# 什么是Zookeeper

Zookeeper是一个分布式开源框架，提供了协调分布式应用的基本服务，它向外部应用暴露一组通用服务——分布式同步（Distributed Synchronization）、命名服务（Naming Service）、集群维护（Group Maintenance）等，简化分布式应用协调及其管理的难度，提供高性能的分布式服务。ZooKeeper本身可以以单机模式安装运行，不过它的长处在于通过分布式ZooKeeper集群（一个Leader，多个Follower），基于一定的策略来保证ZooKeeper集群的稳定性和可用性，从而实现分布式应用的可靠性。

1、Zookeeper是为别的分布式程序服务的

2、Zookeeper本身就是一个分布式程序（只要有半数以上节点存活，zk就能正常服务）

3、Zookeeper所提供的服务涵盖：主从协调、服务器节点动态上下线、统一配置管理、分布式共享锁、统> 一名称服务等

4、虽然说可以提供各种服务，但是zookeeper在底层其实只提供了两个功能：

管理(存储，读取)用户程序提交的数据（类似namenode中存放的metadata）；   
并为用户程序提供数据节点监听服务；

## Zookeeper集群机制

Zookeeper集群的角色： Leader 和 follower   
只要集群中有半数以上节点存活，集群就能提供服务

## Zookeeper特性

1、Zookeeper：一个leader，多个follower组成的集群

2、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

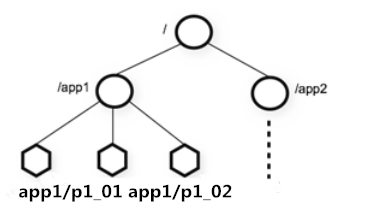
3、分布式读写，更新请求转发，由leader实施

4、更新请求顺序进行，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

## Zookeeper数据结构

1、层次化的目录结构，命名符合常规文件系统规范(类似文件系统）   
   
  
  
2、每个节点在zookeeper中叫做znode,并且其有一个唯一的路径标识 

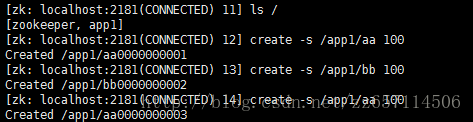
3、节点Znode可以包含数据和子节点（但是EPHEMERAL类型的节点不能有子节点）

节点类型   
a、Znode有两种类型：

短暂（ephemeral）（create -e /app1/test1 “test1” 客户端断开连接zk删除ephemeral类型节点）   
持久（persistent） （create -s /app1/test2 “test2” 客户端断开连接zk不删除persistent类型节点）

b、Znode有四种形式的目录节点（默认是persistent ）

PERSISTENT   
PERSISTENT\_SEQUENTIAL（持久序列/test0000000019 ）   
EPHEMERAL   
EPHEMERAL\_SEQUENTIAL

c、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   
       

d、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

# Zookeeper应用场景

## ****数据发布与订阅（配置中心）****

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到ZK节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，服务式服务框架的服务地址列表等就非常适合使用。

## ****负载均衡****

这里说的负载均衡是指软负载均衡。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，达到对等服务。而消费者就须要在这些对等的服务器中选择一个来执行相关的业务逻辑，其中比较典型的是消息中间件中的生产者，消费者负载均衡。

消息中间件中发布者和订阅者的负载均衡，linkedin开源的KafkaMQ和阿里开源的 metaq都是通过zookeeper来做到生产者、消费者的负载均衡。这里以metaq为例如讲下：

生产者负载均衡：metaq发送消息的时候，生产者在发送消息的时候必须选择一台broker上的一个分区来发送消息，因此metaq在运行过程中，会把所有broker和对应的分区信息全部注册到ZK指定节点上，默认的策略是一个依次轮询的过程，生产者在通过ZK获取分区列表之后，会按照brokerId和partition的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。

消费负载均衡： 在消费过程中，一个消费者会消费一个或多个分区中的消息，但是一个分区只会由一个消费者来消费。MetaQ的消费策略是：

1. 每个分区针对同一个group只挂载一个消费者。

2. 如果同一个group的消费者数目大于分区数目，则多出来的消费者将不参与消费。

3. 如果同一个group的消费者数目小于分区数目，则有部分消费者需要额外承担消费任务。

在某个消费者故障或者重启等情况下，其他消费者会感知到这一变化（通过 zookeeper watch消费者列表），然后重新进行负载均衡，保证所有的分区都有消费者进行消费。

## ****命名服务(Naming Service)****

命名服务也是分布式系统中比较常见的一类场景。在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等——这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用ZK提供的创建节点的API，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。

阿里巴巴集团开源的分布式服务框架Dubbo中使用ZooKeeper来作为其命名服务，维护全局的服务地址列表， 点击这里查看Dubbo开源项目。在Dubbo实现中：

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

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

注意，所有向ZK上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。 另外，Dubbo还有针对服务粒度的监控，方法是订阅/dubbo/${serviceName}目录下所有提供者和消费者的信息。

## ****分布式通知/协调****

ZooKeeper中特有watcher注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容及子节点的），其中一个系统update了znode，那么另一个系统能够收到通知，并作出相应处理

1. 另一种心跳检测机制：检测系统和被检测系统之间并不直接关联起来，而是通过zk上某个节点关联，大大减少系统耦合。

2. 另一种系统调度模式：某系统有控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改了ZK上某些节点的状态，而ZK就把这些变化通知给他们注册Watcher的客户端，即推送系统，于是，作出相应的推送任务。

3. 另一种工作汇报模式：一些类似于任务分发系统，子任务启动后，到zk来注册一个临时节点，并且定时将自己的进度进行汇报（将进度写回这个临时节点），这样任务管理者就能够实时知道任务进度。

总之，使用zookeeper来进行分布式通知和协调能够大大降低系统之间的耦合

## ****集群管理与Master选举****

1. 集群机器监控：这通常用于那种对集群中机器状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。这样的场景中，往往有一个监控系统，实时检测集群机器是否存活。过去的做法通常是：监控系统通过某种手段（比如ping）定时检测每个机器，或者每个机器自己定时向监控系统汇报“我还活着”。 这种做法可行，但是存在两个比较明显的问题：

1. 集群中机器有变动的时候，牵连修改的东西比较多。

2. 有一定的延时。

利用ZooKeeper有两个特性，就可以实现另一种集群机器存活性监控系统：

1. 客户端在节点 x 上注册一个Watcher，那么如果 x?的子节点变化了，会通知该客户端。

2. 创建EPHEMERAL类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。

例如，监控系统在 /clusterServers 节点上注册一个Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。

2. Master选举则是zookeeper中最为经典的应用场景了。

在分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑（例如一些耗时的计算，网络I/O处理），往往只需要让整个集群中的某一台机器进行执行，其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能，于是这个master选举便是这种场景下的碰到的主要问题。

利用ZooKeeper的强一致性，能够保证在分布式高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /currentMaster 节点，最终一定只有一个客户端请求能够创建成功。利用这个特性，就能很轻易的在分布式环境中进行集群选取了。

另外，这种场景演化一下，就是动态Master选举。这就要用到EPHEMERAL\_SEQUENTIAL类型节点的特性了。

上文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终在ZK上创建结果的一种可能情况是这样： /currentMaster/{sessionId}-1 ,/currentMaster/{sessionId}-2,/currentMaster/{sessionId}-3 ….. 每次选取序列号最小的那个机器作为Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是Master了。

1. 在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此之间索引数据一致。因此让集群中的Master来进行全量索引的生成，然后同步到集群中其它机器。另外，Master选举的容灾措施是，可以随时进行手动指定master，就是说应用在zk在无法获取master信息时，可以通过比如http方式，向一个地方获取master。

2. 在Hbase中，也是使用ZooKeeper来实现动态HMaster的选举。在Hbase实现中，会在ZK上存储一些ROOT表的地址和HMaster的地址，HRegionServer也会把自己以临时节点（Ephemeral）的方式注册到Zookeeper中，使得HMaster可以随时感知到各个HRegionServer的存活状态，同时，一旦HMaster出现问题，会重新选举出一个HMaster来运行，从而避免了HMaster的单点问题

## ****分布式锁****

分布式锁，这个主要得益于 ZooKeeper 为我们保证了数据的强一致性。锁服务可以分为两类，一个是 保持独占，另一个是 控制时序。

1. 所谓保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把 zk 上的一个 znode 看作是一把锁，通过 create znode 的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。

2. 控制时序，就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distributelock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制：CreateMode.EPHEMERALSEQUENTIAL 来指定）。Zk 的父节点（/distribute\_lock）维持一份 sequence, 保证子节点创建的时序性，从而也形成了每个客户端的全局时序。

## ****分布式事务****

# Zookeeper安装

## Zookeeper windows环境安装

环境要求:必须要有jdk环境,本次讲课使用jdk1.8

1.安装jdk

2.安装Zookeeper. 在官网[http://zookeeper.apache.org/](http://zookeeper.apache.org/" \t "_blank)下载zookeeper.我下载的是zookeeper-3.4.6版本。

解压zookeeper-3.4.6至D:\machine\zookeeper-3.4.6.

在D:\machine 新建data及log目录。

3.ZooKeeper的安装模式分为三种，分别为：单机模式（stand-alone）、集群模式和集群伪分布模式。ZooKeeper 单机模式的安装相对比较简单，如果第一次接触ZooKeeper的话，建议安装ZooKeeper单机模式或者集群伪分布模式。

安装单击模式。 至D:\machine\zookeeper-3.4.6\conf 复制 zoo\_sample.cfg 并粘贴到当前目录下，命名zoo.cfg.

## Zookeeper集群环境搭建(linux)

环境要求:必须要有jdk环境,本次讲课使用jdk1.8

### 结构

一共三个节点  
(zk服务器集群规模不小于3个节点),要求服务器之间系统时间保持一致。

## **上传zk并且解压** 进行解压： tar -zxvf zookeeper-3.4.6.tar.gz 重命名： mv zookeeper-3.4.6 zookeeper

### 修改zookeeper环境变量

vi /etc/profile

export JAVA\_HOME=/opt/jdk1.8.0\_71

export ZOOKEEPER\_HOME=/usr/local/zookeeper

export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

export PATH=$JAVA\_HOME/bin:$ZOOKEEPER\_HOME/bin:$PATH

source /etc/profile

## **修改zoo\_sample.cfg文件** cd /usr/local/zookeeper/conf mv zoo\_sample.cfg zoo.cfg 修改conf: vi zoo.cfg 修改两处 （1） dataDir=/usr/local/zookeeper/data（注意同时在zookeeper创建data目录） （2）最后面添加 server.0=bhz:2888:3888 server.1=hadoop1:2888:3888 server.2=hadoop2:2888:3888

## **创建服务器标识** 服务器标识配置： 创建文件夹： mkdir data 创建文件myid并填写内容为0： vi myid (内容为服务器标识 ： 0)

## 复制zookeeper

进行复制zookeeper目录到hadoop01和hadoop02  
还有/etc/profile文件  
把hadoop01、 hadoop02中的myid文件里的值修改为1和2  
路径(vi /usr/local/zookeeper/data/myid)

## 启动zookeeper 启动zookeeper： 路径： /usr/local/zookeeper/bin 执行： zkServer.sh start (注意这里3台机器都要进行启动) 状态： zkServer.sh status(在三个节点上检验zk的mode,一个leader和俩个follower)

## 常用命令

zkServer.sh status 查询状态

# Zookeeper配置文件介绍

|  |
| --- |
| # The number of milliseconds of each tick  tickTime=2000    # The number of ticks that the initial  # synchronization phase can take  initLimit=10    # The number of ticks that can pass between  # sending a request and getting an acknowledgement  syncLimit=5    # the directory where the snapshot is stored.  # do not use /tmp for storage, /tmp here is just  # example sakes.  dataDir=/home/myuser/zooA/data    # the port at which the clients will connect  clientPort=2181    # ZooKeeper server and its port no. # ZooKeeper ensemble should know about every other machine in the ensemble # specify server id by creating 'myid' file in the dataDir # use hostname instead of IP address for convenient maintenance  server.1=127.0.0.1:2888:3888  server.2=127.0.0.1:2988:3988  server.3=127.0.0.1:2088:3088    #  # Be sure to read the maintenance section of the  # administrator guide before turning on autopurge.  #  # http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance  #  # The number of snapshots to retain in dataDir  # autopurge.snapRetainCount=3  # Purge task interval in hours  # Set to "0" to disable auto purge feature  <br>  #autopurge.purgeInterval=1  dataLogDir=/home/myuser/zooA/log |

tickTime：心跳时间，为了确保连接存在的，以毫秒为单位，最小超时时间为两个心跳时间

initLimit：多少个心跳时间内，允许其他server连接并初始化数据，如果ZooKeeper管理的数据较大，则应相应增大这个值

clientPort：服务的监听端口

dataDir：用于存放内存数据库快照的文件夹，同时用于集群的myid文件也存在这个文件夹里（注意：一个配置文件只能包含一个dataDir字样，即使它被注释掉了。）

dataLogDir：用于单独设置transaction log的目录，transaction log分离可以避免和普通log还有快照的竞争

syncLimit：多少个tickTime内，允许follower同步，如果follower落后太多，则会被丢弃。

server.A=B：C：D：  
A是一个数字,表示这个是第几号服务器,B是这个服务器的ip地址  
C第一个端口用来集群成员的信息交换,表示的是这个服务器与集群中的Leader服务器交换信息的端口  
D是在leader挂掉时专门用来进行选举leader所用

# Java操作Zookeeper

## Zookeeper说明

创建节点(znode) 方法:

## create: 提供了两套创建节点的方法，同步和异步创建节点方式。 同步方式: 参数1，节点路径《名称) : InodeName (不允许递归创建节点，也就是说在父节点不存在 的情况下，不允许创建子节点) 参数2，节点内容: 要求类型是字节数组(也就是说，不支持序列化方式，如果需要实现序 列化，可使用java相关序列化框架，如Hessian、Kryo框架) 参數3，节点权限: 使用Ids.OPEN\_ACL\_UNSAFE开放权限即可。(这个参数一般在权展 没有太高要求的场景下，没必要关注) 参数4,节点类型: 创建节点的类型: CreateMode，提供四种首点象型

|  |
| --- |
| PERSISTENT 持久化节点  PERSISTENT\_SEQUENTIAL 顺序自动编号持久化节点，这种节点会根据当前已存在的节点数自动加 1  EPHEMERAL 临时节点， 客户端session超时这类节点就会被自动删除  EPHEMERAL\_SEQUENTIAL 临时自动编号节点 |

## Maven依赖信息

|  |
| --- |
| <dependency>  <groupId>org.apache.zookeeper</groupId>  <artifactId>zookeeper</artifactId>  <version>3.4.6</version>  </dependency> |

## Zookeeper客户端连接

|  |
| --- |
| **public** **class** Test001 {  //连接地址  **private** **static** **final** String ***ADDRES*** = "127.0.0.1:2181";  //session 会话  **private** **static** **final** **int** ***SESSION\_OUTTIME*** = 2000;  //信号量,阻塞程序执行,用户等待zookeeper连接成功,发送成功信号，  **private** **static** **final** CountDownLatch ***countDownLatch*** = **new** CountDownLatch(1);  **public** **static** **void** main(String[] args) **throws** IOException, InterruptedException, KeeperException {  ZooKeeper zk = **new** ZooKeeper(***ADDRES***, ***SESSION\_OUTTIME***, **new** Watcher() {  **public** **void** process(WatchedEvent event) {  // 获取事件状态  KeeperState keeperState = event.getState();  // 获取事件类型  EventType eventType = event.getType();  **if** (KeeperState.***SyncConnected*** == keeperState) {  **if** (EventType.***None*** == eventType) {  ***countDownLatch***.countDown();  System.***out***.println("zk 启动连接...");  }  }  }  });  // 进行阻塞  ***countDownLatch***.await();  String result = zk.create("/itmayeidu\_Lasting", "Lasting".getBytes(), Ids.***OPEN\_ACL\_UNSAFE***,  CreateMode.***PERSISTENT***);  System.***out***.println(result);  zk.close();  }  } |

## 创建Zookeeper节点信息

|  |
| --- |
| 1. **创建持久节点，并且允许任何服务器可以操作**   String result = zk.create("/itmayiedu\_Lasting", "Lasting".getBytes(), Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  System.***out***.println("result:" + result);   1. **创建临时节点**   String result = zk.create("/itmayiedu\_temp", "temp".getBytes(), Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL***);  System.***out***.println("result:" + result); |

## Watcher

在ZooKeeper中，接口类Watcher用于表示一个标准的事件处理器，其定义了事件通知相关的逻辑，包含KeeperState和EventType两个枚举类，分别代表了通知状态和事件类型，同时定义了事件的回调方法：process（WatchedEvent event）。

## 什么是Watcher接口

同一个事件类型在不同的通知状态中代表的含义有所不同，表7-3列举了常见的通知状态和事件类型。

表7-3 Watcher通知状态与事件类型一览

|  |  |  |  |
| --- | --- | --- | --- |
| KeeperState | EventType | 触发条件 | 说明 |
|  | None （-1） | 客户端与服务端成功建立连接 |  |
| SyncConnected （0） | NodeCreated （1） | Watcher监听的对应数据节点被创建 |  |
|  | NodeDeleted （2） | Watcher监听的对应数据节点被删除 | 此时客户端和服务器处于连接状态 |
|  | NodeDataChanged （3） | Watcher监听的对应数据节点的数据内容发生变更 |  |
|  | NodeChildChanged （4） | Wather监听的对应数据节点的子节点列表发生变更 |  |
| Disconnected （0） | None （-1） | 客户端与ZooKeeper服务器断开连接 | 此时客户端和服务器处于断开连接状态 |
| Expired （-112） | Node （-1） | 会话超时 | 此时客户端会话失效，通常同时也会受到SessionExpiredException异常 |
| AuthFailed （4） | None （-1） | 通常有两种情况，1：使用错误的schema进行权限检查 2：SASL权限检查失败 | 通常同时也会收到AuthFailedException异常 |

表7-3中列举了ZooKeeper中最常见的几个通知状态和事件类型。

回调方法process（）

process方法是Watcher接口中的一个回调方法，当ZooKeeper向客户端发送一个Watcher事件通知时，客户端就会对相应的process方法进行回调，从而实现对事件的处理。process方法的定义如下：

abstract public void process(WatchedEvent event);

这个回调方法的定义非常简单，我们重点看下方法的参数定义：WatchedEvent。

WatchedEvent包含了每一个事件的三个基本属性：通知状态（keeperState），事件类型（EventType）和节点路径（path），其数据结构如图7-5所示。ZooKeeper使用WatchedEvent对象来封装服务端事件并传递给Watcher，从而方便回调方法process对服务端事件进行处理。

提到WatchedEvent，不得不讲下WatcherEvent实体。笼统地讲，两者表示的是同一个事物，都是对一个服务端事件的封装。不同的是，WatchedEvent是一个逻辑事件，用于服务端和客户端程序执行过程中所需的逻辑对象，而WatcherEvent因为实现了序列化接口，因此可以用于网络传输。

服务端在生成WatchedEvent事件之后，会调用getWrapper方法将自己包装成一个可序列化的WatcherEvent事件，以便通过网络传输到客户端。客户端在接收到服务端的这个事件对象后，首先会将WatcherEvent还原成一个WatchedEvent事件，并传递给process方法处理，回调方法process根据入参就能够解析出完整的服务端事件了。

需要注意的一点是，无论是WatchedEvent还是WatcherEvent，其对ZooKeeper服务端事件的封装都是机及其简单的。举个例子来说，当/zk-book这个节点的数据发生变更时，服务端会发送给客户端一个“ZNode数据内容变更”事件，客户端只能够接收到如下信

## Watcher代码

|  |
| --- |
| **public class ZkClientWatcher implements Watcher {**  **// 集群连接地址**  **private static final String *CONNECT\_ADDRES* = "192.168.110.159:2181,192.168.110.160:2181,192.168.110.162:2181";**  **// 会话超时时间**  **private static final int *SESSIONTIME* = 2000;**  **// 信号量,让zk在连接之前等待,连接成功后才能往下走.**  **private static final CountDownLatch *countDownLatch* = new CountDownLatch(1);**  **private static String *LOG\_MAIN* = "【main】 ";**  **private ZooKeeper zk;**  **public void createConnection(String connectAddres, int sessionTimeOut) {**  **try {**  **zk = new ZooKeeper(connectAddres, sessionTimeOut, this);**  **System.*out*.println(*LOG\_MAIN* + "zk 开始启动连接服务器....");**  ***countDownLatch*.await();**  **} catch (Exception e) {**  **e.printStackTrace();**  **}**  **}**  **public boolean createPath(String path, String data) {**  **try {**  **this.exists(path, true);**  **this.zk.create(path, data.getBytes(), Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);**  **System.*out*.println(*LOG\_MAIN* + "节点创建成功, Path:" + path + ",data:" + data);**  **} catch (Exception e) {**  **e.printStackTrace();**  **return false;**  **}**  **return true;**  **}**  **/\*\***  **\* 判断指定节点是否存在**  **\***  **\* @param path**  **\* 节点路径**  **\*/**  **public Stat exists(String path, boolean needWatch) {**  **try {**  **return this.zk.exists(path, needWatch);**  **} catch (Exception e) {**  **e.printStackTrace();**  **return null;**  **}**  **}**  **public boolean updateNode(String path,String data) throws KeeperException, InterruptedException {**  **exists(path, true);**  **this.zk.setData(path, data.getBytes(), -1);**  **return false;**  **}**  **public void process(WatchedEvent watchedEvent) {**  **// 获取事件状态**  **KeeperState keeperState = watchedEvent.getState();**  **// 获取事件类型**  **EventType eventType = watchedEvent.getType();**  **// zk 路径**  **String path = watchedEvent.getPath();**  **System.*out*.println("进入到 process() keeperState:" + keeperState + ", eventType:" + eventType + ", path:" + path);**  **// 判断是否建立连接**  **if (KeeperState.*SyncConnected* == keeperState) {**  **if (EventType.*None* == eventType) {**  **// 如果建立建立成功,让后程序往下走**  **System.*out*.println(*LOG\_MAIN* + "zk 建立连接成功!");**  ***countDownLatch*.countDown();**  **} else if (EventType.*NodeCreated* == eventType) {**  **System.*out*.println(*LOG\_MAIN* + "事件通知,新增node节点" + path);**  **} else if (EventType.*NodeDataChanged* == eventType) {**  **System.*out*.println(*LOG\_MAIN* + "事件通知,当前node节点" + path + "被修改....");**  **}**  **else if (EventType.*NodeDeleted* == eventType) {**  **System.*out*.println(*LOG\_MAIN* + "事件通知,当前node节点" + path + "被删除....");**  **}**  **}**  **System.*out*.println("--------------------------------------------------------");**  **}**  **public static void main(String[] args) throws KeeperException, InterruptedException {**  **ZkClientWatcher zkClientWatcher = new ZkClientWatcher();**  **zkClientWatcher.createConnection(*CONNECT\_ADDRES*, *SESSIONTIME*);**  **// boolean createResult = zkClientWatcher.createPath("/p15", "pa-644064");**  **zkClientWatcher.updateNode("/pa2","7894561");**  **}**  **}** |

# 使用Zookeeper实现负载均衡原理

## 思路

使用Zookeeper实现负载均衡原理，服务器端将启动的服务注册到，zk注册中心上，采用临时节点。客户端从zk节点上获取最新服务节点信息，本地使用负载均衡算法，随机分配服务器。

## 创建项目工程

### Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.8</version>  </dependency>  </dependencies> |

### 创建Server服务端

### ZkServerScoekt服务

|  |
| --- |
| //##ServerScoekt服务端  **public** **class** ZkServerScoekt **implements** Runnable {  **private** **int** port = 18080;  **public** **static** **void** main(String[] args) **throws** IOException {  **int** port = 18080;  ZkServerScoekt server = **new** ZkServerScoekt(port);  Thread thread = **new** Thread(server);  thread.start();  }  **public** ZkServerScoekt(**int** port) {  **this**.port = port;  }  **public** **void** run() {  ServerSocket serverSocket = **null**;  **try** {  serverSocket = **new** ServerSocket(port);  System.***out***.println("Server start port:" + port);  Socket socket = **null**;  **while** (**true**) {  socket = serverSocket.accept();  **new** Thread(**new** ServerHandler(socket)).start();  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if** (serverSocket != **null**) {  serverSocket.close();  }  } **catch** (Exception e2) {  }  }  }  } |

### ServerHandler

|  |
| --- |
| **public** **class** ServerHandler **implements** Runnable {  **private** Socket socket;  **public** ServerHandler(Socket socket) {  **this**.socket = socket;  }  **public** **void** run() {  BufferedReader in = **null**;  PrintWriter out = **null**;  **try** {  in = **new** BufferedReader(**new** InputStreamReader(**this**.socket.getInputStream()));  out = **new** PrintWriter(**this**.socket.getOutputStream(), **true**);  String body = **null**;  **while** (**true**) {  body = in.readLine();  **if** (body == **null**)  **break**;  System.***out***.println("Receive : " + body);  out.println("Hello, " + body);  }  } **catch** (Exception e) {  **if** (in != **null**) {  **try** {  in.close();  } **catch** (IOException e1) {  e1.printStackTrace();  }  }  **if** (out != **null**) {  out.close();  }  **if** (**this**.socket != **null**) {  **try** {  **this**.socket.close();  } **catch** (IOException e1) {  e1.printStackTrace();  }  **this**.socket = **null**;  }  }  }  } |

### ZkServerClient

|  |
| --- |
| **public** **class** ZkServerClient {  **public** **static** List<String> *listServer* = **new** ArrayList<String>();  **public** **static** **void** main(String[] args) {  *initServer*();  ZkServerClient client= **new** ZkServerClient();  BufferedReader console = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **while** (**true**) {  String name;  **try** {  name = console.readLine();  **if** ("exit".equals(name)) {  System.*exit*(0);  }  client.send(name);  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  // 注册所有server  **public** **static** **void** initServer() {  *listServer*.clear();  *listServer*.add("127.0.0.1:18080");  }  // 获取当前server信息  **public** **static** String getServer() {  **return** *listServer*.get(0);  }    **public** **void** send(String name) {  String server = ZkServerClient.*getServer*();  String[] cfg = server.split(":");  Socket socket = **null**;  BufferedReader in = **null**;  PrintWriter out = **null**;  **try** {  socket = **new** Socket(cfg[0], Integer.*parseInt*(cfg[1]));  in = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));  out = **new** PrintWriter(socket.getOutputStream(), **true**);  out.println(name);  **while** (**true**) {  String resp = in.readLine();  **if** (resp == **null**)  **break**;  **else** **if** (resp.length() > 0) {  System.***out***.println("Receive : " + resp);  **break**;  }  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (out != **null**) {  out.close();  }  **if** (in != **null**) {  **try** {  in.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  } |

### 改造ZkServerScoekt

|  |
| --- |
| **public** **class** ZkServerScoekt **implements** Runnable {  **private** **static** **int** *port* = 18081;  **public** **static** **void** main(String[] args) **throws** IOException {  ZkServerScoekt server = **new** ZkServerScoekt(*port*);  Thread thread = **new** Thread(server);  thread.start();  }  **public** ZkServerScoekt(**int** port) {  **this**.*port* = port;  }  **public** **void** regServer() {  // 向ZooKeeper注册当前服务器  ZkClient client = **new** ZkClient("127.0.0.1:2181", 60000, 1000);  String path = "/test/server" + *port*;  **if** (client.exists(path))  client.delete(path);  client.createEphemeral(path, "127.0.0.1:" + *port*);  }  **public** **void** run() {  ServerSocket serverSocket = **null**;  **try** {  serverSocket = **new** ServerSocket(*port*);  regServer();  System.***out***.println("Server start port:" + *port*);  Socket socket = **null**;  **while** (**true**) {  socket = serverSocket.accept();  **new** Thread(**new** ServerHandler(socket)).start();  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if** (serverSocket != **null**) {  serverSocket.close();  }  } **catch** (Exception e2) {  }  }  }  } |

### 改造ZkServerClient

|  |
| --- |
| **public** **class** ZkServerClient {  **public** **static** List<String> *listServer* = **new** ArrayList<String>();  **public** **static** String *parent* = "/test";  **public** **static** **void** main(String[] args) {  *initServer*();  ZkServerClient client = **new** ZkServerClient();  BufferedReader console = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **while** (**true**) {  String name;  **try** {  name = console.readLine();  **if** ("exit".equals(name)) {  System.*exit*(0);  }  client.send(name);  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  // 注册所有server  **public** **static** **void** initServer() {  // listServer.add("127.0.0.1:18080");  **final** ZkClient zkClient = **new** ZkClient("127.0.0.1:2181", 6000, 1000);  List<String> children = zkClient.getChildren(*parent*);  *getChilds*(zkClient, children);  // 监听事件  zkClient.subscribeChildChanges(*parent*, **new** IZkChildListener() {  **public** **void** handleChildChange(String parentPath, List<String> currentChilds) **throws** Exception {  *getChilds*(zkClient, currentChilds);  }  });  }  **private** **static** **void** getChilds(ZkClient zkClient, List<String> currentChilds) {  *listServer*.clear();  **for** (String p : currentChilds) {  String pathValue = (String) zkClient.readData(*parent* + "/" + p);  *listServer*.add(pathValue);  }  *serverCount* = *listServer*.size();  System.***out***.println("从zk读取到信息:" + *listServer*.toString());  }  // 请求次数  **private** **static** **int** *reqestCount* = 1;  // 服务数量  **private** **static** **int** *serverCount* = 0;  // 获取当前server信息  **public** **static** String getServer() {  // 实现负载均衡  String serverName = *listServer*.get(*reqestCount* % *serverCount*);  ++*reqestCount*;  **return** serverName;  }  **public** **void** send(String name) {  String server = ZkServerClient.*getServer*();  String[] cfg = server.split(":");  Socket socket = **null**;  BufferedReader in = **null**;  PrintWriter out = **null**;  **try** {  socket = **new** Socket(cfg[0], Integer.*parseInt*(cfg[1]));  in = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));  out = **new** PrintWriter(socket.getOutputStream(), **true**);  out.println(name);  **while** (**true**) {  String resp = in.readLine();  **if** (resp == **null**)  **break**;  **else** **if** (resp.length() > 0) {  System.***out***.println("Receive : " + resp);  **break**;  }  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (out != **null**) {  out.close();  }  **if** (in != **null**) {  **try** {  in.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  } |

# 使用Zookeepers实现分布式锁

# 什么是多线程

多线程为了能够提高应用程序的运行效率，在一个进程中有多条不同的执行路径，同时并行执行，互不影响。

# 什么是线程安全

当多个线程同时共享，同一个**全局变量或静态变量**，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作是不会发生数据冲突问题。

## 解决办法

使用同步代码块或者Lock锁机制，保证在多个线程共享同一个变量只能有一个线程进行操作

# 什么是Java内存模型

共享内存模型指的就是Java内存模型(简称JMM)，**JMM决定一个线程对共享变量的写入时,能对另一个线程可见**。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：**线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本**。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

下面通过示意图来说明这两个步骤：   


如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

**总结：什么是Java内存模型：java内存模型简称jmm，定义了一个线程对另一个线程可见。共享变量存放在主内存中，每个线程都有自己的本地内存，当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题。**

# 分布式锁解决办法

# 传统方式生成订单号ID

## 业务场景

在分布式情况，生成全局订单号ID

## 生成订单号方案

1. 使用时间戳
2. 使用UUID
3. 推特 (Twitter) 的 Snowflake 算法——用于生成唯一 ID

## 生成订单类

|  |
| --- |
| //生成订单类  **public** **class** OrderNumGenerator {  //全局订单id  **public** **static** **int** *count* = 0;  **public** String getNumber() {  **try** {  Thread.*sleep*(200);  } **catch** (Exception e) {  }  SimpleDateFormat simpt = **new** SimpleDateFormat("yyyy-MM-dd-HH-mm-ss");  **return** simpt.format(**new** Date()) + "-" + ++*count*;  }  } |

## 使用多线程情况模拟生成订单号

|  |
| --- |
| //使用多线程模拟生成订单号  **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  **for** (**int** i = 0; i < 100; i++) {  **new** Thread(**new** OrderService()).start();  }  }  } |

## 多线程生成订单号，线程安全问题解决

使用synchronized或者loca锁

### Synchronized同步代码块方式

|  |
| --- |
| //使用多线程模拟生成订单号  **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  **synchronized** (**this**) {  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  }  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  OrderService orderService = **new** OrderService();  **for** (**int** i = 0; i < 100; i++) {  **new** Thread(orderService).start();  }  }  } |

### Lock锁方式

|  |
| --- |
| **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  // 使用lock锁  **private** java.util.concurrent.locks.Lock lock = **new** ReentrantLock();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  **try** {  // synchronized (this) {  lock.lock();  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  // }  } **catch** (Exception e) {  } **finally** {  lock.unlock();  }  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  OrderService orderService = **new** OrderService();  **for** (**int** i = 0; i < 100; i++) {  **new** Thread(orderService).start();  }  }  } |

# 分布式场景下生成订单ID

## 业务场景

在分布式情况，生成全局订单号ID

## 产生问题

在分布式(集群)环境下，每台JVM不能实现同步，在分布式场景下使用时间戳生成订单号可能会重复

## 分布式情况下，怎么解决订单号生成不重复

1. 使用分布式锁
2. 提前生成好，订单号，存放在redis取。获取订单号，直接从redis中取。

## 使用分布式锁生成订单号技术

1.使用数据库实现分布式锁

缺点:性能差、线程出现异常时，容易出现死锁

2.使用redis实现分布式锁

缺点:锁的失效时间难控制、容易产生死锁、非阻塞式、不可重入

**3.使用zookeeper实现分布式锁**

**实现相对简单、可靠性强、使用临时节点，失效时间容易控制**

## 什么是分布式锁

分布式锁一般用在分布式系统或者多个应用中，用来控制同一任务是否执行或者任务的执行顺序。在项目中，部署了多个tomcat应用，在执行定时任务时就会遇到同一任务可能执行多次的情况，我们可以借助分布式锁，保证在同一时间只有一个tomcat应用执行了定时任务

## 使用Zookeeper实现分布式锁

### Zookeeper实现分布式锁原理

使用zookeeper创建临时序列节点来实现分布式锁，适用于顺序执行的程序，大体思路就是创建临时序列节点，找出最小的序列节点，获取分布式锁，程序执行完成之后此序列节点消失，通过watch来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理，依次类推……

### Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.10</version>  </dependency>  </dependencies> |

### 创建Lock接口

|  |
| --- |
| **public** **interface** Lock {  //获取到锁的资源  **public** **void** getLock();  // 释放锁  **public** **void** unLock();  } |

### 创建ZookeeperAbstractLock抽象类

|  |
| --- |
| **//将重复代码写入子类中..**  **public abstract class ZookeeperAbstractLock implements Lock {**  **// zk连接地址**  **private static final String *CONNECTSTRING* = "127.0.0.1:2181";**  **// 创建zk连接**  **protected ZkClient zkClient = new ZkClient(*CONNECTSTRING*);**  **protected static final String *PATH* = "/lock";**  **public void getLock() {**  **if (tryLock()) {**  **System.*out*.println("##获取lock锁的资源####");**  **} else {**  **// 等待**  **waitLock();**  **// 重新获取锁资源**  **getLock();**  **}**  **}**  **// 获取锁资源**  **abstract boolean tryLock();**  **// 等待**  **abstract void waitLock();**  **public void unLock() {**  **if (zkClient != null) {**  **zkClient.close();**  **System.*out*.println("释放锁资源...");**  **}**  **}**  **}** |

### ZookeeperDistrbuteLock类

|  |
| --- |
| public class ZookeeperDistrbuteLock extends ZookeeperAbstractLock {  private CountDownLatch countDownLatch = null;  @Override  boolean tryLock() {  try {  zkClient.createEphemeral(*PATH*);  return true;  } catch (Exception e) {  // e.printStackTrace();  return false;  }  }  @Override  void waitLock() {  IZkDataListener izkDataListener = new IZkDataListener() {  public void handleDataDeleted(String path) throws Exception {  // 唤醒被等待的线程  if (countDownLatch != null) {  countDownLatch.countDown();  }  }  public void handleDataChange(String path, Object data) throws Exception {  }  };  // 注册事件  zkClient.subscribeDataChanges(*PATH*, izkDataListener);  if (zkClient.exists(*PATH*)) {  countDownLatch = new CountDownLatch(1);  try {  countDownLatch.await();  } catch (Exception e) {  e.printStackTrace();  }  }  // 删除监听  zkClient.unsubscribeDataChanges(*PATH*, izkDataListener);  }  } |

### 使用Zookeeper锁运行效果

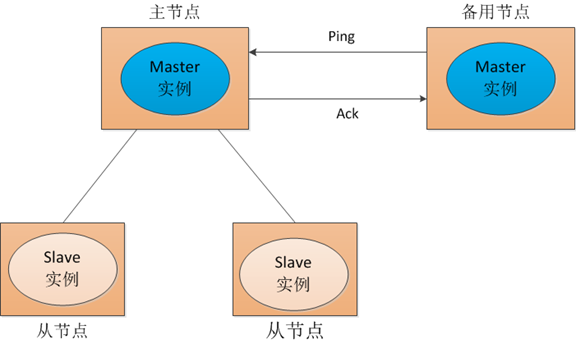
|  |
| --- |
| **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  // 使用lock锁  // private java.util.concurrent.locks.Lock lock = new ReentrantLock();  **private** Lock lock = **new** ZookeeperDistrbuteLock();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  **try** {  lock.getLock();  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  lock.unLock();  }  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  // OrderService orderService = new OrderService();  **for** (**int** i = 0; i < 100; i++) {  **new** Thread( **new** OrderService()).start();  }  }  } |

## Zookeeper实现哨兵机制

### master选举使用场景及结构

　现在很多时候我们的服务需要7\*24小时工作，假如一台机器挂了，我们希望能有其它机器顶替它继续工作。此类问题现在多采用master-salve模式，也就是常说的主从模式，正常情况下主机提供服务，备机负责监听主机状态，当主机异常时，可以自动切换到备机继续提供服务(这里有点儿类似于数据库主库跟备库，备机正常情况下只监听，不工作)，这个切换过程中选出下一个主机的过程就是master选举。

对于以上提到的场景，传统的解决方式是采用一个备用节点，这个备用节点定期给当前主节点发送ping包，主节点收到ping包后会向备用节点发送应答ack，当备用节点收到应答，就认为主节点还活着，让它继续提供服务，否则就认为主节点挂掉了，自己将开始行使主节点职责。如图1所示：



### Maven依赖信息

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.0.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.10</version>  <exclusions>  <exclusion>  <artifactId>slf4j-api</artifactId>  <groupId>org.slf4j</groupId>  </exclusion>  <exclusion>  <artifactId>log4j</artifactId>  <groupId>log4j</groupId>  </exclusion>  <exclusion>  <artifactId>slf4j-log4j12</artifactId>  <groupId>org.slf4j</groupId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

### IndexController

|  |
| --- |
| @RestController  **public** **class** IndexController {  // 获取服务信息  @RequestMapping("/getServerInfo")  **public** String getServerInfo() {  **return** ElectionMaster.*isSurvival* ? "当前服务器为主节点" : "当前服务器为从节点";  }  } |

### MyApplicationRunner

|  |
| --- |
| @Component  **public** **class** MyApplicationRunner **implements** ApplicationRunner {  // 创建zk连接  ZkClient zkClient = **new** ZkClient("127.0.0.1:2181");  **private** String path = "/election";  @Value("${server.port}")  **private** String serverPort;  **public** **void** run(ApplicationArguments args) **throws** Exception {  System.***out***.println("项目启动完成...");  createEphemeral();  // 创建事件监听  zkClient.subscribeDataChanges(path, **new** IZkDataListener() {  // 节点被删除  **public** **void** handleDataDeleted(String dataPath) **throws** Exception {  // 主节点已经挂了，重新选举  System.***out***.println("主节点已经挂了，重新开始选举");  createEphemeral();  }  **public** **void** handleDataChange(String dataPath, Object data) **throws** Exception {  }  });  }  **private** **void** createEphemeral() {  **try** {  zkClient.createEphemeral(path, serverPort);  ElectionMaster.*isSurvival* = **true**;  System.***out***.println("serverPort：" + serverPort + ",选举成功....");  } **catch** (Exception e) {  ElectionMaster.*isSurvival* = **false**;  }  }  } |

### **ElectionMaster**

|  |
| --- |
| @Component  **public** **class** ElectionMaster {  // 服务器info信息 是否存活  **public** **static** **boolean** *isSurvival*;  } |