

# Nosql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。

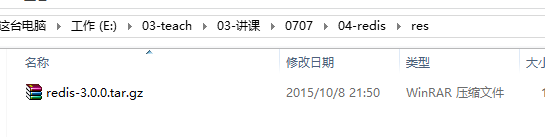
劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

# Redis安装配置

## Redis下载

官网地址：<http://redis.io/>

下载地址：[http://download.redis.io/releases/](http://download.redis.io/releases/redis-3.0.0.tar.gz)[redis-3.0.0.tar.gz](http://download.redis.io/releases/redis-3.0.0.tar.gz)



## R edis安装

Redis是C语言开发，建议在linux上运行，本教程使用Centos6.4作为安装环境。

第一步：在VMware中安装CentOS（参考Linux教程中的安装虚拟机）

第二步：在Linux下安装gcc环境（该步骤可以省略，CentOS中默认自带C环境）

|  |
| --- |
| [root@linux02 redis-3.0.0]# yum install gcc-c++ |

第三步：将下载的Redis源码包上传到Linux服务器中

【需要切换到sftp窗口】

|  |
| --- |
| sftp> put -r "E:\03-teach\03-讲课\0707\04-redis\res\redis-3.0.0.tar.gz" |

第四步：解压缩Redis源码包

|  |
| --- |
| [root@linux02 ~]# tar -zxf redis-3.0.0.tar.gz |

第五步：编译redis源码

|  |
| --- |
| [root@linux02 ~]# cd redis-3.0.0  [root@linux02 redis-3.0.0]# make |

第六步：安装redis

|  |
| --- |
| [root@linux02 redis-3.0.0]# make install PREFIX=/usr/local/redis0707 |

## Redis启动

### 前端启动

* 启动方式：

直接运行bin/redis-server将以前端模式启动。

|  |
| --- |
| [root@linux02 bin]# ./redis-server |

* 启动缺点：

ssh命令窗口关闭则redis-server程序结束，不推荐使用此方法

* 启动图例：



* 前端启动的关闭：ctrl+c

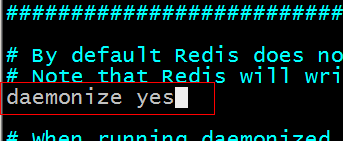
### 后端启动

第一步：将redis源码包中的redis.conf配置文件复制到/usr/local/redis/bin/下

|  |
| --- |
| [root@linux02 /]# cd /root/redis-3.0.0  [root@linux02 redis-3.0.0]# cp redis.conf /usr/local/redis0707/bin/ |

第二步：修改redis.conf，将daemonize由no改为**yes**

|  |
| --- |
| [root@redis01 bin2]# vim redis.conf |



第三步：执行命令

|  |
| --- |
| [root@linux02 bin]# ./redis-server redis.conf |

* 后端启动的关闭方式

非正常关闭（不推荐使用）：

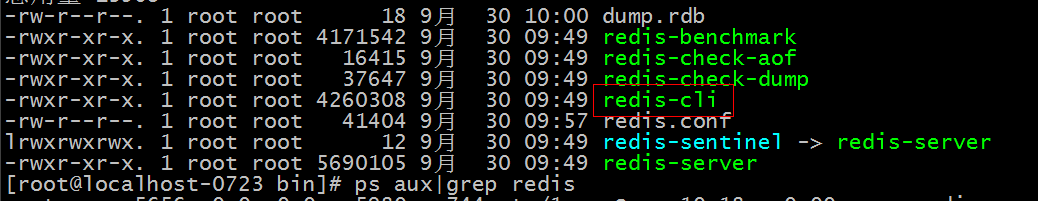
|  |
| --- |
| [root@localhost-0723 bin]# kill 5528 |

正常关闭：

|  |
| --- |
| [root@localhost-0723 bin]# ./redis-cli shutdown |

# Redis客户端

## Redis自带的客户端



* 指定主机和端口

|  |
| --- |
| [root@localhost-0723 bin]# ./redis-cli -h 127.0.0.1 -p 6379 |

-h：redis服务器的ip地址

-p：redis实例的端口号

* 如果不指定主机和端口也可以

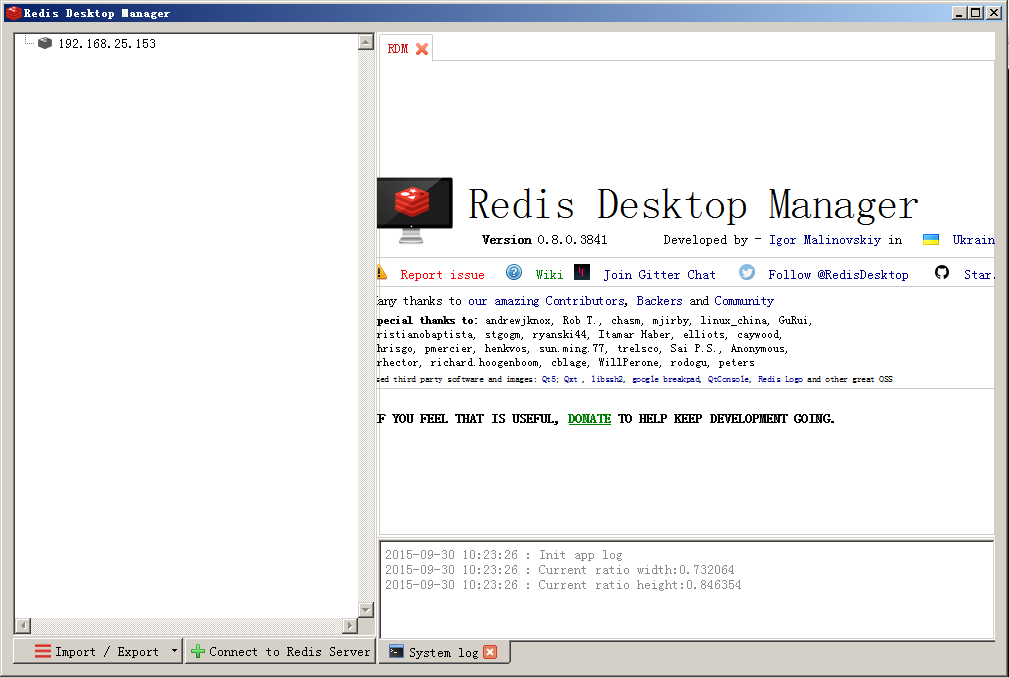
|  |
| --- |
| [root@localhost-0723 bin]# ./redis-cli |

默认主机地址是127.0.0.1

默认端口是6379

## 图形界面客户端（了解）

前提：需要安装图形界面管理器



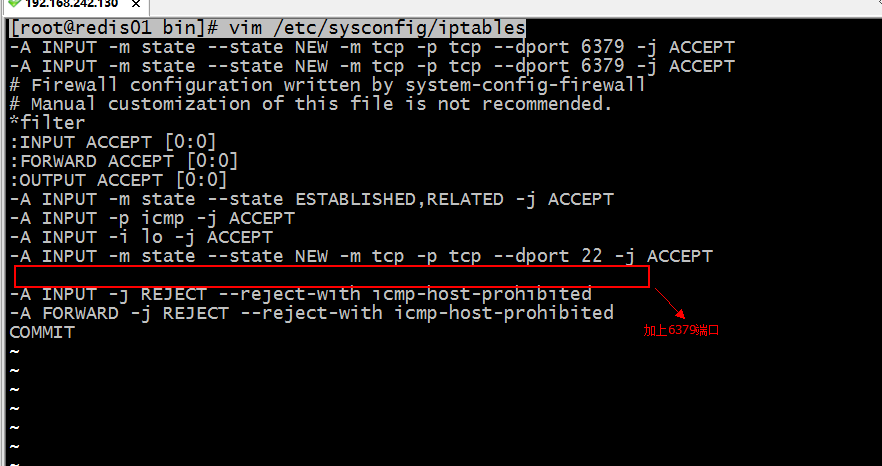
### 连接超时解决

远程连接redis服务，需要关闭或者修改防火墙配置。

第一步：编辑iptables

|  |
| --- |
| [root@redis01 bin]# vim /etc/sysconfig/iptables |

**在命令模式下，选定要复制的那一行的末尾，然后点击键盘yyp，就完成复制，然后修改**。

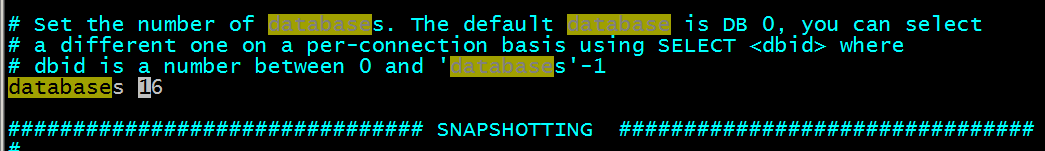


第二步：重启防火墙

|  |
| --- |
| [root@localhost-0723 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@localhost-0723 redis-3.0.0]# |

注意：

默认一共是16个数据库，每个数据库之间是相互隔离。数据库的数量是在redis.conf中配置的。



切换数据库使用命令：select 数据库编号

例如：select 1

## Java客户端Jedis

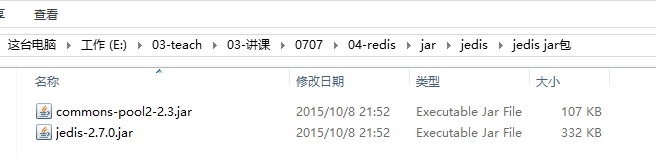
### jedis介绍

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>

### 添加jar包



### 单实例连接

|  |
| --- |
| @Test  **public** **void** testJedis() {  //创建一个Jedis的连接  Jedis jedis = **new** Jedis("127.0.0.1", 6379);  //执行redis命令  jedis.set("mytest", "hello world, this is jedis client!");  //从redis中取值  String result = jedis.get("mytest");  //打印结果  System.*out*.println(result);  //关闭连接  jedis.close();    } |

### 连接池连接

|  |
| --- |
| @Test  **public** **void** testJedisPool() {  //创建一连接池对象  JedisPool jedisPool = **new** JedisPool("127.0.0.1", 6379);  //从连接池中获得连接  Jedis jedis = jedisPool.getResource();  String result = jedis.get("mytest");  System.*out*.println(result);  //关闭连接  jedis.close();    //关闭连接池  jedisPool.close();  } |

### 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

Redis中存储数据是通过key-value存储的，对于value的类型有以下几种：

* 字符串
* Hash类型
* List
* Set
* SortedSet（zset）

PS：

在redis中的命令语句中，命令是忽略大小写的，而key是不忽略大小写的。

## String类型

### 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“* |

#### 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" |

#### 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" |

#### 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类型

### ArrayList与LinkedList的区别

ArrayList使用数组方式存储数据，所以根据索引**查询数据速度快**，而**新增或者删除元素时需要设计到位移操作，所以比较慢**。

LinkedList使用**双向链表方式**存储数据，每个元素都记录前后元素的指针，所以**插入、删除数据时**只是更改前后元素的指针指向即可，**速度非常快**。然后通过下标**查询元素时**需要从头开始索引，所以**比较慢**，但是**如果查询前几个元素或后几个元素速度比较快**。





### redis list介绍

列表类型（list）可以存储一个有序的字符串列表，常用的操作是向**列表两端添加元素**，或者**获得列表的某一个片段**。

**列表类型内部是使用双向链表（double linked list）实现**的，所以向列表两端添加元素的时间复杂度为0(1)，获取越接近两端的元素速度就越快。这意味着即使是一个有几千万个元素的列表，获取头部或尾部的10条记录也是极快的。

### 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"* |

#### 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类型

### redis set介绍

集合中的数据是不重复且没有顺序。

集合类型和列表类型的对比：



集合类型的常用操作是**向集合中加入或删除元素、判断某个元素是否存在**等，由于集合类型的Redis内部是使用值为空的散列表实现，所有这些操作的时间复杂度都为0(1)。

Redis还提供了多个集合之间的**交集、并集、差集**的运算。

### 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类型zset

### redis sorted set介绍

在集合类型的基础上，有序集合类型为集合中的每个元素都关联一个分数，这使得我们不仅可以完成插入、删除和判断元素是否存在在集合中，还能够获得分数最高或最低的前N个元素、获取指定分数范围内的元素等与分数有关的操作。

在某些方面有序集合和列表类型有些相似。

1、二者都是有序的。

2、二者都可以获得某一范围的元素。

但是，二者有着很大区别：

1、列表类型是通过链表实现的，获取靠近两端的数据速度极快，而当元素增多后，访问中间数据的速度会变慢。

2、有序集合类型使用散列表实现，所有即使读取位于中间部分的数据也很快。

3、列表中不能简单的调整某个元素的位置，但是有序集合可以（通过更改分数实现）

4、有序集合要比列表类型更耗内存。

### 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" |

#### 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 |

##### 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 |

### 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（了解）

## 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)* |

## 其它命令（自学）

### 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> |

# Redis持久化方案

## RDB持久化

RDB方式的持久化是**通过快照**（snapshotting）完成的，当**符合一定条件**时Redis会自动将内存中的数据**进行快照并持久化到硬盘**。

RDB是Redis默认采用的持久化方式。

|  |
| --- |
| save 900 1  save 300 10  save 60 10000 |

### 持久化条件配置

save 开头的一行就是持久化配置，可以配置多个条件（每行配置一个条件），每个条件之间是“或”的关系。

“save 900 1”表示15分钟（900秒钟）内至少1个键被更改则进行快照。

“save 300 10”表示5分钟（300秒）内至少10个键被更改则进行快照。

### 配置快照文件目录

配置dir指定rdb快照文件的位置

|  |
| --- |
| # Note that you must specify a directory here, not a file name.  dir ./ |

### 配置快照文件的名称

设置dbfilename指定rdb快照文件的名称

|  |
| --- |
| # The filename where to dump the DB  dbfilename dump.rdb |

Redis启动后会读取RDB快照文件，将数据从硬盘载入到内存。根据数据量大小与结构和服务器性能不同，这个时间也不同。通常将记录一千万个字符串类型键、大小为1GB的快照文件载入到内存中需要花费20～30秒钟。

### 问题总结

通过RDB方式实现持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据。这就需要开发者根据具体的应用场合，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受的范围。

**如果数据很重要以至于无法承受任何损失，则可以考虑使用AOF方式进行持久化**。

## AOF持久化

默认情况下Redis没有开启AOF（append only file）方式的持久化

* 可以通过修改redis.conf配置文件中的appendonly参数开启

|  |
| --- |
| appendonly yes |

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬

盘中的AOF文件。

* AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的。

|  |
| --- |
| dir ./ |

* 默认的文件名是appendonly.aof，可以通过appendfilename参数修改：

|  |
| --- |
| appendfilename appendonly.aof |

# 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)

## 主从配置

### 主redis配置

无需特殊配置。

### 从redis配置

* 修改从redis服务器上的redis.conf文件

|  |
| --- |
| # slaveof <masterip> <masterport>  slaveof 127.0.0.1 6379 |

上边的配置说明当前该【从redis服务器】所对应的【主redis服务器】的IP是192.168.101.3，端口是6379。

# 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：

13000

7000

6000

30

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。

## 搭建Ruby环境

redis集群管理工具redis-trib.rb依赖ruby环境，首先需要安装ruby环境。

* 安装ruby

|  |
| --- |
| [root@redis01 bin]# yum install ruby  [root@redis01 bin]# yum install rubygems |

* 使用sftp工具上传redis-3.0.0.gem至/usr/local下

|  |
| --- |
| sftp> put -r "E:\03-teach\03-讲课\0707\04-redis\res\ruby和redis接口\redis-3.0.0.gem" |

* 安装ruby和redis的接口程序

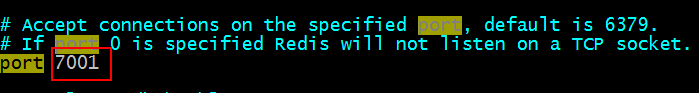
|  |
| --- |
| [root@linux02 local]# gem install /usr/local/redis-3.0.0.gem |

* 将Redis集群搭建脚本文件复制到/usr/local/redis0707目录下

|  |
| --- |
| [root@redis01 /]# cd /root/redis-3.0.0/src/  [root@redis01 src]# ll \*.rb  -rwxrwxr-x. 1 root root 48141 4月 1 2015 redis-trib.rb  [root@redis01 src]# cp redis-trib.rb /usr/local/redis0707/ -r |

## 集群的搭建过程

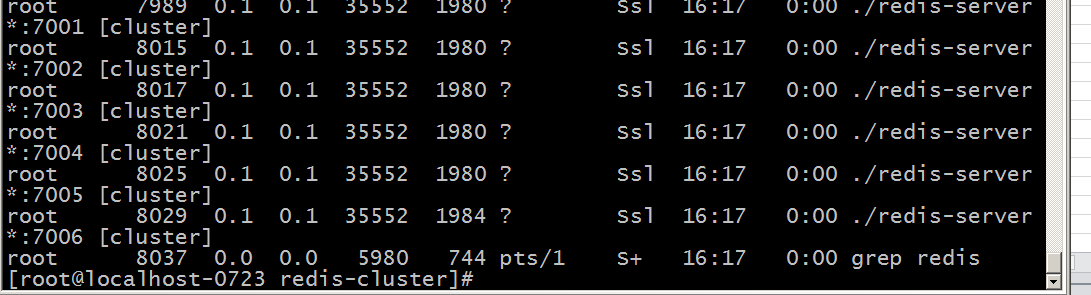
第一步：创建6个redis实例，需要端口号7001~7006



第二步：修改redis.conf配置文件，打开Cluster-enable yes



第三步：启动所有的实例



第四步：创建集群

|  |
| --- |
| [root@localhost-0723 redis]# ./redis-trib.rb create --replicas 1 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006  >>> Creating cluster  Connecting to node 127.0.0.1:7001: OK  Connecting to node 127.0.0.1:7002: OK  Connecting to node 127.0.0.1:7003: OK  Connecting to node 127.0.0.1:7004: OK  Connecting to node 127.0.0.1:7005: OK  Connecting to node 127.0.0.1:7006: OK  >>> Performing hash slots allocation on 6 nodes...  Using 3 masters:  127.0.0.1:7001  127.0.0.1:7002  127.0.0.1:7003  Adding replica 127.0.0.1:7004 to 127.0.0.1:7001  Adding replica 127.0.0.1:7005 to 127.0.0.1:7002  Adding replica 127.0.0.1:7006 to 127.0.0.1:7003  M: d8f6a0e3192c905f0aad411946f3ef9305350420 127.0.0.1:7001  slots:0-5460 (5461 slots) master  M: 7a12bc730ddc939c84a156f276c446c28acf798c 127.0.0.1:7002  slots:5461-10922 (5462 slots) master  M: 93f73d2424a796657948c660928b71edd3db881f 127.0.0.1:7003  slots:10923-16383 (5461 slots) master  S: f79802d3da6b58ef6f9f30c903db7b2f79664e61 127.0.0.1:7004  replicates d8f6a0e3192c905f0aad411946f3ef9305350420  S: 0bc78702413eb88eb6d7982833a6e040c6af05be 127.0.0.1:7005  replicates 7a12bc730ddc939c84a156f276c446c28acf798c  S: 4170a68ba6b7757e914056e2857bb84c5e10950e 127.0.0.1:7006  replicates 93f73d2424a796657948c660928b71edd3db881f  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 127.0.0.1:7001)  M: d8f6a0e3192c905f0aad411946f3ef9305350420 127.0.0.1:7001  slots:0-5460 (5461 slots) master  M: 7a12bc730ddc939c84a156f276c446c28acf798c 127.0.0.1:7002  slots:5461-10922 (5462 slots) master  M: 93f73d2424a796657948c660928b71edd3db881f 127.0.0.1:7003  slots:10923-16383 (5461 slots) master  M: f79802d3da6b58ef6f9f30c903db7b2f79664e61 127.0.0.1:7004  slots: (0 slots) master  replicates d8f6a0e3192c905f0aad411946f3ef9305350420  M: 0bc78702413eb88eb6d7982833a6e040c6af05be 127.0.0.1:7005  slots: (0 slots) master  replicates 7a12bc730ddc939c84a156f276c446c28acf798c  M: 4170a68ba6b7757e914056e2857bb84c5e10950e 127.0.0.1:7006  slots: (0 slots) master  replicates 93f73d2424a796657948c660928b71edd3db881f  [OK] All nodes agree about slots configuration.  >>> Check for open slots...  >>> Check slots coverage...  [OK] All 16384 slots covered.  [root@localhost-0723 redis]# |

## 连接集群

命令：./redis-cli –h 127.0.0.1 –p 7001 **-c**

|  |
| --- |
| [root@localhost-0723 redis]# ./redis-cli -p 7006 -c  127.0.0.1:7006> set key1 123  -> Redirected to slot [9189] located at 127.0.0.1:7002  OK  127.0.0.1:7002> |

## 查看集群的命令

* 查看集群状态

|  |
| --- |
| 127.0.0.1:7003> **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:3  cluster\_stats\_messages\_sent:926  cluster\_stats\_messages\_received:926 |

* 查看集群中的节点：

|  |
| --- |
| 127.0.0.1:7003> **cluster nodes**  7a12bc730ddc939c84a156f276c446c28acf798c 127.0.0.1:7002 master - 0 1443601739754 2 connected 5461-10922  93f73d2424a796657948c660928b71edd3db881f 127.0.0.1:7003 myself,master - 0 0 3 connected 10923-16383  d8f6a0e3192c905f0aad411946f3ef9305350420 127.0.0.1:7001 master - 0 1443601741267 1 connected 0-5460  4170a68ba6b7757e914056e2857bb84c5e10950e 127.0.0.1:7006 slave 93f73d2424a796657948c660928b71edd3db881f 0 1443601739250 6 connected  f79802d3da6b58ef6f9f30c903db7b2f79664e61 127.0.0.1:7004 slave d8f6a0e3192c905f0aad411946f3ef9305350420 0 1443601742277 4 connected  0bc78702413eb88eb6d7982833a6e040c6af05be 127.0.0.1:7005 slave 7a12bc730ddc939c84a156f276c446c28acf798c 0 1443601740259 5 connected  127.0.0.1:7003> |

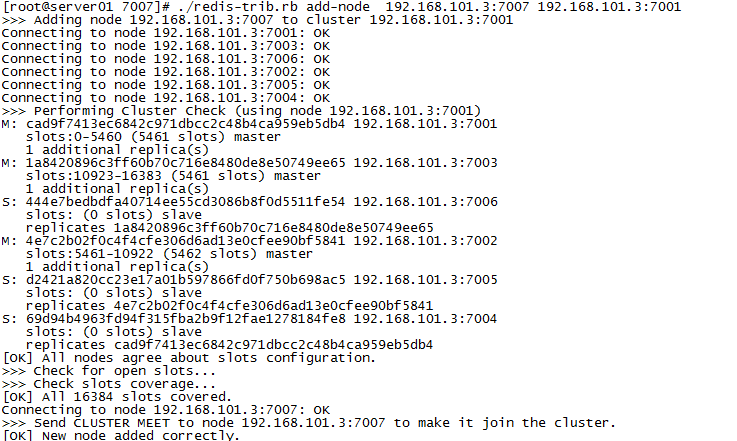
## 维护节点（自学）

### 添加主节点

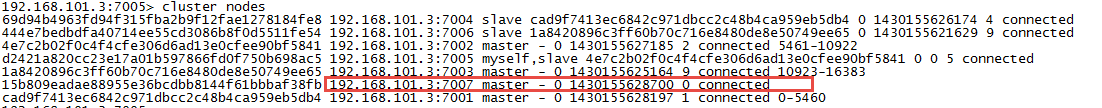
集群创建成功后可以向集群中添加节点，下面是添加一个master主节点

* **添加7007结点作为新节点**

**执行命令：./redis-trib.rb add-node 127.0.0.1:7007 127.0.0.1:7001**



* 查看集群结点发现7007已添加到集群中

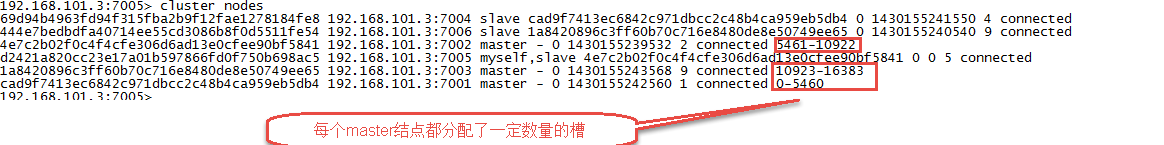


### hash槽重新分配

**添加完主节点需要对主节点进行hash槽分配，这样该主节才可以存储数据。**

* 查看集群中槽占用情况

redis集群有16384个槽，集群中的每个结点分配自已槽，通过查看集群结点可以看到槽占用情况。

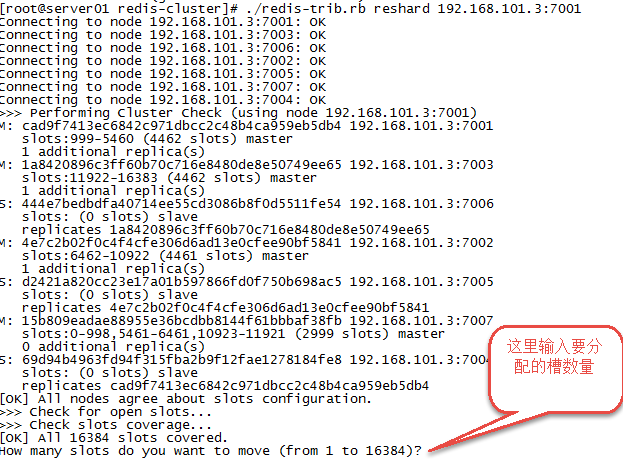


* 给刚添加的7007结点分配槽

**第一步**：连接上集群（连接集群中任意一个可用结点都行）

|  |
| --- |
| [root@redis01 redis0707]# ./redis-trib.rb reshard 192.168.101.3:7001 |

**第二步**：输入要分配的槽数量



输入：500，表示要分配500个槽

**第三步**：输入接收槽的结点id



输入：15b809eadae88955e36bcdbb8144f61bbbaf38fb

PS：这里准备给7007分配槽，通过cluster nodes查看7007结点id为：

15b809eadae88955e36bcdbb8144f61bbbaf38fb

**第四步**：输入源结点id



输入：all

**第五步**：输入yes开始移动槽到目标结点id



输入：yes

### 添加从节点

集群创建成功后可以向集群中添加节点，下面是添加一个slave从节点。

* 添加7008从结点，将7008作为7007的从结点

**命令：./redis-trib.rb add-node --slave --master-id 主节点id 新节点的ip和端口 旧节点ip和端口**

执行如下命令：

|  |
| --- |
| ./redis-trib.rb add-node --slave --master-id cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7008 192.168.101.3:7001 |

cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 是7007结点的id，可通过cluster nodes查看。



注意：如果原来该结点在集群中的配置信息已经生成到cluster-config-file指定的配置文件中（如果cluster-config-file没有指定则默认为nodes.conf），这时可能会报错：

|  |
| --- |
| [ERR] Node XXXXXX is not empty. Either the node already knows other nodes (check with CLUSTER NODES) or contains some key in database 0 |

解决方法是删除生成的配置文件nodes.conf，删除后再执行**./redis-trib.rb add-node**指令

* 查看集群中的结点，刚添加的7008为7007的从节点：



### 删除结点

命令：./redis-trib.rb del-node 127.0.0.1:7005 4b45eb75c8b428fbd77ab979b85080146a9bc017

删除已经占有hash槽的结点会失败，报错如下：

[ERR] Node 127.0.0.1:7005 is not empty! Reshard data away and try again.

需要将该结点占用的hash槽分配出去（参考hash槽重新分配章节）。

# Jedis连接集群

## 防火墙配置

|  |
| --- |
| [root@localhost-0723 bin]# service iptables stop  -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 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 6380 -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 -j REJECT --reject-with icmp-host-prohibited  -A FORWARD -j REJECT --reject-with icmp-host-prohibited  COMMIT  ~  "/etc/sysconfig/iptables" 22L, 1079C 已写入  [root@localhost-0723 bin]# service iptables restart  iptables：应用防火墙规则： [确定]  [root@localhost-0723 bin]# |

## 代码实现

创建JedisCluster类连接redis集群。

|  |
| --- |
| @Test  **public** **void** testJedisCluster() **throws** Exception {  //创建一连接，JedisCluster对象,在系统中是单例存在  Set<HostAndPort> nodes = **new** HashSet<>();  nodes.add(**new** HostAndPort("127.0.0.1", 7001));  nodes.add(**new** HostAndPort("127.0.0.1", 7002));  nodes.add(**new** HostAndPort("127.0.0.1", 7003));  nodes.add(**new** HostAndPort("127.0.0.1", 7004));  nodes.add(**new** HostAndPort("127.0.0.1", 7005));  nodes.add(**new** HostAndPort("127.0.0.1", 7006));  JedisCluster cluster = **new** JedisCluster(nodes);  //执行JedisCluster对象中的方法，方法和redis一一对应。  cluster.set("cluster-test", "my jedis cluster test");  String result = cluster.get("cluster-test");  System.*out*.println(result);  //程序结束时需要关闭JedisCluster对象  cluster.close();  } |

## 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);  } |