# nosql简介

随着互联网**Web2.0**网站的兴起，传统的关系数据库在应付Web2.0网站，特别是超大规模和高并发的SNS类型的Web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题：

## **对数据库高并发读写的需求**

　　网站要根据用户个性化信息来实时生成动态页面和提供动态信息，所以基本上无法使用动态页面静态化技术，因此数据库并发负载非常高，往往要达到每秒上万次读写请求。关系数据库应付上万次SQL查询还勉强顶得住，但是应付上千万次SQL写数据请求，硬盘IO就已经无法承受了。

## **对海量数据的高效率存储和访问的需求**

　　对于大型的SNS网站，每天用户产生海量的用户动态，以国外的Friendfeed为例，一个月就达到了2.5亿条用户动态，对于关系数据库来说，在一张2.5亿条记录的表里面进行SQL查询，效率是极其低下乃至不可忍受的。

## **对数据库的高可扩展性和高可用性的需求**

　　在基于Web的架构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，你的数据库却没有办法像Web服务器和应用服务器那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供7\*24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移，为什么数据库不能通过不断的添加服务器节点来实现扩展呢？

　　在上面提到的“**三高**”的需求面前，关系数据库遇到了难以克服的障碍，而对于Web2.0网站来说，关系数据库的很多主要特性却往往无用武之地，例如：

　　（1）***数据库事务一致性需求***

　　很多Web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求也不高。因此数据库事务管理成了数据库高负载下一个沉重的负担。

　　（2）**数据库的写实时性和读实时性需求**

　　对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的。并不要求这么高的实时性。

　　（3）***对复杂的SQL查询，特别是多表关联查询的需求***

　　任何大数据量的Web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询，特别是SNS类型的网站，从需求以及产品设计角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。



　　因此，**关系数据库在这些越来越多的应用场景下显得不那么合适了，为了解决这类问题的非关系数据库应运而生**。NoSQL 是非关系型数据存储的广义定义。**它打破了长久以来关系型数据库与ACID理论大一统的局面**。NoSQL 数据存储不需要固定的表结构（例如以键值对存储，它的结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销），通常也不存在连接操作。

# NoSQL无与伦比的特点

　　在大数据存取上具备关系型数据库无法比拟的性能优势，例如：

## 易扩展

　　NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。**数据之间无关系**，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

## 大数据量，高性能

　　NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。

## 灵活的数据模型

　　NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的Web2.0时代尤其明显。

## 高可用

　　NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

**综上所述，NoSQL的非关系特性使其成为了后Web2.0时代的宠儿，助力大型Web2.0网站的再次起飞，是一项全新的数据库革命性运动。**

# Redis脱颖而出



　　随着应用对高性能需求的增加，NoSQL逐渐在各大名企的系统架构中生根发芽。时至今日，涌现出的NoSQL产品已经有很多种了，例如Membase、MongoDB、Apache Cassandra、CouchDB等。不过，在国内外互联网巨头例如**社交巨头新浪微博、传媒巨头Viacom及图片分享领域佼佼者Pinterest**等名企都不约而同地采用了**Redis**作为其NoSQL数据库的选择，到底Redis是何方神圣呢？能让如此多的名企为它而痴狂。

　　按照官方的说法，Redis是一个开源的，使用C语言编写，面向“键/值”（Key/Value）对类型数据的分布式NoSQL数据库系统，特点是**高性能**，**持久存储**，**适应高并发的应用场景**。因此，可以说Redis纯粹为应用而产生，它是一个高性能的key-value数据库，并且还提供了多种语言的API（包括我们的大C#）。那么，也许我们会问：到底性能如何呢？以下是官方的bench-mark数据：

　　测试完成了**50个并发**执行**100000个请求**。

　　设置和获取的值是一个256字节字符串。

　　Linux box是运行Linux 2.6,这是X3320 Xeon 2.5 ghz。

　　文本执行使用loopback接口(127.0.0.1)。

　　结果:**读的速度是110000次/s,写的速度是81000次/s** 。（当然不同的服务器配置性能也有所不同）。

　　和Memcached类似，Redis支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，Redis支持各种不同方式的排序。与Memcached一样，为了保证效率，数据都是缓存在内存中。区别的是**Redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步**（数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。）。

因此，**Redis的出现，很大程度补偿了Memcached这类key/value存储的不足，在部分场合可以对关系数据库起到很好的补充作用**。

## Redis是是什么

redis是一个开源的、使用C语言编写的、支持网络交互的、可基于内存也可持久化的Key-Value数据库。

redis的官网地址，非常好记，是redis.io。（特意查了一下，域名后缀io属于国家域名，是british Indian Ocean territory，即英属印度洋领地）

目前，Vmware在资助着redis项目的开发和维护。

**【redis的作者何许人也】**

开门见山，先看照片：



是不是出乎了你的意料，嗯，高手总会有些地方与众不同的。

这位便是redis的作者，他叫Salvatore Sanfilippo，来自意大利的西西里岛，现在居住在卡塔尼亚。目前供职于Pivotal公司。

他使用的网名是antirez，如果你有兴趣，可以去他的博客逛逛，地址是antirez.com，当然也可以去follow他的github，地址是http://github.com/antirez。

**【谁在使用redis】**

Blizzard、digg、stackoverflow、github、flickr …

## 为什么用redis

# Redis安装

redis是用c语言写的，因此安装redis前提linux上必须安装gcc插件，用来编译c和安装c

## 导入安装包



## 安装环境准备

需要安装gcc编译程序插件：

1. 从互联网上直接下载
2. 从磁盘镜像安装(重启后需要从新挂载)

直接磁盘镜像安装：

1. 挂载磁盘
   1. 命令：mount /dev/cdrom /mnt
2. 安装
   1. 安装命令：yum –y install gcc

## 编译(make)

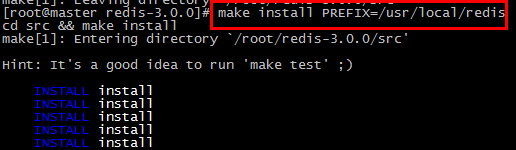
从redis.io下载最新版redis-X.Y.Z.tar.gz后解压，然后进入redis-X.Y.Z文件夹后直接make即可，安装非常简单。

make编译成功后会在src文件夹下产生一些二进制可执行文件，包括redis-server、redis-cli等等：

$ find . -type f -executable  
./redis-benchmark //用于进行redis性能测试的工具  
./redis-check-dump //用于修复出问题的dump.rdb文件  
./redis-cli //redis的客户端  
./redis-server //redis的服务端  
./redis-check-aof //用于修复出问题的AOF文件  
./redis-sentinel //用于集群管理

## 安装(make install)

安装命令：make install PREFIX=/itcast/redis



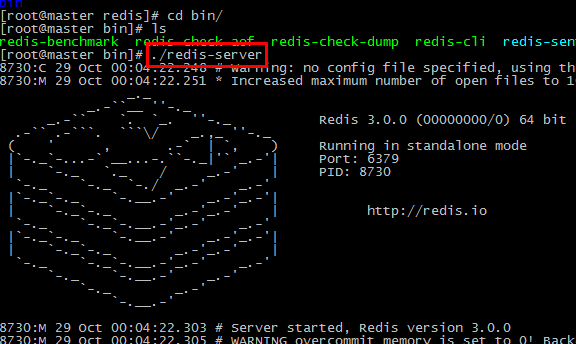
把编译好的文件安装到指定目录

## 前端启动

启动命令：./redis-server（在redis安装目录的bin目录下）

直接运行bin/redis-server将以前端模式启动，

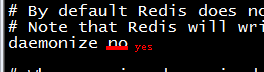
前端模式启动的缺点是ssh命令窗口关闭则redis-server程序结束，不推荐使用此方法。如下图：

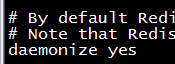


## 后端启动

Redis后台启动需要加载一个配置文件：redis.conf，在配置文件里面配置redis进行后台启动。

1. 拷贝redis.conf
   1. 拷贝redis.conf配置文件到redis安装目录bin.
      1. cp redis.conf ../redis/bin/
         1. 拷贝当前目录的redis.conf到上一级目录下面redis/bin目录下。
2. 修改redis.conf



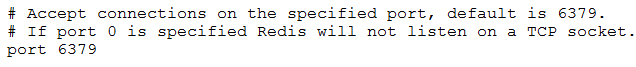


启动命令：./bin/redis-server ./redis.conf

redis默认使用6379端口



也可更改redis.conf文件，修改端口号：



这种方式让redis以进程的方式在后台运行，方便

## 使用redis客户端

### 内置客户端

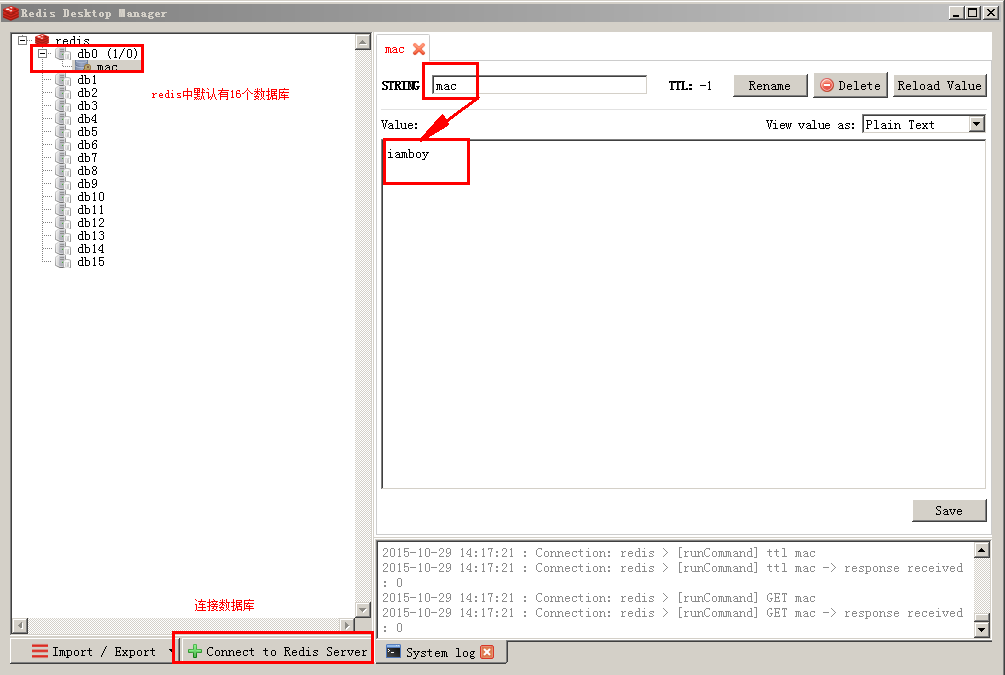
使用redis客户端，必须先启动redis服务端。



//这样来启动redis客户端了  
$ ./redis-cli  
//用set指令来设置key、value  
127.0.0.1:6379> set name "roc"   
OK  
//来获取name的值  
127.0.0.1:6379> get name   
"roc"  
//通过客户端来关闭redis服务端  
127.0.0.1:6379> shutdown   
127.0.0.1:6379>

### 图形化界面





# Redis的数据结构

redis是一种高级的key:value存储系统，其中value支持五种数据类型：

1.字符串（strings）

2.字符串列表（lists）

3.字符串集合（sets）

4.有序字符串集合（sorted sets）

5.哈希（hashes）

而关于key，有几个点要提醒大家：

1.key不要太长，尽量不要超过1024字节，这不仅消耗内存，而且会降低查找的效率；

2.key也不要太短，太短的话，key的可读性会降低；

3.在一个项目中，key最好使用统一的命名模式，例如user:10000:passwd。

## Redis数据结构之Strings

### 命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **APPEND**keyvalue | O(1) | 如果该Key已经存在，APPEND命令将参数Value的数据追加到已存在Value的末尾。如果该Key不存在，APPEND命令将会创建一个新的Key/Value。 | 追加后Value的长度。 |
| **DECR**key | O(1) | 将指定Key的Value原子性的递减1。如果该Key不存在，其初始值为0，在decr之后其值为-1。如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 递减后的Value值。 |
| **INCR**key | O(1) | 将指定Key的Value原子性的递增1。如果该Key不存在，其初始值为0，在incr之后其值为1。如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 递增后的Value值。 |
| **DECRBY**key decrement | O(1) | 将指定Key的Value原子性的减少decrement。如果该Key不存在，其初始值为0，在decrby之后其值为-decrement。如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 减少后的Value值。 |
| **INCRBY**key increment | O(1) | 将指定Key的Value原子性的增加increment。如果该Key不存在，其初始值为0，在incrby之后其值为increment。如果Value的值不能转换为整型值，如Hello，该操作将执行失败并返回相应的错误信息。注意：该操作的取值范围是64位有符号整型。 | 增加后的Value值。 |
| **GET**key | O(1) | 获取指定Key的Value。如果与该Key关联的Value不是string类型，Redis将返回错误信息，因为GET命令只能用于获取string Value。 | 与该Key相关的Value，如果该Key不存在，返回nil。 |
| **SET**key value | O(1) | 设定该Key持有指定的字符串Value，如果该Key已经存在，则覆盖其原有值。 | 总是返回"OK"。 |
| **GETSET**key value | O(1) | 原子性的设置该Key为指定的Value，同时返回该Key的原有值。和GET命令一样，该命令也只能处理string Value，否则Redis将给出相关的错误信息。 | 返回该Key的原有值，如果该Key之前并不存在，则返回nil。 |
| **STRLEN**key | O(1) | 返回指定Key的字符值长度，如果Value不是string类型，Redis将执行失败并给出相关的错误信息。 | 返回指定Key的Value字符长度，如果该Key不存在，返回0。 |
| **SETEX**key seconds value | O(1) | 原子性完成两个操作，一是设置该Key的值为指定字符串，同时设置该Key在Redis服务器中的存活时间(秒数)。该命令主要应用于Redis被当做Cache服务器使用时。 |  |
| **SETNX**key value | O(1) | 如果指定的Key不存在，则设定该Key持有指定字符串Value，此时其效果等价于SET命令。相反，如果该Key已经存在，该命令将不做任何操作并返回。 | 1表示设置成功，否则0。 |
| **SETRANGE**key offset value | O(1) | 替换指定Key的部分字符串值。从offset开始，替换的长度为该命令第三个参数value的字符串长度，其中如果offset的值大于该Key的原有值Value的字符串长度，Redis将会在Value的后面补齐(offset - strlen(value))数量的0x00，之后再追加新值。如果该键不存在，该命令会将其原值的长度假设为0，并在其后添补offset个0x00后再追加新值。鉴于字符串Value的最大长度为512M，因此offset的最大值为536870911。最后需要注意的是，如果该命令在执行时致使指定Key的原有值长度增加，这将会导致Redis重新分配足够的内存以容纳替换后的全部字符串，因此就会带来一定的性能折损。 | 修改后的字符串Value长度。 |
| **GETRANGE**key start end | O(1) | 如果截取的字符串长度很短，我们可以该命令的时间复杂度视为O(1)，否则就是O(N)，这里N表示截取的子字符串长度。该命令在截取子字符串时，将以闭区间的方式同时包含start*(0表示第一个字符)*和end所在的字符，如果end值超过Value的字符长度，该命令将只是截取从start开始之后所有的字符数据。 | 子字符串 |
| **SETBIT**key offset value | O(1) | 设置在指定Offset上BIT的值，该值只能为1或0，在设定后该命令返回该Offset上原有的BIT值。如果指定Key不存在，该命令将创建一个新值，并在指定的Offset上设定参数中的BIT值。如果Offset大于Value的字符长度，Redis将拉长Value值并在指定Offset上设置参数中的BIT值，中间添加的BIT值为0。最后需要说明的是Offset值必须大于0。 | 在指定Offset上的BIT原有值。 |
| **GETBIT**key offset | O(1) | 返回在指定Offset上BIT的值，0或1。如果Offset超过string value的长度，该命令将返回0，所以对于空字符串始终返回0。 | 在指定Offset上的BIT值。 |
| **MGET**key [key ...] | O(N) | N表示获取Key的数量。返回所有指定Keys的Values，如果其中某个Key不存在，或者其值不为string类型，该Key的Value将返回nil。 | 返回一组指定Keys的Values的列表。 |
| **MSET**key value [key value ...] | O(N) | N表示指定Key的数量。该命令原子性的完成参数中所有key/value的设置操作，其具体行为可以看成是多次迭代执行SET命令。 | 该命令不会失败，始终返回OK。 |
| **MSETNX**key value [key value ...] | O(N) | N表示指定Key的数量。该命令原子性的完成参数中所有key/value的设置操作，其具体行为可以看成是多次迭代执行SETNX命令。然而这里需要明确说明的是，*如果在这一批Keys中有任意一个Key已经存在了，那么该操作将全部回滚，即所有的修改都不会生效。* | 1表示所有Keys都设置成功，0则表示没有任何Key被修改。 |

### 实例

有人说，如果只使用redis中的字符串类型，且不使用redis的持久化功能，那么，redis就和memcache非常非常的像了。这说明strings类型是一个很基础的数据类型，也是任何存储系统都必备的数据类型。

我们来看一个最简单的例子：

代码如下:

set mystr "hello world!" //设置字符串类型

get mystr //读取字符串类型

字符串类型的用法就是这么简单，因为是二进制安全的，所以你完全可以把一个图片文件的内容作为字符串来存储。

另外，我们还可以通过字符串类型进行数值操作：

代码如下:

127.0.0.1:6379> set mynum "2"

OK

127.0.0.1:6379> get mynum

**"2"**

127.0.0.1:6379> incr mynum

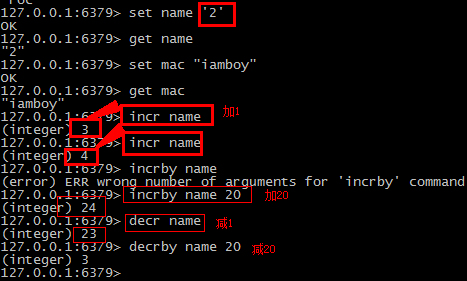
**(integer) 3**

127.0.0.1:6379> get mynum

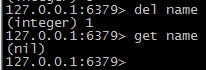
"3"

看，在遇到数值操作时，redis会将字符串类型转换成数值。

由于INCR等指令本身就具有原子操作的特性，所以我们完全可以利用redis的INCR、INCRBY、DECR、DECRBY等指令来实现原子计数的效果，假如，在某种场景下有3个客户端同时读取了mynum的值（值为2），然后对其同时进行了加1的操作，那么，最后mynum的值一定是5。不少网站都利用redis的这个特性来实现业务上的统计计数需求。



我们还可以进行删除：可以发现，删除后获取为null

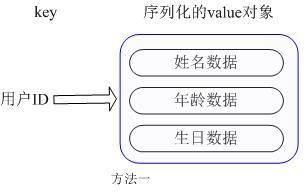


## Redis数据结构之hash

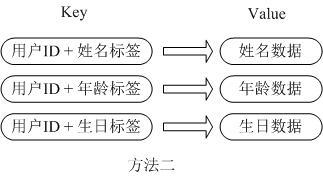
### Hash存储结构

**Hash是一个string 类型的field和value的映射表**。Hash特别适合存储对象，相对于将对象的每个字段存成单个string 类型。一个对象存储在Hash类型中会占用更少的内存，并且可以更方便的存取整个对象。

我们简单举个实例来描述下Hash的应用场景，比如我们要存储一个用户信息对象数据，包含以下信息：用户ID为查找的key，存储的value用户对象包含姓名，年龄，生日等信息，如果用普通的key/value结构来存储，主要有以下2种存储方式：

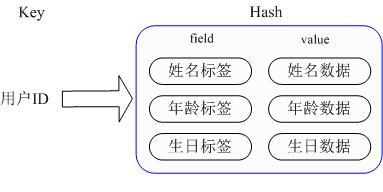


第一种方式将用户ID作为查找key,把其他信息封装成一个对象以序列化的方式存储，这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。



　　第二种方法是这个用户信息对象有多少成员就存成多少个key-value对儿，用用户ID+对应属性的名称作为唯一标识来取得对应属性的值，虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。

　　因此，基于以上两种方式的缺陷，Redis提供的Hash很好的解决了这个问题，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，如下图：



　　也就是说，Key仍然是用户ID, value是一个Map，这个Map的key是成员的属性名，value是属性值，这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过key(用户ID) + field(属性标签) 就可以操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题，也就很好的解决了问题

### 命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **HSET** key field value | O(1) | 为指定的Key设定Field/Value对，如果Key不存在，该命令将创建新Key以参数中的Field/Value对，如果参数中的Field在该Key中已经存在，则用新值覆盖其原有值。 | 1表示新的Field被设置了新值，0表示Field已经存在，用新值覆盖原有值。 |
| **HGET** key field | O(1) | 返回指定Key中指定Field的关联值。 | 返回参数中Field的关联值，如果参数中的Key或Field不存，返回nil。 |
| **HEXISTS**key field | O(1) | 判断指定Key中的指定Field是否存在。 | 1表示存在，0表示参数中的Field或Key不存在。 |
| **HLEN** key | O(1) | 获取该Key所包含的Field的数量。 | 返回Key包含的Field数量，如果Key不存在，返回0。 |
| **HDEL** key field [field ...] | O(N) | 时间复杂度中的N表示参数中待删除的字段数量。从指定Key的Hashes Value中删除参数中指定的多个字段，如果不存在的字段将被忽略。如果Key不存在，则将其视为空Hashes，并返回0. | 实际删除的Field数量。 |
| **HSETNX**key field value | O(1) | 只有当参数中的Key或Field不存在的情况下，为指定的Key设定Field/Value对，否则该命令不会进行任何操作。 | 1表示新的Field被设置了新值，0表示Key或Field已经存在，该命令没有进行任何操作。 |
| **HINCRBY**key field increment | O(1) | 增加指定Key中指定Field关联的Value的值。如果Key或Field不存在，该命令将会创建一个新Key或新Field，并将其关联的Value初始化为0，之后再指定数字增加的操作。该命令支持的数字是64位有符号整型，即increment可以负数。 | 返回运算后的值。 |
| **HGETALL**key | O(N) | 时间复杂度中的N表示Key包含的Field数量。获取该键包含的所有Field/Value。其返回格式为一个Field、一个Value，并以此类推。 | Field/Value的列表。 |
| **HKEYS**key | O(N) | 时间复杂度中的N表示Key包含的Field数量。返回指定Key的所有Fields名。 | Field的列表。 |
| **HVALS**key | O(N) | 时间复杂度中的N表示Key包含的Field数量。返回指定Key的所有Values名。 | Value的列表。 |
| **HMGET**key field [field ...] | O(N) | 时间复杂度中的N表示请求的Field数量。获取和参数中指定Fields关联的一组Values。如果请求的Field不存在，其值返回nil。如果Key不存在，该命令将其视为空Hash，因此返回一组nil。 | 返回和请求Fields关联的一组Values，其返回顺序等同于Fields的请求顺序。 |
| **HMSET** key field value [field value ...] | O(N) | 时间复杂度中的N表示被设置的Field数量。逐对依次设置参数中给出的Field/Value对。如果其中某个Field已经存在，则用新值覆盖原有值。如果Key不存在，则创建新Key，同时设定参数中的Field/Value。 |  |

### 实例

#### HSET/HGET/HDEL/HEXISTS/HLEN/HSETNX

#在Shell命令行启动Redis客户端程序  
    /> redis-cli  
   **#给键值为user的键设置字段为username，值为stephen。  
    redis 127.0.0.1:6379> hset userusername "stephen"  
    (integer) 1  
    #获取键值为user，字段为username的值。  
    redis 127.0.0.1:6379> hget userusername  
    "stephen"  
    #user键中不存在age字段，因此返回nil。  
    redis 127.0.0.1:6379> hget userage  
    (nil)  
    #给user关联的Hashes值添加一个新的字段age，其值为liu。  
    redis 127.0.0.1:6379> hset userage "liu"  
    (integer) 1**    #获取user键的字段数量。  
    redis 127.0.0.1:6379> hlen user  
    (integer) 2  
    #判断user键中是否存在字段名为username的字段，由于存在，返回值为1。  
    redis 127.0.0.1:6379> hexists userusername  
    (integer) 1  
    #删除user键中字段名为username的字段，删除成功返回1。  
    redis 127.0.0.1:6379> hdel userusername  
    (integer) 1  
    #再次删除user键中字段名为username的字段，由于上一条命令已经将其删除，因为没有删除，返回0。  
    redis 127.0.0.1:6379> hdel userusername  
    (integer) 0  
    #判断user键中是否存在username字段，由于上一条命令已经将其删除，因为返回0。  
    redis 127.0.0.1:6379> hexists userusername  
    (integer) 0  
    #通过hsetnx命令给user添加新字段username，其值为stephen，因为该字段已经被删除，所以该命令添加成功并返回1。  
    redis 127.0.0.1:6379> hsetnx userusername stephen  
    (integer) 1

#由于user的username字段已经通过上一条命令添加成功，因为本条命令不做任何操作后返回0。  
    redis 127.0.0.1:6379> hsetnx userusername stephen

(integer) 0

#### HINCRBY

 #删除该键，便于后面示例的测试。  
    redis 127.0.0.1:6379> del user  
    (integer) 1  
    **#准备测试数据，该user的field字段设定值1。  
    redis 127.0.0.1:6379> hset user field 5  
    (integer) 1  
    #给user的field字段的值加1，返回加后的结果。  
    redis 127.0.0.1:6379> hincrby user field 1  
    (integer) 6  
    #给user的field字段的值加-1，返回加后的结果。  
    redis 127.0.0.1:6379> hincrby user field -1  
    (integer) 5  
    #给user的field字段的值加-10，返回加后的结果。  
    redis 127.0.0.1:6379> hincrby user field -10  
    (integer) -5**

#### HGETALL/HKEYS/HVALS/HMGET/HMSET

    #删除该键，便于后面示例测试。  
    redis 127.0.0.1:6379> del user  
    (integer) 1  
   **#为该键user，一次性设置多个字段，分别是username = "hello", age = "world"。  
    redis 127.0.0.1:6379> hmset userusername "hello" age "world"  
    OK**   **#获取user键的多个字段，其中field3并不存在，因为在返回结果中与该字段对应的值为nil。  
    redis 127.0.0.1:6379> hmget userusernameage field3  
    1) "hello"  
    2) "world"  
    3) (nil)**    #返回user键的所有字段及其值，从结果中可以看出，他们是逐对列出的。  
    redis 127.0.0.1:6379> hgetall user  
    1) "username"  
    2) "hello"  
    3) "age"  
    4) "world"  
    #仅获取user键中所有字段的名字。  
    redis 127.0.0.1:6379> hkeys user  
    1) "username"  
    2) "age"  
    #仅获取user键中所有字段的值。  
    redis 127.0.0.1:6379>hvals user  
    1) "hello"  
    2) "world"

## redis数据结构之lists

### 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使用双向链接方式存储数据，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之lists简介

redis的另一个重要的数据结构叫做lists，翻译成中文叫做“列表”。

首先要明确一点，redis中的lists在底层实现上并不是数组，而是链表，也就是说对于一个具有上百万个元素的lists来说，在头部和尾部插入一个新元素，其时间复杂度是常数级别的，比如用LPUSH在10个元素的lists头部插入新元素，和在上千万元素的lists头部插入新元素的速度应该是相同的。

虽然lists有这样的优势，但同样有其弊端，那就是，链表型lists的元素定位会比较慢，而数组型lists的元素定位就会快得多。

lists的常用操作包括LPUSH、RPUSH、LRANGE等。我们可以用LPUSH在lists的左侧插入一个新元素，用RPUSH在lists的右侧插入一个新元素，用LRANGE命令从lists中指定一个范

### 命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **LPUSH**key value [value ...] | O(1) | 在指定Key所关联的List Value的头部插入参数中给出的所有Values。如果该Key不存在，该命令将在插入之前创建一个与该Key关联的空链表，之后再将数据从链表的头部插入。如果该键的Value不是链表类型，该命令将返回相关的错误信息。 | 插入后链表中元素的数量。 |
| **LPUSHX** key value | O(1) | 仅有当参数中指定的Key存在时，该命令才会在其所关联的List Value的头部插入参数中给出的Value，否则将不会有任何操作发生。 | 插入后链表中元素的数量。 |
| **LRANGE** key start stop | O(S+N) | 时间复杂度中的S为start参数表示的偏移量，N表示元素的数量。该命令的参数start和end都是0-based。即0表示链表头部(leftmost)的第一个元素。其中start的值也可以为负值，-1将表示链表中的最后一个元素，即尾部元素，-2表示倒数第二个并以此类推。该命令在获取元素时，start和end位置上的元素也会被取出。如果start的值大于链表中元素的数量，空链表将会被返回。如果end的值大于元素的数量，该命令则获取从start(包括start)开始，链表中剩余的所有元素。 | 返回指定范围内元素的列表。 |
| **LPOP**key | O(1) | 返回并弹出指定Key关联的链表中的第一个元素，即头部元素，。如果该Key不存，返回nil。 | 链表头部的元素。 |
| **LLEN**key | O(1) | 返回指定Key关联的链表中元素的数量，如果该Key不存在，则返回0。如果与该Key关联的Value的类型不是链表，则返回相关的错误信息。 | 链表中元素的数量。 |
| **LREM**key count value | O(N) | 时间复杂度中N表示链表中元素的数量。在指定Key关联的链表中，删除前count个值等于value的元素。如果count大于0，从头向尾遍历并删除，如果count小于0，则从尾向头遍历并删除。如果count等于0，则删除链表中所有等于value的元素。如果指定的Key不存在，则直接返回0。 | 返回被删除的元素数量。 |
| **LSET**key index value | O(N) | 时间复杂度中N表示链表中元素的数量。但是设定头部或尾部的元素时，其时间复杂度为O(1)。设定链表中指定位置的值为新值，其中0表示第一个元素，即头部元素，-1表示尾部元素。如果索引值Index超出了链表中元素的数量范围，该命令将返回相关的错误信息。 |  |
| **LINDEX** key index | O(N) | 时间复杂度中N表示在找到该元素时需要遍历的元素数量。对于头部或尾部元素，其时间复杂度为O(1)。该命令将返回链表中指定位置(index)的元素，index是0-based，表示头部元素，如果index为-1，表示尾部元素。如果与该Key关联的不是链表，该命令将返回相关的错误信息。 | 返回请求的元素，如果index超出范围，则返回nil。 |
| **LTRIM**key start stop | O(N) | N表示被删除的元素数量。该命令将仅保留指定范围内的元素，从而保证链接中的元素数量相对恒定。start和stop参数都是0-based，0表示头部元素。和其他命令一样，start和stop也可以为负值，-1表示尾部元素。如果start大于链表的尾部，或start大于stop，该命令不错报错，而是返回一个空的链表，与此同时该Key也将被删除。如果stop大于元素的数量，则保留从start开始剩余的所有元素。 |  |
| **LINSERT** key BEFORE|AFTER pivot value | O(N) | 时间复杂度中N表示在找到该元素pivot之前需要遍历的元素数量。这样意味着如果pivot位于链表的头部或尾部时，该命令的时间复杂度为O(1)。该命令的功能是在pivot元素的前面或后面插入参数中的元素value。如果Key不存在，该命令将不执行任何操作。如果与Key关联的Value类型不是链表，相关的错误信息将被返回。 | 成功插入后链表中元素的数量，如果没有找到pivot，返回-1，如果key不存在，返回0。 |
| **RPUSH** key value [value ...] | O(1) | 在指定Key所关联的List Value的尾部插入参数中给出的所有Values。如果该Key不存在，该命令将在插入之前创建一个与该Key关联的空链表，之后再将数据从链表的尾部插入。如果该键的Value不是链表类型，该命令将返回相关的错误信息。 | 插入后链表中元素的数量。 |
| **RPUSHX** key value | O(1) | 仅有当参数中指定的Key存在时，该命令才会在其所关联的List Value的尾部插入参数中给出的Value，否则将不会有任何操作发生。 | 插入后链表中元素的数量。 |
| **RPOP**key | O(1) | 返回并弹出指定Key关联的链表中的最后一个元素，即尾部元素，。如果该Key不存，返回nil。 | 链表尾部的元素。 |
| **RPOPLPUSH**source destination | O(1) | 原子性的从与source键关联的链表尾部弹出一个元素，同时再将弹出的元素插入到与destination键关联的链表的头部。如果source键不存在，该命令将返回nil，同时不再做任何其它的操作了。如果source和destination是同一个键，则相当于原子性的将其关联链表中的尾部元素移到该链表的头部。 | 返回弹出和插入的元素。 |

### 实例

#### LPUSH/LPUSHX/LRANGE

/> redis-cli   #在Shell提示符下启动redis客户端工具。  
    redis 127.0.0.1:6379> del mykey  
    (integer) 1  
  #mykey键并不存在，该命令会创建该键及与其关联的List，之后在将参数中的values从左到右依次插入。  
**redis 127.0.0.1:6379> lpush mykey a b c d  
    (integer) 4  
  #取从位置0开始到位置2结束的3个元素。  
    redis 127.0.0.1:6379>lrange mykey 0 2  
    1) "d"  
    2) "c"  
    3) "b"**   **#取链表中的全部元素，其中0表示第一个元素，-1表示最后一个元素。  
    redis 127.0.0.1:6379>lrange mykey 0 -1  
    1) "d"  
    2) "c"  
    3) "b"  
    4) "a"**  
    #mykey2键此时并不存在，因此该命令将不会进行任何操作，其返回值为0。  
    redis 127.0.0.1:6379>lpushx mykey2 e  
    (integer) 0  
    #可以看到mykey2没有关联任何List Value。  
    redis 127.0.0.1:6379> lrange mykey2 0 -1  
    (empty list or set)  
   **#mykey键此时已经存在，所以该命令插入成功，并返回链表中当前元素的数量。  
    redis 127.0.0.1:6379>lpushx mykey e  
    (integer) 5**  
    #获取该键的List Value的头部元素。  
    redis 127.0.0.1:6379>lrange mykey 0 0  
    1) "e"

#### LPOP/LLEN

redis 127.0.0.1:6379> lpush mykey a b c d  
    (integer) 4  
    redis 127.0.0.1:6379> lpop mykey  
    "d"  
    redis 127.0.0.1:6379> lpop mykey  
    "c"  
  #在执行lpop命令两次后，链表头部的两个元素已经被弹出，此时链表中元素的数量是2  
    **redis 127.0.0.1:6379> llen mykey  
    (integer) 2**

#### LREM/LSET/LINDEX/LTRIM

#为后面的示例准备测试数据。  
    redis 127.0.0.1:6379> lpush mykey a b c d a c  
    (integer) 6  
#从头部(left)向尾部(right)变量链表，删除2个值等于a的元素，返回值为实际删除的数量。  
    redis 127.0.0.1:6379>lrem mykey 2 a  
    (integer) 2  
    #看出删除后链表中的全部元素。  
    redis 127.0.0.1:6379>lrange mykey 0 -1  
    1) "c"  
    2) "d"  
    3) "c"  
    4) "b"  
    #获取索引值为1(头部的第二个元素)的元素值。  
    redis 127.0.0.1:6379>lindex mykey 1  
    "d"  
    #将索引值为1(头部的第二个元素)的元素值设置为新值e。  
    redis 127.0.0.1:6379>lset mykey 1 e  
    OK  
    #查看是否设置成功。  
    redis 127.0.0.1:6379>lindex mykey 1  
    "e"  
    #索引值6超过了链表中元素的数量，该命令返回nil。  
    redis 127.0.0.1:6379>lindex mykey 6  
    (nil)  
    #设置的索引值6超过了链表中元素的数量，设置失败，该命令返回错误信息。  
    redis 127.0.0.1:6379> lset mykey 6 hh  
    (error) ERR index out of range  
    #仅保留索引值0到2之间的3个元素，注意第0个和第2个元素均被保留。  
    redis 127.0.0.1:6379>ltrim mykey 0 2  
    OK  
    #查看trim后的结果。  
    redis 127.0.0.1:6379>lrange mykey 0 -1  
    1) "c"  
    2) "e"  
    3) "c"

#### LINSERT

 #删除该键便于后面的测试。  
    redis 127.0.0.1:6379>del mykey  
    (integer) 1  
    #为后面的示例准备测试数据。  
    redis 127.0.0.1:6379> lpush mykey a b c d e  
    (integer) 5  
    #在a的前面插入新元素a1。  
    redis 127.0.0.1:6379>linsert mykey before a a1  
    (integer) 6  
    #查看是否插入成功，从结果看已经插入。注意lindex的index值是0-based。  
    redis 127.0.0.1:6379> lindex mykey 0  
    "e"  
    #在e的后面插入新元素e2，从返回结果看已经插入成功。  
    redis 127.0.0.1:6379>linsert mykey after e e2  
    (integer) 7  
    #再次查看是否插入成功。  
    redis 127.0.0.1:6379>lindex mykey 1  
    "e2"  
    #在不存在的元素之前或之后插入新元素，该命令操作失败，并返回-1。  
    redis 127.0.0.1:6379> linsert mykey after k a  
    (integer) -1  
    #为不存在的Key插入新元素，该命令操作失败，返回0。  
    redis 127.0.0.1:6379>linsert mykey1 after a a2  
    (integer) 0

#### RPUSH/RPUSHX/RPOP/RPOPLPUSH:

#删除该键，以便于后面的测试。  
    redis 127.0.0.1:6379>del mykey  
    (integer) 1  
   **#从链表的尾部插入参数中给出的values，插入顺序是从左到右依次插入。  
    redis 127.0.0.1:6379> rpush mykey a b c d  
    (integer) 4**  
   **#通过lrange的可以获悉rpush在插入多值时的插入顺序。  
    redis 127.0.0.1:6379>lrange mykey 0 -1  
    1) "a"  
    2) "b"  
    3) "c"  
    4) "d"**  
   **#该键已经存在并且包含4个元素，rpushx命令将执行成功，并将元素e插入到链表的尾部。  
    redis 127.0.0.1:6379> rpushx mykey e  
    (integer) 5**  
    #通过lindex命令可以看出之前的rpushx命令确实执行成功，因为索引值为4的元素已经是新元素了。  
    redis 127.0.0.1:6379> lindex mykey 4  
    "e"  
    #由于mykey2键并不存在，因此该命令不会插入数据，其返回值为0。  
    redis 127.0.0.1:6379> rpushx mykey2 e  
    (integer) 0  
    #在执行rpoplpush命令前，先看一下mykey中链表的元素有哪些，注意他们的位置关系。  
    redis 127.0.0.1:6379> lrange mykey 0 -1  
    1) "a"  
    2) "b"  
    3) "c"  
    4) "d"  
    5) "e"  
    **#将mykey的尾部元素e弹出，同时再插入到mykey2的头部(原子性的完成这两步操作)。  
    redis 127.0.0.1:6379>rpoplpush mykey mykey2  
    "e"**  
    #通过lrange命令查看mykey在弹出尾部元素后的结果。  
    redis 127.0.0.1:6379>lrange mykey 0 -1  
    1) "a"  
    2) "b"  
    3) "c"  
    4) "d"  
    #通过lrange命令查看mykey2在插入元素后的结果。  
    redis 127.0.0.1:6379> lrange mykey2 0 -1  
    1) "e"  
    #将source和destination设为同一键，将mykey中的尾部元素移到其头部。  
    redis 127.0.0.1:6379> rpoplpush mykey mykey  
    "d"  
    #查看移动结果。  
    redis 127.0.0.1:6379>lrange mykey 0 -1  
    1) "d"  
    2) "a"  
    3) "b"  
    4) "c"

### 应用

lists的应用相当广泛，随便举几个例子：

1.我们可以利用lists来实现一个消息队列，而且可以确保先后顺序，不必像MySQL那样还需要通过ORDER BY来进行排序。

2.利用LRANGE还可以很方便的实现分页的功能。

3.在博客系统中，每片博文的评论也可以存入一个单独的list中。

## Redis数据结构之set

### Set简介

在Redis中，我们可以将Set类型看作为没有排序的字符集合，和List类型一样，我们也可以在该类型的数据值上执行添加、删除或判断某一元素是否存在等操作。需要说明的是，这些操作的时间复杂度为O(1)，即常量时间内完成次操作。Set可包含的最大元素数量是4294967295。

和List类型不同的是，Set集合中不允许出现重复的元素，这一点和C++标准库中的set容器是完全相同的。换句话说，如果多次添加相同元素，Set中将仅保留该元素的一份拷贝。和List类型相比，Set类型在功能上还存在着一个非常重要的特性，即在服务器端完成多个Sets之间的聚合计算操作，如unions、intersections和differences。由于这些操作均在服务端完成，因此效率极高，而且也节省了大量的网络IO开销。

### 命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **SADD**key member [member ...] | O(N) | 时间复杂度中的N表示操作的成员数量。如果在插入的过程用，参数中有的成员在Set中已经存在，该成员将被忽略，而其它成员仍将会被正常插入。如果执行该命令之前，该Key并不存在，该命令将会创建一个新的Set，此后再将参数中的成员陆续插入。如果该Key的Value不是Set类型，该命令将返回相关的错误信息。 | 本次操作实际插入的成员数量。 |
| **SCARD**key | O(1) | 获取Set中成员的数量。 | 返回Set中成员的数量，如果该Key并不存在，返回0。 |
| **SISMEMBER** key member | O(1) | 判断参数中指定成员是否已经存在于与Key相关联的Set集合中。 | 1表示已经存在，0表示不存在，或该Key本身并不存在。 |
| **SMEMBERS** key | O(N) | 时间复杂度中的N表示Set中已经存在的成员数量。获取与该Key关联的Set中所有的成员。 | 返回Set中所有的成员。 |
| **SPOP**key | O(1) | 随机的移除并返回Set中的某一成员。由于Set中元素的布局不受外部控制，因此无法像List那样确定哪个元素位于Set的头部或者尾部。 | 返回移除的成员，如果该Key并不存在，则返回nil。 |
| **SREM**key member [member ...] | O(N) | 时间复杂度中的N表示被删除的成员数量。从与Key关联的Set中删除参数中指定的成员，不存在的参数成员将被忽略，如果该Key并不存在，将视为空Set处理。 | 从Set中实际移除的成员数量，如果没有则返回0。 |
| **SRANDMEMBER** key | O(1) | 和SPOP一样，随机的返回Set中的一个成员，不同的是该命令并不会删除返回的成员。 | 返回随机位置的成员，如果Key不存在则返回nil。 |
| **SMOVE**source destination member | O(1) | 原子性的将参数中的成员从source键移入到destination键所关联的Set中。因此在某一时刻，该成员或者出现在source中，或者出现在destination中。如果该成员在source中并不存在，该命令将不会再执行任何操作并返回0，否则，该成员将从source移入到destination。如果此时该成员已经在destination中存在，那么该命令仅是将该成员从source中移出。如果和Key关联的Value不是Set，将返回相关的错误信息。 | 1表示正常移动，0表示source中并不包含参数成员。 |
| **SDIFF**key [key ...] | O(N) | 时间复杂度中的N表示所有Sets中成员的总数量。返回参数中第一个Key所关联的Set和其后所有Keys所关联的Sets中成员的差异。如果Key不存在，则视为空Set。 | 差异结果成员的集合。 |
| **SDIFFSTORE**destination key [key ...] | O(N) | 该命令和SDIFF命令在功能上完全相同，两者之间唯一的差别是SDIFF返回差异的结果成员，而该命令将差异成员存储在destination关联的Set中。如果destination键已经存在，该操作将覆盖它的成员。 | 返回差异成员的数量。 |
| **SINTER**key [key ...] | O(N\*M) | 时间复杂度中的N表示最小Set中元素的数量，M则表示参数中Sets的数量。该命令将返回参数中所有Keys关联的Sets中成员的交集。因此如果参数中任何一个Key关联的Set为空，或某一Key不存在，那么该命令的结果将为空集。 | 交集结果成员的集合。 |
| **SINTERSTORE**destination key [key ...] | O(N\*M) | 该命令和SINTER命令在功能上完全相同，两者之间唯一的差别是SINTER返回交集的结果成员，而该命令将交集成员存储在destination关联的Set中。如果destination键已经存在，该操作将覆盖它的成员。 | 返回交集成员的数量。 |
| **SUNION** key [key ...] | O(N) | 时间复杂度中的N表示所有Sets中成员的总数量。该命令将返回参数中所有Keys关联的Sets中成员的并集。 | 并集结果成员的集合。 |
| **SUNIONSTORE**destination key [key ...] | O(N) | 该命令和SUNION命令在功能上完全相同，两者之间唯一的差别是SUNION返回并集的结果成员，而该命令将并集成员存储在destination关联的Set中。如果destination键已经存在，该操作将覆盖它的成员。 | 返回并集成员的数量。 |

### 实例

#### SADD/SMEMBERS/SCARD/SISMEMBER

#在Shell命令行下启动Redis的客户端程序。  
    /> redis-cli  
   **#插入测试数据，由于该键myset之前并不存在，因此参数中的三个成员都被正常插入。  
    redis 127.0.0.1:6379> sadd myset a b c  
    (integer) 3  
    #由于参数中的a在myset中已经存在，因此本次操作仅仅插入了d和e两个新成员。  
    redis 127.0.0.1:6379> sadd myset a d e  
    (integer) 2  
    #判断a是否已经存在，返回值为1表示存在。  
    redis 127.0.0.1:6379> sismember myset a  
    (integer) 1**    #判断f是否已经存在，返回值为0表示不存在。  
    redis 127.0.0.1:6379> sismember myset f  
    (integer) 0  
   **#通过smembers命令查看插入的结果，从结果可以，输出的顺序和插入顺序无关。  
    redis 127.0.0.1:6379>smembers myset  
    1) "c"  
    2) "d"  
    3) "a"  
    4) "b"  
    5) "e"  
    #获取Set集合中元素的数量。  
    redis 127.0.0.1:6379> scard myset  
    (integer) 5**

#### SPOP/SREM/SRANDMEMBER/SMOVE

#删除该键，便于后面的测试。  
    redis 127.0.0.1:6379> del myset  
    (integer) 1  
    #为后面的示例准备测试数据。  
    redis 127.0.0.1:6379> sadd myset a b c d  
    (integer) 4  
    #查看Set中成员的位置。  
    redis 127.0.0.1:6379>smembers myset  
    1) "c"  
    2) "d"  
    3) "a"  
    4) "b"  
    #从结果可以看出，该命令确实是随机的返回了某一成员。  
    redis 127.0.0.1:6379> srandmember myset  
    "c"  
   **#Set中尾部的成员b被移出并返回，事实上b并不是之前插入的第一个或最后一个成员。  
    redis 127.0.0.1:6379> spop myset  
    "b"**   **#查看移出后Set的成员信息。  
    redis 127.0.0.1:6379>smembers myset  
    1) "c"  
    2) "d"  
    3) "a"**    #从Set中移出a、d和f三个成员，其中f并不存在，因此只有a和d两个成员被移出，返回为2。  
    redis 127.0.0.1:6379> srem myset a d f  
    (integer) 2  
    #查看移出后的输出结果。  
    redis 127.0.0.1:6379> smembers myset  
    1) "c"  
    #为后面的smove命令准备数据。  
    redis 127.0.0.1:6379> sadd myset a b  
    (integer) 2  
    redis 127.0.0.1:6379> sadd myset2 c d  
    (integer) 2  
   **#将a从myset移到myset2，从结果可以看出移动成功。  
    redis 127.0.0.1:6379> smove myset myset2 a  
    (integer) 1**   **#再次将a从myset移到myset2，由于此时a已经不是myset的成员了，因此移动失败并返回0。  
    redis 127.0.0.1:6379> smove myset myset2 a  
    (integer) 0**  
    #分别查看myset和myset2的成员，确认移动是否真的成功。  
    redis 127.0.0.1:6379> smembers myset  
    1) "b"  
    redis 127.0.0.1:6379>smembers myset2  
    1) "c"  
    2) "d"  
    3) "a"

#### SDIFF/SDIFFSTORE/SINTER/SINTERSTORE

#为后面的命令准备测试数据。  
    redis 127.0.0.1:6379> sadd myset a b c d  
    (integer) 4  
    redis 127.0.0.1:6379>sadd myset2 c  
    (integer) 1  
    redis 127.0.0.1:6379> sadd myset3 a c e  
    (integer) 3  
    #myset和myset2相比，a、b和d三个成员是两者之间的差异成员。再用这个结果继续和myset3进行差异比较，b和d是myset3不存在的成员。  
    redis 127.0.0.1:6379> sdiff myset myset2 myset3  
    1) "d"  
    2) "b"  
    #将3个集合的差异成员存在在diffkey关联的Set中，并返回插入的成员数量。  
    redis 127.0.0.1:6379> sdiffstore diffkey myset myset2 myset3  
    (integer) 2  
    #查看一下sdiffstore的操作结果。  
    redis 127.0.0.1:6379> smembers diffkey  
    1) "d"  
    2) "b"  
    #从之前准备的数据就可以看出，这三个Set的成员交集只有c。  
    redis 127.0.0.1:6379> sinter myset myset2 myset3  
    1) "c"  
    #将3个集合中的交集成员存储到与interkey关联的Set中，并返回交集成员的数量。  
    redis 127.0.0.1:6379>sinterstore interkey myset myset2 myset3  
    (integer) 1  
    #查看一下sinterstore的操作结果。  
    redis 127.0.0.1:6379> smembers interkey  
    1) "c"  
    #获取3个集合中的成员的并集。      
    redis 127.0.0.1:6379> sunion myset myset2 myset3  
    1) "b"  
    2) "c"  
    3) "d"  
    4) "e"  
    5) "a"  
    #将3个集合中成员的并集存储到unionkey关联的set中，并返回并集成员的数量。  
    redis 127.0.0.1:6379> sunionstore unionkey myset myset2 myset3  
    (integer) 5  
    #查看一下suiionstore的操作结果。  
    redis 127.0.0.1:6379> smembers unionkey  
    1) "b"  
    2) "c"  
    3) "d"  
    4) "e"  
    5) "a"

#### **应用范围**

1). 可以使用Redis的Set数据类型跟踪一些唯一性数据，比如访问某一博客的唯一IP地址信息。对于此场景，我们仅需在每次访问该博客时将访问者的IP存入Redis中，Set数据类型会自动保证IP地址的唯一性。  
      2). 充分利用Set类型的服务端聚合操作方便、高效的特性，可以用于维护数据对象之间的关联关系。比如所有购买某一电子设备的客户ID被存储在一个指定的Set中，而购买另外一种电子产品的客户ID被存储在另外一个Set中，如果此时我们想获取有哪些客户同时购买了这两种商品时，Set的intersections命令就可以充分发挥它的方便和效率的优势了。

## Redis数据结构之sorted-sets

### SortedSets简介

Sorted-Sets和Sets类型极为相似，它们都是字符串的集合，都不允许重复的成员出现在一个Set中。它们之间的主要差别是**Sorted-Sets中的每一个成员都会有一个分数(score)与之关联，Redis正是通过分数来为集合中的成员进行从小到大的排序**。然而需要额外指出的是，尽管Sorted-Sets中的成员必须是唯一的，但是分数(score)却是可以重复的。

在Sorted-Set中添加、删除或更新一个成员都是非常快速的操作，其时间复杂度为集合中成员数量的对数。由于Sorted-Sets中的成员在集合中的位置是有序的，因此，即便是访问位于集合中部的成员也仍然是非常高效的。事实上，Redis所具有的这一特征在很多其它类型的数据库中是很难实现的，换句话说，在该点上要想达到和Redis同样的高效，在其它数据库中进行建模是非常困难的。

### 命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **ZADD** key score member [score] [member] | O(log(N)) | 时间复杂度中的N表示Sorted-Sets中成员的数量。添加参数中指定的所有成员及其分数到指定key的Sorted-Set中，在该命令中我们可以指定多组score/member作为参数。如果在添加时参数中的某一成员已经存在，该命令将更新此成员的分数为新值，同时再将该成员基于新值重新排序。如果键不存在，该命令将为该键创建一个新的Sorted-Sets Value，并将score/member对插入其中。如果该键已经存在，但是与其关联的Value不是Sorted-Sets类型，相关的错误信息将被返回。 | 本次操作实际插入的成员数量。 |
| **ZCARD** key | O(1) | 获取与该Key相关联的Sorted-Sets中包含的成员数量。 | 返回Sorted-Sets中的成员数量，如果该Key不存在，返回0。 |
| **ZCOUNT**key min max | O(log(N)+M) | 时间复杂度中的N表示Sorted-Sets中成员的数量，M则表示min和max之间元素的数量。该命令用于获取分数(score)在min和max之间的成员数量。*针对min和max参数需要额外说明的是，****-inf****和****+inf****分别表示Sorted-Sets中分数的最高值和最低值。缺省情况下，min和max表示的范围是闭区间范围，即****min <= score <= max****内的成员将被返回。然而我们可以通过在min和max的前面添加"****(****"字符来表示开区间，如(min max表示****min < score <= max****，而(min (max表示****min < score < max****。* | 分数指定范围内成员的数量。 |
| **ZINCRBY**key increment member | O(log(N)) | 时间复杂度中的N表示Sorted-Sets中成员的数量。该命令将为指定Key中的指定成员增加指定的分数。如果成员不存在，该命令将添加该成员并假设其初始分数为0，此后再将其分数加上increment。如果Key不存，该命令将创建该Key及其关联的Sorted-Sets，并包含参数指定的成员，其分数为increment参数。如果与该Key关联的不是Sorted-Sets类型，相关的错误信息将被返回。 | 以字符串形式表示的新分数。 |
| **ZRANGE**key start stop [WITHSCORES] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令返回顺序在参数start和stop指定范围内的成员，这里start和stop参数都是0-based，即0表示第一个成员，-1表示最后一个成员。如果start大于该Sorted-Set中的最大索引值，或start > stop，此时一个空集合将被返回。如果stop大于最大索引值，该命令将返回从start到集合的最后一个成员。*如果命令中带有可选参数WITHSCORES选项，该命令在返回的结果中将包含每个成员的分数值，如value1,score1,value2,score2...。* | 返回索引在start和stop之间的成员列表。 |
| **ZRANGEBYSCORE** key min max [WITHSCORES] [LIMIT offset count] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令将返回分数在min和max之间的所有成员，即满足表达式min <= score <= max的成员，其中返回的成员是按照其分数从低到高的顺序返回，如果成员具有相同的分数，则按成员的字典顺序返回。可选参数LIMIT用于限制返回成员的数量范围。可选参数offset表示从符合条件的第offset个成员开始返回，同时返回count个成员。可选参数WITHSCORES的含义参照ZRANGE中该选项的说明。*最后需要说明的是参数中min和max的规则可参照命令ZCOUNT。* | 返回分数在指定范围内的成员列表。 |
| **ZRANK** key member | O(log(N)) | 时间复杂度中的N表示Sorted-Set中成员的数量。Sorted-Set中的成员都是按照分数从低到高的顺序存储，该命令将返回参数中指定成员的位置值，其中0表示第一个成员，它是Sorted-Set中分数最低的成员。 | 如果该成员存在，则返回它的位置索引值。否则返回nil。 |
| **ZREM** key member [member ...] | O(M log(N)) | 时间复杂度中N表示Sorted-Set中成员的数量，M则表示被删除的成员数量。该命令将移除参数中指定的成员，其中不存在的成员将被忽略。如果与该Key关联的Value不是Sorted-Set，相应的错误信息将被返回。 | 实际被删除的成员数量。 |
| **ZREVRANGE** key startstop[WITHSCORES] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令的功能和ZRANGE基本相同，唯一的差别在于该命令是通过反向排序获取指定位置的成员，即从高到低的顺序。如果成员具有相同的分数，则按降序字典顺序排序。 | 返回指定的成员列表。 |
| **ZREVRANK**key member | O(log(N)) | 时间复杂度中的N表示Sorted-Set中成员的数量。该命令的功能和ZRANK基本相同，唯一的差别在于该命令获取的索引是从高到低排序后的位置，同样0表示第一个元素，即分数最高的成员。 | 如果该成员存在，则返回它的位置索引值。否则返回nil。 |
| **ZSCORE**key member | O(1) | 获取指定Key的指定成员的分数。 | 如果该成员存在，以字符串的形式返回其分数，否则返回nil。 |
| **ZREVRANGEBYSCORE**key max min [WITHSCORES] [LIMIT offset count] | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示返回的成员数量。该命令除了排序方式是基于从高到低的分数排序之外，其它功能和参数含义均与ZRANGEBYSCORE相同。 | 返回分数在指定范围内的成员列表。 |
| **ZREMRANGEBYRANK**key start stop | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示被删除的成员数量。删除索引位置位于start和stop之间的成员，start和stop都是0-based，即0表示分数最低的成员，-1表示最后一个成员，即分数最高的成员。 | 被删除的成员数量。 |
| **ZREMRANGEBYSCORE**key min max | O(log(N)+M) | 时间复杂度中的N表示Sorted-Set中成员的数量，M则表示被删除的成员数量。删除分数在min和max之间的所有成员，即满足表达式min <= score <= max的所有成员。对于min和max参数，可以采用开区间的方式表示，具体规则参照ZCOUNT。 | 被删除的成员数量。 |

### 实例

#### ZADD/ZCARD/ZCOUNT/ZREM/ZINCRBY/ZSCORE/ZRANGE/ZRANK

**#在Shell的命令行下启动Redis客户端工具。  
   /> redis-cli  
   #添加一个分数为1的成员。  
    redis 127.0.0.1:6379>zadd myzset 1 "one"  
    (integer) 1  
    #添加两个分数分别是2和3的两个成员。  
    redis 127.0.0.1:6379> zadd myzset 2 "two" 3 "three"  
    (integer) 2  
    #0表示第一个成员，-1表示最后一个成员。WITHSCORES选项表示返回的结果中包含每个成员及其分数，否则只返回成员。  
    redis 127.0.0.1:6379> zrange myzset 0 -1 WITHSCORES  
    1) "one"  
    2) "1"  
    3) "two"  
    4) "2"  
    5) "three"  
    6) "3"**    #获取成员one在Sorted-Set中的位置索引值。0表示第一个位置。  
    redis 127.0.0.1:6379> zrank myzset one  
    (integer) 0  
    #成员four并不存在，因此返回nil。  
    redis 127.0.0.1:6379> zrank myzset four  
    (nil)  
    #获取myzset键中成员的数量。      
    redis 127.0.0.1:6379>zcard myzset  
    (integer) 3  
    #返回与myzset关联的Sorted-Set中，分数满足表达式1 <= score <= 2的成员的数量。  
    redis 127.0.0.1:6379> zcount myzset 1 2  
    (integer) 2  
   **#删除成员one和two，返回实际删除成员的数量。  
    redis 127.0.0.1:6379> zrem myzset one two  
    (integer) 2**  
    #查看是否删除成功。  
    redis 127.0.0.1:6379> zcard myzset  
    (integer) 1  
    #获取成员three的分数。返回值是字符串形式。  
    redis 127.0.0.1:6379> zscore myzset three  
    "3"  
    #由于成员two已经被删除，所以该命令返回nil。  
    redis 127.0.0.1:6379> zscore myzset two  
    (nil)  
    #将成员one的分数增加2，并返回该成员更新后的分数。  
    redis 127.0.0.1:6379> zincrby myzset 2 one  
    "3"  
    #将成员one的分数增加-1，并返回该成员更新后的分数。  
    redis 127.0.0.1:6379> zincrby myzset -1 one  
    "2"  
    #查看在更新了成员的分数后是否正确。  
    redis 127.0.0.1:6379> zrange myzset 0 -1 WITHSCORES  
    1) "one"  
    2) "2"  
    3) "two"  
    4) "2"  
    5) "three"  
    6) "3"

#### ZRANGEBYSCORE/ZREMRANGEBYRANK/ZREMRANGEBYSCORE

redis 127.0.0.1:6379> del myzset  
    (integer) 1  
    redis 127.0.0.1:6379> zadd myzset 1 one 2 two 3 three 4 four  
    (integer) 4  
    #获取分数满足表达式1 <= score <= 2的成员。  
    redis 127.0.0.1:6379>zrangebyscore myzset 1 2  
    1) "one"  
    2) "two"  
    #获取分数满足表达式1 < score <= 2的成员。  
    redis 127.0.0.1:6379> zrangebyscore myzset (1 2  
    1) "two"  
    #-inf表示第一个成员，+inf表示最后一个成员，limit后面的参数用于限制返回成员的自己，  
    #2表示从位置索引(0-based)等于2的成员开始，去后面3个成员。  
    redis 127.0.0.1:6379> zrangebyscore myzset -inf +inf limit 2 3  
    1) "three"  
    2) "four"  
    #删除分数满足表达式1 <= score <= 2的成员，并返回实际删除的数量。  
    redis 127.0.0.1:6379> zremrangebyscore myzset 1 2  
    (integer) 2  
    #看出一下上面的删除是否成功。  
    redis 127.0.0.1:6379> zrange myzset 0 -1  
    1) "three"  
    2) "four"  
    #删除位置索引满足表达式0 <= rank <= 1的成员。  
    redis 127.0.0.1:6379> zremrangebyrank myzset 0 1  
    (integer) 2  
    #查看上一条命令是否删除成功。  
    redis 127.0.0.1:6379> zcard myzset  
    (integer) 0

#### ZREVRANGE/ZREVRANGEBYSCORE/ZREVRANK

    #为后面的示例准备测试数据。  
    redis 127.0.0.1:6379> del myzset  
    (integer) 0  
    redis 127.0.0.1:6379> zadd myzset 1 one 2 two 3 three 4 four  
    (integer) 4  
    #以位置索引从高到低的方式获取并返回此区间内的成员。  
    redis 127.0.0.1:6379> zrevrange myzset 0 -1 WITHSCORES  
    1) "four"  
    2) "4"  
    3) "three"  
    4) "3"  
    5) "two"  
    6) "2"  
    7) "one"  
    8) "1"  
    #由于是从高到低的排序，所以位置等于0的是four，1是three，并以此类推。  
    redis 127.0.0.1:6379> zrevrange myzset 1 3  
    1) "three"  
    2) "two"  
    3) "one"  
    #由于是从高到低的排序，所以one的位置是3。  
    redis 127.0.0.1:6379> zrevrank myzset one  
    (integer) 3  
    #由于是从高到低的排序，所以four的位置是0。  
    redis 127.0.0.1:6379> zrevrank myzset four  
    (integer) 0  
    #获取分数满足表达式3 >= score >= 0的成员，并以相反的顺序输出，即从高到底的顺序。  
    redis 127.0.0.1:6379> zrevrangebyscore myzset 3 0  
    1) "three"  
    2) "two"  
    3) "one"  
    #该命令支持limit选项，其含义等同于zrangebyscore中的该选项，只是在计算位置时按照相反的顺序计算和获取。  
    redis 127.0.0.1:6379>zrevrangebyscore myzset 4 0 limit 1 2  
    1) "three"  
    2) "two"

#### **应用范围**

1). 可以用于一个大型在线游戏的积分排行榜。每当玩家的分数发生变化时，可以执行ZADD命令更新玩家的分数，此后再通过ZRANGE命令获取积分TOP TEN的用户信息。当然我们也可以利用ZRANK命令通过username来获取玩家的排行信息。最后我们将组合使用ZRANGE和ZRANK命令快速的获取和某个玩家积分相近的其他用户的信息。  
    2). Sorted-Sets类型还可用于构建索引数据。

## Redis操作之Keys

### Keys简介

在该系列中，主要讲述的是与Redis数据类型相关的命令，如String、List、Set、Hashes和Sorted-Set。这些命令都具有一个共同点，即所有的操作都是针对与Key关联的Value的。而该篇博客将主要讲述与Key相关的Redis命令。学习这些命令对于学习Redis是非常重要的基础，也是能够充分挖掘Redis潜力的利器。

命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **KEYS** pattern | O(N) | 时间复杂度中的N表示数据库中Key的数量。获取所有匹配pattern参数的Keys。需要说明的是，在我们的正常操作中应该尽量避免对该命令的调用，因为对于大型数据库而言，该命令是非常耗时的，对Redis服务器的性能打击也是比较大的。*pattern支持glob-style的通配符格式，如\*表示任意一个或多个字符，?表示任意字符，[abc]表示方括号中任意一个字母。* | 匹配模式的键列表。 |
| **DEL** key [key ...] | O(N) | 时间复杂度中的N表示删除的Key数量。从数据库删除中参数中指定的keys，如果指定键不存在，则直接忽略。还需要另行指出的是，如果指定的Key关联的数据类型不是String类型，而是List、Set、Hashes和Sorted Set等容器类型，该命令删除每个键的时间复杂度为O(M)，其中M表示容器中元素的数量。而对于String类型的Key，其时间复杂度为O(1)。 | 实际被删除的Key数量。 |
| **EXISTS** key | O(1) | 判断指定键是否存在。 | 1表示存在，0表示不存在。 |
| **MOVE** key db | O(1) | 将当前数据库中指定的键Key移动到参数中指定的数据库中。如果该Key在目标数据库中已经存在，或者在当前数据库中并不存在，该命令将不做任何操作并返回0。 | 移动成功返回1，否则0。 |
| **RENAME** key newkey | O(1) | 为指定指定的键重新命名，如果参数中的两个Keys的命令相同，或者是源Key不存在，该命令都会返回相关的错误信息。如果newKey已经存在，则直接覆盖。 |  |
| **RENAMENX** key newkey | O(1) | 如果新值不存在，则将参数中的原值修改为新值。其它条件和RENAME一致。 | 1表示修改成功，否则0。 |
| **PERSIST** key | O(1) | 如果Key存在过期时间，该命令会将其过期时间消除，使该Key不再有超时，而是可以持久化存储。 | 1表示Key的过期时间被移出，0表示该Key不存在或没有过期时间。 |
| **EXPIRE** key seconds | O(1) | 该命令为参数中指定的Key设定超时的秒数，在超过该时间后，Key被自动的删除。如果该Key在超时之前被修改，与该键关联的超时将被移除。 | 1表示超时被设置，0则表示Key不存在，或不能被设置。 |
| **EXPIREAT** key timestamp | O(1) | 该命令的逻辑功能和EXPIRE完全相同，唯一的差别是该命令指定的超时时间是绝对时间，而不是相对时间。该时间参数是Unix timestamp格式的，即从1970年1月1日开始所流经的秒数。 | 1表示超时被设置，0则表示Key不存在，或不能被设置。 |
| **TTL** key | O(1) | 获取该键所剩的超时描述。 | 返回所剩描述，如果该键不存在或没有超时设置，则返回-1。 |
| **RANDOMKEY** | O(1) | 从当前打开的数据库中随机的返回一个Key。 | 返回的随机键，如果该数据库是空的则返回nil。 |
| **TYPE** key | O(1) | 获取与参数中指定键关联值的类型，该命令将以字符串的格式返回。 | 返回的字符串为string、list、set、hash和zset，如果key不存在返回none。 |
| **SORT** key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination] | O(N+M\*log(M)) | 这个命令相对来说是比较复杂的，因此我们这里只是给出最基本的用法，有兴趣的网友可以去参考redis的官方文档。 | 返回排序后的原始列表。 |

### 实例

#### KEYS/RENAME/DEL/EXISTS/MOVE/RENAMENX

*#在Shell命令行下启动Redis客户端工具。*  
    /> redis-cli  
*#清空当前选择的数据库，以便于对后面示例的理解。*  
    redis 127.0.0.1:6379> *flushdb*  
    OK  
*#添加String类型的模拟数据。*  
    redis 127.0.0.1:6379> *set mykey 2*  
    OK  
    redis 127.0.0.1:6379> *set mykey2 "hello"*  
    OK  
*#添加Set类型的模拟数据。*  
    redis 127.0.0.1:6379> *sadd mysetkey 1 2 3*  
    (integer) 3  
*#添加Hash类型的模拟数据。*  
    redis 127.0.0.1:6379> *hset mmtest username "stephen"*  
    (integer) 1  
*#根据参数中的模式，获取当前数据库中符合该模式的所有key，从输出可以看出，该命令在执行时并不区分与Key关联的Value类型。*  
    redis 127.0.0.1:6379> *keys my\**  
    1) "mysetkey"  
    2) "mykey"  
    3) "mykey2"  
*#删除了两个Keys。*  
    redis 127.0.0.1:6379> *del mykey mykey2*  
    (integer) 2  
*#查看一下刚刚删除的Key是否还存在，从返回结果看，mykey确实已经删除了。*  
    redis 127.0.0.1:6379> *exists mykey*  
    (integer) 0  
*#查看一下没有删除的Key，以和上面的命令结果进行比较。*  
    redis 127.0.0.1:6379> *exists mysetkey*  
    (integer) 1  
 ***#将当前数据库中的mysetkey键移入到ID为1的数据库中，从结果可以看出已经移动成功。*  
    redis 127.0.0.1:6379> *move mysetkey 1*  
    (integer) 1**  
*#打开ID为1的数据库。*  
    redis 127.0.0.1:6379>*select 1*  
    OK  
*#查看一下刚刚移动过来的Key是否存在，从返回结果看已经存在了。*  
    redis 127.0.0.1:6379[1]> *exists mysetkey*  
    (integer) 1  
*#在重新打开ID为0的缺省数据库。*  
    redis 127.0.0.1:6379[1]> *select 0*  
    OK  
*#查看一下刚刚移走的Key是否已经不存在，从返回结果看已经移走。*  
    redis 127.0.0.1:6379> *exists mysetkey*  
    (integer) 0  
*#准备新的测试数据。*  
    redis 127.0.0.1:6379> *set mykey "hello"*  
    OK  
*#将mykey改名为mykey1*  
    redis 127.0.0.1:6379>*rename mykey mykey1*  
    OK  
*#由于mykey已经被重新命名，再次获取将返回nil。*  
    redis 127.0.0.1:6379> *get mykey*  
    (nil)  
*#通过新的键名获取。*  
    redis 127.0.0.1:6379> *get mykey1*  
    "hello"  
*#由于mykey已经不存在了，所以返回错误信息。*  
    redis 127.0.0.1:6379> *rename mykey mykey1*  
    (error) ERR no such key  
*#为renamenx准备测试key*  
    redis 127.0.0.1:6379> *set oldkey "hello"*  
    OK  
    redis 127.0.0.1:6379> *set newkey "world"*  
    OK  
*#由于newkey已经存在，因此该命令未能成功执行。*  
    redis 127.0.0.1:6379> *renamenx oldkey newkey*  
    (integer) 0  
*#查看newkey的值，发现它也没有被renamenx覆盖。*  
    redis 127.0.0.1:6379> *get newkey*  
    "world"

#### PERSIST/EXPIRE/EXPIREAT/TTL

*#为后面的示例准备的测试数据。*  
    redis 127.0.0.1:6379> *set mykey "hello"*  
    OK  
*#将该键的超时设置为100秒。*  
    redis 127.0.0.1:6379> *expire mykey 100*  
    (integer) 1  
*#通过ttl命令查看一下还剩下多少秒。*  
    redis 127.0.0.1:6379> *ttl mykey*  
    (integer) 97  
*#立刻执行persist命令，该存在超时的键变成持久化的键，即将该Key的超时去掉。*  
    redis 127.0.0.1:6379> *persist mykey*  
    (integer) 1  
*#ttl的返回值告诉我们，该键已经没有超时了。*  
    redis 127.0.0.1:6379> *ttl mykey*  
    (integer) -1  
*#为后面的expire命令准备数据。*  
    redis 127.0.0.1:6379>*del mykey*  
    (integer) 1  
    redis 127.0.0.1:6379> *set mykey "hello"*  
    OK  
 *#设置该键的超时被100秒。*  
    redis 127.0.0.1:6379> *expire mykey 100*  
    (integer) 1  
*#用ttl命令看一下当前还剩下多少秒，从结果中可以看出还剩下96秒。*  
    redis 127.0.0.1:6379> *ttl mykey*  
    (integer) 96  
*#重新更新该键的超时时间为20秒，从返回值可以看出该命令执行成功。*  
    redis 127.0.0.1:6379> *expire mykey 20*  
    (integer) 1  
*#再用ttl确认一下，从结果中可以看出果然被更新了。*  
    redis 127.0.0.1:6379> *ttl mykey*  
    (integer) 17  
*#立刻更新该键的值，以使其超时无效。*  
    redis 127.0.0.1:6379> *set mykey "world"*  
    OK  
*#从ttl的结果可以看出，在上一条修改该键的命令执行后，该键的超时也无效了。*  
    redis 127.0.0.1:6379> *ttl mykey*  
    (integer) -1

#### TYPE/RANDOMKEY/SORT

*#由于mm键在数据库中不存在，因此该命令返回none。*  
    redis 127.0.0.1:6379> *type mm*  
    none  
*#mykey的值是字符串类型，因此返回string。*  
    redis 127.0.0.1:6379> *type mykey*  
    string  
*#准备一个值是set类型的键。*  
    redis 127.0.0.1:6379> *sadd mysetkey 1 2*  
    (integer) 2  
    #mysetkey的键是set，因此返回字符串set。  
    redis 127.0.0.1:6379> *type mysetkey*  
    set  
*#返回数据库中的任意键。*  
    redis 127.0.0.1:6379> *randomkey*  
    "oldkey"  
*#清空当前打开的数据库。*  
    redis 127.0.0.1:6379> *flushdb*  
    OK  
*#由于没有数据了，因此返回nil。*  
    redis 127.0.0.1:6379> *randomkey*  
    (nil)

## 多数据库测试

### 实例

一个Redis实例可以包括多个数据库，客户端可以指定连接某个redis实例的哪个数据库，就好比一个mysql中创建多个数据库，客户端连接时指定连接哪个数据库。

一个redis实例最多可提供16个数据库，下标从0到15，客户端默认连接第0号数据库，也可以通过select选择连接哪个数据库，如下连接1号库：



在1号库中查询上节设置的数据，结果查询不到：



重新选择第0号数据库，查询数据：



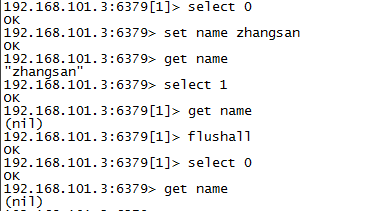
如果选择一个不存在数据库则会报错：



注意：redis不支持修改数据库的名称，只能通过select 0、select 1...选择数据库。

### 注意问题

在0号数据库存储数据，在1号数据库执行清空数据命令却把0号数据库的数据给清空了：



**建议：不同的应用系统要使用不同的redis实例而不是使用同一个redis实例下的不同数据库。**

# Redis事务

## Redis事务简介

和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出Redis中事务的实现特征：

1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

3). 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。

5). 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

## 命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **MULTI** |  | 用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行EXEC时，这些命令才会被原子的执行。 | 始终返回OK |
| **EXEC** |  | 执行在一个事务内命令队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态。如果在事务中执行了WATCH命令，那么只有当WATCH所监控的Keys没有被修改的前提下，EXEC命令才能执行事务队列中的所有命令，否则EXEC将放弃当前事务中的所有命令。 | 原子性的返回事务中各条命令的返回结果。如果在事务中使用了WATCH，一旦事务被放弃，EXEC将返回NULL-multi-bulk回复。 |
| **DISCARD** |  | 回滚事务队列中的所有命令，同时再将当前连接的状态恢复为正常状态，即非事务状态。如果WATCH命令被使用，该命令将UNWATCH所有的Keys。 | 始终返回OK。 |
| **WATCH**key [key ...] | O(1) | 在MULTI命令执行之前，可以指定待监控的Keys，然而在执行EXEC之前，如果被监控的Keys发生修改，EXEC将放弃执行该事务队列中的所有命令。 | 始终返回OK。 |
| **UNWATCH** | O(1) | 取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消。 | 始终返回OK。 |

## 实例

### 事务被正常执行

#在Shell命令行下执行Redis的客户端工具。  
    />redis-cli  
    #在当前连接上启动一个新的事务。  
    redis 127.0.0.1:6379> multi  
    OK  
    #执行事务中的第一条命令，从该命令的返回结果可以看出，该命令并没有立即执行，而是存于事务的命令队列。  
    redis 127.0.0.1:6379> incr t1  
    QUEUED  
    #又执行一个新的命令，从结果可以看出，该命令也被存于事务的命令队列。  
    redis 127.0.0.1:6379> incr t2  
    QUEUED  
    #执行事务命令队列中的所有命令，从结果可以看出，队列中命令的结果得到返回。  
    redis 127.0.0.1:6379> exec  
    1) (integer) 1  
    2) (integer) 1

### 事务中存在失败的命令

#开启一个新的事务。  
    redis 127.0.0.1:6379> multi  
    OK  
    #设置键a的值为string类型的3。  
    redis 127.0.0.1:6379> set a 3  
    QUEUED  
    #从键a所关联的值的头部弹出元素，由于该值是字符串类型，而lpop命令仅能用于List类型，因此在执行exec命令时，该命令将会失败。  
    redis 127.0.0.1:6379> lpop a  
    QUEUED  
    #再次设置键a的值为字符串4。  
    redis 127.0.0.1:6379> set a 4  
    QUEUED  
    #获取键a的值，以便确认该值是否被事务中的第二个set命令设置成功。  
    redis 127.0.0.1:6379> get a  
    QUEUED  
    #从结果中可以看出，事务中的第二条命令lpop执行失败，而其后的set和get命令均执行成功，这一点是Redis的事务与关系型数据库中的事务之间最为重要的差别。  
    redis 127.0.0.1:6379> exec  
    1) OK  
    2) (error) ERR Operation against a key holding the wrong kind of value  
    3) OK  
    4) "4"

### 回滚事务

#为键t2设置一个事务执行前的值。  
    redis 127.0.0.1:6379> set t2 tt  
    OK  
    #开启一个事务。  
    redis 127.0.0.1:6379> multi  
    OK  
    #在事务内为该键设置一个新值。  
    redis 127.0.0.1:6379> set t2 ttnew  
    QUEUED  
    #放弃事务。  
    redis 127.0.0.1:6379> discard  
    OK  
    #查看键t2的值，从结果中可以看出该键的值仍为事务开始之前的值。  
    redis 127.0.0.1:6379> get t2  
    "tt"  
**WATCH命令和基于CAS的乐观锁：**

      在Redis的事务中，WATCH命令可用于提供CAS(check-and-set)功能。假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Null multi-bulk应答以通知调用者事务执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：  
      val = GET mykey  
      val = val + 1  
      SET mykey $val  
      以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：  
      WATCH mykey  
      val = GET mykey  
      val = val + 1  
      MULTI  
      SET mykey $val  
      EXEC  
      和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

# Redis持久化(了解)

## Redis持久化简介

**Redis提供了哪些持久化机制：**  
    1). RDB持久化：  
    该机制是指在指定的时间间隔内将内存中的数据集快照写入磁盘。      
    2). AOF持久化:  
    该机制将以日志的形式记录服务器所处理的每一个写操作，在Redis服务器启动之初会读取该文件来重新构建数据库，以保证启动后数据库中的数据是完整的。  
    3). 无持久化：  
    我们可以通过配置的方式禁用Redis服务器的持久化功能，这样我们就可以将Redis视为一个功能加强版的memcached了。  
    4). 同时应用AOF和RDB。

## **RDB机制的优势和劣势：**

### RDB存在哪些优势呢？

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。  
    2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。  
    3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。  
    4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

### RDB又存在哪些劣势呢？

    1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。  
    2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

## **AOF机制的优势和劣势**

### AOF的优势有哪些呢？

    1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了**3种同步策略，即每秒同步、每修改同步和不同步**。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。  
    2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。  
    3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。  
    4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

### AOF的劣势有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。  
    2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效

## **持久化配置**

### Snapshotting

    缺省情况下，Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开redis.conf文件之后，我们搜索save，可以看到下面的配置信息：  
    save 900 1              #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。  
    save 300 10            #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。  
    save 60 10000        #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

### Dump快照的机制

1). Redis先fork子进程。  
    2). 子进程将快照数据写入到临时RDB文件中。  
    3). 当子进程完成数据写入操作后，再用临时文件替换老的文件。

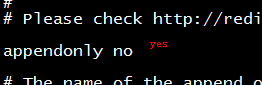
### AOF文件

    上面已经多次讲过，RDB的快照定时dump机制无法保证很好的数据持久性。如果我们的应用确实非常关注此点，我们可以考虑使用Redis中的AOF机制。对于Redis服务器而言，其缺省的机制是RDB，如果需要使用AOF，则需要修改配置文件中的以下条目：  
    将appendonly no改为appendonly yes  
    从现在起，Redis在每一次接收到数据修改的命令之后，都会将其追加到AOF文件中。在Redis下一次重新启动时，需要加载AOF文件中的信息来构建最新的数据到内存中。

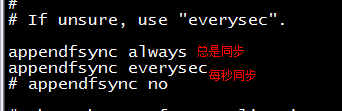
### AOF的配置：

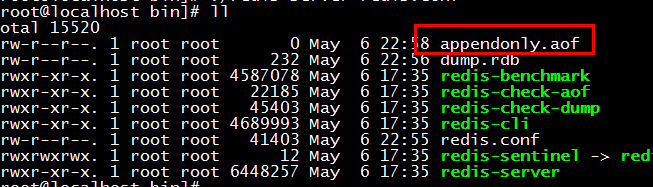
配置aof持久化：

1)开启aof持久化



2)修改aof持久化同步策略





### 如何修复坏损的AOF文件

    1). 将现有已经坏损的AOF文件额外拷贝出来一份。  
    2). 执行"redis-check-aof --fix <filename>"命令来修复坏损的AOF文件。  
    3). 用修复后的AOF文件重新启动Redis服务器。

### Redis的数据备份：

    在Redis中我们可以通过copy的方式在线备份正在运行的Redis数据文件。这是因为RDB文件一旦被生成之后就不会再被修改。Redis每次都是将最新的数据dump到一个临时文件中，之后在利用rename函数原子性的将临时文件改名为原有的数据文件名。因此我们可以说，在任意时刻copy数据文件都是安全的和一致的。鉴于此，我们就可以通过创建cron job的方式定时备份Redis的数据文件，并将备份文件copy到安全的磁盘介质中。

# Redis主从复制(了解)

## Redis主从复制简介

**Redis的Replication：**  
  
    这里首先需要说明的是，在Redis中配置Master-Slave模式真是太简单了。相信在阅读完之后你也可以轻松做到。这里我们还是先列出一些理论性的知识，后面给出实际操作的案例。  
    下面的列表清楚的解释了Redis Replication的特点和优势。  
    1). 同一个Master可以同步多个Slaves。  
    2). Slave同样可以接受其它Slaves的连接和同步请求，这样可以有效的分载Master的同步压力。因此我们可以将Redis的Replication架构视为图结构。  
    3). Master Server是以非阻塞的方式为Slaves提供服务。所以在Master-Slave同步期间，客户端仍然可以提交查询或修改请求。  
    4). Slave Server同样是以非阻塞的方式完成数据同步。在同步期间，如果有客户端提交查询请求，Redis则返回同步之前的数据。  
    5). 为了分载Master的读操作压力，Slave服务器可以为客户端提供只读操作的服务，写服务仍然必须由Master来完成。即便如此，系统的伸缩性还是得到了很大的提高。  
    6). Master可以将数据保存操作交给Slaves完成，从而避免了在Master中要有独立的进程来完成此操作。

## **Replication的工作原理**

    在Slave启动并连接到Master之后，它将主动发送一个SYNC命令。此后Master将启动后台存盘进程，同时收集所有接收到的用于修改数据集的命令，在后台进程执行完毕后，Master将传送整个数据库文件到Slave，以完成一次完全同步。而Slave服务器在接收到数据库文件数据之后将其存盘并加载到内存中。此后，Master继续将所有已经收集到的修改命令，和新的修改命令依次传送给Slaves，Slave将在本次执行这些数据修改命令，从而达到最终的数据同步。  
    如果Master和Slave之间的链接出现断连现象，Slave可以自动重连Master，但是在连接成功之后，一次完全同步将被自动执行。

## **如何配置Replication**

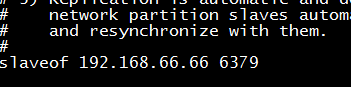
    见如下步骤：

### 临时性主从复制

    1). 同时启动两个Redis服务器，可以考虑在同一台机器上启动两个Redis服务器，分别监听不同的端口，如6379和6380。  
    2). 在Slave服务器上执行一下命令：  
    />redis-cli -p 6380  #这里我们假设Slave的端口号是6380  
    redis 127.0.0.1:6380> slaveof 127.0.0.1 6379#我们假设Master和Slave在同一台主机，Master的端口为6379  
    OK  
    上面的方式只是保证了在执行slaveof命令之后，redis\_6380成为了redis\_6379的slave，一旦服务(redis\_6380)重新启动之后，他们之间的复制关系将终止。

### 永久性主从复制

    如果希望长期保证这两个服务器之间的Replication关系，可以在redis\_6380的配置文件中做如下修改：  
    /> cd /etc/redis  #切换Redis服务器配置文件所在的目录。  
    /> ls  
    6379.conf  6380.conf  
    /> vi 6380.conf  
    将  
**# slaveof <masterip><masterport>**  
    改为  
**slaveof 127.0.0.1 6379**  
    保存退出。

  
    这样就可以保证Redis\_6380服务程序在每次启动后都会主动建立与Redis\_6379的Replication连接了。

## **应用示例**

    这里我们假设Master-Slave已经建立。  
    #启动master服务器。  
    [root@Stephen-PC redis]# redis-cli -p 6379  
    redis 127.0.0.1:6379>  
    #情况Master当前数据库中的所有Keys。  
    redis 127.0.0.1:6379> flushdb  
    OK  
    #在Master中创建新的Keys作为测试数据。  
    redis 127.0.0.1:6379> set mykey hello  
    OK  
    redis 127.0.0.1:6379> set mykey2 world  
    OK  
    #查看Master中存在哪些Keys。  
    redis 127.0.0.1:6379> keys \*  
    1) "mykey"  
    2) "mykey2"  
      
    #启动slave服务器。  
    [root@Stephen-PC redis]# redis-cli -p 6380  
    #查看Slave中的Keys是否和Master中一致，从结果看，他们是相等的。  
    redis 127.0.0.1:6380> keys \*  
    1) "mykey"  
    2) "mykey2"  
      
    #在Master中删除其中一个测试Key，并查看删除后的结果。  
    redis 127.0.0.1:6379> del mykey2  
    (integer) 1  
    redis 127.0.0.1:6379> keys \*  
    1) "mykey"  
      
    #在Slave中查看是否mykey2也已经在Slave中被删除。  
    redis 127.0.0.1:6380> keys \*  
    1) "mykey"

# 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>

## 通过jedis连接redis单机

### jar包

pom坐标：

<dependency>

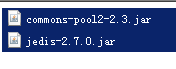
<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.7.0</version>

</dependency>

jar包如下：



### 单实例连接

通过创建单实例jedis对象连接redis服务，如下代码：

// 单实例连接redis

@Test

**publicvoid** testJedisSingle() {

Jedis jedis = **new** Jedis("192.168.101.3", 6379);

jedis.set("name", "bar");

String name = jedis.get("name");

System.*out*.println(name);

jedis.close();

}

#### 连接超时解决

由于linux防火墙默认开启，redis的服务端口6379并不在开放规则之内，所有需要将此端口开放访问或者关闭防火墙。

关闭防火墙命令：sevice iptables stop

如果是修改防火墙规则，可以修改：/etc/sysconfig/iptables文件

### 使用连接池连接

通过单实例连接redis不能对redis连接进行共享，可以使用连接池对redis连接进行共享，提高资源利用率，使用jedisPool连接redis服务，如下代码：

@Test

**publicvoid** pool() {

JedisPoolConfig config = **new** JedisPoolConfig();

//最大连接数

config.setMaxTotal(30);

//最大连接空闲数

config.setMaxIdle(2);

JedisPool pool = **new** JedisPool(config, "192.168.101.3", 6379);

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();

}

}

}

详细的连接池配置参数参考下节jedis和spring整合中applicationContext.xml的配置内容。

### jedis与spring整合

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 -->

<beanid=*"jedisPoolConfig"*class=*"redis.clients.jedis.JedisPoolConfig"*>

<!-- 最大连接数 -->

<propertyname=*"maxTotal"*value=*"30"*/>

<!-- 最大空闲连接数 -->

<propertyname=*"maxIdle"*value=*"10"*/>

<!-- 每次释放连接的最大数目 -->

<propertyname=*"numTestsPerEvictionRun"*value=*"1024"*/>

<!-- 释放连接的扫描间隔（毫秒） -->

<propertyname=*"timeBetweenEvictionRunsMillis"*value=*"30000"*/>

<!-- 连接最小空闲时间 -->

<propertyname=*"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 -->

<propertyname=*"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 -->

<propertyname=*"maxWaitMillis"*value=*"1500"*/>

<!-- 在获取连接的时候检查有效性, 默认false -->

<propertyname=*"testOnBorrow"*value=*"true"*/>

<!-- 在空闲时检查有效性, 默认false -->

<propertyname=*"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 -->

<propertyname=*"blockWhenExhausted"*value=*"false"*/>

</bean>

<!-- redis单机通过连接池 -->

<beanid=*"jedisPool"*class=*"redis.clients.jedis.JedisPool"*destroy-method*="close"*>

<constructor-arg name="poolConfig" ref="jedisPoolConfig"/>

<constructor-arg name="host" value="192.168.25.145"/>

<constructor-arg name="port" value="6379"/>

</bean>

测试代码：

**private** ApplicationContext applicationContext;

@Before

**publicvoid** init() {

applicationContext = **new** ClassPathXmlApplicationContext(

"classpath:applicationContext.xml");

}

@Test

**publicvoid** testJedisPool() {

JedisPool pool = (JedisPool) applicationContext.getBean("jedisPool");

**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集群

## 集群原理

### 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

#### redis-cluster投票:容错



(1)领着投票过程是集群中所有master参与,如果半数以上master节点与master节点通信超过(cluster-node-timeout),认为当前master节点挂掉.

(2):什么时候整个集群不可用(cluster\_state:fail)?

    a:如果集群任意master挂掉,且当前master没有slave.集群进入fail状态,也可以理解成集群的slot映射[0-16383]不完成时进入fail状态. ps : redis-3.0.0.rc1加入cluster-require-full-coverage参数,默认关闭,打开集群兼容部分失败.

    b: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.

  ps:当集群不可用时,所有对集群的操作做都不可用，收到((error) CLUSTERDOWN The cluster is down)错误

## 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

yum install ruby(联网和使用磁盘镜像都可以)

yum install rubygems(联网使用磁盘镜像都可)

安装ruby和redis的接口程序

拷贝redis-3.0.0.gem(ruby和redis接口)至/usr/local下

执行：

gem install /usr/local/redis-3.0.0.gem

或使用 gem install redis --version 3.0.0

## 创建集群：

### 创建8个目录

[root@localhost redis-cluster]# mkdir 700{1..8}

[root@localhost redis-cluster]# ll

total 32

drwxr-xr-x. 2 root root 4096 May 7 00:29 7001

drwxr-xr-x. 2 root root 4096 May 7 00:29 7002

drwxr-xr-x. 2 root root 4096 May 7 00:29 7003

drwxr-xr-x. 2 root root 4096 May 7 00:29 7004

drwxr-xr-x. 2 root root 4096 May 7 00:29 7005

drwxr-xr-x. 2 root root 4096 May 7 00:29 7006

drwxr-xr-x. 2 root root 4096 May 7 00:29 7007

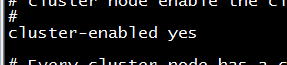
drwxr-xr-x. 2 root root 4096 May 7 00:29 7008

### 开启集群

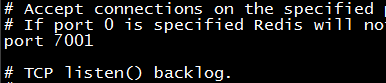
拷贝bin到7001，然后修改

cp -r bin ../redis-cluster/7001/

修改redis配置文件，开启集群。



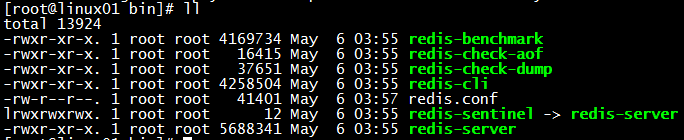
修改端口：端口必须和目录相同：7001



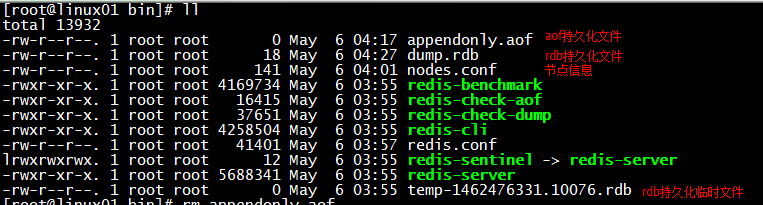
### 拷贝安装目录bin

注意：拷贝前将redis中的持久化文件也清空，在搭建集群时不能有数据存在

最终目录应该是这样的



不应该是这样的



拷贝7001/bin拷贝7002….7008的7个目录。

[root@localhost 7001]# cp -r bin ../7002/

[root@localhost 7001]# cp -r bin ../7003/

[root@localhost 7001]# cp -r bin ../7004/

[root@localhost 7001]# cp -r bin ../7005/

[root@localhost 7001]# cp -r bin ../7006/

[root@localhost 7001]# cp -r bin ../7007/

[root@localhost 7001]# cp -r bin ../7008/

### 修改端口

修改7002。。。7008端口。同时删除持久化数据，创建集群时不允许有数据。

### 拷贝创建集群工具

使用redis-trib.rb工具创建集群。

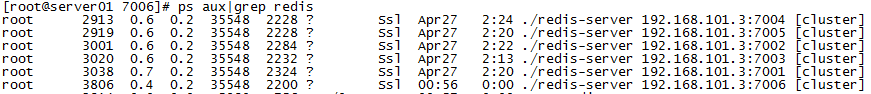
cp redis-3.0.0/src/redis-trib.rb redis-cluster/

### 启动每个结点redis服务

分别进入7001、7002、...7006目录，

执行：./redis-server ./redis.conf

查看redis进程：



### 执行创建集群命令

执行**redis-trib.rb**，此脚本是ruby脚本，它依赖ruby环境。

|  |
| --- |
| ./redis-trib.rb create --replicas 1 192.168.66.60:7001 192.168.66.60:7002 192.168.66.60:7003 192.168.66.60:7004 192.168.66.60:7005 192.168.66.60:7006 |

说明：

redis集群至少需要3个主节点，每个主节点有一个从节点总共6个节点

replicas指定为1表示每个主节点有一个从节点

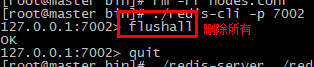
**注意：**

如果执行时报如下错误：

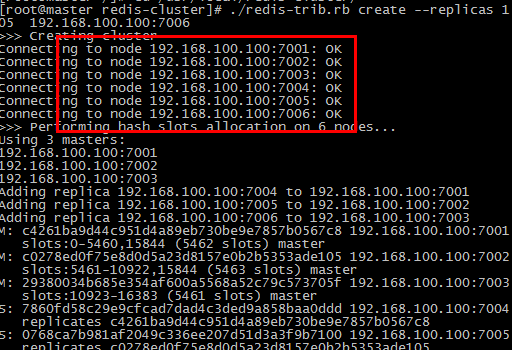
[ERR] Node XXXXXX is not empty. Either the node already knows other nodes (check with CLUSTER NODES) or contains some key in database 0

解决方法是删除生成的配置文件nodes.conf，如果不行则说明现在创建的结点包括了旧集群的结点信息，需要删除redis的持久化文件后再重启redis，比如：appendonly.aof、dump.rdb

如果还是不行：是因为redis数据库存在多余的数据，我们需要登录redis数据进行删除：



启动成功



**注意：修改了配置文件需要重新启动服务**

创建集群输出如下：

>>> Creating cluster

Connecting to node 192.168.101.3:7001: OK

Connecting to node 192.168.101.3:7002: OK

Connecting to node 192.168.101.3:7003: OK

Connecting to node 192.168.101.3:7004: OK

Connecting to node 192.168.101.3:7005: OK

Connecting to node 192.168.101.3:7006: OK

>>> Performing hash slots allocation on 6 nodes...

Using 3 masters:

192.168.101.3:7001

192.168.101.3:7002

192.168.101.3:7003

Adding replica 192.168.101.3:7004 to 192.168.101.3:7001

Adding replica 192.168.101.3:7005 to 192.168.101.3:7002

Adding replica 192.168.101.3:7006 to 192.168.101.3:7003

M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001

slots:0-5460 (5461 slots) master

M: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002

slots:5461-10922 (5462 slots) master

M: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003

slots:10923-16383 (5461 slots) master

S: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004

replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4

S: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005

replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841

S: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006

replicates 1a8420896c3ff60b70c716e8480de8e50749ee65

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 192.168.101.3:7001)

M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001

slots:0-5460 (5461 slots) master

M: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002

slots:5461-10922 (5462 slots) master

M: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003

slots:10923-16383 (5461 slots) master

M: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004

slots: (0 slots) master

replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4

M: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005

slots: (0 slots) master

replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841

M: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006

slots: (0 slots) master

replicates 1a8420896c3ff60b70c716e8480de8e50749ee65

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

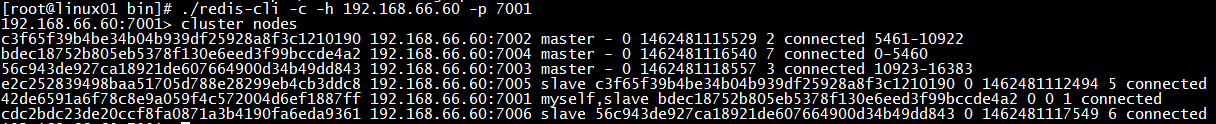
>>> Check slots coverage...

[OK] All 16384 slots covered.

## 查询集群信息

集群创建成功登陆任意redis结点查询集群中的节点情况。

客户端以集群方式登陆：

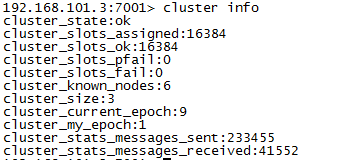


说明：

**./redis-cli -c -h 192.168.101.3 -p 7001 ，其中-c表示以集群方式连接redis，-h指定ip地址，-p指定端口号**

cluster nodes 查询集群结点信息

cluster info 查询集群状态信息



## 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，参考集群结点规划章节添加一个“7007”目录作为新节点。**

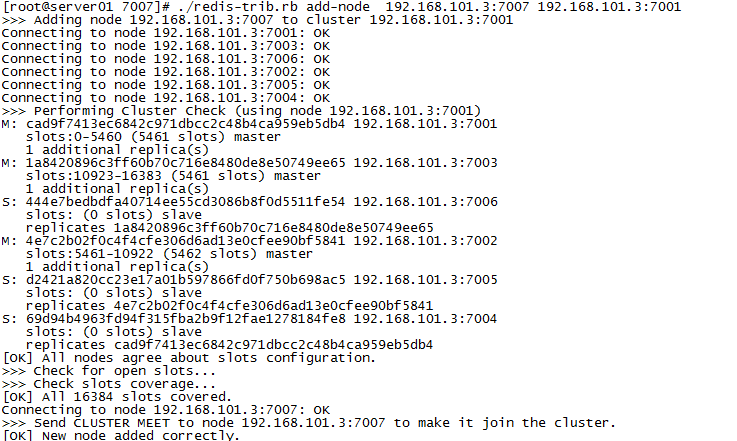
执行下边命令：

**./redis-trib.rb add-node 192.168.66.66:7007 192.168.66.66:7001**

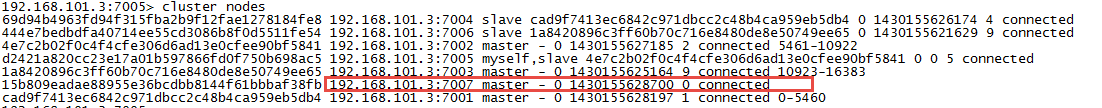
１） add-node就是添加节点

２） add-node默认添加主节点

３） 192.168.66.66:7007是需要添加节点

４） 192.168.66.66:7001：参数节点，把节点添加到7001所在集群。

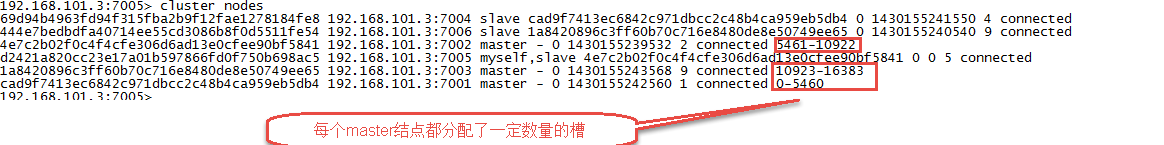
查看集群结点发现7007已添加到集群中：



### hash槽重新分配

添加完主节点需要对主节点进行hash槽分配这样该主节点才可以存储数据。

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。

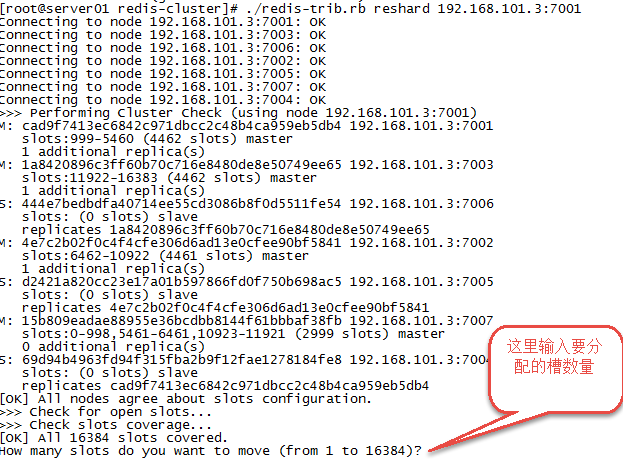


给刚添加的7007结点分配槽：

第一步：连接上集群

**./redis-trib.rb reshard 192.168.66.60:7001（连接集群中任意一个可用结点都行）**

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



这里准备给7007分配槽，通过cluster nodes查看7007结点id为15b809eadae88955e36bcdbb8144f61bbbaf38fb

输入：15b809eadae88955e36bcdbb8144f61bbbaf38fb

第四步：输入源结点id



这里输入all

第五步：输入yes开始移动槽到目标结点id



## 添加从节点

给7007添加一个7008从节点。7008服务必须处于启动状态。

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。**

命令格式：

**./redis-trib.rb add-node --slave --master-id 主节点id添加节点的ip和端口集群中已存在节点ip和端口**

添加命令：

**./redis-trib.rb add-node --slave --master-id 2318f7b27bc4638046628a2e83acdddfff555706 192.168.66.60:7008 192.168.66.60:7001**

1. **add-node：添加节点**
2. **--slave：添加从节点**
3. **--master-Id xx ：表示给此主节点添加从节点。**
4. **192.168.66.66:7008：需要添加从节点**
5. **192.168.66.66:7001：表示需要添加到7001所在集群。**

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：

./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）。

## java代码测试redisCluster集群

### 测试代码

|  |
| --- |
| // 连接redis集群  @Test  **publicvoid** testJedisCluster() {  JedisPoolConfig config = **new** JedisPoolConfig();  // 最大连接数  config.setMaxTotal(30);  // 最大连接空闲数  config.setMaxIdle(2);  //集群结点  Set<HostAndPort> jedisClusterNode = **new** HashSet<HostAndPort>();  jedisClusterNode.add(**new** HostAndPort("192.168.101.3", 7001));  jedisClusterNode.add(**new** HostAndPort("192.168.101.3", 7002));  jedisClusterNode.add(**new** HostAndPort("192.168.101.3", 7003));  jedisClusterNode.add(**new** HostAndPort("192.168.101.3", 7004));  jedisClusterNode.add(**new** HostAndPort("192.168.101.3", 7005));  jedisClusterNode.add(**new** HostAndPort("192.168.101.3", 7006));  JedisCluster jc = **new** JedisCluster(jedisClusterNode, config);    JedisCluster jcd = **new** JedisCluster(jedisClusterNode);  jcd.set("name", "zhangsan");  String value = jcd.get("name");  System.*out*.println(value);  } |

### 使用spring

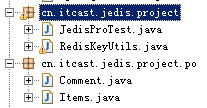
配置applicationContext.xml

|  |
| --- |
| <!-- 连接池配置 -->  <beanid=*"jedisPoolConfig"*class=*"redis.clients.jedis.JedisPoolConfig"*>  <!-- 最大连接数 -->  <propertyname=*"maxTotal"*value=*"30"*/>  <!-- 最大空闲连接数 -->  <propertyname=*"maxIdle"*value=*"10"*/>  <!-- 每次释放连接的最大数目 -->  <propertyname=*"numTestsPerEvictionRun"*value=*"1024"*/>  <!-- 释放连接的扫描间隔（毫秒） -->  <propertyname=*"timeBetweenEvictionRunsMillis"*value=*"30000"*/>  <!-- 连接最小空闲时间 -->  <propertyname=*"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 -->  <propertyname=*"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 -->  <propertyname=*"maxWaitMillis"*value=*"1500"*/>  <!-- 在获取连接的时候检查有效性, 默认false -->  <propertyname=*"testOnBorrow"*value=*"true"*/>  <!-- 在空闲时检查有效性, 默认false -->  <propertyname=*"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 -->  <propertyname=*"blockWhenExhausted"*value=*"false"*/>  </bean>  <!-- redis集群 -->  <beanid=*"jedisCluster"*class=*"redis.clients.jedis.JedisCluster"*>  <constructor-argindex=*"0"*>  <set>  <beanclass=*"redis.clients.jedis.HostAndPort"*>  <constructor-argindex=*"0"*value=*"192.168.101.3"*></constructor-arg>  <constructor-argindex=*"1"*value=*"7001"*></constructor-arg>  </bean>  <beanclass=*"redis.clients.jedis.HostAndPort"*>  <constructor-argindex=*"0"*value=*"192.168.101.3"*></constructor-arg>  <constructor-argindex=*"1"*value=*"7002"*></constructor-arg>  </bean>  <beanclass=*"redis.clients.jedis.HostAndPort"*>  <constructor-argindex=*"0"*value=*"192.168.101.3"*></constructor-arg>  <constructor-argindex=*"1"*value=*"7003"*></constructor-arg>  </bean>  <beanclass=*"redis.clients.jedis.HostAndPort"*>  <constructor-argindex=*"0"*value=*"192.168.101.3"*></constructor-arg>  <constructor-argindex=*"1"*value=*"7004"*></constructor-arg>  </bean>  <beanclass=*"redis.clients.jedis.HostAndPort"*>  <constructor-argindex=*"0"*value=*"192.168.101.3"*></constructor-arg>  <constructor-argindex=*"1"*value=*"7005"*></constructor-arg>  </bean>  <beanclass=*"redis.clients.jedis.HostAndPort"*>  <constructor-argindex=*"0"*value=*"192.168.101.3"*></constructor-arg>  <constructor-argindex=*"1"*value=*"7006"*></constructor-arg>  </bean>  </set>  </constructor-arg>  <constructor-argindex=*"1"*ref=*"jedisPoolConfig"*></constructor-arg>  </bean> |

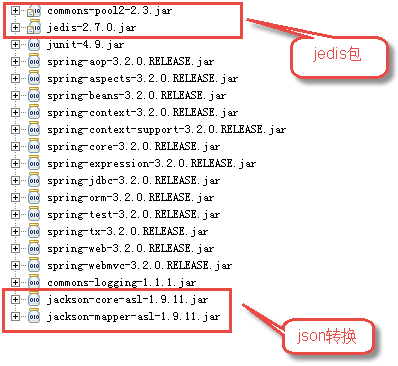
测试代码

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

# redisCluster应用测试代码



## jar包



## RedisKeyUtils

**publicclass**RedisKeyUtils {

//商品id自增key

**publicfinalstatic** String *ITEMS\_ID* = "ITEMS:ID";

//商品排行榜key

**publicstaticfinal** String *ITEMS\_SELLSORT* = "ITEMS:SELLSORT";

//商品信息key

**publicstatic** String getItemKey(Long itemId){

**return**"ITEMS:" + itemId;

}

//商品评论key

**publicstatic** String getItemCommentKey(Long itemId){

**return**"ITEMS:COMMENT" + itemId;

}

}

## po

**publicclass**Comment {

**privatelong**id;

//评论内容

**private** String name;

//评论时间

**private** Date date;

.....

**publicclass** Items {

**private** Integer id;

**private** String name;

**private** Float price;

**private** String pic;

**private** Date createtime;

**private** String detail;

## JedisProTest

**publicclass** JedisProTest {

**private** ApplicationContext applicationContext;

// 使用jackson进行json转换

**privatestaticfinal** ObjectMapper *OBJECT\_MAPPER* = **new** ObjectMapper();

// redis集群

**private** JedisCluster jedisCluster;

@Before

**publicvoid** init() {

applicationContext = **new** ClassPathXmlApplicationContext(

"classpath:applicationContext.xml");

jedisCluster = (JedisCluster) applicationContext

.getBean("jedisCluster");

}

// 获取商品自增id主键

@Test

**publicvoid** testGetItemsId() {

Long item\_id = jedisCluster.incr(RedisKeyUtils.*ITEMS\_ID*);

System.*out*.println(item\_id);

}

// 向hash中存储商品信息

@Test

**publicvoid** testSaveItems() {

jedisCluster.hset(RedisKeyUtils.*getItemKey*(1002l), "id", "3");

jedisCluster.hset(RedisKeyUtils.*getItemKey*(1002l), "name", "苹果4");

jedisCluster.hset(RedisKeyUtils.*getItemKey*(1002l), "price", "999.9");

}

// 根据商品id从hash获取商品信息

@Test

**publicvoid** testGetItemsById() {

Map<String, String> map = jedisCluster.hgetAll(RedisKeyUtils

.*getItemKey*(1002l));

System.*out*.println(map);

}

// 向list存储商品评论列表

@Test

**publicvoid** testSaveItemsComment() **throws** JsonGenerationException,

JsonMappingException, IOException {

// 将商品评价信息转成json

Comment comment = **new** Comment();

comment.setId(1l);

comment.setName("商品不错，很好！！");

comment.setDate(**new** Date());

// 将商品评论转成json串

String comment\_json = *OBJECT\_MAPPER*.writeValueAsString(comment);

System.*out*.println(comment\_json);

// 将商品评论存储到redis

jedisCluster

.lpush(RedisKeyUtils.*getItemCommentKey*(1001l), comment\_json);

// 获取商品评论

List<String> lrange = jedisCluster.lrange(

RedisKeyUtils.*getItemCommentKey*(1001l), 0, -1);

**for** (String content : lrange) {

// 将json数据转成java对象

System.*out*.println(content);

Comment comment\_obj = *OBJECT\_MAPPER*.readValue(content,

Comment.**class**);

System.*out*.println(comment\_obj);

}

}

// 商品排行榜

// 向sortedSet存储商品评论列表

@Test

**publicvoid** testItemsSellSort() {

//初始化商品销量

//商品1001

jedisCluster.zadd(RedisKeyUtils.*ITEMS\_SELLSORT*, 100, "1001");

//商品1002

jedisCluster.zadd(RedisKeyUtils.*ITEMS\_SELLSORT*, 200, "1002");

//商品销售出增加销量

//增加1001商品的销量

jedisCluster.zincrby(RedisKeyUtils.*ITEMS\_SELLSORT*, 1, "1001");

//商品销量前10名

Set<Tuple> zrevrangeWithScores = jedisCluster.zrevrangeWithScores(RedisKeyUtils.*ITEMS\_SELLSORT*, 0, 9);

**for**(Tuple content:zrevrangeWithScores){

System.*out*.println(content.getElement()+" "+content.getScore());

}

}

//设置key的过期时间

@Test

**publicvoid** testSetKeyExpire() **throws** InterruptedException{

//设置test的值为1

jedisCluster.set("test", "1");

System.*out*.println(jedisCluster.get("test"));

//设置生成时间为5秒，单位为秒

jedisCluster.expire("test", 5);

//设置生成时间为5秒，单位为毫秒

// jedisCluster.pexpire("test", 5000);

Thread.*sleep*(5000);

System.*out*.println(jedisCluster.get("test"));

}

}

# 附

## redis.conf

Redis 支持很多的参数，但都有默认值。

 daemonize:

默认情况下，redis 不是在后台运行的，如果需要在后台运行，把该项的值更改为yes

 pidfile

当Redis 在后台运行的时候，Redis 默认会把pid 文件放在/var/run/redis.pid，你可以配

置到其他地址。当运行多个redis 服务时，需要指定不同的pid 文件和端口

 bind

指定Redis 只接收来自于该IP 地址的请求，如果不进行设置，那么将处理所有请求，在

生产环境中最好设置该项

 port

监听端口，默认为6379

 timeout

设置客户端连接时的超时时间，单位为秒。当客户端在这段时间内没有发出任何指令，

那么关闭该连接

 loglevel

log 等级分为4 级，debug, verbose, notice, 和warning。生产环境下一般开启notice

 logfile

配置log 文件地址，默认使用标准输出，即打印在命令行终端的窗口上

 databases

设置数据库的个数，可以使用SELECT <dbid>命令来切换数据库。默认使用的数据库是0

 save

设置Redis 进行数据库镜像的频率。

if(在60 秒之内有10000 个keys 发生变化时){

进行镜像备份

}else if(在300 秒之内有10 个keys 发生了变化){

进行镜像备份

}else if(在900 秒之内有1 个keys 发生了变化){

进行镜像备份

}

 rdbcompression

在进行镜像备份时，是否进行压缩

 dbfilename

镜像备份文件的文件名

 dir

数据库镜像备份的文件放置的路径。这里的路径跟文件名要分开配置是因为Redis 在进

行备份时，先会将当前数据库的状态写入到一个临时文件中，等备份完成时，再把该该

临时文件替换为上面所指定的文件，而这里的临时文件和上面所配置的备份文件都会放

在这个指定的路径当中

 slaveof

设置该数据库为其他数据库的从数据库

 masterauth

当主数据库连接需要密码验证时，在这里指定

 requirepass

设置客户端连接后进行任何其他指定前需要使用的密码。警告：因为redis 速度相当快，

所以在一台比较好的服务器下，一个外部的用户可以在一秒钟进行150K 次的密码尝试，

这意味着你需要指定非常非常强大的密码来防止暴力破解。

 maxclients

限制同时连接的客户数量。当连接数超过这个值时，redis 将不再接收其他连接请求，

客户端尝试连接时将收到error 信息。

 maxmemory

设置redis 能够使用的最大内存。当内存满了的时候，如果还接收到set 命令，redis 将

先尝试剔除设置过expire 信息的key，而不管该key 的过期时间还没有到达。在删除时，

将按照过期时间进行删除，最早将要被过期的key 将最先被删除。如果带有expire 信息

的key 都删光了，那么将返回错误。这样，redis 将不再接收写请求，只接收get 请求。

maxmemory 的设置比较适合于把redis 当作于类似memcached 的缓存来使用。

 appendonly

默认情况下，redis 会在后台异步的把数据库镜像备份到磁盘，但是该备份是非常耗时

的，而且备份也不能很频繁，如果发生诸如拉闸限电、拔插头等状况，那么将造成比较

大范围的数据丢失。所以redis 提供了另外一种更加高效的数据库备份及灾难恢复方式。

开启append only 模式之后，redis 会把所接收到的每一次写操作请求都追加到

appendonly.aof 文件中，当redis 重新启动时，会从该文件恢复出之前的状态。但是这样

会造成appendonly.aof 文件过大，所以redis 还支持了BGREWRITEAOF 指令，对

appendonly.aof 进行重新整理。所以我认为推荐生产环境下的做法为关闭镜像，开启

appendonly.aof，同时可以选择在访问较少的时间每天对appendonly.aof 进行重写一次。

 appendfsync

设置对appendonly.aof 文件进行同步的频率。always 表示每次有写操作都进行同步，

everysec 表示对写操作进行累积，每秒同步一次。这个需要根据实际业务场景进行配置

 vm-enabled

是否开启虚拟内存支持。因为redis 是一个内存数据库，而且当内存满的时候，无法接

收新的写请求，所以在redis 2.0 中，提供了虚拟内存的支持。但是需要注意的是，redis

中，所有的key 都会放在内存中，在内存不够时，只会把value 值放入交换区。这样保

证了虽然使用虚拟内存，但性能基本不受影响，同时，你需要注意的是你要把

vm-max-memory 设置到足够来放下你的所有的key

 vm-swap-file

设置虚拟内存的交换文件路径

 vm-max-memory

这里设置开启虚拟内存之后，redis 将使用的最大物理内存的大小。默认为0，redis 将

把他所有的能放到交换文件的都放到交换文件中，以尽量少的使用物理内存。在生产环

境下，需要根据实际情况设置该值，最好不要使用默认的0

 vm-page-size

设置虚拟内存的页大小，如果你的value 值比较大，比如说你要在value 中放置博客、

新闻之类的所有文章内容，就设大一点，如果要放置的都是很小的内容，那就设小一点。

 vm-pages

设置交换文件的总的page 数量，需要注意的是，page table 信息会放在物理内存中，每

8 个page 就会占据RAM 中的1 个byte。总的虚拟内存大小＝ vm-page-size \* vm-pages

 vm-max-threads

设置VM IO 同时使用的线程数量。因为在进行内存交换时，对数据有编码和解码的过

程，所以尽管IO 设备在硬件上本上不能支持很多的并发读写，但是还是如果你所保存

的vlaue 值比较大，将该值设大一些，还是能够提升性能的

 glueoutputbuf

把小的输出缓存放在一起，以便能够在一个TCP packet 中为客户端发送多个响应，具体

原理和真实效果我不是很清楚。所以根据注释，你不是很确定的时候就设置成yes

 hash-max-zipmap-entries

在redis 2.0 中引入了hash 数据结构。当hash 中包含超过指定元素个数并且最大的元素

没有超过临界时，hash 将以一种特殊的编码方式（大大减少内存使用）来存储，这里

可以设置这两个临界值

 activerehashing

开启之后，redis 将在每100 毫秒时使用1 毫秒的CPU 时间来对redis 的hash 表进行重

新hash，可以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能

够接受Redis 时不时的对请求有2 毫秒的延迟的话，把这项配置为no。如果没有这么严

格的实时性要求，可以设置为yes，以便能够尽可能快的释放内存

## /usr/local/redis/bin目录下的几个文件时什么

　　redis-benchmark：redis性能测试工具

　　redis-check-aof：检查aof日志的工具

　　redis-check-dump：检查rdb日志的工具

　　redis-cli：连接用的客户端

　　redis-server：redis服务进程