**Redis**

# Redis简介

Redis是C语言编写的一个开源的、高性能的、基于键值对的缓存于存储系统，通过提供多种键值数据类型来适应不同场景下的缓存于存储需求。同时Redis的诸多高层级功能使其可以胜任消息队列、任务队列等不同的角色。

## Redis的特性

* **存储结构**

Redis是Remote Dictionary Server（远程字典服务器）的缩写，它以字典结构存储数据，并允许其他应用通过TCP协议读写字典中的内容。同大多数脚本语言中的字典一样，Redis字典中的键值除了可以是字符串，还可以是其他数据类型。到目前为止Redis支持的值数据类型如下（键全都是字符串类型）：

1. 字符串类型："小明，小红"。
2. 散列类型：{uname:"张三",age:"18"}。
3. 列表类型：[1,2,3,4,5]
4. 集合类型：['a','b']
5. 有序集合类型：[5000 'a',1000 'b',10 'c']

* **内存存储与持久化**

Redis数据库中的所有数据都存储在内存中。由于内存的读写速度远快于硬盘，因此Redis在性能上对比其他基于硬盘存储的数据库有非常明显的优势。然而将数据存储在内存中也有问题，比如程序退出后内存中的数据会丢失。不过Redis提供了对持久化的支持，即可以将内存中的数据异步写入到硬盘中，同时不影响继续提供服务。

* **功能丰富**

Redis虽然是作为数据库开发的，但由于其提供了丰富的功能，越来越多的人将其用作缓存、队列系统等。

Redis可以为每个键设置生存时间，生存时间到期后键会自动被删除。这一功能配合出色的性能让Redis可以作为缓存系统来使用。作为缓存系统，Redis还可以限定数据占用的最大内存空间，在数据达到空间限制后可以按照一定的规则自动淘汰不需要的键。

除此之外，Redis的列表类型键可以用来实现队列，并且支持阻塞式读取，可以很容易地实现一个高性能的优先级队列。同时在更高层面上，Redis还支持“发布/订阅”的消息模式，可以基于此构建聊天室系统。

* **简单稳定**

Redis直观的寸尺结构使得通过程序与Redis交互十分简单。在Redis中使用命令来读写数据。例如：在Redis中读取键名为post:1的散列类型键的title字段的值，可以使用如下命令语句实现：

HGET post:1 title

Redis提供了几十种不同编程语言的客户端库，这些库都很好地封装了Redis的命令，使得在程序中与Redis进行交互变得更容器。这些库还提供了可以将编程语言中的数据类型直接以相应的形式存储到Redis中的简单方法，使用起来非常方便。

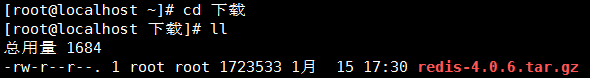
# Redis的安装和操作

## Linux下的安装和操作

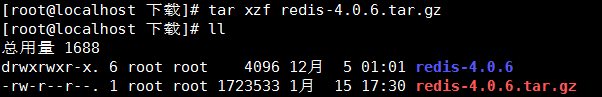
在安装Redis之前需要了解Redis的版本规则：为偶数的版本是稳定版（如2.8版、3.0版），技术版是非稳定版（如2.7版、2.9版）。

### Redis的安装

1. 在Linux系统中推荐直接下载Redis源代码编译安装以获得最新的稳定版本。下载地址：http://www.redis.cn/download.html。下载好源码文件后将其拷贝到Linux系统中。
2. 使用Xshell客户端工具连接到Linux系统（使用root账户），然后使用cd命令进入到存放压缩包的文件夹中（/root/下载/redis-4.0.6.tar.gz）。



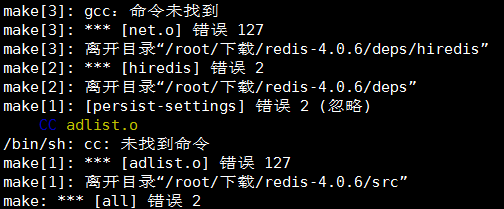
1. 使用tar xzf redis-4.0.6.tar.gz命令将压缩文件解压（解压到当前目录）解压后使用ll命令可以查看当前文件夹下的所有文件夹和文件。可以看到多出一个文件夹redis-4.0.6



1. 使用cd 命令进入到redis-4.0.6文件夹中使用make命令开始编译Redis。



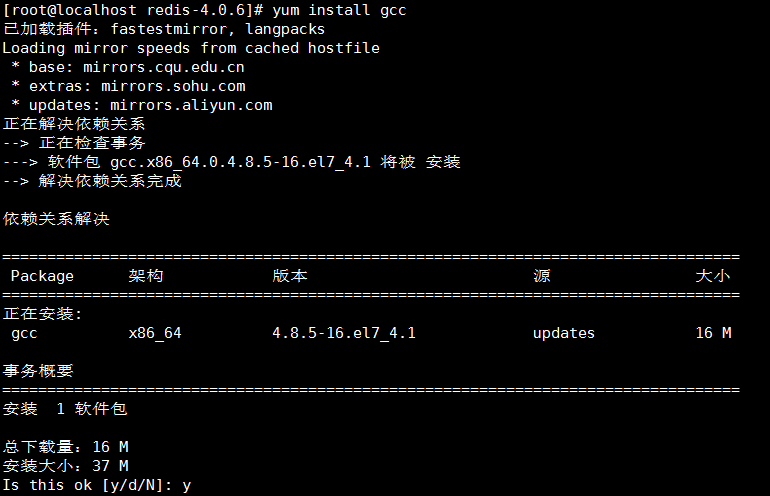
如果编译报错，提示未找到gcc命令，说明该Linux系统中还未安装gcc编译器。



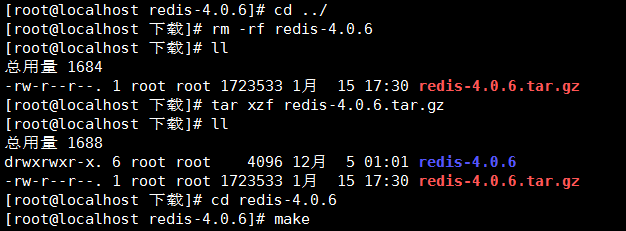
1. 使用yum install gcc命令安装gcc编译器（在线安装）。



然后系统就会开始下载gcc插件并进行安装操作。

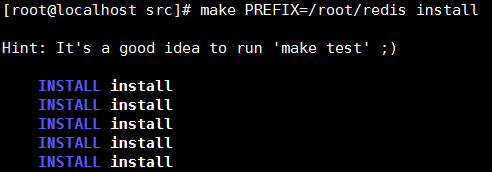


1. 重新进入解压后的Redis文件夹中使用make命令进行Redis的编译。如果编译依然报错，可以使用cd ../命令先退回到上级目录，再使用rm –rf redis-4.0.6命令将之前解压后的Redis文件夹删除后，重新解压后进行编译。

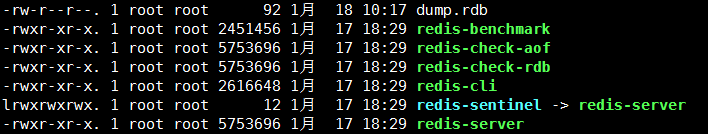


到此系统开始将解压后的文件进行编译。

1. 编译成功后，进入src文件夹，执行make PREFIX=/root/redis install将Redis安装到/root/redis目录下。



此时Linux下安装Redis基本完成。我们进入/root/redis下会看到有一个bin文件夹，其中包含了Redis的命令文件。



dump.rdb：该文件是由Redis服务器自动生成的。

redis-benchmark ：用于进行redis性能测试的工具

redis-check-aof：用于修复出问题的AOF文件

redis-check-rdb ：用于修复出问题的dump.rdb文件

redis-cli ：redis的客户端

redis-sentinel ：用于集群管理

redis-server ：redis的服务端

1. 设置防火墙端口

Redis默认使用的是6379端口，如果需要远程访问Redis服务需要对防火墙进行设置。

1. 关闭防火墙的设置。

在CentOS 7中默认使用firewall做为防火墙，下面是启动&关闭防火墙的命令：

# 启动firewall

systemctl start firewalld.service

# 关闭firewall

systemctl stop firewalld.service

# 禁止firewall开机启动

systemctl disable firewalld.service

1. 开放端口限制

# 开放6379端口

firewall-cmd --zone=public --add-port=6379/tcp --permanent

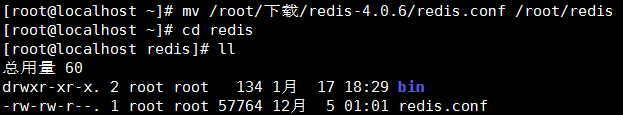
# 重启防火墙

firewall-cmd --reload

### Redis的部署

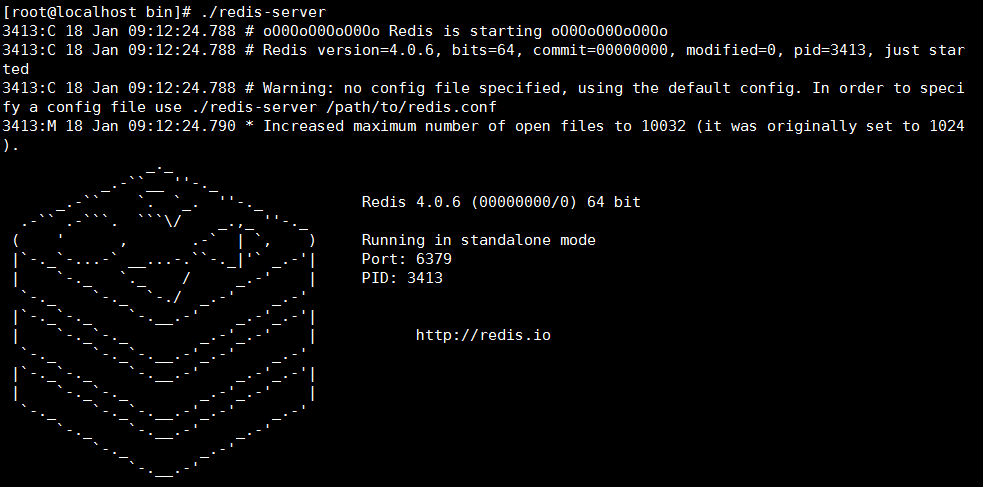
安装成功后，下面对Redis 进行部署

1. 首先为了方便管理，将解压后的Redis文件夹中的redis.conf配置文件移动到Redis的安装目录下（/root/redis）



redis.conf文件是Redis的配置文件，主要用于管理诸如端口号…等之类的设置。

1. 进入到Redis的安装路径中的bin目录执行./redis-server 命令，启动Redis 服务



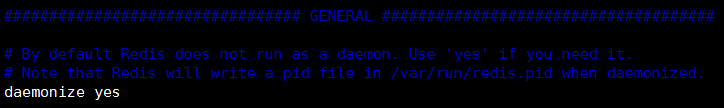
此时Redis会以默认的配置启动Redis服务.

注意：这里直接执行Redis-server 启动的Redis服务，是在前台直接运行的(效果如上图)，也就是说，执行完该命令后，如果Lunix关闭当前会话，则Redis服务也随即关闭。正常情况下，启动Redis服务需要从后台启动，并且指定启动配置文件。

1. 后台启动redis服务
2. 首先编辑redis.conf文件，将daemonize属性改为yes（表明需要在后台运行）

使用vim 命令打开配置文件，然后按i键转为编辑模式进行设置。



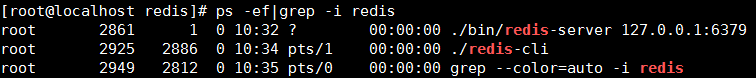


修改完成后，按esc返回键退出编辑模式，再使用 :wq 命令保存并退出。

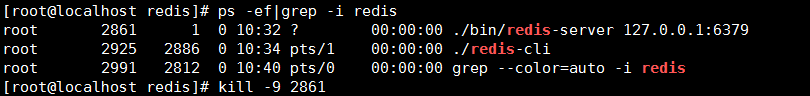
1. 再次启动Redis服务，并指定加载的配置文件



此时我们可以查看Redis的启动进程



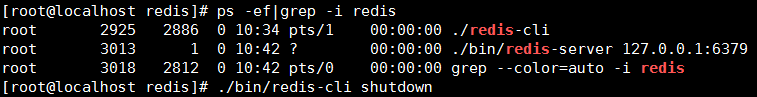
1. 关闭Redis服务
2. 使用关闭进程的方式（不推介，属于断电操作，可能会造成数据丢失）。首先先查询服务进程PID，然后执行kill -9命令结束进程。



此时我们再去查询进程会看到redis-server进程没有了。



1. 使用正常关闭（正常关闭可以做到数据保存，保证数据的安全）./redis-cli shutdown
2. 关闭服务器的命令由客户端发起。



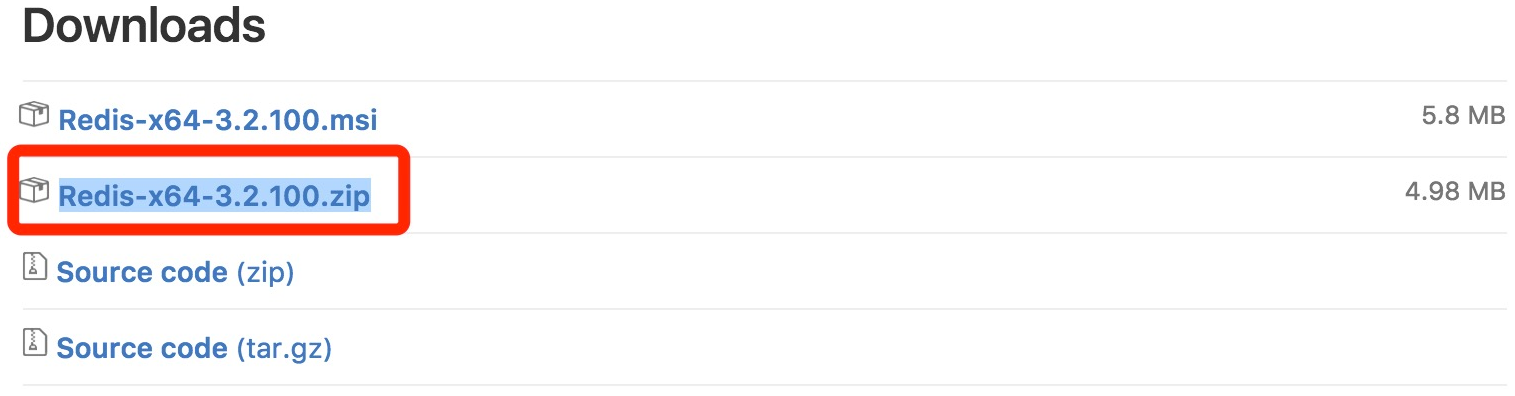
此时我们再去查询进程会看到redis-server进程没有了。



## Window下的安装和操作

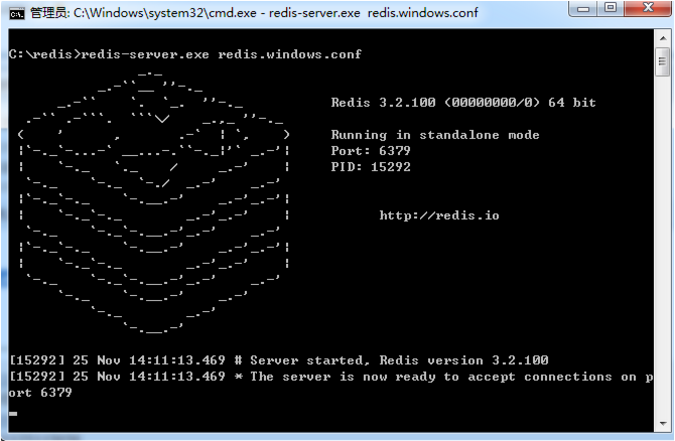
* **Window下启动**

1. Redis 支持 32 位和 64 位。这个需要根据你系统平台的实际情况选择，这里我们下载 Redis-x64-xxx.zip压缩包到 C 盘，解压后，将文件夹重新命名为 redis。下载地址：https://github.com/MicrosoftArchive/redis/releases



1. 打开一个 cmd 窗口 使用cd命令切换目录到 C:\redis 运行 redis-server.exe redis.windows.conf 。

如果想方便的话，可以把 redis 的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个 redis.windows.conf 可以省略，如果省略，会启用默认的。输入之后，会显示如下界面：



这时候另启一个cmd窗口，原来的不要关闭，不然就无法访问服务端了。

切换到redis目录下运行 redis-cli.exe -h 127.0.0.1 -p 6379 。

设置键值对 set myKey abc

取出键值对 get myKey

## redis.conf配置文件说明

#redis.conf

# Redis configuration file example.

# ./redis-server /path/to/redis.conf

################################## INCLUDES ###################################

#这在你有标准配置模板但是每个redis服务器又需要个性设置的时候很有用。

# include /path/to/local.conf

# include /path/to/other.conf

################################ GENERAL #####################################

#是否在后台执行，yes：后台运行；no：不是后台运行（老版本默认）

daemonize yes

#3.2里的参数，是否开启保护模式，默认开启。要是配置里没有指定bind和密码。开启该参数后，redis只会本地进行访问，拒绝外部访问。要是开启了密码 和bind，可以开启。否 则最好关闭，设置为no。

protected-mode yes

#redis的进程文件

pidfile /var/run/redis/redis-server.pid

#redis监听的端口号。

port 6379

#此参数确定了TCP连接中已完成队列(完成三次握手之后)的长度， 当然此值必须不大于Linux系统定义的/proc/sys/net/core/somaxconn值，默认是511，而Linux的默认参数值是128。当系统并发量大并且客户端速度缓慢的时候，可以将这二个参数一起参考设定。该内核参数默认值一般是128，对于负载很大的服务程序来说大大的不够。一般会将它修改为2048或者更大。在/etc/sysctl.conf中添加:net.core.somaxconn = 2048，然后在终端中执行sysctl -p。

tcp-backlog 511

#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

bind 127.0.0.1

#配置unix socket来让redis支持监听本地连接。

# unixsocket /var/run/redis/redis.sock

#配置unix socket使用文件的权限

# unixsocketperm 700

# 此参数为设置客户端空闲超过timeout，服务端会断开连接，为0则服务端不会主动断开连接，不能小于0。

timeout 0

#tcp keepalive参数。如果设置不为0，就使用配置tcp的SO\_KEEPALIVE值，使用keepalive有两个好处:检测挂掉的对端。降低中间设备出问题而导致网络看似连接却已经与对端端口的问题。在Linux内核中，设置了keepalive，redis会定时给对端发送ack。检测到对端关闭需要两倍的设置值。

tcp-keepalive 0

#指定了服务端日志的级别。级别包括：debug（很多信息，方便开发、测试），verbose（许多有用的信息，但是没有debug级别信息多），notice（适当的日志级别，适合生产环境），warn（只有非常重要的信息）

loglevel notice

#指定了记录日志的文件。空字符串的话，日志会打印到标准输出设备。后台运行的redis标准输出是/dev/null。

logfile /var/log/redis/redis-server.log

#是否打开记录syslog功能

# syslog-enabled no

#syslog的标识符。

# syslog-ident redis

#日志的来源、设备

# syslog-facility local0

#数据库的数量，默认使用的数据库是DB 0。可以通过”SELECT “命令选择一个db

databases 16

################################ SNAPSHOTTING ################################

# 快照配置

# 注释掉“save”这一行配置项就可以让保存数据库功能失效

# 设置sedis进行数据库镜像的频率。

# 900秒（15分钟）内至少1个key值改变（则进行数据库保存--持久化）

# 300秒（5分钟）内至少10个key值改变（则进行数据库保存--持久化）

# 60秒（1分钟）内至少10000个key值改变（则进行数据库保存--持久化）

save 900 1

save 300 10

save 60 10000

#当RDB持久化出现错误后，是否依然进行继续进行工作，yes：不能进行工作，no：可以继续进行工作，可以通过info中的rdb\_last\_bgsave\_status了解RDB持久化是否有错误

stop-writes-on-bgsave-error yes

#使用压缩rdb文件，rdb文件压缩使用LZF压缩算法，yes：压缩，但是需要一些cpu的消耗。no：不压缩，需要更多的磁盘空间

rdbcompression yes

#是否校验rdb文件。从rdb格式的第五个版本开始，在rdb文件的末尾会带上CRC64的校验和。这跟有利于文件的容错性，但是在保存rdb文件的时候，会有大概10%的性能损耗，所以如果你追求高性能，可以关闭该配置。

rdbchecksum yes

#rdb文件的名称

dbfilename dump.rdb

#数据目录，数据库的写入会在这个目录。rdb、aof文件也会写在这个目录

dir /var/lib/redis

################################# REPLICATION #################################

#复制选项，slave复制对应的master。

# slaveof <masterip> <masterport>

#如果master设置了requirepass，那么slave要连上master，需要有master的密码才行。masterauth就是用来配置master的密码，这样可以在连上master

后进行认证。

# masterauth <master-password>

#当从库同主机失去连接或者复制正在进行，从机库有两种运行方式：1) 如果slave-serve-stale-data设置为yes(默认设置)，从库会继续响应客户端的请求。2) 如果slave-serve-stale-data设置为no，除去INFO和SLAVOF命令之外的任何请求都会返回一个错误”SYNC with master in progress”。

slave-serve-stale-data yes

#作为从服务器，默认情况下是只读的（yes），可以修改成NO，用于写（不建议）。

slave-read-only yes

#是否使用socket方式复制数据。目前redis复制提供两种方式，disk和socket。如果新的slave连上来或者重连的slave无法部分同步，就会执行全量同步，master会生成rdb文件。有2种方式：disk方式是master创建一个新的进程把rdb文件保存到磁盘，再把磁盘上的rdb文件传递给slave。socket是master创建一个新的进程，直接把rdb文件以socket的方式发给slave。disk方式的时候，当一个rdb保存的过程中，多个slave都能共享这个rdb文件。socket的方式就的一个个slave顺序复制。在磁盘速度缓慢，网速快的情况下推荐用socket方式。

repl-diskless-sync no

#diskless复制的延迟时间，防止设置为0。一旦复制开始，节点不会再接收新slave的复制请求直到下一个rdb传输。所以最好等待一段时间，等更多的slave连上来。

repl-diskless-sync-delay 5

#slave根据指定的时间间隔向服务器发送ping请求。时间间隔可以通过 repl\_ping\_slave\_period 来设置，默认10秒。

# repl-ping-slave-period 10

#复制连接超时时间。master和slave都有超时时间的设置。master检测到slave上次发送的时间超过repl-timeout，即认为slave离线，清除该slave信息。slave检测到上次和master交互的时间超过repl-timeout，则认为master离线。需要注意的是repl-timeout需要设置一个比repl-ping-slave-period更大的值，不然会经常检测到超时。

# repl-timeout 60

#是否禁止复制tcp链接的tcp nodelay参数，可传递yes或者no。默认是no，即使用tcp nodelay。如果master设置了yes来禁止tcp nodelay设置，在把数据复制给slave的时候，会减少包的数量和更小的网络带宽。但是这也可能带来数据的延迟。默认我们推荐更小的延迟，但是在数据量传输很大的场景下，建议选择yes。

repl-disable-tcp-nodelay no

#复制缓冲区大小，这是一个环形复制缓冲区，用来保存最新复制的命令。这样在slave离线的时候，不需要完全复制master的数据，如果可以执行部分同步，只需要把缓冲区的部分数据复制给slave，就能恢复正常复制状态。缓冲区的大小越大，slave离线的时间可以更长，复制缓冲区只有在有slave连接的时候才分配内存。没有slave的一段时间，内存会被释放出来，默认1m。

# repl-backlog-size 5mb

#master没有slave一段时间会释放复制缓冲区的内存，repl-backlog-ttl用来设置该时间长度。单位为秒。

# repl-backlog-ttl 3600

#当master不可用，Sentinel会根据slave的优先级选举一个master。最低的优先级的slave，当选master。而配置成0，永远不会被选举。

slave-priority 100

#redis提供了可以让master停止写入的方式，如果配置了min-slaves-to-write，健康的slave的个数小于N，mater就禁止写入。master最少得有多少个健康的slave存活才能执行写命令。这个配置虽然不能保证N个slave都一定能接收到master的写操作，但是能避免没有足够健康的slave的时候，master不能写入来避免数据丢失。设置为0是关闭该功能。

# min-slaves-to-write 3

#延迟小于min-slaves-max-lag秒的slave才认为是健康的slave。

# min-slaves-max-lag 10

# 设置1或另一个设置为0禁用这个特性。

# Setting one or the other to 0 disables the feature.

# By default min-slaves-to-write is set to 0 (feature disabled) and

# min-slaves-max-lag is set to 10.

################################## SECURITY ###################################

#requirepass配置可以让用户使用AUTH命令来认证密码，才能使用其他命令。这让redis可以使用在不受信任的网络中。为了保持向后的兼容性，可以注释该命令，因为大部分用户也不需要认证。使用requirepass的时候需要注意，因为redis太快了，每秒可以认证15w次密码，简单的密码很容易被攻破，所以最好使用一个更复杂的密码。

# requirepass foobared

#把危险的命令给修改成其他名称。比如CONFIG命令可以重命名为一个很难被猜到的命令，这样用户不能使用，而内部工具还能接着使用。

# rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52

#设置成一个空的值，可以禁止一个命令

# rename-command CONFIG ""

################################### LIMITS ####################################

# 设置能连上redis的最大客户端连接数量。默认是10000个客户端连接。由于redis不区分连接是客户端连接还是内部打开文件或者和slave连接等，所以maxclients最小建议设置到32。如果超过了maxclients，redis会给新的连接发送’max number of clients reached’，并关闭连接。

# maxclients 10000

#redis配置的最大内存容量。当内存满了，需要配合maxmemory-policy策略进行处理。注意slave的输出缓冲区是不计算在maxmemory内的。所以为了防止主机内存使用完，建议设置的maxmemory需要更小一些。

# maxmemory <bytes>

#内存容量超过maxmemory后的处理策略。

#volatile-lru：利用LRU算法移除设置过过期时间的key。

#volatile-random：随机移除设置过过期时间的key。

#volatile-ttl：移除即将过期的key，根据最近过期时间来删除（辅以TTL）

#allkeys-lru：利用LRU算法移除任何key。

#allkeys-random：随机移除任何key。

#noeviction：不移除任何key，只是返回一个写错误。

#上面的这些驱逐策略，如果redis没有合适的key驱逐，对于写命令，还是会返回错误。redis将不再接收写请求，只接收get请求。写命令包括：set setnx setex append incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby getset mset msetnx exec sort。

# maxmemory-policy noeviction

#lru检测的样本数。使用lru或者ttl淘汰算法，从需要淘汰的列表中随机选择sample个key，选出闲置时间最长的key移除。

# maxmemory-samples 5

############################## APPEND ONLY MODE ###############################

#默认redis使用的是rdb方式持久化，这种方式在许多应用中已经足够用了。但是redis如果中途宕机，会导致可能有几分钟的数据丢失，根据save来策略进行持久化，Append Only File是另一种持久化方式，可以提供更好的持久化特性。Redis会把每次写入的数据在接收后都写入 appendonly.aof 文件，每次启动时Redis都会先把这个文件的数据读入内存里，先忽略RDB文件。

appendonly no

#aof文件名

appendfilename "appendonly.aof"

#aof持久化策略的配置

#no表示不执行fsync，由操作系统保证数据同步到磁盘，速度最快。

#always表示每次写入都执行fsync，以保证数据同步到磁盘。

#everysec表示每秒执行一次fsync，可能会导致丢失这1s数据。

appendfsync everysec

# 在aof重写或者写入rdb文件的时候，会执行大量IO，此时对于everysec和always的aof模式来说，执行fsync会造成阻塞过长时间，no-appendfsync-on-rewrite字段设置为默认设置为no。如果对延迟要求很高的应用，这个字段可以设置为yes，否则还是设置为no，这样对持久化特性来说这是更安全的选择。设置为yes表示rewrite期间对新写操作不fsync,暂时存在内存中,等rewrite完成后再写入，默认为no，建议yes。Linux的默认fsync策略是30秒。可能丢失30秒数据。

no-appendfsync-on-rewrite no

#aof自动重写配置。当目前aof文件大小超过上一次重写的aof文件大小的百分之多少进行重写，即当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。当前AOF文件大小是上次日志重写得到AOF文件大小的二倍（设置为100）时，自动启动新的日志重写过程。

auto-aof-rewrite-percentage 100

#设置允许重写的最小aof文件大小，避免了达到约定百分比但尺寸仍然很小的情况还要重写

auto-aof-rewrite-min-size 64mb

#aof文件可能在尾部是不完整的，当redis启动的时候，aof文件的数据被载入内存。重启可能发生在redis所在的主机操作系统宕机后，尤其在ext4文件系统没有加上data=ordered选项（redis宕机或者异常终止不会造成尾部不完整现象。）出现这种现象，可以选择让redis退出，或者导入尽可能多的数据。如果选择的是yes，当截断的aof文件被导入的时候，会自动发布一个log给客户端然后load。如果是no，用户必须手动redis-check-aof修复AOF文件才可以。

aof-load-truncated yes

################################ LUA SCRIPTING ###############################

# 如果达到最大时间限制（毫秒），redis会记个log，然后返回error。当一个脚本超过了最大时限。只有SCRIPT KILL和SHUTDOWN NOSAVE可以用。第一个可以杀没有调write命令的东西。要是已经调用了write，只能用第二个命令杀。

lua-time-limit 5000

################################ REDIS CLUSTER ###############################

#集群开关，默认是不开启集群模式。

# cluster-enabled yes

#集群配置文件的名称，每个节点都有一个集群相关的配置文件，持久化保存集群的信息。这个文件并不需要手动配置，这个配置文件有Redis生成并更新，每个Redis集群节点需要一个单独的配置文件，请确保与实例运行的系统中配置文件名称不冲突

# cluster-config-file nodes-6379.conf

#节点互连超时的阀值。集群节点超时毫秒数

# cluster-node-timeout 15000

#在进行故障转移的时候，全部slave都会请求申请为master，但是有些slave可能与master断开连接一段时间了，导致数据过于陈旧，这样的slave不应该被提升为master。该参数就是用来判断slave节点与master断线的时间是否过长。判断方法是：

#比较slave断开连接的时间和(node-timeout \* slave-validity-factor) + repl-ping-slave-period

#如果节点超时时间为三十秒, 并且slave-validity-factor为10,假设默认的repl-ping-slave-period是10秒，即如果超过310秒slave将不会尝试进行故障转移

# cluster-slave-validity-factor 10

#master的slave数量大于该值，slave才能迁移到其他孤立master上，如这个参数若被设为2，那么只有当一个主节点拥有2 个可工作的从节点时，它的一个从节点会尝试迁移。

# cluster-migration-barrier 1

#默认情况下，集群全部的slot有节点负责，集群状态才为ok，才能提供服务。设置为no，可以在slot没有全部分配的时候提供服务。不建议打开该配置，这样会造成分区的时候，小分区的master一直在接受写请求，而造成很长时间数据不一致。

# cluster-require-full-coverage yes

################################## SLOW LOG ###################################

###slog log是用来记录redis运行中执行比较慢的命令耗时。当命令的执行超过了指定时间，就记录在slow log中，slog log保存在内存中，所以没有IO操作。

#执行时间比slowlog-log-slower-than大的请求记录到slowlog里面，单位是微秒，所以1000000就是1秒。注意，负数时间会禁用慢查询日志，而0则会强制记录所有命令。

slowlog-log-slower-than 10000

#慢查询日志长度。当一个新的命令被写进日志的时候，最老的那个记录会被删掉。这个长度没有限制。只要有足够的内存就行。你可以通过 SLOWLOG RESET 来释放内存。

slowlog-max-len 128

################################ LATENCY MONITOR ##############################

#延迟监控功能是用来监控redis中执行比较缓慢的一些操作，用LATENCY打印redis实例在跑命令时的耗时图表。只记录大于等于下边设置的值的操作。0的话，就是关闭监视。默认延迟监控功能是关闭的，如果你需要打开，也可以通过CONFIG SET命令动态设置。

latency-monitor-threshold 0

############################# EVENT NOTIFICATION ##############################

#键空间通知使得客户端可以通过订阅频道或模式，来接收那些以某种方式改动了 Redis 数据集的事件。因为开启键空间通知功能需要消耗一些 CPU ，所以在默认配置下，该功能处于关闭状态。

#notify-keyspace-events 的参数可以是以下字符的任意组合，它指定了服务器该发送哪些类型的通知：

##K 键空间通知，所有通知以 \_\_keyspace@\_\_ 为前缀

##E 键事件通知，所有通知以 \_\_keyevent@\_\_ 为前缀

##g DEL 、 EXPIRE 、 RENAME 等类型无关的通用命令的通知

##$ 字符串命令的通知

##l 列表命令的通知

##s 集合命令的通知

##h 哈希命令的通知

##z 有序集合命令的通知

##x 过期事件：每当有过期键被删除时发送

##e 驱逐(evict)事件：每当有键因为 maxmemory 政策而被删除时发送

##A 参数 g$lshzxe 的别名

#输入的参数中至少要有一个 K 或者 E，否则的话，不管其余的参数是什么，都不会有任何 通知被分发。详细使用可以参考http://redis.io/topics/notifications

notify-keyspace-events ""

############################### ADVANCED CONFIG ###############################

#数据量小于等于hash-max-ziplist-entries的用ziplist，大于hash-max-ziplist-entries用hash

hash-max-ziplist-entries 512

#value大小小于等于hash-max-ziplist-value的用ziplist，大于hash-max-ziplist-value用hash。

hash-max-ziplist-value 64

#数据量小于等于list-max-ziplist-entries用ziplist，大于list-max-ziplist-entries用list。

list-max-ziplist-entries 512

#value大小小于等于list-max-ziplist-value的用ziplist，大于list-max-ziplist-value用list。

list-max-ziplist-value 64

#数据量小于等于set-max-intset-entries用iniset，大于set-max-intset-entries用set。

set-max-intset-entries 512

#数据量小于等于zset-max-ziplist-entries用ziplist，大于zset-max-ziplist-entries用zset。

zset-max-ziplist-entries 128

#value大小小于等于zset-max-ziplist-value用ziplist，大于zset-max-ziplist-value用zset。

zset-max-ziplist-value 64

#value大小小于等于hll-sparse-max-bytes使用稀疏数据结构（sparse），大于hll-sparse-max-bytes使用稠密的数据结构（dense）。一个比16000大的value是几乎没用的，建议的value大概为3000。如果对CPU要求不高，对空间要求较高的，建议设置到10000左右。

hll-sparse-max-bytes 3000

#Redis将在每100毫秒时使用1毫秒的CPU时间来对redis的hash表进行重新hash，可以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受Redis时不时的对请求有2毫秒的延迟的话，把这项配置为no。如果没有这么严格的实时性要求，可以设置为yes，以便能够尽可能快的释放内存。

activerehashing yes

##对客户端输出缓冲进行限制可以强迫那些不从服务器读取数据的客户端断开连接，用来强制关闭传输缓慢的客户端。

#对于normal client，第一个0表示取消hard limit，第二个0和第三个0表示取消soft limit，normal client默认取消限制，因为如果没有寻问，他们是不会接收数据的。

client-output-buffer-limit normal 0 0 0

#对于slave client和MONITER client，如果client-output-buffer一旦超过256mb，又或者超过64mb持续60秒，那么服务器就会立即断开客户端连接。

client-output-buffer-limit slave 256mb 64mb 60

#对于pubsub client，如果client-output-buffer一旦超过32mb，又或者超过8mb持续60秒，那么服务器就会立即断开客户端连接。

client-output-buffer-limit pubsub 32mb 8mb 60

#redis执行任务的频率为1s除以hz。

hz 10

#在aof重写的时候，如果打开了aof-rewrite-incremental-fsync开关，系统会每32MB执行一次fsync。这对于把文件写入磁盘是有帮助的，可以避免过大的延迟峰值。

aof-rewrite-incremental-fsync yes

# 数据类型

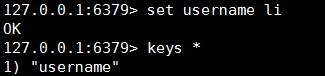
## 常用命令

* **获取符合规则的键名列表**

KEYS pattern

pattern支持glob风格通配符格式

|  |  |
| --- | --- |
| 符号 | 说明 |
| ? | 匹配一个字符 |
| \* | 匹配任意个（包括0个）字符 |
| [] | 匹配括号间的任一字符，可以使用“-”符号表示一个范围，如a[b-d]可以匹配“ab”、“ac”和“ad” |
| \x | 匹配字符x，用于转义符号。如要匹配“?”就需要使用\? |

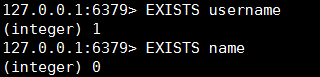


默认Redis中没有任何的key，所以我们需要先用set命令设置key-value。

* **判断一个键是否存在**

EXISTS key

如果键存在则返回整数类型1，否则返回0。



* **删除键**

DEL key [key …]

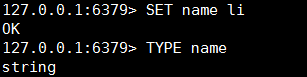
可以删除一个或多个键，返回值是删除的键的个数。



* **获得键值的数据类型**

TYPE key

TYPE命令用来获得键值的数据类，返回值可能是string（字符串类型）、hash（散列类型）、list（列表类型）、set（集合类型）、zset（有序集合类型）。



## 字符串类型

### 简介

字符串类型是Redis中最基本的数据类型，它能存储任何形式的字符串，包括二进制数据。一个字符串类型键允许存储的数据的最大容量是512MB。

### 命令

1. **赋值与取值**

SET key value

GET key

INCR

SET和GET是Redis中最简单的两个命令，它们实现的功能和编程语言中的读写变量相似，如key=”hello”在Redis中是这样表示的：



想要读取键值则更简单：

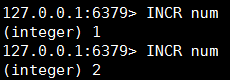


当键不存在时返回空结果。

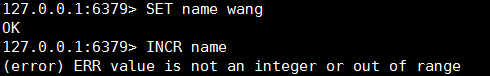
1. **递增数字**

前面说过字符串类型可以存储任何形式的字符串，当存储的字符串是整数形式时，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：

INCR key



当操作的键不存在时会默认键值为0，所以第一次递增后的结果是1.当键值不是整数时Redis会提示错误。



1. **递减数字**

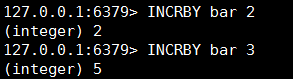
DECR key



1. **增加指定的整数**

INCRBY key increment

INCRBY命令与INCR命令基本一样，只不过前者可以通过increment参数指定一次增加的数值。



1. **减少指定的整数**

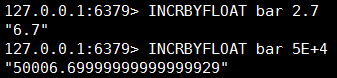
DECRBY key decrement



1. **增加指定浮点数**

INCRBYFLOAT key increment

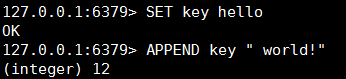
INCRBYFLOAT命令类似INCRBY命令，差别是前者可以递增一个双精度浮点数。



1. **向尾部追加值**

APPEND key value

APPEND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。返回值是追加后字符串的总长度。

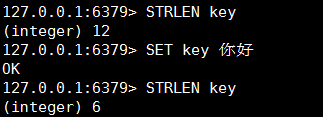


此时key的值是“hello world!”

1. **获取字符串长度**

STRLEN key

STRLEN命令返回键值的长度，如果键不存在则返回0.



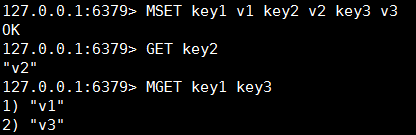
前面提到字符串类型可以存储二进制数据，所以它可以存储任何编码的字符会窜。例子中Redis接收到的是使用UTF-8编码的中文，由于“你”和“好”两个字的UTF-8编码的长度都是3，所以此例中会返回6.

1. **同时获得/设置多个键值**

MGET key [key …]

MSET key value [key value …]

MGET/MSET与GET/SET相似，不过MGET/MSET可以同时获得/设置多个键的键值。



1. **位操作**

GETBIT key offset

SETBIT key offset value

BITCOUNT key [start] [end]

BITOP operation destkey key [key …]

一个字节由8个二进制位组成，Redis提供了4个命令可以直接对二进制位进行操作。



bar的3个字母“b”“a”和“r”对应的ASSCII码分别为98、97和114，转换成二进制后分别为1100010、1100001和1110010，所以foo键中的二进制位结果如下

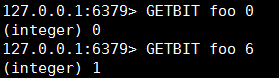
r

a

b

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

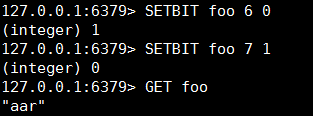
GETBIT命令可以获得一个字符串类型键指定位置的二进制位的值（0或1），索引从0开始。



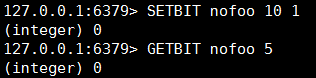
如果需要获取的二进制位的索引超出了键值的二进制位的实际长度则默认位值是0.



SETBIT命令可以设置字符串类型键指定位置的二进制位的值，返回值是该位置的旧指。如我们要将foo键值设置为aar，可以通过位操作将foo键的二进制位的索引第6位设为0，第7为设为1



如果要设置的位置超过了键值的二进制位的长度，SETBIT命令会自动将中间的二进制位设置为0，同理设置一个不存在的键的指定二进制位的值会自动将其前面的位赋值为0



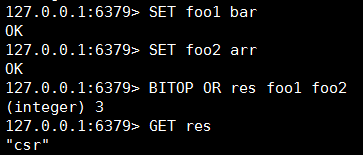
BITCOUNT命令可以获得字符串类型键中值是1的二进制位个数。



可以通过参数来限制统计的字节范围，如果我们只希望统计前两个字节（即“aa”）



BITOP命令可以对多个字符串类型键进行位运算，并将结果存储在destkey参数指定的键中。BITOP命令支持的运算操作有AND、OR、XOR和NOT。如我们可以对bar和aar进行OR运算。



b

r

a

foo1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

OR

r

a

a

foo1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

=

r

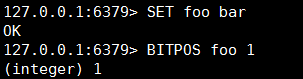
a

c

foo1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

Redis2.8.7引入了BITPOS命令，可以获得指定键的第一个位值是0或者1的位置。还是以“bar”这个键值为例，如果想获取键值的第一个二进制位为1的偏移量，则可以执行：



BITPOS命令的第二个和第三个参数分别可以 用来指定要查询的起始字节和结束字节。注意这里的单位不再是二进制位，而是字节。

如果想查询第二个字节和第三个字节之间（即“a”和“r”）出现的第一个值为1的二进制位的偏移量。



这里的返回结果的偏移量是从头开始算起的，与起始字节无关。另外需要特别说明的是如果不设置结束字节且键值的所有二进制位都是1，则当要查询值为0 的二进制位偏移量时，返回结果会是键值长度的下一个字节的偏移量，这是因为Redis会认为键值长度之后的二进制位都是0.

## 散列类型

### 简介

Redis是采用字典结构以键值对的形式存储数据的，而散列类型（hash）的键值也是一种字典结果，其存储了字段和字段值的 映射，但字段值只能是字符串，不支持其他数据类型。一个散列类型键可以包含至多2³²-1个字节。

散列类型适合存储对象：使用对象类别和ID构成键名，使用字段表示对象的属性，而字段值则存储属性值。例如要存储ID为2的汽车对象，可以分别使用名为color、name和price的3个字段来存储该辆汽车的颜色、名称和价格。

键 字段 字段值

白色

color

car:2

奥迪

name

90万

price

### 命令

1. **赋值与取值**

HSET key field value

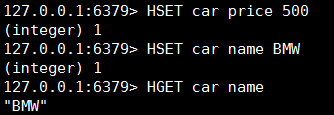
HGET key field

HMSET key field value [field value …]

HMGET key field [field …]

HGETALL key

HSET命令用来给字段赋值，而HGET命令用来获得字段的值。



HSET命令的方便之处在于不区分插入和更新操作，这意味着修改数据时不用事先判断字段是否存在来决定要执行的插入操作还是更新操作。当执行的是插入操作时HSET命令会返回1，当执行的是更新操作时，HSET命令返回0。当键本身不存在时，HSET命令还会自动建立它。

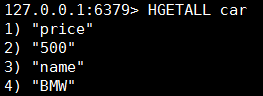
当需要同时设置多个字段的值时，可以使用HMSET命令。

HMSET key field1 value1 field2 value2

相应的HMGET命令可以同时获得多个字段的值



如果想获取键中所有字段和字段值却不知道键中有哪些字段时，应该使用HGETALL命令

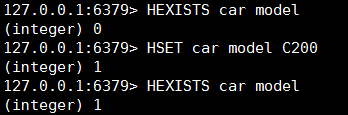


返回的结果是字段和字段值组成的列表，不是很直观，好在很多语言 Redis客户端会将HGETALL的返回结果封装成编程语言张中的对象。

1. **判断字段是否存在**

HEXISTS key field

HEXISTS命令用来判断一个字段是否存在。如果存在则返回1，否则返回0（如果键不存在也返回0）。



1. **当字段不存在时赋值**

HSETNX key field value

HSETNX命令与HSET命令类似，区别在于如果字段已经存在，HSETNX命令不执行任何操作。伪代码如下：

def hsetnx($key, $field, $value)

$isExists = HEXISTS $key, $field

if $isExists is 0

HSET $key, $field, $value

return 1

else

return 0

只不过HSETNX命令是原子操作，不用担心竞态条件。

1. **增加数字**

HINCRBY key field increment

HINCRBY与字符串类型的INCRBY命令类似，可以使字段值增加指定的整数。散列类型没有HINCR命令，但是可以通过HINCRBY key field 1来实现。

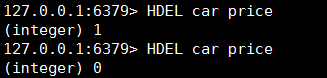


之前person键不寻在，HINCRBY命令会自动建立该键并默认score字段在执行命令前的值为“0”。命令的返回值是增值后的字段值。

1. **删除字段**

HDEL key field [field …]

HDEL命令可以删除一个或多个字段，返回值是被删除的字段个数：



1. **只获取字段名或字段值**

HKEYS key

HVALS key

有时仅仅需要获取键中所有字段的名字而不需要字段值，那么可以使用HKEYS命令。



HVALS命令与HKEYS命令相对应，HVALS命令用来获得键中所有字段值



1. **获得字段数量**

HLEN key



## 列表类型

### 简介

列表类型（list）可以存储一个有序的字符串列表，常用的操作是向列表两端添加元素，或者获得列表的某一片段。

列表类型内部是使用双向链表是实现的，所以向列表两端添加元素的时间复杂度为O(1)，获取越近两端的元素速度越快。这意味着即使是一个有几千万个元素的列表，获取头部或尾部的10条记录也是极快的。不过使用链表的代价是通过索引访问元素比较慢。

与散列类型键最多能容纳的字段数量相同，一个列表类型键最多能容纳2³²-1个元素。

### 命令

1. **向列表两端添加元素**

LPUSH key value [value …]

RPUSH key value [value …]

LPUSH命令用来向列表左边添加元素，返回值表示增加元素后列表的长度。



此时numbers键中的数据如下图所示。

1

LPUSH命令还支持同时添加多个元素。



LPUSH会先向列表左边加入“2”，然后再加入“3”，所以此时numbers键中的数据如下图：

3

2

1

向列表右边添加元素则使用RPUSH命令，用法和LPUSH命令一样。



此时numbers键中的数据如下图：

-1

0

1

2

3

1. **从列表两端弹出元素**

LPOP key

RPOP key

LPOP命令可以从列表左边弹出一个元素。LPOP命令执行两步操作：第一步是将列表左边的元素从列表中移除，第二部是返回被移除的元素值。



此时numbers键中的数据如下图：

1

2

-1

0

同样，RPOP命令可以从列表右边弹出一个元素。



此时numbers键中的数据如下图：

0

2

1

结合上面提到的4个命令可以使用列表类型来模拟栈和队列的操作：如果想把列表当作栈，则搭配使用LPUSH和LPOP或RPUSH和RPOP，如果想当成队列，则搭配使用LPUSH和RPOP或RPUSH和LPOP。

1. **获取列表中元素的个数**

LLEN key

当键不存在时LLEN会返回0

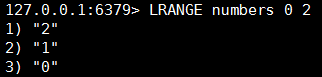


LLEN命令的功能类似于SQL语句SELECT COUNT(\*) FROM table\_name，但是LLEN的时间复杂度为0(1)，使用时Redis会直接读取现成的值，而不需要像部分关系数据库那样需要遍历一遍数据表来统计条目数量。

1. **获取列表片段**

LRANGE key start stop

LRANGE命令是列表类型最常用的命令之一，它能够获得列表中的某一片段。LRANGE命令将会返回索引从start到stop之间的所有元素（包含两端的元素）。



LRANGE命令在取得列表片段的同时不会像LPOP一样删除该片段，另外LRANGE命令与很多语言中用来截取数组片段的方法slice有一点区别是LRANGE返回的值包含最右边的元素。

LRANGE命令也支持负索引，表示从右边开始计算序数，如“-1”表示最右边第一个元素，“-2”表示最右边第二个元素。



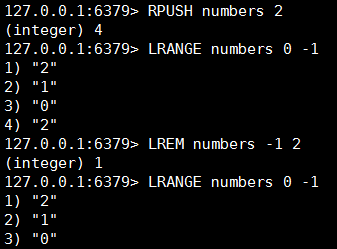
显然，LRANGE numbers 0 -1可以获取列表中的所有元素。另外一些特殊情况如下。

1. 如果start的索引位置比stop的索引位置靠后，则返回空列表。
2. 如果stop大于实际的索引范围，则会返回到列表最右边的元素。
3. **删除列表中之指定的值**

LREM key count value

LREM命令会删除列表中前count个值为value的元素，返回值是实际删除的元素个数。根据count值的不同，LREM命令的执行方式会略有差异。

1. 当count > 0时LREM命令会从列表左边开始删除前count个值为value的元素。
2. 当count < 0时LREM命令会从列表右边开始删除前count个值为value的元素
3. 当count = 0时LREM命令会删除所有值为value的元素。



1. **获得/设置指定索引的元素值**

LINDEX key index

LSET key index value

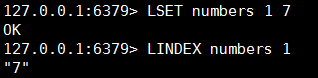
如果要将列表类型当作数组来用，LINDEX命令是必不可少的。LINDEX命令用来返回指定索引的元素，索引从0开始。



如果index是负数则表示从右边开始计算的索引，最右边元素的索引是-1.



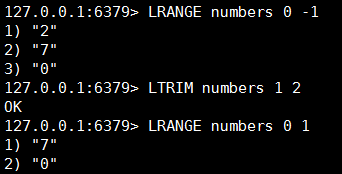
LSET是另外一个通过索引操作列表的命令，它会将索引为index的元素赋值为value。



1. **只保留列表指定片段**

LTRIM key start end

LTRIM命令可以删除指定索引范围之外的所有元素，其指定列表范围的方法和LRANGE命令相同。



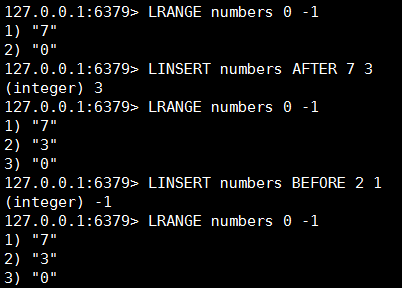
LTRIM命令常和LPUSH命令一起使用来限制列表中元素的数量，比如记录日志时我们希望只保留最近的100条日志，则每次加入新元素时调用一次LTRIM命令。

1. **向列表中插入元素**

LINSERT key BEFORE | AFTER pivot value

LINSERT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。

LINSERT命令的返回值是插入后列表的元素个数。



1. **将元素从一个列表转到另一个列表**

RPOPLPUSH source destination

RPOPLPUSH命令先执行RPOP命令再执行LPUSH命令。RPOPLPUSH命令会先从source列表类型键的右边弹出一个元素，然后将其加入到destination列表类型键的左边，并返回这个元素的值，整个过程是原子的。其具体实现可以表示为伪代码：

def rpoplpush ($source, $destination)

$value = RPOP $source

LPUSH $destination, $value

return $value

当吧列表类型作为队列使用时，RPOPLPUSH命令可以很直观地在多个队列中传递数据。当source和destination相同时，RPOPLPUSH命令会不断地将队尾的元素移到队首。

## 集合类型

### 简介

集合类型中的每个元素都是不同的，且没有顺序。一个集合类型（set）键可以存储2³²-1个字符串。集合类型和列表类型有相似之处，但也有不相同的。

|  |  |  |
| --- | --- | --- |
|  | 集合类型 | 列表类型 |
| 存储内容 | 至多2³²-1个字符串 | 至多2³²-1个字符串 |
| 有序性 | 否 | 是 |
| 唯一性 | 是 | 否 |

集合类型的常用操作是向集合中加入或删除元素、判断某个元素是否存在等，由于集合类型在Redis内部是使用值为空的散列表实现的，所以这些操作的时间复杂度是O(1)。最方便的是多个集合类型键之间还可以进行并集、交集和差集运算。

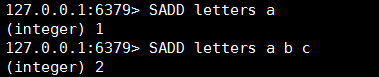
### 命令

1. **增加/删除元素**

SADD key member [member …]

SREM key member [member …]

SADD命令用来向集合中添加一个或多个元素，如果键不存在则会自动创建。因为在一个集合中不能有相同的元素，所以如果要加入的元素已经存在于集合中就会忽略这个元素。该命令的返回值是成功加入的元素数量（忽略的元素不计算在内）。



SREM命令用来从集合中删除一个或多个元素，并返回删除成功的个数。



由于元素“d”在集合中不存在，所以只删除了一个元素，返回值为1.

1. **获得集合中的所有元素**

SMEMBERS key

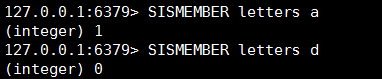
SMEMBERS命令会返回 集合中的所有元素。



1. **判断元素是否在集合中**

SISMEMBER key member

判断一个元素是否在集合中是一个时间复杂度为O(1)的操作，无论集合中有多少个元素，SISMEMBER命令始终可以极快地返回结果。当值存在时SISMEMBER命令返回1，当值不存在或键不存在时返回0.



1. **集合间运算**

SDIFF key [key …]

SINTER key [key …]

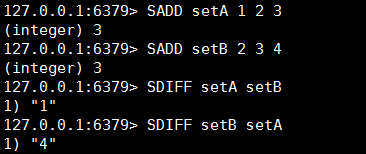
SUNION key [key …]

1. SDIFF命令用来对多个集合执行差集运算。集合A与集合B的差集表示为A - B，代表所有属于A且不属于B的元素构成的集合。

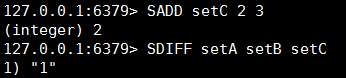
{1, 2, 3} – {2, 3, 4} = {1}

{2, 3, 4} – {1, 2, 3} = {4}

SDIFF命令的使用方法如下：



SDIFF命令支持同时传毒多个键



计算顺序是先计算setA - setB，再计算结果与setC的差集

1. SINTER命令用来对多个集合执行交集 运算。集合A与集合B的交集表示为A ∩ B，代表所有属于A且属于B的元素构成的集合。

{1, 2, 3} ∩ {2, 3, 4} = {2, 3}

SINTER命令的使用方法如下：



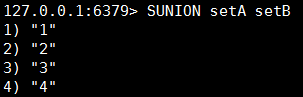
SINTER命令同样支持同时传入多个键。



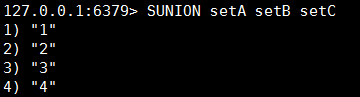
1. SUNION命令用来对多个集合执行并集运算。集合A与集合B的并集表示为A ∪ B，代表所有属于A或属于B的元素构成的集合。

{1, 2, 3} ∪ {2, 3, 4} = {1, 2, 3, 4}

SUNION命令的使用方法如下：



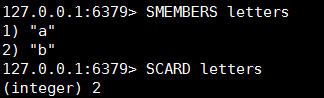
SUNION命令同样支持同时传入多个键。



1. **获得集合中元素个数**

SCARD key

SCARD命令用来获得集合中的元素个数。



1. **进行集合运算并将结果存储**

SDIFFSTORE destination key [key …]

SINTERSTORE destination key [key …]

SUNIONSTORE destination key [key …]

SDIFFSTORE命令和SDIFF命令功能一样，唯一的区别就是前者不会直接返回运算结果，而是将结果存储在destination键中。

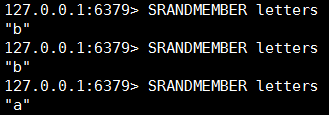
SDIFFSTORE命令常用于需要进行多步集合运算的场景中，如需要先计算差集再将结果和其他键计算交集。

SINTERSTORE和SUNIONSTORE命令与之类似。

1. **随机获得集合中的元素**

SRANDMEMBER key [count]

SRANDMEMBER命令用来随机从集合中获取一个元素。

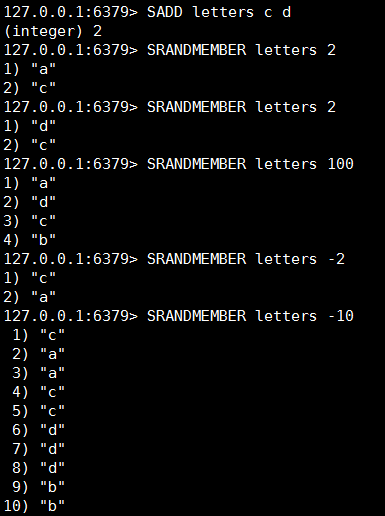


还可以传递count参数来一次随机获得多个元素，根据count的正负不同，具体表现也不同。

1. 当count为正数时，SRANDMEMBER会随机从集合里获得count个不重复的元素。如果count的值大于集合中的元素个数，则SRANDMEMBER会但会集合中的全部元素。
2. 当count为负数时，SRANDMEMBER会随机从集合里获得|count|个的元素，这些元素有可能相同。

首先我们现在letters集合中加入两个元素，此时letters集合中总共有4个元素。

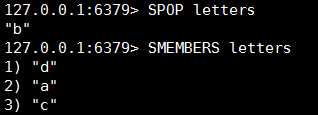
下面是使用不同参数对SRANDMEMBER命令进行测试：



1. **从集合中弹出一个元素**

SPOP key

SPOP命令与LPOP命令类似，但是由于集合类型的元素时无序的，所以SPOP命令会从集合中随机选择一个元素弹出。



## 有序集合类型

### 简介

有序集合类型在集合类型的基础上为集合中的每个元素都关联了一个分数，这使得我们不仅可以完成插入、删除和判断元素是否存在等集合类型支持的操作，还能够获得分数最高（或最低）的前N个元素、获得指定分数范围内的元素等于分数有关的操作。虽然集合中每个元素是不相同的，但是他们的分数却可以相同。

有序集合类型在某些方面和列表类型有些相似：

1. 二者都是有序的
2. 二者可以获得某一范围的元素

但是二者有着很大的区别，这使得它们的应用场景也是不同的：

1. 列表类型是通过链表实现的，获取靠近两端的数据速度极快，而当元素增多后，访问中间数据的速度会较慢，所以它更加适合实现如“新鲜事”或“日志”这样很少访问中间元素的应用。
2. 有序集合类型是使用散列表和跳跃表实现的，所以即使读取位于中间部分的数据速度也很快
3. 列表中不能简单地调用某个元素的位置，但是有序集合可以。
4. 有序集合要比散列类型更耗内存。

### 命令

1. **增加元素**

ZADD key score member [score member …]

ZADD命令用来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。ZADD命令的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）。

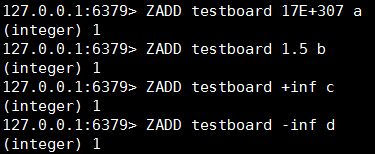
假如我们用有序集合模拟记分板，现在要记录Tom、Peter和David三名运动员的分数（分别是89分、67分和100分）：



这时我们发现Peter的分数录入有误，实际的分数应该是76分，可以用ZADD命令修改Peter的分数：



分数不仅可以是整数，还支持双精度浮点数：



其中+inf和-inf分别表示正无穷和负无穷。

1. **获得元素的分数**

ZSCORE key member

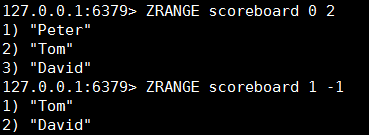


1. **获得排名在某个范围的元素列表**

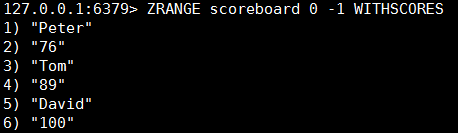
ZRANGE key start stop [WITHSCORES]

ZREVRANGE key start stop [WITHSCORES]

ZRANGE命令会按照元素分数从小到大的顺序返回索引从start到stop之前的所有元素（包含两端的元素）。ZRANGE命令与LRANGE命令十分相似，如索引都是从0开始，负数代表从后向前查找（-1表示最后一个元素）。

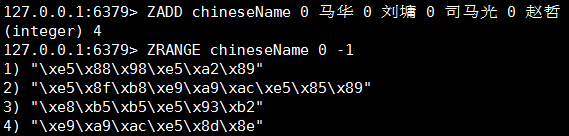


如果需要同时获得元素的分数的话可以在ZRANGE命令的尾部加上WITHSCORES参数 ，这时返回的数据格式就从“元素1，元素2，….，元素n”变为了“元素1，分数1，元素2，分数2，…，元素n，分数n”。



ZRANGE命令的时间复杂度为O(log n+m)（其中n为有序集合的基数，m为返回的元素个数）。

如果两个元素的分数相同，Redis会按照字典顺序（即”0”<”9”<”A”<”Z”<”a”<”z”这样的顺序）来进行排列、再进一步，如果元素的值是中文则取决于中文的编码方式。例如使用UTF-8编码：



可见此时Redis依然按照字典顺序排列这些元素。

ZREVRANGE命令和ZRANGE的唯一不同在于ZREVRANGE命令是按照元素分数从大到小的顺序给出结果的。

1. **获得指定分数范围的元素**

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

ZRANGEBYSCORE命令参数虽然都，但是都很好理解。该命令按照元素分数从小到大的顺序返回分数在min和max之间（包含min和max）的元素。



如果希望分数范围不包含端点值，可以在分数前加上“(”符号。例如，希望返回80分到100分的数据，可以含80分，但不包含100分：



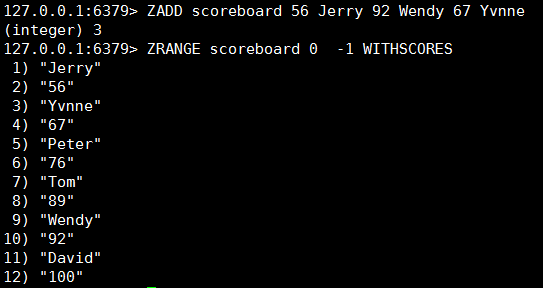
min和max还支持无穷大，同ZADD命令一样，-inf和+inf分别表示负无穷和正无穷。比如你希望得到所有分数高于80（不包含80分）的人的名单，但你不知道最高分是多少，这时就可以用上+inf了。



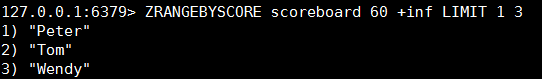
WITHSCORES参数的用法与ZRANGE命令一样。

LIMIT offset count与SQL中的用法基本相同，即在获得的元素列表的基础上向后偏移offset个元素，并且只获取前count个儿元素。

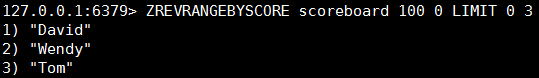
先添加了几个元素，然后获得所有的元素和分数



想获得分数高于60分的从第二个人开始的3个人



如果想获取分数低于或等于100分的前3个人，可以借助ZREVRANGEBYSCORE命令实现。对照前面的ZRANGE命令和ZREVRANGE命令之间的关系。需要注意的是ZREVRANGEBYSCORE命令不仅是按照元素分数从大往小的顺序给出结果的，而且它的min和max参数的顺序和ZRANGEBYSCORE命令是相反的。



1. **增加某个元素的分数**

ZINCRBY key increment menber

ZINCRBY命令可以增加一个元素的分数，返回值是更改后的分数。



increment也可以是个负数宝石减分，例如，给Jerry减分：



如果指定的元素不存在，Redis在执行命令前会先建立它并将它的分数赋值为0再执行操作。

1. **获得几个中元素的数量**

ZCARD key



1. **获得指定分数范围内的元素个数**

ZCOUNT key min max



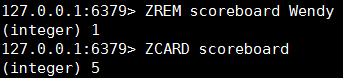
ZCOUNT命令的min和max参数的特性与ZRANGEBYSCORE命令中的一样



1. **删除一个或多个元素**

ZREM key member [member …]

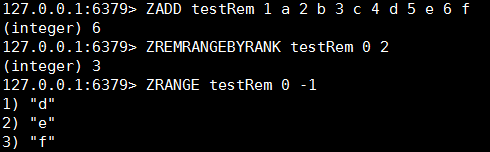
ZREM命令的返回值是成功删除的元素数量（不包含本来就不存在的元素）



1. **按照排名范围删除元素**

ZREMRANGEBYRANK key start stop

ZREMRANGEBYRANK命令按照元素分数从小到大的顺序（即索引0表示最小的值）删除处在指定排名范围内的所有元素，并返回删除的元素数量。



1. **按照分数范围删除元素**

ZREMRANGEBYSCORE key min max

ZREMRANGEBYSCORE命令会删除指定分数范围内的所有元素，参数min和max的特性和ZRANGEBYSCORE命令中的一样。返回值是删除的元素数量。



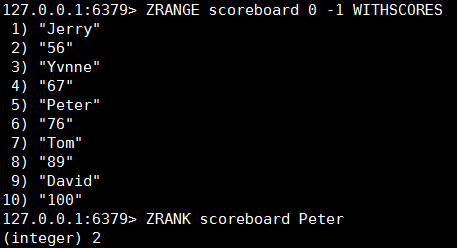


1. **获得元素的排名**

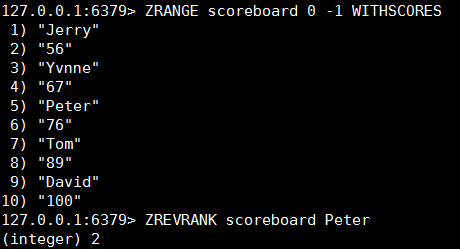
ZRANK key member

ZREVRANK key member

ZRANK命令会按照元素分数从小到大的顺序获得指定的元素的排名（从0开始，即分数最小的元素排名为0）。



ZREVRANK命令则相反（分数最大的元素排名为0 ）



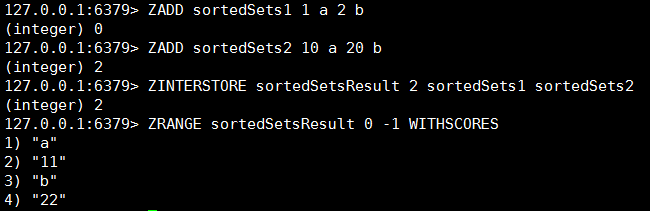
1. **计算有序集合的交集**

ZINTERSTORE destination numkeys key [key …] [WEIGHTS weight [weight …]] [AGGREGATE SUM|MIN|MAX]

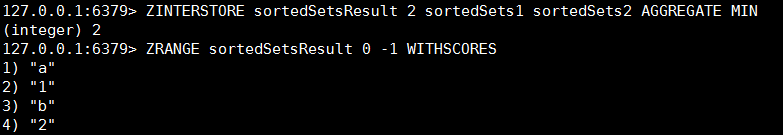
ZINTERSTORE命令用来计算多个有序集合的交集并将结果存储在destination键中（同样以有序几个类型存储），返回值为destination键中的元素个数。

destination键中元素的分数是由AGGREGATE参数决定的。

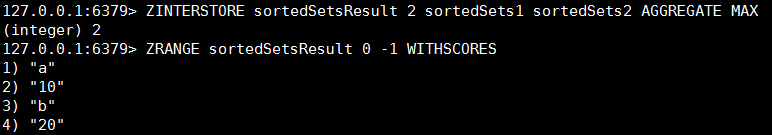
1. 当AGGREGATE是SUM时（也就是默认值），destination键中元素的分数是每个参与计算的集合中该元素分数的和。



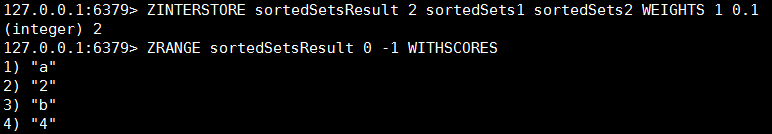
1. 当AGGREGATE是MIN时，destination键中元素的分数是每个参与计算的集合中该元素分数的最小值。



1. 当AGGREGATE是MAX时，destination键中元素的分数是每个参与计算的集合中该元素分数的最大值。



ZINTERSTORE命令还能够通过WEIGHTS参数设置每个集合的权重，每个集合在参与计算时分数会被乘上该集合的权重。



1. **计算有序集合的并集**

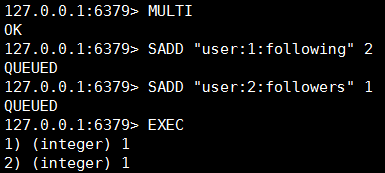
ZUNIONSTORE命令的作用是计算集合间的并集，它的用法与ZINTERSTORE命令一样。

# Redis事务

## 概述

Redis中的事务是一组命令的集合。事务同命令一样都是Redis的最小执行单位，一个事务中的命令要么都执行，要么都不执行。

事务的原理是先将属于一个事务的命令发送给Redis，然后再让Redis依次执行这些命令。



上面的代码演示了事务的使用方式。首先使用MULTI命令告诉Redis:“下面我发给你的命令属于同一事务，你先不要执行，而是把他们暂时存起来。”Redis回答：“OK”。

而我们发送了两个SADD命令来实现关注和被关注操作，可以看到Redis没有执行这些命令，而是返回QUEUED表示这两条命令已经进入等待执行的事务队列中。

当把所有要在同一事务中执行的命令都发给Redis后，我们使用EXEC命令告诉Redis将等待执行的事务队列中的所有命令（即刚才所有返回QUEUED的命令）按照发送顺序依次执行。EXEC命令的返回值就是这些命令的返回值组成的列表，返回值顺序和命令的顺序相同。

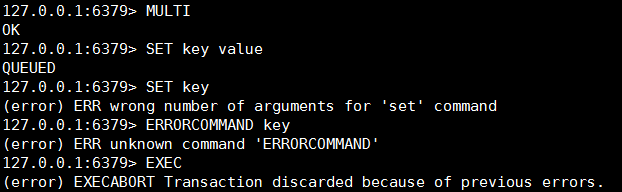
Redis保证一个事务中的所有命令要么都执行，要么都不执行。如果在发送EXEC命令前客户端断线，则Redis会清空事务队列，事务中的所有命令都不会执行。而一旦客户端发送了EXEC命令，所有的命令就都会被执行，即使此后客户端断线也没关系，因为Redis中已经记录了所有要执行的命令。

除此之外，Redis的事务还能保证一个事务内的命令依次执行而不被其他命令插入。

## 错误处理

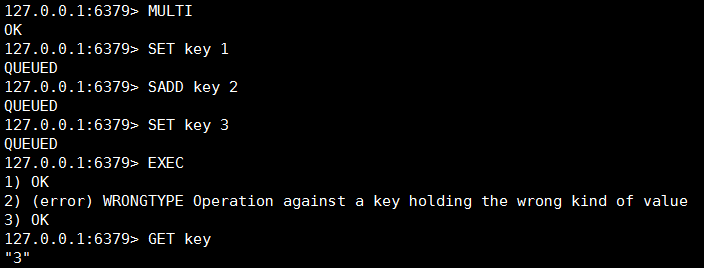
如果在一个事务中某个命令执行出现错误，Redis会怎么处理呢。首先需要知道什么原因会导致命令执行出错。

1. 语法所悟。语法错误指命令不存在或者命令参数的个数不对。



跟在MULTI命令后执行了3个命令：一个是正确的命令，成功地加入了事务队列；其余两个命令都有语法错误。执行EXEC命令后Redis就会直接返回错误，连语法正确的命令也不会执行。

1. 运行错误。运行错误指在命令执行时出现的错误，比如使用散列类型的命令操作集合类型的键，这种错误在实际执行之前Redis是无法发现的，所以在事务里这样的命令是会被Redis接受并执行的。如果事务里的一条命令出现了运行错误，事务里其他的命令依然会继续执行（包括出错命令之后的命令）。



Redis的事务没有关系数据库事务提供的回滚功能，为此开发者必须在事务执行出错后自己处理错误数据。

## WATCH命令

在一个事务中只有当所有命令都依次执行完后才能得到每个结果的返回值，可是有些情况下需要先获得一条命令的返回值，然后再根据这个值执行下一条命令。例如介绍INCR命令时曾经说过使用GET和SET命令自己实现incr函数会出现竞态条件，伪代码如下：

def incr($key)

$value = GET $key

if not $value

$value = 0

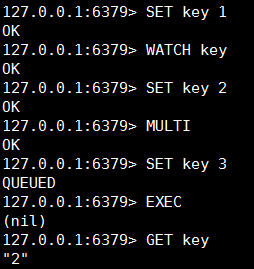
$value = $value + 1

SET $key, $value

return $value

如果使用事务来实现incr函数防止竞态条件，同样也无法做到增1的功能。因为事务中的每个命令的执行结果都是最后一起返回的，所以无法将前一条命令的结果作为后一条命令的参数，即在执行SET命令是无法获得GET命令的返回值。

WATCH命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行。监控一直持续到EXEC命令（事务中的命令是在EXEC之后才执行的，所以在MULTI命令后可以修改WATCH监控的键值）。



上例中在执行WATCH命令后、事务执行前修改了key的值（即SET key 2），所以最后事务中的命令SET key 3没有执行，EXEC命令返回空结果。

使用WACH命令就可以通过事务自己实现incr函数了。

def incr($key)

WATCH $key

$value = GET $key

if not $value

$value = 0

$value = $value + 1

MULTI

SET $key, $value

return = EXEC

return result[0]

因为EXEC命令返回值是多行字符串类型，所以代码中使用return[0]来获得其中第一个结果。

执行EXEC命令后会取消对所有键的监控，如果不想执行事务中的命令也可以使用UNWATCH命令来取消监控。比如我们要实现hsetxx函数，作用于HSETNX命令类似。只不过是仅当字段存在时才赋值。为了避免竞态条件我们使用事务来完成这一功能。

def hsetxx($key, $field, $value)

WATCH $key

$isFieldExists = HEXISTS $key, $field

if $isFieldExists is 1

MULTI

HSET $key, $field, $value

EXEC

else

UNWATCH

return $isFieldExists

在代码中会判断要赋值的字段是否存在，如果字段不存在的话就不执行事务中的命令，但需要使用UNWATCH命令来保证下一个事务的执行不会受到影响。

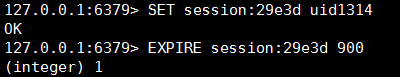
# 过期时间

## 命令介绍

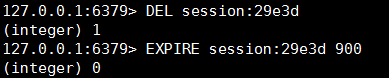
在实际的开发中经常会遇到一些有时效的数据，比如限时优惠活动、缓存或验证码等，过了一定的时间就需要删除这些数据。在关系型数据库中一般需要额外的一个字段记录到期时间，然后定期检测删除过期数据。而在Redis中可以使用EXPIRE命令设置一个键的过期时间，到时候Redis会自动删除它。

EXPIRE key seconds

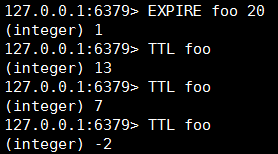
其中seconds参数表示键的过期时间，单位是秒。如要想让session:29e3d键在15分钟后被删除。



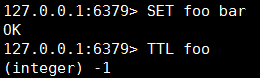
EXPIRE命令返回1表示设置成功，返回0则表示键不存在或者设置失败。



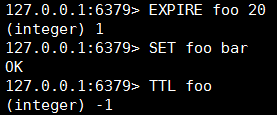
如果想知道一个键还有多久的时间会被删除，可以使用TTL命令。返回值是键的剩余时间（单位是秒）



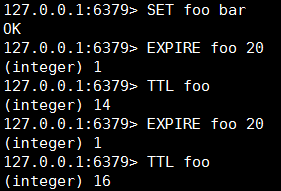
当键不存在时TTL命令会返回-2。如果没有为键设置过期时间的情况下会返回-1。



除了PERSIST命令之外，使用SET或GETSET命令为键赋值也会同时清除键的过期时间。



使用EXPIRE命令会重新设置键的过期时间。



其他只对键值进行操作的命令（如INCR、LPUSH、HSET、ZREM）均不会影响键的过期时间。

EXPIRE命令的seconds参数必须是整数，所以最小单位是1秒。如果想要更精确的控制键的过期时间应该使用PEXPIRE命令，PEXPIRE命令与EXPIRE的唯一区别是前者的时间单位是毫秒。对应地可以使用PTTL命令以毫秒为单位返回键的剩余时间。

提示：如果使用WATCH命令检测一个拥有过期时间的键，该键时间到期自动删除并不会被WATCH命令认为该键被改变。

另外还有两个相对不太常用的命令：EXPIREAT和PEXPIREAT。

EXPIREAT命令与EXPIRE命令的差别在于前者使用Unix时间作为第二个参数表示键的过期时间。PEXPIREAT命令与EXPIREAT命令的区别是前者时间单位是毫秒。

## 实现缓存

为了提高网站的负载能力，常常需要将一些访问频率较高但是对CPU或IO资源消耗较大的操作的结果缓存起来，并希望让这些缓存过一段时间自动过期。但是当服务器内存有限时，如果大量地使用缓存键且过期时间设置得过长就会导致Redis沾满内存；另一方面如果为了防止Redis占用内存过大而将缓存键的过期时间设得太短，就可能导致缓存命中率过低并且大量内存白白闲置。实际开发中会发现很难为缓存键设置合理的过期时间，为此可以限制Redis能够使用的最大内存，并让Redis按照一定的规则淘汰不需要的缓存键，这种方式在只将Redis用作缓存系统时非常实用。

具体的设置方法为：修改配置文件的maxmemory参数，限制Redis最大可用内存大小（单位是字节），当超出了这个限制时Redis会依据maxmemory-policy参数指定的策略来删除不需要的键直到Redis占用的内存小于指定内存。

maxmemory-policy支持的规则如下表。其中的LRU（Least Recently Used）算法即“最近最少使用”，其认为最近最少使用的键在未来一段时间内也不会被用到，即当需要空间时这些键是可以被删除的。

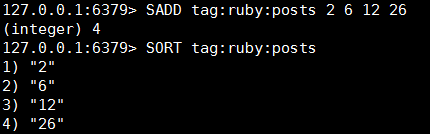
|  |  |
| --- | --- |
| Redis支持的淘汰键的规则 | |
| 规则 | 说明 |
| volatile-lru | 使用LRU算法删除一个键（只对设置了过期时间的键） |
| allkeys-lru | 使用LRU算法删除一个键 |
| volatile-random | 随机删除一个键（只对设置了过期时间的键） |
| allkeys-random | 随机删除一个键 |
| volatile-ttl | 删除过期时间最近的一个键 |
| noeviction | 不删除键，只返回错误 |

如当maxmemory-policy设置为allkeys-lru时，一旦Redis占用的内存超过了限制值，Redis会不断地删除数据库中最近最少使用的键，直到占用的内存小于限制值。

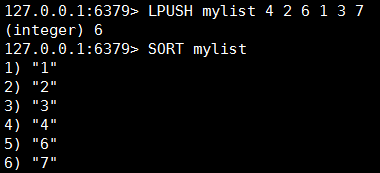
# Redis排序

## SORT命令

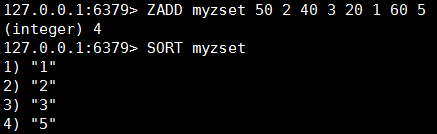
SORT命令可以对列表类型、集合类型和有序集合类型键进行排序，并且可以完成与关系数据库中的连接查询相类似的任务。



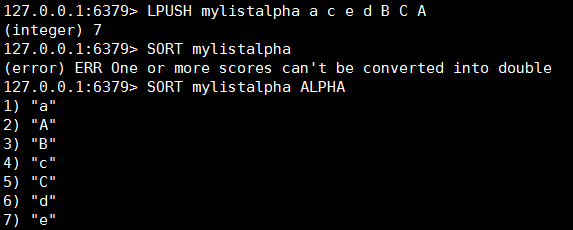
除了集合类型，SORT命令还可以对列表类型和有序集合类型进行排序：



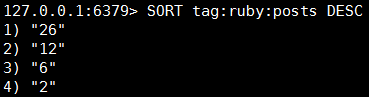
在对有序集合类型排序时会忽略元素的分数，只针对元素自身的值进行排序。



除了可以排列数字外，SORT命令还可以通过ALPHA参数实现按照字典顺序排列非数字元素。



SORT命令默认是按照从小到大的顺序排列的，SORT命令的DESC参数可以实现将元素按照从大到小的顺序排列



SORT命令还支持LIMIT参数来返回指定范围的结果。用法和SQL语句一样，LIMIT offset count，表示跳过前offset个元素怒并获取之后的count个元素。

SORT命令的参数可以组合使用



## BY参数

很多情况下列表（或集合、有序集合）中存储的元素值代表的是对象的ID，单纯对这些ID自身排序有时意义并不大。更多的时候我们希望根据ID对应的对象的某个属性进行排序。这种情况我们需要使用SORT命令的另一个强大的参数：BY。

BY参数的语法为BY参考键。其中参考键可以是字符串类型键或者是散列类型键的某个字段（表示为键名—>字段名）。如果提供了BY参数，SORT命令将不再依据元素自身的值进行排序，而是对每个元素使用元素的值替换参考键中的第一个“\*”，并获取其值，然后依据该值对元素排序。

127.0.0.1:6379> SORT tag:ruby:posts BY post:\*->time DESC

1) “12”

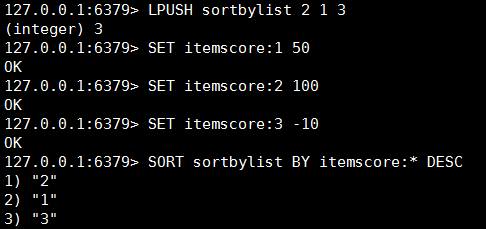
2) “26”

3) “6”

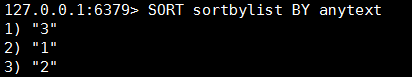
4) “2”

在上例中SORT命令会读取post:2、post:6、post:12、post:26几个散列键中的time字段的值并以此决定tag:ruby:posts键中各个文章ID的顺序。

除了散列类型之外，参考键还可以是字符串类型

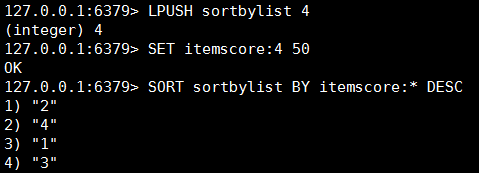


当参考键名不包含“\*”时（即常量键名，与元素值无关），SORT命令将不会执行排序操作，因为Redis认为这种情况是没有意义的（因为所有要比较的值都一样）



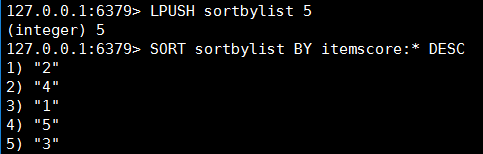
例子中anytext是常量键名（甚至anytext键可以不存在），此时SORT的结果与LRANGE的结果相同，没有执行排序操作。在不需要排序但需要借助SORT命令获得与元素相关联的数据时，常量键名是很有用的。

如果几个元素的参考键值想沟通，则SORT命令会再比较元素本身的值来决定元素的顺序。



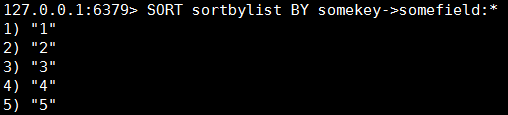
实例中元素“4”的参考键itemscore:4的值和元素“1”的参考键itemscore:1的值都是50，所以SORT命令会再比较“4”和“1”元素本身的大小来决定二者的顺序。

当某个元素的参考键不存在时，会默认参考键的值为0



上例中“5”排在“3”的前面，是因为“5”的参考键不存在，所以默认为0，而“3”的参考键值为-10.

补充：参考键虽然支持散列类型，但是“\*”只能在“->”符号前面（即键名部分）才有用，在“->”后（即字段名部分）会被当成字段名本身而不会作为占位符被元素的值替换，即常量键名。但是实际运行时会发现一个有趣的结果：



上面提到了当参考键名是常量键名时SORT命令将不会执行排序操作，然而上例中却进行了排序，而且只对元素本身进行排序。这时因为Redis判断参考键名是不是长了键名的方式是判断参考键名中是否包含“\*”，而somekey->somefield:\*中包含“\*”所以不是常量键名。所以在排序的时候Redis对每个元素都会读取键somekey中的somefield:\*字段（“\*”不会被替换），无论能否获得其值，每个元素的参考键值是相同的，所以Redis会按照元素本身的大小排序。

## GET参数

SORT命令的GET参数不影响排序，它的作用是使SORT命令的返回结果不再是元素自身的值，而是GET参数指定的键值。GET参数的规则和BY参数一样，GET参数也支持字符串类型和散列类型的键，并使用“\*”作为占位符。

例如：要返回在排序后直接返回ID对应的文章标题，可以这样写：

127.0.0.1:6379> SORT tag:ruby:posts BY post:\*->time DESC GET post:\*->title

1) “Windows 8 app designs”

2) “RethinkDB – An open-source distributed database built with love”

3) “Uses for cURL”

4) “The Nature of Ruby”

在一个SORT命令中可以有多个GET参数（而BY参数只能有一个），所以还可以这样用：

127.0.0.1:6379> SORT tag:ruby:posts BY post:\*->time DESC GET post:\*->title GET post:\*->time

1) “Windows 8 app designs”

2) “1352620100”

3) “RethinkDB – An open-source distributed database built with love”

4) “1352620000”

5) “Uses for cURL”

6) “1352619600”

7) “The Nature of Ruby”

8) “1352619200”

可见有N个GET参数，每个元素返回的结果就有N行。这时有个问题：如果还需要返回文章ID该怎么办，可以使用GET #。

127.0.0.1:6379> SORT tag:ruby:posts BY post:\*->time DESC GET post:\*->title GET post:\*->time GET #

1) “Windows 8 app designs”

2) “1352620100”

3) “12”

4) “RethinkDB – An open-source distributed database built with love”

5) “1352620000”

6) “26”

7) “Uses for cURL”

8) “1352619600”

9) “6”

10) “The Nature of Ruby”

11) “1352619200”

12) “2”

GET #会返回元素本身的值。

## STORE参数

默认情况下SORT会直接返回排序结果，如果希望保存排序结果，可以使用STORE参数。如希望把结果保存在sort.result键中：

127.0.0.1:6379> SORT tag:ruby:posts BY post:\*->time DESC GET post:\*->title GET post:\*->time GET # STORE sort.result

(integer) 12

127.0.0.1:6379> LRANGE sort.result 0 -1

1) “Windows 8 app designs”

2) “1352620100”

3) “12”

4) “RethinkDB – An open-source distributed database built with love”

5) “1352620000”

6) “26”

7) “Uses for cURL”

8) “1352619600”

9) “6”

10) “The Nature of Ruby”

11) “1352619200”

12) “2”

保存后的键的类为散列类型，如果键已经存在则会覆盖它。加上STORE参数后SORT命令的返回值为结果的个数。

STORE参数常用来结合EXPIRE命令缓存排序结果，如下面的伪代码：

#判断是否存在之前排序结果的缓存

$isCacheExists = EXISTS cache.sort

if $isCacheExists is 1

#如果存在则直接返回

return LRANGE cache.sort, 0, -1

else

#如果不存在，则使用SORT命令排序并将结果存入cache.sort键中作为缓存

$sortResult = SORT some.list STORE cache.sort

#设置缓存的过期时间为10分钟

EXPIRE cache.sort, 600

#返回排序结果

return $sortResult

## 性能优化

SORT是Redis中最强大最复杂的命令之一，如果使用不好很容易成为性能瓶颈。SORT命令的时间复杂度是O(n+mlog(m))，其中n表示要排序的列表（集合或有序集合）中的元素个数，m表示要返回的元素个数。当n较大的时候SORT命令的性能相对较低，并且Redis在排序前会建立一个长度为n的容器来存储待排序的元素，虽然是一个临时的过程，但如果同时进行较多的大数据量排序操作则会严重影响性能。

所以开发中使用SORT命令时需要注意以下几点：

1. 尽可能减少待排序键中元素的数量（使n尽可能小）
2. 使用LIMIT参数只获取需要的数据（使m尽可能小）
3. 如果要排序的数据数量较大，尽可能使用STORE参数将结果缓

# 消息通知

## 任务队列

当页面需要进行如发送邮件、复杂数据运算等耗时较长的操作时会阻塞页面的渲染。为了避免用户等待太久，应该使用独立的线程来完成这类操作。不过一些编程语言或框架不容易实现多线程，这时很容易就会想到通过其他进程来实现。

任务队列顾名思义就是“传递任务的队列”。与任务队列进行交互的实体有两类，一类是生产者（producer），另一类是消费者（consumer）。生产者会将需要处理的任务放入任务队列中，而消费者则不断地从任务队列中读入任务信息并执行。

使用任务队列的好处:

1. 松耦合

生产者和消费者无需知道彼此的实现细节，只需要约定好任务的描述格式。这使得生产者和消费者可以由不同的团队使用不同的编程语言编写。

1. 易于扩展

消费者可以有多个，而且可以分布在不同的服务器中。借此可以轻易地降低单台服务器的负载。

## 使用Redis实现任务队列

说到队列很自然就能想到Redis的列表类型，使用LPUSH和RPOP命令实现队列的概念。如果要实现任务队列，只需要让生产者将任务使用LPUSH命令加入到某个键中，另外一边让消费者不断地使用RPOP命令从该键中取出任务即可。

例如：完成发邮件的任务需要知道收件地址、邮件主题和邮件正文。所以生产者需要将这3个信息组成对象并序列化成字符串，然后将其加入到任务队列中。而消费者则循环从任务队列中拉取任务，就像如下伪代码：

#无限循环读取任务队列中的内容

loop

$task = RPOP queue

if $task

#如果任务队列中有任务则执行它

execute($task)

else

#如果没有则等待1秒以免过于频繁地请求数据

wait 1 second

到此一个使用Redis实现的简单的任务队列就写好了。不过还有一点不完美的地方：当任务队列中没有任务时消费者每秒都会调用一次RPOP命令查看是否有新任务。如果可以实现一旦有新任务加入任务队列就通知消费者就好了。其实借助BRPOP命令就可以实现这样的需求。

BRPOP命令和RPOP命令相似，唯一的区别是当列表中没有元素时BRPOP命令会一直阻塞住连接，直到有新元素加入。BRPOP命令接收两个参数，第一个是键名，第二个是超时时间，单位是秒。当超时了此时间仍然没有获得新元素就会返回nil。超时时间设置为0表示不限制等待的时间，即如果没有新元素加入列表就会永远阻塞下去。当获得一个元素后BRPOP命令返回两个值，分别是键名和元素值。

除了BRPOP命令外，Redis还提供了BLPOP命令，和BRPOP的区别在与从队列取元素时BLPOP会从队列左边取。

## 优先级队列

如果在任务队列中同时存在多个任务，那么就会存在那个任务先执行的问题，这就需要实现一个优先级队列的情况。

BRPOP命令可以同时接收多个键，其完整的命令格式为:

BLPOP key [key …] timeout

例如打开两个redis-cli客户端模拟队列。

实例A：

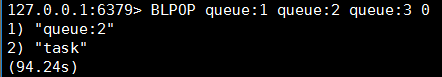


此时实例A一直处于阻塞状态

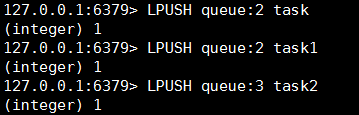
实例B：



此时会看到实例A中会立即返回结果



如果多个键都有元素则按照从左到右的顺序取第一个键中的一个元素。我们先在queue:2和queue:3中各加入一个元素：



然后执行BRPOP命令：



借此特性可以实现区分优先级的任务队列。我们分别使用queue:confirmation.email和queue:notification.email两个键存储发送确认邮件和发送通知邮件两种任务，然后将消费代码改为：

loop

$task = BRPOP queue:confirmation.email, queue:notification.email, 0

execute($task[1])

这时一旦发送确认邮件的任务被加入到queue:confirmation.email队列中，无论queue:notification.email还有多少任务，消费者都会优先完成发送确认邮件的任务。

## 发布/订阅 模式

除了实现任务队列外，Redis还提供了一组命令可以让开发者实现“发布/订阅”模式。“发布/订阅”模式同样可以实现进程间的消息传递。

“发布/订阅”模式中包含两种角色，分别是发布者和订阅者。订阅者可以订阅一个或若干个频道，而发布者可以向指定的频道发送消息，所有订阅此频道的订阅者都会收到此消息。

发布者发布消息的命令是PUBLISH，语法是

PUBLISH channel message

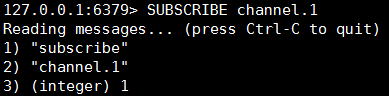


这样消息就发出了。PUBLISH命令的返回值表示接收到这条消息的订阅者数量。因为此时没有客户端订阅channel.1，所以返回0。发出去的消息不会被持久化，也就是说当有客户端订阅channel.1后只能收到后续发布到该频道的消息，之前发送的就收不到了。

订阅频道的命令是SUBSCRIBE，可以同时订阅多个频道，语法是：

SUBSCRIBE channel [channel …]

现在新开一个redis-cli实例B，用它来订阅channel.1



执行SUBSCRIBE命令后客户端会进入订阅状态，处于此状态下客户端不能使用除SUBSCRIBE、UNSUBSCRIBE、PSUBSCRIBE和PUNSUBSCRIBE这4个属于“发布/订阅”模式的命令之外的命令，否则会报错。

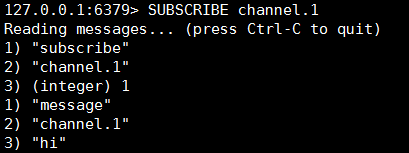
进入订阅状态后客户端可能收到3种类型的回复。每种类型的回复都包含3个值，第一个值是消费的类型，根据消息类型的不同，第二、三个值的含义也不同。消息类型可能的取值有以下3个。

1. subscribe。表示订阅成功的反馈信息。第二个值是订阅成功的频道名称，第三个值是当前客户端订阅的频道数量
2. message。这个类型的回复是我们最关心的，它表示接收到的消息。第二个值表示生产消息的频道名称，第三个值是消息的内容。
3. unsubscribe。表示成功取消订阅某个频道。第二个值是对应的频道名称，第三个值是当前客户端订阅的频道数量，当此值为0时客户端会退出订阅状态，之后就可以执行其他非“发布/订阅”模式的命令了。

此时我们在客户端A中向channel.1发送一条消息



返回值为1表示有一个客户端订阅了channel.1，此时在客户端B中会收到类型为message的回复



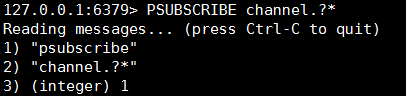
使用UNSUBSCRIBE命令可以取消订阅指定的频道。

UNSUBSCRIBE [channel [channel …]]

如果不指定频道则会取消订阅所有频道。

## 按照规则订阅

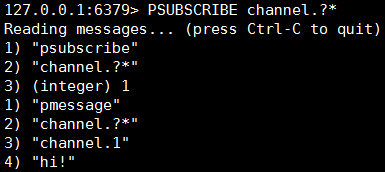
除了可以使用SUBSCRIBE命令订阅指定名称的频道，还可以使用PSUBSCRIBE命令订阅指定的规则。规则支持glob风格通配符格式，下面我们新打开一个redis-cli实例C进行演示：



规则channel.?\*可以匹配channel.1和channel.10，但不会匹配channel.。这时在实例A中发布消息：



返回结果为2是因为实例B和实例C两个客户端都订阅了channel.1频道。实例C接收到的回复是：



第一个值"pmessage"表示这条消息是通过PSUBSCRIBE命令订阅频道而收到的，第二个值"channel.?\*"表示订阅时使用的通配符，第三个值"channel.1"表示实际收到消息的频道命令，第四个值"hi!"则是消息内容。

提示：使用PSUBSCRIBE命令可以重复订阅一个频道，如某客户端执行了PSUBSCRIBE channel.? channel.?\*，这时向channel.2发布消息后该客户端会收到两条消息，而同时PUBLISH命令返回的值也是2而不是1.同样的，如果有另一个客户端执行了SUBSCRIBE channel.10和PSUBSCRIBE channel.?\*的话，向channel.10发送命令该客户端也会收到两条消息（但是是两种类型：message和pmessage），同时PUBLISH命令会返回2。

PUNSUBSCRIBE命令可以退订指定的规则。

PUNSUBSCRIBE [pattern [pattern …]]

如果没有参数则胡退订所有规则

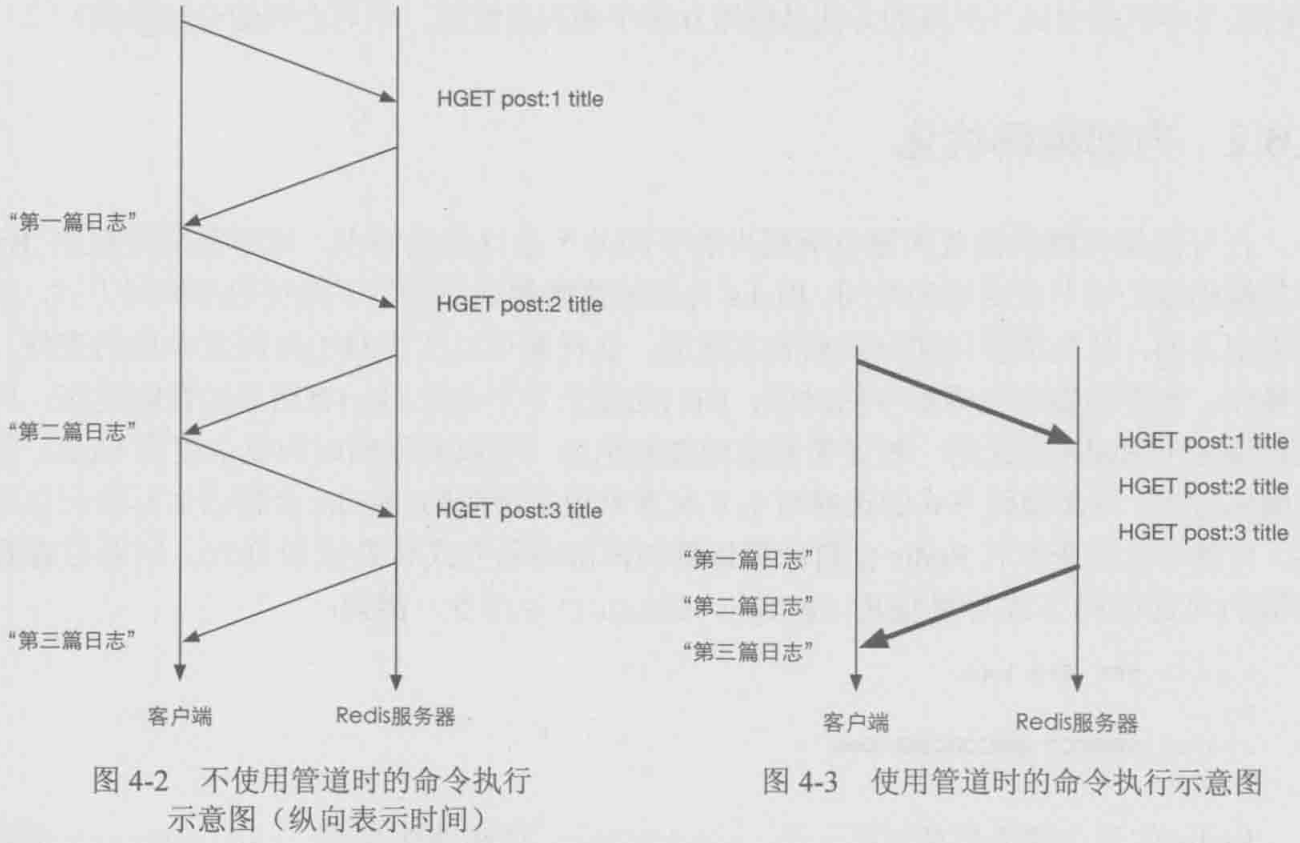
注意：使用PUNSUBSCRIBE命令只能退订通过PSUBSCRIBE命令订阅的规则，不会影响直接通过SUBSCRIBE命令订阅的频道；同样UNSUBSCRIBE命令也不会影响通过PSUBSCRIBE命令订阅的规则，另外容器出错的一点是使用PUNSUBSCRIBE命令退订某个规则时不会将其中的通配符展开，而是进行严格的字符串匹配，所以PUNSUBSCRIBE无法退订channel.\*规则，而必须使用PUNSUBSCRIBE channel.\*才能退订。

# 管道技术

客户端和Redis使用TCP协议连接。不论是客户端向Redis发送命令还是Redis向客户端返回命令的执行结果，都需要经过网络传输，这两个部分的总耗时称为往返延时。根据网络性能不同，往返时延也不同，大致来说到本地回环地址的往返时延在数量级上相当于Redis处理一条简单明了的时间。如果执行较多的命令，每个命令的往返时延累加起来对性能还是有一定影响的。

在执行多个命令时每条命令都需要等待上一条命令执行完（即收到Redis的返回结果）才能执行，即使命令不需要上一条命令的执行结果。如要获得post:1、post:2和post:3这3个键中的title字段，需要执行3条命令。

Redis的底层通信协议对管道提供了支持。通过管道客户一次性发送多条命令并在执行完后一次性将结果返回，当一组命令中每条命令都不依赖于之前命令的执行结果时就可以将这组命令一起通过管道发出。管道通过减少客户端与Redis的通信次数来实现降低往返时延累计值的目的。



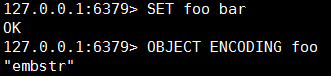
# 节省空间

## 精简键名和键值

精简键名和键值是最直观的减少内存占用的方式，如将键名very.important.person:20还成VIP:20.当然精简键名一定要把握好尺度，不能单纯为了节约空间而使用不易理解的键名。

## 内部编码优化

有时候仅凭精简键名和键值所减少的空间并不足以满足需求，这时就需要根据Redis内部编码规则来节省更多的空间。Redis为每种数据类型都提供了两种内部编码方式，以散列类型为例，散列类型是通过散列表实现的，这样就可以实现O(1)时间复杂度的查找、赋值操作，然而当键中元素很少的时候，O(1)的操作并不会比O(n)有明显的性能提高，所以这种情下Redis会采用一种更为紧凑但性能稍差（获取元素的时间复杂度为O(n)）的内部编码方式。内部编码方式的选择对于开发者来说是透明的，Redis会根据实际情况自动调整。当键中元素变多时Redis会自动将该键的内部编码方式转换成散列表。如果想查看一个键的内部编码方式可以使用OBJECT ENCODING命令。



Redis的每个键值都是使用一个redisObject结构体保存的，redisObject的定义如下：

typedef struct redisObject {

unsigned type:4;

unsigned notused:2; /\* Not used \*/

unsigned encoding:4;

unsigned lru:22; /\* lru time (relative to server.lruclock) \*/

int refcount;

void \*ptr;

} robj;

其中type字段表示的是键值的数据类型，取值可以是如下内容：

#define REDIS\_STRING 0

#define REDIS\_LIST 1

#define REDIS\_SET 2

#define REDIS\_ZSET 3

#define REDIS\_HASH 4

encoding字段表示的就是Redis键值的内部编码方式，取值可以是：

#define REDIS\_ENCODING\_RAW 0 /\* Raw representation \*/

#define REDIS\_ENCODING\_INT 1 /\* Encoded as integer \*/

#define REDIS\_ENCODING\_HT 2 /\* Encoded as hash table \*/

#define REDIS\_ENCODING\_ZIPMAP 3 /\* Encoded as zipmap \*/

#define REDIS\_ENCODING\_LINKEDLIST 4 /\* Encoded as regular linked list \*/

#define REDIS\_ENCODING\_ZIPLIST 5 /\* Encoded as ziplist \*/

#define REDIS\_ENCODING\_INTSET 6 /\* Encoded as intset \*/

#define REDIS\_ENCODING\_SKIPLIST 7 /\* Encoded as skiplist \*/

#define REDIS\_ENCODING\_EMBSTR 8 /\* Encoded as string encoding \*/

各个数据类型可能采用的内部编码方式以及相应的OBJECT ENCODING命令执行结果如下表所示：

|  |  |  |
| --- | --- | --- |
| 数据类型 | 内部编码方式 | OBJECT ENCODING命令结果 |
| 字符串类型 | REDIS\_ENCODING\_RAW | “raw” |
| REDIS\_ENCODING\_INT | “int” |
| REDIS\_ENCODING\_EMBSTR | “embstr” |
| 散列类型 | REDIS\_ENCODING\_HT | “hashtable” |
| REDIS\_ENCODING\_ZIPLIST | “ziplist” |
| 列表类型 | REDIS\_ENCODING\_LINKEDLIST | “linkedlist” |
| REDIS\_ENCODING\_ZIPLIST | “ziplist” |
| 集合类型 | REDIS\_ENCODING\_HT | “hashtable” |
| REDIS\_ENCODING\_INTSET | “intset” |
| 有序集合类型 | REDIS\_ENCODING\_SKIPLIST | “skiplist” |
| REDIS\_ENCODING\_ZIPLIST | “ziplist” |

### 字符串类型

Redis使用一个sdshdr类型的变量来存储字符串，而redisObject的ptr字段只想的是该变量的地址。sdshdr的定义如下：

struct sdshdr {

int len;

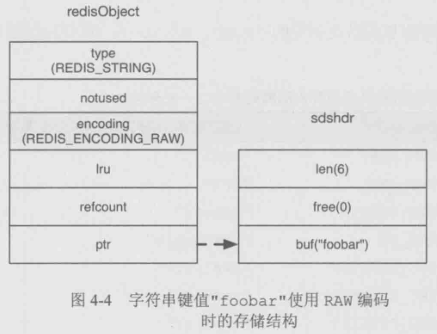
int free;

char buf[];

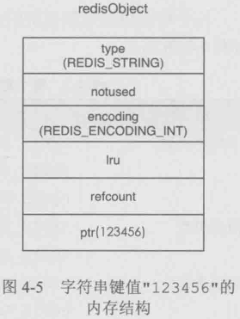
};

其中len字段表示的是字符串的长度，free字段表示buf中的剩余空间，而buf字段存储的才是字符串的内容。

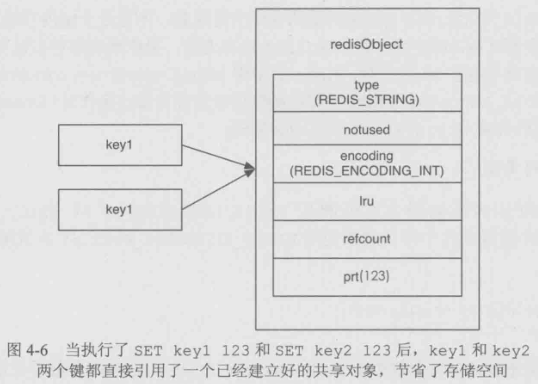
所以当执行SET key foobar时，存储键值需要占用的空间时sizeof(redisObject) + sizeof(sdshdr) + strlen(“foobar”) = 30字节。



而当键值内容可以用一个64位有符号整数表示时，Redis会将键值转换成long类型来存储。如SET key 123456，实际占用的空间时sizeof(redisObject) = 16字节，比存储”foobar”节省了一办的存储空间。



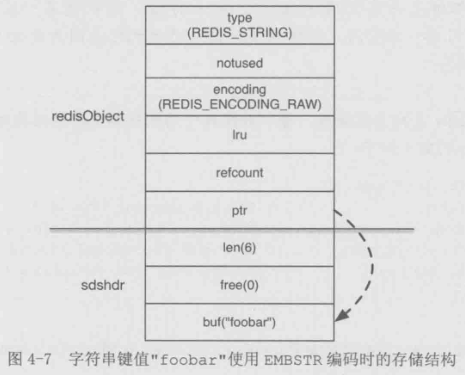
redisObject中的refcount字段存储的是该键值被引用数量，即一个键值可以被多个键引用。Redis启动后会预先设置10000个分别存储从0到9999这些数字的redisObject类型变量作为共享对象，如果要设置的字符串键值在这10000个数字内（如SET key1 123）则可以直接引用共享对象而不用再建立一个redisObject了，也就是说存储键值占用的空间时0字节。



由此可见，使用字符串类型键存储对象ID这种小数字是非常节省存储空间的，Redis只需存储键名和一个对共享对象的引用即可。

提示：当通过配置文件参数maxmemory设置了Redis可用的最大空间大小时，Redis不会使用共享对象，因为对于每一个键值都需要使用一个redisObject来记录其LRU信息。

Redis3.0中加入的REDIS\_ENCODING\_EMBSTR的字符串编码方式，该编码方式与REDIS\_ENCODING\_RAW类似，都是基于sdshdr实现的，只不过sdshdr的结构体与其对应的分配在同一块连续的内存空间中



使用REDIS\_ENCODING\_EMBSTR编码存储字符串后，不论是分配内存还是释放内存，所需要的操作都从两次减少为一次，而且由于内存连续，操作系统缓存可以更好地发挥作用。当键值内容不超过39字节时，Redis会采用REDIS\_ENCODING\_EMBSTR编码，同时对使用REDIS\_ENCODING\_EMBSTR编码的键值进行任何修改操作时（如APPEND命令），Redis会将其转换成REDIS\_ENCODING\_RAW编码。

### 散列类型

散列类似的内部编码方式可能是REDIS\_ENCODING\_HT或REDIS\_ENCODING\_ZIPLIST。在配置文件中可以定义使用REDIS\_ENCODING\_ZIPLIST方式编码散列类型的时机：

hash-max-ziplist-entries 512

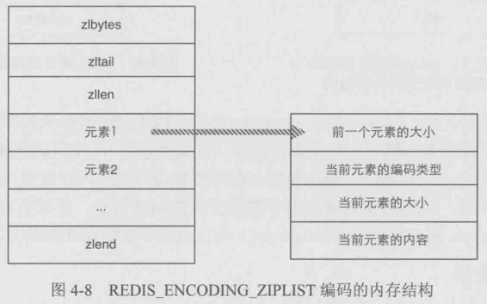
hash-max-ziplist-value 64

当散列类型键的字段个数少于hash-max-ziplist-entries参数值且每个字段名和字段值的长度都小于hash-max-ziplist-value参数值（单位为字节）时，RedisIU会使用REDIS\_ENCODING\_ZIPLIST来存储该键，否则就会使用REDIS\_ENCODING\_HT。转换过程是透明的，每当键值变更后Redis都会自动判断是否满足条件来完成转换。

REDIS\_ENCODING\_HT编码即散列表，可以实现O(1)时间复杂度的赋值取值等操作，其字段和字段值都是使用redisObject存储的，所以前面讲到的字符串类型键值的优化方法同样适用于散列类型键的字段和字段值。

提示：Redis的键值对存储也是通过散列表实现的，与REDIS\_ENCODING\_HT编码方式类似，但键名并非使用redisObject存储，所以键名“123456”并不会比“abcdef”占用更少的空间，之所以不对键名进行优化是因为绝大多数情况下键名不会是纯数字。

REDIS\_ENCODING\_ZIPLIST编码类型是一种紧凑的编码格式，它牺牲了部分读取性能以换取极高的空间利用率，适合在元素较少的时候。该编码类型同样还在列表类型和有序集合类型中使用。REDIS\_ENCODING\_ZIPLIST编码结构如下图：



其中zlbytes是uint32\_t类型，表示整个结构占用的空间。zltail也是uint32\_t类似，表示到最后一个元素的偏移，记录zltail使得程序可以直接定位到尾部元素而无需遍历整个结构，执行从尾部弹出（对列表类型而言）等操作时速度更快。zllen是uint16\_t类型，存储的是元素的数量。zlend是一个单字节标识，标记结构的末尾，值永远是255。

在REDIS\_ENCODING\_ZIPLIST中每个元素由4部分组成。

第一部分用来存储前一个元素的大小以实现倒序查找，当前一个元素的大小小于254字节时第一部分占用1个字节，否则会占用5个字节。

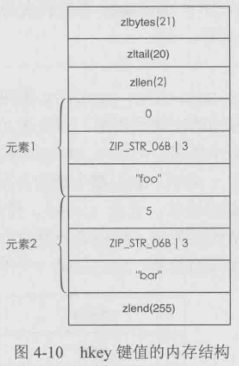
第二、三部分分别是元素的编码类型和元素的大小，当元素的大小小于或等于63个字节时，元素的编码类型是ZIP\_STR\_06B（即0<<6），同时第三个部分用6个二进制位来记录元素的长度，所以第二、三个部分总占用空间时1字节。当元素的大小大于63且小于或等于16383字节时，第二、三个部分总占用空间是2字节。当元素的大小大于16383字节时，第二、三个部分总占用空间时5字节。

第四部分是元素的实际内容，如果元素可以转换成数字的话Redis会使用相应的数字类型来存储以节省空间，并且第二、三部分来表示数字的类型（int16\_t、int32\_t等）。

使用REDIS\_ENCODING\_ZIPLIST编码存储散列类型时元素的排列方式是：元素1存储字段1，元素2存储字段值1以此类推。



例如，当执行命令HSET hkey foo bar命令后，hkey键值的内存结果如下图：



下次需要执行HSET hkey foo anothervalue时Redis需要从头开始找到值为foo的元素（查找时每次都会跳过一个元素以保证只查找字段名），找到后删除其下一个元素，并将新值anothervalue插入。删除和插入都需要移动后面的内存数据，而且查找操作也需要遍历才能完成，可想而知当散列键中数据多时性能将很低，所以不宜将hash-max-ziplist-entries和hash-max-ziplist-value两个参数设置得很大。

### 列表类型

列表类型的内部编码方式可能是REDIS\_ENCODING\_LINKEDLIST或REDIS\_ENCODING\_ZIPLIST。同样在配置文件中可以定义使用REDIS\_ENCODING\_ZIPLIST方式编码的时机：

list-max-ziplist-entries 512

list-max-ziplist-value 64

具体转换方式和散列类型一样。

REDIS\_ENCODING\_LINKEDLIST编码方式即双向链表，链表中的每个元素是用redisObject存储的，所以此种编码方式下元素值的优化方法与字符串类型的键值相同。

而使用REDIS\_ENCODING\_ZIPLIST编码方式时具体的表现和散列类型一样，由于REDIS\_ENCODING\_ZIPLIST编码方式同样支持倒序访问，所以采用此种编码方式时获取两端的数据依然较快。

Redis最新的开发版本新增了REDIS\_ENCODING\_QUICKLIST编码方式，该编码方式是REDIS\_ENCODING\_LINKEDLIST和REDIS\_ENCODING\_ZIPLIST的结合，其原理是将一个长列表分成若干个以链表形式组织的ziplist，从而达到减少空间占用的同时提升REDIS\_ENCODING\_ZIPLIST编码的性能的效果。

### 集合类型

集合类型的内部编码方式可能是REDIS\_ENCODING\_HT或REDIS\_ENCODING\_INTSET。当集合中的所有元素都是整数且元素的个数小于配置文件中的set-max-intset-entries参数指定值（默认是512）时Redis会使用REDIS\_ENCODING\_INTSET编码存储该集合，否则会使用REDIS\_ENCODING\_HT来存储。

REDIS\_ENCODING\_INTSET编码存储结构体intset的定义是：

typedef struct intset {

uint32\_t encoding;

uint32\_t length;

int8\_t contents[];

}intset;

其中contents存储的就是集合中的元素值，根据encoding的不同，每个元素占用的字节大小不同。默认的encoding是INTSET\_ENC\_INT16（即2个字节），当新增加的整数元素无法使用2个字节表示时，Redis会将该集合的encoding升级为INTSET\_ENC\_INT32（即4个字节）并调整之前所有元素的位置和长度，同样集合的encoding还可升级为INTSET\_ENC\_INT64（即8个字节）。

REDIS\_ENCODING\_INTSET编码以有序的方式存储元素（所有使用SMEMBERS命令获得的结果是有序的），使得可以使用二分算法查找元素。然而无论是添加还是删除元素，Redis都需要调整后面元素的内存位置，所以当集合中的元素太多时性能较差。

当新增加的元素不是整数或集合中的元素数量超过了set-max-intset-entries参数指定值时，Redis会自动将该集合的存储结构转换成REDIS\_ENCODING\_HT。

注意：当集合的存储结构转换成REDIS\_ENCODING\_HT后，即使将集合中的所有非整数元素删除，Redis也不会自动将存储结构转换回REDIS\_ENCODING\_INTSET。因为如果要支持自动回转，就意味着Redis在每次删除元素时都需要遍历集合中的键来判断是否可以转换回原来的编码，这会使得删除元素变成了时间复杂度为O(n)的操作。

### 有序集合类型

有序集合类型的内部编码方式可能是REDIS\_ENCODING\_SKIPLIST或REDIS\_ENCODING\_ZIPLIST。同样在配置文件中可以定义使用REDIS\_ENCODING\_ZIPLIST方式编码的时机：

zset-max-ziplist-entries 128

zset-max-ziplist-value 64

具体规则和散列类型以及列表类型一样。

当编码方式是REDIS\_ENCODING\_SKIPLIST时，Redis使用散列表和跳跃列表两种数据结构来存储有序集合类型键值，其中散列表用来存储元素值与元素分数的映射关系以实现O(1)时间复杂度的ZSCORE等命令。跳跃列表用来存储元素的分数及其到元素值的映射以实现排序的功能。Redis对跳跃列表的实现进行了几点修改，其中包括允许跳跃列表中的元素（即分数）相同，还有为跳跃列表每个节点增加了指向前一个元素的指针以实现倒序查找。

采用此种编码方式时，元素值是使用redisObject存储的，所以可以使用字符串类型键值的优化方式优化元素值，而元素的分数是使用double类型存储的。

使用REDIS\_ENCODING\_ZIPLIST编码时有序集合存储的方式按照“元素1的值，元素1的分数，元素2的值，元素2的分数”的顺序排列，并且分数是有序的。

# Java 使用 Redis

## 安装

开始在 Java 中使用 Redis 前， 我们需要确保已经安装了 redis 服务及 Java redis 驱动，且你的机器上能正常使用 Java。

安装Java redis 驱动：

1. 首先你需要下载驱动包 下载 jedis.jar，确保下载最新驱动包。https://mvnrepository.com/artifact/redis.clients/jedis。因为jedis.jar依赖commons-pool2.jar，所以还需要下载此依赖包。
2. 将下载的jedis.jar放入项目中。

## 连接Redis服务器

Java连接Redis服务有两种方式，第一种是使用jedis对象连接，第二种是使用连接池连接。

### 单例模式连接

使用单例模式通过Jedis连接Redis需要实例化一个Jedis对象进行连接。

**import** redis.clients.jedis.Jedis;

**public** **class** RedisJava {

**private** **static** RedisJava *redisJava*;

**private** Jedis jedis;

**private** RedisJava(String host, **int** port) {

**if**(**null** == jedis) {

jedis = **new** Jedis(host, port);

}

}

//定义获取Jedis对象的方法

**public** Jedis getJedis() {

**return** jedis;

}

//定义线程安全的单例模式获取RedisJava对象实例。实例化该对象的时候实例化Jedis对象

**public** **static** RedisJava getRedisJava(String host, **int** port) {

**if**(**null** == *redisJava*) {

**synchronized** (RedisJava.**class**) {

**if**(**null** == *redisJava*) {

*redisJava* = **new** RedisJava(host, port);

}

}

}

**return** *redisJava*;

}

**public** **static** **void** main(String[] args) {

RedisJava redisJava = RedisJava.*getRedisJava*("192.168.0.110", 6379);

Jedis jedis = redisJava.getJedis();

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

System.***out***.println(jedis.get("name"));

}

}

### 连接池连接

创建连接jedis 连接池，jedis 提供JedisPool对象，使用比较方便。jedis pool 是基于apache common pool 实现。

**import** redis.clients.jedis.Jedis;

**import** redis.clients.jedis.JedisPool;

**import** redis.clients.jedis.JedisPoolConfig;

**public** **class** JedisPoolUtils {

**private** **static** JedisPool *jedisPool* = **null**;

**private** **static** String *host* = "192.168.0.110";

**private** **static** **int** *port* = 6379;

/\*\*

\* 建立连接池 真实环境，一般把配置参数缺抽取出来。

\*

\*/

**private** **static** **void** createJedisPool() {

// 建立连接池配置参数

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

// 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true

config.setBlockWhenExhausted(**true**);

// 设置的逐出策略类名, 默认DefaultEvictionPolicy(当连接超过最大空闲时间,或连接数超过最大空闲连接数)

config.setEvictionPolicyClassName("org.apache.commons.pool2.impl.DefaultEvictionPolicy");

// 是否启用pool的jmx管理功能, 默认true

config.setJmxEnabled(**true**);

// MBean ObjectName = new

// ObjectName("org.apache.commons.pool2:type=GenericObjectPool,name=" + "pool" +

// i); 默认为"pool", JMX不熟,具体不知道是干啥的...默认就好.

config.setJmxNamePrefix("pool");

// 是否启用后进先出, 默认true

config.setLifo(**true**);

// 最大空闲连接数, 默认8个

config.setMaxIdle(8);

// 最大连接数, 默认8个

config.setMaxTotal(8);

// 获取连接时的最大等待毫秒数(如果设置为阻塞时BlockWhenExhausted),如果超时就抛异常, 小于零:阻塞不确定的时间, 默认-1

config.setMaxWaitMillis(-1);

// 逐出连接的最小空闲时间 默认1800000毫秒(30分钟)

config.setMinEvictableIdleTimeMillis(1800000);

// 最小空闲连接数, 默认0

config.setMinIdle(0);

// 每次逐出检查时 逐出的最大数目 如果为负数就是 : 1/abs(n), 默认3

config.setNumTestsPerEvictionRun(3);

// 对象空闲多久后逐出, 当空闲时间>该值 且 空闲连接>最大空闲数 时直接逐出,不再根据MinEvictableIdleTimeMillis判断

// (默认逐出策略)

config.setSoftMinEvictableIdleTimeMillis(1800000);

// 在获取连接的时候检查有效性, 默认false

config.setTestOnBorrow(**false**);

// 在空闲时检查有效性, 默认false

config.setTestWhileIdle(**false**);

// 逐出扫描的时间间隔(毫秒) 如果为负数,则不运行逐出线程, 默认-1

config.setTimeBetweenEvictionRunsMillis(-1);

// 创建连接池

*jedisPool* = **new** JedisPool(config, *host*, *port*, 3000);

}

/\*\*

\* 在多线程环境同步初始化

\*/

**private** **static** **synchronized** **void** poolInit() {

**if** (*jedisPool* == **null**) {

*createJedisPool*();

}

}

/\*\*

\* 创建连接对象

\*

\* **@return**

\*/

**public** **static** Jedis getJedis() {

**if** (*jedisPool* == **null**) {

*poolInit*();

}

**return** *jedisPool*.getResource();

}

/\*\*

\* 关闭连接，归还到连接池

\*

\* **@param** jedis

\*/

**public** **static** **void** close(**final** Jedis jedis) {

**if** (jedis != **null**) {

jedis.close();

}

}

}

### Jedis连接Redis常见错误

1. SocketTimeoutException 连接超时



这种情况先排除是否开启了redis，或者ip，端口是否正确。如果都没问题，就是防火墙了。可能是redis的6379端口无法访问，

如果不能访问，需要在远程机器关掉防火墙或者添加允许通过

CentOS7，root用户登陆，输入如下命令开启端口：

1. 关闭防火墙的设置。

在CentOS 7中默认使用firewall做为防火墙，下面是启动&关闭防火墙的命令：

# 启动firewall

systemctl start firewalld.service

# 关闭firewall

systemctl stop firewalld.service

# 禁止firewall开机启动

systemctl disable firewalld.service

1. 开放端口限制

# 开放6379端口

firewall-cmd --zone=public --add-port=6379/tcp --permanent

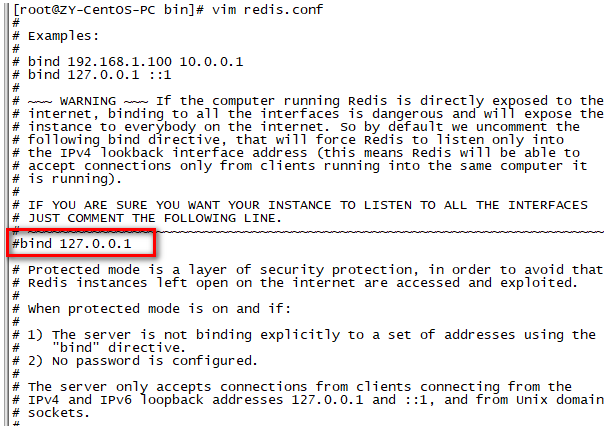
# 重启防火墙

firewall-cmd --reload

1. Connection refused 拒绝访问



修改redis.conf配置文件，使用vim命令打开，找到bind 127.0.0.1那行，将这行注释掉，重启redis-server。



1. DENIED Redis is running in protected mode （保护模式）

Exception in thread "main" redis.clients.jedis.exceptions.JedisDataException: DENIED Redis is running in protected mode because protected mode is enabled, no bind address was specified, no authentication password is requested to clients. In this mode connections are only accepted from the loopback interface. If you want to connect from external computers to Redis you may adopt one of the following solutions: 1) Just disable protected mode sending the command 'CONFIG SET protected-mode no' from the loopback interface by connecting to Redis from the same host the server is running, however MAKE SURE Redis is not publicly accessible from internet if you do so. Use CONFIG REWRITE to make this change permanent. 2) Alternatively you can just disable the protected mode by editing the Redis configuration file, and setting the protected mode option to 'no', and then restarting the server. 3) If you started the server manually just for testing, restart it with the '--protected-mode no' option. 4) Setup a bind address or an authentication password. NOTE: You only need to do one of the above things in order for the server to start accepting connections from the outside.

at redis.clients.jedis.Protocol.processError(Protocol.java:127)

at redis.clients.jedis.Protocol.process(Protocol.java:161)

at redis.clients.jedis.Protocol.read(Protocol.java:215)

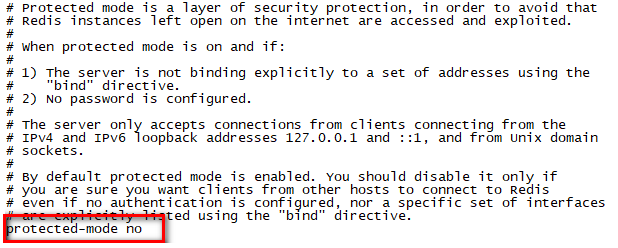
at redis.clients.jedis.Connection.readProtocolWithCheckingBroken(Connection.java:340)

at redis.clients.jedis.Connection.getStatusCodeReply(Connection.java:239)

at redis.clients.jedis.Jedis.set(Jedis.java:121)

at com.uc.RedisJava.main(RedisJava.java:34)

报错信息很长，但是主要是说redis开启了protected mode，这也是Redis3.2加入的新特性，开启保护模式的redis只允许本机登录，同样设置在配置文件redis.conf中，如图



这里原来是yes代表开启了保护模式，后面可以填密码也可以填no代表关闭，我们这里选择关闭保护模式，wq保存退出后再重启redis-server。

## Java操作Redis示例

### Jedis操作键

**public** **class** RedisJava {

**public** **static** **void** main(String[] args) {

//连接本地的 Redis 服务

Jedis jedis = **new** Jedis("192.168.0.110", 6379);

System.***out***.println("连接成功");

// 获取数据并输出

Set<String> keys = jedis.keys("\*");

Iterator<String> it=keys.iterator() ;

**while**(it.hasNext()){

String key = it.next();

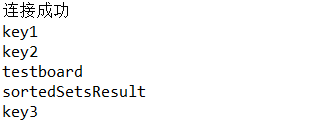
System.***out***.println(key);

}

}

}

运行结果如下：



### 字符串类型

**public** **class** RedisJava {

**public** **static** **void** main(String[] args) {

//连接Redis服务，获取Jedis对象

Jedis jedis = **new** Jedis("192.168.0.110", 6379);

System.***out***.println("Connection to server sucessfully");

//set the data in redis string

jedis.set("tutorial-name", "Redis tutorial");

// Get the stored data and print it

System.***out***.println("Stored string in redis:: "+ jedis.get("tutorial-name"));

}

}

运行结果如下：



### 列表类型

**public** **class** RedisJava {

**public** **static** **void** main(String[] args) {

//连接本地的 Redis 服务

Jedis jedis = **new** Jedis("192.168.0.110", 6379);

System.***out***.println("连接成功");

//存储数据到列表中

jedis.lpush("site-list", "Runoob");

jedis.lpush("site-list", "Google");

jedis.lpush("site-list", "Taobao");

// 获取存储的数据并输出

List<String> list = jedis.lrange("site-list", 0 ,2);

**for**(**int** i=0; i<list.size(); i++) {

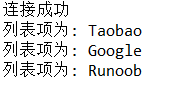
System.***out***.println("列表项为: "+list.get(i));

}

}

}

运行结果如下：



# Redis脚本

## 脚本介绍

Redis在2.6版推出了脚本的功能，允许开发者使用Lua语言编写脚本传到Redis中执行。在Lua脚本中可以调用大部分的Redis命令。

使用脚本的好处：

1. 减少网络开销：原先执行一组功能需要向Redis发送多条命令，现在使用脚本只需要发送一次，减少了网络的传输次数。
2. 原子操作：Redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。
3. 复用：客户端发送的脚本会永远存储在Redis中，这就意味着其他客户端可以复用这一脚本而不需要使用代码完成同样的逻辑。

## Lua语言

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

### Lua 特性

1. 轻量级: 它用标准C语言编写并以源代码形式开放，编译后仅仅一百余K，可以很方便的嵌入别的程序里。
2. 可扩展: Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。
3. 其它特性:

支持面向过程(procedure-oriented)编程和函数式编程(functional programming)；

自动内存管理；只提供了一种通用类型的表（table），用它可以实现数组，哈希表，集合，对象；

语言内置模式匹配；闭包(closure)；函数也可以看做一个值；提供多线程（协同进程，并非操作系统所支持的线程）支持；

通过闭包和table可以很方便地支持面向对象编程所需要的一些关键机制，比如数据抽象，虚函数，继承和重载等。

### Lua 应用场景

1. 游戏开发
2. 独立应用脚本
3. Web 应用脚本
4. 扩展和数据库插件如：MySQL Proxy 和 MySQL WorkBench
5. 安全系统，如入侵检测系统

### Lua语法

1. **数据类型**

Lua是一个动态类型语言，一个变量可以存储任何类型的值。

|  |  |
| --- | --- |
| 类型名 | 取值 |
| 空（nil） | 空类型只包含一个值，即nil，nil表示空，所以没有赋值的变量或表的字段都是nil |
| 布尔（boolean） | 布尔类型包含true和false两个值 |
| 数字（number） | 整数和浮点数都是使用数字类型存储，如1、0.2、3.5e20等 |
| 字符串（string） | 字符串类型可以存储字符串，且与Redis的键值一样都是二进制安全的。字符串可以使用单引号或双引号表示，两个符号是相同的。比如’a’，”b”都可以。字符串中可以包含转义字符，如\n、\r等。 |
| 表（table） | 表类型是Lua语言中唯一的数据结构，既可以当数组又可以当字典，十分灵活 |
| 函数（function） | 函数在Lua中是一等值，可以存储在变量中、作为函数的参数或返回结果。 |

1. **变量**

Lua的变量分为全局变量和局部变量。全局变量无需声明就可以直接使用，默认值是nil。

a = 1 -- 为全局变量a赋值

print(b) -- 无需声明即可使用，默认值是nil

a = nil --删除全局变量a的方法是将其赋值为nil，全局变量没有声明和未声明之分只有非nil和nil的区分。

在Redis脚本中不能使用全局变量，只允许使用局部变量以防止脚本之间相互影响。声明局部变量的方法为local变量名。

local c -- 声明一个局部变量c，默认值是nil

local d = 1 -- 声明一个局部变量d并赋值为1

local e, f -- 可以同时声明多个局部变量

同样声明一个存储函数的局部变量的方法为：

local say\_hi = function ()

print ’hi’

end

变量名必须是非数字开头，只能包含字母、数字和下划线，区分大小写。变量名不能与Lua的保留关键字相同。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| and | break | do | else | elseif | end |
| false | for | function | if | in | local |
| nil | not | or | repeat | return | then |
| true | until | while |  |  |  |

局部变量的作用域为从声明开始到所在层的语句块末尾。例如：

local x = 10

if true then

local x = x + 1

print(x)

do

local x = x + 1

print(x)

end

pring(x)

end

print(x)

打印结果为：

11

12

11

10

1. **注释**

Lua的注释有单行和多行两种。

单行注释以--开始，到行尾结束，多行注释以--[[开始，到]]结束。

--[[

这是一个多行注释

]]

1. **赋值**

Lua支持多重赋值，比如：

local a, b = 1, 2 -- a的值是1，b的值是2

local c, d = 1, 2, 3 -- c的值是1，d的值是2，3被舍弃了

local e, f = 1 -- e的值是1，f的值是nil

在执行多重赋值时，Lua会先计算所有表达式的值，比如：

local a = {1, 2, 3}

local i = 1

i, a[i] = i + 1, 5

Lua计算所有表达式的值后，上面最后一个赋值语句变为i, a[i] = 2, 5，所以赋值后i的值为2，a则为{5, 2, 3}。

1. **操作符**

Lua有以下5类操作符

1. **数学操作符**

数学操作符包括常见的+、-、\*、/、%（取模）、-（医院操作符，取负）和幂运算符号^。

数学操作符的操作数如果是字符串会自动转换成数字，比如：

print('1' + 1) -- 2

print('10' \* 2) -- 20

1. **比较操作符**

|  |  |
| --- | --- |
| 操作符 | 说明 |
| == | 比较两个操作数的类型和值是否都相等 |
| ~= | 与==的结果相反 |
| <、>、<=、>= | 小于、大于、小于等于、大于等于 |

比较操作符的结果一定是布尔类型。比较操作符不同于数学操作符，不会对两边的操作数进行自动类型转换。

print(1 == '1') -- false,二者类型不同，不会进行自动类型转换

print({'a'} == {'a'}) -- false,对于表类型值比较的是二者的引用

如果需要比较字串和数字，可以手动进行类型转换。

print(1 == tonumber('1'))

print('1' == tostring(1))

其中tonumber函数还可以进行进制转换

print(tonumber('F', 16)) -- 将字符串'F'从16进制转成10进制结果是15

1. **逻辑操作符**

|  |  |
| --- | --- |
| 操作符 | 说明 |
| not | 根据操作数的真和假相应地返回false和true |
| and | a and b 中如果a是真则返回b，否则返回a |
| or | a or b 中如果a是假则返回a，否则返回b |

只要操作数不是nil或false，逻辑操作符就认为操作数是真，否则是假。特别需要注意的是即使是0或空字符串也被当作真。

print(1 and 5) -- 5

print(1 or 5) -- 1

print(not 0) -- false

print('' or 1) -- ''

Lua的逻辑操作符支持短路，也就是说对于false and foo()，Lua不会调用foo函数，因为第一个操作数已经决定了无论foo函数返回的结果是什么，该表达式的值都是false。or操作符与之类似。

1. **连接操作符**

连接操作符只有一个：..，用来连接两个字符串。

print('hello' .. ' ' .. 'world!') -- 'hell world!'

连接操作符会自动把数字类型的值转换成字符串类型：

print('The price is' .. 25) -- 'The price is 25'

1. **取长度操作符**

取长度操作符是Lua5.1中新增加的操作符，同样只有一个，即#，用来获取字符串或表的长度。

print(#'hello') -- 5

运算符的优先级顺序

^

not # -(一元)

\* / %

+ -

..

< > <= >= ~= ==

and

or

1. **if语句**

Lua的if语句格式如下：

if 条件表达式 then

语句块

elseif 条件表达式 then

语句块

else

语句块

end

Lua与JavaScript一样每个语句都可以;结尾，但一般来说编写Lua时都会省略;。Lua也并不强制要求缩进，所有语句也可以写在一行中，例如：

a = 1

b = 2

if a then

b = 3

else

b = 4

end

可以写成：

a = 1 b = 2 if a then b = 3 else b = 4 end

甚至如下代码也是正确的：

a =

1 b = 2 if a

then b = 3 else b

= 4 end

但为了增强可读性，在编写的时候一定要注意缩进。

1. **循环语句**

Lua支持while、repeat和for循环语句。

while语句的格式为：

while 条件表达式 do

语句块

end

repeat语句的格式为：

repeat

语句块

until 表达式

for语句有两种形式，一种是数字形式：

for 变量 = 初值, 终值, 步长 do

语句块

end

其中步长可以省略，默认步长为1.比如使用for循环计算1~100的和：

local sum = 0

for i = 1, 100 do

sum = sum + i;

end

for语句的通用格式为：

for 变量1, 变量2, ..., 变量n in 迭代器 do

语句块

end

在编写Redis脚本时我们常用通用格式的for语句遍历表的值。

1. **表类型**

表示Lua中唯一的数据结构，可以理解为关联数组，任何类型的值（除了空类型）都可以作为表的索引。

表的定义格式为：

a = {} -- 将变量a赋值为一个空表

a['field'] = 'value' -- 将field字段赋值value

print(a.field) -- 打印内容为'value'

people = { -- 也可以这样定义

name = 'Bob',

age = 29

}

print(people.name) -- 打印的内容为'Bob'

当索引为整数的时候表和传统的数组一样。

a = {}

a[1] = 'Bob'

a[2] = 'Jeff'

可以写成下面这样：

a = {'Bob', 'Jeff'}

print(a[1]) -- 打印的内容为'Bob'

注意：Lua约定数组的索引是从1开始的，而不是0.

可以使用通用形式的for语句遍历数组。例如：

for index, value in ipairs(a) do

print(index) -- index迭代数组a的索引

print(value) -- value迭代数组a的值

end

打印的结果是：

1

Bob

2

Jeff

ipairs是Lua内置的函数，实现类似迭代器的功能。当然还可以使用数字形式的for语句遍历数组，例如：

for i = 1, #a do

print(i)

print(a[i])

end

输出的结果和上例相同。#a的作用是获取表a的长度。

Lua还提供了一个迭代器pairs，用来遍历非数组的表值，例如：

people = {

name = 'Bob',

age = 29

}

for index, value in pairs(people) do

print(index)

print(value)

end

打印结果为：

name

Bob

age

29

pairs与ipairs的区别在于前者会遍历所有值部位nil的索引，而后者只会从索引1开始递增遍历到最后一个值部位nil的整数索引。

1. **函数**

函数的定义为：

function (参数列表)

函数体

end

可以将其赋值给一个局部变量，比如：

local square = function (num)

return num \* num

end

如果没有参数，括号也不能省略。Lua还提供了一个语法来简化函数的定义，比如：

local function square(num)

return num \* num

end

这段代码会被转换为：

local square

square = function square(num)

return num \* num

end

因为在赋值前声明了局部变量square，所以可以在函数内部引用自身（实现递归）。

如果实参的个数小于形参的个数，则没有匹配到的形参的值为nil。相对应的，如果实参的个数大于形参的个数，则多出的实参会被忽略。如果希望捕获多 出的实参（即实现可变参数个数），可以让最后一个形参为…。比如，希望传入若干个参数计算这些数的平方：

local function square(...)

local argv = {...}

for i = 1, #argv do

argv[i] = argv[i] \* argv[i]

end

return unpack(argv)

end

a, b, c = square(1, 2, 3)

print(a)

print(b)

print(c)

输出结果为：

1

4

9

在第二个square函数中，我们首先将…转换为表argv，然后对表的每个元素计算其平方值。unpack函数用来返回表中的元素，在上例中argv表中有3个元素，所以return unpack(argv)相当于return argv[1], argc[2], argv[3]。

在Lua中return和break（用于跳出循环）语句必须是语句块中的最后一条语句，简单地说在这两条语句后面只能是end，else或until三者之一。如果希望在语句块的中间使用这两条语句的话可以人为地使用do和end将其包围。

### 标准库

Lua的标准库中提供了很多实用的函数，比如迭代器ipairs和pairs，类型转换函数tonumber和tostring，还有unpack函数都属于标准库中的“Base”库。

Redis支持大部分Lua标准库

|  |  |
| --- | --- |
| 库名 | 说明 |
| Base | 提供了一些基础函数 |
| String | 提供了用于字符串操作的函数 |
| Table | 提供了用于表操作的函数 |
| Math | 提供了数学计算函数 |
| Debug | 提供了用于调试的函数 |

1. **String库**

String库的函数可以通过字符串类型的变量以面向对象的形式访问。如string.len(string\_var)可以写成string\_var:len()。

1. 获取字符串长度

string.len(string)

string.len()的作用和操作符#类似。

print(string.len(’hello’)) -- 5

1. 转换大小写

string.lower(string)

string.upper(string)

例如：

print(string.lower('HELLO')) -- hello

print(string.upper('hello')) -- HELLO

1. 获取子字符串

string.sub(string, start [, end])

string.sub()可以获取一个字符串从索引start开始到end结束的子字符串，索引从1开始。索引可以是负数，-1代表最后一个元素。end参数如果省略则默认是-1（即截取到字符串末尾）

print(string.sub('hello', 1)) -- hello

print(string.sub('hello', 2)) -- ello

print(string.sub('hello', 2, -2)) -- ell

print(string.sub('hello', -2)) -- lo

1. **Table库**

Table库中的大部分函数都需要表的形式是数组形式。

1. 将数组转换为字符串

table.concat(table [, sep [, I [, j]]])

table.concat()与JavaScript中的join()类似，可以将一个数组转换成字符串，中间以sep参数指定的字符串分隔，默认为空。i和j用来限制要转换的表元素的索引范围，默认分别是1和表的长度，不支持负索引。

print(table.concat({1, 2, 3})) -- 123

print(table.concat({1, 2, 3}, ',', 2)) -- 2,3

print(table.concat({1, 2, 3}, ',', 2, 2)) -- 2

1. 向数组中插入元素

table.insert(table, [pos, ] value)

向指定索引位置pos插入元素value，并将后面的元素顺序后移。默认pos的值是数组长度加1，即在数组尾部插入。

a = {1, 2, 4}

table.insert(a, 3, 3)

table.insert(a, 5)

print(table.concat(a, ', ')) -- 1, 2, 3, 4, 5

1. 从数组中弹出一个元素

table.remove(table [, pos])

从指定的索引删除一个元素，并将后面的元素前移，返回删除的元素值。默认pos的值是数组的长度，即从数组尾部弹出一个元素。

table.remove(a)

table.remove(a, 1)

print(table.concat(a, ', ')) -- 2, 3, 4

1. **Math库**

Math库提供了常用的数学运算函数，如果参数是字符串会自动尝试转换成数字。

|  |  |
| --- | --- |
| 函数定义 | 说明 |
| math.abs(x) | 获得数字的绝对值 |
| math.sin(x) | 求三角函数sin值 |
| math.cos(x) | 求三角函数cos值 |
| math.tan(x) | 求三角函数tan值 |
| math.ceil(x) | 进一取整，如1.2取整后是2 |
| math.floor(x) | 向下取整，如1.8取整后是1 |
| math.max(x, …) | 获得参数中最大的值 |
| math.min(x, …) | 获得参数中最小的值 |
| math.pow(x, y) | 获得xy的值 |
| math.sqrt(x) | 获得x的平方根 |

除此之外，Math库还提供了随机数函数：

math.random([m, [, n]])

math.randomseed(x)

math.random()函数用来生成一个随机数，根据参数不同其返回值范围也不同：

没有提供参数：返回范围在[0, 1)的实数；

只提供了m参数：返回范围在[1, m]的整数；

同时提供了m和n参数：返回范围在[m, n]的整数。

math.random函数生产的随机数是依据种子计算得来的伪随机数，意味着使用同一种子生成的随机数序列是相同的。可以使用math.randomseed()函数设置种子的值，例如：

math.randomseed(1)

print(math.random(1, 100)) - 1

print(math.random(1, 100)) - 14

print(math.random(1, 100)) - 76

math.randomseed(1)

print(math.random(1, 100)) - 1

print(math.random(1, 100)) - 14

print(math.random(1, 100)) - 76

### 其他库

除了标准库以外，Redis还通过cjson库和cmsgpack库提供了对JSON和MessagePack的支持。Redis自动加载了这两个库，在脚本中可以分别通过cjson和cmsgpack两个全局变量来访问对应的库，两者的用法如下：

local people = {

name = 'Bob',

age = 29

}

-- 使用cjson序列化成字符串：

local json\_people\_str = cjson.encode(people)

-- 使用cmsgpack序列化成字符串：

local msgpack\_people\_str = cmsgpack.pack(people)

-- 使用cjson序列化后的字符串还原成表：

local json\_people\_obj = cjson.decode(people)

print(json\_people\_obj.name)

-- 使用cmsgpack序列化后的字符串还原成表：

local msgpack\_people\_obj = cmsgpack.unpack(people)

print(msgpack\_people\_obj.name)

## Redis与Lua

编写Redis脚本的目的就是用于读写操作Redis的数据，我们使用Lua脚本语言与Redis交互来操作Redis的数据。

### Lua调用Redis命令

在Lua脚本语言中可以使用redis.call函数调用Redis命令。

redis.call('set', 'foo', 'bar')

local value = redis.call('get', 'foo') -- value的值为bar

redis.call函数的返回值就是Redis命令的执行结果。redis.call函数会将这5中类型的回复转换成对应的Lua的数据类型，其具体对应规则如下：

|  |  |
| --- | --- |
| Redis返回值类型 | Lua数据类型 |
| 整数回复 | 数字类型 |
| 字符串回复 | 字符串类型 |
| 多行字符串回复 | 表类型（数组形式） |
| 状态回复 | 表类型（只有一个ok字段存储状态信息） |
| 错误回复 | 表类型（只有一个err字段存储错误信息） |

Redis还提供了redis.pcall函数，功能与redis.call相同，唯一的区别是当命令执行出错时redis.pcall会记录错误并继续执行，而redis.call会直接返回错误，不会继续执行。

### 从脚本中返回值

在很多情况下需要脚本可以返回值，在脚本中可以使用return语句将值返回给客户端，如果没有执行return语句则默认返回nil。因为我们可以像调用其他Redis内置命令一样调用我们自己写的脚本，所以同样Redis会自动将脚本返回值的Lua数据类型转换为Redis的返回值类型（其中Lua的false比较特殊，会被转换成空结果）。

|  |  |
| --- | --- |
| Lua数据类型 | Redis返回值类型 |
| 数字类型 | 整数回复（Lua的数字类型会被自动转换成整数） |
| 字符串类型 | 字符串回复 |
| 表类型（数组形式） | 多行字符串回复 |
| 表类型（只有一个ok字段存储状态信息） | 状态回复 |
| 表类型（只有一个err字段存储错误信息） | 错误回复 |

### 脚本相关命令

1. **EVAL命令**

编写完脚本后最重要的就是在程序中执行脚本。Redis提供了EVAL命令可以使开发者像调用其他Redis内置命令一样调用脚本。EVAL命令格式如下：

EVAL 脚本内容 key参数的数量 [key …] [arg …]

可以通过key和arg这两类参数向脚本传递数据，它们的值可以在脚本中分别使用KEYS和ARGV两个表类型的全局变量访问。

比如希望用脚本功能实现一个SET命令（实际开发不这样做），脚本内容是这样的：

return redis.call('SET', KEYS[1], ARGV[1])

现在打开redis-cli执行此脚本：

127.0.0.1:6379> EVAL "return redis.call('SET', KEYS[1], ARGV[1])" 1 foo bar

OK

127.0.0.1:6379> GET foo

"bar"

其中要读写的键名应该作为key参数，其他的数据都作为arg参数。

1. **EVALSHA命令**

Redis提供了EVALSHA命令允许开发者通过脚本内容的SHAI摘要来执行脚本，该命令的用法和EVAL一样，只不过是将脚本内容替换成脚本内容的SHAI摘要。

Redis在执行EVAL命令时会计算脚本的SHAI摘要并记录在脚本缓存中，执行EVALSHA命令时Redis会根据提供的摘要从脚本缓冲中查找对应的脚本内容，如果找到了则执行脚本，否则会返回错误：“NOSCRIPT NO matching script. Please use EVAL.”

在程序中使用EVALSHA命令的一般流程如下：

1. 先计算脚本的SHAI摘要，并使用EVALSHA命令执行脚本。
2. 获得返回值，如果返回“NOSCRIPT”错误则使用EVAL命令重新执行脚本

虽然这一流程略显麻烦，但值得庆幸的是很多编程语言的Redis客户端都会代替开发者完成这一流程。

## 深入脚本

### KEYS与ARGV

前面提到过向脚本传递的参数分为KEYS和ARGV两类，前者表示要操作的键名，后者表示非键名参数。但事实上这一要求并不是强制的，比如EVAL "return redis.call('get', KEYS[1])" 1 user:Bob可以获得user:Bob的键值，同样还可以使用EVAL "return redis.call('get', 'user:' .. ARGV[1])" 0 Bob完成同样的功能，此时我们虽然并未按照Redis的规则使用KEYS参数传递键名，但还是获得了正确的结果。

虽然规则不是强制的，但不遵守规则依然有一定的代价。Redis的集群功能将数据库中的键分散到不同的节点上。这意味着在脚本执行前就需要知道脚本会操作哪些键以便找到对应的节点，所以如果脚本中的键名没有使用KEYS参数传递则无法兼容集群。

有时候键名是根据脚本某部分的执行结果生成的，这时就无法在执行前将键名明确标出。这样脚本同样无法兼容集群功能，但却十分实用，避免了数据往返客户端和服务端的开销。

### 沙盒与随机数

Redis脚本禁止使用Lua标准库中与文件或系统调用相关的函数，在脚本中只允许对Redis的数据进行处理。并且Redis还通过禁用脚本的全局变量的方式保证每个脚本都是相对隔离的，不会互相干扰。

使用沙盒不仅是为了保证服务器的安全性，而且还确保了脚本的执行结果只和脚本本身和执行时传递的参数有关，不依赖外界条件。这是因为在执行复制和AOF持久化操作时记录的是脚本的内容而不是脚本调用的命令，所以必须保证在脚本内容和参数一样的前提下脚本的执行结果必须是一样的。

除了使用沙盒外，为了确保执行的结果可以重现，Redis还对随机数和会产生随机结果的命令进行了特殊的处理。

对于随机数而言，Redis替换了math.random和math.randomseed函数使得每次执行脚本时生成的随机数列都相同，如果希望获得不同的随机数序列，最简单的方法是由程序生成随机数并通过参数传递给脚本，或者采用更灵活的方法，即在程序中生成随机数传给脚本作为随机数种子，这样在脚本中再调用math.random产生的随机数就不同了。

对于会产生随机结果的命令如SMEMBERS（因为集合类型是无需的）或HKEYS（因为散列类型的字段也是无序的）等Redis会对结果按照字典顺序排序。内部是通过调用Lua标准库的table.sort函数实现的。

对于会产生随机结果但无法排序的命令（比如只会产生一个元素），Redis会在这类命令执行后将该脚本状态标记为lua\_random\_dirty，此后只允许调用只读命令，不允许修改数据库的值，否则会返回错误：“Write commands not allowed after non deterministic commands”属于此类的Redis命令有SPOP、SRANDMEMBER、RANDOMKEY和TIME。

### 其他脚本相关命令

除了EVAL和EVALSHA外，Redis还提供了其他4个脚本相关的命令，一般都会被客户端封装起来，开发者很少能使用到。

1. 将脚本加入缓存：SCRIPT LOAD

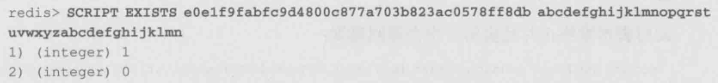
每次执行EVAL命令时Redis都会将脚本的SHAI摘要加入到脚本缓存中，以便下次客户端可以使用EVALSHA命令调用该脚本。如果只是希望将脚本加入脚本缓存而不执行则可以使用SCRIPT LOAD命令，返回值是脚本的SHAI摘要。

127.0.0.1:6379> SCRIPT LOAD "return 1"

"e0e1f9fabfc9d4800c877a703b823ac0578ff8db"

1. 判断脚本是否已经被缓存：SCRIPT EXISTS

SCRIPT EXISTS命令可以同时查找1个或多个脚本的SHAI摘要是否被缓存。



1. 清空脚本缓存：SCRIPT FLUSH

Redis将脚本的SHAI摘要加入到脚本缓存后会永久保留，不会删除，但可以手动使用SCRIPT FLUSH命令清空脚本缓存

127.0.0.1:6379> SCRIPT FLUSH

OK

1. 强制终止当前脚本的执行：SCRIPT KILL

如果想终止当前正在执行的脚本可以使用SCRIPT KILL命令。

### 原子性和执行时间

Redis的脚本执行时原子的，即脚本执行期间Redis不会执行其他命令。所有的命令都必须等待脚本执行完成后才能执行。为了防止某个脚本时间过长导致Redis无法提供服务，Redis提供了lua-time-limit参数限制脚本的最长运行时间，默认为5秒钟。当脚本运行时间超过这一限制后，Redis将开始接受其他命令但不会执行（以确保脚本的原子性，因为此时脚本并没有被终止），而是会返回“BUSY”错误。

此时Redis虽然可以接受任何命令，但实际会执行的只有两个命令：SCRIPT KILL和SHUTDOWN NOSAVE。

需要注意的是如果当前执行的脚本对Redis的数据进行了修改（如调用SET、LPUSH或DEL等命令）则SCRIPT KILL命令不会终止脚本的运行以防止脚本只执行了一部分。因为如果脚本只执行了一部分就被终止，则违背脚本了原子性要求，即脚本中的所有命令要么都执行，要么都不执行。

由于Redis脚本非常高效，所以在大部分情况下不用担心脚本的性能。但同时由于脚本的强大功能，很多原本在程序中执行的逻辑都可以放到脚本中执行，这时就需要开发者根据具体应用权衡到底哪些任务适合交给脚本。通常来讲不应该在脚本中进行大量耗时的计算，因为毕竟Redis是单进程单线程执行脚本，而程序却能够多进程或多线程运行。

# Redis持久化

Redis的强劲性能很大程序上是由于其将所有数据都存储在了内存中，然后当Redis重启后，所以存储在内存中的数据就会丢失，Redis同样也支持将数据存储在硬盘中。Redis支持两种方式的持久化，一种是RDB方式，另一种是AOF方式。前者会根据指定的规则“定时”将内存中的数据存储在硬盘上，而后者在每次执行命令后将命令本身记录下来。两种持久化方式可以单独使用其中一种，但更多情况下是将二者结合使用。

## RDB方式

RDB方式的持久化是通过快照完成的，当符合一定条件时Redis会自动将内存中的所有数据生成一份副本并存储在硬盘上，这个过程即为“快照”。Redis会在以下几种情况下对数据进行快照：

1. 根据配置规则进行自动快照
2. 用户执行SAVE或BGSAVE命令
3. 执行FLUSHALL命令
4. 执行复制时

### 根据配置规则进行自动快照

Redis允许用户自定义快照条件，当符合快照条件时，Redis会自动执行快照操作。进行快照的条件可以由用户在配置文件中自定义，由两个参数构成：时间窗口M和改动的键的个数N。每当时间M内被更改的键的个数大于N时，即符合自动快照条件。例如Redis安装目录中包含的样例配置文件中预置的3个条件：

save 900 1

save 300 10

save 60 10000

每条快照条件占一行，并且以save参数开头。同时可以存在多个条件，条件之间是“或”的关系。就这个例子而言，save 900 1的意思是在15分钟（900秒）内有一个或一个以上的键被更改则进行快照。同理，save 300 10表示在300秒内至少有10个键被修改则进行快照。

### 用户执行SAVE或BGSAVE命令

除了让Redis自动进行快照外，当进行服务重启、手动迁移以及备份时我们也会需要手动执行快照操作。Redis提供了两个命令来完成这一任务。

1. **SAVE命令**

当执行SAVE命令时，Redis同步地进行快照操作，在快照执行的过程中会阻塞所有来自客户端的请求。当数据库中的数据比较多时，这一过程会导致Redis较长时间不响应，所以要尽量避免在生产环境中使用这一命令。

1. **BGSAVE命令**

需要手动执行快照时推荐使用BGSAVE命令。BGSAVE命令可以在后台异步地进行快照操作，快照的同时服务器还可以继续响应来自客户端的请求。执行BGSAVE后Redis会立即返回OK表示开始执行快照操作，如果想知道快照是否完成，可以通过LASTSAVE命令获取最近一次成功执行快照的时间，返回结果是一个Unix时间戳。

127.0.0.1:6379> LASTSAVE

(integer) 1423537869

执行自动快照时Redis采用的策略即是异步快照。

### 执行FLUSHALL命令

当执行FLUSHALL命令时，Redis会清除数据库中的所有数据。需要注意的是，不论清空数据库的过程是否触发了自动快照条件，只要自动快照条件不为空，Redis就会执行一次快照操作。例如，当定义的快照条件为当1秒内修改10000个键时进行自动快照，而当数据库里只有一个键时，执行FLUSHALL命令也会触发快照，即使这一过程实际上只有一个键被修改了。

当没有定义自动快照条件时，执行FLUSHALL则不会进行快照。

### 执行复制时

当设置了主从模式时，Redis会在复制初始化时进行自动快照。当使用复制操作时，即使没有定义自动快照条件，并且没有手动执行过快照操作，也会生成RDB快照文件。

### 快照原理

Redis默认会将快照文件存储在Redis当前进程的工作目录中的dump.rdb文件中，可以通过配置dir和dbfilename两个参数分别指定快照文件的存储路径和文件名。快照的过程如下：

1. Redis使用fork函数复制一份当前进程（父进程）的副本（子进程）。
2. 父进程继续接收并处理客户端发来的命令，而子进程开始将内存中的数据写入硬盘中的临时文件。
3. 当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此一次快照操作完成。

通过上述过程可以发现Redis在进行快照的过程中不会修改RDB文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候RDB文件都是完整的。这使得我们可以通过定时备份RDB文件来实现Redis数据库备份。RDB文件是经过压缩的二进制格式，所以占用的空间会小于内存中的数据大小，更加利于传输。

Redis启动后会读取RDB快照文件，将数据从硬盘载入到内存。根据数据量大小与结构和服务器性能不同，这个时间也不同。通常将一个记录1000万个字符串类型键、大小为1GB的快照文件载入到内存中需要花费20~30秒。

通过RDB方式实现持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据。这就需要开发者根据具体的应用场合，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受的范围。如果数据相对重要，希望将损失降到最小，则可以使用AOF方式进行持久化。

## AOF方式

当使用Redis存储非临时数据时，一般需要打开AOF持久化来降低进程终止导致的数据丢失。AOF可以将Redis执行的每一条写命令追加到硬盘文件中，这一过程显然会降低Redis的性能，但是大部分情况下这个影响是可以接受的，另外使用较快的硬盘可以提高AOF的性能。

### 开启AOF

默认情况下Redis没有开启AOF方式的持久化，可以通过appendonly参数启动：

appendonly yes

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof，可以通过appendfilename参数修改：

appendfilename appendonly.aof

### AOF的实现

AOF文件以纯文本的形式记录了Redis执行的写命令，例如在开启AOF持久化的情况下执行了如下4个命令：

SET foo 1

SET foo 2

SET foo 3

SET foo

Redis会将前3条命令写入AOF文件中，此时AOF文件中的内容如下：

\*2

$6

SELECT

$1

0

\*3

$3

set

$3

foo

$1

1

\*3

$3

set

$3

foo

$1

2

\*3

$3

set

$3

foo

$1

3

可见AOF文件的内容正是Redis客户端向Redis发送的原始通信协议的内容，从中可见Redis确实只记录了前3条命令。然而这时有一个问题是前2条命令其实都是冗余的，因为这两条的执行结果会被第三条命令覆盖。随着执行的命令越来越多，AOF文件的大小也会越来越大，即使内存中实际的数据可能并没有多少。很自然地，我们希望Redis可以自动优化AOF文件。实际上每当达到一定条件时Redis就会自动重写AOF文件，这个条件可以在配置文件中设置：

auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb

auto-aof-rewrite-percentage参数的意义是当目前的AOF文件大小超过上一次重写时的AOF文件大小的百分之多少时会再次进行重写，如果之前没有重写过，则以启动时的AOF文件大小为依据。auto-aof-rewrite-min-size参数限制了允许重写的最小AOF文件大小，通常在AOF文件很小的情况下即使其中有很多冗余的命令我们也并不太关心。除了让Redis自动执行重写外，我们还可以主动使用BGREWRITEAOF命令手动执行AOF重写。

上例中的AOF文件重写后的内容为：

\*2

$6

SELECT

$1

0

\*3

$3

set

$3

foo

$1

3

可见冗余的命令已经被删除了。重写的过程只和内存中的数据有关，和之前的AOF文件无关，这与RDB很相似，只不过二者的文件格式完全不同。

在启动时Redis会逐个执行AOF文件中的命令来将硬盘中的数据载入到内存中，载入的速度相较RDB会慢一些。

### 同步硬盘数据

虽然每次执行更改数据库内容的操作时，AOF都会将命令记录在AOF文件中，但是事实上，由于操作系统的缓存机制，数据并没有真正地写入硬盘，而是进入了系统的硬盘缓存。在默认情况下系统每30秒会执行一次同步操作，以便将硬盘缓存中的内容真正地写入硬盘，在这30秒的过程中如果系统异常退出则会导致硬盘缓存中的数据丢失。一般来讲启动AOF持久化的应用都无法容忍这样的损失，这就需要Redis在写入AOF文件后主动要求系统将缓存内容同步到硬盘中。在Redis中我们可以通过appendfsync参数设置同步的时机：

# appendfsync always

appendfsync everysec

# appendfsync no

默认情况下Redis采用everysec规则，即每秒执行一次同步操作。always表示每次执行写入都会执行同步，这时最安全也是最慢的方式。no表示不主动进行同步操作，而是完全交由操作系统来做（即每30秒一次），这是最快也是最不安全的方式。一般情况下使用默认值everysec就足够了，既兼顾了性能又保证了安全。

Redis允许同时开启AOF和RDB，既保证了数据安全又使得进行备份等操作十分容易。此时重新启动Redis后Redis会使用AOF文件来恢复数据，因为AOF方式的持久化可能丢失的数据更少。

# Redis集群

在实现项目中，单个Redis服务器有以下弊端：从结构上，单个Redis服务器会发生单点故障，同时一台服务器需要承受所有的请求负载。这就需要为数据库生成多个副本并分配在不同的服务器上；从容量上，单个Redis服务器的内存非常容易成为存储瓶颈，所以需要进行数据分片。同时拥有多个Redis服务器后就会面临如何管理集群的问题，包括如何增加节点、故障恢复等操作。

## 复制

通过持久化功能，Redis保证了即使服务器重启的情况下也不会损失数据。但是由于数据时存储在一台服务器上的，如果这台服务器出现硬盘故障等问题，也会导致数据丢失。为了避免单点故障，通常的做法是将数据库复制多个副本以部署在不同的服务器上，这样即使有一台服务器出现故障，其他服务器依然可以继续提供服务。为此Redis提供了复制功能，可以实现当一台数据库中的数据更新后，自动将更新的数据同步到其他数据库上。

### 配置

在复制的概念中，数据库分为两类，一类是主数据库，另一类是从数据库。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

主数据库A

从数据库B

从数据库C

在Redis中使用复制功能非常容易，只需要在从数据库的配置文件中如下配置，主数据库无需进行任何配置。

slaveof 主数据库地址 主数据库端口

默认情况下，从数据时只读的，如果直接修改从数据库的数据会出现错误，可以通过设置从数据库的配置文件中的slave-read-only为no以使从数据库可写，但是因为对从数据库的任何更改都不会同步给任何其他数据库，并且一旦主数据库中更新了对应的数据就会覆盖从数据库中的改动，所以通常的场景下不应该设置从数据库可写，以面导致易被忽略的潜在应用逻辑错误。

除了通过配置文件或命令行参数设置slaveof参数，还可以在运行时使用SLAVEOF命令修改：

127.0.0.1:6379> SLAVEOF 127.0.0.1 6379

如果该数据库已经是其他主数据库的从数据库了，SLAVEOF命令会停止和原来数据库的同步而和新数据库同步。此外对于从数据库来说，还可以使用SLAVEOF NO ONE命令来使当前数据库停止接收其他数据库的同步并转换成为主数据库。

### 原理

当一个从数据库启动后，会向主数据库发送SYNC命令。同时主数据库接收到SYNC命令后会开始在后台保存快照（即RDB持久化的过程），并将保存快照期间接收到的命令缓存起来。当快照完成后，Redis会将快照文件和所有缓存的命令发送给从数据库。从数据库接收到后，会载入快照文件并执行收到的缓存的命令，以上过程称为复制初始化。复制初始化结束后，主数据库每当收到写命令时就会将命令同步给从数据库，从而保证主从数据库数据一致。

复制初始化阶段结束后，主数据库执行的任何会导致数据变化的命令都会异步地传送给从数据库，这一过程为复制同步阶段。复制同步阶段会贯穿整个主从同步过程的始终，直到主从关系终止为止。在复制的过程中，快照无论在主数据库还是从数据库中都起了很大的作用，只要执行复制就会进行快照，即使我们关闭了RDB方式的持久化。

从数据库不仅可以接收主数据库的同步数据，自己也可以同时作为主数据库存在，形成类似图的结构。例如下图中，数据库A的数据会同步到B和C中，而B中的数据会同步到D和E中。向B中写入数据不会同步到A或C中，只会同步到D和E中。

主数据库A

从数据库E

从数据库D

从数据库C

从数据库B

### 读写分类与一致性

通过复制可以实现读写分离，以提高服务器的负载能力。在常见的场景下（如电子商务网站），读的频率大于写，当单机的Redis无法应付大量的读请求时，可以通过复制功能建立多个从数据库节点，主数据库只进行写操作，而从数据库负责读操作。这种一主多从的结构很适合读多写少的场景，当单个的主数据库不能够满足需求时，就需要使用Redis3.0退出的集群功能。

### 从数据库持久化

另一个相对耗时的操作是持久化，为了提高性能，可以通过复制功能建立一个（或若干个）从数据库，并在从数据库中启用持久化，同时在主数据库禁用持久化。当从数据库崩溃重启后主数据库会自动将数据同步过来，所以无需担心数据丢失。

然而当主数据库崩溃时，情况就稍显复杂。手工通过从数据数据恢复主数据库数据时，需要严格按照以下两步进行：

1. 在从数据库中使用SLAVEOF NO ONE命令将从数据库提升成主数据库继续服务。
2. 启动之前崩溃的主数据库，然后使用SLAVEOF命令将其设置成新的主数据的从数据库，即可将数据同步回来。

注意：当开启复制且主数据库关闭持久化功能时，一定不要使用Supervisor以及类似的进程管理工具令主数据库崩溃后自动重启，同样当主数据库所在的服务器因故关闭时，也要避免直接重写启动，这是因为当主数据库重新启动后，因为没有开启持久化功能，所以数据库中所有数据都被清空，这时从数据库依然会从主数据库中接收数据，使得所有从数据库也被清空，导致从数据库的持久化失去意义。

无论哪种情况，手工维护从数据库或主数据库的重启以及数据恢复都相对麻烦，好在Redis提供了一种自动化方案哨兵来实现这一过程，避免了手工维护的麻烦和容易出错的问题。

### 无硬盘复制

前面介绍Redis复制的工作原理时介绍了复制时基于RDB方式的持久化实现的，即主数据库端在后台保存RDB快照，从数据库端则接收并载入快照文件。这样的实现优点是可以显著地简化逻辑，复用已有的代码，但是缺点也是很明显的。

1. 当主数据库禁用RDB快照时，如果执行了复制初始化操作，Redis依然会生成RDB快照，所以下次启动后主数据库会以该快照恢复数据。因为复制发生的时间不能确定，这使得恢复的数据可能是任何时间点的。
2. 因为复制初始化时需要在硬盘中创建RDB快照文件，所以如果硬盘性能很慢时这一过程会对性能产生影响。

因此从2.8.18版本开始，Redis引入了无硬盘复制选项，开启该选项时，Redis在与从数据库进行复制初始化时将不会将快照内容存储到硬盘上，而是直接通过网络发送给从数据库，避免了硬盘的性能的瓶颈。

可以在配置文件中使用如下配置来开启该功能：

repl-diskless-sunc yes

### 增量复制

当主从数据库连接断开后，从数据库会发送SYNC命令来重新进行一次完整复制操作。这样即使断开期间数据库的变化很下（甚至没有），也需要将数据库中的所有数据重新快照并传送一次。在正常的网络应用环境中，这种实现方式显然不太理想。在Redis2.8版本就是实现了主从断线重连的情况下的增量复制。

增量复制是基于如下3点实现的：

1. 从数据库会存储主数据库的运行ID（run id）。每个Redis运行实例均会拥有一个唯一的运行ID，每当实例重启后，就会自动生成一个新的运行ID
2. 在复制同步阶段，主数据库没将一个命令传送给从数据库时，都会同时把该命令存放到一个积压队列中，并记录下当前积压队列中存放的命令的偏移量范围。
3. 同时，从数据库接收到主数据库传来的命令时，会记录下该命令的偏移量。

这3点事实现增量复制的基础。主从通信流程可以看到，当主从连接准备就绪后，从数据库会发送一条格式为“PSYNC 数数据库的运行ID 断开前最新的命令偏移量”的命令来告诉主数据库可以开始把数据不同过来。主数据库收到PSYNC命令后，会执行以下判断来决定此次连接是否可以执行增量复制。

1. 首先主数据库会判断从数据库传送来的运行ID是否和自己的运行ID相同。这一步骤的意义在于确保从数据库之间确实是和自己同步的。以免从数据库拿到错误的数据（比如主数据库在断线期间重启过，会造成数据的不一致）。
2. 然后判断从数据库最后同步成功的命令偏移量是否在积压队列中，如果在则可以执行增量复制，并将积压队列中相应的命令发送给从数据库。如果此次重连不满足增量复制的条件，主数据库会进行一此全部同步。

积压队列在本质上是一个固定长度的循环队列，默认情况下积压队列的大小为1MB。可以通过配置文件的repl-backlog-size选项来调整。积压队列越大，其允许的主从数据库断线的时间就越长。根据主从数据库之间的网络状况，设置一个合理的积压队列很重要。因为积压队列存储的内容是命令本身，如SET foo bar，所以估算积压队列的大小只需要估计主从数据库断线的时间中主数据库可能执行的命令的大小即可。

与积压队列相关的另一个配置选项是repl-backlog-ttl，即当所有从数据库与主数据库断开连接后，经过多久时间可以释放积压队列的内容空间。默认时间是1小时。

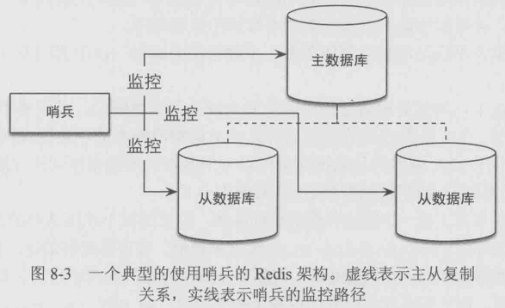
## 哨兵

### 什么是哨兵

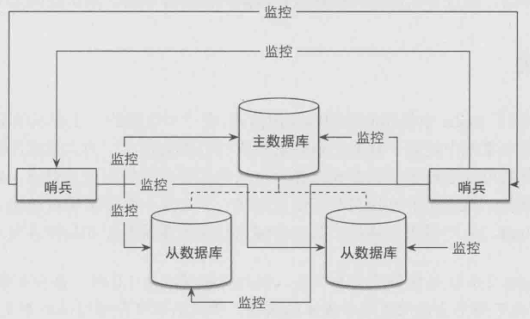
哨兵的作用就是监控Redis系统的运行情况，它的功能包括以下两个：

1. 监控主数据库和从数据库是否正常运行
2. 主数据库出现故障时自动将从数据库转换为主数据库。

哨兵是一个独立的进程，使用哨兵的一个典型架构如图



在一个一主多从的Redis系统中，可以使用多个哨兵进行监控任务以保证系统足够稳健。如下图所示，此时不仅哨兵会同时监控主数据库和从数据库，哨兵之间也会互相监控。



### 哨兵的实现

首先我们先建立3个Redis实例，其中包括一个主数据库和两个从数据库。主数据库的端口为6379，两个从数据库的端口分别是6380和6381.我们使用Redis命令客户端来获取复制状态，以保证复制配置正确。

127.0.0.1:6379> INFO replication

# Replication

role:master

connected\_slaves:2

slave0:ip=127.0.0.1,port=6380,stae=online,offset=10125,lag=0

slave0:ip=127.0.0.1,port=6381,stae=online,offset=10125,lag=1

可见其连接了两个从数据库，配置正确，然后用同样的方式查看两个从数据库的配置:

127.0.0.1:6379> INFO replication

# Replication

role:slave

master\_host:127.0.0.1

master\_port:6379

127.0.0.1:6379> INFO replication

# Replication

role:slave

master\_host:127.0.0.1

master\_port:6379

当出现的信息如上时，即证明一主二从的复制配置已经完成了。

接下来开始配置哨兵。新建一个配置文件，如sentinel.conf，其配置内容为：

sentinel monitor mymaster 127.0.0.1 6379 1

其中mymaster表示要监控的主数据库的名称，可以自定义一个。这个名字必须仅由大小写字母、数字和“.-\_”3个字符组成。后面两个参数表示主数据库的地址和端口号，这里我们要监控的是主数据库6379。最后的1表示最低通过票数，后面会介绍。

接下来执行来启动Sentinel进程，并将上述配置文件的路传递给哨兵：

[root@localhost ~]# redis-sentinel /root/redis/sentinel.conf

需要注意的是，配置哨兵监控一个系统时，只需要配置其监控主数据库即可，哨兵会自动发现所有复制该主数据库的从数据库。

启动哨兵后，哨兵输出如下内容：

[71835] 19 Feb 22:32:28.730 # Sentinel runid is e3290844cla404699479771846b716c7fc830e80

[71835] 19 Feb 22:32:28.730 # +monitor master mymaster 127.0.0.1 6379 quorum 1

[71835] 19 Feb 22:32:09.997 \* +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379

[71835] 19 Feb 22:32:30.068 \* +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379

其中+slave表示新发现了从数据库，可见哨兵成功地发现了两个从数据库。现在哨兵已经在监控这3个Redis实例了，这时我们将主数据库关闭，等待指定时间后（可配置，默认为30秒），哨兵会输出如下内容：

[71835] 19 Feb 22:32:03.780 # +sdown master mymaster 127.0.0.1 6379

[71835] 19 Feb 22:32:03.780 # +odown master mymaster 127.0.0.1 6379 #quorum 1/1

其中+sdown表示哨兵主观认为主数据库停止服务了，而+odown则表示哨兵客观认为主数据库停止服务了。此时哨兵开始执行故障恢复，即挑选一个从数据库，将其升格为主数据库，同时输出如下内容：

[71835] 19 Feb 22:32:03.780 # +try-failover master mymaster 127.0.0.1 6379

…….

[71835] 19 Feb 22:32:05.913 # +failover-end master mymaster 127.0.0.1 6379

[71835] 19 Feb 22:32:05.913 # +switch-master mymaster 127.0.0.1 6379 127.0.0.1 6380

[71835] 19 Feb 22:32:05.914 \* +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6380

[71835] 19 Feb 22:32:05.914 \* +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380

+try-failover表示哨兵开始进程故障恢复，+failover-end表示哨兵完成故障恢复，期间设计的内容比较复杂，包括领头哨兵的选举、备选从数据库的选择等，此处只需要关注最后3条输出。+switch-master表示主数据库从6379端口前移到6380端口，即6380端口的从数据库被升格为主数据库，同时两个+slave则列出新的主数据库的两个从数据库，端口分别是6381和6379。其中6379就是之前停止服务的主数据库，可见哨兵并没有彻底清除停止服务的实例的信息，这时因为停止服务的实例有可能会在之后的某个时间恢复服务，这时哨兵会让其从新加入进来，所以当实例停止服务后，哨兵会更新该实例的信息，使得当其重新加入可以按照当前信息继续对外提供服务。

故障恢复完成后，可以使用Redis命令行客户端重新检查6380和6381两个端口上的实例的复制信息：

127.0.0.1:6379> INFO replication

# Replication

role:master

connected\_slaves:1

slave0:ip=127.0.0.1,port=6381,stae=online,offset=10125,lag=1

127.0.0.1:6379> INFO replication

# Replication

role:slave

master\_host:127.0.0.1

master\_port:6380

可以看到6380端口上的实例已经确实升格为主数据库了，同时6381端口上的实例是其从数据库。整个故障恢复过程就此完成。

此时我们将6379端口上的实例重新启动，会发生情况呢？首先哨兵会监控到这一变化，并输出：

[71835] 19 Feb 23:46:14.573 # -sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380

[71835] 19 Feb 23:46:24.504 \* +convert-to-slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380

-sdown表示实例6379已经恢复服务了（以+sdown相反）同时+convert-to-slave表示将6379端口的实例设置为6380端口实例的从数据库。这时使用Redis命令行客户端查看6379端口实例的复制信息为：

127.0.0.1:6379> INFO replication

# Replication

role:slave

master\_host:127.0.0.1

master\_port:6380

同时6380端口实例的复制信息为：

127.0.0.1:6379> INFO replication

# Replication

role:master

connected\_slaves:2

slave0:ip=127.0.0.1,port=6381,stae=online,offset=292948,lag=1

slave0:ip=127.0.0.1,port=6379,stae=online,offset=292948,lag=1

正如预期一样，6380端口实例的从数据库变为了两个，6379成功恢复服务。

### 实现原理

一个哨兵进程启动时会读取配置文件的内容，通过如下的配置找出需要监控的主数据库：

sentinel monitor master-name ip redis-port quorum

其中master-name是一个由大小写字母、数字和“.-\_”组成的主数据库的名字，因为考虑到故障恢复后当前监控的系统的主数据库的地址和端口会产生变化，所以哨兵提供了命令可以通过主数据库的名字获取当前系统的主数据库的地址和端口号。

ip表示当前系统中主数据库的地址，而redis-port则表示端口号。

quorum用来表示执行故障恢复操作前至少需要几个哨兵节点同一。

一个哨兵节点可以同时监控多个Redis主从系统，只需要提供多个sentinel monitor配置即可，例如：

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel monitor othermaster 192.168.1.3 6380 4

同时多个哨兵节点也可以同时监控同一个Redis主从系统，从而形成网状结构。

配置文件中还可以定义其他监控相关的参数，每个配置选项都包含主数据库的名字使得监控不同主数据库时可以使用不同的配置参数。例如：

sentinel down-after-milliseconds mymaster 60000

sentinel down-after-milliseconds othermaster 10000

上面的两行配置分别配置了mymaster和othermaster的down-after-milliseconds选项分别为60000和10000。

哨兵启动后，会与要监控的主数据库建立两条连接，这两条连接的建立方式与普通的Redis客户端无异。其中一条连接用来订阅该主数据库的\_sentinel\_:hello频道以获取其他同样监控该数据库的哨兵节点的信息，另外哨兵也需要定期向主数据库发送INFO等命令来获取主数据库本身的信息。应为客户端的连接进入订阅模式时就不能再执行其他命令了，所以这时哨兵会使用另外一条连接来发送这些命令。

和主数据库的连接建立完成后，哨兵会定时执行下面3个操作：

1. 每10秒哨兵会向主数据库和从数据库发送INFO命令。
2. 每2秒哨兵会向主数据库和从数据库的\_sentinel\_:hello频道发送自己的信息。
3. 每1秒哨兵会向主数据库、从数据库和其他哨兵节点发送PING命令。

这3个操作贯穿哨兵进程的整个生命周期，非常重要，可以说了解了这3个操作的意义就能够了解哨兵工作原理的一般内容了。

首先，发送INFO命令使得哨兵可以获得当前数据库的相关信息（包括允许ID、复制信息等）从而实现新节点的自动发现。前面说配置哨兵监控Redis主从系统时只需要指定主数据库的信息即可，因为哨兵正是借助INFO命令来获取所有复制该主数据库的从数据库信息的。启动后，哨兵向主数据库发送INFO命令，通过解析返回结果来得知从数据库列表，而后对每个从数据库同样建立两个连接，两个连接的作用和前面介绍的与主数据库建立的两个连接完全一致。在此之后，哨兵会每10秒定时向已知的所有主从数据库发送INFO命令来获取信息更新并进行相应操作，比如对新增的从数据库建立连接并加入监控列表，对主从数据库的角色变化进行信息更新等。

接下来哨兵向主从数据库的\_sentinel\_:hello频道发送信息来与同样监控该数据库的哨兵分享自己的信息。发送的消息内容为：

<哨兵的地址>,<哨兵的端口>,<哨兵的运行ID>,<哨兵的配置版本>,<主数据库的名字>,<主数据库的地址>,<主数据库的端口>,<主数据库的配置版本>

可以看到消息包括的哨兵的基本信息，以及其监控的主数据库的信息。前文介绍过，哨兵会订阅每个其监控的数据库的\_sentinel\_:hello频道，所以当其他哨兵收到消息后，会判断发消息的哨兵是不是新发现的哨兵。如果是则将其加入已发现的哨兵列表中并创建一个到其的连接（与数据库不同，哨兵与哨兵之间只会创建 一条连接用来发送PING命令，而不需要创建另一条连接来订阅频道，因为哨兵只需要订阅数据库的频道即可实现自动发现其他哨兵）。同时哨兵会判断信息中主数据库的配置版本，如果该版本比当前记录的主数据库的版本高，则更新主数据库的数据。

实现了自动发现从数据库和其他哨兵节点后，哨兵要做的就是定时监控这些数据库和节点有没有停止服务。这时通过每隔一定时间向这些节点发送PING命令实现的。时间间隔与down-after-milliseconds选项有关，当down-after-milliseconds的值小于1秒时，哨兵会每隔down-after-milliseconds指定的时间发送一次PING命令，当down-after-milliseconds的值大于1秒时，哨兵会每隔1秒发送一次PING命令。

//每隔1秒发送一个PING命令

sentinel down-after-milliseconds mymaster 60000

//每隔600毫秒发送一个PING命令

sentinel down-after-milliseconds othermaster 600

当超过down-after-milliseconds选项指定时间后，如果被PING的数据库或节点仍然未进行回复，则哨兵认为其主观下线。主观下线表示从当前的哨兵进程看来，该节点已经下线。如果该节点是主数据库，则哨兵会进一步判断是否需要对其进行故障恢复：哨兵发送SENTINEL is-master-down-by-addr命令询问其他哨兵节点以了解他们是否也认为该主数据库主观下线，如果达到指定数量时，哨兵会认为其客观下线，并选举领头的哨兵节点对主从系统发起故障恢复。这个指定数量即为前文介绍的quorum参数。

sentinel monitor mymaster 127.0.0.1 6379 2

该配置表示只有当至少两个Sentinel节点（包括当前节点）认为该主数据库主观下线时，当前哨兵节点才会认为该主数据库客观下线。进行接下来的选举领头哨兵步骤。

虽然当前哨兵节点发现了主数据库客观下线，需要故障祸福，但是故障恢复需要由领头的哨兵来完成，这样可以保证同一时间是一个哨兵节点来执行故障恢复。选举领头哨兵的过程使用了Redis算法，具体过程如下：

1. 发现主数据库客观下线的哨兵节点（A）向每个哨兵节点发送命令，要求对方选自己成为领头哨兵。
2. 如果目标哨兵节点没有选过其他人，则会同意将A设置成领头哨兵。
3. 如果A发现在有超过半数且超过quorum参数值的哨兵节点同意选自己成为领头哨兵，则A成功成为领头哨兵。
4. 当有多个哨兵节点同时参选领头哨兵，则会出现没有任何节点当选的可能。此时每个参选节点将等待 一个随机事件重新发起参选请求，进行下一轮选举，直到选举成功。

因为要成为领头哨兵必须有超过半数的哨兵节点支持，所以每次选举最多只会选出一个领头哨兵。

选出领头哨兵后，领头哨兵会开始对主数据库进行故障恢复。故障恢复的过程相对简单，具体如下：

1. 所有在线的从数据库中，选择优先级最高的从数据库。优先级可以通过slave-priority选项来设置。
2. 如果有多个最高优先级的从数据库，则复制的命令偏移量越大（即复制越完整）越优先。
3. 如果以上条件都一样，则选择运行ID较小的从数据库。

选出一个从数据库后，领头哨兵将向从数据库发送SLAVEOF NO ONE命令使其升格为主数据库。而后领头哨兵向其他从数据库发送SLAVEOF命令来使其成为新主数据库的从数据库。最后一步则是更新内部的记录，将已经停止服务的旧的主数据库更新为新的主数据库的从数据库，使得当其恢复服务时自动以从数据库的身份继续服务。

### 哨兵的部署

哨兵以独立进程的方式对一个主从系统进行监控，监控的效果好坏与否决定了哨兵的视角是否有代表性。如果一个主从系统中配置的哨兵较少，哨兵对整个系统的判断的可靠性就会降低。相对稳妥的哨兵部署方案是使得哨兵的视角尽可能地与每个节点的视角一致，即：

1. 为每个节点部署一个哨兵
2. 使每个哨兵与其对应的节点的网络环境相同或相近。

这样的部署方案可以保证哨兵的视角拥有较高的代表性和可靠性。

同时设置quorum的值为N / 2 + 1（其中N为哨兵节点数量），这样使得只有当大部分哨兵节点同意后才会进行故障恢复。

当系统中的节点较多时，考虑到每个哨兵都会和系统中的所有节点建立连接，为每个节点分配一个哨兵会产生较多连接，尤其是当进行客户端分片时使用多个哨兵节点监控多个主数据库会因为Redis不支持连接复用而产生大量冗余连接；同时如果Redis节点负载较高，会在一定程度上影响其对哨兵的回复以及与其同机的哨兵与其他节点的通信。所以配置哨兵时需要根据实际的生产环境情况进行选择。

## 集群

即使使用哨兵，此时的Redis集群的每个数据库依然存有集群中的所有数据，从而导致集群的总数据存储受限于可用存储内存最小的数据库节点，形成木桶效应。由于Redis中的所有数据都是基于内存存储，这一问题就尤为突出了，尤其是当使用Redis做持久化存储服务使用时。

对Redis进行水平扩容，在旧版Redis中通常使用客户端分片来解决这个问题，即启动多个Redis数据库节点，由客户端决定每个键交由那个数据库节点存储，下次客户端读取该键时直接到该节点读取。这样可以实现将整个数据分布存储在N个数据库节点中，每个节点只存放总数据量的1/N。但对于需要扩容的场景来说，在客户端分片后，如果想增加更多的节点，就需要对数据进行手工迁移，同时在迁移的过程中为了保证数据的一致性，还需要将集群暂时下线，相对比较复杂。

考虑到Redis实例非常轻量的特点，可以采用预分片技术来在一定程度上避免此问题，具体来说是在节点部署初期，就提前考虑日后的存储规模，建立足够多的实例，初期时数据很少，所以每个节点存储的数据也非常少，但由于节点轻量的特性，数据之外的内存开销并不大，这使得只需要很少的服务器即可运行这些实例。日后存储规模扩大后，所要做的不过是将某些实例迁移到其他服务器上，而不需要对所有数据进行重新分片并进行集群下线和数据迁移了。

无论如何，客户端分片终归是有非常多的缺点，比如维护成本高，增加、移除节点比较繁琐等。Redis3.0版本支持集群功能。集群的特点在于拥有和单片实例同样的性能，同时在网络分区后能够提供一定的可访问性以及对主数据库故障恢复的支持。另外每个键都位于同一个节点中，则可以正常支持，否则会提示错误。除此之外集群还有一个限制是只能使用默认的0号数据库，如果执行SELECT切换数据库则会提示错误。

哨兵与集群式两个独立的功能，但从特性来看哨兵可以视为集群的子集，当不需要数据分片或者已经在客户端进行分片的场景下哨兵就足够使用了，但如果需要进行水平扩容，则集群式一个非常好的选择。

### 配置集群

使用集群，只需要将每个数据库节点的cluster-enabled配置选项打开即可。每个集群中至少需要3个主数据库才能正常运行。

cluster-enabled yes

集群会将当前节点记录的集群状态持久化地存储在指定文件中，这个文件默认为当前工作目录下的nodes.conf文件。每个节点对应的文件必须相同，否则会造成启动失败，所以启动节点时要注意最后为每个节点使用不同的工作目录，或者通过cluster-config-file选项修改持久化文件的名称：

cluster-config-file nodes.conf

每个节点启动后都会输出类似下面的内容

No cluster configuration found, I’m c21d9182eec935720f1622….

其中c21d9182eec935720f1622….表示该节点的运行ID，运行ID是节点在集群中的唯一标识；同一个运行ID可能地址和端口是不同的。

启动后，可以使用Redis命令行客户端连接任意一个节点使用INFO命令来判断集群是否正常启用了：

127.0.0.1:6379> INFO cluster

# Cluster

cluster\_enabled:1

其中cluster\_enabled为1表示集群正常启用了。现在每个节点都是完全独立的，要将它们加入同一个集群里面需要几个步骤。

Redis源代码中提供了一个辅助工具redis-trib.rb可以非常方便地完成这一任务。因为redis-trib.rb是用Ruby语言编写的，所以运行前需要在服务器上安装Ruby程序。redis-trib.rb依赖于gem包redis，可以执行gem install redis来安装。

使用redis-trib.rb来初始化集群，只需要执行：

[root@localhost ~]# /path/to/redis-trib.rb create --replicas 1 127.0.0.1:6380 127.0.0.1:6381 127.0.0.1:6382 127.0.0.1:6383 127.0.0.1:6384 127.0.0.1:6385

其中create参数表示要初始化集群，--replicas 1表示每个主数据库拥有的从数据库个数为1，所以整个集群共有3个主数据库以及3个从数据库。

首先redis-trib.rb会以客户端的形式尝试连接所有的节点，并发送PING命令以确定节点能够正常服务。如果有任何节点无法连接，则创建失败。同时发送INFO命令获取每个节点的运行ID以及是否开启了集群功能（即cluster\_enabled为1）

准备就绪后集群会向每个节点发送CLUSTER MEET命令，格式为CLUSTER MEET ip port，这个命令用来告诉当前节点指定ip和port上在运行的节点也是集群的一部分，从而使得6个节点最终可以归入一个集群。

然后redis-trib.rb会分配主从数据库节点，分配的原则是尽量保证每个主数据库运行在不同的IP地址上，同时每个从数据库和主数据库均不运行在同一IP地址上，以保证系统的容灾能力。

分配完成后，会为每个主数据库分配插槽，分配插槽的过程其实就是分配哪些键归哪些节点负责。之后对每个要成为子数据库的节点发送CLUSTER REPLICATE 主数据库的运行ID来将当前节点转换成从数据库并复制指定运行ID的节点（主数据库）。

此时整个集群的过程即创建完成，使用Redis命令行客户端连接任意一个节点执行CLUSTER NODES可以获得集群中的所有节点信息。

### 节点的增加

前面介绍过redis-trib.rb是使用CLUSTER MEET命令来使每个节点认识集群中的其他节点的，可想而知如果想要向集群中加入新的节点，也需要使用CLUSTER MEET命令实现。加入新节点非常简单，只需要向新节点（A）发送如下命令即可：

CLUSTER MEET ip port

ip和port是集群中任意一个节点的地址和端口号，A接收到客户端发来的命令后，会与该地址和端口号的节点B进行握手，使B将A认作当前集群中的一员。当B与A握手成功后，B会使用Gossip协议将节点A的信息通知给集群中的每一个节点。通过这一方式，即使集群中有多个节点，也只需要选择MEET其中任意一个节点，即可使新节点最终加入整个集群中。

### 插槽的分配

新的节点加入集群后有两种选择，要么使用CLUSTER REPLICATE命令复制每个主数据库来以从数据库的形式运行，要么向集群申请分配插槽来以主数据库的形式运行。

在一个集群中，所有的键会被分配给16384个插槽，而每个主数据库会负责处理其中的一部分插槽。虽然redis-trib.rb初始化集群时分配给每个节点的插槽都是连续的，但是实际上Redis并没有此限制，可以将任意的几个插槽分配给任意的节点负责。