[command 1](#_Toc30222)

[1.1 basic 1](#_Toc14294)

[1.1.1 cmd connect to server(choose db) 1](#_Toc11876)

[1.1.1 (client list)see all connect 1](#_Toc28089)

[1.1.1 start/config server 1](#_Toc688)

[1.1.1 redis default db and select 7](#_Toc22299)

[1.1.1 show redis version 8](#_Toc15193)

[1.1.1 ubuntu12/install/uninstall 8](#_Toc22479)

[1.1.1 select db 8](#_Toc7626)

[1.1.1 (flushdb)delete selected db keys 8](#_Toc21747)

[1.1.1 (flushall)delete all the db keys 9](#_Toc17051)

[1.1 string(key->value) 9](#_Toc29923)

[1.1.1 set/setnx(set if not exist)/get 9](#_Toc3645)

[1.1.1 append value by key 9](#_Toc3023)

[1.1.1 setex(set within time expire) 9](#_Toc26544)

[1.1.1 mset(set multi pairs of key->value) 10](#_Toc503)

[1.1.1 mget(get multi value by multi key) 10](#_Toc26194)

[1.1.1 getset(set new value and return old value) 10](#_Toc10463)

[1.1.1 strlen(get value length) 10](#_Toc14030)

[1.1.1 incr(thread safe) 10](#_Toc24306)

[1.1 list 11](#_Toc14497)

[1.1.1 key type/size 11](#_Toc7535)

[1.1.1 show list(range,size,push,pop) 11](#_Toc8420)

[1.1.1 lpush,linsert, rpush,lrange 12](#_Toc10904)

[1.1.1 lset(set value by index) 12](#_Toc5170)

[1.1.1 lindex(get value by index) 13](#_Toc14966)

[1.1.1 lpop,rpop(delete from top or bottom) 13](#_Toc30943)

[1.1 sets 14](#_Toc23224)

[1.1.1 sadd(add item)/smembers(show all items) 14](#_Toc3043)

[1.1.1 srem(remove item) 14](#_Toc10860)

[1.1.1 sdiff(get different set) 14](#_Toc22054)

[1.1.1 sunion(get union set) 14](#_Toc9383)

[1.1.1 scard(count of set) 15](#_Toc19721)

[1.1 sorted sets 15](#_Toc15709)

[1.1.1 zadd(add item) 15](#_Toc102)

[1.1.1 zrange(zrange withscores) 15](#_Toc18926)

[1.1 hashes 16](#_Toc2394)

[1.1.1 hset/hget(add item) 16](#_Toc15164)

[1.1.1 hsetnx(set value if not exists) 16](#_Toc18581)

[1.1.1 hmset(set multi key value) 17](#_Toc7488)

[1.1.1 hmget(get multi key value) 17](#_Toc14347)

[1.1.1 hgetall(get all the keys and values) 17](#_Toc12409)

[1.1.1 hkeys/hvals(get all the keys, or values) 17](#_Toc29489)

[1.1.1 hlen(key length) 17](#_Toc21423)

[1.1.1 hdel(del field) 18](#_Toc29237)

[1.1.1 hincrby(append add value) 18](#_Toc26837)

[1.1.1 hexists(check field exists) 18](#_Toc6089)

[1.1 other command 18](#_Toc30239)

[1.1.1 ubuntu run redis(nonstop) 18](#_Toc28285)

[1.1.1 subscribe/publish 18](#_Toc3190)

[1.1.1 linux wildcard grep 19](#_Toc6591)

[1.1.1 psubscribe(subscribe wildcard) 19](#_Toc1786)

[1.1.1 ( pubsub channels )list all channels 19](#_Toc24564)

[1.1.1 multi(transaction) 20](#_Toc4918)

[1.1.1 move key to db 21](#_Toc7756)

[1.1.1 key exists 21](#_Toc12325)

[1.1.1 keys regex 21](#_Toc15420)

[1.1.1 del key 21](#_Toc577)

[1.1.1 (setex)set expire key 22](#_Toc9156)

[1.1.1 show expire time(TTL) 22](#_Toc3379)

[1.1.1 rename key 23](#_Toc11501)

[1.1.1 type key 23](#_Toc10207)

[1.1.1 info memory 23](#_Toc7708)

[1.1.1 info stats 23](#_Toc26073)

[1.1 question 23](#_Toc12342)

[1.1.1 Redis storing list inside hash ? 23](#_Toc18830)

[1.1.1 Redis VS kafka 24](#_Toc15529)

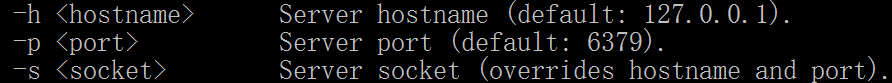
[1.1.1 sentinel/master/slave 25](#_Toc27810)

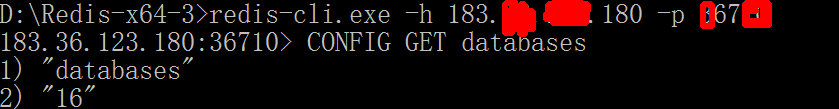
# command

## 1.1 basic

### 1.1.1 cmd connect to server(choose db)

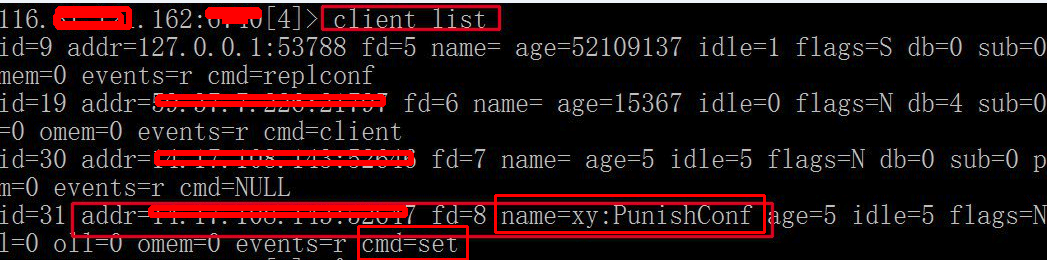






redis-cli -h 221.228.105.186 -p 6391 -n 1

### 1.1.1 (client list)see all connect



### 1.1.1 start/config server

redis-server /etc/redis/redis.conf



# 当配置中需要配置内存大小时，可以使用 1k, 5GB, 4M 等类似的格式，其转换方式如下(不区分大小写)

#

# 1k => 1000 bytes

# 1kb => 1024 bytes

# 1m => 1000000 bytes

# 1mb => 1024\*1024 bytes

# 1g => 1000000000 bytes

# 1gb => 1024\*1024\*1024 bytes

#

# 内存配置大小写是一样的.比如 1gb 1Gb 1GB 1gB

# daemonize no 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

daemonize yes

# 指定redis运行的端口，默认是6379

port 6379

# 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

# bind 127.0.0.1

# Specify the path for the unix socket that will be used to listen for

# incoming connections. There is no default, so Redis will not listen

# unixsocket /tmp/redis.sock

# unixsocketperm 755

# 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

# 0是关闭此设置

timeout 0

# 指定日志记录级别

# Redis总共支持四个级别：debug、verbose、notice、warning，默认为verbose

# debug 记录很多信息，用于开发和测试

# varbose 有用的信息，不像debug会记录那么多

# notice 普通的verbose，常用于生产环境

# warning 只有非常重要或者严重的信息会记录到日志

loglevel debug

# 配置log文件地址

# 默认值为stdout，标准输出，若后台模式会输出到/dev/null

#logfile stdout

logfile /var/log/redis/redis.log

# 可用数据库数

# 默认值为16，默认数据库为0，数据库范围在0-（database-1）之间

databases 16

################################ 快照 #################################

# 保存数据到磁盘，格式如下:

# save <seconds> <changes>

# 指出在多长时间内，有多少次更新操作，就将数据同步到数据文件rdb。

# 相当于条件触发抓取快照，这个可以多个条件配合

# 比如默认配置文件中的设置，就设置了三个条件

# save 900 1 900秒内至少有1个key被改变

# save 300 10 300秒内至少有300个key被改变

# save 60 10000 60秒内至少有10000个key被改变

save 900 1

save 300 10

save 60 10000

# 存储至本地数据库时（持久化到rdb文件）是否压缩数据，默认为yes

rdbcompression yes

# 本地持久化数据库文件名，默认值为dump.rdb

dbfilename dump.rd

# 主从复制. 设置该数据库为其他数据库的从数据库.

# 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

#

# slaveof <masterip> <masterport>

# 当master服务设置了密码保护时(用requirepass制定的密码)

# slav服务连接master的密码

# masterauth <master-password>

Redis 约束

# 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，

# 如果设置 maxclients 0，表示不作限制。

# 当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

#

# maxclients 128

# 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key

# Redis同时也会移除空的list对象

#

# 当此方法处理后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作

# 注意：Redis新的vm机制，会把Key存放内存，Value会存放在swap区

# maxmemory的设置比较适合于把redis当作于类似memcached的缓存来使用，而不适合当做一个真实的DB。

# 当把Redis当做一个真实的数据库使用的时候，内存使用将是一个很大的开销

# maxmemory <bytes>

# 当内存达到最大值的时候Redis会选择删除哪些数据？有五种方式可供选择

# volatile-lru -> 利用LRU算法移除设置过过期时间的key (LRU:最近使用 Least Recently Used )

# allkeys-lru -> 利用LRU算法移除任何key

# volatile-random -> 移除设置过过期时间的随机key

# allkeys->random -> remove a random key, any key

# volatile-ttl -> 移除即将过期的key(minor TTL)

# noeviction -> 不移除任何可以，只是返回一个写错误

# 默认是:

# maxmemory-policy volatile-lru

其他配置

# 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。

# 你可以同时开启asynchronous dumps 和 AOF

appendonly no

# AOF文件名称 (默认: "appendonly.aof")

# appendfilename appendonly.aof

# Redis支持三种同步AOF文件的策略:

#

# no: 不进行同步，系统去操作 . Faster.

# always: always表示每次有写操作都进行同步. Slow, Safest.

# everysec: 表示对写操作进行累积，每秒同步一次. Compromise.

#

# 默认是"everysec"，按照速度和安全折中这是最好的。

# 如果想让Redis能更高效的运行，你也可以设置为"no"，让操作系统决定什么时候去执行

# 或者相反想让数据更安全你也可以设置为"always"

#

# 如果不确定就用 "everysec".

# appendfsync always

appendfsync everysec

# appendfsync no

# AOF策略设置为always或者everysec时，后台处理进程(后台保存或者AOF日志重写)会执行大量的I/O操作

# 在某些Linux配置中会阻止过长的fsync()请求。注意现在没有任何修复，即使fsync在另外一个线程进行处理

#

# 为了减缓这个问题，可以设置下面这个参数no-appendfsync-on-rewrite

#

# This means that while another child is saving the durability of Redis is

# the same as "appendfsync none", that in pratical terms means that it is

# possible to lost up to 30 seconds of log in the worst scenario (with the

# default Linux settings).

#

# If you have latency problems turn this to "yes". Otherwise leave it as

# "no" that is the safest pick from the point of view of durability.

no-appendfsync-on-rewrite no

# Automatic rewrite of the append only file.

# AOF 自动重写

# 当AOF文件增长到一定大小的时候Redis能够调用 BGREWRITEAOF 对日志文件进行重写

#

# 它是这样工作的：Redis会记住上次进行些日志后文件的大小(如果从开机以来还没进行过重写，那日子大小在开机的时候确定)

#

# 基础大小会同现在的大小进行比较。如果现在的大小比基础大小大制定的百分比，重写功能将启动

# 同时需要指定一个最小大小用于AOF重写，这个用于阻止即使文件很小但是增长幅度很大也去重写AOF文件的情况

# 设置 percentage 为0就关闭这个特性

auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb

config redis when start redis server in ubuntu.



# If you want you can bind a single interface, if the bind option is not

# specified all the interfaces will listen for incoming connections.

#bind 127.0.0.1

注释掉上面那个127其他的机器就可以直接连接上

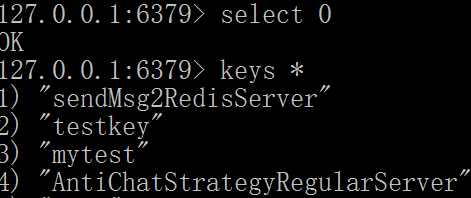
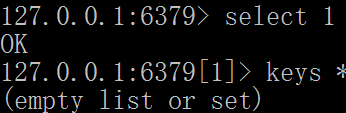




### 1.1.1 redis default db and select

You want to keep different kinds of data — belonging to different applications, for example — separate from each other, but store them all in the same Redis instance.

Use different Redis databases for different kinds of data. In Redis, databases are identified by an integer index, not by a database name. By default, a client is connected to database 0. With the SELECT command you can switch to a different database:



### 1.1.1 show redis version

redis-server --version

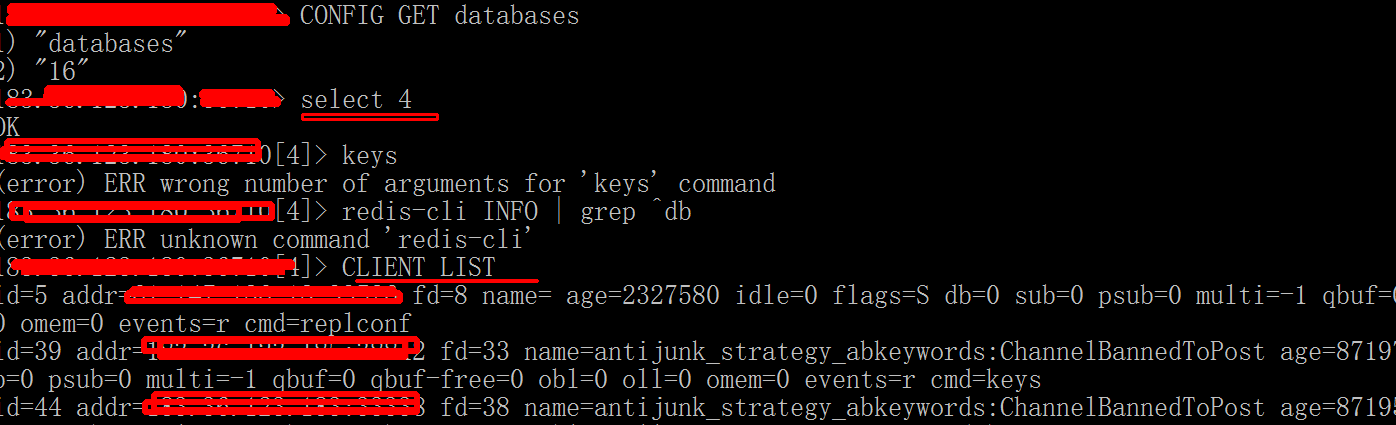
Run the command INFO. The version will be the first item displayed.

### 1.1.1 ubuntu12/install/uninstall

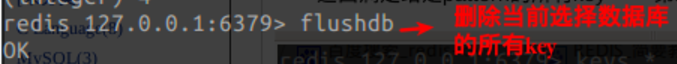
sudo apt-get remove redis-server

<http://grainier.net/how-to-uninstall-redis-server-from-ubuntu/>

### 1.1.1 select db



### 1.1.1 (flushdb)delete selected db keys

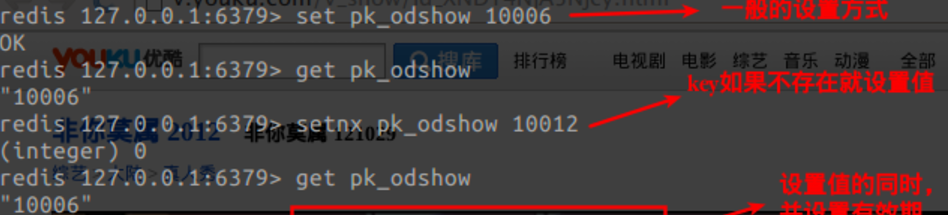


### 1.1.1 (flushall)delete all the db keys

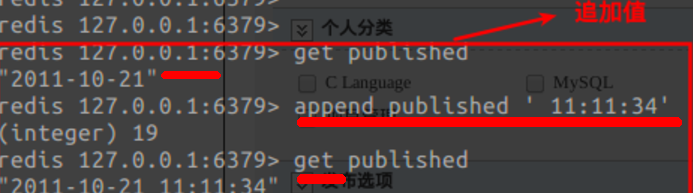


## 1.1 string(key->value)

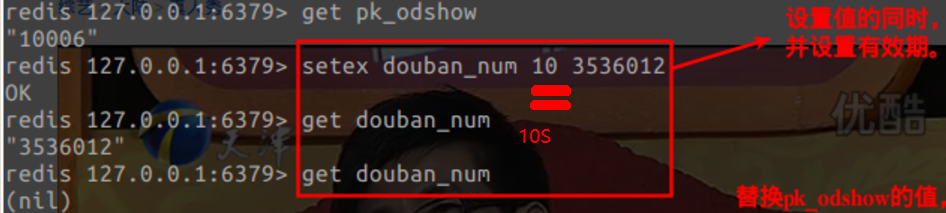
### 1.1.1 set/setnx(set if not exist)/get



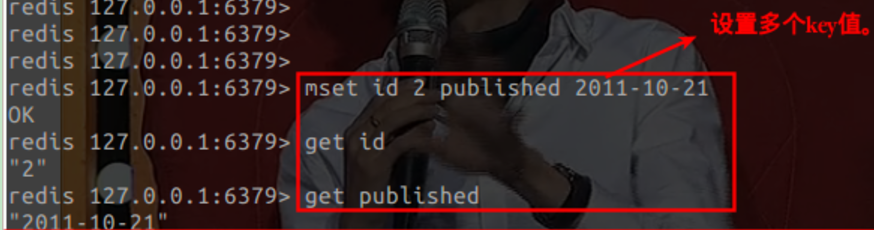
### 1.1.1 append value by key



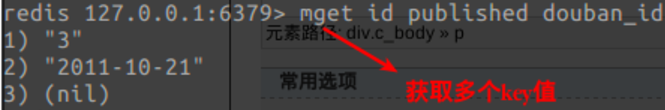
### 1.1.1 setex(set within time expire)



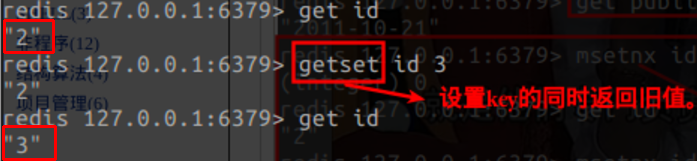
### 1.1.1 mset(set multi pairs of key->value)



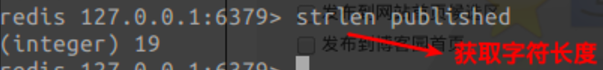
### 1.1.1 mget(get multi value by multi key)



### 1.1.1 getset(set new value and return old value)



### 1.1.1 strlen(get value length)



### 1.1.1 incr(thread safe)

Redis INCR and multi processes?

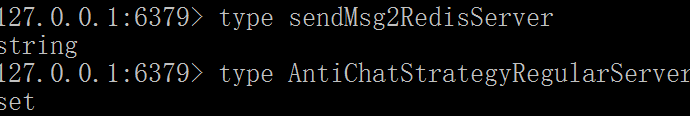
There is no risk of getting the same number when several processes run this command or do I have to use a Redis lock?

Redis is a single-threaded engine, so the execution of all commands is serialized. It always provides atomicity and isolation (in the sense of ACID) for each individual command.

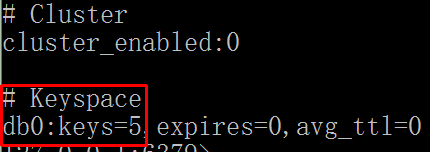
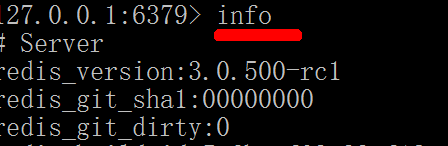
The consequence is applying a single INCR command and exploiting its result is safe (even if multiple connections do it concurrently).

## 1.1 list

### 1.1.1 key type/size

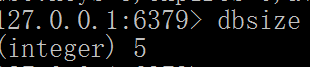


count of keys



use DBSIZE this will give you no of key

Return the number of keys in the currently-selected database.



On a Linux box:

redis-cli KEYS "\*" | wc -l

Note: As mentioned in comments below, this is an O(N) operation, so on a large DB with many keys you should not use this. For smaller deployments, it should be fine.

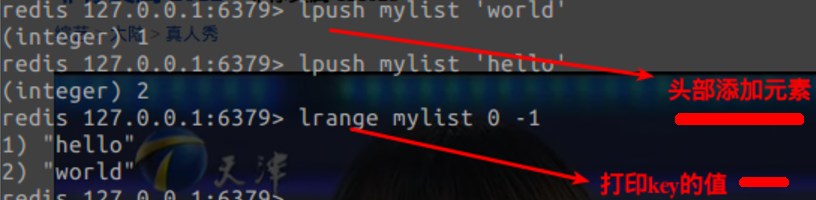
### 1.1.1 show list(range,size,push,pop)

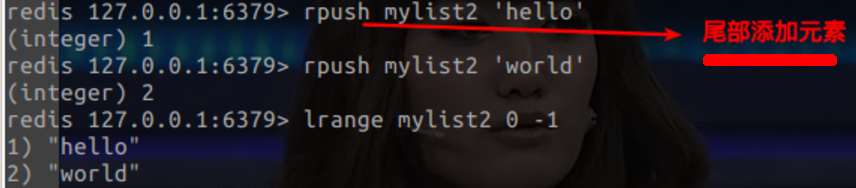
lists：

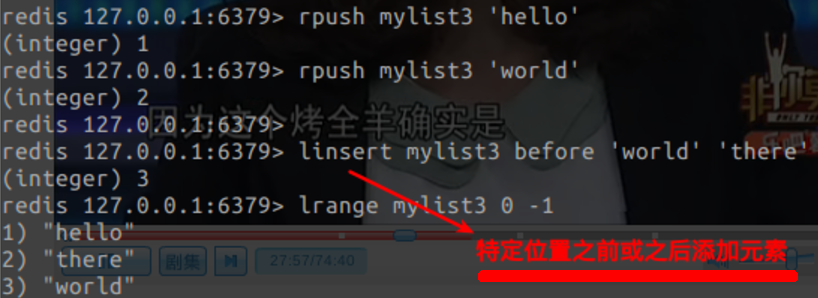
　　lists是一个双向链表结构，操作中key理解为链表的名字。——可以用作栈，也可以用作队列。

注意：lists有长度限制，最大长度是2的32次方

### 1.1.1 lpush,linsert, rpush,lrange

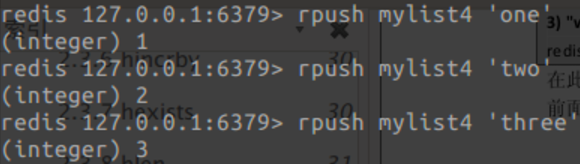




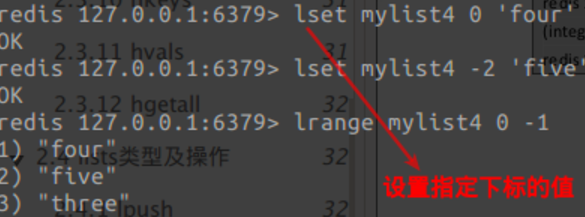


set value by subindex, lset lst idx value :

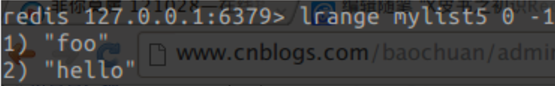
for example, lset lst 0 ‘zero’



### 1.1.1 lset(set value by index)



返回指定下标的值：lindex

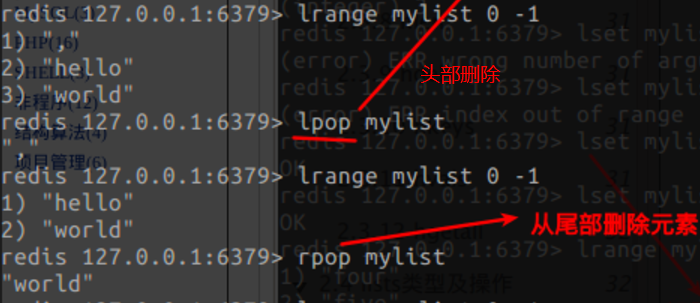


### 1.1.1 lindex(get value by index)



### 1.1.1 lpop,rpop(delete from top or bottom)

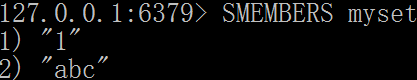
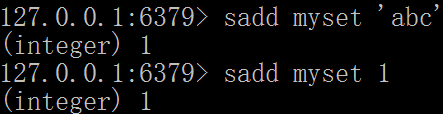
头部，尾部删除 lpop lst, rpop lst



## 1.1 sets

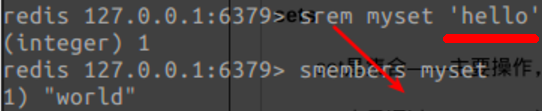
### 1.1.1 sadd(add item)/smembers(show all items)

set是集合——主要操作，对多个集合求交并差等操作，key理解为集合的名字,set也是通过hash table实现的。

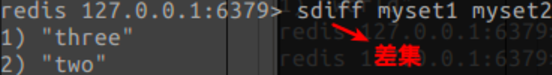


### 1.1.1 srem(remove item)

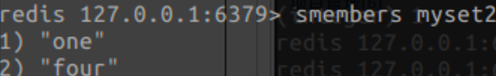
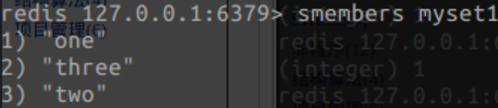
remove ‘hello’

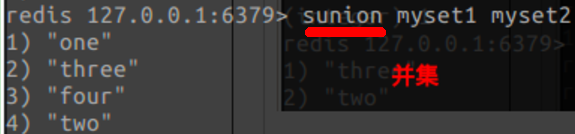


### 1.1.1 sdiff(get different set)

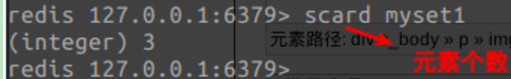


### 1.1.1 sunion(get union set)





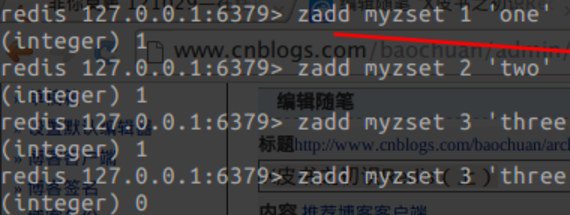
### 1.1.1 scard(count of set)

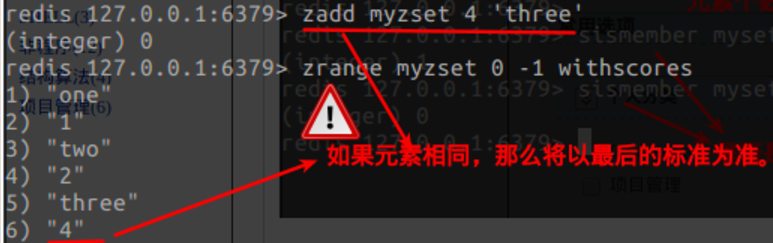


## 1.1 sorted sets

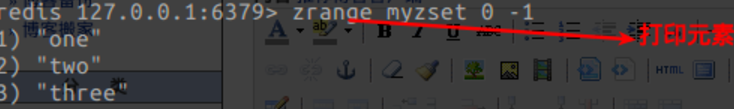
### 1.1.1 zadd(add item)

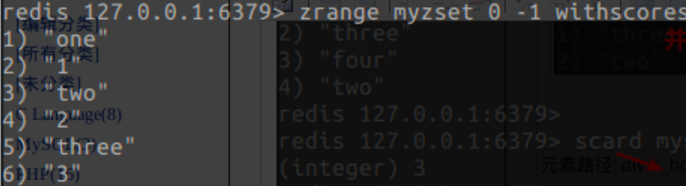
它是set的一个升级版本，它的实现是skip list和hash table的混合体。





### 1.1.1 zrange(zrange withscores)

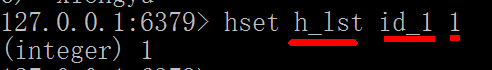




## 1.1 hashes

### 1.1.1 hset/hget(add item)

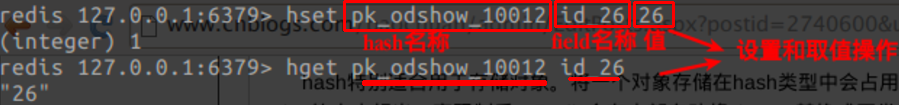
hash特别适合用于存储对象。将一个对象存储在hash类型中会占用更少的内存，并且可以更方便的存储整个对象。如果value的大小超出一定限制后，Redis会在内部自动将zipmap替换成正常的hash实现。

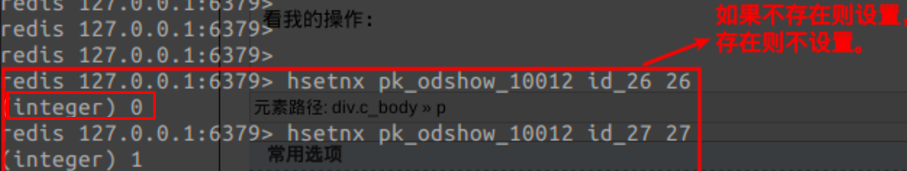


hset h\_lst id\_1 1 (其中h\_lst为hash名称,id\_1为field，1为field取值)



### 1.1.1 hsetnx(set value if not exists)

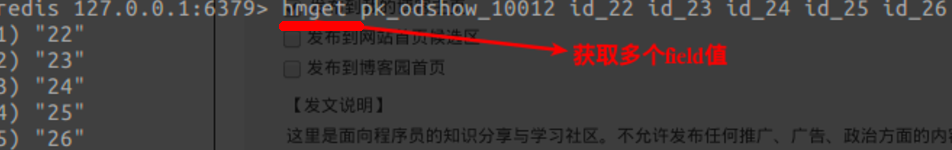




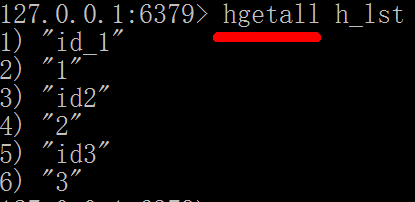
### 1.1.1 hmset(set multi key value)



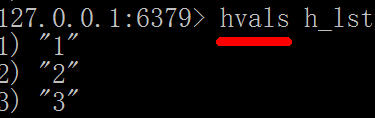
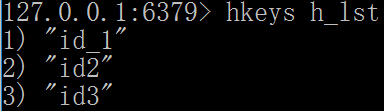
### 1.1.1 hmget(get multi key value)



### 1.1.1 hgetall(get all the keys and values)



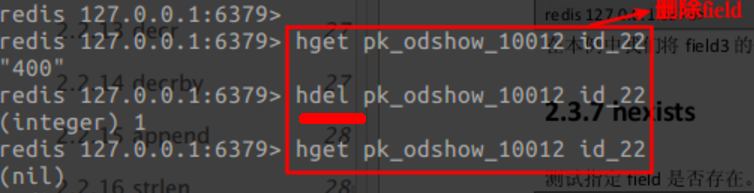
### 1.1.1 hkeys/hvals(get all the keys, or values)



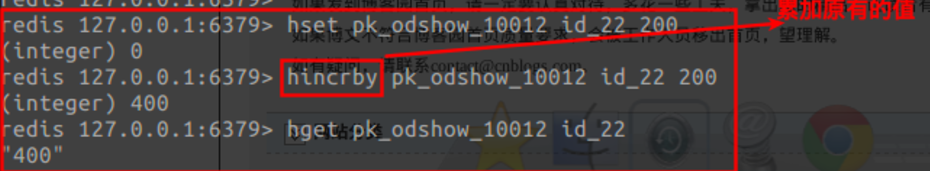
### 1.1.1 hlen(key length)



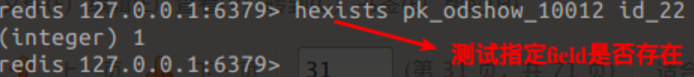
### 1.1.1 hdel(del field)



### 1.1.1 hincrby(append add value)



### 1.1.1 hexists(check field exists)



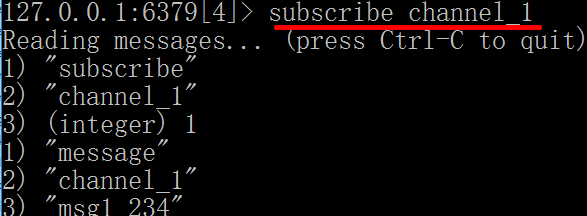
## 1.1 other command

### 1.1.1 ubuntu run redis(nonstop)

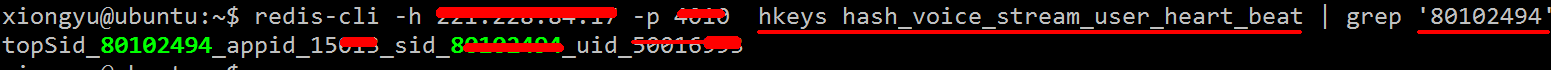
redis-server --port 33220 --daemonize yes

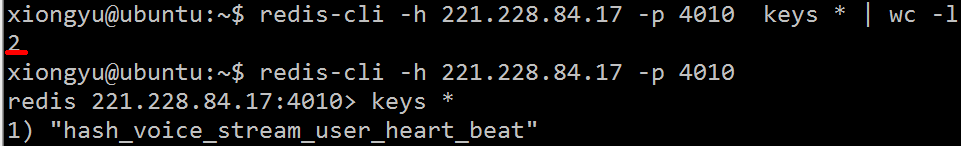
### 1.1.1 subscribe/publish



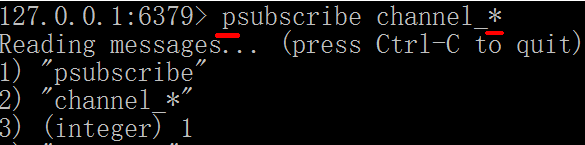


### 1.1.1 linux wildcard grep

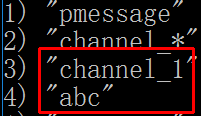
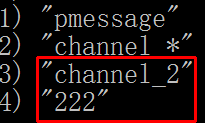




### 1.1.1 psubscribe(subscribe wildcard)



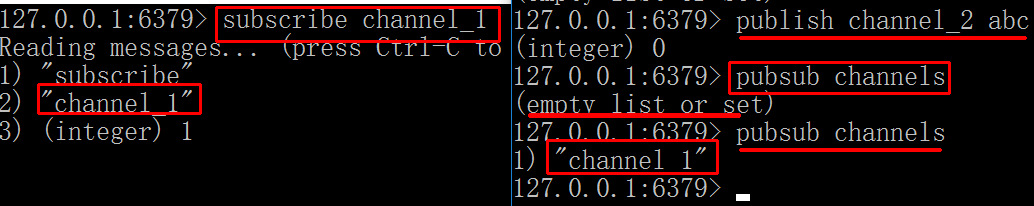
 

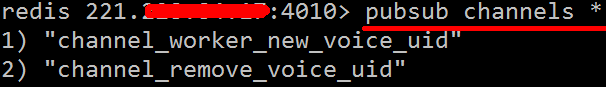
### 1.1.1 ( pubsub channels )list all channels

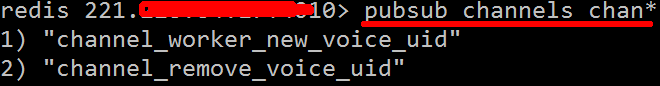
PUBSUB CHANNELS (list channels that has been subscribed)

PUBSUB CHANNELS \*

PUBSUB CHANNELS mystarter\*







### 1.1.1 multi(transaction)

2. Redis 事务

Redis 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

以下是一个事务的例子， 它先以 **MULTI 开始一个事务**， 然后将多个命令入队到事务中， 最后由 **EXEC 命令触发事务**， 一并执行事务中的所有命令：

127.0.0.1:6379> MULTI

OK

127.0.0.1:6379> set name "xiaofeng"

QUEUED

127.0.0.1:6379> get name

QUEUED

127.0.0.1:6379> sadd girlfriends "Jenny" "Amy" "lily"

QUEUED

127.0.0.1:6379> SMEMBERS girlfriends

QUEUED

127.0.0.1:6379> EXEC

1) OK

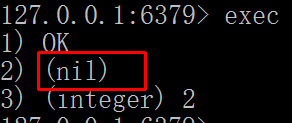
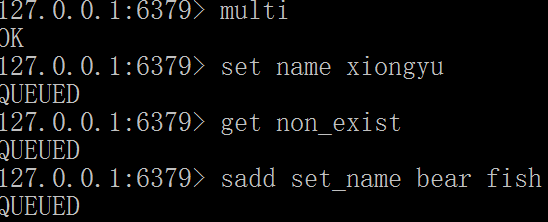
2) "xiaofeng"

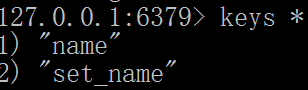
3) (integer) 3

4) 1) "Jenny"

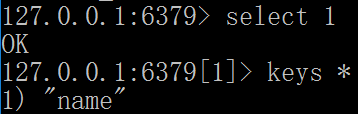
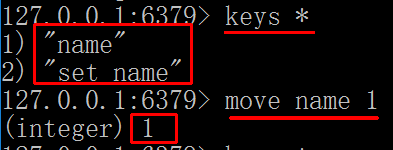
2) "Amy"

3) "lily"





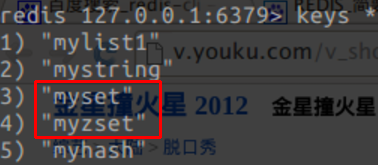
### 1.1.1 move key to db

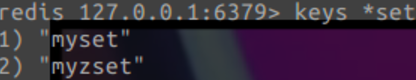


### 1.1.1 key exists

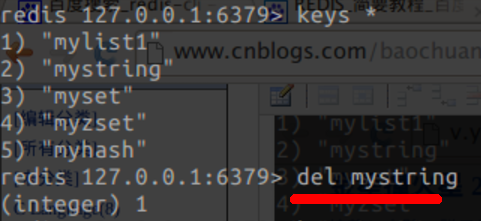


### 1.1.1 keys regex





### 1.1.1 del key



I want to remove keys that match "user\*".

how do I do that in redis command line?

redis-cli KEYS "user\*" | xargs redis-cli DEL

for key in `redis-cli "KEYS" "user\*" | awk '{print $1}'`

do redis-cli "DEL" "$key"

done

### 1.1.1 (setex)set expire key

设置一个过期时间：expire（单位：秒）



SETEX key seconds value

Set key to hold the string value and set key to timeout after a given number of seconds. This command is equivalent to executing the following commands:

SET mykey value

EXPIRE mykey seconds

redis> SETEX mykey 10 "Hello"

"OK"

redis> TTL mykey

(integer) 10

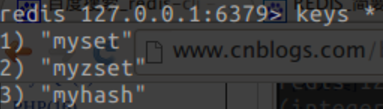
redis> GET mykey

"Hello"

### 1.1.1 show expire time(TTL)

|  |  |
| --- | --- |
| redis> SET mykey "Hello"  "OK"  redis> EXPIRE mykey 10  (integer) 1 | redis> TTL mykey  (integer) 10 |

### 1.1.1 rename key

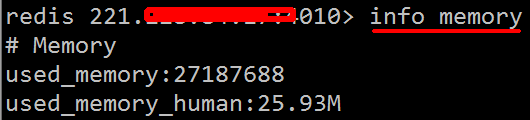




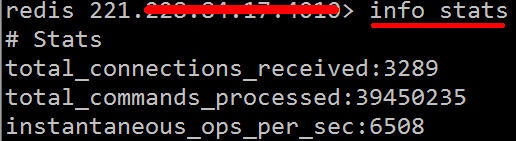
### 1.1.1 type key



### 1.1.1 info memory



### 1.1.1 info stats



## 1.1 question

### 1.1.1 Redis storing list inside hash ?

Redis' data structures cannot be nested inside other data structures, so storing a List inside a Hash is not possible. Instead, use different keys for your servers' CPU values (e.g. server1:cpu).

It's possible to do this with Redisson framework. It allows to store a reference to Redis object in another Redis object though special reference objects which handled by Redisson.

So your task could be solved using List inside Map:

RMap<String, RList<Option>> settings = redisson.getMap("settings");

RList<Option> options1 = redisson.getList("settings\_server1\_option");

options1.add(new Option("name", "s1"));

options1.add(new Option("cpu", "80"));

settings.put("server1", options1);

<https://redisson.org/>

### 1.1.1 Redis VS kafka

I am a beginner to Redis and Kafka. Redis can be used as realtime pub-sub. However Kafka is also realtime pub-sub. I am confused which one to use when.

Redis pub-sub is mostly like a fire and forget system where all the messages you produced will be delivered to all the consumers at once and the data is kept no where. You have limitation in memory with respect to redis. Also number of producers and consumers can affect the performance in Redis.

Kafka on other hand is a high throughput, distributed log that can be used like a queue. Here any number of users can produce and consumers can consume at any time they want. It also provides persistence for the messages sent through the queue.

Final Take:

Use Redis:

If you want fire and forget kind of system, where all the messages that you produce are delivered instantly to consumers.

If speed is most concerned.

If you can live up with data loss.

If you don't want your system to hold the message that has been sent.

Amount of data that is gonna be dealt is not huge.

Use kafka:

If you want reliability.

If you want your system to have a copy of messages that has been sent even after consumption.

If you can't live up with data loss.

If Speed is not a big concern.

data size is huge

redis 消息推送（基于分布式 pub/sub）多用于实时性较高的消息推送，并不保证可靠。其他的mq和kafka保证可靠但有一些延迟（非实时系统没有保证延迟）。redis-pub/sub断电就清空，而使用redis-list作为消息推送虽然有持久化，但是又太弱智，也并非完全可靠不会丢。

另外一点，redis 发布订阅除了表示不同的 topic 外，并不支持分组，比如kafka中发布一个东西，多个订阅者可以分组，同一个组里只有一个订阅者会收到该消息，这样可以用作负载均衡。

比如，kafka 中发布：topic = "发布帖子" data="文章1" 这个消息，后面有一百台服务器每台服务器都是一个订阅者，都订阅了这个 topic，但是他们可能分为三组，A组50台，用来真的做发布文章，A组50台里所有 subscriber 都订阅了这个topic。

由于在同一组，这条消息 （topic="发布帖子", data="文章1"）只会被A组里面一台当前空闲的机器收到。而B组25台服务器用于统计，C组25台服务器用于存档备份，每组只有一台会收到。用不同的组来决定每条消息要抄送出多少分去，用同组内哪些订阅者忙，哪些订阅者空闲来决定消息会被分到哪台服务器去处理，生产者消费者模型嘛。redis完全没有这类机制，这两点是最大的区别。。。。。

kafka的性能要在topic数量小于64的时候，才能发挥威力。partition决定的。极限情况下丢消息，例如：主写入消息后，主机器宕机，并硬盘损坏。review代码的时候发现的。

rabbit不知道，但是rocket的性能是（万条每秒），并且能够横向无限扩展，单机topic数量在256时，性能损失较小。rocket可以说是kafka的变种，是阿里在充分reviewkafka代码后，开发的metaQ。在不断更新，修补以后，阿里把metaQ3.0更名为rocket，并且rocket是java写的易于维护。另外就是rocket和kafka有类似无限堆积的能力。想想，断电不丢消息，积压两亿条消息毫无压力，niubility

### 1.1.1 sentinel/master/slave