1. Nosql定义和分类
   1. 什么是NoSql

NoSql，叫非关系型数据库，它的全名Not only sql。为了解决高并发、高可用、高可扩展，大数据存储等一系列问题而产生的数据库解决方案，就是NoSql。它不能替代关系型数据库，只能作为关系型数据库的一个良好补充。

* 1. NoSql的分类
     1. 键值(Key-Value)存储数据库

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

典型应用： 内容缓存，主要用于处理大量数据的高访问负载。

数据模型： 一系列键值对

优势： 快速查询

劣势： 存储的数据缺少结构化

* + 1. 列存储数据库

相关产品：Cassandra, **HBase**, Riak

典型应用：分布式的文件系统

数据模型：以列簇式存储，将同一列数据存在一起

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

* + 1. 文档型数据库

相关产品：CouchDB、**MongoDB**

典型应用：Web应用（与Key-Value类似，Value是结构化的）

数据模型： 一系列键值对

优势：数据结构要求不严格

劣势： 查询性能不高，而且缺乏统一的查询语法

* + 1. 图形(Graph)数据库

相关数据库：Neo4J、InfoGrid、Infinite Graph

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法。

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

1. Redis定义,历史和应用场景
   1. 什么是redis

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

键值类型：

String字符类型

map散列类型

list列表类型

set集合类型

sortedset有序集合类型

* 1. redis历史发展

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

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

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

* 1. redis的应用场景

缓存（数据查询、短连接、新闻内容、商品内容等等）。（**最多使用**）

分布式集群架构中的session分离。

聊天室的在线好友列表。

任务队列。（秒杀、抢购、12306等等）

应用排行榜。

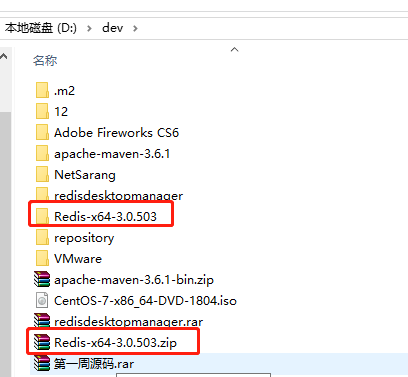
网站访问统计。

数据过期处理（可以精确到毫秒）

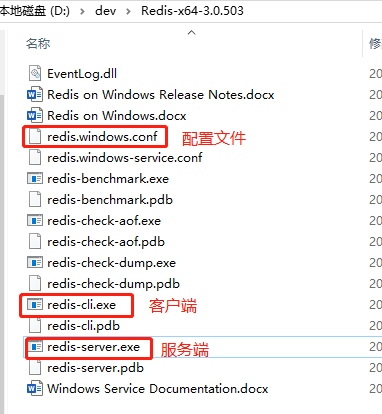
1. Redis下载和安装
   1. windows版下载和安装

 window版下载地址：https://github.com/MicrosoftArchive/redis/releases

下载Redis For  window X64.zip,解压到指定目录



* 1. Redis的目录结构



Redis.windows.conf:redis的配置文件。文件内可以声明redis的端口号，database的数量，密码，是否后台运行，持久化机制，日志等信息。

Redis-server.exe：redis服务端，可以启动redis服务。

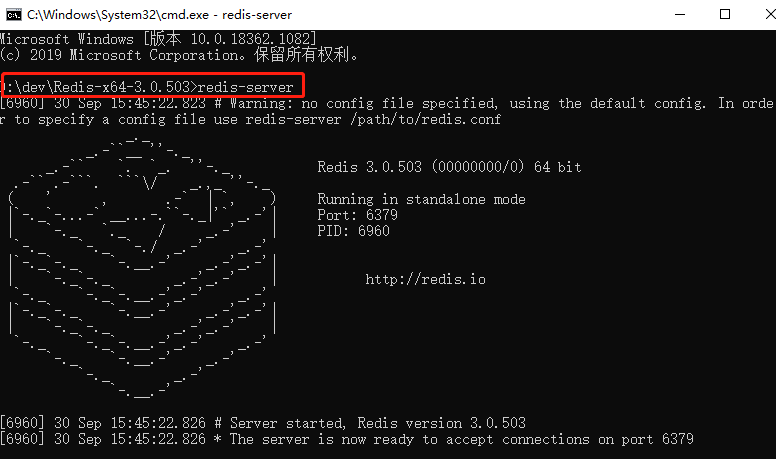
Redis-cli.exe: redis自带客户端，启动以后即可连接服务端，通过以命令的形式向redis服务中存取数据，也可以关闭redis服务。

* + 1. 启动redis服务端

双击 redis-server.exe ，启动成功，出现如下界面。端口号6379，pid：16316，如果关闭当前窗口，相当于关闭redis服务。



或者以命令行的方式启动redis，在当前目录下输入redis-server即可启动redis服务

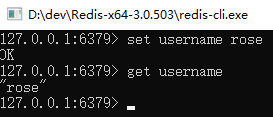


* + 1. 启动redis客户端

双击redis-cli.exe即可启动客户端：如下图表示，则已经连上服务端，可以通过命令存取数据了。

如果服务端设置了密码，在运行客户端连接服务端时，需要加上密码

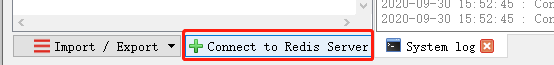
./redis-cli -a 123456 -------123456为密码



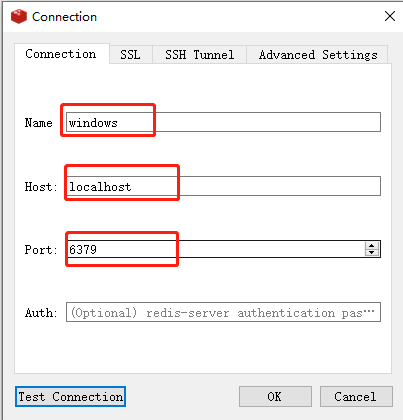
* + 1. 图形化界面客户端

RedisDesktopManager以图形化界面的方式操作redis中的数据。安装redis桌面管理工具。

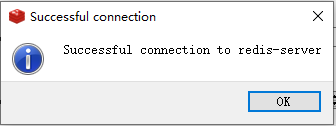
建立连接：



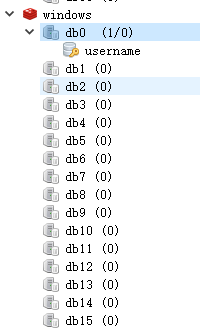
输入名称，主机ip，默认端口号6379，即可连接redis服务器。



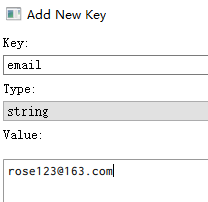
可以测试连接，如果连接成功，会有如下弹框：

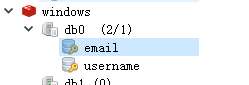


Redis默认有16个数据库，如果不指定，默认使用database0



向第一个数据库中添加，修改，删除数据。



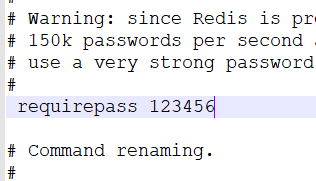




* + 1. 指定配置启动redis服务

Redis启动时可以不指定配置文件，会使用默认配置启动。也可以指定配置文件。如果我们修改了配置文件的内容，希望指定以某个配置文件启动，则使用如下方法启动

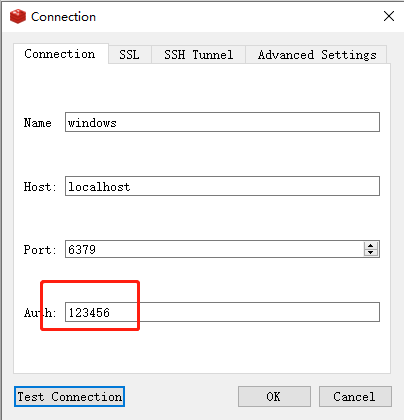
复制配置文件redis.windows.conf,改名为redis.conf，修改使用密码123456登录。



以该配置文件运行redis服务：

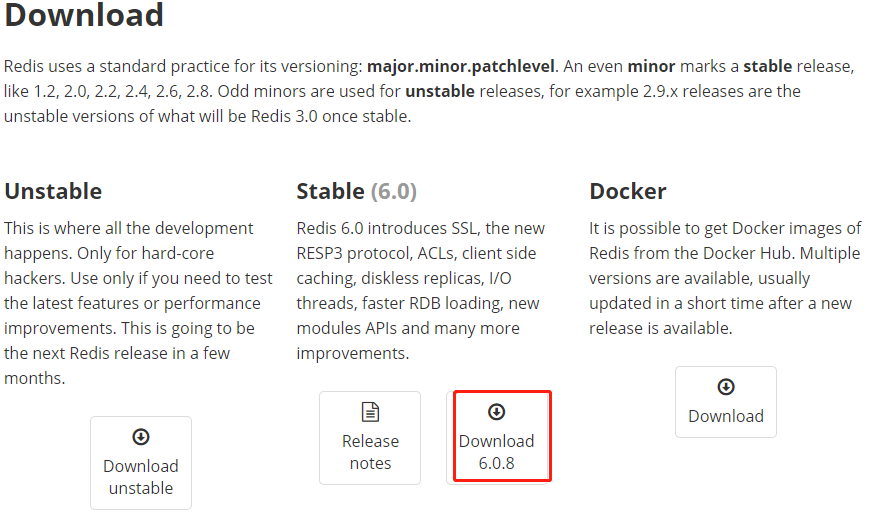


此时，再通过图形化客户端连接服务时，必须指定密码才可以，否则会连接失败。



* 1. Linux版下载和安装
     1. 下载

官网：<https://redis.io/download>，选择稳定版本。

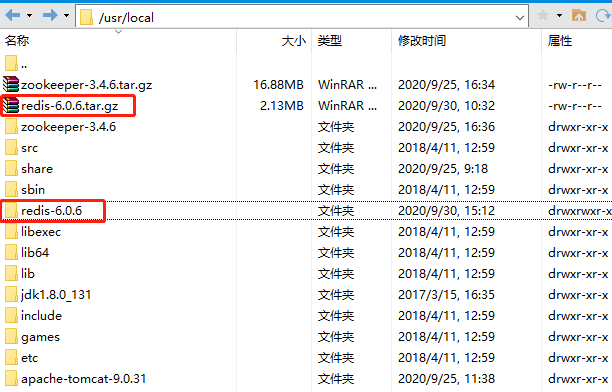


* + 1. 安装

1.解压

通过xftp把redis安装包上传到指定文件夹。解压：

tar -zxvf redis-6.0.6.tar.gz



2．安装

由于redis依赖c语言环境，所以先安装gcc

[root@localhost ~]# yum install gcc

如果是线上下载安装包的话，先进入目录 usr/local，下载安装包

[root@localhost local]# wget http://download.redis.io/releases/redis-6.0.6.tar.gz

解压

[root@localhost local]# tar -xvf redis-6.0.6.tar.gz

安装好了c语言环境以后，需要编译redis，进入解压后的目录

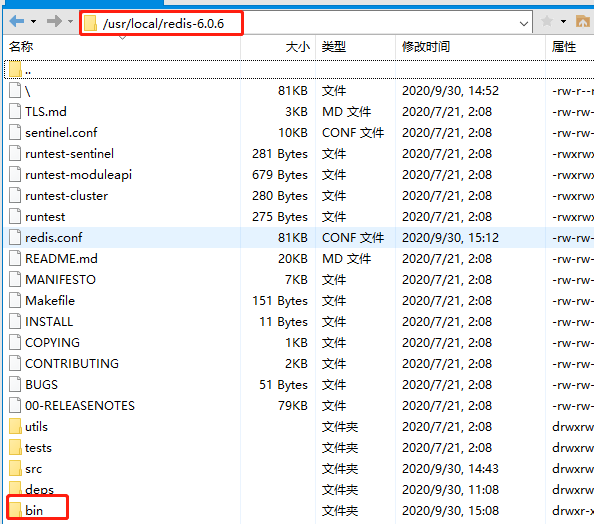
[root@localhost local]# cd /usr/local/redis-6.0.6/

编译

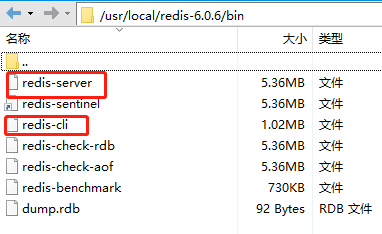
[root@localhost local]# **make**

安装，如果不指明安装路径，就会在当前路径下安装，指明安装路径，会在指明的路径下生成bin文件夹，bin里面是redis的命令

[root@localhost redis-6.0.1]# make PREFIX=/usr/local/redis.6.0.6 install



进入bin



注意：如果在编译的时候报错如下：等等，太长了，不列举了。

make[1]: \*\*\* [server.o] 错误 1

make[1]: 离开目录“/usr/redis-6.0.6/src”

make: \*\*\* [all] 错误 2

server.c:2402:11: 错误：‘struct redisServer’没有名为‘assert\_file’的成员

server.assert\_file = "<no file>";

^

server.c:2403:11: 错误：‘struct redisServer’没有名为‘assert\_line’的成员

server.assert\_line = 0;

^

server.c:2404:11: 错误：‘struct redisServer’没有名为‘bug\_report\_start’的成员

server.bug\_report\_start = 0;

^

解决办法：升级gcc版本

[root@localhost redis-6.0.6]# gcc -v # 查看gcc版本

[root@localhost redis-6.0.6]# yum -y install centos-release-scl # 升级到9.1版本

[root@localhost redis-6.0.6]# yum -y install devtoolset-9-gcc devtoolset-9-gcc-c++ devtoolset-9-binutils

[root@localhost redis-6.0.6]# scl enable devtoolset-9 bash

以上为临时启用，如果要长期使用gcc 9.1的话：

[root@localhost redis-6.0.6]# echo "source /opt/rh/devtoolset-9/enable" >>/etc/profile

在执行编译就没有问题了，安装成功，会提示：



执行make test，可能会出现如下错误：

[root@localhost redis-6.0.6]# make test

cd src && make test

make[1]: 进入目录“/usr/redis-6.0.6/src”

CC Makefile.dep

make[1]: 离开目录“/usr/redis-6.0.6/src”

make[1]: 进入目录“/usr/redis-6.0.6/src”

You need tcl 8.5 or newer in order to run the Redis test

make[1]: \*\*\* [test] 错误 1

make[1]: 离开目录“/usr/redis-6.0.6/src”

make: \*\*\* [test] 错误 2

解决办法：

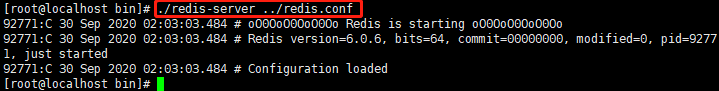
[root@localhost redis-6.0.6]# yum install tcl

[root@localhost redis-6.0.6]# make test

重新测试成功。

* + 1. 运行redis

以某个配置文件启动redis服务，一定要注意配置文件的路径

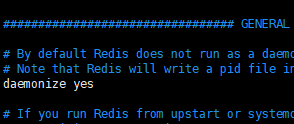


* + 1. 后台启动

上面redis的启动方式，是前端启动，一关闭客户端，redis的服务也就停掉了，所以这种启动方式非常不友好。我们可以修改配置文件中的启动方式：

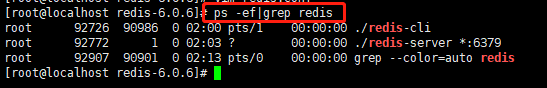
[root@localhost redis-6.0.6]# vim redis.conf

进到redis.conf文件里，然后找到daemonize no把no改为yes

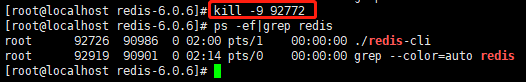


启动的时候指定修改之后的配置文件即可。

* + 1. 查看是否启动成功



* + 1. 关闭redis服务



或者：

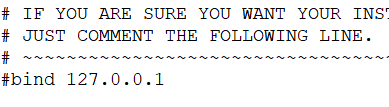


如果redis设置了密码，则以下面方式关闭，123456为密码

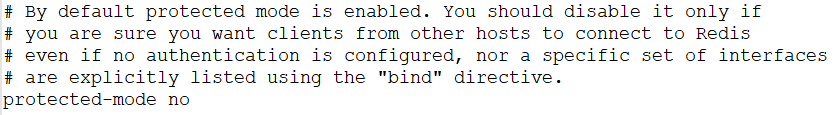


* + 1. 使用桌面管理工具连接Linux redis

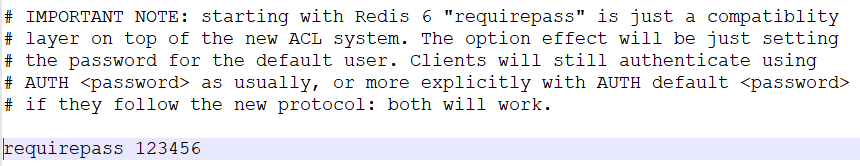
修改redis配置文件，注释掉bind 127.0.0.1



保护模式关闭：



设置密码：



* + 1. 以服务方式启动redis

1. 把redis加入service服务

vim /lib/systemd/system/redis.service

1. 写入

[Unit]

Description=redis

After=network.target

[Service]

Type=forking

PIDFile=/var/run/redis\_6379.pid

ExecStart=/usr/local/redis-6.0.6/src/redis-server /usr/local/redis-6.0.6/etc/redis.conf

ExecReload=/bin/kill -s HUP $MAINPID

ExecStop=/bin/kill -s QUIT $MAINPID

PrivateTmp=true

[Install]

WantedBy=multi-user.target

**注意路径问题，以及PIDFile的路径对应的是redis-6.0.6/redis.conf里面的PIDFile**

保存，退出。

1. 运行以下

[root@localhost redis-6.0.6]# systemctl enable redis.service # 加入开机启动

[root@localhost redis-6.0.6]# systemctl is-enabled redis.service # 查看开机是否启动成功

enabled

[root@localhost redis-6.0.6]# systemctl start redis #开启redis服务

[root@localhost redis-6.0.6]# systemctl status redis //查看redis运行状态

* 1. 清缓存命令

flushall

把缓存键值对清空。

1. Redis的String数据类型的赋值取值删除
   1. 命令
      1. 赋值

语法：SET key value

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

* + 1. 取值

语法：GET key

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

* + 1. 取值并赋值

语法：GETSET key value

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

* + 1. 设置/获取多个键值

*语法：*

*MSET key value [key value …]*

*MGET key [key …]*

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

* + 1. 删除

语法：DEL key

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

* + 1. 数值增减
* 递增数字

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

语法：INCR key

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

* 增加指定的整数

语法：INCRBY key increment

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

* 递减数值

语法：DECR key

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

* 减少指定的整数

语法：DECRBY key decrement

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

* + 1. 其它命令(自学)

#### 向尾部追加值

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

*语法：APPEND key value*

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

#### 获取字符串长度

STRLEN命令返回键值的长度，如果键不存在则返回0。

*语法：STRLEN key*

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

* 1. 应用
     1. 自增主键

商品编号、订单号采用string的递增数字特性生成。

定义商品编号key：items:id

192.168.101.3:7003> INCR items:id

(integer) 2

192.168.101.3:7003> INCR items:id

(integer) 3

1. Redis的Hash数据类型的赋值取值删除

散列类型

* 1. 使用string的问题

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

保存、更新：

User对象 🡪 json(string) 🡪 redis

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

* 1. redis hash介绍

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



* 1. 命令
     1. 赋值

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

* *一次只能设置一个字段值*

*语法：HSET key field value*

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

* *一次可以设置多个字段值*

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

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

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

*语法：HSETNX key field value*

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

* + 1. 取值
* *一次只能获取一个字段值*

*语法：HGET key field*

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

* *一次可以获取多个字段值*

*语法：HMGET key field [field ...]*

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

* *获取所有字段值*

*语法：HGETALL key*

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

* + 1. 删除字段

可以删除一个或多个字段，返回值是被删除的字段个数

*语法：HDEL key field [field ...]*

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

* + 1. 增加数字

*语法：HINCRBY key field increment*

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

* + 1. 其它命令(自学)
       1. 判断字段是否存在

*语法：HEXISTS key field*

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

* + - 1. 只获取字段名或字段值

*语法：*

*HKEYS key*

*HVALS key*

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

* + - 1. 获取字段数量

*语法：HLEN key*

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

* 1. 应用
     1. 存储商品信息
* 商品字段

【商品id、商品名称、商品描述、商品库存、商品好评】

* 定义商品信息的key

商品1001的信息在 Redis中的key为：[items:1001]

* 存储商品信息

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

* 获取商品信息

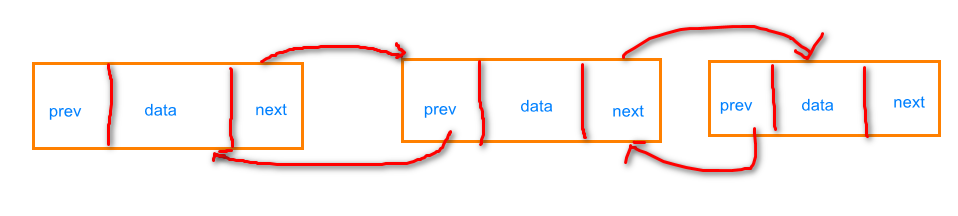
|  |
| --- |
| 192.168.101.3:7003> HGET items:1001 id  "3"  192.168.101.3:7003> HGETALL items:1001  1) "id"  2) "3"  3) "name"  4) "apple"  5) "price"  6) "999.9" |

1. Redis的List数据类型的赋值取值删除
   1. Arraylist和linkedlist的区别

Arraylist是使用数组来存储数据，特点：查询快、增删慢

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

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



* 1. 命令
     1. 向列表两端增加元素
* *向列表左边增加元素*

*语法：LPUSH key value [value ...]*

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

* *向列表右边增加元素*

语法：*RPUSH key value [value ...]*

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

* + 1. 查看列表

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

*语法：LRANGE key start stop*

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

* + 1. 从列表两端弹出元素

LPOP命令从列表左边弹出一个元素，会分两步完成：

第一步是将列表左边的元素从列表中移除

第二步是返回被移除的元素值。

*语法：*

*LPOP key*

*RPOP key*

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

* + 1. 获取列表中元素的个数

*语法：LLEN key*

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

* + 1. 其它命令(自学)
       1. 删除列表中指定的值

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

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

*语法：LREM key count value*

* + - 1. 设置指定索引的元素值
* 获得指定索引的元素值

语法：LINDEX key index

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

* 设置指定索引的元素值

语法：LSET key index value

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

* + - 1. 只保留列表指定片段

指定范围和LRANGE一致

*语法：LTRIM key start stop*

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

* + - 1. 向列表中插入元素

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

*语法：LINSERT key BEFORE|AFTER pivot value*

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

* + - 1. 将元素从一个列表转移到另一个列表中

*语法：RPOPLPUSH source destination*

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

* 1. 应用
     1. 商品评论列表

##### 思路：

在Redis中创建商品评论列表

用户发布商品评论，将评论信息转成json存储到list中。

用户在页面查询评论列表，从redis中取出json数据展示到页面。

##### 定义商品评论列表key：

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

|  |
| --- |
| 192.168.101.3:7001> LPUSH items:comment:1001 '{"id":1,"name":"very good!","date":1430295077289}' |

1. Redis的Set数据类型的赋值取值删除

集合类型

集合类型：无序、不可重复

列表类型：有序、可重复

* 1. 命令
     1. 增加/删除元素

*语法：SADD key member [member ...]*

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

*语法：SREM key member [member ...]*

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

* + 1. 获得集合中的所有元素

*语法：SMEMBERS key*

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

* + 1. 判断元素是否在集合中

*语法：SISMEMBER key member*

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

* 1. 运算命令
     1. 集合的差集运算 A-B

属于A并且不属于B的元素构成的集合。



*语法：SDIFF key [key ...]*

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

* + 1. 集合的交集运算 A ∩ B

属于A且属于B的元素构成的集合。



语法：SINTER key [key ...]

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

* + 1. 集合的并集运算 A ∪ B

属于A或者属于B的元素构成的集合



语法：SUNION key [key ...]

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

* 1. 其它命令(自学)
     1. 获得集合中元素的个数

*语法：SCARD key*

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

* + 1. 从集合中弹出一个元素

注意：由于集合是无序的，所有SPOP命令会从集合中随机选择一个元素弹出

*语法：SPOP key*

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

1. Redis的SortedSet数据类型的赋值取值删除
   1. Sortedset

Sortedset又叫zset

Sortedset是有序集合，可排序的，但是唯一。

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

* 1. 命令
     1. 增加元素

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

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

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

* + 1. 获取元素的分数

语法：ZSCORE key member

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

* + 1. 删除元素

移除有序集key中的一个或多个成员，不存在的成员将被忽略。

当key存在但不是有序集类型时，返回一个错误。

语法：ZREM key member [member ...]

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

* + 1. 获得排名在某个范围的元素列表

获得排名在某个范围的元素列表

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

*语法：ZRANGE key start stop [WITHSCORES]*

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

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

*语法：ZREVRANGE key start stop [WITHSCORES]*

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

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

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

* + 1. 获取元素的排名
* 从小到大

*语法：ZRANK key member*

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

* 从大到小

*语法：ZREVRANK key member*

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

* 1. 其它命令(自学)
     1. 获得指定分数范围的元素

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

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

* + 1. 增加某个元素的分数

返回值是更改后的分数

*语法：ZINCRBY key increment member*

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

* + 1. 获得集合中元素的数量

*语法：ZCARD key*

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

* + 1. 获得指定分数范围内的元素个数

*语法：ZCOUNT key min max*

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

* + 1. 按照排名范围删除元素

*语法：ZREMRANGEBYRANK key start stop*

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

* + 1. 按照分数范围删除元素

*语法：ZREMRANGEBYSCORE key min max*

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

* 1. 应用
     1. 商品销售排行榜

需求：根据商品销售量对商品进行排行显示

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

**写入商品销售量**：

* 商品编号1001的销量是9，商品编号1002的销量是10

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

* 商品编号1001的销量加1

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

* 商品销量前10名：

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



1. Keys命令
   1. 常用命令
      1. keys

返回满足给定pattern 的所有key

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

* + 1. exists

确认一个key 是否存在

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

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

* + 1. del

删除一个key

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

* + 1. rename

重命名key

示例：age 成功的被我们改名为age\_new 了

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

* + 1. type

返回值的类型

示例：这个方法可以非常简单的判断出值的类型

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

* + 1. 设置key的生存时间

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

|  |
| --- |
| *EXPIRE key seconds 设置key的生存时间（单位：秒）key在多少秒后会自动删除*  *TTL key 查看key剩余的生存时间*  *PERSIST key 清除生存时间 -1*  *PEXPIRE key milliseconds 生存时间设置单位为：毫秒* |

*-1:清除生存时间,redis一直存在该key*

*-2:到期,redis没有该key*

*正整数:剩余的生存时间*

*例子：*

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

1. SpringBoot整合Redis
   1. 新建SpringBoot工程,引入Web,Redis的场景启动器

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  </dependency> |

* 1. application.yml配置redis地址

spring.redis.host

|  |
| --- |
| spring:  redis:  host: 127.0.0.1 |

* 1. 在使用处注入StringRedisTemplate或者RedisTemplate
     1. StringRedisTemplate

处理字符串相关操作

|  |
| --- |
| @Autowired  **private** StringRedisTemplate stringRedisTemplate;  @Test  **void** contextLoads() {  ValueOperations<String, String> opsForValue = stringRedisTemplate.opsForValue();  opsForValue.set("msg", "Hello Redis");  String msg = opsForValue.get("msg");  System.***out***.println(msg);  }    @Test  **void** test2() **throws** Exception {  ListOperations<String, String> opsForList = stringRedisTemplate.opsForList();  opsForList.leftPush("ls", "list1");  opsForList.leftPush("ls", "list2");  opsForList.leftPush("ls", "list1");  opsForList.leftPush("ls", "list4");  List<String> list = opsForList.range("ls", 0, -1);  System.***out***.println(list);  }    @Test  **void** test3() **throws** Exception {  SetOperations<String, String> opsForSet = stringRedisTemplate.opsForSet();  Long add = opsForSet.add("set", "set1", "set1", "set1", "set1", "set1", "set5");  System.***out***.println(add);  Set<String> members = opsForSet.members("set");  System.***out***.println(members);  }  @Test  **void** test4() **throws** Exception {  HashOperations<String, Object, Object> opsForHash = stringRedisTemplate.opsForHash();  opsForHash.put("person", "name", "zs");  opsForHash.put("person", "age", "20");  opsForHash.put("person", "sex", "nan");  Object object = opsForHash.get("person", "name");  System.***out***.println(object);  object = opsForHash.get("person", "age");  System.***out***.println(object);  } |
| @RequestMapping("/hello4")  @ResponseBody  **public** String hello4() {  ZSetOperations<String, String> opsForZSet = stringRedisTemplate.opsForZSet();  opsForZSet.add("score", "liubei", 90);  opsForZSet.add("score", "guanyu", 92);  opsForZSet.add("score", "zhangfei", 60);  opsForZSet.add("score", "wangping", 99);  Set<TypedTuple<String>> set = opsForZSet.rangeByScoreWithScores("score", 90, 99);  **for** (TypedTuple<String> typedTuple : set) {  System.***out***.println(typedTuple.getValue()+":"+typedTuple.getScore());  }  **return** "hello boot!";  } |
| liubei:90.0  guanyu:92.0  wangping:99.0 |

* + 1. RedisTemplate

处理对象相关操作

* + - 1. redisTemplate默认如果保存对象,使用jdk序列化机制

|  |
| --- |
| @Autowired  **private** RedisTemplate redisTemplate;  @Test  **void** test6() **throws** Exception {  ValueOperations<String, Person> opsForValue =redisTemplate.opsForValue();  Person person2 = mapper.selectByPrimaryKey(1);  System.***out***.println(person2);  opsForValue.set("p", person2);  Person person = opsForValue.get("p");  System.***out***.println(person);  } |
| 127.0.0.1:6379> get "\xac\xed\x00\x05t\x00\x01p"  "\xac\xed\x00\x05sr\x00\x17com.tjetc.domain.Personp3\\+\x04\_\x9d\xb5\x02\x00\x04  L\x00\x03aget\x00\x13Ljava/lang/Integer;L\x00\x02idq\x00~\x00\x01L\x00\x04namet\  x00\x12Ljava/lang/String;L\x00\tphotopathq\x00~\x00\x02xpsr\x00\x11java.lang.Int  eger\x12\xe2\xa0\xa4\xf7\x81\x878\x02\x00\x01I\x00\x05valuexr\x00\x10java.lang.N  umber\x86\xac\x95\x1d\x0b\x94\xe0\x8b\x02\x00\x00xp\x00\x00\x00\x16sq\x00~\x00\x  04\x00\x00\x00\x01t\x00\x06\xe5\x88\x98\xe6\x9c\x88t\x00\x0cupload/1.jpg" |

* + - 1. 改变jdk的序列化规则

##### 在配置类中配置自定义序列化规则

|  |
| --- |
| @Bean  **public** RedisTemplate<Object, Person> personRedisTemplate(RedisConnectionFactory redisConnectionFactory){  RedisTemplate<Object, Person> template = **new** RedisTemplate<>();  template.setConnectionFactory(redisConnectionFactory);  Jackson2JsonRedisSerializer<Person> serializer = **new** Jackson2JsonRedisSerializer<>(Person.**class**);  template.setDefaultSerializer(serializer);  **return** template;  } |

##### 使用

|  |
| --- |
| @Autowired  **private** RedisTemplate personRedisTemplate;  @Test  **void** test6() **throws** Exception {  ValueOperations<String, Person> opsForValue = personRedisTemplate.opsForValue();  Person person2 = mapper.selectByPrimaryKey(1);  System.***out***.println(person2);  opsForValue.set("p", person2);  Person person = opsForValue.get("p");  System.***out***.println(person);  } |
| 127.0.0.1:6379> get "\"p\""  "{\"id\":1,\"name\":\"\xe5\x88\x98\xe6\x9c\x88\",\"age\":22,\"photopath\":\"uplo  ad/1.jpg\"}" |

如果在一个系统中，需要序列化不同的对象，则将参数改为Object即可。

|  |
| --- |
| @Configuration public class RedisConfig {   */\*\*  \* 设置序列化  \* 重新设置redisTemplate的序列化  \* @param redisConnectionFactory  \* @return  \*/* @Bean  public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory){  RedisTemplate<String, Object> template = new RedisTemplate<>();  *//设置连接工厂* template.setConnectionFactory(redisConnectionFactory);  *//指定序列化方式  //将对象序列化成json* Jackson2JsonRedisSerializer serializer = new Jackson2JsonRedisSerializer<>(Object.class);  template.setDefaultSerializer(serializer);  template.setKeySerializer(new StringRedisSerializer());  return template;  }  } |

* + 1. 手动解析json

转成json串用StringRedisTemplate存储

|  |
| --- |
| 单个对象:  @Test  **void** test7() **throws** Exception {  Person person2 = mapper.selectByPrimaryKey(1);  System.***out***.println(person2);  ObjectMapper objectMapper=**new** ObjectMapper();  String string = objectMapper.writeValueAsString(person2);  System.***out***.println(string);  ValueOperations<String, String> opsForValue = stringRedisTemplate.opsForValue();  opsForValue.set("p", string);  String value= opsForValue.get("p");  System.***out***.println(value);  Person person = objectMapper.readValue(value, Person.**class**);  System.***out***.println(person);  } |
| 127.0.0.1:6379> get "p"  "{\"id\":1,\"name\":\"\xe5\x88\x98\xe6\x9c\x88\",\"age\":22,\"photopath\":\"uplo  ad/1.jpg\"}" |
| 集合:  @Test  **void** test5() **throws** Exception {  List<Person> list = mapper.selectByExample(**null**);  **for** (Person person : list) {  System.***out***.println(person.getId());  }  ObjectMapper objectMapper = **new** ObjectMapper();  String string = objectMapper.writeValueAsString(list);  System.***out***.println(string);  ValueOperations<String, String> opsForValue = stringRedisTemplate.opsForValue();  opsForValue.set("list",string );  String string2 = opsForValue.get("list");  List<Person> list2 = objectMapper.readValue(string2, **new** TypeReference<List<Person>>(){});  System.***out***.println("=================");  **for** (Person person : list2) {  System.***out***.println(person);  }  } |
| 127.0.0.1:6379> get "list"  "[{\"id\":1,\"name\":\"\xe5\x88\x98\xe6\x9c\x88\",\"age\":22,\"photopath\":\"upl  oad/1.jpg\"},{\"id\":2,\"name\":\"\xe5\x8e\x86\xe5\x8f\xb2\",\"age\":20,\"photop  ath\":\"upload/1.jpg\"},{\"id\":3,\"name\":\"\xe5\x95\x8a\xe6\x89\x93\xe7\xae\x9  7\",\"age\":11,\"photopath\":\"upload/2.jpg\"}]" |

1. 持久化
   1. Redis高可用概述

我们知道，在web服务器中，高可用是指服务器可以正常访问的时间，衡量的标准是在多长时间内可以提供正常服务（99.9%、99.99%、99.999% 等等）。但是在Redis语境中，高可用的含义似乎要宽泛一些，除了保证提供正常服务(如主从分离、快速容灾技术)，还需要考虑数据容量的扩展、数据安全不会丢失等。

在Redis中，实现高可用的技术主要包括持久化、复制、哨兵和集群，下面分别说明它们的作用，以及解决了什么样的问题。

持久化：持久化是最简单的高可用方法(有时甚至不被归为高可用的手段)，主要作用是数据备份，即将数据存储在硬盘，保证数据不会因进程退出而丢失。

复制：复制是高可用Redis的基础，哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份，以及对于读操作的负载均衡和简单的故障恢复。缺陷：故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制。

哨兵：在复制的基础上，哨兵实现了自动化的故障恢复。缺陷：写操作无法负载均衡；存储能力受到单机的限制。

集群：通过集群，Redis解决了写操作无法负载均衡，以及存储能力受到单机限制的问题，实现了较为完善的高可用方案。

* 1. Redis持久化概述

持久化的功能：Redis是内存数据库，数据都是存储在内存中，为了避免进程退出导致数据的永久丢失，需要定期将Redis中的数据以某种形式(数据或命令)从内存保存到硬盘；当下次Redis重启时，利用持久化文件实现数据恢复。除此之外，为了进行灾难备份，可以将持久化文件拷贝到一个远程位置。

Redis持久化分为RDB持久化和AOF持久化**：前者将当前数据保存到硬盘，后者则是将每次执行的写命令保存到硬盘（类似于MySQL的binlog）；**由于AOF持久化的实时性更好，即当进程意外退出时丢失的数据更少，因此AOF是目前主流的持久化方式，不过RDB持久化仍然有其用武之地。

下面依次介绍RDB持久化和AOF持久化；由于Redis各个版本之间存在差异，如无特殊说明，以Redis3.0为准。

* 1. RDB持久化
  2. AOF持久化

1. 主从复制
   1. 主从复制（读写分离）

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。

默认情况下，每台Redis服务器都是主节点；且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点。

**主从复制的作用**

主从复制的作用主要包括：

数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。

故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。

负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。

高可用基石：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是Redis高可用的基础。

* + 1. 安装多个redis



分别设置3个配置文件redis.conf的内容，修改配置文件中的如下内容：

1. 注释掉 #bind 127.0.0.1



1. 关闭保护模式



1. 修改端口号



1. 后台启动



1. 修改pidfile，改成相应端口号



1. 开启主节点授权，如果主节点需要密码的话，这里一定要开启授权才可以，123456为主节点密码



1. 开启密码，如果需要密码的话，123456为密码



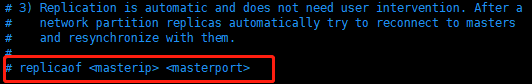
* + 1. 建立复制

主从复制的开启，完全是客户端的行为，不需要主节点做任何事情。主从复制的三种方式：

1. 从节点的配置文件redis.conf中添加：

slaveof <masterip> <masterport> ，在6.0.6中，slaveof可以用replicaof代替

例如：replicaof 127.0.0.1 6379



1. 从节点服务端启动命令：

redis-server启动命令后加入 --slaveof <masterip> <masterport>

例如：



1. 从节点客户端命令

slaveof <masterip> <masterport>，则该Redis实例成为从节点。

例如：

启动客户端，连接端口号为6380的服务端，需要密码



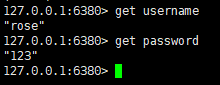
开启主从复制：



此时，6380作为6379的从节点。从节点是只读的，不能写入数据



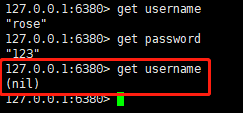
同步主节点数据，直接获取是可以的



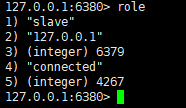
主节点删除数据：



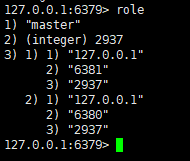
从节点获取数据：这时已经没有username了。



查看当前节点的角色:role,从节点

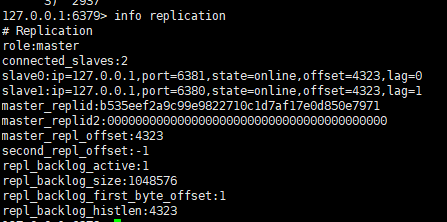


查看当前节点的角色：主节点

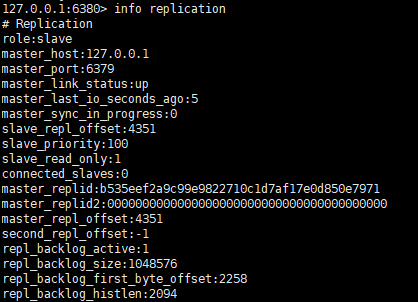


Info replication:查看主从复制信息

主节点信息：



从节点信息：

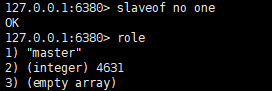


* + 1. 断开复制

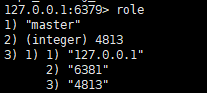
通过slaveof <masterip> <masterport>命令建立主从复制关系以后，可以通过slaveof no one断开。需要注意的是，从节点断开复制后，不会删除已有的数据，只是不再接受主节点新的数据变化。

从节点执行slaveof no one后，又变回为主节点。

从节点：



主节点：



1. 哨兵
   1. 作用

在介绍哨兵之前，首先从宏观角度回顾一下Redis实现高可用相关的技术。它们包括：持久化、复制、哨兵和集群，其主要作用和解决的问题是：

* 持久化：持久化是最简单的高可用方法(有时甚至不被归为高可用的手段)，主要作用是数据备份，即将数据存储在硬盘，保证数据不会因进程退出而丢失。
* 复制：复制是高可用Redis的基础，哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份，以及对于读操作的负载均衡和简单的故障恢复。缺陷：故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制。
* 哨兵：在复制的基础上，哨兵实现了自动化的故障恢复。缺陷：写操作无法负载均衡；存储能力受到单机的限制。
* 集群：通过集群，Redis解决了写操作无法负载均衡，以及存储能力受到单机限制的问题，实现了较为完善的高可用方案。

下面说回哨兵。

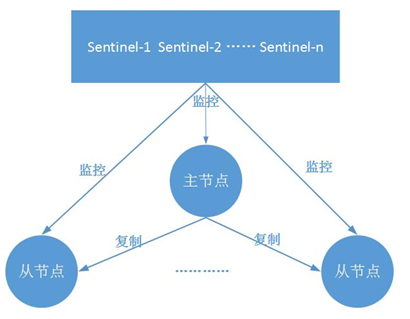
Redis Sentinel，即Redis哨兵，在Redis 2.8版本开始引入。**哨兵的核心功能是主节点的自动故障转移。**下面是Redis官方文档对于哨兵功能的描述：

* 监控（Monitoring）：哨兵会不断地检查主节点和从节点是否运作正常。
* 自动故障转移（Automatic failover）：当主节点不能正常工作时，哨兵会开始自动故障转移操作，它会将失效主节点的其中一个从节点升级为新的主节点，并让其他从节点改为复制新的主节点。
* 配置提供者（Configuration provider）：客户端在初始化时，通过连接哨兵来获得当前Redis服务的主节点地址。
* 通知（Notification）：哨兵可以将故障转移的结果发送给客户端。

其中，监控和自动故障转移功能，使得哨兵可以及时发现主节点故障并完成转移；而配置提供者和通知功能，则需要在与客户端的交互中才能体现。

* 1. 架构

典型的哨兵模式架构图：



它由两部分组成，哨兵节点和数据节点：

* 哨兵节点：哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的redis节点，不存储数据。
* 数据节点：主节点和从节点都是数据节点。
  1. 部署

这一部分将部署一个简单的哨兵系统，包含1个主节点、2个从节点和3个哨兵节点。

* + 1. 部署主从节点

根据前面主从复制的例子，部署1个主节点，2个从节点。端口号等信息，如前面所述。部署成功以后，依次启动主从节点。

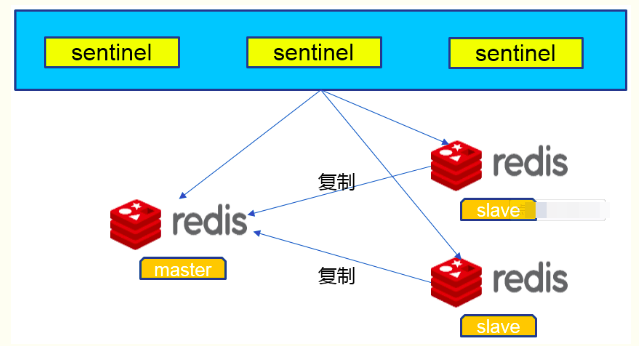
* + 1. 部署哨兵节点

https://blog.csdn.net/yaooch/article/details/80167571

<https://www.cnblogs.com/kevingrace/p/9004460.html>

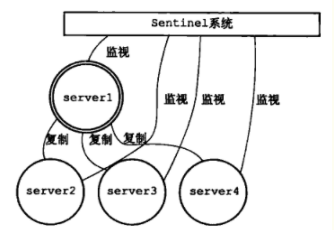
Sentinel(哨兵)是用于监控redis集群中Master状态的工具，是Redis 的高可用性解决方案，sentinel哨兵模式已经被集成在redis2.4之后的版本中。sentinel是redis高可用的解决方案，sentinel系统可以监视一个或者多个redis master服务，以及这些master服务的所有从服务；当某个master服务下线时，自动将该master下的某个从服务升级为master服务替代已下线的master服务继续处理请求。

sentinel可以让redis实现主从复制，当一个集群中的master失效之后，sentinel可以选举出一个新的master用于自动接替master的工作，集群中的其他redis服务器自动指向新的master同步数据。一般建议sentinel采取奇数台，防止某一台sentinel无法连接到master导致误切换。其结构如下:

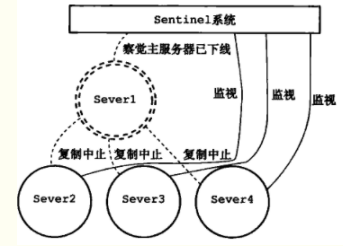


**Redis-Sentinel是Redis官方推荐的高可用性(HA)解决方案**，当用Redis做Master-slave的高可用方案时，假如master宕机了，Redis本身(包括它的很多客户端)都没有实现自动进行主备切换，而Redis-sentinel本身也是一个独立运行的进程，它能监控多个master-slave集群，发现master宕机后能进行自动切换。Sentinel由一个或多个Sentinel 实例 组成的Sentinel 系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器。

例如下图所示：



在Server1 掉线后：



升级Server2 为新的主服务器：

