# Redis集群实现

## Redis集群

### 为什么要搭建集群

通常，为了提高网站响应速度，总是把热点数据保存在内存中而不是直接从后端数据库中读取。

Redis是一个很好的Cache工具。大型网站应用，热点数据量往往巨大，几十G上百G是很正常的事儿。

由于内存大小的限制，使用一台 Redis 实例显然无法满足需求，这时就需要使用多台 Redis作为缓存数据库。但是如何保证数据存储的一致性呢,这时就需要搭建redis集群.采用合理的机制,保证用户的正常的访问需求.

采用redis集群,可以保证数据分散存储,同时保证数据存储的一致性.并且在内部实现高可用的机制.实现了服务故障的自动迁移.

## Redis集群搭建步骤

### 集群搭建规模

规划:3台主机6台从机,每台主机跟随2台从机.

端口号:7000-7008端口

### 搭建准备:

在redis根目录中创建cluster文件夹后新增7000-7008文化夹

如图-1所示



图- 1

### 搭建步骤

1.复制redis.conf文件到7000下

cp redis.conf cluster/7000

2.关闭aof使用rdb模式 593行

如图-2所示

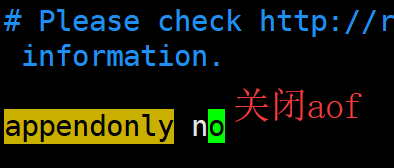


图- 2

3.取消IP绑定

如图-3所示



图- 3

4.关闭保护模式

如图-4所示

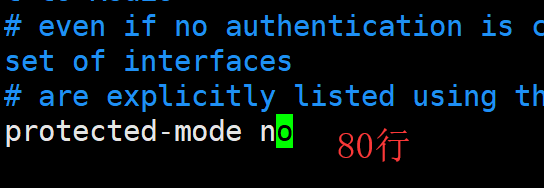


图- 4

5.修改端口号

如图-5所示

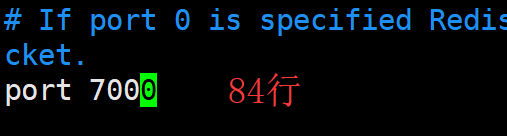


图- 5

6.开启后台启动

如图-6所示

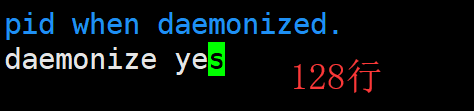


图- 6

7.修改PID位置 **150行**

如图-7所示

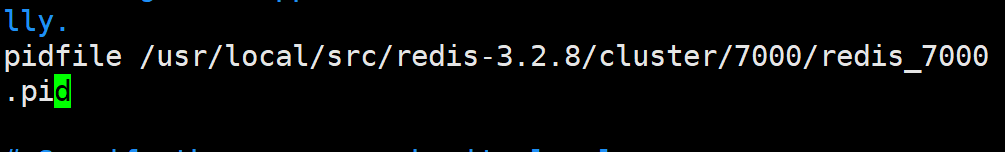


图- 7

8.修改持久化文件路径

如图-8所示

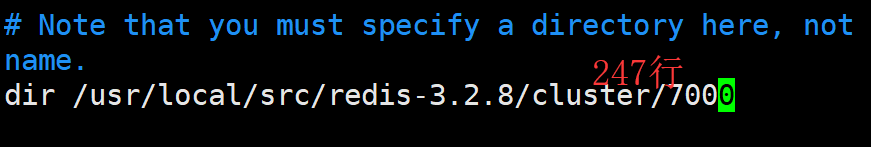


图- 8

9.修改内存策略

如图-9所示

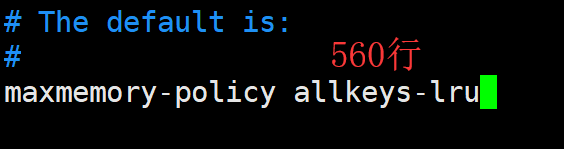


图- 9

10.开启集群

如图-10所示

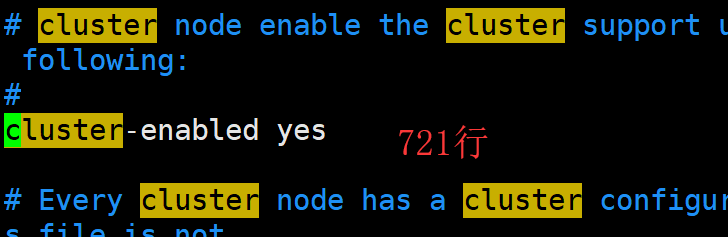


图- 10

11.开启集群配置

如图-11所示

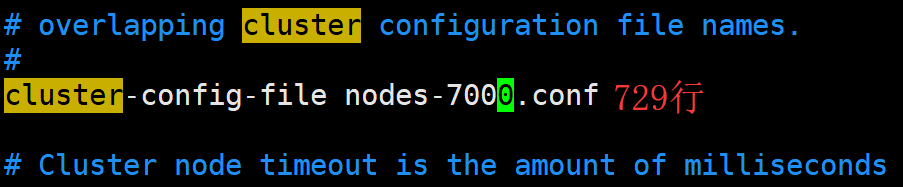


图- 11

12.修改集群推选时间

如图-12所示

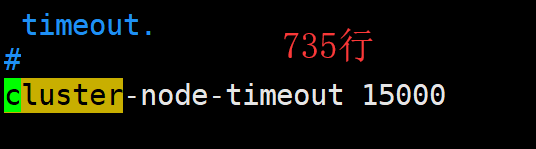


图- 12

### 拷贝配置文件

说明:将redis中的7000下的redis.conf文件导入7001-7008中

cp 7000/redis.conf 7003/

### 批量修改配置文件

说明:搭建redis节点之前,需要启动全部9台redis节点,但是配置文件都是由第一个配置文件复制来的.所以启动redis节点,则必须修改端口号.但是一个一个修改需要修改8次.所以需要批量修改的命令

在vim命令下执行

命令: :%s/7000/7001/g

如图-13所示

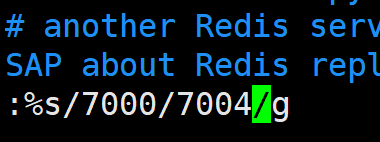


图- 13

修改完成后,分别修改7001-7008端口号

如图-14所示

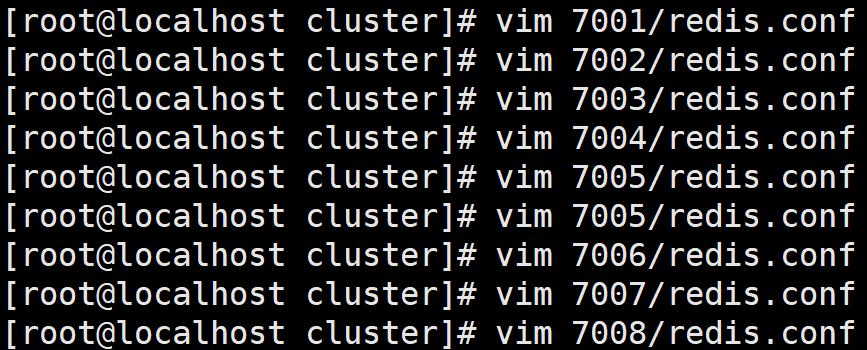


图- 14

### 编辑redis启动脚本命令

如果需要启动redis,需要执行redis启动命令9次.不是特别方便,所以编辑.sh脚本文件.

在redis根目录中新建启动和关闭的脚本文件.之后编辑脚本指令

vim start.sh

vim stop.sh

如图-15所示:编辑启动脚本文件

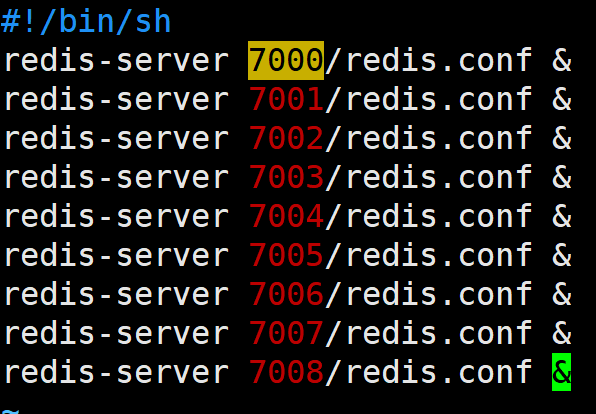


图- 15

编辑关闭脚本文件:

#!/bin/sh

redis-cli -p 7000 shutdown &

redis-cli -p 7001 shutdown &

redis-cli -p 7002 shutdown &

redis-cli -p 7003 shutdown &

redis-cli -p 7004 shutdown &

redis-cli -p 7005 shutdown &

redis-cli -p 7006 shutdown &

redis-cli -p 7007 shutdown &

redis-cli -p 7008 shutdown &

### 安装redis集群管理工具ruby

1.解压2个jar包文件将文件导入后解压

如图-16所示

计算机生成了可选文字:
LI。。L甲L，。甘。111uL,y甘七川。一‘·。·占占J什
[rootobogonredis]#15
ruby·2.4.0ruby·2.4.0.tar.gz
[rootoboqonredis]#.
'U。。I
rubyg骊s·2.6.11rubygems·2.6.11.tgz

图- 16

2.进入ruby-2.4.0文件夹

执行命令:

$ ./configure 进行编译时间较长

$ make 进行安装时间较长

$ make install 将ruby安装到linux环境中

3.检查ruby环境

如图-17所示

计算机生成了可选文字:
[root@bogonredis]#ruby
ruby2.4.0p0(2016一12一24
[root@bogonred主s]#.
·V
reVISIOn
57164)[x86_64一linux]

图- 17

4.安装rubyGem

进入到rubygems文件夹下 执行 ruby setup.rb

如图-18所示

计算机生成了可选文字:
[root@bogonrubygems一2.6.
RubyGems2.6.11installed
Parsingdocumentationfor
--‘一
,"＃座全望p竺口
rubygems一2.6.11
Installingridocumentationforrubygems一2.6.11
===2.6.11/2017一03一16

图- 18

5.将gem-redis-3.3.3文件导入src文件中

如图-19所示

计算机生成了可选文字:
八”．、．卜、、二‘.',
[r。。t@bogon［应亚
appVeyor.yml
bin
gems三2：妞11#ts刃
bUndleF
CODEOFCONDUCT.md
CONTRIBUTING.Fd0C
CVE一2013一4287.tXt
CVE一2013一4363.txt
CVE一2015一3900.txt
hide一ib_fo几update
History.txt
lib
LICENSE.txt
MAINTAINERS.tXt
M日nif6St.tXt
MIT.tXt
POLICIES.FdOC
R日kefile
README.rd0C
redis一3.3.3.ge厅
set亦丁rb一
test
UPGRADING.rdoc
util
[root@b。g。nrubygems一2.6.11]#.

图- 19

6.执行安装redis接口命令

 如图-20所示

计算机生成了可选文字:
[rootobogonrubygems一2.6.11］券马em'install气1'
Successfullyinstalledredis一3;3二3一―-
Parsingdocumentationforredis一3.3.3
Installingridocumentationforredis一3.3.3
redis:3.'3丁－gem
Doneinstallingdocumentationforredisafter1seconds
1geminstalled
[rootobogonrubygems一2.6.11]#gemlist

图- 20

7.检查是否安装完成

命令: gem list

如图-21所示:

计算机生成了可选文字:
[root@bogonrubygems一2.6.11]#gemlist
***LOCALGENS***
bigdecimal(default:1.3.0)
did_you_mean(1.1.0)
10一console(default:0.4.6)
json(default:2.0.2)
minitest(5.10.1)
net一telnet(0.1.1)
open551(default:
power--assert(0.4.
psych(default:2.
rake(12.0.0)
rdoc(default二只一n
redis(4·0.1,L3二3二
2.0.2)
1)
2.2)
3.3.3即可

图- 21

如图所示即ruby安装成功

### Redis集群创建命令

在redis安装的根目录中执行集群创建命令,其中2表示一台主机有2台从机.程序会自己计算.

./src/redis-trib.rb create --replicas 2 192.168.126.166:7000 192.168.126.166:7001 192.168.126.166:7002 192.168.126.166:7003 192.168.126.166:7004 192.168.126.166:7005 192.168.126.166:7006 192.168.126.166:7007 192.168.126.166:7008

如图-22所示：

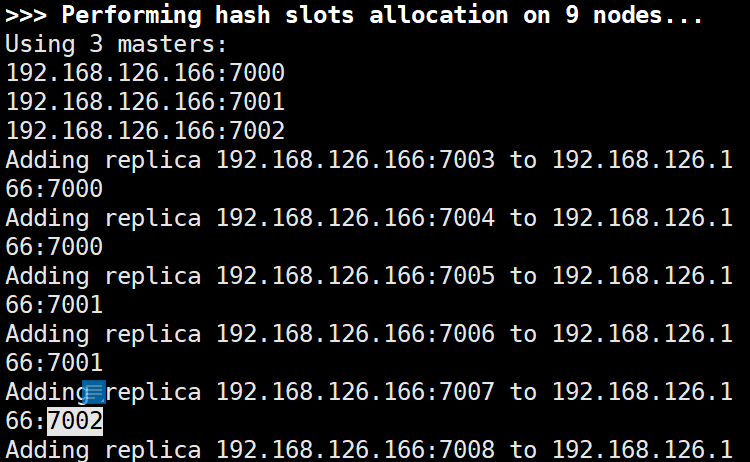


图- 22

控制台输入yes后 输出如下表示搭建成功

如图-23所示

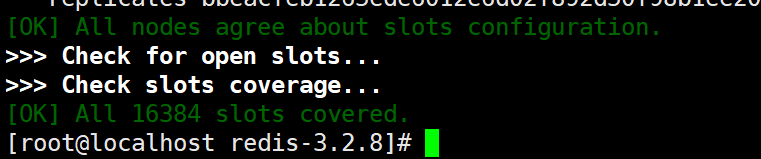


图- 23

### Redis集群高可用测试

根据Redis集群配置文件可以了解7000-7003是redis主节点.将redis主节点宕机.检查redis的从机能否实现自动的故障迁移.

## Redis集群原理

### Redis集群高可用推选原理

如图-24所示

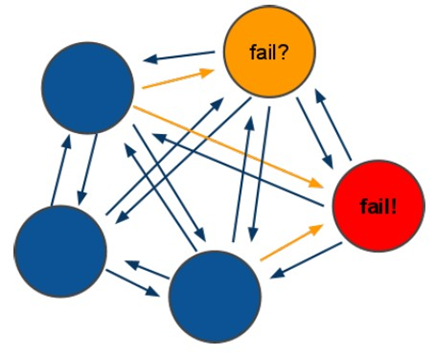


图- 24

原理说明:

Redis的所有节点都会保存当前redis集群中的全部主从状态信息.并且每个节点都能够相互通信.当一个节点发生宕机现象.则集群中的其他节点通过PING-PONG检测机制检查Redis节点是否宕机.当有半数以上的节点认为宕机.则认为主节点宕机.同时由**Redis剩余的主节点**进入选举机制.投票选举链接宕机的主节点的从机.实现故障迁移.

### Redis集群宕机条件

特点:集群中如果主机宕机,那么从机可以继续提供服务,

当主机中没有从机时,则向其它主机借用多余的从机.继续提供服务.如果主机宕机时没有从机可用,则集群崩溃.

答案:9个redis节点,节点宕机5-7次时集群才崩溃.

如图-25所示:

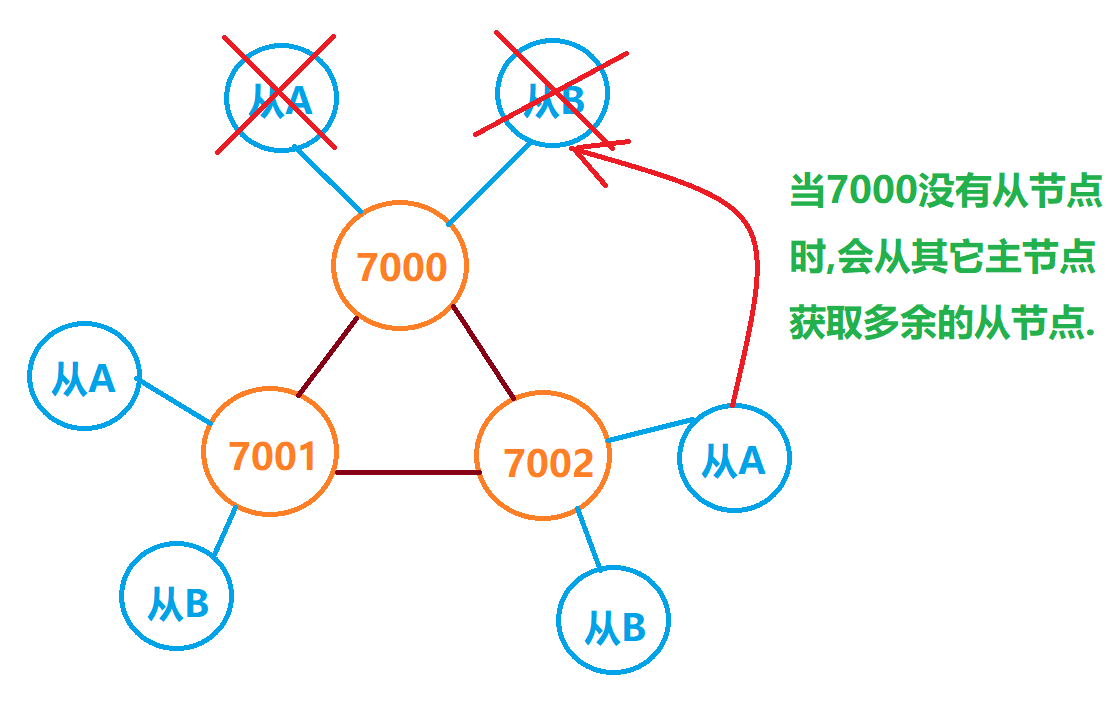


图- 25

### Redis hash槽存储数据原理

说明: RedisCluster采用此分区，所有的键根据哈希函数(CRC16[key]&16383)映射到0－16384槽内，共16384个槽位，每个节点维护部分槽及槽所映射的键值数据.根据主节点的个数,均衡划分区间.

 算法:哈希函数: Hash()=CRC16[key]&16384按位与

如图-26所示

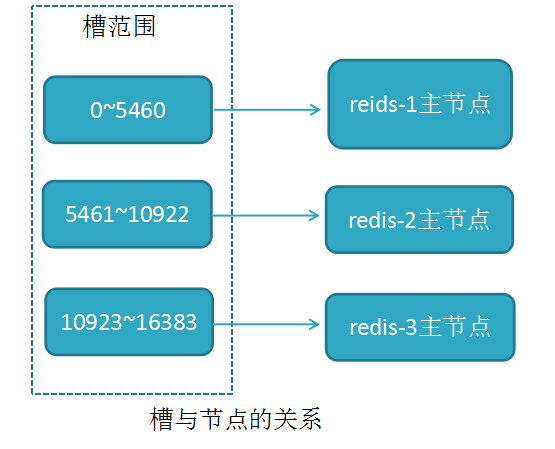


图- 26

当向redis集群中插入数据时,首先将key进行计算.之后将计算结果匹配到具体的某一个槽的区间内,之后再将数据set到管理该槽的节点中.

如图-27所示

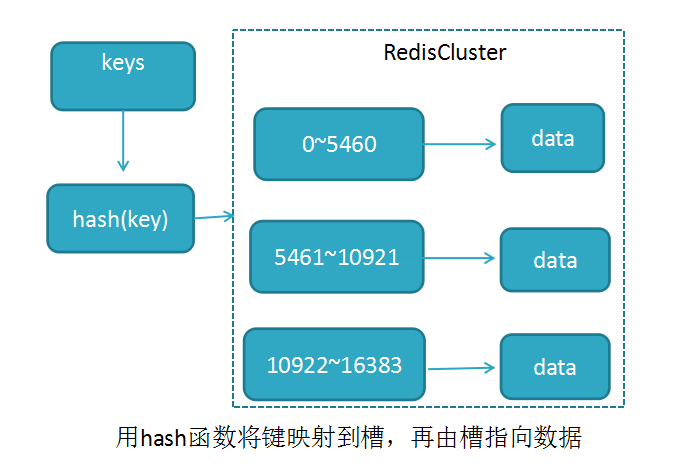


图- 27

### Redis入门案例

@Test

**public** **void** test01(){

Set<HostAndPort> nodes = **new** HashSet<>();

nodes.add(**new** HostAndPort("192.168.126.166", 7000));

nodes.add(**new** HostAndPort("192.168.126.166", 7001));

nodes.add(**new** HostAndPort("192.168.126.166", 7002));

nodes.add(**new** HostAndPort("192.168.126.166", 7003));

nodes.add(**new** HostAndPort("192.168.126.166", 7004));

nodes.add(**new** HostAndPort("192.168.126.166", 7005));

nodes.add(**new** HostAndPort("192.168.126.166", 7006));

nodes.add(**new** HostAndPort("192.168.126.166", 7007));

nodes.add(**new** HostAndPort("192.168.126.166", 7008));

JedisCluster jedisCluster = **new** JedisCluster(nodes);

jedisCluster.set("redis", "学习集群使用");

System.***out***.println("获取数据:"+jedisCluster.get("redis"));

}

说明:如果按照测试类中的方式,通过Spring容器管理对象需要创建很多个bean标签,不具有通用性.

## Spring工厂模式

### 工厂模式的作用

作用:通过工厂模式实例化对象

原因:

1. 有些对象需要经过业务处理,添加参数才可以使用.(减少代码的耦合性)
2. 有些类可能是接口或者是抽象类.不能直接实例化.需要工厂模式帮助创建对象.

### 静态工厂

特点:静态工厂中必须有静态方法.

调用逻辑:类名.静态方法

调用过程:

1. 编辑spring配置文件

<!--配置静态工厂 -->

<bean id=*"calendar1"* class=*"com.jt.manage.factory.StaticFactory"* factory-method=*"getInstance"*/>

1. 编辑工厂模式代码

**public** **class** StaticFactory {

**public** **static** Calendar getInstance(){

//JDK原生提供了实例化方法.

**return** Calendar.*getInstance*();

}

}

1. 编辑测试类

@Test

**public** **void** testStaticFactory(){

//1.实例化spring容器

ApplicationContext app =

**new** ClassPathXmlApplicationContext("spring/factory.xml");

//2.获取对象

Calendar calendar = (Calendar) app.getBean("calendar1");

System.***out***.println("获取当前时间:"+calendar.getTime());

}

### 实例化工厂

特点:首先实例化工厂,之后通过工厂对象调用工厂方法创建对象

调用特点:对象.方法

调用方法:

1. 编辑配置文件

<!--实例化工厂 -->

<bean id=*"instanceFactory"* class=*"com.jt.manage.factory.InstanceFactory"*/>

<bean id=*"calendar2"* factory-bean=*"instanceFactory"*

factory-method=*"getInstance"*/>

1. 编辑工厂模式

**public** **class** InstanceFactory {

**public** Calendar getInstance(){

**return** Calendar.*getInstance*();

}

}

1. 测试类

@Test

**public** **void** testInstanceFacotory(){

//1.实例化spring容器

ApplicationContext app =

**new** ClassPathXmlApplicationContext("spring/factory.xml");

//2.获取对象

Calendar calendar = (Calendar) app.getBean("calendar2");

System.***out***.println("获取当前时间:"+calendar.getTime());

}

### Spring工厂

特点:实现Spring中特定接口FactoryBean,容器内部自动调用

调用方法:

1. 编辑配置文件

<!--spring工厂 -->

<bean id=*"calendar3"* class=*"com.jt.manage.factory.SpringFactory"*/>

1. 编辑工厂模式

**public** **class** SpringFactory **implements** FactoryBean<Calendar>{

@Override

**public** Calendar getObject() **throws** Exception {

// **TODO** Auto-generated method stub

**return** Calendar.*getInstance*();

}

@Override

**public** Class<?> getObjectType() {

**return** Calendar.**class**;

}

//是否单例

@Override

**public** **boolean** isSingleton() {

// **TODO** Auto-generated method stub

**return** **false**;

}

}

1. 编辑测试类

@Test

**public** **void** testSpringFacotory(){

//1.实例化spring容器

ApplicationContext app =

**new** ClassPathXmlApplicationContext("spring/factory.xml");

//2.获取对象

Calendar calendar = (Calendar) app.getBean("calendar3");

System.***out***.println("获取当前时间:"+calendar.getTime());

}

## Spring整合Redis集群

### 修改配置文件

#最小空闲数

redis.minIdle=100

#最大空闲数

redis.maxIdle=300

#最大连接数

redis.maxTotal=1000

#客户端超时时间单位是毫秒

redis.timeout=5000

#最大建立连接等待时间

redis.maxWait=1000

#是否在从池中取出连接前进行检验,如果检验失败,则从池中去除连接并尝试取出另一个

redis.testOnBorrow=true

#redis cluster

redis.cluster0=192.168.126.166:7000

redis.cluster1=192.168.126.166:7001

redis.cluster2=192.168.126.166:7002

redis.cluster3=192.168.126.166:7003

redis.cluster4=192.168.126.166:7004

redis.cluster5=192.168.126.166:7005

redis.cluster6=192.168.126.166:7006

redis.cluster7=192.168.126.166:7007

redis.cluster8=192.168.126.166:7008

### 编辑工厂模式

**public** **class** JedisClusterFactory **implements** FactoryBean<JedisCluster>{

**private** JedisPoolConfig poolConfig; //注入配置文件

**private** String redisNodePrefix; //注入配置前缀

**private** Resource propertySource; //注入redis配置文件

@Override

**public** JedisCluster getObject() **throws** Exception {

Set<HostAndPort> nodes = getNodes();

JedisCluster jedisCluster = **new** JedisCluster(nodes, poolConfig);

**return** jedisCluster;

}

/\*\*

\* 1.定义Set集合

\* 2.获取配置文件中的IP和端口

\* 3.封装Set集合

\* **@return**

\*/

**public** Set<HostAndPort> getNodes(){

Set<HostAndPort> nodes = **new** HashSet<>();

**try** {

//获取Property数据

Properties properties = **new** Properties();

properties.load(propertySource.getInputStream());

//获取Redis节点数据

**for** (Object key : properties.keySet()) {

String strKey = (String) key;

//判断当前遍历的key是否为redis节点的key

**if**(strKey.startsWith(redisNodePrefix)){

//IP:端口号

String value = properties.getProperty(strKey);

String[] args = value.split(":");

HostAndPort hostAndPort =

**new** HostAndPort(args[0],Integer.*parseInt*(args[1]));

nodes.add(hostAndPort);

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** nodes;

}

@Override

**public** Class<?> getObjectType() {

**return** JedisCluster.**class**;

}

@Override

**public** **boolean** isSingleton() {

**return** **false**;

}

**public** JedisPoolConfig getPoolConfig() {

**return** poolConfig;

}

**public** **void** setPoolConfig(JedisPoolConfig poolConfig) {

**this**.poolConfig = poolConfig;

}

**public** String getRedisNodePrefix() {

**return** redisNodePrefix;

}

**public** **void** setRedisNodePrefix(String redisNodePrefix) {

**this**.redisNodePrefix = redisNodePrefix;

}

**public** Resource getPropertySource() {

**return** propertySource;

}

**public** **void** setPropertySource(Resource propertySource) {

**this**.propertySource = propertySource;

}

}

项目编辑完成后,将项目打包

### 编辑Spring配置文件

<!--配置redis集群 -->

<bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"* >

<!--最大空闲数-->

<property name=*"maxIdle"* value=*"${redis.maxIdle}"* />

<!--最大建立连接等待时间-->

<property name=*"maxWaitMillis"* value=*"${redis.maxWait}"* />

<!--是否在从池中取出连接前进行检验,如果检验失败,则从池中去除连接并尝试取出另一个-->

<property name=*"testOnBorrow"* value=*"${redis.testOnBorrow}"* />

<property name=*"maxTotal"* value=*"${redis.maxTotal}"* />

<property name=*"minIdle"* value=*"${redis.minIdle}"* />

</bean>

<!--通过工厂模式获取数据 -->

<bean id=*"jedisCluster"* class=*"com.jt.common.facotry.JedisClusterFactory"*>

<!--引入配置文件源文件 -->

<property name=*"propertySource"*>

<value>classpath:/property/redis.properties</value>

</property>

<!--引入池配置文件 -->

<property name=*"poolConfig"* ref=*"poolConfig"*/>

<!--添加配置前缀-->

<property name=*"redisNodePrefix"* value=*"redis.cluster"*/>

</bean>

### 实现集群操作

1. 注入jedisCluster对象

如图-28所示

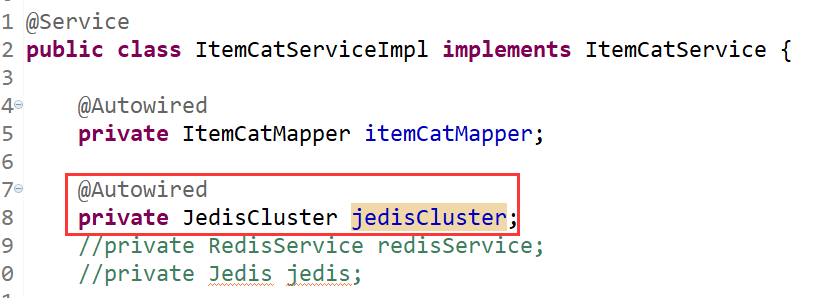


图- 28

1. 业务代码实现

/\*\*

\* 1.查询缓存

\* 2.判断是否有数据

\* 3.如果没有数据查询数据库,之后将返回的结果转化为JSON串,在保存到redis

\* 4.如果查询的结果有数据,需要将json串转化为Java对象之后返回.

\*/

@Override

public List<EasyUITree> findItemCatCache(Long parentId) {

String key = "ITEM\_CAT\_" +parentId;

String result = jedisCluster.get(key); //获取缓存的数据

List<EasyUITree> treeList = null;

try {

if(StringUtils.isEmpty(result)){

//证明缓存中没有数据

treeList = findItemCatList(parentId);

//将数据存入缓存

String easyUIJSON =

objectMapper.writeValueAsString(treeList);

jedisCluster.set(key, easyUIJSON);

System.out.println("查询数据库");

}else{

EasyUITree[] trees = objectMapper.readValue(result,EasyUITree[].class);

treeList = Arrays.asList(trees);

System.out.println("查询缓存");

}

} catch (Exception e) {

e.printStackTrace();

}

return treeList;

}

### 页面效果

如图-29所示

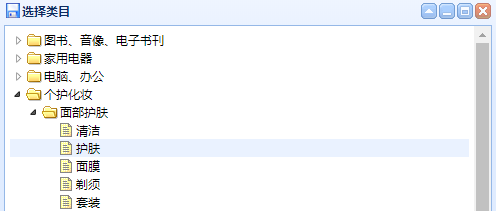


图- 29

## Spring整合Redis集群(二)

### 编辑pro文件

#redis.host=192.168.126.174

#redis.port1=6379

#redis.port2=6380

#redis.port3=6381

#redis.sentinel=192.168.126.174:26379

#redis.masterName=mymaster

redis.nodes=192.168.126.174:7000,192.168.126.174:7001,192.168.126.174:7002,192.168.126.174:7003,192.168.126.174:7004,192.168.126.174:7005,192.168.126.174:7006,192.168.126.174:7007,192.168.126.174:7008

### 编辑工厂模式

**public** **class** JedisClusterFactory **implements** FactoryBean<JedisCluster>{

@Value("${redis.nodes}") //动态获取配置文件数据

**private** String redisNodes;

@Override

**public** JedisCluster getObject() **throws** Exception {

Set<HostAndPort> nodes = getNodes();

**return** **new** JedisCluster(nodes);

}

**public** Set<HostAndPort> getNodes() {

Set<HostAndPort> nodes = **new** HashSet<>();

String[] strNodes = redisNodes.split(",");

//node IP:端口

**for** (String node : strNodes) {

String[] params = node.split(":");

String host = params[0];

**int** port = Integer.*parseInt*(params[1]);

HostAndPort hostAndPort =

**new** HostAndPort(host, port);

nodes.add(hostAndPort);

}

**return** nodes;

}

@Override

**public** Class<?> getObjectType() {

**return** JedisCluster.**class**;

}

@Override

**public** **boolean** isSingleton() {

**return** **true**;

}

}

### 编辑Spring配置文件

<!--定义对象与JSON转化工具类 -->

<bean id=*"objectMapper"* class=*"com.fasterxml.jackson.databind.ObjectMapper"*/>

<!--引入spring工厂,创建jedisCluster对象 -->

<bean id=*"jedisCluster"* class=*"com.jt.common.factory.JedisClusterFactory"*/>

### 效果展现

