**Redis**

**NoSql数据库**

# 课程计划

1. redis介绍
   1. 什么是NoSql
   2. NoSql的分类
   3. 什么是redis
   4. Redis应用场景
2. redis安装（重点）
3. redis客户端
   1. redis自带客户端
   2. 图形界面的客户端（了解）
   3. Java客户端jedis（重点）
4. Redis数据类型（重点）
   1. String类型
   2. Map类型
   3. List类型
   4. Set类型
   5. SortedSet
5. Keys命令（了解）
6. Redis的持久化方案
   1. Rbd方式
   2. Aof方式
7. Redis的主从复制
8. Redis的集群（重点）
9. Jedis连接redis集群

# Redis介绍

## 什么是NoSql

为了解决高并发、高可用、高可扩展，大数据存储等一系列问题而产生的数据库解决方案，就是NoSql。

NoSql，叫非关系型数据库，它的全名Not only sql。它不能替代关系型数据库，只能作为关系型数据库的一个良好补充。

## NoSql的分类

* 键值(Key-Value)存储数据库

相关产品： Tokyo Cabinet/Tyrant、**Redis**、Voldemort、Berkeley DB

典型应用： 内容缓存，主要用于处理大量数据的高访问负载。

数据模型： 一系列键值对

优势： 快速查询

劣势： 存储的数据缺少结构化

* 列存储数据库

相关产品：Cassandra, **HBase**, Riak

典型应用：分布式的文件系统

数据模型：以列簇式存储，将同一列数据存在一起

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

* 文档型数据库

相关产品：CouchDB、**MongoDB**

典型应用：Web应用（与Key-Value类似，Value是结构化的）

数据模型： 一系列键值对

优势：数据结构要求不严格

劣势： 查询性能不高，而且缺乏统一的查询语法

* 图形(Graph)数据库

相关数据库：Neo4J、InfoGrid、Infinite Graph

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法。

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

## 什么是redis

Redis是使用c语言开发的一个高性能键值数据库。Redis可以通过一些键值类型来存储数据。

键值类型：

String字符类型

map散列类型

list列表类型

set集合类型

sortedset有序集合类型

## redis历史发展

2008年，意大利的一家创业公司Merzia推出了一款基于MySQL的网站实时统计系统LLOOGG，然而没过多久该公司的创始人 Salvatore Sanfilippo便 对MySQL的性能感到失望，于是他决定亲自为LLOOGG量身定做一个数据库，并于2009年开发完成，这个数据库就是Redis。 不过Salvatore Sanfilippo并不满足只将Redis用于LLOOGG这一款产品，而是希望更多的人使用它，于是在同一年Salvatore Sanfilippo将Redis开源发布，并开始和Redis的另一名主要的代码贡献者Pieter Noordhuis一起继续着Redis的开发，直到今天。

Salvatore Sanfilippo自己也没有想到，短短的几年时间，Redis就拥有了庞大的用户群体。Hacker News在2012年发布了一份数据库的使用情况调查，结果显示有近12%的公司在使用Redis。国内如新浪微博、街旁网、知乎网，国外如GitHub、Stack Overflow、Flickr等都是Redis的用户。

VMware公司从2010年开始赞助Redis的开发， Salvatore Sanfilippo和Pieter Noordhuis也分别在3月和5月加入VMware，全职开发Redis。

## redis的应用场景

缓存（数据查询、短连接、新闻内容、商品内容等等）。（**最多使用**）

分布式集群架构中的session分离。

聊天室的在线好友列表。

任务队列。（秒杀、抢购、12306等等）

应用排行榜。

网站访问统计。

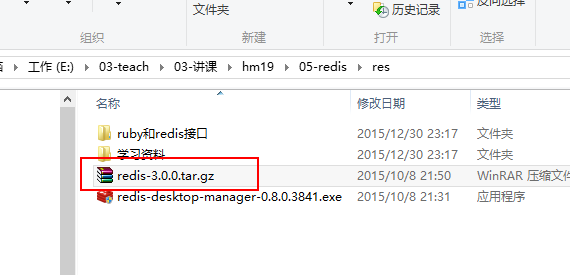
数据过期处理（可以精确到毫秒）

# redis安装

## redis下载

官网地址：<http://redis.io/>

下载地址：<http://download.redis.io/releases/redis-3.0.0.tar.gz>



## redis的安装

redis的安装环境会安装到linux系统中。

第一步：安装VMware，并且在VMware中安装centos系统（参考linux教程）。

第二步：将redis的压缩包，上传到linux系统

第三步：对redis的压缩包进行解压缩

Redis解压缩之后的文件是用c语言写的源码文件

[root@itheima ~]# tar -zxf redis-3.0.0.tar.gz

第四步：安装c语言环境（安装centos之后，自带c语言环境）

[root@itheima ~]# yum install gcc-c++

第五步：编译redis源码

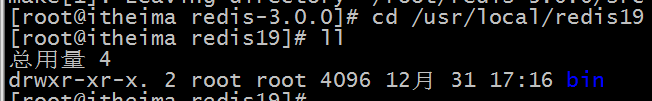
[root@itheima ~]# cd redis-3.0.0

[root@itheima redis-3.0.0]# make

第六步：安装redis

[root@itheima redis-3.0.0]# make install PREFIX=/usr/local/redis19

第七步：查看是否安装成功



## redis启动

### 前端启动

前端启动的命令：

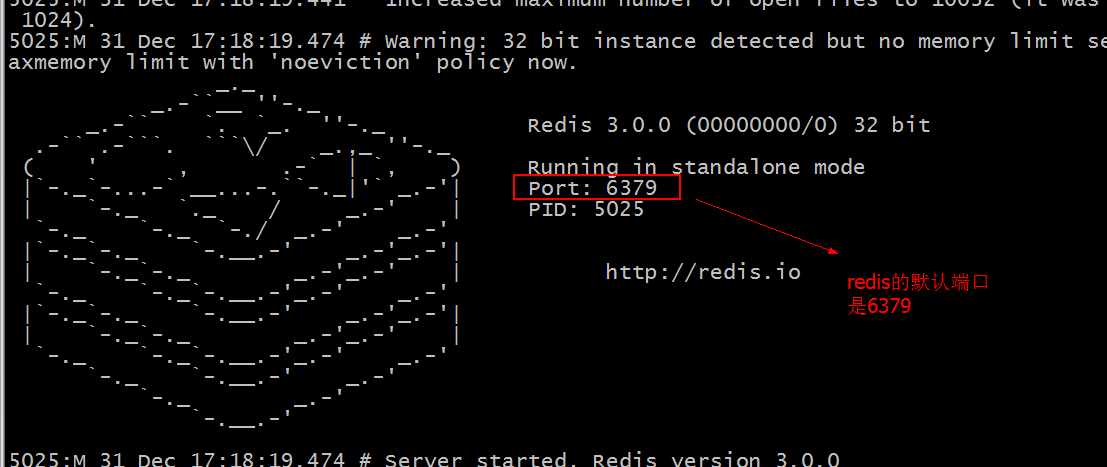
[root@itheima bin]# ./redis-server

前端启动的关闭：

强制关闭：Ctrl+c

正常关闭：[root@itheima bin]# ./redis-cli shutdown

启动界面：



前端启动的问题：

一旦客户端关闭，则redis服务也停掉。

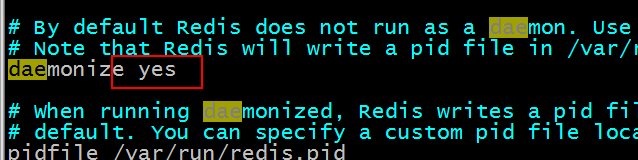
### 后端启动

第一步：需要将redis解压之后的源码包中的redis.conf文件拷贝到bin目录下

[root@itheima bin]# cp /root/redis-3.0.0/redis.conf ./

第二步：修改redis.conf文件，将daemonize改为yes

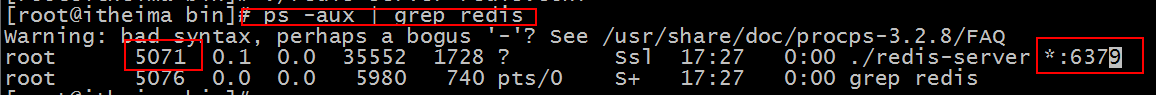
先要使用vim redis.conf



第三步：使用命令后端启动redis

[root@itheima bin]# ./redis-server redis.conf

第四步：查看是否启动成功



关闭后端启动的方式：

强制关闭：[root@itheima bin]# kill -9 5071

正常关闭：[root@itheima bin]# ./redis-cli shutdown

在项目中，建议使用正常关闭。

因为redis作为缓存来使用的话，将数据存储到内存中，如果使用正常关闭，则会将内存数据持久化到本地之后，再关闭。

如果是强制关闭，则不会进行持久化操作，可能会造成部分数据的丢失。

# Redis客户端

## Redis自带的客户端

* 启动

启动客户端命令：[root@itheima bin]# ./redis-cli -h 127.0.0.1 -p 6379

-h：指定访问的redis服务器的ip地址

-p：指定访问的redis服务器的port端口

还可以写成：[root@itheima bin]# ./redis-cli

使用默认配置：默认的ip【127.0.0.1】，默认的port【6379】

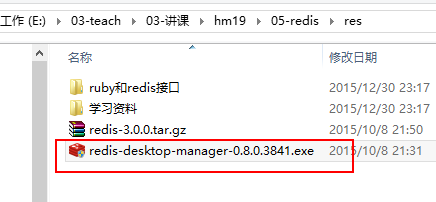
* 关闭

Ctrl+c

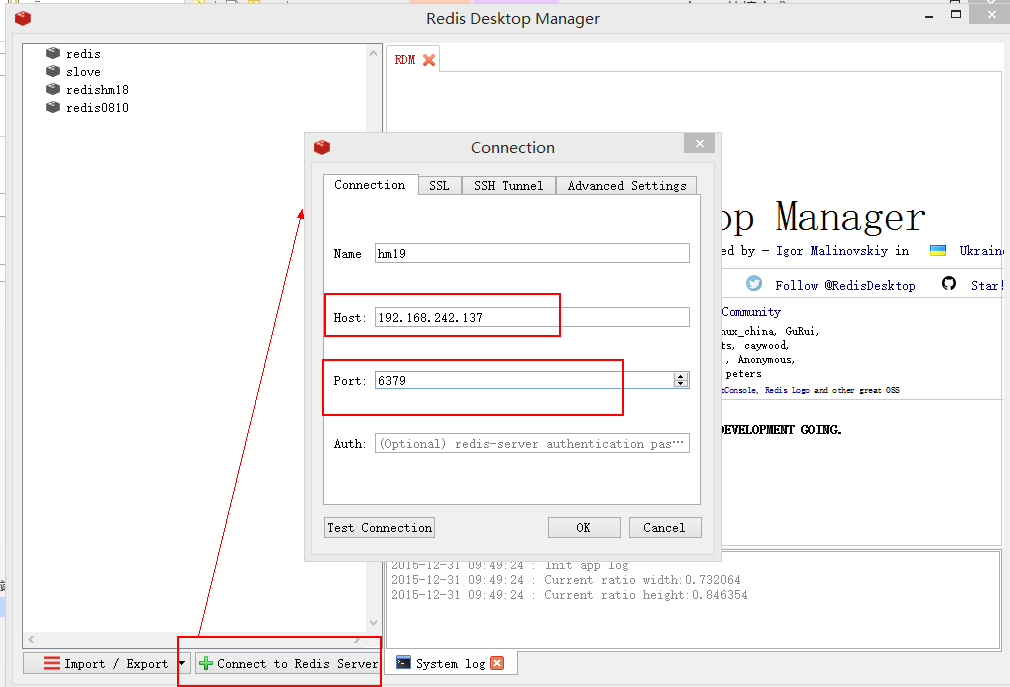
127.0.0.1:6379> quit

## 图形界面客户端

安装文件位置：

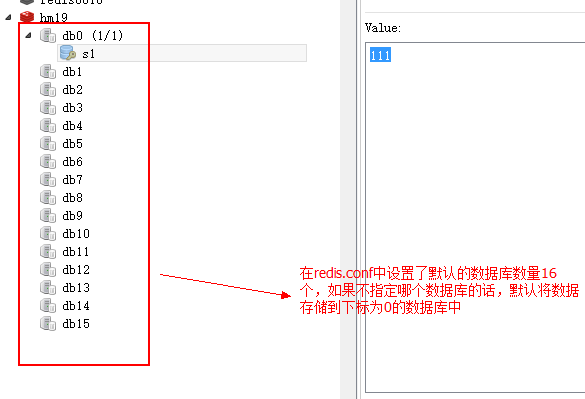


安装之后，打开如下：

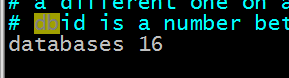


防火墙设置：

|  |
| --- |
| [root@itheima redis-3.0.0]# **vim /etc/sysconfig/iptables**  # Firewall configuration written by system-config-firewall  # Manual customization of this file is not recommended.  \*filter  :INPUT ACCEPT [0:0]  :FORWARD ACCEPT [0:0]  :OUTPUT ACCEPT [0:0]  -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT  -A INPUT -p icmp -j ACCEPT  -A INPUT -i lo -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 3306 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 8080 -j ACCEPT  **-A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT**  -A INPUT -j REJECT --reject-with icmp-host-prohibited  -A FORWARD -j REJECT --reject-with icmp-host-prohibited  COMMIT  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  "/etc/sysconfig/iptables" 16L, 677C 已写入  [root@itheima redis-3.0.0]# **service iptables restart**  iptables：清除防火墙规则： [确定]  iptables：将链设置为政策 ACCEPT：filter [确定]  iptables：正在卸载模块： [确定]  iptables：应用防火墙规则： [确定]  [root@itheima redis-3.0.0]# |



Redis.conf中的数据库数量的设置：



选择数据库的方式：

使用select 加上数据库的下标 就可以选择指定的数据库来使用，下标从0开始

127.0.0.1:6379> select 15

OK

127.0.0.1:6379[15]>

## Jedis客户端

### jedis介绍

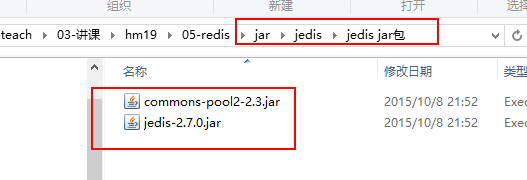
Redis不仅是使用命令来操作，现在基本上主流的语言都有客户端支持，比如java、C、C#、C++、php、Node.js、Go等。

在官方网站里列一些Java的客户端，有**Jedis**、Redisson、Jredis、JDBC-Redis、等其中官方推荐使用Jedis和Redisson。 在企业中用的最多的就是Jedis，下面我们就重点学习下Jedis。

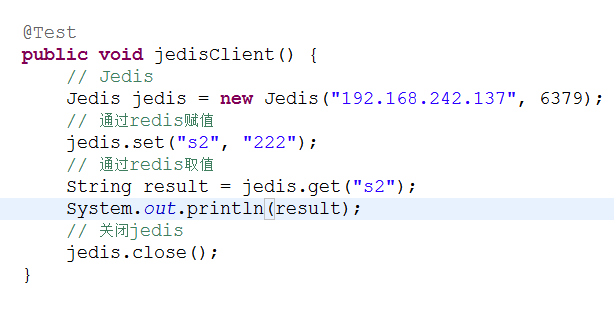
Jedis同样也是托管在github上，地址：<https://github.com/xetorthio/jedis>

### 工程搭建

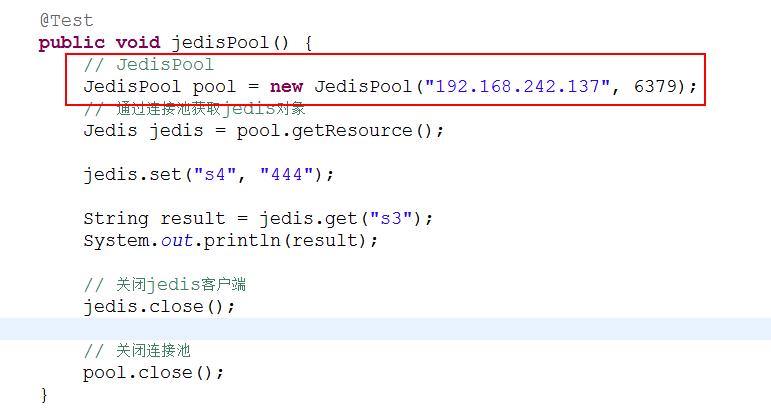
添加jar包



### 单实例连接redis



### 使用jedis连接池连接redis服务器



### Spring整合jedisPool（自学）

* 添加spring的jar包
* 配置spring配置文件applicationContext.xml

|  |
| --- |
| <?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:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-3.2.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-3.2.xsd "*>  <!-- 连接池配置 -->  <bean id=*"jedisPoolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <!-- 最大连接数 -->  <property name=*"maxTotal"* value=*"30"* />  <!-- 最大空闲连接数 -->  <property name=*"maxIdle"* value=*"10"* />  <!-- 每次释放连接的最大数目 -->  <property name=*"numTestsPerEvictionRun"* value=*"1024"* />  <!-- 释放连接的扫描间隔（毫秒） -->  <property name=*"timeBetweenEvictionRunsMillis"* value=*"30000"* />  <!-- 连接最小空闲时间 -->  <property name=*"minEvictableIdleTimeMillis"* value=*"1800000"* />  <!-- 连接空闲多久后释放, 当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->  <property name=*"softMinEvictableIdleTimeMillis"* value=*"10000"* />  <!-- 获取连接时的最大等待毫秒数,小于零:阻塞不确定的时间,默认-1 -->  <property name=*"maxWaitMillis"* value=*"1500"* />  <!-- 在获取连接的时候检查有效性, 默认false -->  <property name=*"testOnBorrow"* value=*"false"* />  <!-- 在空闲时检查有效性, 默认false -->  <property name=*"testWhileIdle"* value=*"true"* />  <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->  <property name=*"blockWhenExhausted"* value=*"false"* />  </bean>  <!-- redis单机 通过连接池 -->  <bean id=*"jedisPool"* class=*"redis.clients.jedis.JedisPool"*  destroy-method=*"close"*>  <constructor-arg name=*"poolConfig"* ref=*"jedisPoolConfig"* />  <constructor-arg name=*"host"* value=*"192.168.242.130"* />  <constructor-arg name=*"port"* value=*"6379"* />  </bean>  </beans> |

* 测试代码

|  |
| --- |
| @Test  **public** **void** testJedisPool() {  JedisPool pool = (JedisPool) applicationContext.getBean("jedisPool");  Jedis jedis = **null**;  **try** {  jedis = pool.getResource();  jedis.set("name", "lisi");  String name = jedis.get("name");  System.*out*.println(name);  } **catch** (Exception ex) {  ex.printStackTrace();  } **finally** {  **if** (jedis != **null**) {  // 关闭连接  jedis.close();  }  }  } |

# Redis数据类型

## String

### 命令

#### 赋值

语法：SET key value

|  |
| --- |
| *127.0.0.1:6379> set test 123*  *OK* |

#### 取值

语法：GET key

|  |
| --- |
| *127.0.0.1:6379> get test*  *"123“* |

#### 取值并赋值

语法：GETSET key value

|  |
| --- |
| 127.0.0.1:6379> getset s2 222  "111"  127.0.0.1:6379> get s2  "222" |

#### 设置/获取多个键值

*语法：*

*MSET key value [key value …]*

*MGET key [key …]*

|  |
| --- |
| 127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3  OK  127.0.0.1:6379> get k1  "v1"  127.0.0.1:6379> mget k1 k3  1) "v1"  2) "v3" |

#### 删除

语法：DEL key

|  |
| --- |
| *127.0.0.1:6379> del test*  (integer) 1 |

#### 数值增减

* 递增数字

当存储的字符串是整数时，Redis提供了一个实用的命令INCR，其作用是让当前键值递增，并返回递增后的值。

语法：INCR key

|  |
| --- |
| 127.0.0.1:6379> incr num  (integer) 1  127.0.0.1:6379> incr num  (integer) 2  127.0.0.1:6379> incr num  (integer) 3 |

* 增加指定的整数

语法：INCRBY key increment

|  |
| --- |
| 127.0.0.1:6379> incrby num 2  (integer) 5  127.0.0.1:6379> incrby num 2  (integer) 7  127.0.0.1:6379> incrby num 2  (integer) 9 |

* 递减数值

语法：DECR key

|  |
| --- |
| 127.0.0.1:6379> decr num  (integer) 9  127.0.0.1:6379> decr num  (integer) 8 |

* 减少指定的整数

语法：DECRBY key decrement

|  |
| --- |
| 127.0.0.1:6379> decr num  (integer) 6  127.0.0.1:6379> decr num  (integer) 5  127.0.0.1:6379> decrby num 3  (integer) 2  127.0.0.1:6379> decrby num 3  (integer) -1 |

#### 其它命令(自学)

##### 向尾部追加值

APPEND的作用是向键值的末尾追加value。如果键不存在则将该键的值设置为value，即相当于 SET key value。返回值是追加后字符串的总长度。

*语法：APPEND key value*

|  |
| --- |
| 127.0.0.1:6379> set str hello  OK  127.0.0.1:6379> append str " world!"  (integer) 12  127.0.0.1:6379> get str  "hello world!" |

##### 获取字符串长度

STRLEN命令返回键值的长度，如果键不存在则返回0。

*语法：STRLEN key*

|  |
| --- |
| *127.0.0.1:6379> strlen str*  *(integer) 0*  *127.0.0.1:6379> set str hello*  *OK*  *127.0.0.1:6379> strlen str*  *(integer) 5* |

### 应用

#### 自增主键

商品编号、订单号采用string的递增数字特性生成。

定义商品编号key：items:id

192.168.101.3:7003> INCR items:id

(integer) 2

192.168.101.3:7003> INCR items:id

(integer) 3

## Hash

散列类型

### 使用string的问题

假设有User对象以JSON序列化的形式存储到Redis中，User对象有id，username、password、age、name等属性，存储的过程如下：

保存、更新：

User对象 🡪 json(string) 🡪 redis

如果在业务上只是更新age属性，其他的属性并不做更新我应该怎么做呢？ 如果仍然采用上边的方法在传输、处理时会造成资源浪费，下边讲的hash可以很好的解决这个问题。

### redis hash介绍

hash叫散列类型，它提供了字段和字段值的映射。字段值只能是字符串类型，不支持散列类型、集合类型等其它类型。如下：



### 命令

#### 赋值

HSET命令不区分插入和更新操作，当执行插入操作时HSET命令返回1，当执行更新操作时返回0。

* *一次只能设置一个字段值*

*语法：HSET key field value*

|  |
| --- |
| *127.0.0.1:6379> hset user username zhangsan*  *(integer) 1* |

* *一次可以设置多个字段值*

*语法：HMSET key field value [field value ...]*

|  |
| --- |
| *127.0.0.1:6379> hmset user age 20 username lisi*  *OK* |

* 当字段不存在时赋值，类似HSET，区别在于如果字段存在，该命令不执行任何操作

*语法：HSETNX key field value*

|  |
| --- |
| *127.0.0.1:6379> hsetnx user age 30 如果user中没有age字段则设置age值为30，否则不做任何操作*  *(integer) 0* |

#### 取值

* *一次只能获取一个字段值*

*语法：HGET key field*

|  |
| --- |
| *127.0.0.1:6379> hget user username*  *"zhangsan“* |

* *一次可以获取多个字段值*

*语法：HMGET key field [field ...]*

|  |
| --- |
| *127.0.0.1:6379> hmget user age username*  *1) "20"*  *2) "lisi"* |

* *获取所有字段值*

*语法：HGETALL key*

|  |
| --- |
| *127.0.0.1:6379> hgetall user*  *1) "age"*  *2) "20"*  *3) "username"*  *4) "lisi"* |

#### 删除字段

可以删除一个或多个字段，返回值是被删除的字段个数

*语法：HDEL key field [field ...]*

|  |
| --- |
| *127.0.0.1:6379> hdel user age*  *(integer) 1*  *127.0.0.1:6379> hdel user age name*  *(integer) 0*  *127.0.0.1:6379> hdel user age username*  *(integer) 1* |

#### 增加数字

*语法：HINCRBY key field increment*

|  |
| --- |
| *127.0.0.1:6379> hincrby user age 2 将用户的年龄加2*  *(integer) 22*  *127.0.0.1:6379> hget user age 获取用户的年龄*  *"22“* |

#### 其它命令(自学)

##### 判断字段是否存在

*语法：HEXISTS key field*

|  |
| --- |
| *127.0.0.1:6379> hexists user age 查看user中是否有age字段*  *(integer) 1*  *127.0.0.1:6379> hexists user name 查看user中是否有name字段*  *(integer) 0* |

##### 只获取字段名或字段值

*语法：*

*HKEYS key*

*HVALS key*

|  |
| --- |
| *127.0.0.1:6379> hmset user age 20 name lisi*  *OK*  *127.0.0.1:6379> hkeys user*  *1) "age"*  *2) "name"*  *127.0.0.1:6379> hvals user*  *1) "20"*  *2) "lisi"* |

##### 获取字段数量

*语法：HLEN key*

|  |
| --- |
| *127.0.0.1:6379> hlen user*  *(integer) 2* |

### 应用

#### 存储商品信息

* 商品字段

【商品id、商品名称、商品描述、商品库存、商品好评】

* 定义商品信息的key

商品1001的信息在 Redis中的key为：[items:1001]

* 存储商品信息

|  |
| --- |
| 192.168.101.3:7003> HMSET items:1001 id 3 name apple price 999.9  OK |

* 获取商品信息

|  |
| --- |
| 192.168.101.3:7003> HGET items:1001 id  "3"  192.168.101.3:7003> HGETALL items:1001  1) "id"  2) "3"  3) "name"  4) "apple"  5) "price"  6) "999.9" |

## List

### Arraylist和linkedlist的区别

Arraylist是使用数组来存储数据，特点：查询快、增删慢

Linkedlist是使用双向链表存储数据，特点：增删快、查询慢，但是查询链表两端的数据也很快。

Redis的list是采用来链表来存储的，所以对于redis的list数据类型的操作，是操作list的两端数据来操作的。

### 命令

#### 向列表两端增加元素

* *向列表左边增加元素*

*语法：LPUSH key value [value ...]*

|  |
| --- |
| *127.0.0.1:6379> lpush list:1 1 2 3*  *(integer) 3* |

* *向列表右边增加元素*

语法：*RPUSH key value [value ...]*

|  |
| --- |
| *127.0.0.1:6379> rpush list:1 4 5 6*  *(integer) 3* |

#### 查看列表

LRANGE命令是列表类型最常用的命令之一，获取列表中的某一片段，将返回start、stop之间的所有元素（包含两端的元素），索引从0开始。索引可以是负数，如：**“-1”代表最后边的一个元素**。

*语法：LRANGE key start stop*

|  |
| --- |
| *127.0.0.1:6379> lrange list:1 0 2*  *1) "2"*  *2) "1"*  *3) "4"*  127.0.0.1:6379> lrange list1 0 -1 |

#### 从列表两端弹出元素

LPOP命令从列表左边弹出一个元素，会分两步完成：

第一步是将列表左边的元素从列表中移除

第二步是返回被移除的元素值。

*语法：*

*LPOP key*

*RPOP key*

|  |
| --- |
| 127.0.0.1:6379> lpop list:1  "3“  127.0.0.1:6379> rpop list:1  "6“ |

#### 获取列表中元素的个数

*语法：LLEN key*

|  |
| --- |
| *127.0.0.1:6379> llen list:1*  *(integer) 2* |

#### 其它命令(自学)

##### 删除列表中指定的值

LREM命令会**删除列表中前count个值为value的元素**，返回实际删除的元素个数。根据count值的不同，该命令的执行方式会有所不同：

* 当count>0时， LREM会从列表左边开始删除。
* 当count<0时， LREM会从列表后边开始删除。
* 当count=0时， LREM删除所有值为value的元素。

*语法：LREM key count value*

##### 获得/设置指定索引的元素值

* 获得指定索引的元素值

语法：LINDEX key index

|  |
| --- |
| 127.0.0.1:6379> lindex l:list 2  "1" |

* 设置指定索引的元素值

语法：LSET key index value

|  |
| --- |
| 127.0.0.1:6379> lset l:list 2 2  OK  127.0.0.1:6379> lrange l:list 0 -1  1) "6"  2) "5"  3) "2"  4) "2" |

##### 只保留列表指定片段

指定范围和LRANGE一致

*语法：LTRIM key start stop*

|  |
| --- |
| *127.0.0.1:6379> lrange l:list 0 -1*  *1) "6"*  *2) "5"*  *3) "0"*  *4) "2"*  *127.0.0.1:6379> ltrim l:list 0 2*  *OK*  *127.0.0.1:6379> lrange l:list 0 -1*  *1) "6"*  *2) "5"*  *3) "0"* |

##### 向列表中插入元素

该命令首先会在列表中从左到右查找值为pivot的元素，然后根据第二个参数是BEFORE还是AFTER来决定将value插入到该元素的前面还是后面。

*语法：LINSERT key BEFORE|AFTER pivot value*

|  |
| --- |
| 127.0.0.1:6379> lrange list 0 -1  1) "3"  2) "2"  3) "1"  127.0.0.1:6379> linsert list after 3 4  (integer) 4  127.0.0.1:6379> lrange list 0 -1  1) "3"  2) "4"  3) "2"  4) "1" |

##### 将元素从一个列表转移到另一个列表中

*语法：RPOPLPUSH source destination*

|  |
| --- |
| *127.0.0.1:6379> rpoplpush list newlist*  *"1"*  *127.0.0.1:6379> lrange newlist 0 -1*  *1) "1"*  *127.0.0.1:6379> lrange list 0 -1*  *1) "3"*  *2) "4"*  *3) "2"* |

### 应用

#### 商品评论列表

思路：

在Redis中创建商品评论列表

用户发布商品评论，将评论信息转成json存储到list中。

用户在页面查询评论列表，从redis中取出json数据展示到页面。

定义商品评论列表key：

商品编号为1001的商品评论key【items: comment:1001】

|  |
| --- |
| 192.168.101.3:7001> LPUSH items:comment:1001 '{"id":1,"name":"商品不错，很好！！","date":1430295077289}' |

## Set

集合类型

集合类型：无序、不可重复

列表类型：有序、可重复

### 命令

#### 增加/删除元素

*语法：SADD key member [member ...]*

|  |
| --- |
| *127.0.0.1:6379> sadd set a b c*  *(integer) 3*  *127.0.0.1:6379> sadd set a*  *(integer) 0* |

*语法：SREM key member [member ...]*

|  |
| --- |
| *127.0.0.1:6379> srem set c d*  *(integer) 1* |

#### 获得集合中的所有元素

*语法：SMEMBERS key*

|  |
| --- |
| 127.0.0.1:6379> smembers set  1) "b"  2) "a” |

#### 判断元素是否在集合中

*语法：SISMEMBER key member*

|  |
| --- |
| 127.0.0.1:6379> sismember set a  (integer) 1  127.0.0.1:6379> sismember set h  (integer) 0 |

### 运算命令

#### 集合的差集运算 A-B

属于A并且不属于B的元素构成的集合。



*语法：SDIFF key [key ...]*

|  |
| --- |
| *127.0.0.1:6379> sadd setA 1 2 3*  *(integer) 3*  *127.0.0.1:6379> sadd setB 2 3 4*  *(integer) 3*  *127.0.0.1:6379> sdiff setA setB*  *1) "1"*  *127.0.0.1:6379> sdiff setB setA*  *1) "4"* |

#### 集合的交集运算 A ∩ B

属于A且属于B的元素构成的集合。



语法：SINTER key [key ...]

|  |
| --- |
| 127.0.0.1:6379> sinter setA setB  1) "2"  2) "3" |

#### 集合的并集运算 A ∪ B

属于A或者属于B的元素构成的集合



语法：SUNION key [key ...]

|  |
| --- |
| *127.0.0.1:6379> sunion setA setB*  *1) "1"*  *2) "2"*  *3) "3"*  *4) "4"* |

### 其它命令(自学)

#### 获得集合中元素的个数

*语法：SCARD key*

|  |
| --- |
| 127.0.0.1:6379> smembers setA  1) "1"  2) "2"  3) "3"  127.0.0.1:6379> scard setA  (integer) 3 |

#### 从集合中弹出一个元素

注意：由于集合是无序的，所有SPOP命令会从集合中随机选择一个元素弹出

*语法：SPOP key*

|  |
| --- |
| *127.0.0.1:6379> spop setA*  *"1“* |

## Sortedset

Sortedset又叫zset

Sortedset是有序集合，可排序的，但是唯一。

Sortedset和set的不同之处，是会给set中的元素添加一个分数，然后通过这个分数进行排序。

### 命令

#### 增加元素

向有序集合中加入一个元素和该元素的分数，如果该元素已经存在则会用新的分数替换原有的分数。返回值是新加入到集合中的元素个数，不包含之前已经存在的元素。

*语法：ZADD key score member [score member ...]*

|  |
| --- |
| *127.0.0.1:6379> zadd scoreboard 80 zhangsan 89 lisi 94 wangwu*  *(integer) 3*  *127.0.0.1:6379> zadd scoreboard 97 lisi*  *(integer) 0* |

#### 获取元素的分数

语法：ZSCORE key member

|  |
| --- |
| 127.0.0.1:6379> zscore scoreboard lisi  "97" |

#### 删除元素

移除有序集key中的一个或多个成员，不存在的成员将被忽略。

当key存在但不是有序集类型时，返回一个错误。

语法：ZREM key member [member ...]

|  |
| --- |
| 127.0.0.1:6379> zrem scoreboard lisi  (integer) 1 |

#### 获得排名在某个范围的元素列表

获得排名在某个范围的元素列表

* 按照元素分数**从小到大**的顺序返回索引从start到stop之间的所有元素（包含两端的元素）

*语法：ZRANGE key start stop [WITHSCORES]*

|  |
| --- |
| 127.0.0.1:6379> zrange scoreboard 0 2  1) "zhangsan"  2) "wangwu"  3) "lisi“ |

* 按照元素分数**从大到小**的顺序返回索引从start到stop之间的所有元素（包含两端的元素）

*语法：ZREVRANGE key start stop [WITHSCORES]*

|  |
| --- |
| 127.0.0.1:6379> zrevrange scoreboard 0 2  1) " lisi "  2) "wangwu"  3) " zhangsan “ |

如果需要**获得元素的分数**的可以在命令尾部加上***WITHSCORES***参数

|  |
| --- |
| 127.0.0.1:6379> zrange scoreboard 0 1 WITHSCORES  1) "zhangsan"  2) "80"  3) "wangwu"  4) "94" |

#### 获取元素的排名

* 从小到大

*语法：ZRANK key member*

|  |
| --- |
| 127.0.0.1:6379> ZRANK scoreboard lisi  (integer) 0 |

* 从大到小

*语法：ZREVRANK key member*

|  |
| --- |
| 127.0.0.1:6379> ZREVRANK scoreboard zhangsan  (integer) 1 |

#### 其它命令(自学)

##### 获得指定分数范围的元素

*语法：ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]*

|  |
| --- |
| *127.0.0.1:6379> ZRANGEBYSCORE scoreboard 90 97 WITHSCORES*  *1) "wangwu"*  *2) "94"*  *3) "lisi"*  *4) "97"*  *127.0.0.1:6379> ZRANGEBYSCORE scoreboard 70 100 limit 1 2*  *1) "wangwu"*  *2) "lisi"* |

##### 增加某个元素的分数

返回值是更改后的分数

*语法：ZINCRBY key increment member*

|  |
| --- |
| 127.0.0.1:6379> ZINCRBY scoreboard 4 lisi  "101“ |

##### 获得集合中元素的数量

*语法：ZCARD key*

|  |
| --- |
| 127.0.0.1:6379> ZCARD scoreboard  (integer) 3 |

##### 获得指定分数范围内的元素个数

*语法：ZCOUNT key min max*

|  |
| --- |
| *127.0.0.1:6379> ZCOUNT scoreboard 80 90*  *(integer) 1* |

##### 按照排名范围删除元素

*语法：ZREMRANGEBYRANK key start stop*

|  |
| --- |
| *127.0.0.1:6379> ZREMRANGEBYRANK scoreboard 0 1*  *(integer) 2*  *127.0.0.1:6379> ZRANGE scoreboard 0 -1*  *1) "lisi"* |

##### 按照分数范围删除元素

*语法：ZREMRANGEBYSCORE key min max*

|  |
| --- |
| 127.0.0.1:6379> zadd scoreboard 84 zhangsan  (integer) 1  127.0.0.1:6379> ZREMRANGEBYSCORE scoreboard 80 100  (integer) 1 |

### 应用

#### 商品销售排行榜

需求：根据商品销售量对商品进行排行显示

思路：定义商品销售排行榜（sorted set集合），Key为items:sellsort，分数为商品销售量。

**写入商品销售量**：

* 商品编号1001的销量是9，商品编号1002的销量是10

|  |
| --- |
| 192.168.101.3:7007> ZADD items:sellsort 9 1001 10 1002 |

* 商品编号1001的销量加1

|  |
| --- |
| 192.168.101.3:7001> ZINCRBY items:sellsort 1 1001 |

* 商品销量前10名：

|  |
| --- |
| 192.168.101.3:7001> ZRANGE items:sellsort 0 9 withscores |

# Keys命令

## 常用命令

### keys

返回满足给定pattern 的所有key

|  |
| --- |
| redis 127.0.0.1:6379> keys mylist\*  1) "mylist"  2) "mylist5"  3) "mylist6"  4) "mylist7"  5) "mylist8" |

### exists

确认一个key 是否存在

示例：从结果来看，数据库中不存在HongWan 这个key，但是age 这个key 是存在的

|  |
| --- |
| redis 127.0.0.1:6379> exists HongWan  (integer) 0  redis 127.0.0.1:6379> exists age  (integer) 1  redis 127.0.0.1:6379> |

### del

删除一个key

|  |
| --- |
| redis 127.0.0.1:6379> del age  (integer) 1  redis 127.0.0.1:6379> exists age  (integer) 0 |

### rename

重命名key

示例：age 成功的被我们改名为age\_new 了

|  |
| --- |
| redis 127.0.0.1:6379[1]> keys \*  1) "age"  redis 127.0.0.1:6379[1]> rename age age\_new  OK  redis 127.0.0.1:6379[1]> keys \*  1) "age\_new"  redis 127.0.0.1:6379[1]> |

### type

返回值的类型

示例：这个方法可以非常简单的判断出值的类型

|  |
| --- |
| redis 127.0.0.1:6379> type addr  string  redis 127.0.0.1:6379> type myzset2  zset  redis 127.0.0.1:6379> type mylist  list  redis 127.0.0.1:6379> |

## 设置key的生存时间

Redis在实际使用过程中更多的用作缓存，然而缓存的数据一般都是需要设置生存时间的，即：到期后数据销毁。

|  |
| --- |
| *EXPIRE key seconds 设置key的生存时间（单位：秒）key在多少秒后会自动删除*  *TTL key 查看key生于的生存时间*  *PERSIST key 清除生存时间*  *PEXPIRE key milliseconds 生存时间设置单位为：毫秒* |

*例子：*

|  |
| --- |
| *192.168.101.3:7002> set test 1 设置test的值为1*  *OK*  *192.168.101.3:7002> get test 获取test的值*  *"1"*  *192.168.101.3:7002> EXPIRE test 5 设置test的生存时间为5秒*  *(integer) 1*  *192.168.101.3:7002> TTL test 查看test的生于生成时间还有1秒删除*  *(integer) 1*  *192.168.101.3:7002> TTL test*  *(integer) -2*  *192.168.101.3:7002> get test 获取test的值，已经删除*  *(nil)* |

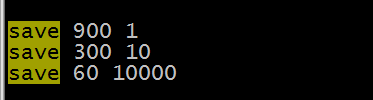
# Redis持久化方案

## Rdb方式

Redis默认的方式，redis通过快照来将数据持久化到磁盘中。

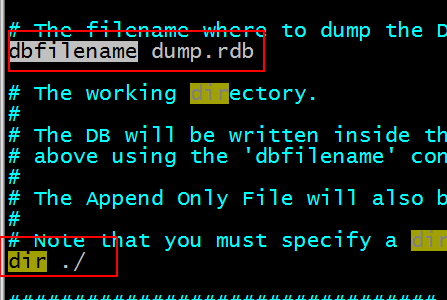
### 设置持久化快照的条件

在redis.conf中修改持久化快照的条件，如下：



### 持久化文件存储的目录

在redis.conf中可以指定持久化文件存储的目录



### Rdb问题

一旦redis非法关闭，那么会丢失最后一次持久化之后的数据。

如果数据不重要，则不必要关心。

如果数据不能允许丢失，那么要使用aof方式。

## Aof方式

Redis默认是不使用该方式持久化的。Aof方式的持久化，是操作一次redis数据库，则将操作的记录存储到aof持久化文件中。

第一步：开启aof方式的持久化方案

将redis.conf中的appendonly改为yes，即开启aof方式的持久化方案。



Aof文件存储的目录和rdb方式的一样。

Aof文件存储的名称



### 结论

在使用aof和rdb方式时，如果redis重启，则数据从aof文件加载。

# Redis的主从复制

## 什么是主从复制

持久化保证了即使redis服务重启也不会丢失数据，因为redis服务重启后会将硬盘上持久化的数据恢复到内存中，但是当redis服务器的硬盘损坏了可能会导致数据丢失，如果**通过redis的主从复制机制就可以避免这种单点故障**，如下图：

主Redis

(master)

从Redis1

(slave)

从Redis2

(slave)

说明：

* 主redis中的数据有两个副本（replication）即从redis1和从redis2，即使一台redis服务器宕机其它两台redis服务也可以继续提供服务。
* 主redis中的数据和从redis上的数据保持实时同步，当主redis写入数据时通过主从复制机制会复制到两个从redis服务上。
* 只有一个主redis，可以有多个从redis。
* 主从复制不会阻塞master，在同步数据时，master 可以继续处理client 请求
* 一个redis可以即是主又是从，如下图：

主Redis

(master)

从Redis1

(slave)

从Redis2

(slave)

从Redis3

(slave)

从Redis4

(slave)

## 主从复制设置

### 主机配置

无需配置

### 从机配置

第一步：复制出一个从机

[root@itheima redis19]# cp bin/ bin2 –r

第二步：修改从机的redis.conf

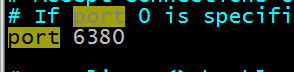
语法：Slaveof masterip masterport

slaveof 192.168.242.137 6379



第三步：修改从机的port地址为6380

在redis.conf中修改



第四步：清除从机中的持久化文件

[root@itheima bin2]# rm -rf appendonly.aof dump.rdb

第五步：启动从机

[root@itheima bin2]# ./redis-server redis.conf

第六步：启动6380的客户端

[root@itheima bin2]# ./redis-cli -p 6380

注意：

主机一旦发生增删改操作，那么从机会将数据同步到从机中

从机不能执行写操作

|  |
| --- |
| 127.0.0.1:6380> set s2 222  (error) **READONLY You can't write against a read only slave.** |

# Redis集群

## redis-cluster架构图



架构细节:

(1)所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.

(2)节点的fail是通过集群中**超过半数的节点检测**失效时才生效.

(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

(4)redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->value

Redis 集群中内置了 16384 个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

示例如下：

Server3

10001-16383

Server2

5001-10000

Server1

0-5000

## redis-cluster投票:容错



(1)集群中所有master参与投票,如果半数以上master节点与其中一个master节点通信超过(cluster-node-timeout),认为该master节点挂掉.

(2):什么时候整个集群不可用(cluster\_state:fail)?

* 如果集群任意master挂掉,且当前master没有slave，则集群进入fail状态。也可以理解成集群的[0-16383]slot映射不完全时进入fail状态。
* 如果集群超过半数以上master挂掉，无论是否有slave，集群进入fail状态。

## 安装ruby

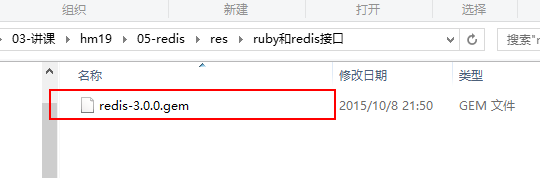
集群管理工具（redis-trib.rb）是使用ruby脚本语言编写的。

第一步：安装ruby

[root@itheima bin2]# yum install ruby

[root@itheima bin2]# yum install rubygems

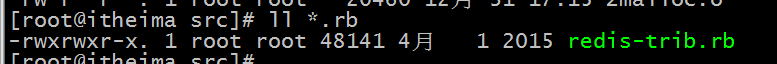
第二步：将以下文件上传到linux系统



第三步：安装ruby和redis接口

[root@itheima ~]# gem install redis-3.0.0.gem

第四步：将redis-3.0.0包下src目录中的以下文件拷贝到redis19/redis-cluster/



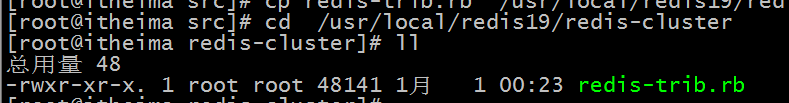
[root@itheima src]# cd /usr/local/redis19/

[root@itheima redis19]# mkdir redis-cluster

[root@itheima redis19]# cd /root/redis-3.0.0/src/

[root@itheima src]# cp redis-trib.rb /usr/local/redis19/redis-cluster

第五步：查看是否拷贝成功



## 搭建集群

搭建集群最少也得需要3台主机，如果每台主机再配置一台从机的话，则最少需要6台机器。

端口设计如下：7001-7006

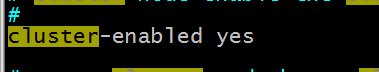
第一步：复制出一个7001机器

[root@itheima redis19]# cp bin ./redis-cluster/7001 –r

第二步：如果存在持久化文件，则删除

[root@itheima 7001]# rm -rf appendonly.aof dump.rdb

第三步：设置集群参数



第四步：修改端口



第五步：复制出7002-7006机器

[root@itheima redis-cluster]# cp 7001/ 7002 -r

[root@itheima redis-cluster]# cp 7001/ 7003 -r

[root@itheima redis-cluster]# cp 7001/ 7004 -r

[root@itheima redis-cluster]# cp 7001/ 7005 -r

[root@itheima redis-cluster]# cp 7001/ 7006 –r

第六步：修改7002-7006机器的端口

第七步：启动7001-7006这六台机器



第八步：修改start-all.sh文件的权限

[root@itheima redis-cluster]# chmod u+x start-all.sh

[root@itheima redis-cluster]# ./start-all.sh

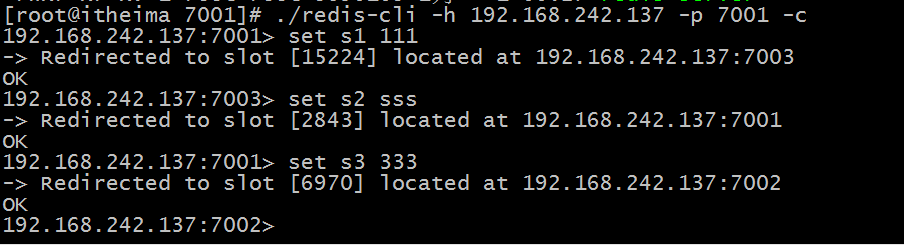
第九步：创建集群

|  |
| --- |
| [root@itheima redis-cluster]# **./redis-trib.rb create --replicas 1 192.168.242.137:7001 192.168.242.137:7002 192.168.242.137:7003 192.168.242.137:7004 192.168.242.137:7005 192.168.242.137:7006**  >>> Creating cluster  Connecting to node 192.168.242.137:7001: OK  Connecting to node 192.168.242.137:7002: OK  Connecting to node 192.168.242.137:7003: OK  Connecting to node 192.168.242.137:7004: OK  Connecting to node 192.168.242.137:7005: OK  Connecting to node 192.168.242.137:7006: OK  >>> Performing hash slots allocation on 6 nodes...  Using 3 masters:  192.168.242.137:7001  192.168.242.137:7002  192.168.242.137:7003  Adding replica 192.168.242.137:7004 to 192.168.242.137:7001  Adding replica 192.168.242.137:7005 to 192.168.242.137:7002  Adding replica 192.168.242.137:7006 to 192.168.242.137:7003  M: 8240cd0fe6d6f842faa42b0174fe7c5ddcf7ae24 192.168.242.137:7001  slots:0-5460 (5461 slots) master  M: 4f52a974f64343fd9f1ee0388490b3c0647a4db7 192.168.242.137:7002  slots:5461-10922 (5462 slots) master  M: cb7c5def8f61df2016b38972396a8d1f349208c2 192.168.242.137:7003  slots:10923-16383 (5461 slots) master  S: 66adf006fed43b3b5e499ce2ff1949a756504a16 192.168.242.137:7004  replicates 8240cd0fe6d6f842faa42b0174fe7c5ddcf7ae24  S: cbb0c9bc4b27dd85511a7ef2d01bec90e692793b 192.168.242.137:7005  replicates 4f52a974f64343fd9f1ee0388490b3c0647a4db7  S: a908736eadd1cd06e86fdff8b2749a6f46b38c00 192.168.242.137:7006  replicates cb7c5def8f61df2016b38972396a8d1f349208c2  Can I set the above configuration? (type 'yes' to accept): **yes**  >>> Nodes configuration updated  >>> Assign a different config epoch to each node  >>> Sending CLUSTER MEET messages to join the cluster  Waiting for the cluster to join..  >>> Performing Cluster Check (using node 192.168.242.137:7001)  M: 8240cd0fe6d6f842faa42b0174fe7c5ddcf7ae24 192.168.242.137:7001  slots:0-5460 (5461 slots) master  M: 4f52a974f64343fd9f1ee0388490b3c0647a4db7 192.168.242.137:7002  slots:5461-10922 (5462 slots) master  M: cb7c5def8f61df2016b38972396a8d1f349208c2 192.168.242.137:7003  slots:10923-16383 (5461 slots) master  M: 66adf006fed43b3b5e499ce2ff1949a756504a16 192.168.242.137:7004  slots: (0 slots) master  replicates 8240cd0fe6d6f842faa42b0174fe7c5ddcf7ae24  M: cbb0c9bc4b27dd85511a7ef2d01bec90e692793b 192.168.242.137:7005  slots: (0 slots) master  replicates 4f52a974f64343fd9f1ee0388490b3c0647a4db7  M: a908736eadd1cd06e86fdff8b2749a6f46b38c00 192.168.242.137:7006  slots: (0 slots) master  replicates cb7c5def8f61df2016b38972396a8d1f349208c2  [OK] All nodes agree about slots configuration.  >>> Check for open slots...  >>> Check slots coverage...  [OK] All 16384 slots covered.  [root@itheima redis-cluster]# |

## 连接集群

[root@itheima 7001]# ./redis-cli -h 192.168.242.137 -p 7001 **–c**

-c：指定是集群连接



## 查看集群信息

* 查看集群信息

|  |
| --- |
| * 192.168.242.137:7002> cluster info * cluster\_state:ok * cluster\_slots\_assigned:16384 * cluster\_slots\_ok:16384 * cluster\_slots\_pfail:0 * cluster\_slots\_fail:0 * cluster\_known\_nodes:6 * cluster\_size:3 * cluster\_current\_epoch:6 * cluster\_my\_epoch:2 * cluster\_stats\_messages\_sent:2372 * cluster\_stats\_messages\_received:2372 * 192.168.242.137:7002> |

* 查看集群节点

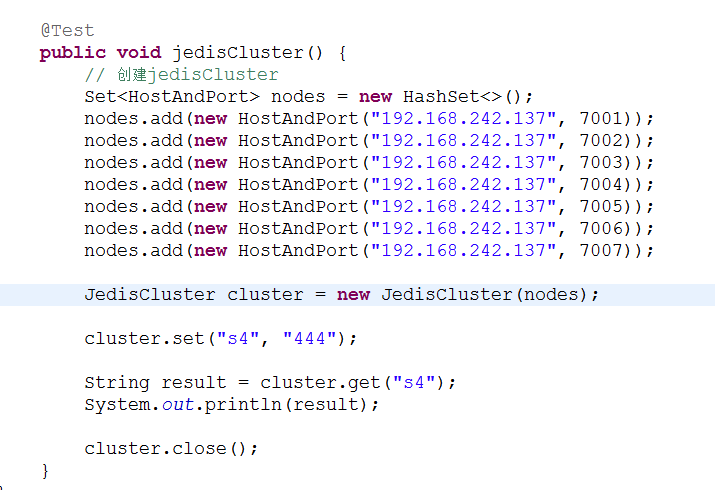
|  |
| --- |
| * 192.168.242.137:7002> cluster nodes * 8240cd0fe6d6f842faa42b0174fe7c5ddcf7ae24 192.168.242.137:7001 master - 0 1451581348093 1 connected 0-5460 * cb7c5def8f61df2016b38972396a8d1f349208c2 192.168.242.137:7003 master - 0 1451581344062 3 connected 10923-16383 * 66adf006fed43b3b5e499ce2ff1949a756504a16 192.168.242.137:7004 slave 8240cd0fe6d6f842faa42b0174fe7c5ddcf7ae24 0 1451581351115 1 connected * a908736eadd1cd06e86fdff8b2749a6f46b38c00 192.168.242.137:7006 slave cb7c5def8f61df2016b38972396a8d1f349208c2 0 1451581349101 3 connected * 4f52a974f64343fd9f1ee0388490b3c0647a4db7 192.168.242.137:7002 myself,master - 0 0 2 connected 5461-10922 * cbb0c9bc4b27dd85511a7ef2d01bec90e692793b 192.168.242.137:7005 slave 4f52a974f64343fd9f1ee0388490b3c0647a4db7 0 1451581350108 5 connected |

# jedis连接集群

## 设置防火墙

|  |
| --- |
| [root@itheima redis-cluster]# vim /etc/sysconfig/iptables  -A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT  # Firewall configuration written by system-config-firewall  # Manual customization of this file is not recommended.  \*filter  :INPUT ACCEPT [0:0]  :FORWARD ACCEPT [0:0]  :OUTPUT ACCEPT [0:0]  -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT  -A INPUT -p icmp -j ACCEPT  -A INPUT -i lo -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 3306 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 8080 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7001 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7002 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7003 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7004 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7005 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7006 -j ACCEPT  -A INPUT -m state --state NEW -m tcp -p tcp --dport 7007 -j ACCEPT  -A INPUT -j REJECT --reject-with icmp-host-prohibited  -A FORWARD -j REJECT --reject-with icmp-host-prohibited  COMMIT  ~  ~  ~  ~  "/etc/sysconfig/iptables" 23L, 1146C 已写入  [root@itheima redis-cluster]# service iptables restart  iptables：清除防火墙规则： [确定]  iptables：将链设置为政策 ACCEPT：filter [确定]  iptables：正在卸载模块： [确定]  iptables：应用防火墙规则： [确定]  [root@itheima redis-cluster]# |

## 代码



## 使用spring

* 配置applicationContext.xml

|  |
| --- |
| <!-- 连接池配置 -->  <bean id=*"jedisPoolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <!-- 最大连接数 -->  <property name=*"maxTotal"* value=*"30"* />  <!-- 最大空闲连接数 -->  <property name=*"maxIdle"* value=*"10"* />  <!-- 每次释放连接的最大数目 -->  <property name=*"numTestsPerEvictionRun"* value=*"1024"* />  <!-- 释放连接的扫描间隔（毫秒） -->  <property name=*"timeBetweenEvictionRunsMillis"* value=*"30000"* />  <!-- 连接最小空闲时间 -->  <property name=*"minEvictableIdleTimeMillis"* value=*"1800000"* />  <!-- 连接空闲多久后释放, 当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->  <property name=*"softMinEvictableIdleTimeMillis"* value=*"10000"* />  <!-- 获取连接时的最大等待毫秒数,小于零:阻塞不确定的时间,默认-1 -->  <property name=*"maxWaitMillis"* value=*"1500"* />  <!-- 在获取连接的时候检查有效性, 默认false -->  <property name=*"testOnBorrow"* value=*"true"* />  <!-- 在空闲时检查有效性, 默认false -->  <property name=*"testWhileIdle"* value=*"true"* />  <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->  <property name=*"blockWhenExhausted"* value=*"false"* />  </bean>  <!-- redis集群 -->  <bean id=*"jedisCluster"* class=*"redis.clients.jedis.JedisCluster"*>  <constructor-arg index=*"0"*>  <set>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg index=*"0"* value=*"192.168.101.3"*></constructor-arg>  <constructor-arg index=*"1"* value=*"7001"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg index=*"0"* value=*"192.168.101.3"*></constructor-arg>  <constructor-arg index=*"1"* value=*"7002"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg index=*"0"* value=*"192.168.101.3"*></constructor-arg>  <constructor-arg index=*"1"* value=*"7003"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg index=*"0"* value=*"192.168.101.3"*></constructor-arg>  <constructor-arg index=*"1"* value=*"7004"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg index=*"0"* value=*"192.168.101.3"*></constructor-arg>  <constructor-arg index=*"1"* value=*"7005"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg index=*"0"* value=*"192.168.101.3"*></constructor-arg>  <constructor-arg index=*"1"* value=*"7006"*></constructor-arg>  </bean>  </set>  </constructor-arg>  <constructor-arg index=*"1"* ref=*"jedisPoolConfig"*></constructor-arg>  </bean> |

* 测试代码

|  |
| --- |
| **private** ApplicationContext applicationContext;  @Before  **public** **void** init() {  applicationContext = **new** ClassPathXmlApplicationContext(  "classpath:applicationContext.xml");  }  // redis集群  @Test  **public** **void** testJedisCluster() {  JedisCluster jedisCluster = (JedisCluster) applicationContext  .getBean("jedisCluster");  jedisCluster.set("name", "zhangsan");  String value = jedisCluster.get("name");  System.*out*.println(value);  } |