# Redis实际业务应用

## 商品分类信息缓存处理

### 业务思路

1. 当用户点击商品分类信息时会发起request请求.之后根据parentId检索子集目录数据.
2. 当用户检索数据时应该先查询redis缓存.
3. 结果为null 缓存中没有数据. 需要查询数据库.将查询的结果通过工具API转化为json串.之后保存到redis中.
4. 结果不为null 获取的数据一定是json串.需要利用工具API将json串转化为对象.

### 编辑业务实现

//该方法,查询缓存

@SuppressWarnings("unchecked")

@Override

**public** List<EasyUI\_Tree> findItemCatByCache(Long parentId) {

List<EasyUI\_Tree> treeList = **new** ArrayList<EasyUI\_Tree>();

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

String result = jedis.get(key);

**if**(StringUtils.*isEmpty*(result)) {

//表示缓存中没有数据,应该查询数据库

treeList = findItemCatByParentId(parentId);

//将对象转化为json

String json =

ObjectMapperUtil.*toJSON*(treeList);

//将数据保存到缓存中

jedis.set(key, json);

}**else** {

//表示缓存中有数据,将json串转化为对象

treeList =

ObjectMapperUtil.*toObject*(result, treeList.getClass());

}

**return** treeList;

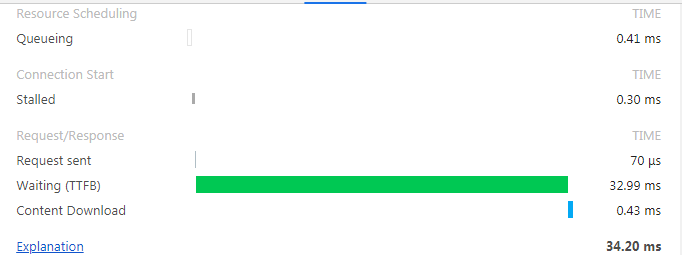
}

### 速度对比

1. 没有缓存



1. 添加缓存



## Redis分片(分区)机制

### 业务需求

说明:单台redis管理内存是有上限的.一般设定512M-1024M,如果公司业务要求保存海量的缓存数据.则一台redis不能实现该操作.

思路:内存动态扩容

准备多台redis共同为用户提供服务.

总结:

Redis分片主要目的实现redis内存扩容.对用户而言使用的就是一大台redis.但是内部每个redis中保存的数据都是不相同的.

### 准备多台redis

设计:

1. redis-6379.conf 默认端口 6379
2. redis-6380.conf 6380
3. redis-6381.conf 6381

步骤:

1. 在redis根目录中准备新文件夹 shards
2. 复制配置文件

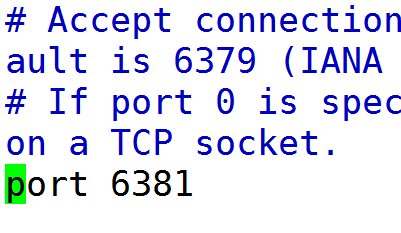
[root@localhost redis]# cp redis.conf shards/redis-6379.conf

[root@localhost redis]# cp redis.conf shards/redis-6380.conf

[root@localhost redis]# cp redis.conf shards/redis-6381.conf

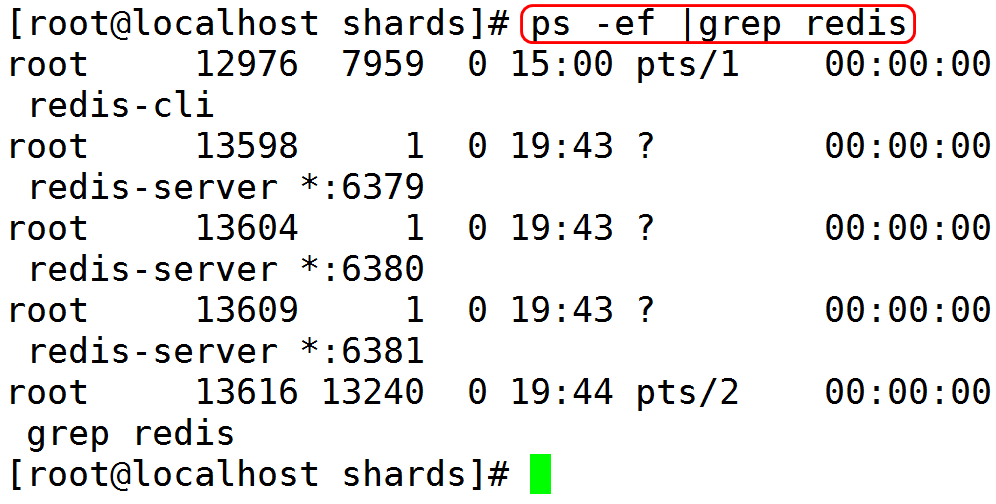
1. 分别修改端口号

改为6379/6380/6381



1. 启动多台redis

redis-server redis-6379.conf



### Redis分片入门案例

/\*\*

\* 测试redis分片机制

\*/

@Test

**public** **void** testShards() {

List<JedisShardInfo> shards =

**new** ArrayList<JedisShardInfo>();

String host = "192.168.182.129";

shards.add(**new** JedisShardInfo(host, 6379));

shards.add(**new** JedisShardInfo(host, 6380));

shards.add(**new** JedisShardInfo(host, 6381));

ShardedJedis jedis =

**new** ShardedJedis(shards);

jedis.set("1903", "分片测试");

System.***out***.println("获取数据:"+jedis.get("1903"));

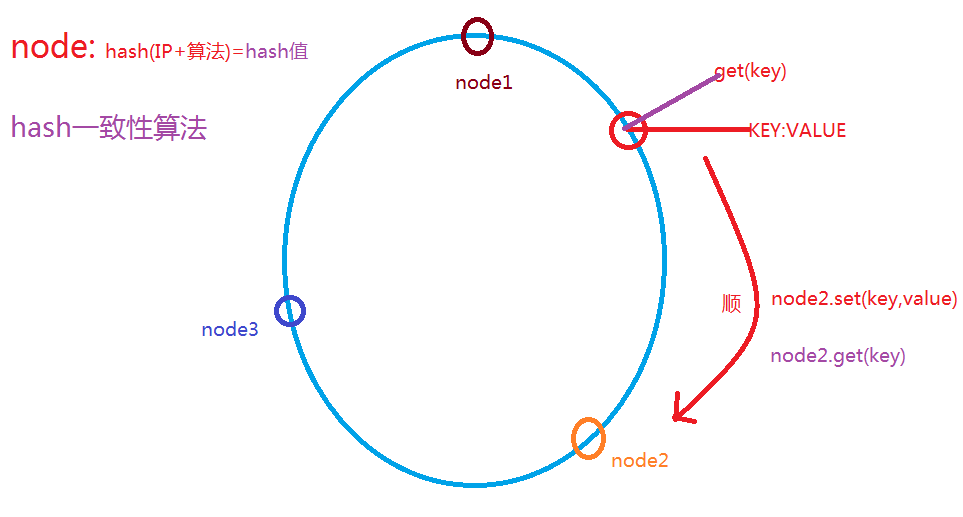
}

## hash一致性原则

### 原理说明

hash一致性算法是在程序API中进行的.与redis节点没有必然关系.如何保证数据可以正确存储.

hash: hash值一般是32位数据16进制数. 共有2^128种可能性



1. 确定节点node在hash位置.
2. 当用户保存数据时 将key经过hash计算获取位置.
3. 按照顺时针的方向查找最近node节点之后进行赋值操作.

### hash一致性特性

#### 均衡性

问题说明:由于所有的节点都是通过hash计算获取的位置,则可能会出现位置分布不均匀的现象.导致数据负载不均.

解决方案: 引入**虚拟节点**.

说明:当hash一致性算法发现现在数据保存不均匀时会引入虚拟节点的概念.目的是为了平衡数据.

#### 单调性

说明:当节点**新增**时,根据hash一致性算法要求,能够实现自动化的数据的迁移.

难点问题:node节点能否减少?答一定不能减少,如果减少则内存缺失.分片失效.

#### 分散性

概念:由于分布式部署,某些模块不能使用全部的内存空间(node节点),这时可能会出现 **一个key可能会有多个位置**.

#### 负载

概念:由于分布式部署.某些模块不能使用全部内存空间.则可能会出现**同一个位置有多个key的**现象.

## SpringBoot整合redis分片

### 编辑pro配置文件

#redis分片

redis.nodes=192.168.182.129:6379,192.168.182.129:6380,192.168.182.129:6381

### 编辑配置类

@Configuration //标识我是一个配置类

@PropertySource("classpath:/properties/redis.properties")

**public** **class** RedisConfig {

@Value("${redis.nodes}")

**private** String nodes;

@Bean

**public** ShardedJedis shardedJedis() {

List<JedisShardInfo> shards = **new** ArrayList<>();

String[] strNodes = nodes.split(","); //[node,node,node]

**for** (String strNode : strNodes) { //IP:端口

String[] node = strNode.split(":");//[IP,port]

String host = node[0];

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

shards.add(**new** JedisShardInfo(host, port));

}

**return** **new** ShardedJedis(shards);

}

}

### 分片总结

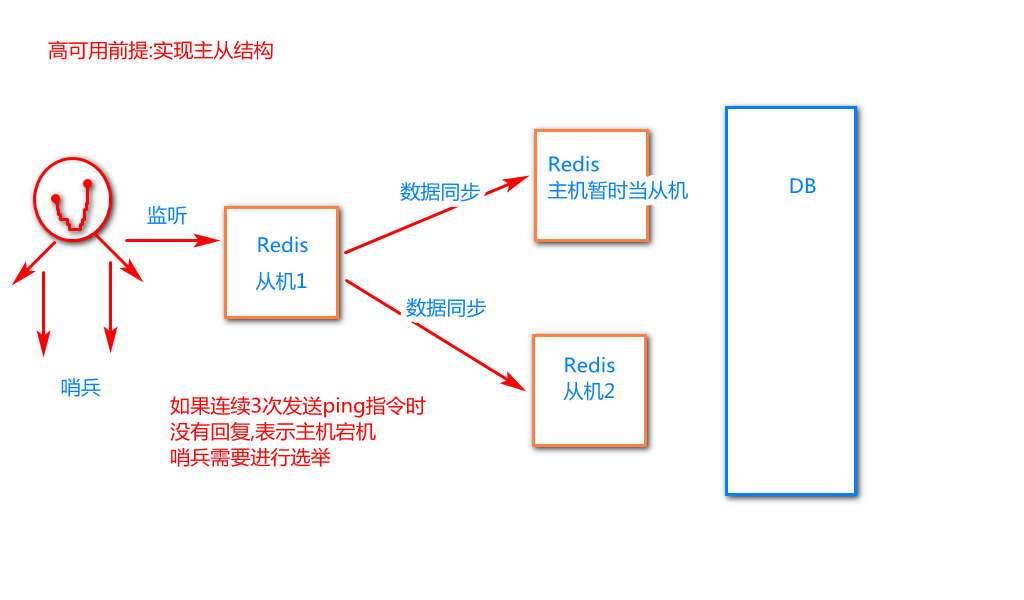
1. 分片主要的作用实现了内存的扩容.
2. 使用redis分片的性能是最高的.(hash操作都发生在服务器端tomcat)
3. 如果一个redis分片宕机.则整个分片不能使用.

## Redis哨兵机制

### 哨兵说明

redis可以利用哨兵实现redis高可用.使用时必须满足主从结构.之后在内部有心跳检测机制.如果发现某个节点宕机,则可以动态实现故障的迁移.

### 哨兵工作原理

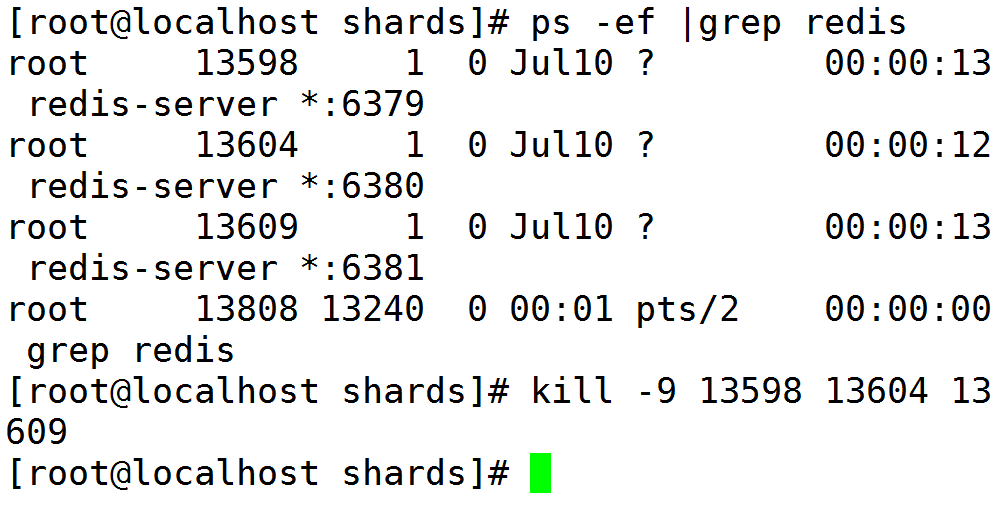


1. 当主机和从机实现类主从结构之后,程序开始正常的同步数据.
2. 当哨兵启动时整个redis高可用生效.
3. 当哨兵启动时,会监听主机的信息.利用心跳检测(PING-PONG机制)检查当前主机是否存活.如果连续3次没有返回值则表明主机宕机.

哨兵监听主机时会记录当前主机的全部从节点信息.当主机宕机之后由哨兵负责推选出新的主节点.如果哨兵确定了新的主节点.则其余的节点都是当前主节点的从机.

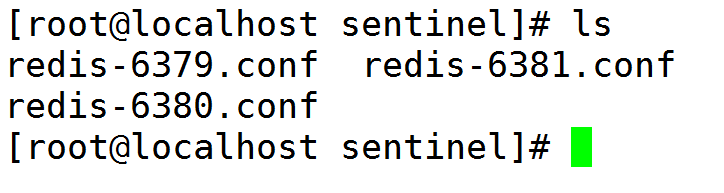
### 主从搭建步骤

#### 关闭原始服务



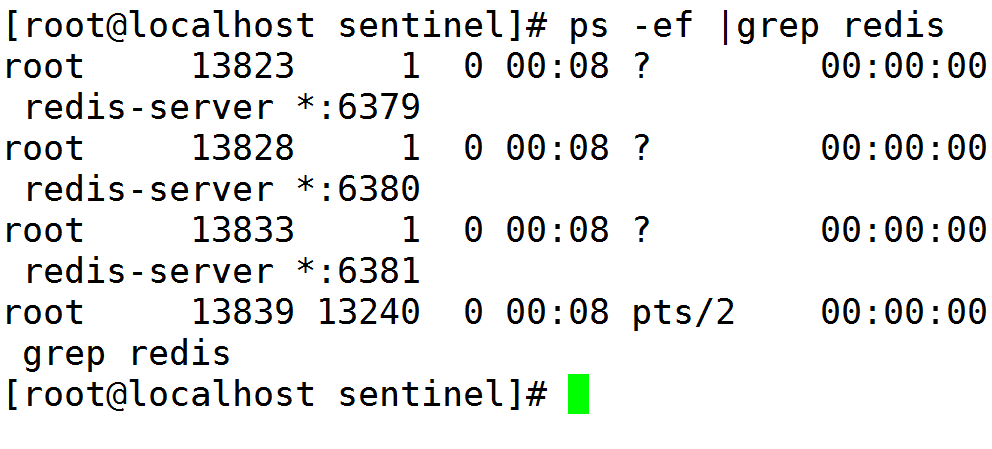
#### 复制分片改名为哨兵

cp -r shards sentinel



#### 启动三台redis

redis-server redis-6379.conf …….



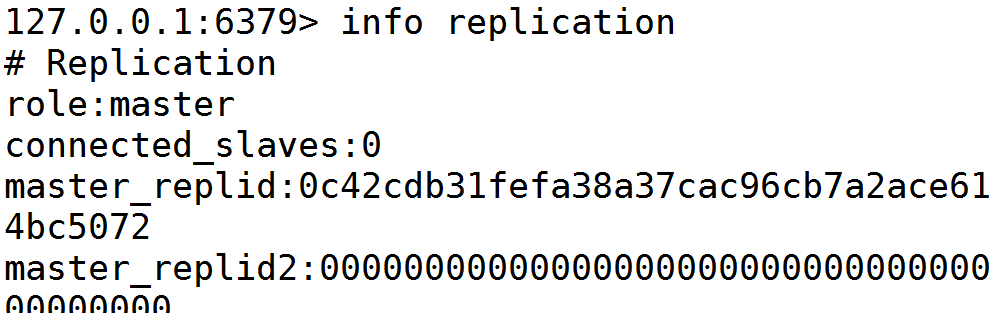
#### 配置主从

主从划分:

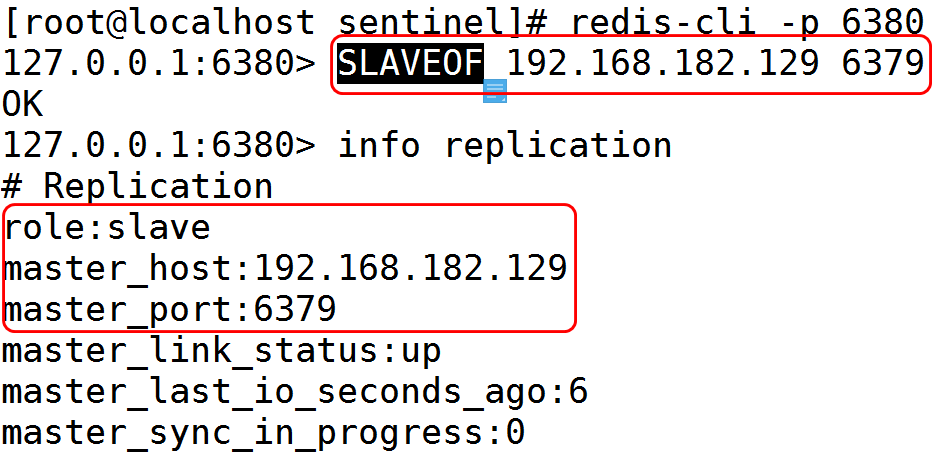
主机:6379

从机:6380/6381

默认条件下主机状态:



主从挂载:

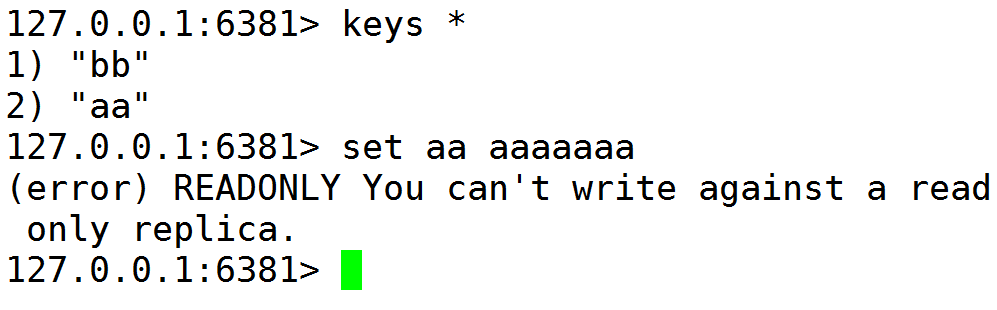


分别将6380/6381挂载到6379上.

#### 主从同步测试

说明:在redis6379中添加数据,检查6380/6381是否实现了数据同步.

经过测试发现.从机已经实时同步数据,并且从机不可以写入数据,是只读的

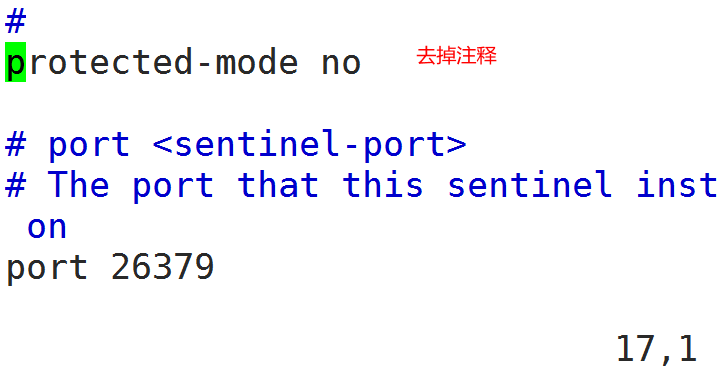


### 哨兵搭建

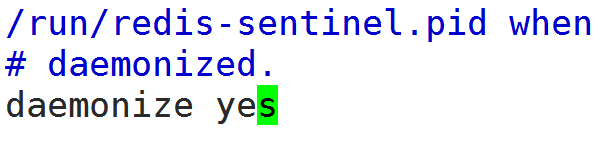
#### 复制哨兵的配置文件

[root@localhost redis]# cp sentinel.conf sentinel

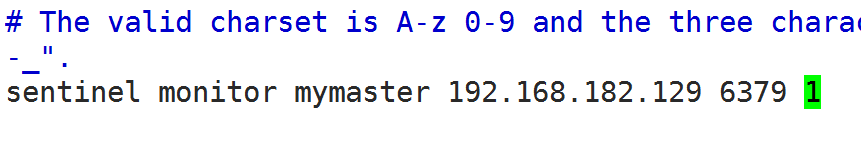
#### 修改保护模式



#### 开启后台启动



#### 修改哨兵监听

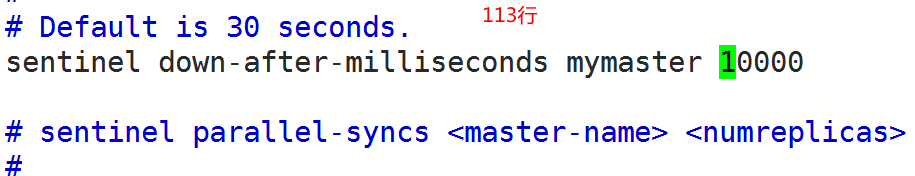


mymaster:代表主机的变量名

IP 端口:表示真正的主机.

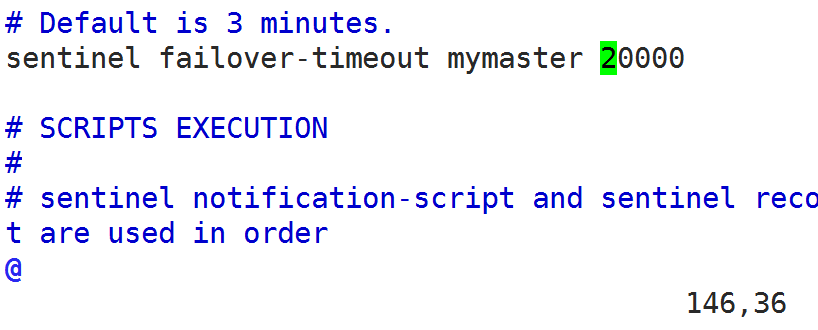
1: 哨兵投票 生效的票数

#### 修改投票时间



#### 修改重选时间

如果在20秒内,选举没有成功,则重新选举.



### 哨兵高可用测试

测试方案:

1. 将主机宕机
2. 10秒之后检查节点状态,是否切换了主机.
3. 将旧的主机重启.检查旧的主机是否为新主机的从机.

启动哨兵命令:

[root@localhost sentinel]# redis-sentinel sentinel.conf

## SpringBoot整合redis哨兵

### 入门案例

/\*\*

\* 测试redis哨兵机制

\* 说明:链接哨兵时,HOST和端口写的是哨兵的地址

\* 内部由哨兵将请求转给主机.

\*/

@Test

**public** **void** testSentinel() {

Set<String> sentinels = **new** HashSet<>();

sentinels.add("192.168.182.129:26379");

JedisSentinelPool pool =

**new** JedisSentinelPool("mymaster", sentinels);

Jedis jedis = pool.getResource();

jedis.set("1903aaaaa","哨兵测试!!!!");

System.***out***.println(jedis.get("1903aaaaa"));

}

### 编辑properties

#redis哨兵

redis.sentinel.masterName=mymaster

redis.sentinels=192.168.182.129:26379

### 编辑Spring配置文件

@Configuration //标识我是一个配置类

@PropertySource("classpath:/properties/redis.properties")

**public** **class** RedisConfig {

/\*\*

\* 实现redis哨兵配置

\*/

@Value("${redis.sentinel.masterName}")

**private** String masterName;

@Value("${redis.sentinels}")

**private** String nodes;

@Bean(name = "jedisSentinelPool") //该对象是单例的

**public** JedisSentinelPool jedisSentinelPool() {

Set<String> sentinels = **new** HashSet<>();

sentinels.add(nodes);

JedisSentinelPool pool =

**new** JedisSentinelPool(masterName, sentinels);

**return** pool;

}

//@Qualifier该注解表示指定bean赋值 用在方法中

@Bean

@Scope("prototype") //多例对象

**public** Jedis jedis(@Qualifier("jedisSentinelPool")JedisSentinelPool pool)

{

Jedis jedis = pool.getResource();

**return** jedis;

}

}