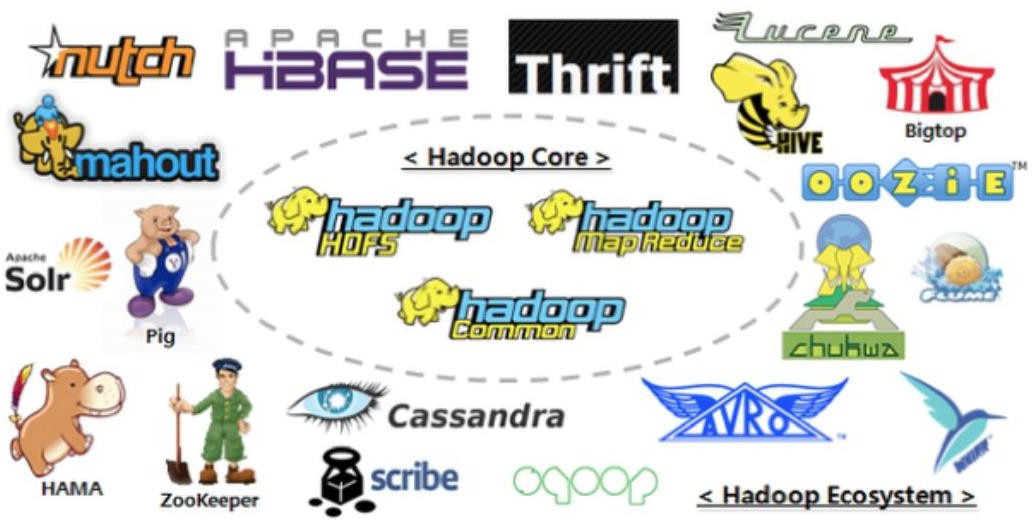
（作者：尚硅谷研究院）

版本：V3.3

# 第 1 章 Zookeeper 入门

## 概述

Zookeeper 是一个开源的分布式的，为分布式框架提供协调服务的 Apache 项目。



**Zookeeper工作机制**

Zookeeper从设计模式角度来理解：是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的 注册，一旦这些数据的状态发生变化，Zookeeper 就将负责通知已经在Zookeeper上注册的那些观察者做出相应的反应。

**服务器1**

业务功能

**服务器2**

业务

功能

**服务器3**

业务

功能

**3 服务器节点下线**

**1 服务端启动时去注册信息（创建都是临时节点）**

Zookeeper=文件系统+通知机制

**2 获取到当前在线服务器列表，并且注册监听**

**4 服务器节点上下线事件通知**

**客户端1**

**5 process(){**

**重新再去获取服务器**

**列表，并注册监听**

**}**

客户端2

客户端3

**Zookeeper集群**

**/servers/server1 hadoop101 80 nodes**

**/server2 hadoop102 90 nodes**

**/server3 hadoop103 95 nodes**

## 特点



**Zookeeper特点**

**Zookeeper Service**

follower follower **leader** follower follower Server1 Server2 Server4 Server5

myid=1 myid=2 myid=4 myid=5

数据 数据 数据 数据

①

②

每次写操作都

有事务id（zxid）

Client

Client

Client

Client

Client

Client

Client

Client

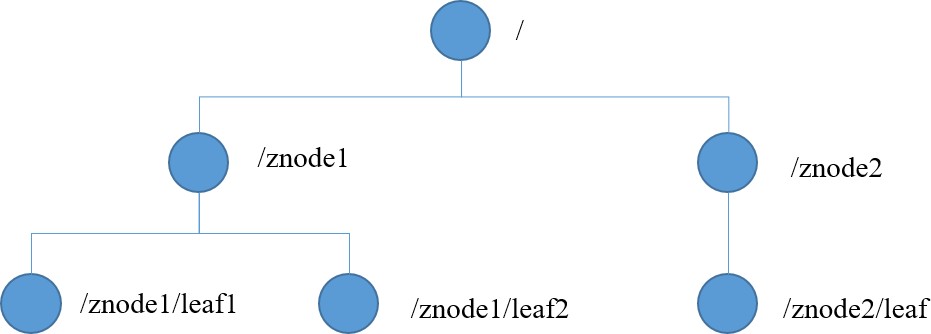
1. Zookeeper：一个领导者（Leader），多个跟随者（Follower）组成的集群。
2. 集群中只要有**半数以上**节点存活，Zookeeper集群就能正常服务。所以Zookeeper适合安装奇数台服务器。
3. 全局数据一致：每个Server保存一份相同的数据副本，Client无论连接到哪个Server，数据都是一致的。
4. 更新请求顺序执行，来自同一个Client的更新请求按其发送顺序依次执行。
5. 数据更新原子性，一次数据更新要么成功，要么失败。
6. 实时性，在一定时间范围内，Client能读到最新数据。

Server3 myid=3

数据

* 1. **数据结构**

ZooKeeper 数据模型的结构与 Unix 文件系统很类似，整体上可以看作是一棵树，每个节点称做一个 ZNode。每一个 ZNode 默认能够存储 1MB 的数据，每个 ZNode 都可以通过其路径唯一标识。



## 应用场景

提供的服务包括：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、软负载均衡等。



**统一命名服务**

在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。

例如：IP不容易记住，而域名容易记住。

**Zookeeper Service**

/

192.168.22.13

访问

访问

访问

**统一配置管理**

1）分布式环境下，配置文件同步非常常见。

**Zookeeper Service**

1. 一般要求一个集群中，所有节点的配置信息是

一致的，比如 Kafka 集群。

1. 对配置文件修改后，希望能够快速同步到各个

节点上。

/

Config

Data

/Configuration

2）配置管理可交由ZooKeeper实现。

（1）可将配置信息写入ZooKeeper上的一个Znode。

watch

watch

watch

1. 各个客户端服务器监听这个Znode。
2. 一旦Znode中的数据被修改，ZooKeeper将通知

各个客户端服务器。

192.168.22.15

192.168.22.14

[www.baidu.com](http://www.baidu.com/)

/service

client

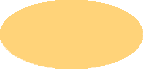
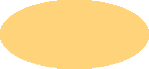
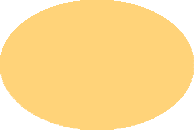
client

client

client3

client2

client1



**统一集群管理**

1）分布式环境中，实时掌握每个节点的状态是必要的。

**Zookeeper Service**

（1）可根据节点实时状态做出一些调整。

/

1. ZooKeeper可以实现实时监控节点状态变化
   1. 可将节点信息写入ZooKeeper上的一个ZNode。
   2. 监听这个ZNode可获取它的实时状态变化。

/GroupManager

/client1

/client2

/client3

Register and watch

Register

and watch

Register and watch

**服务器动态上下线**

**客户端能实时洞察到服务器上下线的变化**

**3 服务器节点下线**

**1 服务端启动时去注册信**

**息（创建都是临时节点）**

**2 获取到当前在线服务器列**

**表，并且注册监听**

**4 服务器节点上下线事件通知**

**客户端1**

**5 process(){**

**重新再去获取服务器列表，并注册监听**

**}**

客户端2

客户端3

**Zookeeper集群**

**/servers/server1 hadoop101 80 nodes**

**/server2 hadoop102 90 nodes**

**/server3 hadoop103 95 nodes**

**服务器3**

业务功能

**服务器2**

业务功能

**服务器1**

业务功能

client

client

client



**软负载均衡**

在Zookeeper中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求

**Zookeeper Service**

/

访问

访问

访问

192.168.22.15 访问数55

192.168.22.14 访问数50

192.168.22.13 访问数60

/注册登录服务

[www.atguigu.com](http://www.atguigu.com/)

/service

client3

client2

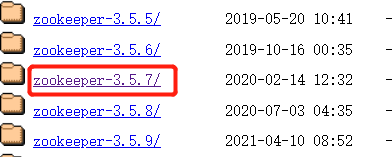
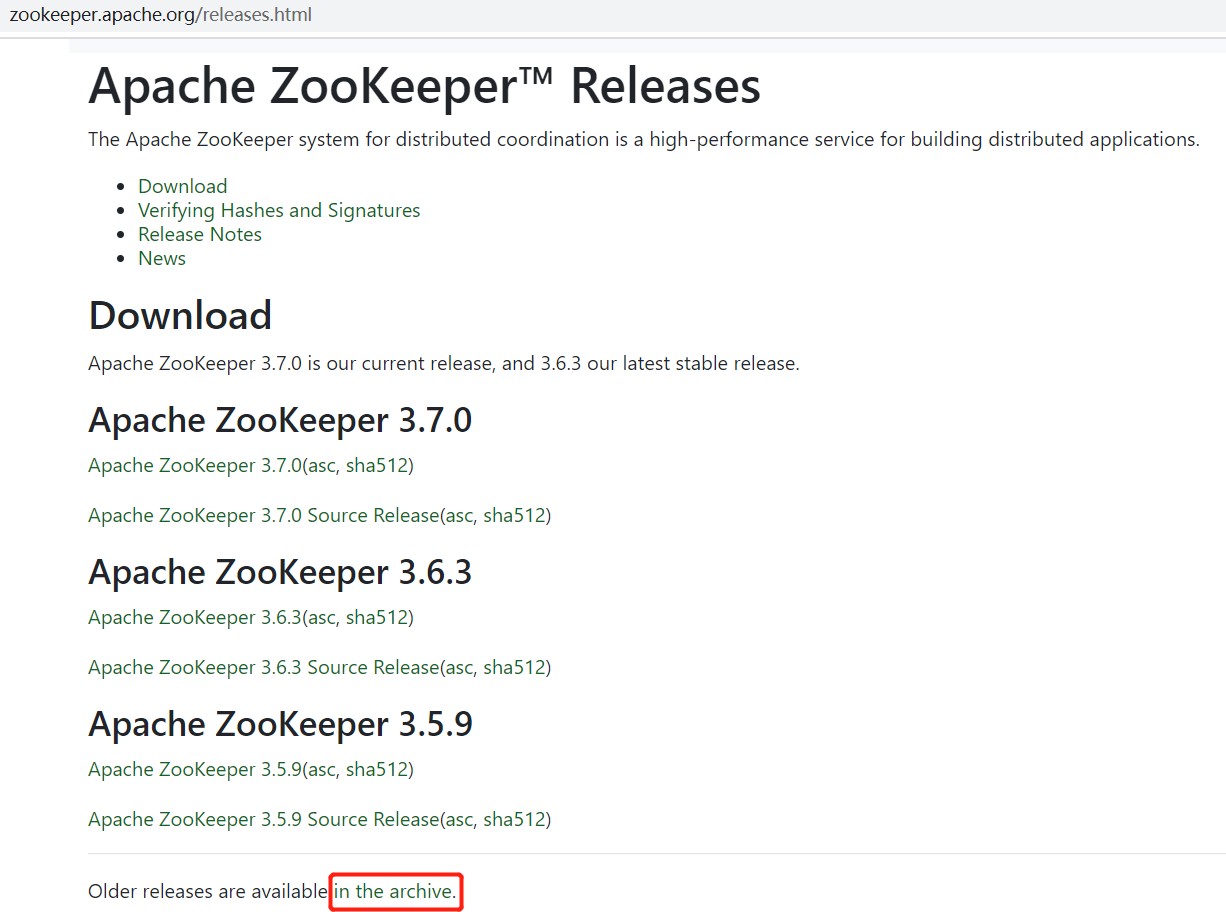
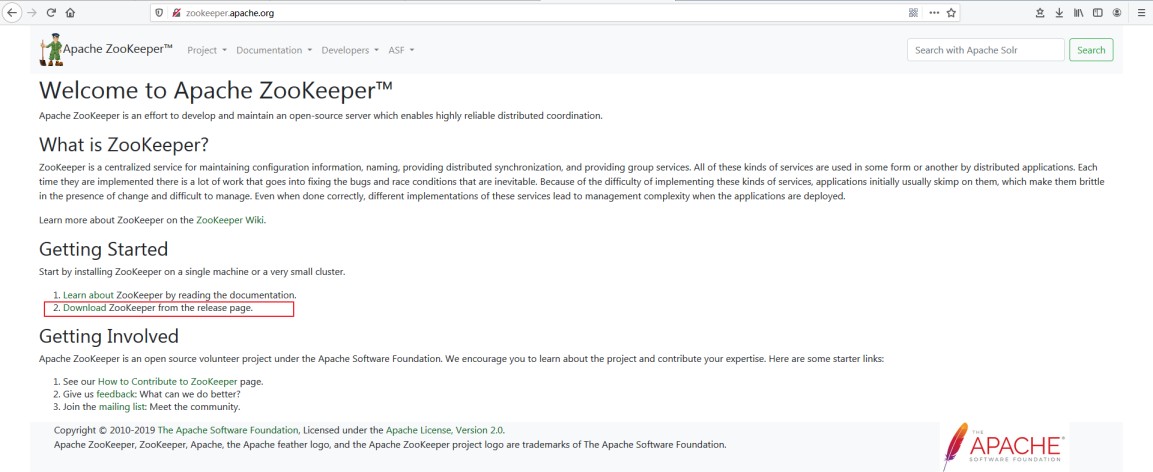
client1

## 下载地址

### 官网首页：

https://zookeeper.apache.org/

### 下载截图



1. **下载 Linux 环境安装的 tar 包**



# 第 2 章 Zookeeper 本地安装

## 本地模式安装

### 安装前准备

* 1. 安装 JDK
  2. 拷贝 apache-zookeeper-3.5.7-bin.tar.gz 安装包到 Linux 系统下
  3. 解压到指定目录

[atguigu@hadoop102 software]$ tar -zxvf apache-zookeeper-3.5.7- bin.tar.gz -C /opt/module/

* 1. 修改名称

[atguigu@hadoop102 module]$ mv apache-zookeeper-3.5.7 -bin/ zookeeper-3.5.7

### 配置修改

1. 将/opt/module/zookeeper-3.5.7/conf 这个路径下的 zoo\_sample.cfg 修改为 zoo.cfg；

[atguigu@hadoop102 conf]$ mv zoo\_sample.cfg zoo.cfg

1. 打开zoo.cfg 文件，修改 dataDir 路径：

[atguigu@hadoop102 zookeeper-3.5.7]$ vim zoo.cfg

修改如下内容：

dataDir=/opt/module/zookeeper-3.5.7/zkData

1. 在/opt/module/zookeeper-3.5.7/这个目录上创建 zkData 文件夹

[atguigu@hadoop102 zookeeper-3.5.7]$ mkdir zkData

### 操作 Zookeeper

1. 启动Zookeeper

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh start

1. 查看进程是否启动

[atguigu@hadoop102 zookeeper-3.5.7]$ jps 4020 Jps

4001 QuorumPeerMain

1. 查看状态

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh status ZooKeeper JMX enabled by default

Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg Mode: standalone

1. 启动客户端

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkCli.sh

1. 退出客户端：

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

1. 停止Zookeeper

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh stop

## 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：通信心跳时间，Zookeeper服务器与客户端心跳时间，单位毫秒



**2）initLimit = 10：LF初始通信时限**



Leader和Follower初始连接时能容忍的最多心跳数（tickTime的数量）

### 3）syncLimit = 5：LF同步通信时限



Leader和Follower之间通信时间如果超过syncLimit \* tickTime，Leader认为Follwer死掉，从服务器列表中删除Follwer。

1. **dataDir：**保存Zookeeper中的数据

注意：默认的tmp目录，容易被Linux系统定期删除，所以一般不用默认的tmp目录。

**5）clientPort = 2181：客户端连接端口，通常不做修改。**

# 第 3 章 Zookeeper 集群操作

## 集群操作

* + 1. **集群安装**

### 集群规划

在 hadoop102、hadoop103 和 hadoop104 三个节点上都部署 Zookeeper。

*思考：如果是 10 台服务器，需要部署多少台 Zookeeper？*

### 解压安装

* 1. 在 hadoop102 解压Zookeeper 安装包到/opt/module/目录下

[atguigu@hadoop102 software]$ tar -zxvf apache-zookeeper-3.5.7- bin.tar.gz -C /opt/module/

* 1. 修改 apache-zookeeper-3.5.7-bin 名称为 zookeeper-3.5.7

[atguigu@hadoop102 zookeeper-3.5.7

module]$

mv apache-zookeeper-3.5.7-bin/

### 配置服务器编号

1. 在/opt/module/zookeeper-3.5.7/这个目录下创建 zkData

[atguigu@hadoop102 zookeeper-3.5.7]$ mkdir zkData

1. 在/opt/module/zookeeper-3.5.7/zkData 目录下创建一个 myid 的文件

[atguigu@hadoop102 zkData]$ vi myid

在文件中添加与 server 对应的编号（注意：上下不要有空行，左右不要有空格）

2

注意：添加 myid 文件，一定要在 Linux 里面创建，在 notepad++里面很可能乱码

1. 拷贝配置好的 zookeeper 到其他机器上

[atguigu@hadoop102 module ]$ xsync zookeeper-3.5.7

并分别在 hadoop103、hadoop104 上修改 myid 文件中内容为 3、4

### 配置zoo.cfg文件

1. 重命名/opt/module/zookeeper-3.5.7/conf 这个目录下的 zoo\_sample.cfg 为 zoo.cfg

[atguigu@hadoop102 conf]$ mv zoo\_sample.cfg zoo.cfg

1. 打开zoo.cfg 文件

[atguigu@hadoop102 conf]$ vim zoo.cfg

#修改数据存储路径配置

dataDir=/opt/module/zookeeper-3.5.7/zkData

#增加如下配置

#######################cluster##########################

server.2=hadoop102:2888:3888 server.3=hadoop103:2888:3888 server.4=hadoop104:2888:3888

1. 配置参数解读

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。

1. 同步zoo.cfg 配置文件

[atguigu@hadoop102 conf]$ xsync zoo.cfg

### 集群操作

1. 分别启动Zookeeper

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh start

[atguigu@hadoop103 zookeeper-3.5.7]$ bin/zkServer.sh start [atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh start

1. 查看状态

[atguigu@hadoop102 zookeeper-3.5.7]# bin/zkServer.sh status JMX enabled by default

Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg Mode: follower

[atguigu@hadoop103 zookeeper-3.5.7]# bin/zkServer.sh status JMX enabled by default

Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg Mode: leader

[atguigu@hadoop104 zookeeper-3.4.5]# bin/zkServer.sh status JMX enabled by default

Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg Mode: follower

## 选举机制（面试重点）



**Zookeeper选举机制——第一次启动**

**Zookeeper Service**

Client

follower

Server1 myid=1

LOOKING

1 0

follower

Server2 myid=2

LOOKING

1 2 0

**leader** follower follower

Server4 Server5

myid=4 myid=5

每次写操作都有事务id（zxid）

Client

**SID**：服务器ID。用来唯一标识一台ZooKeeper集群中的机器，每台机器不能重复，和myid一致。

**ZXID**：事务ID。ZXID是一个事务ID，用来标识一次服务器状态的变更。在某一时刻， 集群中的每台机器的ZXID值不一定完全一致，这和ZooKeeper服务器对于客户端“更 新请求”的处理逻辑有关。

3

**Epoch**：每个Leader任期的代号。没有Leader时同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加

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；

1. 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myid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
2. 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；

1. 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；

1. 服务器5启动，同4一样当小弟。

Server3 myid=3



**Zookeeper选举机制——非第一次启动**

**Zookeeper Service**

Client

follower follower **leader** follower follower

Server1 Server2 Server4 Server5

myid=1 myid=2 myid=4 myid=5

每次写操作都有事务id（zxid）

Client

**SID**：服务器ID。用来唯一标识一台ZooKeeper集群中的机器，每台机器不能重复，和myid一致。

**ZXID**：事务ID。ZXID是一个事务ID，用来标识一次服务器状态的变更。在某一时刻， 集群中的每台机器的ZXID值不一定完全一致，这和ZooKeeper服务器对于客户端“更 新请求”的处理逻辑有关。

（1）当ZooKeeper集群中的一台服务器出现以下两种情况之一时，就会开始进入Leader选举：

* 服务器初始化启动。
* 服务器运行期间无法和Leader保持连接。

1. 而当一台机器进入Leader选举流程时，当前集群也可能会处于以下两种状态：
   * 集群中本来就已经存在一个Leader。

**Epoch**：每个Leader任期的代号。没有Leader时同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加

对于第一种已经存在Leader的情况，机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器来说，仅仅需要和Leader机器建立连 接，并进行状态同步即可。

* **集群中确实不存在Leader。**

假设ZooKeeper由5台服务器组成，SID分别为1、2、3、4、5，ZXID分别为8、8、8、7、7，并且此时SID为3的服务器是Leader。某一时刻，

3和5服务器出现故障，因此开始进行Leader选举。

（EPOCH，ZXID，SID ） （EPOCH，ZXID，SID ） （EPOCH，ZXID，SID ）

SID为1、2、4的机器投票情况： （1，8，1）

（1，8，2）

（1，7，4）

**选举Leader规则：** ①EPOCH大的直接胜出 ②EPOCH相同，事务id大的胜出 ③事务id相同，服务器id大的胜出

Server3 myid=3

* + 1. **ZK 集群启动停止脚本**
       1. 在hadoop102 的/home/atguigu/bin 目录下创建脚本

[atguigu@hadoop102 bin]$ vim zk.sh

在脚本中编写如下内容

#!/bin/bash

case $1 in "start"){

for i in hadoop102 hadoop103 hadoop104 do

echo ---------- zookeeper $i 启动 ------------

ssh $i "/opt/module/zookeeper-3.5.7/bin/zkServer.sh

start"

done

};;

"stop"){

for i in hadoop102 hadoop103 hadoop104 do

echo ---------- zookeeper $i 停止 ------------

ssh $i "/opt/module/zookeeper-3.5.7/bin/zkServer.sh

stop"

done

};;

"status"){

for i in hadoop102 hadoop103 hadoop104 do

echo ---------- zookeeper $i 状态 ------------

ssh status"

done

};;

esac

$i

"/opt/module/zookeeper-3.5.7/bin/zkServer.sh

* + - 1. 增加脚本执行权限

[atguigu@hadoop102 bin]$ chmod u+x zk.sh

* + - 1. Zookeeper 集群启动脚本

[atguigu@hadoop102 module]$ zk.sh start

* + - 1. Zookeeper 集群停止脚本

[atguigu@hadoop102 module]$ zk.sh stop

## 客户端命令行操作

* + 1. **命令行语法**

|  |  |
| --- | --- |
| **命令基本语法** | **功能描述** |
| help | 显示所有操作命令 |
| ls path | 使用 ls 命令来查看当前 znode 的子节点 [可监听]  -w 监听子节点变化  -s 附加次级信息 |

|  |  |
| --- | --- |
| create | 普通创建  -s 含有序列  -e 临时（重启或者超时消失） |
| get path | 获得节点的值 [可监听]  -w 监听节点内容变化  -s 附加次级信息 |
| set | 设置节点的具体值 |
| stat | 查看节点状态 |
| delete | 删除节点 |
| deleteall | 递归删除节点 |

### 启动客户端

[atguigu@hadoop102 hadoop102:2181

zookeeper-3.5.7]$

bin/zkCli.sh -server

1. **显示所有操作命令**

[zk: hadoop102:2181(CONNECTED) 1] help

## znode 节点数据信息

### 查看当前znode中所包含的内容

[zk: hadoop102:2181(CONNECTED) 0] ls / [zookeeper]

1. **查看当前节点详细数据**

[zk: hadoop102:2181(CONNECTED) 5] ls -s / [zookeeper]cZxid = 0x0

ctime = Thu Jan 01 08:00:00 CST 1970 mZxid = 0x0

mtime = Thu Jan 01 08:00:00 CST 1970 pZxid = 0x0

cversion = -1

dataVersion = 0

aclVersion = 0 ephemeralOwner = 0x0 dataLength = 0

numChildren = 1

* 1. czxid：创建节点的事务 zxid

每次修改ZooKeeper 状态都会产生一个ZooKeeper 事务 ID。事务 ID 是ZooKeeper 中所有修改总的次序。每次修改都有唯一的 zxid，如果 zxid1 小于 zxid2，那么zxid1 在 zxid2 之前发生。

* 1. ctime：znode 被创建的毫秒数（从 1970 年开始）
  2. mzxid：znode 最后更新的事务zxid
  3. mtime：znode 最后修改的毫秒数（从 1970 年开始）
  4. pZxid：znode 最后更新的子节点zxid
  5. cversion：znode 子节点变化号，znode 子节点修改次数
  6. dataversion：znode 数据变化号
  7. aclVersion：znode 访问控制列表的变化号
  8. ephemeralOwner：如果是临时节点，这个是 znode 拥有者的 session id。如果不是临时节点则是 0。
  9. dataLength：znode 的数据长度
  10. numChildren：znode 子节点数量

## 节点类型（持久/短暂/有序号/无序号）



**节点类型**

持久（Persistent）：客户端和服务器端断开连接后，创建的节点不删除

短暂（Ephemeral）：客户端和服务器端断开连接后，创建的节点自己删除

Servers

/

/znode1

Persistent

/znode2\_001

Persistent\_sequential

/znode3

Ephemeral

说明：创建znode时设置顺序标识，znode名称后会附加一个值，顺序号是一个单调递增的计数器，由父节点维护

注意：在分布式系统中，顺序号可以被用于

为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

/znode4\_001

Ephemeral\_sequential

Client1

Client2

Client3

Client4

（1）持久化目录节点

客户端与Zookeeper断开连接后，该节点依旧存在

（3）临时目录节点

客户端与Zookeeper断开连接后，该节点被删除

（2）持久化顺序编号目录节点

客户端与Zookeeper断开连接后，该节点依旧存

在，只是Zookeeper给该节点名称进行顺序编号

（4）临时顺序编号目录节点

客户端与Zookeeper 断开连接后， 该节点被删除， 只是

Zookeeper给该节点名称进行顺序编号。

### 1）分别创建2个普通节点（永久节点 + 不带序号）

[zk: localhost:2181(CONNECTED) 3] create /sanguo "diaochan" Created /sanguo

[zk: localhost:2181(CONNECTED) "liubei"

Created /sanguo/shuguo

4]

create /sanguo/shuguo

注意：创建节点时，要赋值

### 获得节点的值

[zk: localhost:2181(CONNECTED) 5] get -s /sanguo diaochan

cZxid = 0x100000003

ctime = Wed Aug 29 00:03:23 CST 2018 mZxid = 0x100000003

mtime = Wed Aug 29 00:03:23 CST 2018 pZxid = 0x100000004

cversion = 1

dataVersion = 0

aclVersion = 0 ephemeralOwner = 0x0

dataLength = 7

numChildren = 1

[zk: localhost:2181(CONNECTED) 6] get -s /sanguo/shuguo liubei

cZxid = 0x100000004

ctime = Wed Aug 29 00:04:35 CST 2018 mZxid = 0x100000004

mtime = Wed Aug 29 00:04:35 CST 2018 pZxid = 0x100000004

cversion = 0

dataVersion = 0

aclVersion = 0 ephemeralOwner = 0x0 dataLength = 6

numChildren = 0

**3）创建带序号的节点（永久节点 + 带序号）**

* 1. 先创建一个普通的根节点/sanguo/weiguo

[zk: localhost:2181(CONNECTED) 1] create /sanguo/weiguo "caocao"

Created /sanguo/weiguo

* 1. 创建带序号的节点

[zk:

localhost:2181(CONNECTED)

2]

create

-s

/sanguo/weiguo/zhangliao "zhangliao" Created /sanguo/weiguo/zhangliao0000000000

[zk:

localhost:2181(CONNECTED)

3]

create

-s

/sanguo/weiguo/zhangliao "zhangliao" Created /sanguo/weiguo/zhangliao0000000001

[zk:

localhost:2181(CONNECTED)

4]

create

-s

/sanguo/weiguo/xuchu "xuchu"

Created /sanguo/weiguo/xuchu0000000002

如果原来没有序号节点，序号从 0 开始依次递增。如果原节点下已有 2 个节点，则再排

序时从 2 开始，以此类推。

### 4）创建短暂节点（短暂节点 + 不带序号 or 带序号）

1. 创建短暂的不带序号的节点

[zk: localhost:2181(CONNECTED) 7] create -e /sanguo/wuguo "zhouyu"

Created /sanguo/wuguo

1. 创建短暂的带序号的节点

[zk: localhost:2181(CONNECTED) 2] create -e -s /sanguo/wuguo "zhouyu"

Created /sanguo/wuguo0000000001

1. 在当前客户端是能查看到的

[zk: localhost:2181(CONNECTED) 3] ls /sanguo [wuguo, wuguo0000000001, shuguo]

1. 退出当前客户端然后再重启客户端
2. 再次查看根目录下短暂节点已经删除

[zk: localhost:2181(CONNECTED) 0] ls /sanguo [shuguo]

### 5）修改节点数据值

[zk: localhost:2181(CONNECTED) 6] set /sanguo/weiguo "simayi"

## 监听器原理

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、节点删除、子目录节点增加删除）时，ZooKeeper 会通知客户端。监听机制保证 ZooKeeper 保存的任何的数

据的任何改变都能快速的响应到监听了该节点的应用程序。

**监听器原理**



connect

**6 process()**

Listener

1. **Main()线程**
2. **创建zkClient**

**1、监听原理详解**

1. 首先要有一个main()线程
2. 在main线程中创建Zookeeper客户端，这时就会创建两个线程，一个负责网络连接通信（connet），一个负责监听（listener）。
3. 通过connect线程将注册的监听事件发送给Zookeeper。
4. 在Zookeeper的注册监听器列表中将注册的监听事件添加到列表中。5）Zookeeper监听到有数据或路径变化，就会将这个消息发送给listener线程。6）listener线程内部调用了process()方法。

**2、常见的监听**

1. 监听节点数据的变化

get path [watch]

1. 监听子节点增减的变化ls path [watch]

**ZK客户端 ZK服务端**

### 节点的值变化监听

**port**

**5 “/”路径数据发生变化**

**3 getChildren(“/”,true)**

|  |  |
| --- | --- |
| **注册的监听器列表** | |
| **4 Client:ip:port:/path** |  |
|  |

[atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkCli.sh

1. 在 hadoop104 主机上注册监听/sanguo 节点数据变化

[zk: localhost:2181(CONNECTED) 26] get -w /sanguo

1. 在 hadoop103 主机上修改/sanguo 节点的数据

[zk: localhost:2181(CONNECTED) 1] set /sanguo "xisi"

1. 观察hadoop104 主机收到数据变化的监听

WATCHER::

WatchedEvent path:/sanguo

state:SyncConnected

type:NodeDataChanged

注意：在hadoop103再多次修改/sanguo的值，hadoop104上不会再收到监听。因为注册一次，只能监听一次。想再次监听，需要再次注册。

### 节点的子节点变化监听（路径变化）

* 1. 在 hadoop104 主机上注册监听/sanguo 节点的子节点变化

[shuguo, weiguo]

* 1. 在 hadoop103 主机/sanguo 节点上创建子节点

[zk: localhost:2181(CONNECTED) 2] create /sanguo/jin "simayi" Created /sanguo/jin

* 1. 观察hadoop104 主机收到子节点变化的监听

WATCHER::

WatchedEvent path:/sanguo

state:SyncConnected

type:NodeChildrenChanged

注意：节点的路径变化，也是注册一次，生效一次。想多次生效，就需要多次注册。

## 节点删除与查看

### 删除节点

[zk: localhost:2181(CONNECTED) 4] delete /sanguo/jin

1. **递归删除节点**

[zk: localhost:2181(CONNECTED) 15] deleteall /sanguo/shuguo

1. **查看节点状态**

[zk: localhost:2181(CONNECTED) 17] stat /sanguo cZxid = 0x100000003

ctime = Wed Aug 29 00:03:23 CST 2018 mZxid = 0x100000011

mtime = Wed Aug 29 00:21:23 CST 2018 pZxid = 0x100000014

cversion = 9

dataVersion = 1

aclVersion = 0 ephemeralOwner = 0x0 dataLength = 4

numChildren = 1

## 客户端 API 操作

前提：保证hadoop102、hadoop103、hadoop104 服务器上Zookeeper 集群服务端启动。

## IDEA 环境搭建

### 1）创建一个工程：zookeeper 2）添加pom文件

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>RELEASE</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.8.2</version>

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.5.7</version>

</dependency>

</dependencies>

1. **拷贝log4j.properties文件到项目根目录**

需要在项目的 src/main/resources 目录下，新建一个文件，命名为“log4j.properties”，在文件中填入。

log4j.rootLogger=INFO, stdout log4j.appender.stdout=org.apache.log4j.ConsoleAppender log4j.appender.stdout.layout=org.apache.log4j.PatternLayout log4j.appender.stdout.layout.ConversionPattern=%d %p [%c]

- %m%n log4j.appender.logfile=org.apache.log4j.FileAppender log4j.appender.logfile.File=target/spring.log

log4j.appender.logfile.layout=org.apache.log4j.PatternLayout

log4j.appender.logfile.layout.ConversionPattern=%d

- %m%n

%p

[%c]

1. **创建包名com.atguigu.zk 5）创建类名称zkClient**

## 创建 ZooKeeper 客户端

// 注意：逗号前后不能有空格

private static String connectString = "hadoop102:2181,hadoop103:2181,hadoop104:2181";

private static int sessionTimeout = 2000; private ZooKeeper zkClient = null;

**@Before**

public void **init**() throws Exception {

zkClient = new ZooKeeper(connectString, sessionTimeout, new Watcher() {

@Override

public void process(WatchedEvent watchedEvent) {

// 收到事件通知后的回调函数（用户的业务逻辑）

System.out.println(watchedEvent.getType() + "--"

+ watchedEvent.getPath());

// 再次启动监听

try {

List<String> children = zkClient.getChildren("/",

true);

for (String child : children) { System.out.println(child);

}

} catch (Exception e) { e.printStackTrace();

}

}

});

}

}

* + 1. **创建子节点**

// 创建子节点

**@Test**

public void **create**() throws Exception {

// 参数 1：要创建的节点的路径； 参数 2：节点数据 ； 参数 3：节点权限 ； 参数 4：节点的类型

String nodeCreated =

"shuaige".getBytes(),

zkClient.create("/atguigu", Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

测试：在 hadoop102 的 zk 客户端上查看创建节点情况

[zk: localhost:2181(CONNECTED) 16] get -s /atguigu shuaige

## 获取子节点并监听节点变化

// 获取子节点

**@Test**

public void **getChildren**() throws Exception {

List<String> children = zkClient.getChildren("/", true); for (String child : children) {

System.out.println(child);

}

// 延时阻塞

Thread.sleep(Long.MAX\_VALUE);

}

1. 在 IDEA 控制台上看到如下节点：

zookeeper sanguo atguigu

1. 在 hadoop102 的客户端上创建再创建一个节点/atguigu1，观察 IDEA 控制台

[zk: localhost:2181(CONNECTED) 3] create /atguigu1 "atguigu1"

1. 在 hadoop102 的客户端上删除节点/atguigu1，观察 IDEA 控制台

[zk: localhost:2181(CONNECTED) 4] delete /atguigu1

## 判断 Znode 是否存在

// 判断 znode 是否存在

**@Test**

public void **exist**() throws Exception {

Stat stat = zkClient.exists("/atguigu", false);

System.out.println(stat == null ? "not exist" : "exist");

}

* 1. **客户端向服务端写数据流程**



**写流程之写入请求直接发送给Leader节点**

6 ack

5 write

3 ack 2 write

4 ack

ZK Server Leader

1 write

ZK Server Follower

Client

ZK Server Follower



**写流程之写入请求发送给follower节点**

6 ack

1 write

ZK Server Follower

5 ack 2 write请求 4 ack

3 write 7 write 8 ack

ZK Server Leader

Client

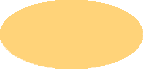
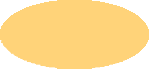
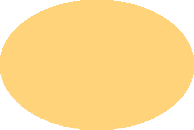
ZK Server Follower

**第 4 章 服务器动态上下线监听案例**

## 需求

某分布式系统中，主节点可以有多台，可以动态上下线，任意一台客户端都能实时感知到主节点服务器的上下线。

## 需求分析



**服务器动态上下线**

**客户端能实时洞察到服务器上下线的变化**

**3 服务器节点下线**

**1 服务端启动时去注册信**

**息（创建都是临时节点）**

**2 获取到当前在线服务器列**

**表，并且注册监听**

**4 服务器节点上下线事件通知**

客户端1

**5 process(){**

**重新再去获取服务器列表，并注册监听**

**}**

客户端2

客户端3

**Zookeeper集群**

**/servers/server1 hadoop101 80 nodes**

**/server2 hadoop102 90 nodes**

**/server3 hadoop103 95 nodes**

**服务器3**

业务功能

**服务器2**

业务功能

**服务器1**

业务功能

* 1. **具体实现**
     1. 先在集群上创建/servers 节点

[zk: localhost:2181(CONNECTED) 10] create /servers "servers" Created /servers

* + 1. 在 Idea 中创建包名：com.atguigu.zkcase1
    2. 服务器端向 Zookeeper 注册代码

package com.atguigu.zkcase1; import java.io.IOException;

import org.apache.zookeeper.CreateMode; import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; import org.apache.zookeeper.ZooDefs.Ids;

public class DistributeServer {

private

static

String

connectString

=

"hadoop102:2181,hadoop103:2181,hadoop104:2181"; private static int sessionTimeout = 2000; private ZooKeeper zk = null;

private String parentNode = "/servers";

**// 创建到 zk 的客户端连接**

public void getConnect() throws IOException{

zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {

@Override

public void process(WatchedEvent event) {

}

});

}

**// 注册服务器**

public void registServer(String hostname) throws Exception{

String create = zk.create(parentNode + "/server", hostname.getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);

System.out.println(hostname +" is online "+ create);

}

**// 业务功能**

public void business(String hostname) throws Exception{ System.out.println(hostname + " is working ...");

Thread.sleep(Long.MAX\_VALUE);

}

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

**// 1 获取 zk 连接**

DistributeServer server = new DistributeServer(); server.getConnect();

**// 2 利用 zk 连接注册服务器信息**

server.registServer(args[0]);

**// 3 启动业务功能**

server.business(args[0]);

}

}

（3）客户端代码

package com.atguigu.zkcase1; import java.io.IOException; import java.util.ArrayList; import java.util.List;

import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper;

public class DistributeClient {

private

static

String

connectString

=

"hadoop102:2181,hadoop103:2181,hadoop104:2181"; private static int sessionTimeout = 2000; private ZooKeeper zk = null;

private String parentNode = "/servers";

**// 创建到 zk 的客户端连接**

public void getConnect() throws IOException {

zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {

@Override

public void process(WatchedEvent event) {

**// 再次启动监听**

try {

getServerList();

} catch (Exception e) { e.printStackTrace();

}

}

});

}

**// 获取服务器列表信息**

public void getServerList() throws Exception {

**// 1 获取服务器子节点信息，并且对父节点进行监听**

List<String> children = zk.getChildren(parentNode, true);

**// 2 存储服务器信息列表**

ArrayList<String> servers = new ArrayList<>();

**// 3 遍历所有节点，获取节点中的主机名称信息**

for (String child : children) {

byte[] data = zk.getData(parentNode + "/" + child, false, null);

servers.add(new String(data));

}

**// 4 打印服务器列表信息**

System.out.println(servers);

}

**// 业务功能**

public void business() throws Exception{

System.out.println("client is working ..."); Thread.sleep(Long.MAX\_VALUE);

}

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

**// 1 获取 zk 连接**

DistributeClient client = new DistributeClient(); client.getConnect();

**// 2 获取 servers 的子节点信息，从中获取服务器信息列表**

client.getServerList();

**// 3 业务进程启动**

client.business();

}

}

## 测试

1. 在Linux 命令行上操作增加减少服务器
   1. 启动DistributeClient 客户端
   2. 在 hadoop102 上 zk 的客户端/servers 目录上创建临时带序号节点

[zk:

localhost:2181(CONNECTED)

1]

create

-e

-s

/servers/hadoop102 "hadoop102"

[zk:

localhost:2181(CONNECTED)

2]

create

-e

-s

/servers/hadoop103 "hadoop103"

* 1. 观察 Idea 控制台变化

[hadoop102, hadoop103]

* 1. 执行删除操作

[zk:

localhost:2181(CONNECTED)

8]

delete

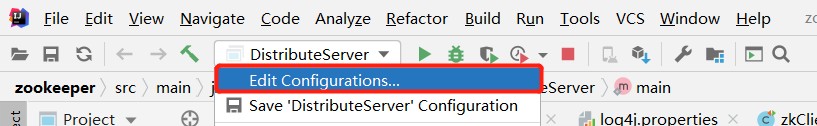
/servers/hadoop1020000000000

* 1. 观察 Idea 控制台变化

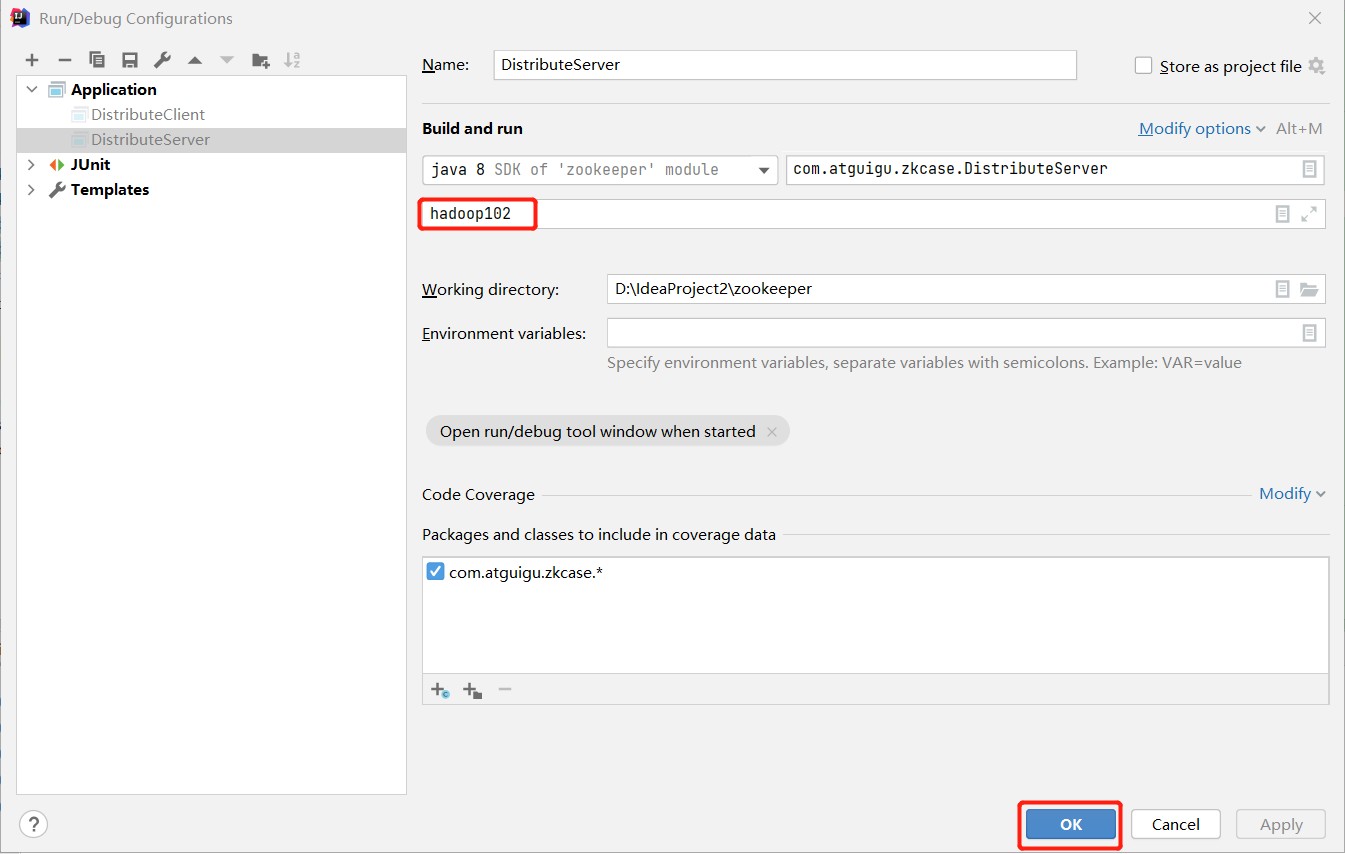
[hadoop103]

1. 在 Idea 上操作增加减少服务器
2. 启动DistributeClient 客户端（如果已经启动过，不需要重启）
3. 启动DistributeServer 服务

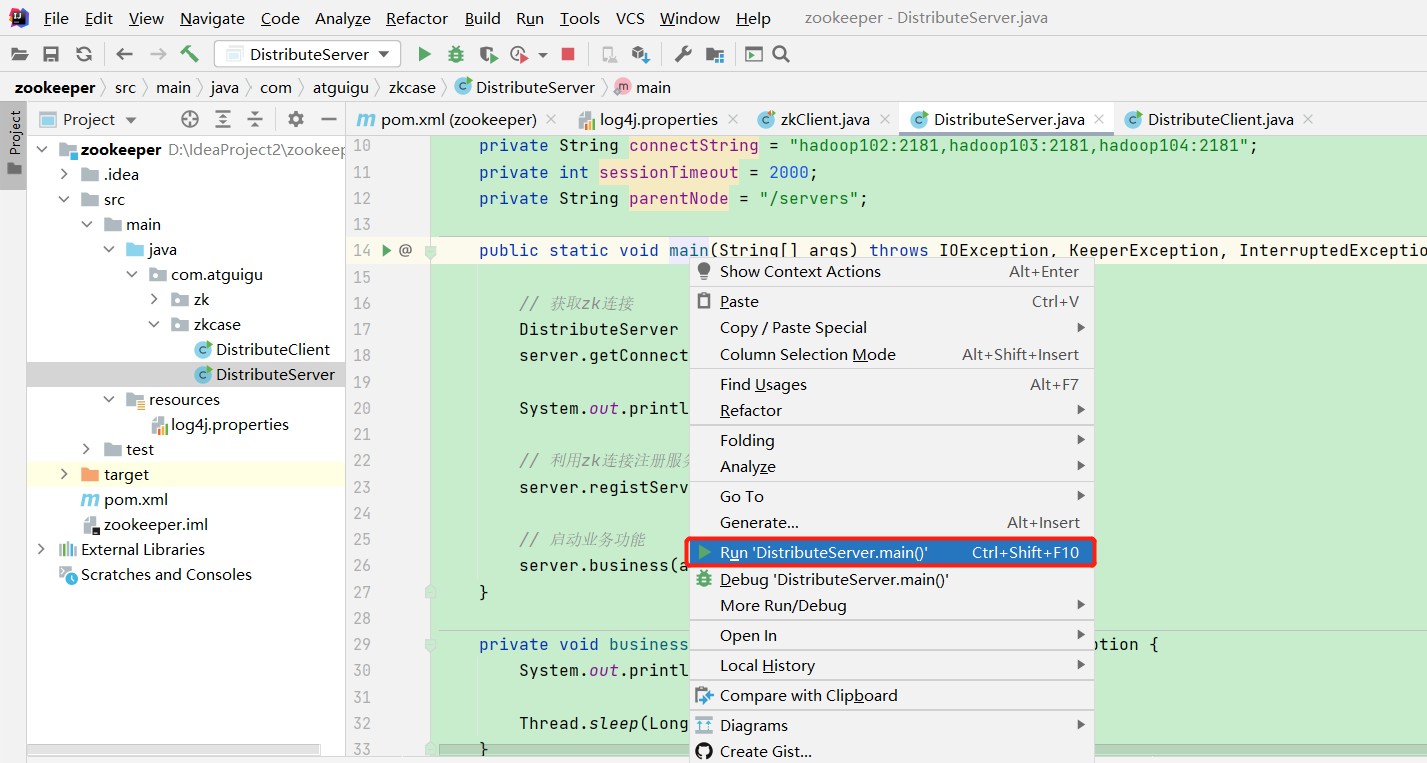
①点击 Edit Configurations…



②在弹出的窗口中（Program arguments）输入想启动的主机，例如，hadoop102



③回到 DistributeServer 的 main 方 法 ， 右 键 ， 在 弹 出 的 窗 口 中 点 击 Run “DistributeServer.main()”



④观察 DistributeServer 控制台，提示hadoop102 is working

⑤观察 DistributeClient 控制台，提示hadoop102 已经上线

# 第 5 章 ZooKeeper 分布式锁案例

什么叫做分布式锁呢？

比如说"进程 1"在使用该资源的时候，会先去获得锁，"进程 1"获得锁以后会对该资源

保持独占，这样其他进程就无法访问该资源，"进程 1"用完该资源以后就将锁释放掉，让其他进程来获得锁，那么通过这个锁机制，我们就能保证了分布式系统中多个进程能够有序的访问该临界资源。那么我们把这个分布式环境下的这个锁叫作分布式锁。



分布式锁案例分析

create -e -s /locks/seq-

创建临时顺序节点

1）接收到请求后，在/locks节点下创建一个临时顺序节点

watch

2）判断自己是不是当前节点下最小的节点：是，获取到

锁；不是，对前一个节点进行监听

watch

watch

3）获取到锁，处理完业务后，delete节点释放锁，然后

下面的节点将收到通知，重复第二步判断

/seq-000000003

/seq-000000002

/seq-000000001

Client

/seq-000000000

/locks

Client

获取到锁后，处理业务

Client

## 原生 Zookeeper 实现分布式锁案例

### 分布式锁实现

package com.atguigu.lock2;

import org.apache.zookeeper.\*;

import org.apache.zookeeper.data.Stat;

import java.io.IOException; import java.util.Collections; import java.util.List;

import java.util.concurrent.CountDownLatch; public class DistributedLock {

// zookeeper server 列表

private String connectString

=

"hadoop102:2181,hadoop103:2181,hadoop104:2181";

// 超时时间

private int sessionTimeout = 2000; private ZooKeeper zk;

private String rootNode = "locks"; private String subNode = "seq-";

// 当前 client 等待的子节点

private String waitPath;

//ZooKeeper 连接

private CountDownLatch connectLatch = new CountDownLatch(1);

//ZooKeeper 节点等待

private CountDownLatch waitLatch = new CountDownLatch(1);

// 当前 client 创建的子节点

private String currentNode;

// 和 zk 服务建立连接，并创建根节点

public DistributedLock() throws IOException, InterruptedException, KeeperException {

zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {

@Override

public void process(WatchedEvent event) {

// 连接建立时, 打开 latch, 唤醒 wait 在该 latch 上的线程

if (event.getState() == Event.KeeperState.SyncConnected) {

connectLatch.countDown();

}

// 发生了 waitPath 的删除事件

if (event.getType() == Event.EventType.NodeDeleted && event.getPath().equals(waitPath))

{

waitLatch.countDown();

}

}

});

// 等待连接建立

connectLatch.await();

//获取根节点状态

Stat stat = zk.exists("/" + rootNode, false);

//如果根节点不存在，则创建根节点，根节点类型为永久节点if (stat == null) {

System.out.println("根节点不存在");

zk.create("/" + rootNode, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);

}

}

// 加锁方法

public void zkLock() {

try {

//在根节点下创建临时顺序节点，返回值为创建的节点路径currentNode = zk.create("/" + rootNode + "/" + subNode,

null, ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL\_SEQUENTIAL);

// wait 一小会, 让结果更清晰一些

Thread.sleep(10);

// 注意, 没有必要监听"/locks"的子节点的变化情况

List<String> childrenNodes = zk.getChildren("/" + rootNode, false);

// 列表中只有一个子节点, 那肯定就是 currentNode , 说明

client 获得锁

if (childrenNodes.size() == 1) { return;

} else {

//对根节点下的所有临时顺序节点进行从小到大排序Collections.sort(childrenNodes);

//当前节点名称

String thisNode = currentNode.substring(("/" + rootNode + "/").length());

//获取当前节点的位置

int index = childrenNodes.indexOf(thisNode);

if (index == -1) { System.out.println("数据异常");

} else if (index == 0) {

// index == 0, 说明 thisNode 在列表中最小, 当前

client 获得锁

return;

} else {

// 获得排名比 currentNode 前 1 位的节点

this.waitPath childrenNodes.get(index - 1);

=

"/" + rootNode + "/" +

// 在 waitPath 上注册监听器, 当 waitPath 被删除时,

zookeeper 会回调监听器的 process 方法

zk.getData(waitPath, true, new Stat());

//进入等待锁状态waitLatch.await();

return;

}

}

} catch (KeeperException e) { e.printStackTrace();

} catch (InterruptedException e) { e.printStackTrace();

}

}

// 解锁方法

public void zkUnlock() { try {

zk.delete(this.currentNode, -1);

} catch (InterruptedException | KeeperException e) { e.printStackTrace();

}

}

}

### 分布式锁测试

（1）创建两个线程

package com.atguigu.lock2;

import org.apache.zookeeper.KeeperException; import java.io.IOException;

public class DistributedLockTest {

public

static

void

main(String[]

args)

throws

InterruptedException, IOException, KeeperException {

// 创建分布式锁 1

final DistributedLock lock1 = new DistributedLock();

// 创建分布式锁 2

final DistributedLock lock2 = new DistributedLock();

new Thread(new Runnable() { @Override

public void run() {

// 获取锁对象

try {

lock1.zkLock(); System.out.println("线程 1 获取锁");

Thread.sleep(5 \* 1000);

lock1.zkUnlock(); System.out.println("线程 1 释放锁");

} catch (Exception e) { e.printStackTrace();

}

}

}).start();

new Thread(new Runnable() { @Override

public void run() {

// 获取锁对象

try {

lock2.zkLock(); System.out.println("线程 2 获取锁");

Thread.sleep(5 \* 1000);

lock2.zkUnlock(); System.out.println("线程 2 释放锁");

} catch (Exception e) { e.printStackTrace();

}

}

}).start();

}

}

（2）观察控制台变化： 线程 1 获取锁

线程 1 释放锁

线程 2 获取锁

线程 2 释放锁

## Curator 框架实现分布式锁案例

### 原生的 Java API 开发存在的问题

* 1. 会话连接是异步的，需要自己去处理。比如使用 CountDownLatch
  2. Watch 需要重复注册，不然就不能生效
  3. 开发的复杂性还是比较高的
  4. 不支持多节点删除和创建。需要自己去递归

1. **Curator 是一个专门解决分布式锁的框架，解决了原生 Java API 开发分布式遇到的问题。**详情请查看官方文档：<https://curator.apache.org/index.html>

### Curator 案例实操

1. 添加依赖

<dependency>

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

<artifactId>curator-framework</artifactId>

<version>4.3.0</version>

</dependency>

<dependency>

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

<artifactId>curator-recipes</artifactId>

<version>4.3.0</version>

</dependency>

<dependency>

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

<artifactId>curator-client</artifactId>

<version>4.3.0</version>

</dependency>

1. 代码实现

package com.atguigu.lock;

import org.apache.curator.RetryPolicy;

import org.apache.curator.framework.CuratorFramework; import org.apache.curator.framework.CuratorFrameworkFactory; import

org.apache.curator.framework.recipes.locks.InterProcessLock; import org.apache.curator.framework.recipes.locks.InterProcessMutex; import org.apache.curator.retry.ExponentialBackoffRetry;

public class CuratorLockTest {

private String rootNode = "/locks";

// zookeeper server 列表

private String

connectString

=

"hadoop102:2181,hadoop103:2181,hadoop104:2181";

// connection 超时时间

private int connectionTimeout = 2000;

// session 超时时间

private int sessionTimeout = 2000; public static void main(String[] args) {

new CuratorLockTest().test();

}

// 测试

private void test() {

// 创建分布式锁 1

final InterProcessLock lock1 = new InterProcessMutex(getCuratorFramework(), rootNode);

// 创建分布式锁 2

final

InterProcessLock

lock2

=

new

InterProcessMutex(getCuratorFramework(), rootNode);

new Thread(new Runnable() { @Override

public void run() {

// 获取锁对象

try {

lock1.acquire(); System.out.println("线程 1 获取锁");

// 测试锁重入

lock1.acquire(); System.out.println("线程 1 再次获取锁");

Thread.sleep(5 \* 1000); lock1.release(); System.out.println("线程 1 释放锁"); lock1.release(); System.out.println("线程 1 再次释放锁");

} catch (Exception e) { e.printStackTrace();

}

}

}).start();

new Thread(new Runnable() { @Override

public void run() {

// 获取锁对象

try {

lock2.acquire(); System.out.println("线程 2 获取锁");

// 测试锁重入

lock2.acquire();

System.out.println("线程 2 再次获取锁");

Thread.sleep(5 \* 1000); lock2.release(); System.out.println("线程 2 释放锁"); lock2.release(); System.out.println("线程 2 再次释放锁");

} catch (Exception e) { e.printStackTrace();

}

}

}).start();

}

// 分布式锁初始化

public CuratorFramework getCuratorFramework (){

//重试策略，初试时间 3 秒，重试 3 次

RetryPolicy policy = new ExponentialBackoffRetry(3000, 3);

//通过工厂创建 Curator

CuratorFramework client = CuratorFrameworkFactory.builder()

.connectString(connectString)

.connectionTimeoutMs(connectionTimeout)

.sessionTimeoutMs(sessionTimeout)

.retryPolicy(policy).build();

//开启连接client.start();

System.out.println("zookeeper 初始化完成..."); return client;

}

}

（2）观察控制台变化： 线程 1 获取锁

线程 1 再次获取锁

线程 1 释放锁

线程 1 再次释放锁

线程 2 获取锁

线程 2 再次获取锁

线程 2 释放锁

线程 2 再次释放锁

# 第 6 章 企业面试真题（面试重点）

## 选举机制

半数机制，超过半数的投票通过，即通过。

* + 1. 第一次启动选举规则：

投票过半数时，服务器 id 大的胜出

* + 1. 第二次启动选举规则：

①EPOCH 大的直接胜出

②EPOCH 相同，事务 id 大的胜出

③事务 id 相同，服务器 id 大的胜出

## 生产集群安装多少 zk 合适？

安装奇数台。生产经验：

* 10 台服务器：3 台 zk；
* 20 台服务器：5 台 zk；
* 100 台服务器：11 台 zk；
* 200 台服务器：11 台 zk

服务器台数多：好处，提高可靠性；坏处：提高通信延时

## 常用命令

ls、get、create、delete