

# Redis介绍

## 什么是Redis？

* **Redis**是用**C语言**开发的一个**开源**的高性能**键值对**（key-value）**内存数据库**。
* 它提供**五种数据类型**来存储值：字符串类型、散列类型、列表类型、集合类型、有序集合类型
* 它是一种**NoSQL**数据库。

## 什么是NoSQL？

* NoSQL，即Not-Only SQL（不仅仅是SQL），泛指**非关系型的数据库**。
* 什么是**关系型**数据库？**数据结构是一种有行有列的数据库**
* NoSQL数据库是为了解决**高并发、高可用、高可扩展、大数据存储**问题而产生的数据库解决方案。
* NoSQL可以作为关系型数据库的良好补充，但是**不能替代关系型数据库**。

## NoSQL数据库分类

* **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：

* 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

* **内存数据库**（登录信息、购物车信息、用户浏览记录等）
* **缓存服务器**（商品数据、广告数据等等）。（最多使用）
* 解决分布式集群架构中的session分离问题（session共享）。
* 任务队列。（秒杀、抢购、12306等等）
* 支持发布订阅的消息模式
* 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单机版安装配置

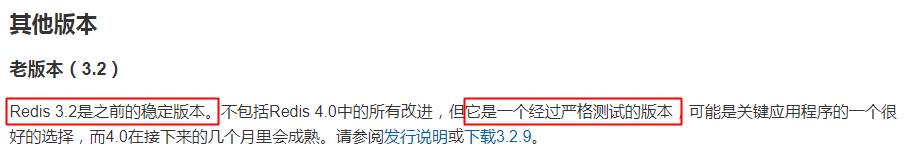
## Redis下载

\* 官网地址：<http://redis.io/>

\* 中文官网地址：<http://www.redis.cn/>

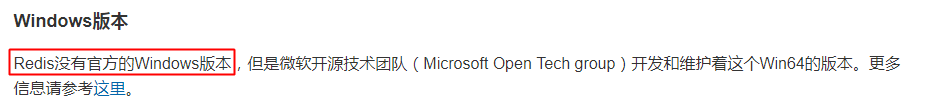
\* 下载地址：[http://download.redis.io/releases/](http://download.redis.io/releases/redis-3.0.0.tar.gz)





## Redis安装环境

Redis没有官方的Windows版本，所以建议在Linux系统上安装运行，本教程使用CentOS6.5（Linux操作系统的一个系列）作为安装环境。



## Redis安装

* 第一步：在VMware中安装CentOS（参考Linux教程中的安装虚拟机）
* **第二步：安装C语言需要的GCC环境**

|  |
| --- |
| yum install gcc-c++ |

* **第三步：解压缩Redis源码压缩包**

|  |
| --- |
| tar -zxf redis-3.2.9.tar.gz |

* **第四步：编译Redis源码**，进入redis-3.2.9目录，执行编译命令

|  |
| --- |
| make |

* **第五步：安装Redis**，需要通过PREFIX指定安装路径

|  |
| --- |
| make install PREFIX=/usr/local/redis |

## Redis启动

### 前端启动

* 启动命令：**redis-server**，直接运行bin/redis-server将以前端模式启动。

|  |
| --- |
| ./redis-server |

* 关闭命令：ctrl+c

启动缺点：客户端窗口关闭则redis-server程序结束，不推荐使用此方法

启动图例：



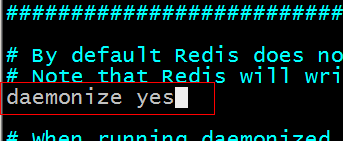
### 后端启动（守护进程启动）

* 第一步：拷贝redis-3.2.9/redis.conf配置文件到Redis安装目录的bin目录

|  |
| --- |
| cp redis.conf /usr/local/redis/bin/ |

* 第二步：修改redis.conf，将daemonize由no改为yes

|  |
| --- |
| vim redis.conf |



修改redis.conf中的61行代码

bind 127.0.0.1

改为

bind <redis实例所在机器的真实IP>，比如192.168.10.133

* 第三步：执行命令

|  |
| --- |
| ./redis-server redis.conf |

### 后端启动的关闭方式

* 非正常关闭（不推荐使用）：

|  |
| --- |
| kill 5528 |

* 正常关闭：

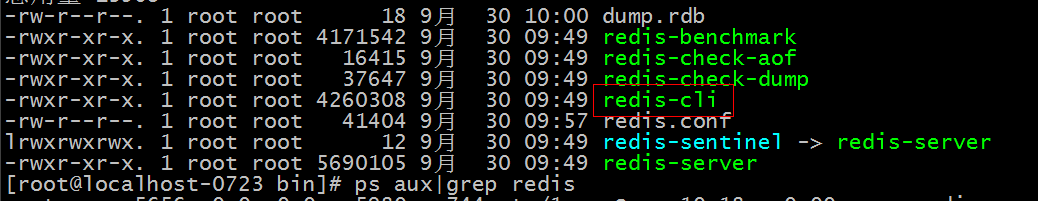
|  |
| --- |
| **./redis-cli shutdown** |

## 其他命令说明

* **redis-server ：启动redis服务**
* **redis-cli ：进入redis命令客户端**
* redis-benchmark： 性能测试的工具
* redis-check-aof ： aof文件进行检查的工具
* redis-check-dump ： rdb文件进行检查的工具
* redis-sentinel ： 启动哨兵监控服务

# Redis客户端

## 自带命令行客户端



* **命令格式：**

|  |
| --- |
| ./**redis-cli** -h 127.0.0.1 -p 6379 |

* **修改redis配置文件（解决IP绑定问题）**

**# bind 127.0.0.1 绑定的IP才能访问redis服务器，注释掉该配置**

**protected-mode yes 是否开启保护模式，由yes该为no**

* **参数说明：**

-h：redis服务器的ip地址

-p：redis实例的端口号

* **默认方式**

**如果不指定主机和端口也可以**

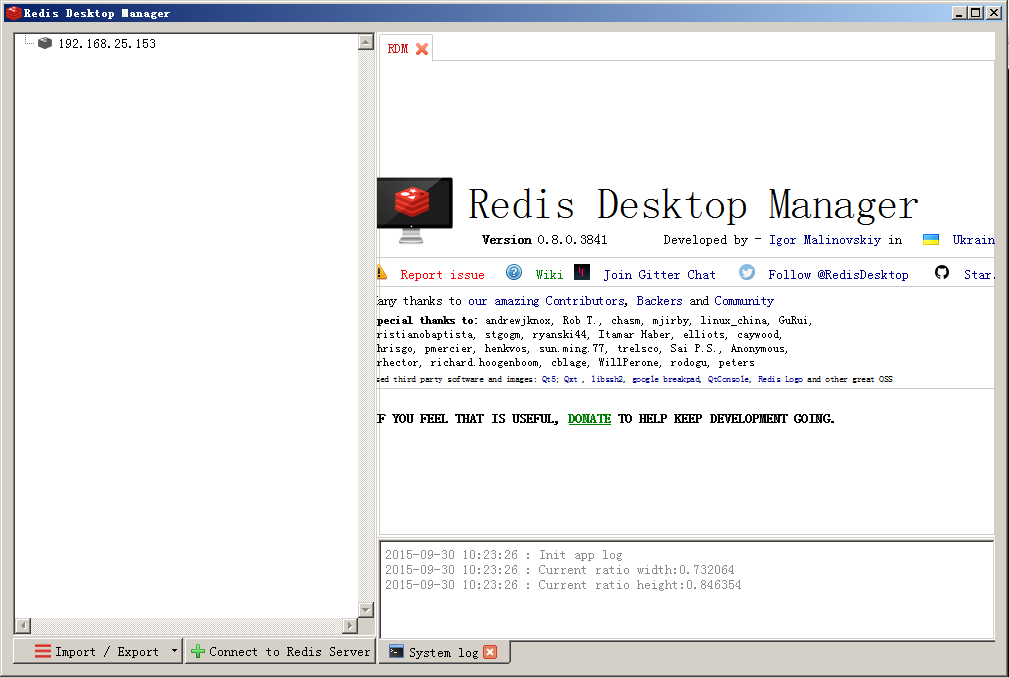
|  |
| --- |
| ./redis-cli |

\* 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

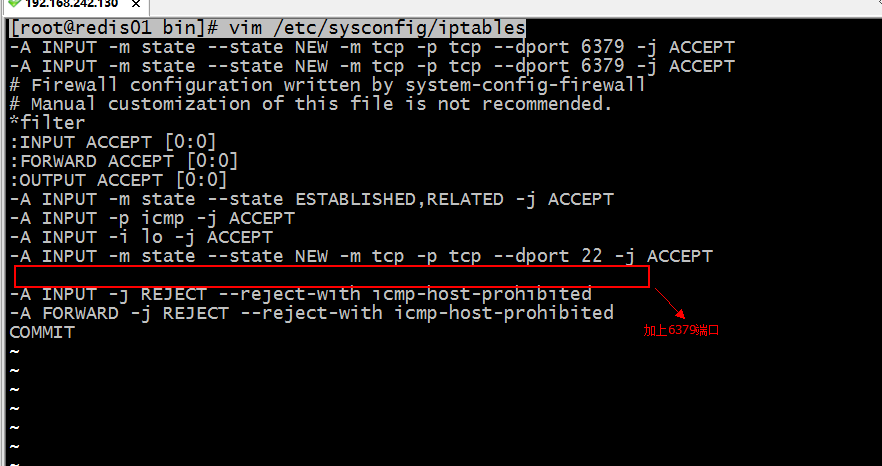
远程连接redis服务，需要关闭或者修改防火墙配置。

**修改防火墙设置：**

* 第一步：编辑iptables

|  |
| --- |
| 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。

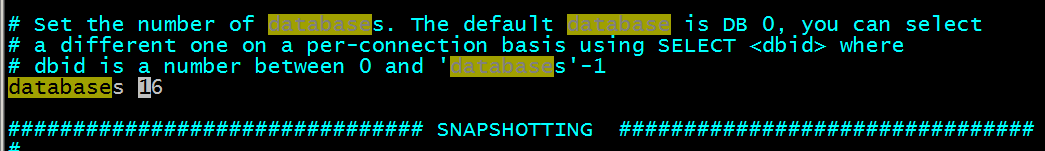


* 第二步：重启防火墙

|  |
| --- |
| **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] |

### 多数据库支持

* 默认一共是**16**个数据库，每个数据库之间是**相互隔离（但是可以使用flushall一次清空所有的库）**。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（0-15）

例如：select 1

## 程序客户端之Java客户端Jedis

### Jedis介绍

* Redis不仅使用命令客户端来操作，而且可以使用**程序客户端**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

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency> |

### 单实例连接

注意事项：需要去设置redis服务器的防火墙策略（临时关闭、永久关闭、端口暴露）

|  |
| --- |
| @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("key1", "hello world");  //从redis中取值  String result = jedis.get("key1");  //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("key1") ;  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.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx.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

官方命令大全网址：<http://www.redis.cn/commands.html>

Redis中存储数据是通过key-value格式存储数据的，其中value可以定义五种数据类型：

* **String（字符类型）**
* **Hash（散列类型）**
* **List（列表类型）**
* **Set（集合类型）**
* **SortedSet（有序集合类型，简称zset）**

注意：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“ |

#### 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" |

#### 数值增减

**注意实现：**

1. 当value为整数数据时，才能使用以下命令操作数值的增减。
2. 数值递增都是**原子**操作。

|  |
| --- |
| 非原子性操作示例：  int i = 1;  i++;  System.out.println(i) |

* 递增数字

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 |

#### 仅当不存在时赋值

**使用该命令可以实现分布式锁的功能，后续讲解！！！**

**语法：setnx key value**

redis> EXISTS job # job 不存在

(integer) 0

redis> SETNX job "programmer" # job 设置成功

(integer) 1

redis> SETNX job "code-farmer" # 尝试覆盖 job ，失败

(integer) 0

redis> GET job # 没有被覆盖

"programmer"

#### 其它命令

##### 向尾部追加值

APPEND命令，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 |

##### 同时设置/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" |

### 应用场景之自增主键

* **需求：商品编号、订单号采用INCR命令生成。**
* **设计：**key命名要有一定的设计
* **实现：**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

### 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 |

##### 获取所有字段

**作用：获得hash的所有信息，包括key和value**

**语法：hgetall key**

### 应用之存储商品信息

**注意事项：存储那些对象数据，特别是对象属性经常发生增删改操作的数据。**

* 商品信息字段

【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ID为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。





### list介绍

Redis的列表类型（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

**语法：LRANGE key start stop**

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。**

|  |
| --- |
| 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)

##### 删除列表中指定个数的值

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" |

### 应用之商品评论列表

* 需求1：用户针对某一商品发布评论，一个商品会被不同的用户进行评论，存储商品评论时，要按时间顺序排序。
* 需求2：用户在前端页面查询该商品的评论，需要安装时间顺序降序排序。

思路：

使用list存储商品评论信息，KEY是该商品的ID，VALUE是商品评论信息列表

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

### set介绍

set类型即**集合类型**，其中的数据是**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 |

### 集合运算命令

#### 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“ |

## Zset类型 （sortedset）

### Zset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 |

#### 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 |

### 应用之商品销售排行榜

* 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 |

## 通用命令

### keys

返回满足给定pattern 的所有key

语法：**keys pattern**

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

### del

**语法：DEL key**

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

### exists

**作用：确认一个key 是否存在**

**语法：exists key**

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> |

### expire

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) |

### rename

**作用：重命名key**

**语法：rename oldkey newkey**

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

**作用：显示指定key的数据类型**

**语法：type key**

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事务

## Redis事务介绍

* Redis的事务是通过**MULTI，EXEC，DISCARD和WATCH**这四个命令来完成的。
* Redis的单个命令都是**原子性**的，所以这里确保事务性的对象是**命令集合**。
* Redis将命令集合序列化并确保处于同一事务的**命令集合连续且不被打断**的执行
* Redis不支持回滚操作

## 相关命令

* **MULTI**

**用于标记事务块的开始**。

Redis会将后续的命令逐个放入队列中，然后使用EXEC命令原子化地执行这个命令序列。

**语法：multi**

* **EXEC**

**在一个事务中执行所有先前放入队列的命令**，然后恢复正常的连接状态

**语法：exec**

* **DISCARD**

**清除所有先前在一个事务中放入队列的命令**，然后恢复正常的连接状态。

**语法：discard**

* **WATCH**

当某个**事务需要按条件执行**时，就要使用这个命令将给定的**键设置为受监控**的状态。

**语法：watch key [key…]**

**注意事项：**使用该命令可以实现redis的**乐观锁**。

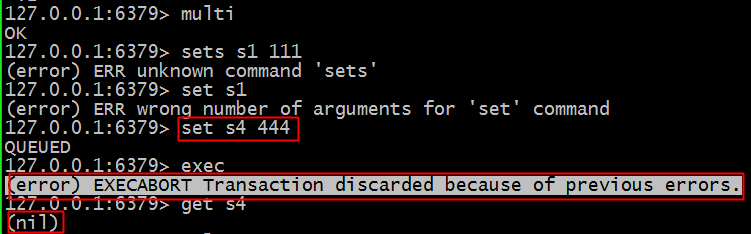
* **UNWATCH**

清除所有先前为一个事务监控的键。

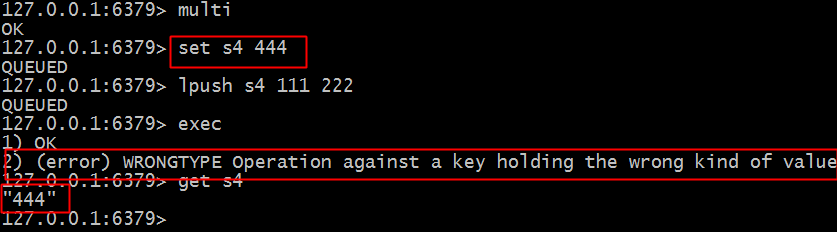
**语法：unwatch**

## 事务失败处理

* Redis语法错误（可以理解为编译期错误）



* Redis类型错误（可以理解为运行期错误）



* **Redis不支持事务回滚**

**为什么redis不支持事务回滚？**

1. 大多数事务失败是因为**语法错误或者类型错误**，这两种错误，在开发阶段都是可以预见的
2. redis为了**性能方面**就忽略了事务回滚

# Redis实现分布式锁

## 锁的处理

* 单应用中使用锁：单进程多线程

**synchronize、Lock**

* 分布式应用中使用锁：多进程

## 分布式锁的实现方式

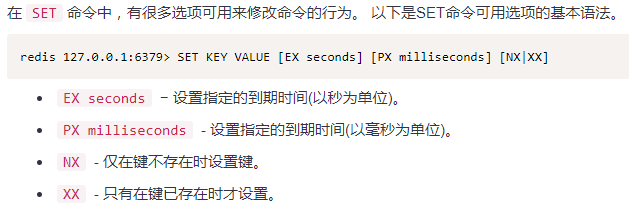
* 基于数据库的乐观锁实现分布式锁
* 基于zookeeper临时节点的分布式锁
* **基于redis的分布式锁**

## 分布式锁的注意事项

* **互斥性：**在任意时刻，只有一个客户端能持有锁
* **同一性：**加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。
* **可重入性：**即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。

## 实现分布式锁

### 获取锁



* 方式1（**使用set命令实现**） --推荐：

|  |
| --- |
| /\*\*  \* 使用redis的set命令实现获取分布式锁  \* **@param** lockKey 可以就是锁  \* **@param** requestId 请求ID，保证同一性  \* **@param** expireTime 过期时间，避免死锁  \* **@return**  \*/  **public** **static** **boolean** getLock(String lockKey,String requestId,**int** expireTime) {  //NX:保证互斥性  String result = *jedis*.set(lockKey, requestId, "NX", "EX", expireTime);  **if**("OK".equals(result)) {  **return** **true**;  }    **return** **false**;  } |

* 方式2（**使用setnx命令实现**）：

|  |
| --- |
| **public** **static** **boolean** getLock(String lockKey,String requestId,**int** expireTime) {  Long result = *jedis*.setnx(lockKey, requestId);  **if**(result == 1) {  *jedis*.expire(lockKey, expireTime);  **return** **true**;  }    **return** **false**;  } |

### 释放锁

* 方式1（del命令实现）：

|  |
| --- |
| /\*\*  \* 释放分布式锁  \* **@param** lockKey  \* **@param** requestId  \*/  **public** **static** **void** releaseLock(String lockKey,String requestId) {  **if** (requestId.equals(*jedis*.get(lockKey))) {  *jedis*.del(lockKey);  }  } |

* 方式2（**redis+lua脚本实现**）--推荐：

|  |
| --- |
| **public** **static** **boolean** releaseLock(String lockKey, String requestId) {  String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";  Object result = *jedis*.eval(script, Collections.*singletonList*(lockKey), Collections.*singletonList*(requestId));  **if** (result.equals(1L)) {  **return** **true**;  }  **return** **false**;  } |

# Redis持久化方案

Redis是一个**内存**数据库，为了保证数据的持久性，它提供了两种持久化方案:

* RDB方式（默认）
* AOF方式

## RDB方式

### 介绍

* RDB是Redis**默认**采用的持久化方式。
* RDB方式是通过**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。
* Redis会在**指定的情况**下触发快照

1. **符合自定义配置的快照规则**
2. **执行save或者bgsave命令**
3. **执行flushall命令**
4. **执行主从复制操作**

* **在redis.conf中设置自定义快照规则**

1. **RDB持久化条件**

格式：*save <seconds> <changes>*

**示例：**

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。

save 60 10000 ：表示1分钟内至少10000个键被更改则进行快照。

**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  save 300 10  save 60 10000 |

1. **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 ./ |

1. **配置dbfilename指定rdb快照文件的名称**

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

* **特别说明：**

\* 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。

### 快照的实现原理

* **快照过程**

1. redis使用fork函数**复制**一份当前进程的**副本**(子进程)
2. **父进程**继续接收并处理客户端发来的命令，而**子进程**开始将内存中的数据写入硬盘中的临时文件。
3. 当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此，一次快照操作完成。

* **注意事项**

1. redis在进行快照的过程中不会修改RDB文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候RDB文件都是完整的。
2. 这就使得我们可以通过定时备份RDB文件来实现redis数据库的备份， **RDB文件是经过压缩的二进制文件**，占用的空间会小于内存中的数据，更加利于传输。

### RDB优缺点

* **缺点：**使用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**。这个时候我们就需要根据具体的应用场景，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范围。如果数据相对来说比较重要，希望将损失降到最小，则可以使用**AOF**方式进行持久化
* **优点：** RDB可以最大化Redis的性能：父进程在保存RDB文件时唯一要做的就是fork出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无序执行任何磁盘I/O操作。同时这个也是一个缺点，如果数据集比较大的时候，fork可以能比较耗时，造成服务器在一段时间内停止处理客户端的请求；

## AOF方式

### 介绍

* 默认情况下Redis没有开启AOF（append only file）方式的持久化
* 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，这一过程显然**会降低Redis的性能**，但大部分情况下这个影响是能够接受的，另外使**用较快的硬盘可以提高AOF的性能**。
* 可以通过修改redis.conf配置文件中的appendonly参数开启

|  |
| --- |
| appendonly yes |

* 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** |

### AOF重写原理（优化AOF文件）

* Redis 可以在 AOF **文件体积变得过大**时，自动地在后台对 AOF **进行重写**
* 重写后的新 AOF 文件包含了恢复当前数据集所需的**最小命令集合**。
* 整个重写操作是绝对安全的，因为 Redis 在创建**新 AOF 文件**的过程中，会继续将命令追加到**现有的 AOF 文件**里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦**新 AOF 文件**创建完毕，Redis 就会从**旧 AOF 文件**切换到**新 AOF 文件**，并开始对**新 AOF 文件**进行追加操作。
* AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松
* **参数说明**

# **auto-aof-rewrite-percentage 100** 表示当前aof文件大小超过上一次aof文件大小的百分之多少的时候会进行重写。如果之前没有重写过，以启动时aof文件大小为准

# **auto-aof-rewrite-min-size 64mb** 限制允许重写最小aof文件大小，也就是文件大小小于64mb的时候，不需要进行优化

### 同步磁盘数据

Redis每次更改数据的时候， aof机制都会将命令记录到aof文件，但是实际上由于操作系统的缓存机制，数据并没有实时写入到硬盘，而是进入硬盘缓存。再通过硬盘缓存机制去刷新到保存到文件

* **参数说明：**

\* **appendfsync always** 每次执行写入都会进行同步 ， 这个是最安全但是是效率比较低的方式

**\* appendfsync everysec** 每一秒执行

\* **appendfsync no** 不主动进行同步操作，由操作系统去执行，这个是最快但是最不安全的方式

### AOF文件损坏以后如何修复

服务器可能在程序正在对 AOF 文件进行写入时停机， 如果停机造成了 AOF 文件出错（corrupt）， 那么 Redis 在重启时会拒绝载入这个 AOF 文件， 从而确保数据的一致性不会被破坏。

**当发生这种情况时， 可以用以下方法来修复出错的 AOF 文件：**

1. 为现有的 AOF 文件创建一个**备份**。
2. 使用 Redis 附带的 **redis-check-aof** 程序，对原来的 AOF 文件进行修复。

redis-check-aof --fix readonly.aof

1. 重启 Redis 服务器，等待服务器载入修复后的 AOF 文件，并进行数据恢复。

## 如何选择RDB和AOF

* 一般来说,如果对数据的安全性要求非常高的话，应该同时使用两种持久化功能。
* 如果可以承受数分钟以内的数据丢失，那么可以只使用 RDB 持久化。
* 有很多用户都只使用 AOF 持久化， 但并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快 。
* 两种持久化策略可以同时使用，也可以使用其中一种。如果同时使用的话， 那么Redis重启时，会优先**使用AOF文件来**还原数据

# 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，不过通过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

* 修改从服务器上的redis.conf文件

|  |
| --- |
| # slaveof <masterip> <masterport>  **slaveof 192.168.101.3 6379** |

上边的配置说明当前【从服务器】对应的【主服务器】的IP是192.168.101.3，端口是6379。

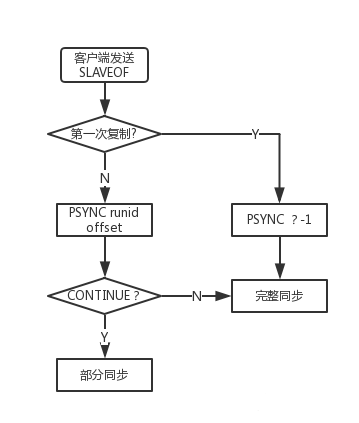
## 实现原理

\* Redis的主从同步，分为全量同步和增量同步。

\* 只有从机第一次连接上主机是全量同步

\* 断线重连有可能触发全量同步也有可能是增量同步（master判断runid是否一致）

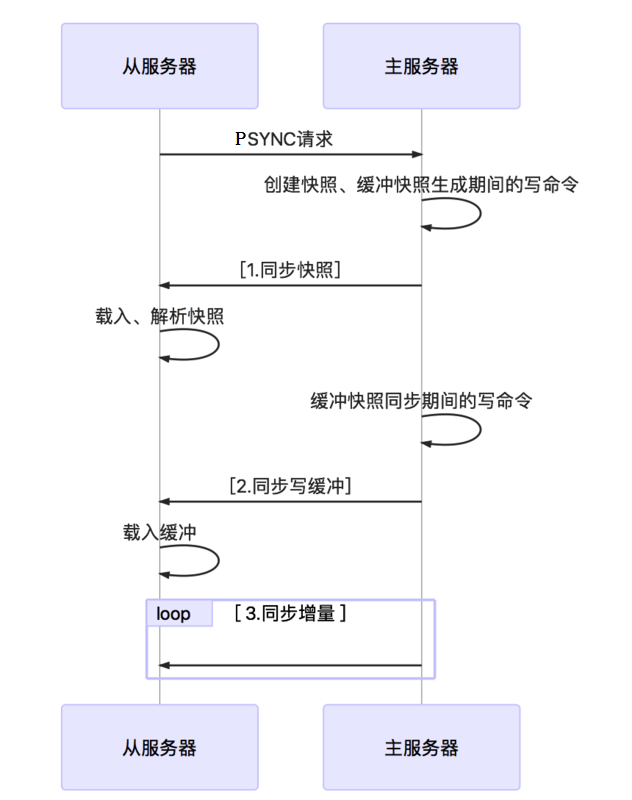
\* 除此之外的情况都是增量同步



### 全量同步

**Redis的全量同步过程主要分三个阶段：**

* 同步快照阶段：Master创建并发送**快照**给Slave,Slave载入并解析快照。Master同时将此阶段所产生的新的写命令存储到缓冲区。
* 同步写缓冲阶段：Master向Slave同步存储在缓冲区的写操作命令。
* 同步增量阶段：Master向Slave同步写操作命令。



### 增量同步

\* Redis增量同步主要指**Slave完成初始化后开始正常工作**时，**Master发生的写操作同步到Slave的过程**。

\* 通常情况下，Master每执行一个写命令就会向Slave发送相同的**写命令**，然后Slave接收并执行。

# Redis Sentinel哨兵机制

**Redis主从复制的缺点：没有办法对master进行动态选举，需要使用Sentinel机制完成动态选举**

## 简介

\* Sentinel(哨兵)进程是用于**监控redis集群中Master主服务器工作的状态**

\* 在Master主服务器发生故障的时候，可以实现Master和Slave服务器的切换，保证系统的高可用（HA）

\* 其已经被集成在redis2.6+的版本中，Redis的哨兵模式到了2.8版本之后就稳定了下来。

## 哨兵进程的作用

1. **监控(Monitoring):** 哨兵(sentinel) 会不断地检查你的Master和Slave是否运作正常。
2. **提醒(Notification)**： 当被监控的某个Redis节点出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。
3. **自动故障迁移(Automatic failover)**：当**一个Master不能正常工作**时，哨兵(sentinel) 会开始**一次自动故障迁移操作**。

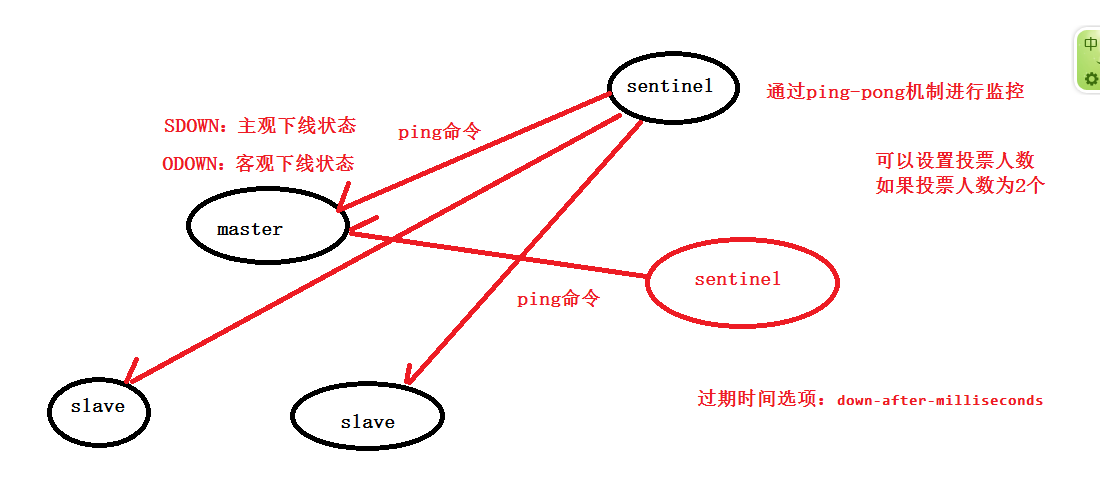
\* 它会将失效Master的其中一个Slave升级为新的Master, 并让失效Master的其他Slave改为复制新的Master；

\* 当客户端试图连接失效的Master时，集群也会向客户端返回新Master的地址，使得集群可以使用现在的Master替换失效Master。

\* Master和Slave服务器切换后，Master的redis.conf、Slave的redis.conf和sentinel.conf的配置文件的内容都会发生相应的改变，即，Master主服务器的redis.conf配置文件中会多一行slaveof的配置，sentinel.conf的监控目标会随之调换。

## 哨兵进程的工作方式

1. 每个Sentinel（哨兵）进程以**每秒钟一次**的频率向整个集群中的**Master主服务器，Slave从服务器以及其他Sentinel（哨兵）进程**发送一个 PING 命令。
2. 如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel（哨兵）进程标记为**主观下线**（**SDOWN**）。
3. 如果一个Master主服务器被标记为主观下线（SDOWN），则正在监视这个Master主服务器的**所有 Sentinel（哨兵）**进程要以每秒一次的频率**确认Master主服务器**的确**进入了主观下线状态**。
4. 当**有足够数量的 Sentinel（哨兵）**进程（大于等于配置文件指定的值）在指定的时间范围内确认Master主服务器进入了主观下线状态（SDOWN）， 则Master主服务器会被标记为**客观下线（ODOWN）**。
5. 在一般情况下， 每个 Sentinel（哨兵）进程会以每 10 秒一次的频率向集群中的所有Master主服务器、Slave从服务器发送 INFO 命令。
6. 当Master主服务器被 Sentinel（哨兵）进程标记为**客观下线（ODOWN）**时，Sentinel（哨兵）进程向下线的 Master主服务器的所有 Slave从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。
7. 若没有足够数量的 Sentinel（哨兵）进程同意 Master主服务器下线， Master主服务器的客观下线状态就会被移除。若 Master主服务器重新向 Sentinel（哨兵）进程发送 PING 命令返回有效回复，Master主服务器的主观下线状态就会被移除。



## 案例演示

* 修改从机的**sentinel.conf**

|  |
| --- |
| **#sentinel monitor <master-name> <master ip> <master port> <quorum>**  **sentinel monitor mymaster 192.168.10.133 6379 1** |

**其他配置项说明**

|  |
| --- |
| **# Example sentinel.conf**    **# 哨兵sentinel实例运行的端口 默认26379**  **port 26379**    **# 哨兵sentinel的工作目录**  **dir /tmp**    **# 哨兵sentinel监控的redis主节点的 ip port**  **# master-name 可以自己命名的主节点名字 只能由字母A-z、数字0-9 、这三个字符".-\_"组成。**  **# quorum 当这些quorum个数sentinel哨兵认为master主节点失联 那么这时 客观上认为主节点失联了**  **# sentinel monitor <master-name> <ip> <redis-port> <quorum>**  **sentinel monitor mymaster 127.0.0.1 6379 2**    **# 当在Redis实例中开启了requirepass foobared 授权密码 这样所有连接Redis实例的客户端都要提供密码**  **# 设置哨兵sentinel 连接主从的密码 注意必须为主从设置一样的验证密码**  **# sentinel auth-pass <master-name> <password>**  **sentinel auth-pass mymaster MySUPER--secret-0123passw0rd**      **# 指定多少毫秒之后 主节点没有应答哨兵sentinel 此时 哨兵主观上认为主节点下线 默认30秒**  **# sentinel down-after-milliseconds <master-name> <milliseconds>**  **sentinel down-after-milliseconds mymaster 30000**    **# 这个配置项指定了在发生failover主备切换时最多可以有多少个slave同时对新的master进行 同步，**  **这个数字越小，完成failover所需的时间就越长，**  **但是如果这个数字越大，就意味着越 多的slave因为replication而不可用。**  **可以通过将这个值设为 1 来保证每次只有一个slave 处于不能处理命令请求的状态。**  **# sentinel parallel-syncs <master-name> <numslaves>**  **sentinel parallel-syncs mymaster 1**        **# 故障转移的超时时间 failover-timeout 可以用在以下这些方面：**  **#1. 同一个sentinel对同一个master两次failover之间的间隔时间。**  **#2. 当一个slave从一个错误的master那里同步数据开始计算时间。直到slave被纠正为向正确的master那里同步数据时。**  **#3.当想要取消一个正在进行的failover所需要的时间。**  **#4.当进行failover时，配置所有slaves指向新的master所需的最大时间。不过，即使过了这个超时，slaves依然会被正确配置为指向master，但是就不按parallel-syncs所配置的规则来了**  **# 默认三分钟**  **# sentinel failover-timeout <master-name> <milliseconds>**  **sentinel failover-timeout mymaster 180000**    **# SCRIPTS EXECUTION**    **#配置当某一事件发生时所需要执行的脚本，可以通过脚本来通知管理员，例如当系统运行不正常时发邮件通知相关人员。**  **#对于脚本的运行结果有以下规则：**  **#若脚本执行后返回1，那么该脚本稍后将会被再次执行，重复次数目前默认为10**  **#若脚本执行后返回2，或者比2更高的一个返回值，脚本将不会重复执行。**  **#如果脚本在执行过程中由于收到系统中断信号被终止了，则同返回值为1时的行为相同。**  **#一个脚本的最大执行时间为60s，如果超过这个时间，脚本将会被一个SIGKILL信号终止，之后重新执行。**    **#通知型脚本:当sentinel有任何警告级别的事件发生时（比如说redis实例的主观失效和客观失效等等），将会去调用这个脚本，**  **这时这个脚本应该通过邮件，SMS等方式去通知系统管理员关于系统不正常运行的信息。调用该脚本时，将传给脚本两个参数，**  **一个是事件的类型，**  **一个是事件的描述。**  **如果sentinel.conf配置文件中配置了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则sentinel无法正常启动成功。**  **#通知脚本**  **# sentinel notification-script <master-name> <script-path>**  **sentinel notification-script mymaster /var/redis/notify.sh**    **# 客户端重新配置主节点参数脚本**  **# 当一个master由于failover而发生改变时，这个脚本将会被调用，通知相关的客户端关于master地址已经发生改变的信息。**  **# 以下参数将会在调用脚本时传给脚本:**  **# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>**  **# 目前<state>总是“failover”,**  **# <role>是“leader”或者“observer”中的一个。**  **# 参数 from-ip, from-port, to-ip, to-port是用来和旧的master和新的master(即旧的slave)通信的**  **# 这个脚本应该是通用的，能被多次调用，不是针对性的。**  **# sentinel client-reconfig-script <master-name> <script-path>**  **sentinel client-reconfig-script mymaster /var/redis/reconfig.sh** |

* 通过redis-sentinel启动哨兵服务

|  |
| --- |
| **./redis-sentinel sentinel.conf** |

# Redis Cluster集群

redis3.0以后推出的redis cluster 集群方案，redis cluster集群保证了高可用、高性能、高可扩展性。

## 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



最小节点数：**3台**

(1)**节点失效判断**：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)**集群失效判断**: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环境

redis集群需要使用**集群管理脚本redis-trib.rb**，它的执行相应依赖ruby环境。

* 第一步：安装ruby

|  |
| --- |
| yum install ruby  yum install rubygems |

* 第三步：安装ruby和redis的接口程序redis-3.2.2.gem

|  |
| --- |
| gem install redis -V 3.2.2 |

* 第四步：复制redis-3.2.9/src/redis-trib.rb文件到/usr/local/redis目录

|  |
| --- |
| cp redis-3.2.9/src/redis-trib.rb /usr/local/redis-cluster/ -r |

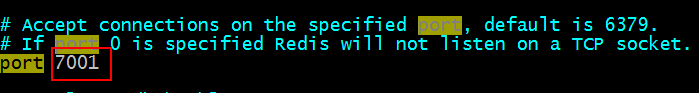
## 安装Redis集群（RedisCluster）

Redis集群最少需要**三台主服务器，三台从服务器**。

端口号分别为：**7001~7006**

* 第一步：创建7001实例，并编辑redis.conf文件，修改port为7001。

注意：创建实例，即拷贝单机版安装时，生成的bin目录，为7001目录。



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



* 第三步：复制7001，创建7002~7006实例，**注意端口修改**。
* 第四步：启动所有的实例
* 第五步：创建Redis集群

|  |
| --- |
| **./redis-trib.rb create --replicas 1 192.168.10.133:7001 192.168.10.133:7002 192.168.10.133:7003 192.168.10.133:7004 192.168.10.133:7005 192.168.10.133:7006**  >>> Creating cluster  Connecting to node 192.168.10.133:7001: OK  Connecting to node 192.168.10.133:7002: OK  Connecting to node 192.168.10.133:7003: OK  Connecting to node 192.168.10.133:7004: OK  Connecting to node 192.168.10.133:7005: OK  Connecting to node 192.168.10.133:7006: OK  >>> Performing hash slots allocation on 6 nodes...  Using 3 masters:  192.168.10.133:7001  192.168.10.133:7002  192.168.10.133:7003  Adding replica 192.168.10.133:7004 to 192.168.10.133:7001  Adding replica 192.168.10.133:7005 to 192.168.10.133:7002  Adding replica 192.168.10.133:7006 to 192.168.10.133:7003  M: d8f6a0e3192c905f0aad411946f3ef9305350420 192.168.10.133:7001  slots:0-5460 (5461 slots) master  M: 7a12bc730ddc939c84a156f276c446c28acf798c 192.168.10.133:7002  slots:5461-10922 (5462 slots) master  M: 93f73d2424a796657948c660928b71edd3db881f 192.168.10.133:7003  slots:10923-16383 (5461 slots) master  S: f79802d3da6b58ef6f9f30c903db7b2f79664e61 192.168.10.133:7004  replicates d8f6a0e3192c905f0aad411946f3ef9305350420  S: 0bc78702413eb88eb6d7982833a6e040c6af05be 192.168.10.133:7005  replicates 7a12bc730ddc939c84a156f276c446c28acf798c  S: 4170a68ba6b7757e914056e2857bb84c5e10950e 192.168.10.133:7006  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 192.168.10.133:7001)  M: d8f6a0e3192c905f0aad411946f3ef9305350420 192.168.10.133:7001  slots:0-5460 (5461 slots) master  M: 7a12bc730ddc939c84a156f276c446c28acf798c 192.168.10.133:7002  slots:5461-10922 (5462 slots) master  M: 93f73d2424a796657948c660928b71edd3db881f 192.168.10.133:7003  slots:10923-16383 (5461 slots) master  M: f79802d3da6b58ef6f9f30c903db7b2f79664e61 192.168.10.133:7004  slots: (0 slots) master  replicates d8f6a0e3192c905f0aad411946f3ef9305350420  M: 0bc78702413eb88eb6d7982833a6e040c6af05be 192.168.10.133:7005  slots: (0 slots) master  replicates 7a12bc730ddc939c84a156f276c446c28acf798c  M: 4170a68ba6b7757e914056e2857bb84c5e10950e 192.168.10.133:7006  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]# |

## 命令客户端连接集群

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

注意：-c 表示是以redis集群方式进行连接

|  |
| --- |
| **./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> |

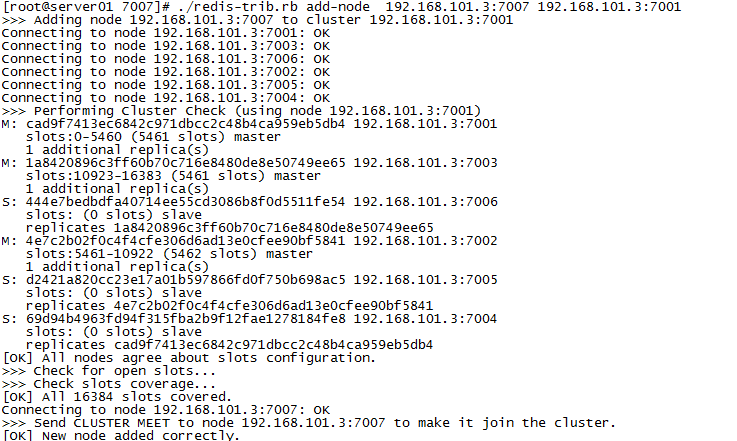
## 维护节点

**集群创建成功后可以继续向集群中添加节点**

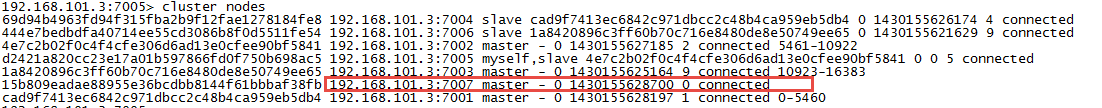
### 添加主节点

* 先创建7007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

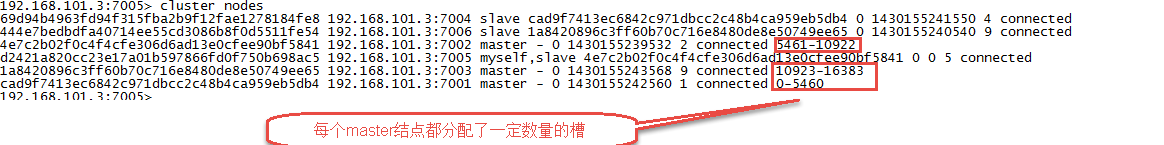


### hash槽重新分配（数据迁移）

**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。

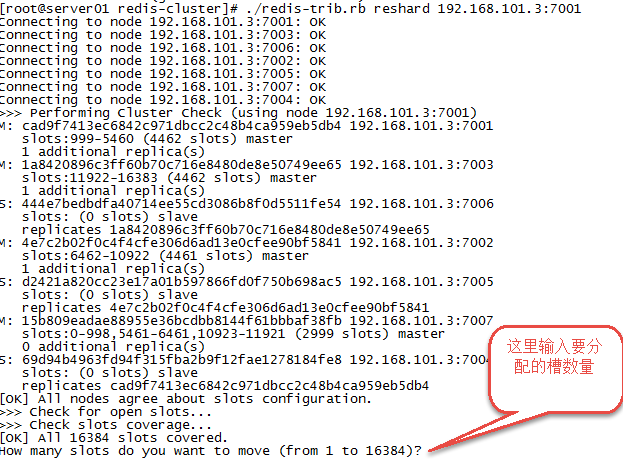


* **给刚添加的7007结点分配槽**

**\* 第一步：连接上集群（连接集群中任意一个可用结点都行）**

|  |
| --- |
| **./redis-trib.rb reshard 192.168.10.133:7001** |

**\* 第二步：输入要分配的槽数量**



输入：**3000**，表示要给目标节点分配3000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

* **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（集群中任一节点都可以）**

执行如下命令：

|  |
| --- |
| **./redis-trib.rb add-node --slave --master-id 35da64607a02c9159334a19164e68dd95a3b943c 192.168.10.103:7008 192.168.10.103:7001** |

**35da64607a02c9159334a19164e68dd95a3b943c**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

需要开启防火墙，或者直接关闭防火墙。

|  |
| --- |
| **service iptables stop** |

### 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("192.168.10.133", 7001));  nodes.add(new HostAndPort("192.168.10.133", 7002));  nodes.add(new HostAndPort("192.168.10.133", 7003));  nodes.add(new HostAndPort("192.168.10.133", 7004));  nodes.add(new HostAndPort("192.168.10.133", 7005));  nodes.add(new HostAndPort("192.168.10.133", 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);  } |

# Redis+LUA整合使用

## 什么是LUA？

* Lua 是一种轻量小巧的**脚本语言**，用标准**C语言**编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

## Redis中使用LUA的好处

1. **减少网络开销**，在Lua脚本中可以把多个命令放在同一个脚本中运行

2. **原子操作**，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件

3. **复用性**，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑

## LUA的安装

* 下载

地址：<http://www.lua.org/download.html>

可以本地下载上传到linux，也可以使用curl命令在linux系统中进行在线下载

***curl -R -O http://www.lua.org/ftp/lua-5.3.5.tar.gz***

* 安装

1. ***yum -y install readline-devel ncurses-devel***
2. ***tar -zxvf lua-5.3.5.tar.gz***
3. ***make linux***
4. ***make install***

如果报错，说找不到readline/readline.h, 可以通过yum命令安装

***yum -y install readline-devel ncurses-devel***

安装完以后再make linux / make install

最后，直接输入 lua命令即可进入lua的控制台

## LUA常见语法

详见<http://www.runoob.com/lua/lua-tutorial.html>

## Redis + LUA整合使用

从Redis2.6.0版本开始，通过**内置的Lua解释器**，可以使用EVAL命令对Lua脚本进行求值。

### EVAL命令

* **命令的格式如下：**

**EVAL script numkeys key [key ...] arg [arg ...]**

* **命令说明：**

\* script参数：是一段Lua脚本程序，它会被运行在Redis服务器上下文中，这段脚本不必(也不应该)定义为一个Lua函数。

\* numkeys参数：用于指定键名参数的个数。

\* key [key ...]参数： 从EVAL的第三个参数开始算起，使用了numkeys个键（key），表示在脚本中所用到的那些Redis键(key)，这些键名参数可以在Lua中通过全局变量KEYS数组，用1为基址的形式访问( KEYS[1] ， KEYS[2] ，以此类推)。

\* arg [arg ...]参数：，可以在Lua中通过全局变量**ARGV**数组访问，访问的形式和KEYS变量类似( ARGV[1] 、 ARGV[2] ，诸如此类)。

\* 例如：

> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second

1) "key1"

2) "key2"

3) "first"

4) "second"

### lua脚本中调用redis命令

**redis.call()：**

**redis.pcall()：**

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误

示例：

> eval "return **redis.call('set',KEYS[1],'bar')**" 1 foo

OK

### EVALSHA

\* EVAL 命令要求你在**每次执行脚本**的时候都**发送一次脚本主体**(script body)。

\* Redis 有一个**内部的缓存机制**，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

\* 为了**减少带宽的消耗**， Redis 实现了 **EVALSHA 命令**，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 **SHA1** 校验和(sum)

**EVALSHA 命令的表现如下：**

\* 如果服务器还记得给定的 SHA1 校验和所指定的脚本，那么执行这个脚本

\* 如果服务器不记得给定的 SHA1 校验和所指定的脚本，那么它返回一个特殊的错误，提醒用户使用 EVAL 代替 EVALSHA

**示例**

> evalsha 6b1bf486c81ceb7edf3c093f4c48582e38c0e791 0

"bar"

### SCRIPT命令

* **SCRIPT FLUSH ：**清除所有脚本缓存
* **SCRIPT EXISTS ：**根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存
* **SCRIPT LOAD ：**将一个脚本装入脚本缓存，**返回SHA1摘要**，但并不立即运行它
* **SCRIPT KILL ：**杀死当前正在运行的脚本

### redis-cli --eval

\* 可以使用redis-cli 命令直接执行脚本

\* 格式如下：

**$ redis-cli --eval script KEYS[1] KEYS[2] , ARGV[1] ARGV[2] ...**

# Redis消息模式

## 队列模式

使用list类型的**lpush和rpop**实现消息队列



**注意事项：**

\* 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。

\* 可以使用**brpop**命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null、

## 发布订阅模式

* 订阅消息（**subscribe**）



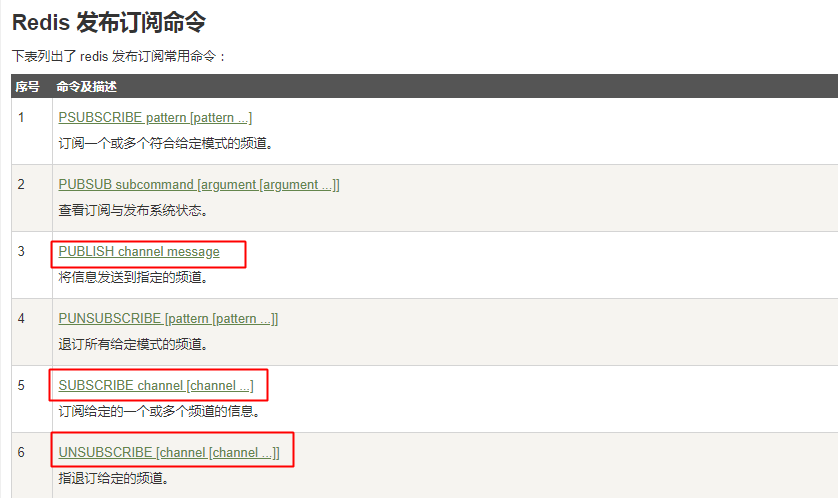
示例：**subscribe kkb-channel**

* 发布消息（**publish**）



publish kkb-channel “我是灭霸詹”

* Redis发布订阅命令



# 缓存穿透、缓存击穿、缓存失效

## 缓存数据的步骤

1. 查询缓存，如果没有数据，则查询数据库
2. 查询数据库，如果数据不为空，将结果写入缓存

## 缓存穿透

* 什么叫缓存穿透？

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

* 如何解决？

**1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。**

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。（**布隆表达式**）

## 缓存雪崩

* 什么叫缓存雪崩？

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

* 如何解决？

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

**2：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。**

3：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期（此点为补充）

## 缓存击穿

* 什么叫缓存击穿？

\* 对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常**“热点”**的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

\* 缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

* 如何解决？

**使用redis的setnx互斥锁先进行判断，这样其他线程就处于等待状态，保证不会有大并发操作去操作数据库。**

**if(redis.sexnx()==1){**

**//查询数据库**

**//加入线程**

**}**

# 缓存淘汰策略之LRU

## Redis内置缓存淘汰策略

* **最大缓存**

\* 在 redis 中，允许用户设置最大使用内存大小**maxmemory**，默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。

\* redis 内存数据集大小上升到一定大小的时候，就会实行**数据淘汰策略**。

* **淘汰策略**

redis淘汰策略配置：**maxmemory-policy** voltile-lru，支持热配置

* **redis 提供 6种数据淘汰策略：**

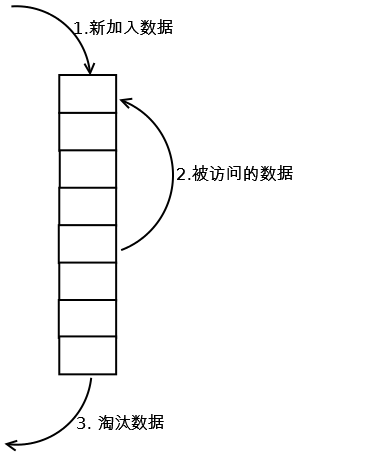
1. voltile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-enviction（驱逐）：禁止驱逐数据

## LRU原理

LRU（**Least recently used**，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

## LRU实现

最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：



1. 新数据插入到链表头部；

2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；

3. 当链表满的时候，将链表尾部的数据丢弃。

在Java中可以使用**LinkHashMap**去实现LRU。

## 分析

【命中率】

当存在热点数据时，**LRU**的效率很好，但偶发性的、周期性的批量操作会导致LRU命中率急剧下降，缓存污染情况比较严重。

【复杂度】

实现简单。

【代价】

命中时需要遍历链表，找到命中的数据块索引，然后需要将数据移到头部