

分布式技术-Zookeeper

---- 老孙

课程目标：

- 1、Zookeeper概述
- 2、Zookeeper本地模式安装
- 3、Zookeeper内部原理
- 4、Zookeeper实战

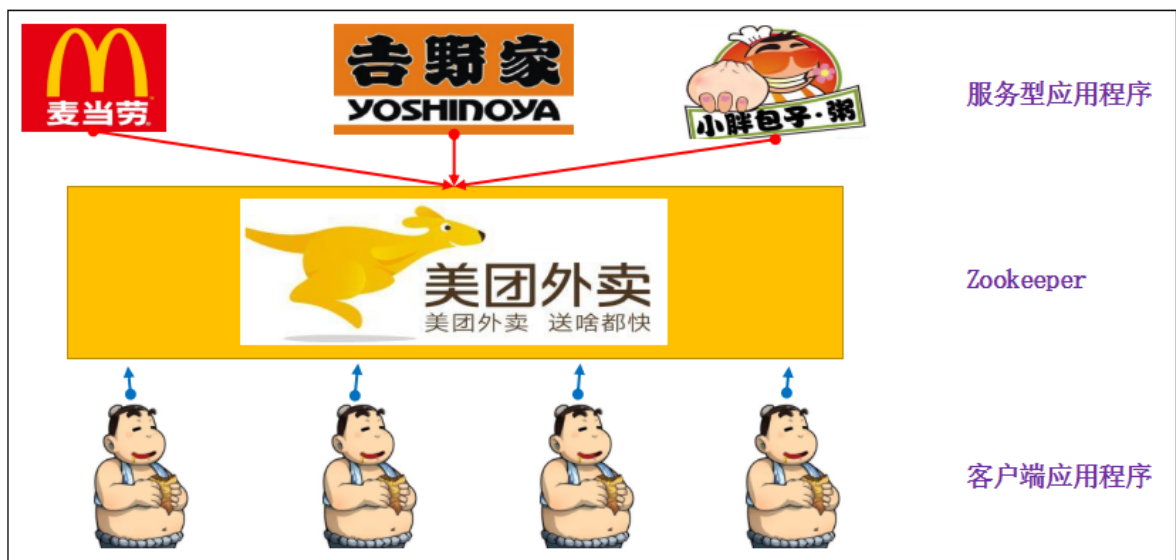
1. Zookeeper概述

1.1 概述

- 美团，饿了么，淘宝，58同城等等应用都是zookeeper的现实生活版
- 老孙我开了个饭店，如何才能让大家都能吃到我们的饭菜？需要入驻美团，这样大家就可以在美团app中看到我的饭店，下订单，从而完成一次交易
- Zookeeper是一个开源的分布式（多台服务器干一件事）的，为分布式应用提供**协调服务**的Apache项目。
- 在大数据技术生态圈中，zookeeper（动物管理员），Hadoop（大象），Hive（蜜蜂），Pig（猪）等技术

1.2 工作机制

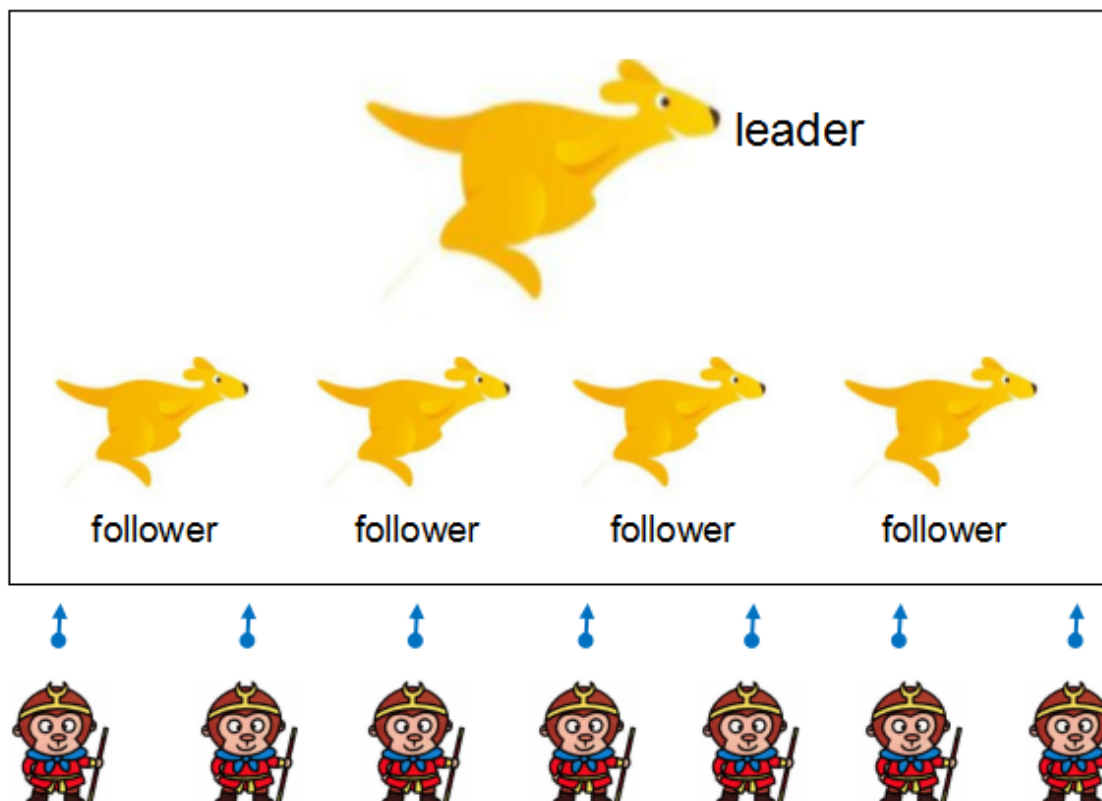
- Zookeeper从设计模式角度来理解：是一个基于**观察者模式**（一个人干活，有人盯着他）设计的分布式服务管理框架
- 它负责 存储 和 管理 大家都关心的数据
 - 然后接受观察者的**注册**，一旦这些数据发生变化
 - Zookeeper就将负责**通知**已经注册的那些观察者做出相应的反应
 - 从而实现集群中类似Master/Slave管理模式
- Zookeeper = 文件系统 + 通知机制



1. 商家营业并入驻
2. 获取到当前营业的饭店列表
3. 服务器节点下线
4. 服务器节点上下线事件通知
5. 重新再去获取服务器列表，并注册监听

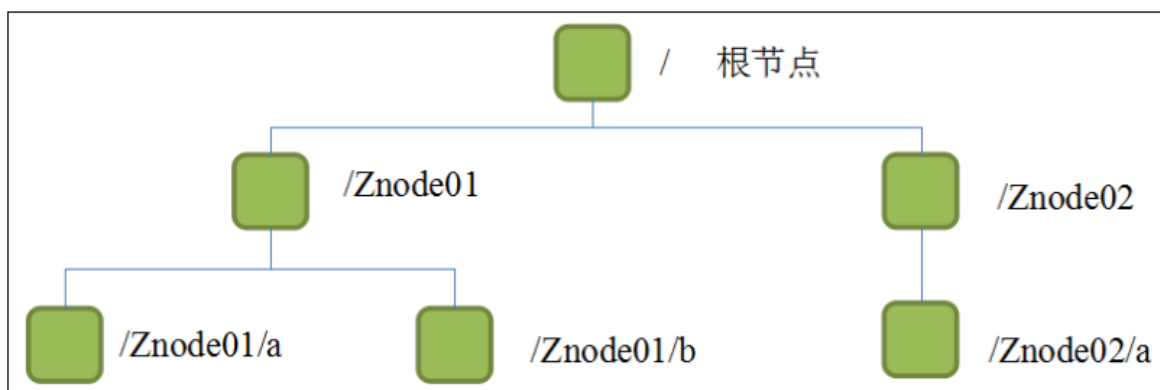
1.3 特点

- 分布式和集群的区别？
 - 无论分布式和集群，都是很多人在做事情。具体区别如下：
 - 例如：我有一个饭店，越来越火爆，我得多招聘一些工作人员
 - 分布式：招聘1个厨师，1个服务员，1个前台，**三个人负责的工作不一样**，但是最终目的都是为饭店工作
 - 集群：招聘3个服务员，**3个人的工作一样**



1. 是一个leader和多个follower来组成的集群（狮群中，一头雄狮，N头母狮）
2. 集群中只要有半数以上的节点存活，Zookeeper就能正常工作（5台服务器挂2台，没问题；4台服务器挂2台，就停止）
3. 全局数据一致性，每台服务器都保存一份相同的数据副本，无论client连接哪台server，数据都是一致的
4. 数据更新原子性，一次数据要么成功，要么失败（不成功便成仁）
5. 实时性，在一定时间范围内，client能读取到最新数据
6. 更新的请求按照顺序执行，会按照发送过来的顺序，逐一执行（发来123，执行123，而不是321或者别的）

1.4 数据结构



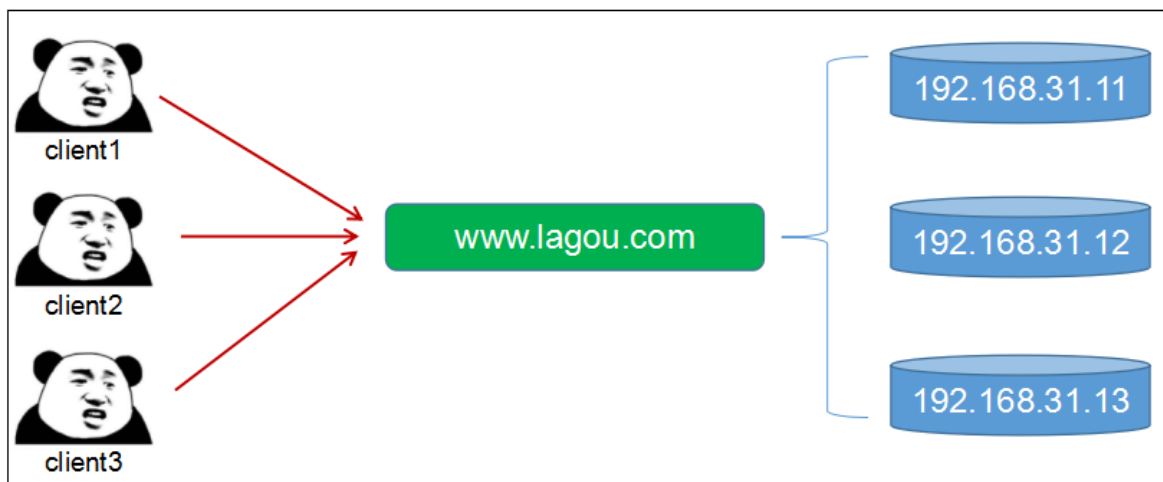
- ZooKeeper数据模型的结构与linux文件系统很类似，整体上可以看作是一棵树，每个节点称做一个ZNode（ZookeeperNode）。
- 每一个ZNode默认能够存储1MB的数据（元数据），每个ZNode的路径都是唯一的
 - 元数据（Metadata），又称中介数据、中继数据，为描述数据的数据（data about data），主要是描述数据属性（property）的信息，用来支持如指示存储位置、历史数据、资源查找、文件记录等功能

1.5 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

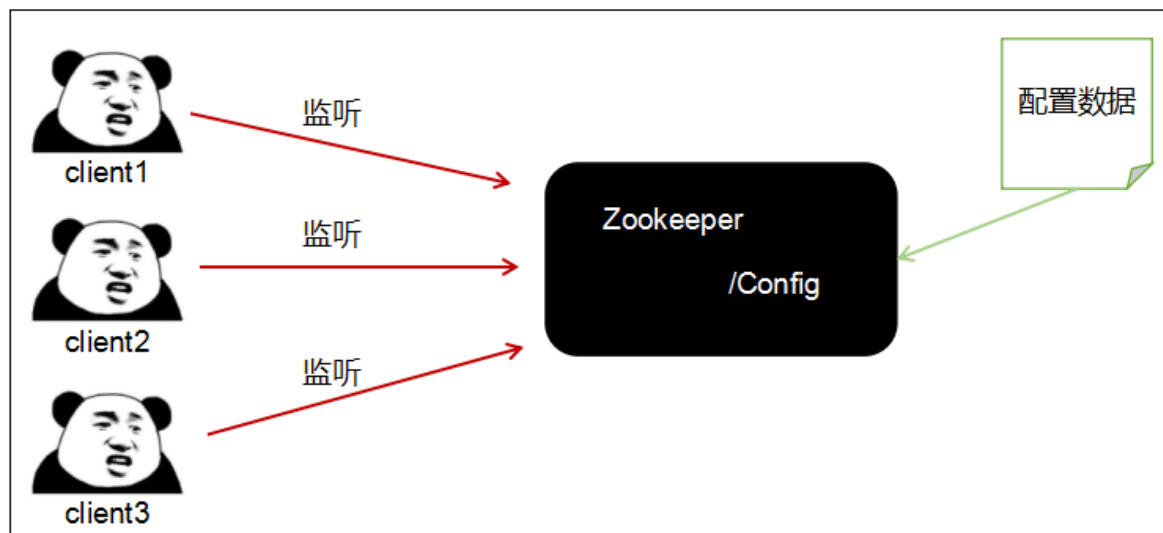
1.5.1 统一命名服务

- 在分布式环境下，通常需要对应用或服务进行统一的命名，便于识别
- 例如：服务器的IP地址不容易记，但域名相比之下却是很容易记住



1.5.2 统一配置管理

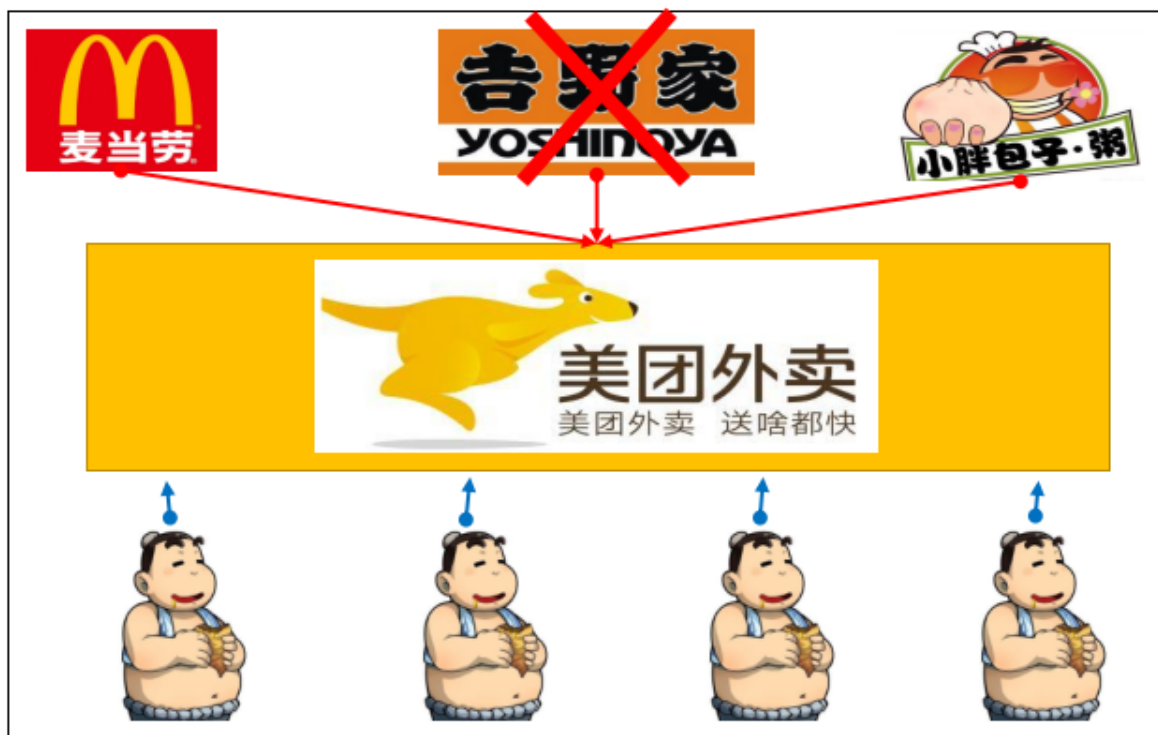
- 分布式环境下，配置文件做同步是必经之路
- 1000台服务器，如果配置文件作出修改，那一台一台的修改，运维人员肯定会疯，如何做到修改一处就快速同步到每台服务器上



- 将配置管理交给Zookeeper
 - 1、将配置信息写入到Zookeeper的某个节点上
 - 2、每个客户端都监听这个节点
 - 3、一旦节点中的数据文件被修改，Zookeeper这个话匣子就会通知每台客户端服务器

1.5.3 服务器节点动态上下线

- 客户端能实时获取服务器上下线的变化
- 在美团APP上实时可以看到商家是否正在营业或打样






1.5.4 软负载均衡

- Zookeeper会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请求（雨露均沾）
- 都是自己的孩子，得一碗水端平



1.6 下载地址

镜像库地址：<http://archive.apache.org/dist/zookeeper/>

	zookeeper-3.5.8/	2020-05-11 10:10
	zookeeper-3.6.0/	2020-03-03 21:30
	zookeeper-3.6.1/	2020-04-30 19:53



Parent Directory			
	apache-zookeeper-3.6.0-bin.tar.gz	2020-03-03 21:30	12M
	apache-zookeeper-3.6.0-bin.tar.gz.asc	2020-03-03 21:30	488
	apache-zookeeper-3.6.0-bin.tar.gz.sha512	2020-03-03 21:30	163
	apache-zookeeper-3.6.0.tar.gz	2020-03-03 21:30	3.2M
	apache-zookeeper-3.6.0.tar.gz.asc	2020-03-03 21:30	488
	apache-zookeeper-3.6.0.tar.gz.sha512	2020-03-03 21:30	159

- apache-zookeeper-3.6.0.tar.gz需要安装maven，然后再运行mvn clean install 和mvn javadoc:aggregate，前一个命令会下载安装好多jar包，不知道要花多长时间
- apache-zookeeper-3.6.0-bin.tar.gz已经自带所需要的各种jar包

2. Zookeeper本地模式安装

2.1 本地模式安装

2.1.1 安装前准备

1. 安装jdk
2. 拷贝apache-zookeeper-3.6.0-bin.tar.gz到opt目录
3. 解压安装包

```
[root@localhost opt]# tar -zxvf apache-zookeeper-3.6.0-bin.tar.gz
```

4. 重命名

```
[root@localhost opt]# mv apache-zookeeper-3.6.0-bin zookeeper
```

2.1.2 配置修改

1. 在/opt/zookeeper/这个目录上创建zkData和zkLog目录

```
[root@localhost zookeeper]# mkdir zkData  
[root@localhost zookeeper]# mkdir zkLog
```

2. 进入/opt/zookeeper/conf这个路径，复制一份 zoo_sample.cfg 文件并命名为 zoo.cfg

```
[root@localhost conf]# cp zoo_sample.cfg zoo.cfg
```

3. 编辑zoo.cfg文件，修改dataDir路径：

```
dataDir=/opt/zookeeper/zkData  
dataLogDir=/opt/zookeeper/zkLog
```

2.1.3 操作Zookeeper

1. 启动Zookeeper

```
[root@localhost bin]# ./zkServer.sh start
```

2. 查看进程是否启动

```
[root@localhost bin]# jps
```

QuorumPeerMain：是zookeeper集群的启动入口类，是用来加载配置启动QuorumPeer线程的

3. 查看状态：

```
[root@localhost bin]# ./zkServer.sh status
```

4. 启动客户端

```
[root@localhost bin]# ./zkCli.sh
```

5. 退出客户端

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

2.2 配置参数解读

Zookeeper中的配置文件zoo.cfg中参数含义解读如下：

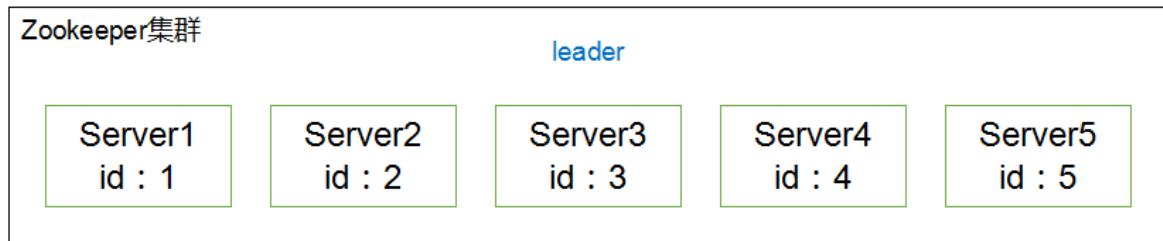
- **tickTime = 2000**：通信心跳数，Zookeeper服务器与客户端心跳时间，单位毫秒
 - Zookeeper使用的基本时间，服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个tickTime时间就会发送一个心跳，时间单位为毫秒。
- **initLimit = 10**：LF初始通信时限
 - 集群中的Follower跟随者服务器与Leader领导者服务器之间，**启动时**能容忍的最多心跳数
 - 10*2000（10个心跳时间）如果领导和跟随者没有发出心跳通信，就视为失效的连接，领导和跟随者彻底断开
- **syncLimit = 5**：LF同步通信时限
 - 集群**启动后**，Leader与Follower之间的最大响应时间单位，假如响应超过syncLimit * tickTime->10秒，Leader就认为Follower已经死掉，会将Follower从服务器列表中删除
- **dataDir**：数据文件目录+数据持久化路径
 - 主要用于保存Zookeeper中的数据。
- **dataLogDir**：日志文件目录
- **clientPort = 2181**：客户端连接端口
 - 监听客户端连接的端口。

3. Zookeeper内部原理

3.1 选举机制（面试重点）

- **半数机制**：集群中半数以上机器存活，集群可用。所以Zookeeper适合安装奇数台服务器

- 虽然在配置文件中并没有指定Master和Slave。但是，Zookeeper工作时，是有一个节点为Leader，其他则为Follower，Leader是通过内部的选举机制临时产生的



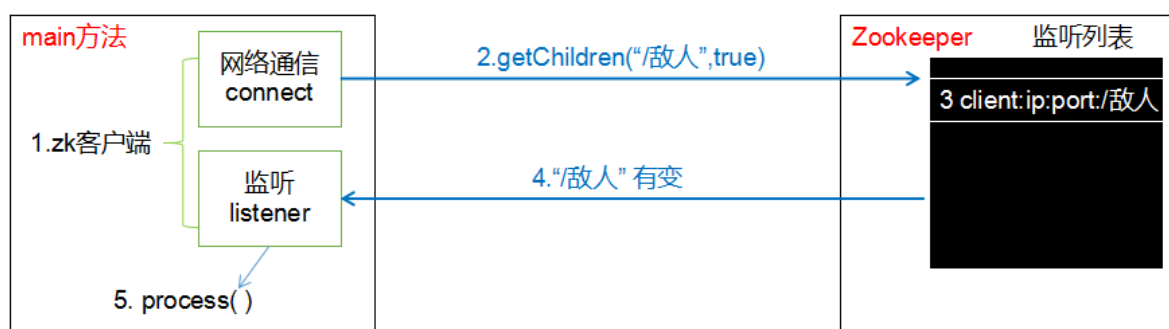
1. Server1先投票，**投给自己**，自己为1票，没有超过半数，根本无法成为leader，顺水推舟将票数投给了id比自己大的Server2
2. Server2也把自己的票数投给了自己，再加上Server1给的票数，总票数为2票，没有超过半数，也无法成为leader，也学习Server1，顺水推舟，将自己所有的票数给了id比自己大的Server3
3. Server3得到了Server1和Server2的两票，再加上自己投给自己的一票。3票超过半数，顺利成为leader
4. Server4和Server5都投给自己，但是无法改变Server3的票数，只好听天由命，承认Server3是leader

3.2 节点类型

- 持久型 (persistent) :
 - 持久化目录节点 (persistent) 客户端与zookeeper断开连接后，该节点依旧存在
 - 持久化顺序编号目录节点 (persistent_sequential) 客户端与zookeeper断开连接后，该节点依旧存在，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，例如：Znode001，Znode002...
- 短暂型 (ephemeral) :
 - 临时目录节点 (ephemeral) 客户端和服务端断开连接后，创建的节点自动删除
 - 临时顺序编号目录节点 (ephemeral_sequential) 客户端与zookeeper断开连接后，该节点被删除，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，例如：Znode001，Znode002...

注意：序号是相当于i++，和数据库中的自增长类似

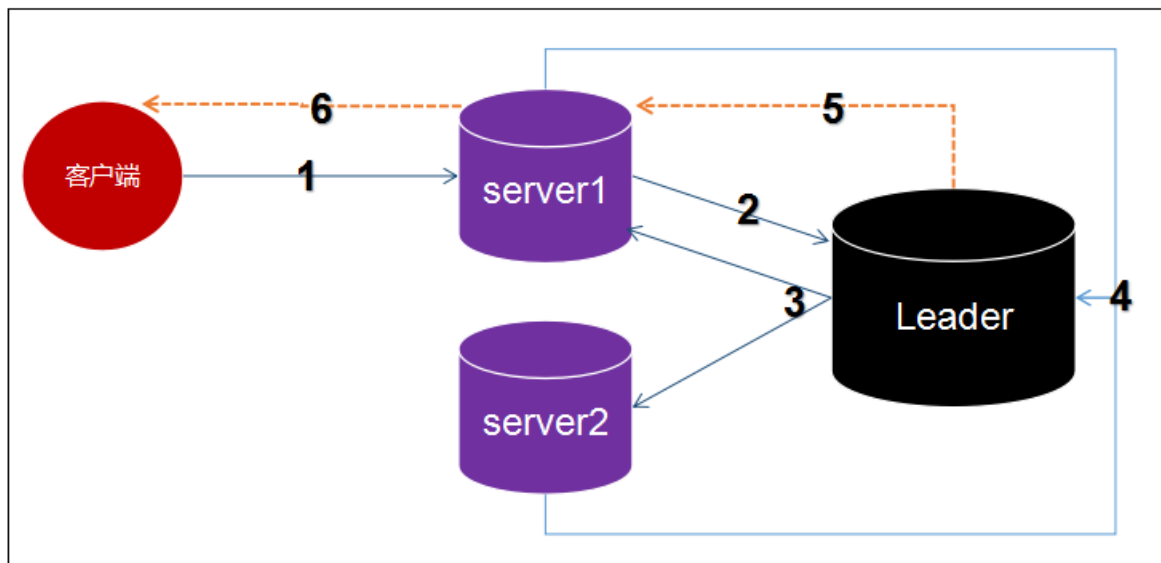
3.3 监听器原理（面试重点）



1. 在main方法中创建Zookeeper客户端的同时就会创建两个线程，一个负责网络连接通信，一个负责监听
2. 监听事件就会通过网络通信发送给zookeeper
3. zookeeper获得注册的监听事件后，立刻将监听事件添加到监听列表里
4. zookeeper监听到 数据变化 或 路径变化，就会将这个信息发送给监听线程

- 常见的监听
 1. 监听节点数据的变化：get path [watch]
 2. 监听子节点增减的变化：ls path [watch]
- 5. 监听线程就会在内部调用process方法（需要我们实现process方法内容）

3.4 写数据流程



1. Client 想向 ZooKeeper 的 Server1 上写数据，必须先发送一个写的请求
2. 如果Server1不是Leader，那么Server1 会把接收到的请求进一步转发给Leader。
3. 这个Leader 会将写请求广播给各个Server，各个Server写成功后就会通知Leader。
4. 当Leader收到半数以上的 Server 数据写成功了，那么就说明数据写成功了。
5. 随后，Leader会告诉Server1数据写成功了。
6. Server1会反馈通知 Client 数据写成功了，整个流程结束

4. Zookeeper实战（开发重点）

4.1 分布式安装部署

集群思路：先搞定一台服务器，再克隆出两台，形成集群！

4.1.1 安装zookeeper

请参考本文 2.1

4.1.2 配置服务器编号

- 在/opt/zookeeper/zkData创建myid文件

```
[root@localhost zkData]# vim myid
```

- 在文件中添加与server对应的编号：1
- 其余两台服务器分别对应2和3

4.1.3 配置zoo.cfg文件

- 打开zoo.cfg文件，增加如下配置

```
#####cluster#####
server.1=192.168.204.141:2888:3888
server.2=192.168.204.142:2888:3888
server.3=192.168.204.143:2888:3888
```

- 配置参数解读 server.A=B:C:D
 - **A**：一个数字，表示第几号服务器
集群模式下配置的/opt/zookeeper/zkData/myid文件里面的数据就是A的值
 - **B**：服务器的ip地址
 - **C**：与集群中Leader服务器交换信息的端口
 - **D**：选举时专用端口，万一集群中的Leader服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

4.1.4 配置其余两台服务器

1. 在虚拟机数据目录vms下，创建zk02
2. 将本台服务器数据目录下的.vmx文件和所有的.vmdk文件分别拷贝zk02下
3. 虚拟机->文件->打开（选择zk02下的.vmx文件）
4. 开启此虚拟机，弹出对话框，选择“我已复制该虚拟机”
5. 进入系统后，修改linux中的ip，修改/opt/zookeeper/zkData/myid中的数值为2

第三台服务器zk03，重复上面的步骤

4.1.5 集群操作

1. 每台服务器的防火墙**必须关闭**

```
[root@localhost bin]# systemctl stop firewalld.service
```

2. 启动第1台

```
[root@localhost bin]# ./zkServer.sh start
```

3. 查看状态

```
[root@localhost bin]# ./zkServer.sh status
```

```
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper/bin/../conf/zoo.cfg
Client port found: 2181. Client address: localhost.
Error contacting service. It is probably not running.
```

注意：因为没有超过半数以上的服务器，所以集群失败（防火墙没有关闭也会导致失败）

4. 当启动第2台服务器时

- 查看第1台的状态：Mode: **follower**
- 查看第2台的状态：Mode: **leader**

4.2 客户端命令行操作

- 启动客户端

```
[root@localhost bin]# ./zkCli.sh
```

- 显示所有操作命令

```
help
```

- 查看当前znode中所包含的内容

```
ls /
```

- 查看当前节点详细数据

zookeeper老版本使用 ls2 / ，现在已经被新命令替代

```
ls -s /
```

- cZxid：创建节点的事务
 - 每次修改ZooKeeper状态都会收到一个zxid形式的时间戳，也就是ZooKeeper事务ID。
 - 事务ID是ZooKeeper中所有修改总的次序。
 - 每个修改都有唯一的zxid，如果zxid1小于zxid2，那么zxid1在zxid2之前发生。
- ctime：被创建的毫秒数(从1970年开始)
- mZxid：最后更新的事务zxid
- mtime：最后修改的毫秒数(从1970年开始)
- pZxid：最后更新的子节点zxid
- cversion：创建版本号，子节点修改次数
- dataVersion：数据变化版本号
- aclVersion：权限版本号
- 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。
- dataLength：数据长度
- numChildren：子节点数

- 分别创建2个普通节点

- 在根目录下，创建中国和美国两个节点

```
create /china  
create /usa
```

- 在根目录下，创建俄罗斯节点，并保存“普京”数据到节点上

```
create /ru "pujing"
```

- 多级创建节点
 - 在日本下，创建东京“热”
 - japan必须提前创建好，否则报错“节点不存在”

```
create /japan/Tokyo "hot"
```

- 获得节点的值

```
get /japan/Tokyo
```

- 创建短暂节点：创建成功之后，quit退出客户端，重新连接，短暂的节点消失

```
create -e /uk  
ls /  
quit  
ls /
```

- 创建带序号的节点

- 在俄罗斯ru下，创建3个city

```
create -s /ru/city # 执行三次  
ls /ru  
[city0000000000, city0000000001, city0000000002]
```

- 如果原来没有序号节点，序号从0开始递增。
- 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

- 修改节点数据值

```
set /japan/Tokyo "too hot"
```

- 监听 **节点的值变化** 或 **子节点变化（路径变化）**

1. 在server3主机上注册监听/usa节点的数据变化

```
addwatch /usa
```

2. 在Server1主机上修改/usa的数据

```
set /usa "telangpu"
```

3. Server3会立刻响应

WatchedEvent state:SyncConnected type:**NodeDataChanged** path:/usa

4. 如果在Server1的/usa下面创建子节点NewYork

```
create /usa/NewYork
```

5. Server3会立刻响应

WatchedEvent state:SyncConnected type:**NodeCreated** path:/usa/NewYork

- 删除节点

```
delete /usa/NewYork
```

- 递归删除节点（非空节点，节点下有子节点）

```
deleteall /ru
```

不仅删除/ru，而且/ru下的所有子节点也随之删除

4.3 API应用

4.3.1 IDEA环境搭建

1. 创建一个Maven工程
2. 添加pom文件

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.8.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.6.0</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
</dependencies>
```

3. 在resources下创建log4j.properties

```
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/zk.log
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
```

4.3.2 创建ZooKeeper客户端

```

public class TestZK {
    // 集群ip
    private String connStr
    ="192.168.249.81:2181,192.168.249.82:2181,192.168.249.83:2181";

    /*
    session超时 60秒：一定不能太少，因为连接zookeeper和加载集群环境会因为性能原因延迟略高
    如果时间太少，还没有创建好客户端，就开始操作节点，会报错的
    */
    private int sessionTimeout = 60000;

    @Test
    public void init() throws IOException {
        // 创建监听器
        watcher watcher = new Watcher() {
            public void process(WatchedEvent watchedEvent) {

            }
        };

        // 创建zookeeper客户端
        ZooKeeper zk = new ZooKeeper(connStr, sessionTimeout, watcher);
    }
}

```

4.3.3 创建节点

- 一个ACL对象就是一个Id和permission对
 - 表示哪个/哪些范围的Id (Who) 在通过了怎样的鉴权 (How) 之后，就允许进行那些操作 (What) : Who How What ;
 - permission (What) 就是一个int表示的位码，每一位代表一个对应操作的允许状态。
 - 类似linux的文件权限，不同的是共有5种操作：CREATE、READ、WRITE、DELETE、ADMIN(对应更改ACL的权限)
 - OPEN_ACL_UNSAFE：创建开放节点，允许任意操作（用的最少，其余的权限用的很少）
 - READ_ACL_UNSAFE：创建只读节点
 - CREATOR_ALL_ACL：创建者才有全部权限

```

@Before
public void init() throws IOException{
    // 省略...
}

@Test
public void createNode() throws Exception {
    String nodeCreated = zkCli.create("/lagou", "laosun".getBytes(),
    Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
    // 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
    System.out.println("nodeCreated = " + nodeCreated);
}

```

4.3.4 查询节点的值

```
@Test
public void find() throws Exception{
    byte[] bs = zkCli.getData("/lagou", false, new Stat()); // 路径不存在时会报错
    String data = new String(bs);
    System.out.println("查询到数据: "+data);
}
```

4.3.5 修改节点的值

```
@Test
public void update()throws Exception{
    Stat stat = zkCli.setData("/lagou", "laosunA".getBytes(), 0); //先查看节点详情, 获得dataVersion = 0
    System.out.println(stat);
}
```

4.3.6 删除节点

```
@Test
public void delete() throws Exception {
    zkCli.delete("/lagou", 1); // 先查看节点详情, 获得dataVersion = 1
    System.out.println("删除成功! ");
}
```

4.3.7 获取子节点

```
@Test
public void getChildren() throws Exception {
    List<String> children = zkCli.getChildren("/", false); // false:不监听
    for (String child : children) {
        System.out.println(child);
    }
}
```

4.3.8 监听子节点的变化

```
@Test
public void getChildren() throws Exception {
    List<String> children = zkCli.getChildren("/", true); // true: 注册监听
    for (String child : children) {
        System.out.println(child);
    }
    // 让线程不停止, 等待监听的响应
    System.in.read();
}
```

- 程序在运行的过程中, 我们在linux下创建一个节点
- IDEA的控制台就会做出响应: **NodeChildrenChanged--/**

4.3.9 判断Znode是否存在

```

@Test
public void exist() throws Exception {
    Stat stat = zkCli.exists("/lagou", false);
    System.out.println(stat == null ? "不存在" : "存在");
}

```

4.4 案例-模拟美团商家上下线

4.4.1 需求

- 模拟美团服务平台，商家营业通知，商家打烊通知
- 提前在根节点下，创建好 /meituan 节点

4.4.2 商家服务类

```

public class ShopServer {

    private static String connectString =
"192.168.204.141:2181,192.168.204.142:2181,192.168.204.143:2181";
    private static int sessionTimeout = 60000;
    private ZooKeeper zk = null;

    // 创建到zk的客户端连接
    public void getConnect() throws IOException {
        zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {
            public void process(WatchedEvent event) {

            }
        });
    }

    // 注册到集群
    public void register(String ShopName) throws Exception {
        // 一定是"EPHEMERAL_SEQUENTIAL短暂有序型"的节点，才能给shop编号，shop1,
shop2..."
        String create = zk.create("/meituan/Shop", ShopName.getBytes(),
ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
        System.out.println("【"+ShopName+"】 开始营业! " + create);
    }

    // 业务功能
    public void business(String ShopName) throws Exception {
        System.out.println("【"+ShopName+"】 正在营业中 ...");
        System.in.read();
    }

    public static void main(String[] args) throws Exception {
        ShopServer shop = new ShopServer();
        // 1.连接zookeeper集群（和美团取得联系）
        shop.getConnect();

        // 2.将服务器节点注册（入住美团）
        shop.register(args[0]);
    }
}

```



```

        // 3.业务逻辑处理（做生意）
        shop.business(args[0]);
    }
}

```

4.4.3 客户类

```

public class Customers {
    private static String connectString =
"192.168.204.141:2181,192.168.204.142:2181,192.168.204.143:2181";
    private static int sessionTimeout = 60000;
    private ZooKeeper zk = null;

    // 创建到zk的客户端连接
    public void getConnect() throws IOException {
        zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {

            public void process(WatchedEvent event) {

                // 再次获取所有商家
                try {
                    getShopList();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

    // 获取服务器列表信息
    public void getShopList() 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> shops = zk.getChildren("/meituan", true);

        // 2存储服务器信息列表
        ArrayList<String> shoplist = new ArrayList();

        // 3遍历所有节点，获取节点中的主机名称信息
        for (String shop : shops) {
            byte[] data = zk.getData("/meituan/" + shop, false, new Stat());
            shoplist.add(new String(data));
        }

        // 4打印服务器列表信息
        System.out.println(shoplist);
    }

    // 业务功能
    public void business() throws Exception {
        System.out.println("客户正在浏览商家 ...");
        System.in.read();
    }

    public static void main(String[] args) throws Exception {
        // 1.获取zk连接 （客户打开美团）
    }
}

```

```

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

// 2.获取/meituan的子节点信息，从中获取服务器信息列表（从美团中获取商家列表）
client.getShopList();

// 3.业务进程启动（对比商家，点餐）
client.business();
}
}

```

1. 运行客户类，就会得到商家列表
2. 首先在linux中添加一个商家，然后观察客户端的控制台输出（商家列表会立刻更新出最新商家），多添加几个，也会实时输出商家列表

```

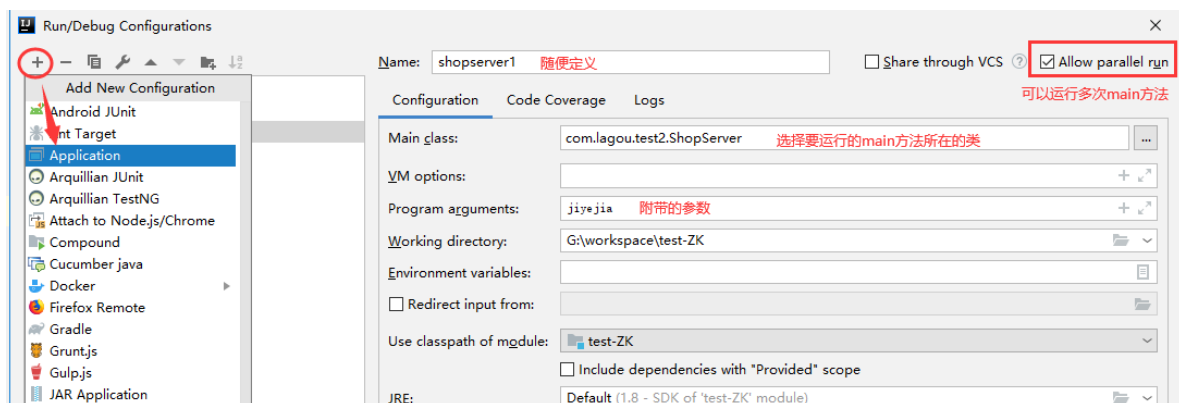
create /meituan/KFC "KFC"
create /meituan/BKC "BurgerKing"
create /meituan/baozi "baozi"

```

3. 在linux中删除商家，在客户端的控制台也会实时看到商家移除后的最新商家列表

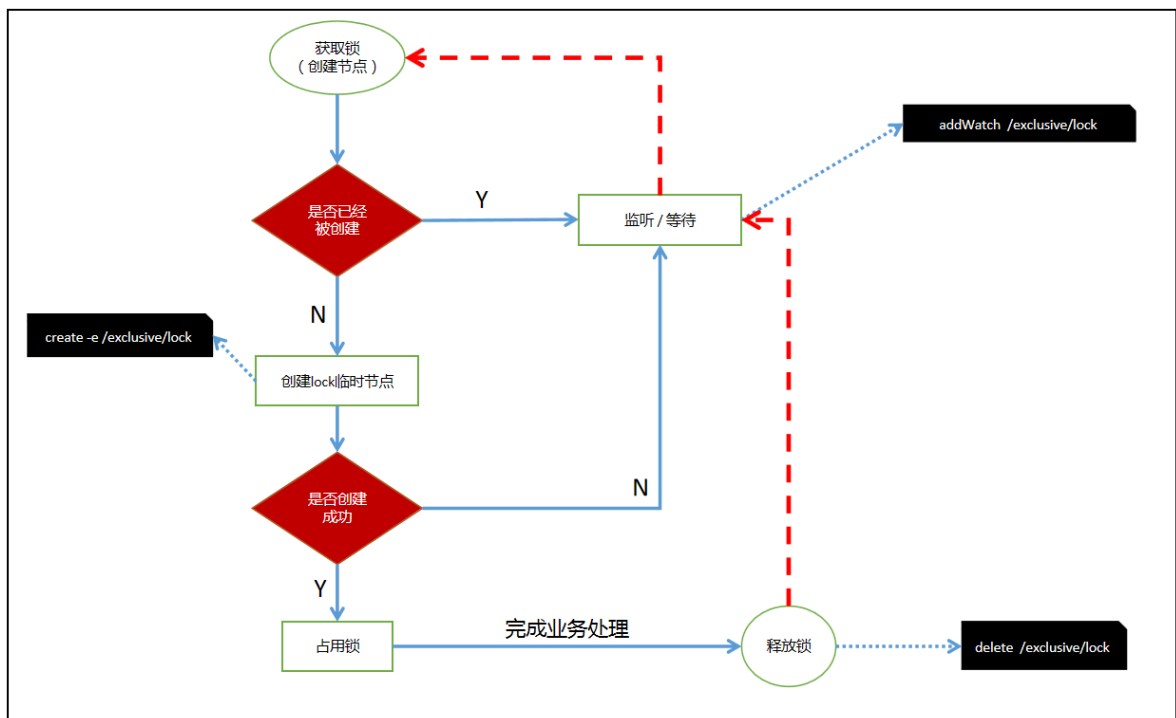
```
delete /meituan/baozi
```

4. 运行商家服务类（以main方法带参数的形式运行）

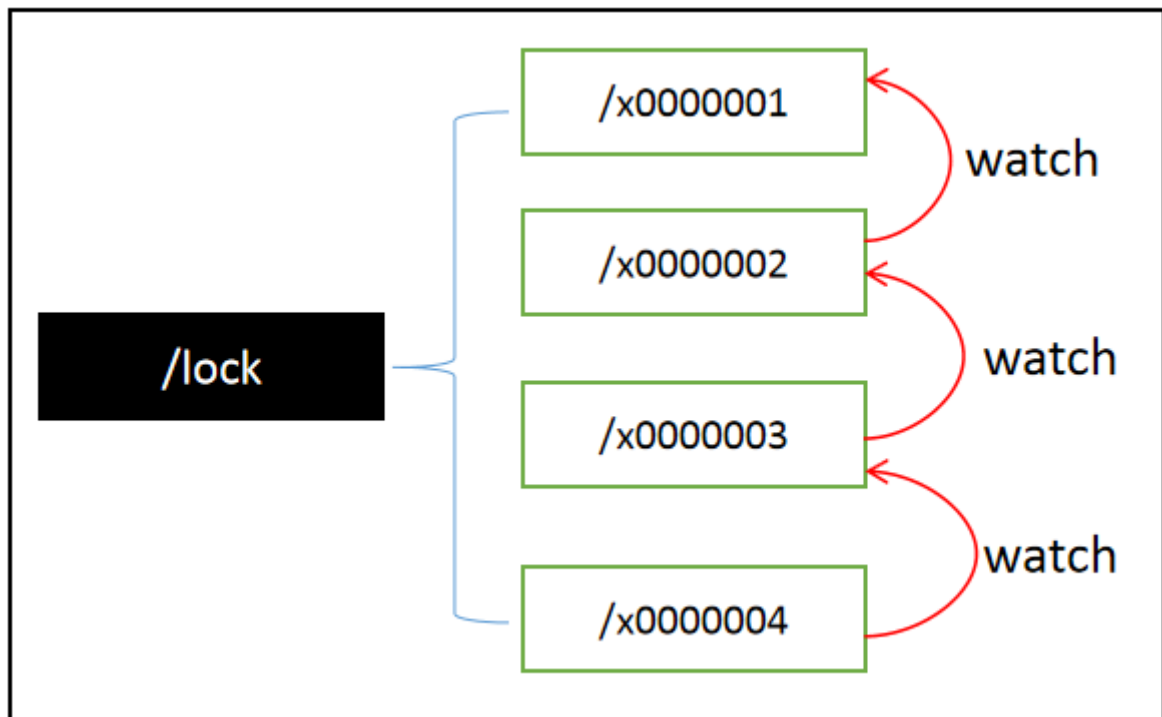


4.5 案例-分布式锁-商品秒杀

- 锁：我们在多线程中接触过，作用就是让当前的资源不会被其他线程访问！
 - 我的日记本，不可以被别人看到。所以要锁在保险柜中
 - 当我打开锁，将日记本拿走了，别人才能使用这个保险柜
- 在zookeeper中使用传统的锁引发的“羊群效应”：1000个人创建节点，只有一个人能成功，999人需要等待！
- 羊群是一种很散乱的组织，平时在一起也是盲目地左冲右撞，但一旦有一只头羊动起来，其他的羊也会不假思索地一哄而上，全然不顾旁边可能有的狼和不远处更好的草。羊群效应就是比喻人都有从众心理，从众心理很容易导致盲从，而盲从往往会陷入骗局或遭到失败。



- 避免“羊群效应”，zookeeper采用分布式锁



1. 所有请求进来，在/lock下创建 **临时顺序节点**，放心，zookeeper会帮你编号排序
2. 判断自己是不是/lock下**最小的节点**
 1. 是，获得锁（创建节点）
 2. 否，对前面小我一级的节点进行监听
3. 获得锁请求，处理完业务逻辑，释放锁（删除节点），后一个节点得到通知（比你年轻的死了，你成为最嫩的了）
4. 重复步骤2

实现步骤

1. 初始化数据库

创建数据库zkproduct，使用默认的字符集utf8

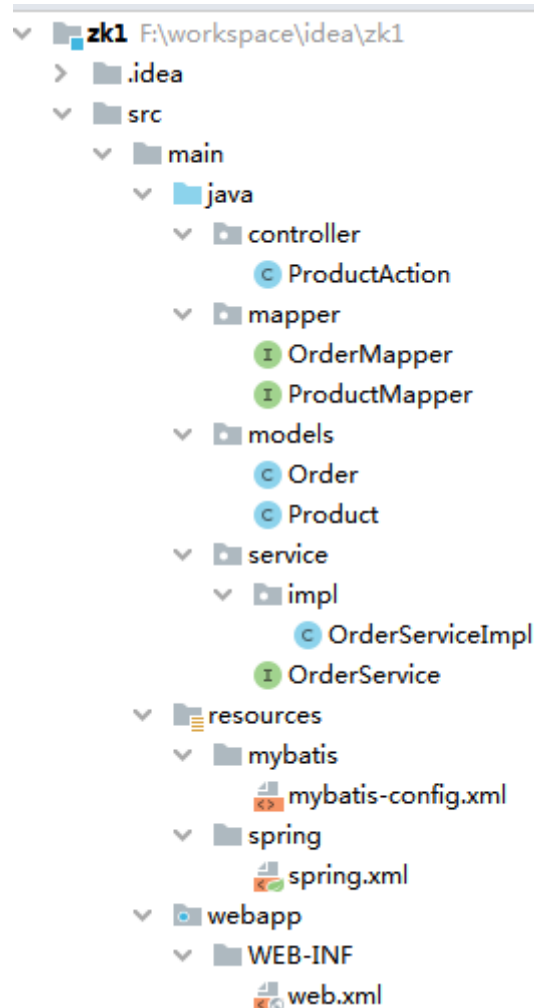
```
-- 商品表
create table product(
    id int primary key auto_increment, -- 商品编号
    product_name varchar(20) not null, -- 商品名称
    stock int not null, -- 库存
    version int not null -- 版本
)

insert into product (product_name,stock,version) values('锦鲤-清空购物车-大奖',5,0)
```

```
-- 订单表
create table `order`(
    id varchar(100) primary key, -- 订单编号
    pid int not null, -- 商品编号
    userid int not null -- 用户编号
)
```

2. 搭建工程

搭建ssm框架，对库存表-1，对订单表+1



```
<packaging>war</packaging>

<properties>
  <spring.version>5.2.7.RELEASE</spring.version>
</properties>

<dependencies>
  <!-- Spring -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <!-- Mybatis -->
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.5</version>
  </dependency>
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.5</version>
  </dependency>
  <!-- 连接池 -->
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.10</version>
  </dependency>
  <!-- 数据库 -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
  </dependency>
  <!-- junit -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
```

```

</dependencies>
<build>
  <plugins>
    <!-- maven内嵌的tomcat插件 -->
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <!-- 目前apache只提供了tomcat6和tomcat7两个插件 -->
      <artifactId>tomcat7-maven-plugin</artifactId>
      <configuration>
        <port>8001</port>
        <path>/</path>
      </configuration>
      <executions>
        <execution>
          <!-- 打包完成后,运行服务 -->
          <phase>package</phase>
          <goals>
            <goal>run</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <!-- 后台的日志输出: 针对开发者-->
  <settings>
    <setting name="logImpl" value="STDOUT_LOGGING"/>
  </settings>
</configuration>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">
  <!-- 1.扫描包下的注解 -->
  <context:component-scan base-package="controller,service,mapper"/>
  <!-- 2.创建数据连接池对象 -->
  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
    <property name="url" value="jdbc:mysql://192.168.204.131:3306/zkproduct?
serverTimezone=GMT" />
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="username" value="root" />

```

```

        <property name="password" value="123123" />
        <property name="maxActive" value="10" />
        <property name="minIdle" value="5" />
    </bean>
    <!-- 3.创建SqlSessionFactory, 并引入数据源对象 -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <property name="configLocation" value="classpath:mybatis-
config.xml"></property>
    </bean>
    <!-- 4.告诉spring容器, 数据库语句代码在哪个文件中-->
    <!-- mapper.xDao接口对应resources/mapper/xDao.xml-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="mapper"></property>
    </bean>
    <!-- 5.将数据源关联到事务 -->
    <bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <!-- 6.开启事务 -->
    <tx:annotation-driven/>
</beans>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">

    <servlet>
        <servlet-name>springMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring/spring.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>springMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

```

@Mapper
@Component
public interface OrderMapper {

    // 生成订单
    @Insert("insert into `order` (id,pid,userid) values (#{id},#{pid},#{userid})")
    int insert(Order order);
}

```

```

@Mapper
@Component
public interface ProductMapper {

    // 查询商品（目的查库存）
    @Select("select * from product where id = #{id}")
    Product getProduct(@Param("id") int id);

    // 减库存
    @Update("update product set stock = stock-1 where id = #{id}")
    int reduceStock(@Param("id") int id);
}

```

```

@Service
public class OrderServiceImpl implements OrderService {

    @Autowired
    ProductMapper productMapper;

    @Autowired
    OrderMapper orderMapper;

    @Override
    public void reduceStock(int id) throws Exception {

        // 1.获取库存
        Product product = productMapper.getProduct(id);

        // 模拟网络延迟
        Thread.sleep(1000);

        if(product.getStock() <= 0)
            throw new RuntimeException("已抢光!");

        // 2.减库存
        int i = productMapper.reduceStock(id);
        if(i == 1){
            Order order = new Order();
            order.setId(UUID.randomUUID().toString());
            order.setPid(id);
            order.setUserId(101);
            orderMapper.insert(order);
        }else
            throw new RuntimeException("减库存失败，请重试!");
    }
}

```



```
@Controller
public class ProductAction {

    @Autowired
    private OrderService orderService;

    @GetMapping("/product/reduce")
    @ResponseBody
    public Object reduceStock(int id) throws Exception{
        orderService.reduceStock(id);
        return "ok";
    }
}
```

3. 启动测试

1. 启动两次工程，端口号分别8001和8002
2. 使用nginx做负载均衡

```
upstream sga{
    server 192.168.204.1:8001;
    server 192.168.204.1:8002;
}

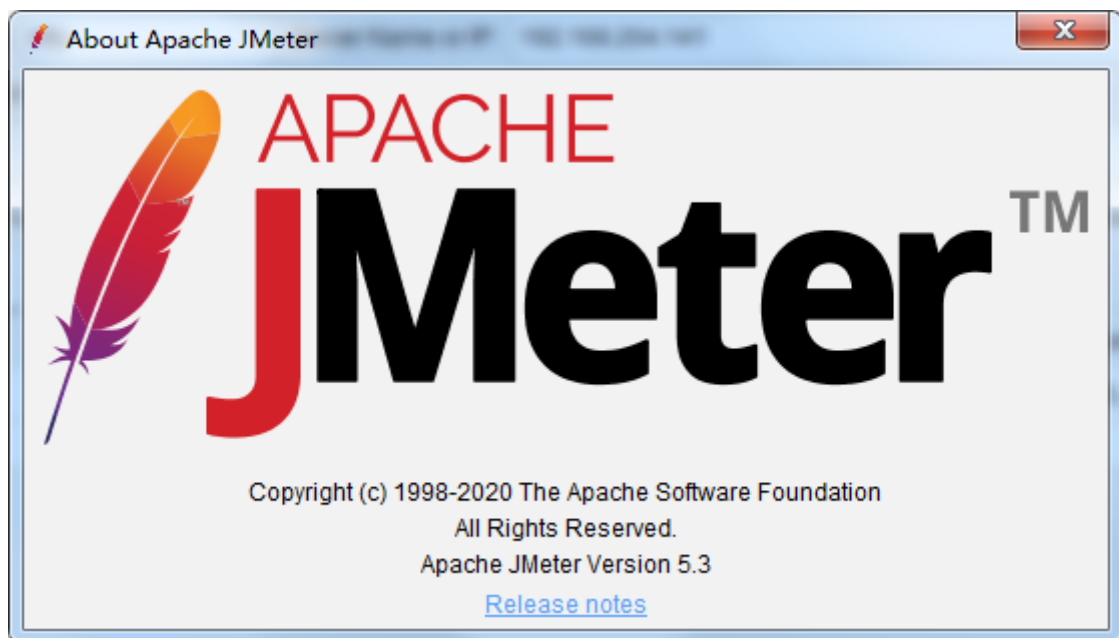
server {
    listen      80;
    server_name localhost;

    #charset koi8-r;

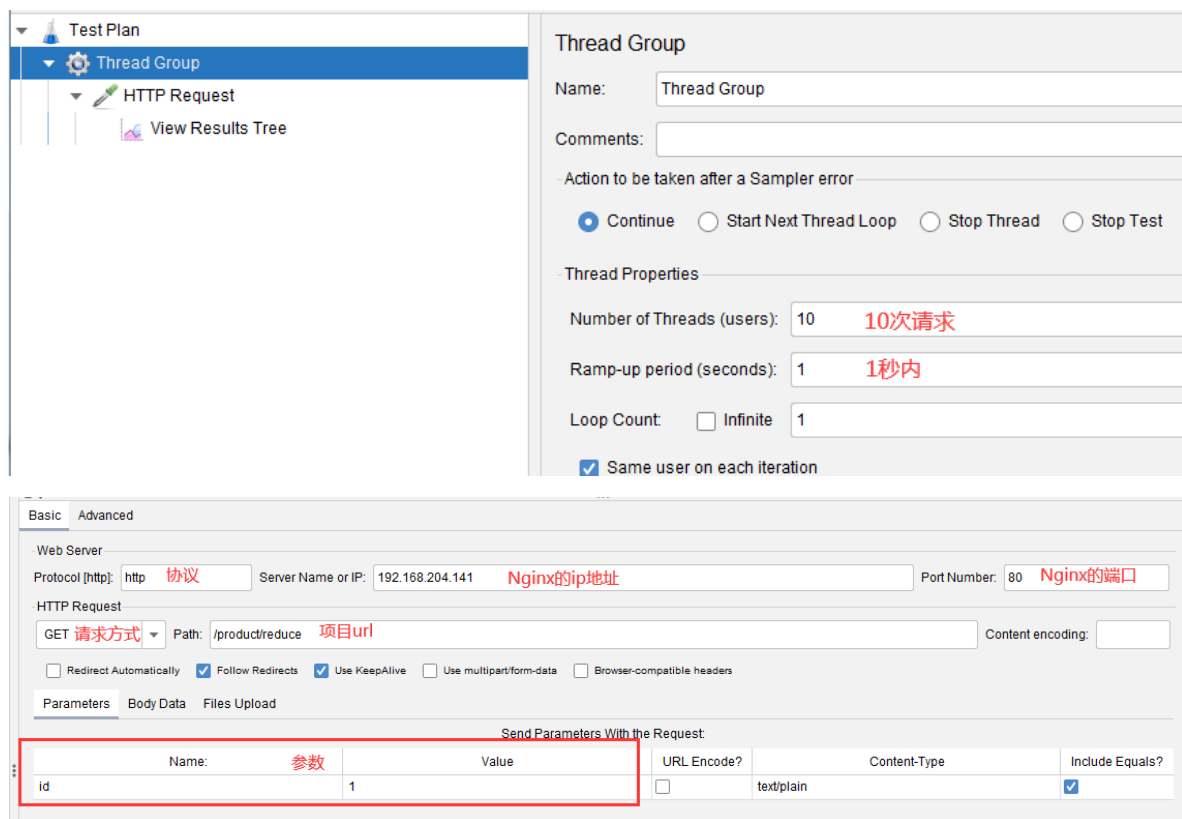
    #access_log logs/host.access.log main;

    location / {
        proxy_pass http://sga;
        root    html;
        index  index.html index.htm;
    }
}
```

3. 使用JMeter 模拟1秒内发出10个http请求



- 下载地址 : http://jmeter.apache.org/download_jmeter.cgi



1. 查看测试结果，10次请求全部成功
2. 查看数据库，stock库存变成 -5 （并发导致的数据结果错误）

4. apache提供的zookeeper客户端

基于zookeeper原生态的客户端类实现分布式是非常麻烦的，我们使用apache提供了一个zookeeper客户端来实现

Curator : <http://curator.apache.org/>

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>4.2.0</version> <!-- 网友投票最牛逼版本 -->
</dependency>
```

recipes是curator族谱大全，里面包含zookeeper和framework

5. 在控制层中加入分布式锁的逻辑代码

```
@Controller
public class ProductAction {

    @Autowired
    private ProductService productService;

    private static String connectString =
        "192.168.204.141:2181,192.168.204.142:2181,192.168.204.143:2181";

    @GetMapping("/product/reduce")
    @ResponseBody
    public Object reduce( int id) throws Exception {
        // 重试策略（1000毫秒试1次，最多试3次）
        RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
        //1. 创建curator工具对象
        CuratorFramework client =
            CuratorFrameworkFactory.newClient(connectString, retryPolicy);
        client.start();
        //2. 根据工具对象创建“内部互斥锁”
        InterProcessMutex lock = new InterProcessMutex(client, "/product_"+id);
        try {
            //3. 加锁
            lock.acquire();
            productService.reduceStock(id);
        } catch (Exception e) {
            if (e instanceof RuntimeException) {
                throw e;
            }
        } finally {
            //4. 释放锁
            lock.release();
        }
        return "ok";
    }
}
```

6. 再次测试，并发问题解决！