# 1 Redis简介

## 1.1 什么是redis

Redis官网: http://redis.io/

Redis是一个开源的高性能键值对**数据库**

是NoSQL技术阵营中的一员

它通过提供多种键值数据类型来适应不同场景下的存储需求

借助一些高层级的接口使其可以胜任，如缓存、队列系统的不同角色

源码托管在github

<https://github.com/antirez/redis>

## 1.2 redis特性

* 多种数据类型存储
* 字符串类型
* 散列类型
* 列表类型
* 集合类型
* 有序集合类型
* 内存存储与持久化结合
* 内存的读写速度远快于硬盘
* 自身提供了持久化功能（RDB、AOF两种方式）
* 功能丰富
* 可用作缓存、队列、消息订阅/发布
* 支持键的生存时间
* 按照一定规则删除相应的键
* 简单稳定
* 相比SQL而言更加简单
* 不同语言的客户端丰富
* 基于C语言开发，代码量只有3万多行

## 1.3 Redis版本说明

Redis原生只支持linux系统

目前linux平台下：

Redis最新稳定版本是：3.0.3

最新测试版为：3.2

Redis官方是不支持Windows平台的，Windows版本是由微软自己（Microsoft Open Tech group）建立的分支，基于官方的Redis源码上进行编译、发布、维护的，所以Windows平台的Redis版本都要略低于官方版本。

## 1.4 Redis下载及安装

最新稳定版下载链接：

<http://download.redis.io/releases/redis-3.0.3.tar.gz>

安装

1/解压源码并进入目录 tar -zxvf redis-2.8.19.tar.gz -C ./redis-src/

2/ make

3/安装到指定目录,如 /usr/local/redis

make PREFIX=/usr/local/redis install

4/拷贝一份配置文件到安装目录下

切换到源码目录，里面有一份配置文件 redis.conf，然后将其拷贝到安装路径下

cp redis.conf /usr/local/redis/

安装完后，在/usr/local/redis/bin下有几个可执行文件

|  |  |
| --- | --- |
| Redis-server | Redis服务器 |
| Redis-cli | Redis命令行客户端 |
| Redis-benchmark | Redis性能测试工具 |
| Redis-check-aof | AOF文件修复工具 |
| Redis-check-dump | RDB文件检查工作 |

运行

bin/redis-server

(默认前端运行模式。后台进程模式： 修改redis.conf daemonize yes)

也可以手动控制后端运行，按这种形式来启动：

nohup bin/redis-server 1 > /dev/null 2 > &1 &

用客户端连接redis服务器：

bin/redis-cli -h 127.0.0.1 –p 6379

测试：

redis 127.0.0.1:6379> PING

PONG

## 1.5 Redis的JAVA客户端Jedis

# 1 Redis安装

## 导入安装包



## 编译(make)

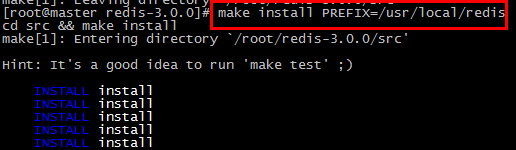
进入到redis目录下 输入make命令进行编译

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)





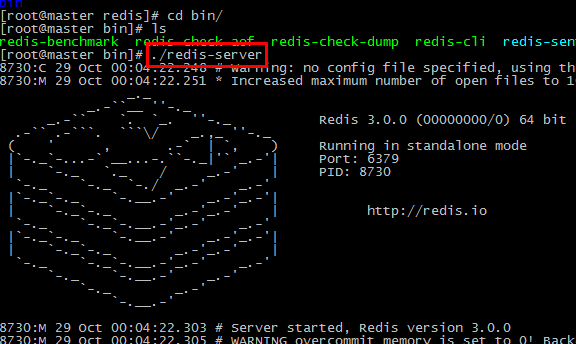
把编译好的文件安装到指定目录

到你解压出来的redis目录下将redis.conf复制到安装目录下



## 前端启动

切换到安装目录下（cd /usr/local/redis）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

修改redis.conf配置文件， daemonize yes 以后端模式启动。

执行如下命令启动redis：

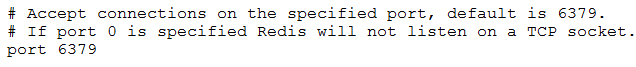
cd /usr/local/redis

./bin/redis-server ./redis.conf

redis默认使用6379端口

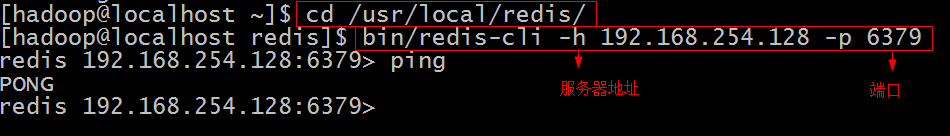


也可更改redis.conf文件，修改端口号：



## 使用redis客户端

### 内置客户端



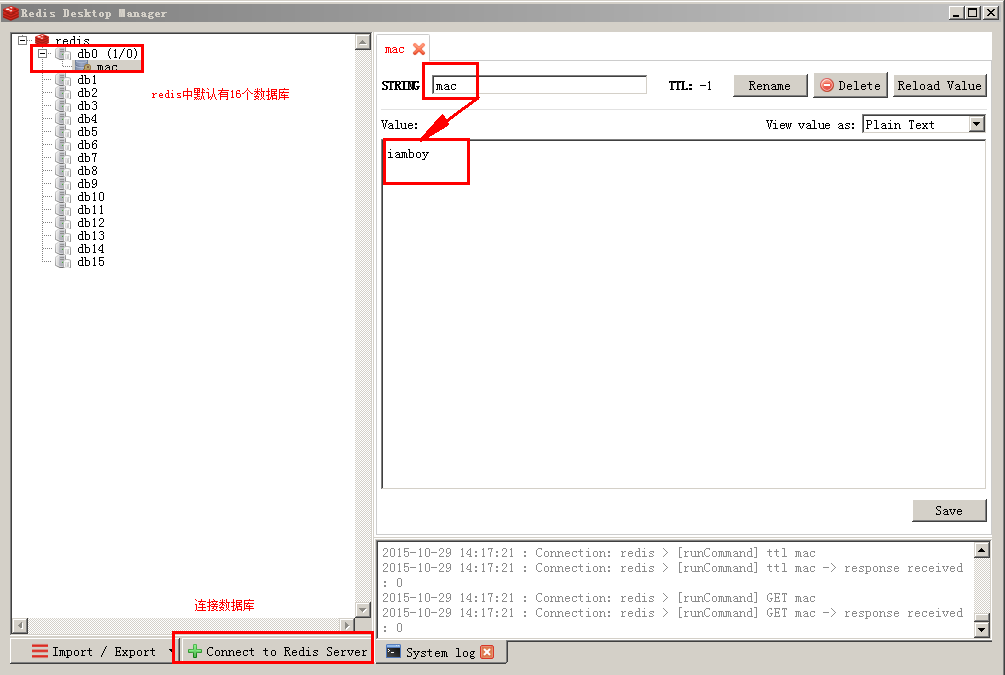
使用redis客户端，必须先启动redis服务端。



//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。

# 2 Redis数据结构之String

## 2.1 概述：

字符串类型是Redis中最为基础的数据存储类型，它在Redis中是二进制安全的，这便意味着该类型可以接受任何格式的数据，如JPEG图像数据或Json对象描述信息等。在Redis中字符串类型的Value最多可以容纳的数据长度是512M。

## 2.2命令列表

|  |  |  |  |
| --- | --- | --- | --- |
| **命令原型** | **时间复杂度** | **命令描述** | **返回值** |
| **APPEND** key value | 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（创建一个key）。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。（expire） |  |
| **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) | 截取字符串。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。最后一个字符从右边开始数的话是-1，往左数为-2… 所以想从开始截取到最后可以让start为0，end为-1，截取到倒数第二个的话end就为-2 | 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。（布隆过滤器） | 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 [key1 value1 key2 ...] | 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。 |

## 2.3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

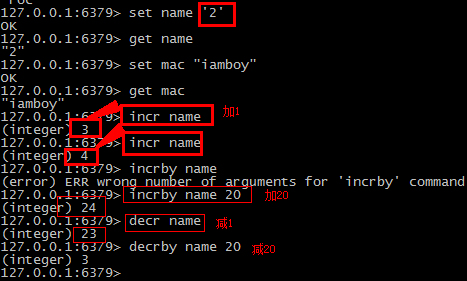
**(integer) 3**

127.0.0.1:6379> get mynum

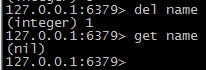
"3"

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



## 2.4String相关功能（Jedis客户端API）

2.3.1 Redis JAVA客户端Jedis

首先，引入依赖，用maven的方式：

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.7.2</version>  <type>jar</type>  <scope>compile</scope>  </dependency> |

接下来，获取到一个redis客户端连接对象

|  |
| --- |
| public static void main(String[] args) {  Jedis jedis = new Jedis("192.168.2.199");    //测试客户端与redis服务器的联通性  String ping = jedis.ping();  System.out.println(ping);  } |

2.3.2 Jedis的String类型api示例

|  |
| --- |
| public class JedisDemo {  Jedis jedis = null;  @Before  public void init() {  jedis = new Jedis("192.168.2.199");  }  /\*\*  \* String类型数据的api测试  \*/  @Test  public void testString() {  // 插入一条string类型的数据  String res = jedis.set("jedis-s-key-01", "itcast ia the greatest it school");  System.out.println(res);  // 获取一条string类型的数据  String value = jedis.get("jedis-s-key-01");  System.out.println(value);  // 子字符串的获取  String value2 = jedis.getrange("jedis-s-key-01", 0, 5);  System.out.println(value2);  // 子字符串的替换  jedis.setrange("jedis-s-key-01", 0, "itheima");  String value3 = jedis.get("jedis-s-key-01");  System.out.println(value3);  jedis.setrange("jedis-s-key-01", 26, "trainningschool");  String value4 = jedis.get("jedis-s-key-01");  System.out.println(value4);  //如果偏移量超出字符串长度，则会自动补\0x00  jedis.setrange("jedis-s-key-01", 50, "very good");  String value5 = jedis.get("jedis-s-key-01");  System.out.println(value5);  //setnx会判断指定的key是否存在，如果已存在，则不会插入数据  for (int i = 0; i < 5; i++) {  jedis.setnx("jedis-s-key-0" + i, "000-" + i);  }  }  public static void main(String[] args) {  Jedis jedis = new Jedis("192.168.2.199");  // 测试客户端与redis服务器的联通性  String ping = jedis.ping();  System.out.println(ping);  }  } |

# 3 Redis数据结构之List

## 3.0 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涉及到位移操作，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。





## 3.1 redis之List概述：

在Redis中，List类型是按照插入顺序排序的字符串链表。和数据结构中的**普通链表**一样，我们可以在其头部(left)和尾部(right)添加新的元素。在插入时，如果该键并不存在，Redis将为该键创建一个新的链表。与此相反，如果链表中所有的元素均被移除，那么该键也将会被从数据库中删除。List中可以包含的最大元素数量是4294967295。

从元素插入和删除的效率视角来看，如果我们是在链表的两头插入或删除元素，这将会是非常高效的操作，即使链表中已经存储了百万条记录，该操作也可以在常量时间内完成。然而需要说明的是，如果元素插入或删除操作是作用于链表中间，那将会是非常低效的。

## 3.2 List相关功能（Redis客户端命令）

LPUSH key value [value ...]

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。

返回值：插入后链表中元素的数量。

LPUSHX key value

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。

返回值：插入后链表中元素的数量。

LRANGE key start stop

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。

返回值：返回指定范围内元素的列表。

LPOP key

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。（弹出既从集合中弹出，集合中不在有这个数据了）

返回值：链表头部的元素。

LLEN key

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。

返回值：链表中元素的数量。

LREM key count value

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。

返回值：返回被删除的元素数量。

LSET key index value

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。

(对指定脚标的值进行设置)

LINDEX key index

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。

返回值：返回请求的元素，如果index超出范围，则返回nil。

(读出指定脚标的值)

LTRIM key start stop

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

（如果集合中有多个pivot则从前边遍历 碰到pivot就插入）

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。

返回值：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。

(在指定的某个value前或后插入一个新的value)

RPUSH key value [value ...]

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。

返回值：插入后链表中元素的数量。

RPUSHX key value 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。

返回值：插入后链表中元素的数量。

RPOP key 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。

返回值：链表尾部的元素。

RPOPLPUSH source destination

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。

返回值：返回弹出和插入的元素。

## 3.3 List相关功能（Jedis客户端API）

|  |
| --- |
| @Test  public void testList(){  //在链表的头部插入元素  // Long count = jedis.lpush("jedis-l-key-01", "zhangsan","lisi","wangwu","zhaoliu");  // System.out.println("插入的元素个数为： " + count);    List<String> res = jedis.lrange("jedis-l-key-01", 0, -1);  System.out.println("创建的链表中的元素为：");  for(String s:res){    System.out.println(s);  }    //在链表的中间插入新元素  /\* jedis.linsert("jedis-l-key-01", LIST\_POSITION.BEFORE, "lisi", "田七");    System.out.println("插入新元素之后的链表元素为： ");    List<String> res2 = jedis.lrange("jedis-l-key-01", 0, -1);  for(String s:res2){  System.out.println(s);  }\*/    Long count = jedis.lrem("jedis-l-key-01", 3, "zhangsan");  System.out.println("成功删除了" + count + "个zhangsan");  List<String> res3 = jedis.lrange("jedis-l-key-01", 0, -1);  for(String s:res3){  System.out.println(s);  }    } |

## 3.4 List数据结构应用场景

用List来实现一个简单的任务队列调度机制

|  |
| --- |
| public class TaskScheduler {  /\*\*  \* 任务生产者线程逻辑  \*  \* @author  \*  \*/  static class TaskProducer implements Runnable {  public void run() {  Jedis jedis = new Jedis("192.168.2.199");  System.out.println("任务生产者启动.......");  jedis.del("task-list");  while (true) {  UUID newTaskId = UUID.randomUUID();  jedis.lpush("task-list", newTaskId.toString());  System.out.println("生产者插入了一个新任务： " + newTaskId);  try {  int nextInt = new Random().nextInt(2);  Thread.sleep(1000 + nextInt \* 1000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  /\*\*  \* 任务处理者线程逻辑  \*  \* @author  \*  \*/  static class TaskWorker implements Runnable {  public void run() {  Jedis jedis = new Jedis("192.168.2.199");  System.out.println("任务处理者启动.......");  jedis.del("status-list");  while (true) {  try {  Thread.sleep(1000);  // 从task-list弹出一个任务，并插入到status-list队列中  String taskId = jedis.rpoplpush("task-list", "status-list");  // 处理任务的逻辑......  int nextInt = new Random().nextInt(9);  // 模拟任务处理成功的情况  if (nextInt % 4 != 0) {  // 从status-list队列中弹出这个处理成功的任务  jedis.lpop("status-list");  System.out.println(taskId + ": 处理成功，并从任务调度系统中彻底删除");  } else {  // 模拟任务处理失败的情况  jedis.rpoplpush("status-list", "task-list");  System.out.println(taskId + ": 处理失败，并从status-list中弹回task-list");  }  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  public static void main(String[] args) throws Exception {  new Thread(new TaskProducer()).start();  Thread.sleep(200);  new Thread(new TaskWorker()).start();  }  } |

# 4 Redis数据结构之Hash

## 4.1 概述：

Redis中的Hashes类型可以看成具有String Key和String Value的map容器。所以该类型非常适合于存储值对象的信息。如用户信息：Username、Password和Age等。每一个Hash可以存储4294967295个键值对。

## 4.2 Hash相关功能（Redis客户端命令）

* HSET key field value

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。

返回值：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

返回指定Key中指定Field的关联值。

返回值：返回参数中Field的关联值，如果参数中的Key或Field不存，返回nil。

* HSETNX key field value

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。

返回值：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。

* HEXISTS key field

判断指定Key中的指定Field是否存在。

返回值：1表示存在，0表示参数中的Field或Key不存在。

* HLEN key

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 ...]

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。

返回值：如果Key不存在，则将其视为空Hashes，并返回0，否则返回实际删除的Field数量。

* HINCRBY key field increment

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。返回值：运算后的值。

* HGETALL key

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。

返回值：Field/Value的列表。

* HKEYS key

返回指定Key的所有Fields名。

返回值：Field的列表。

* HVALS key

返回指定Key的所有Values名。

返回值：Value的列表。

* HMGET key field [field ...]

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。

返回值：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 ...]

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。

## 4.3 Hash相关功能（Jedis客户端API）

|  |
| --- |
| @Test  public void testHash(){    jedis.hset("jedis-h-key-01", "name", "zhangsan");    String name = jedis.hget("jedis-h-key-01", "name");  System.out.println(name);    HashMap<String, String> fields = new HashMap<String, String>();  fields.put("password", "123");  fields.put("age", "18");  jedis.hmset("jedis-h-key-01", fields);      Set<String> keys = jedis.hkeys("jedis-h-key-01");  System.out.println("所有的key如下：");  for(String key :keys){  System.out.println(key);    }    List<String> hvals = jedis.hvals("jedis-h-key-01");  System.out.println("所有的value如下： ");  for(String val:hvals){    System.out.println(val);  }    System.out.println("一次性取出所有的key-value对：");  Map<String, String> kvs = jedis.hgetAll("jedis-h-key-01");  Set<Entry<String, String>> entrySet = kvs.entrySet();  for(Entry<String, String> ent :entrySet){  System.out.println(ent.getKey() + " : " + ent.getValue());    }    } |

# 5 Redis数据结构之Set

## 5.1 概述

在Redis中，我们可以将Set类型看作为没有排序的字符串集合。Set可包含的最大元素数量是4294967295。

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。

## 5.2 Set相关功能（Redis客户端命令）

SADD key member [member ...]

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。

返回值：本次操作实际插入的成员数量.

SCARD key

获取Set中成员的数量。

返回值：返回Set中成员的数量，如果该Key并不存在，返回0。

SISMEMBER key member

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。

返回值：1表示已经存在，0表示不存在，或该Key本身并不存在。

SMEMBERS key

获取与该Key关联的Set中所有的成员。

返回值：返回Set中所有的成员。

SPOP key

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。

返回值：返回移除的成员，如果该Key并不存在，则返回nil。

SRANDMEMBER key

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。

返回值：返回随机位置的成员，如果Key不存在则返回nil。

SREM key member [member ...]

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。

返回值：从Set中实际移除的成员数量，如果没有则返回0。

SMOVE source destination member

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。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。

返回值：1表示正常移动，0表示source中并不包含参数成员。

SDIFF key [key ...] （求差集）

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。

返回值：差异结果成员的集合。

SDIFFSTORE destination key [key ...] （注意是结果成员）

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。

返回值：返回差异成员的数量。

SINTER key [key ...] （交集）

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。

返回值：交集结果成员的集合。

SINTERSTORE destination key [key ...]

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。

返回值：返回交集成员的数量。

SUNION key [key ...] （并集）

该命令将返回参数中所有Keys关联的Sets中成员的并集。

返回值：并集结果成员的集合。

SUNIONSTORE destination key [key ...]

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。

返回值：返回并集成员的数量。

## 5.3 Set相关功能（Jedis客户端API）

|  |
| --- |
| /\*\*  \* 测试set类型的数据结构  \*/  @Test  public void testSet(){    //创建一条set类型的数据并插入一些members  Long sadd = jedis.sadd("jedis-set-key-01", "java","c","c++","js","ruby");  System.out.println("创建了一个set类型的数据，并且插入"+sadd+"个成员");    //获取一条set类型的数据的成员  Set<String> members = jedis.smembers("jedis-set-key-01");  System.out.println("获取到的成员为：");  for(String m:members){  System.out.println(m);  }    jedis.sadd("jedis-set-key-02", "c#","c",".net","python","ruby");      //求两个集合的差集  Set<String> set12 = jedis.sdiff("jedis-set-key-01","jedis-set-key-02");    System.out.println("set01 减去 set02 的差集结果为：");    for(String s:set12){  System.out.println(s);  }    Set<String> set21 = jedis.sdiff("jedis-set-key-02","jedis-set-key-01");    System.out.println("set02 减去 set01 的差集结果为：");    for(String s:set21){  System.out.println(s);  }    //求两个集合的并集  Set<String> union12 = jedis.sunion("jedis-set-key-01","jedis-set-key-02");  System.out.println("set01和set02的并集结果为:");  for(String s:union12){  System.out.println(s);  }  //求两个集合的交集  } |

## 5.4 Set应用场景示例

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。

# 6 Redis数据结构之SortedSet

## 6.1 概述：

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，由于Sorted-Sets中的**成员在集合中的位置是有序的（排序）**，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。

## 6.2 SortedSet相关功能（Redis客户端命令）

ZADD key score member [score] [member]

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。

返回值：本次操作实际插入的成员数量。

ZINCRBY key increment member

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。

返回值：以字符串形式表示的新分数。

ZCARD key

获取与该Key相关联的Sorted-Sets中包含的成员总数量。

返回值：返回Sorted-Sets中的成员数量，如果该Key不存在，返回0。

ZCOUNT key min max

该命令用于获取**分数(score)**在min和max之间的成员数量。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。

返回值：分数指定范围内成员的数量。

ZRANGE key start stop [WITHSCORES]

该命令返回**排名**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...。

返回值：返回索引在start和stop之间的成员列表。

ZREVRANGE key start stop [WITHSCORES]

**该命令的功能和ZRANGE基本相同，唯一的差别在于顺序相反。**

返回值：返回指定的成员列表。

ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

该命令将返回**分数**在min和max范围内的成员，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**。

返回值：返回分数在指定范围内的成员列表。

ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]

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。

返回值：返回分数在指定范围内的成员列表。

ZRANK key member

该命令将返回参数中**指定成员的位置值**（按分数由低到高的顺序），其中0表示第一个成员，它是Sorted-Set中分数最低的成员。

返回值：如果该成员存在，则返回它的位置索引值。否则返回nil。

既 zeank key 成员 返回成员位置

ZREVRANK key member

该命令的功能和ZRANK基本相同，唯一的差别在于顺序相反

返回值：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

**获取指定成员的分数**。

返回值：如果该成员存在，以字符串的形式返回其分数，否则返回nil。

ZREM key member [member ...]

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。

返回值：实际被删除的成员数量。

ZREMRANGEBYRANK key start stop

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。

返回值：被删除的成员数量。

ZREMRANGEBYSCORE key min max

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。

返回值：被删除的成员数量。

## 6.3 SortedSet相关功能（Jedis客户端API）

|  |
| --- |
| /\*\*  \* 测试sortedset数据结构  \*/  @Test  public void testSortedSet(){    HashMap<String, Double> scoreMembers = new HashMap<String, Double>();  scoreMembers.put("zhangsan", 100.00);  scoreMembers.put("lisi", 90.00);  scoreMembers.put("wangwu", 80.00);  scoreMembers.put("zhaoliu", 70.00);  scoreMembers.put("tianqi", 60.00);    jedis.zadd("jedis-zset-key-01", scoreMembers);    //获取指定名次区间的所有成员，顺序为分数的由低到高  Set<String> allMembers = jedis.zrange("jedis-zset-key-01", 0, -1);  System.out.println("所有的成员为：");  for(String m: allMembers){    System.out.println(m);  }      System.out.println("给赵六加40分之后，成员的排名情况为：");  //给指定的成员增加分数  jedis.zincrby("jedis-zset-key-01", 40, "zhaoliu");    //获取指定名次区间的所有成员及其关联的分数，顺序为分数的由低到高  Set<Tuple> zrangeWithScores = jedis.**zrangeWithScores**("jedis-zset-key-01", 0, -1);  for(Tuple t:zrangeWithScores){    System.out.println(t.getElement() + " : " +t.getScore());  }      System.out.println("按照分数由高到低的顺序打印排行榜：");    //获取指定名次区间的所有成员及其关联的分数，顺序为分数的由高到低  Set<Tuple> zrevrangeWithScores = jedis.**zrevrangeWithScores**("jedis-zset-key-01", 0, -1);  for(Tuple t:zrevrangeWithScores){    System.out.println(t.getElement() + " : " +t.getScore());  }      //扩展练习其他方法      } |

## 6.4 SortedSet应用范围

各种排行榜应用：

微博

游戏排行榜



# 7 通用key的操作

## 7.1 概述

不区分类型，针对key的通用操作

## 7.2 通用key操作（Redis客户端命令）

KEYS pattern

**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 ...]

**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)。

返回值：实际被删除的Key数量。

EXISTS key

**判断指定键是否存在**。

返回值：1表示存在，0表示不存在。

MOVE key db

**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。

返回值：移动成功返回1，否则0。

RENAME key newkey

**为指定指定的键重新命名**，如果参数中的两个Keys的名字相同，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

**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。

返回值：1表示修改成功，否则0。

PERSIST key

**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**。

返回值：1表示Key的过期时间被移除，0表示该Key不存在或没有过期时间。

EXPIRE key seconds

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。

返回值：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

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。

返回值：1表示超时被设置，0则表示Key不存在，或不能被设置。

TTL key

获取该键所剩的超时描述。

返回值：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

从当前打开的数据库中随机的返回一个Key。

返回值：返回的随机键，如果该数据库是空的则返回nil。

TYPE key

**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。

返回值：返回的字符串为string、list、set、hash和zset，如果key不存在返回none。

## 7.2 通用key操作（Jedis客户端API）

|  |
| --- |
| /\*\*  \* 测试针对key的通用操作  \*/  @Test  public void testGeneralKey(){    Set<String> keys = jedis.keys("\*");  for(String key:keys){  System.out.println(key);  }  } |

# 8 Redis的持久化

Redis将内存存储和持久化存储相结合，即可提供数据访问的高效性，又可保证数据存储的安全性

## 8.1 Redis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(append only file)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). 同时应用AOF和RDB。

4). 无持久化：

可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

## 8.2 RDB机制的优势和劣势：

RDB存在哪些优势呢？

1). **数据的备份和恢复非常方便**，**因为一个数据库只有一个持久化文件**

2). **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。

3). 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).**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**。

## 8.3 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**。

2).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**。

3). 如果日志过大，**Redis可以自动启用rewrite机制迅速“瘦身”**(**也可手动触发aof的rewrite操作**，命令： **bgrewriteaof)**

4). AOF日志格式清晰、易于理解，很容易用AOF日志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**。

## 8.4 持久化相关配置

1. RDB快照方式:

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。

2. AOF日志文件方式：

* AOF日志持久化机制的开启：

将appendonly no 改为

appendonly yes

* AOF同步方式的配置：

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no #从不同步。高效但是数据不会被持久化。

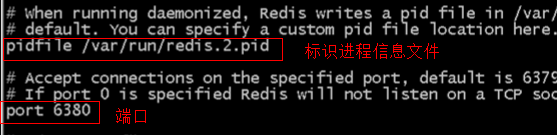
# 9 Redis集群

## 9.1 Redis主从复制集群

由于要在一个Linux服务器上，安装两个redis，所以

创建**两个redis.conf文件**（redis.conf、redis.2.conf）

**更改**其中一个文件的 **端口号** 和 **标识进程信息文件** 配置

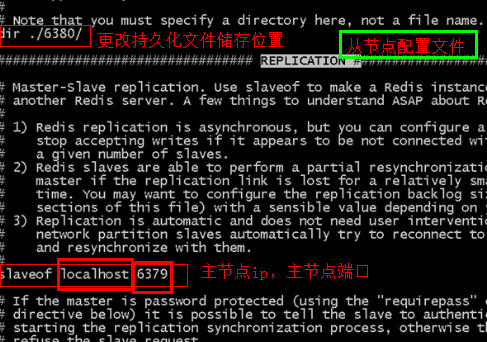




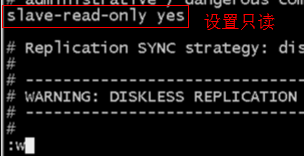
表示我就算关闭了窗口进程也不关闭（**后台进程**）

将命令 输出的标准输出 重定向到一个文件中（1表示标准输出，此文件相当于黑洞文件，相当于把输出内容丢掉了）（2表示错误输出， &1表示引用1的位置）最后一个&表示让程序在后台执行

设置从节点redis.conf配置文件



**不要忘记将注释打开**

 **默认就是只读**

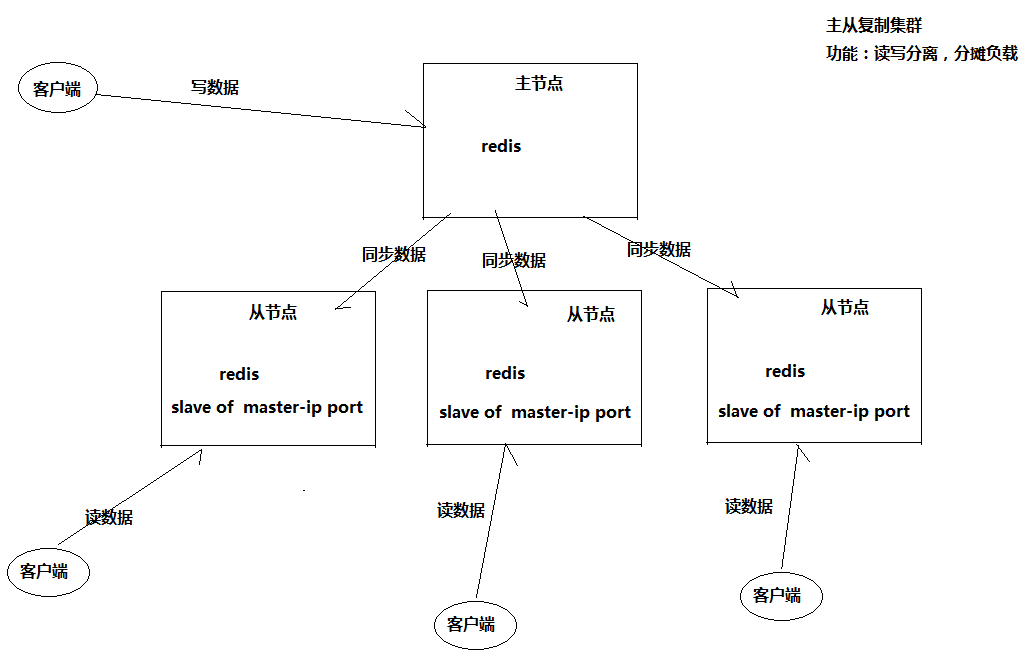
**文件更改后 创建持久化文件目录 （如：mkdir ./6380）**

**启动另一个端口redis服务器**

**实现方式：**选择一台redis服务器作为master节点(负责写操作)，另外一台或多台服务器作为slave节点(负责读操作)，slave节点上的数据完全由master节点同步过来

**作用：**降低单节点redis服务器的读写负载，将读写分离到不同的服务器

提高数据的可用性



**配置方式：**

Master节点不用作任何变动

Slave节点上的配置文件redis.conf需要修改如下配置项

|  |
| --- |
| # slaveof <masterip> <masterport>  改为  slaveof 127.0.0.1 6379  最好让从服务器配成只读属性  slave-read-only yes |
|  |

## 9.2 Redis数据分片（Sharding）集群

**实现方式：**由若干台互不相干的redis服务器组成一个集群，互相独立，由集群的前置节点或者客户端实现将数据分散插入到集群中的各服务器上

**作用：**扩大数据存储的容量，降低单台服务器负载

使用上边从服务器作为零一台redis服务器，只需关闭从服务器中配置文件中指定主服务器的配置（加上了注释）



**实现示例：**

由jedis客户端实现数据分片

|  |
| --- |
| public class ShardClusterTest {    public static void main(String[] args) {    //poolConfig是连接池的配置参数  GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();  //shards是分片集群中所有分片服务器信息列表,JedisShardInfo是分片服务器信息  ArrayList<JedisShardInfo> shards = new ArrayList<JedisShardInfo>();    //将集群中的两台shard服务器信息封装到两个JedisShardInfo对象中  JedisShardInfo shard1 = new JedisShardInfo("192.168.2.199", 6379);  JedisShardInfo shard2 = new JedisShardInfo("192.168.2.199", 6380);    //将分片服务器信息对象添加到分片服务器信息列表shards中  shards.add(shard1);  shards.add(shard2);    //创建一个带数据分片功能的jedis连接池  ShardedJedisPool shardedJedisPool = new ShardedJedisPool(poolConfig, shards);    //从连接池中获取一个(带数据分片功能的)jedis连接  ShardedJedis jedis = shardedJedisPool.getResource();    for(int i=0;i<1000;i++){  jedis.set("string-key-"+i, "1000" + i);  }  jedis.close();    shardedJedisPool.close();    }  } |