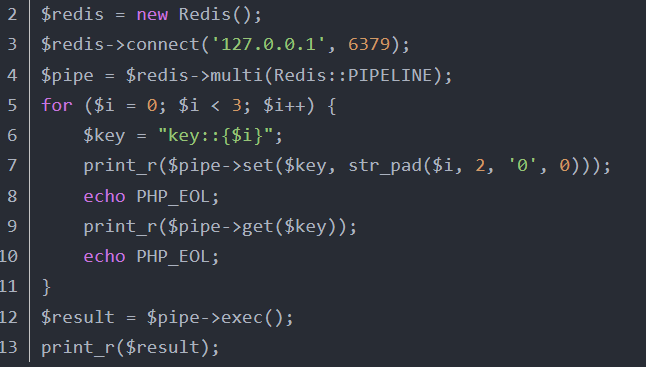
## redis

达到高性能

1. value尽可能小，大value在带宽限制下可能造成QPS骤降，redis是单线程的，过大尺寸的数据访问会block所有的其他操作
2. 使用Pipeline或者lua script，redis的所有的请求都是跨网络进行的，所以TCP Round Trip的长短对redis性能表现很重要。通常使用PipeLine，使得客户端可以一次性把一组redis命令发给redis server，或者预先在redis server中定义Lua script



1. 不开启RDB或者AOF

redis事务

ACID事务是一个非常重要的抽象，但是redis事务和ACID事务是不同的，redis将multi和exec指令之间的多个指令试做一个事务，一旦redis看到了exec就开始执行这一组指令，并保证执行过程中不被打断，除非redis本身或者机器挂掉，如果发生了就可能出现部分执行被执行。Redis的事务只支持Isolation不支持ACD

redis原子性

原子性是指一组操作，不受其他并发操作的干扰，这样进行的数据操作的值不会被相互覆盖。一组操作，要么不完成，要么完成。由于是单线程的，在完成一个操作前，不会有其他操作被执行。实际应用中，其实要保证一组命令的原子性，比如



可以使用redis事务或者Lua script来实现一组原子性

redis队列的局限性

1. 队列可能丢东西，如果redis队列挂掉，生产者必须停止服务，还是继续服务器但不在插入队列
2. 消费者是否需要一个commit，表示处理完了一个事件
3. 是否有事件重放的需要，比如队列的数据处理错了，希望能重新处理一遍之前所有的数据
4. 队列是不是需要有最大长度限制，如果生产者跟不上消费者速度，是否需要卡主生产者
5. 如果丢了一部分数据，找不回来

## redis config

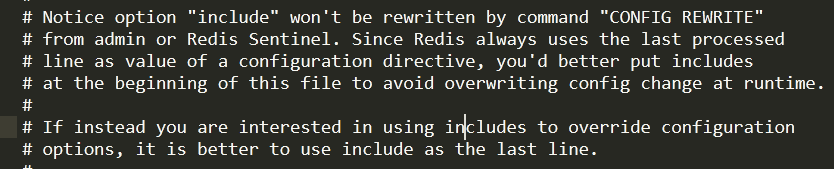
### daemonize

设置为yes时，代表开启守护进程模式，redis会在后台运行，并将pid号写入redis.conf的pidfile，此时redis会一直运行，除非手动kill

### include

引入一些个性化配置

这个指令不能被config rewrite重写????(疑问)



### pidfile

当redis运行，会将pid写入到这个文件中

pidfile /var/run/redis\_6379.pid

### port

监听端口，默认端口是6379

如果同一个机器，起了两个不同的端口，则两个redis的rdb和aof会写到同一个文件里面，除非指定不通端口的文件位置

### bind

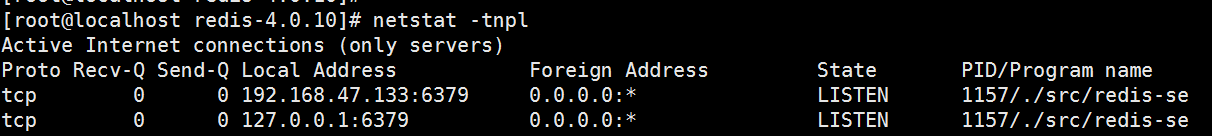
默认情况下，如果没有bind指令提供，redis将监听服务器上所有可用的网络端口的连接

bind 127.0.0.1 192.168.47.133，访问者可以使用127.0.0.1和192.168.47.133访问

如果其他机器访问192.168.47.133访问超时，则查看防火墙

iptables -I INPUT -p tcp --dport 6379 -j ACCEPT

bind 127.0.0.1 192.168.47.133

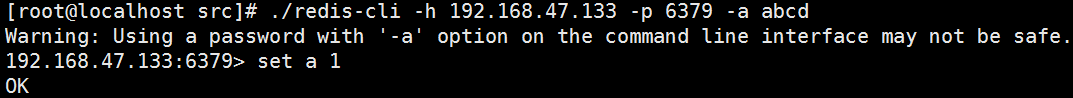


### requirepass

提供密码访问

requirepass 123456

redis-cli –a 123456



### protected-mode

为一个安全防护层，为了避免redis在网络上被利用

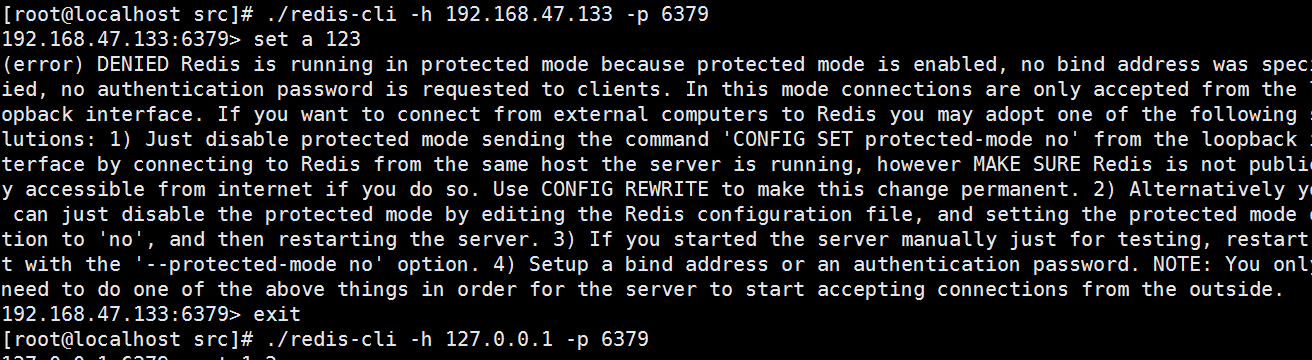
启动条件如果：

1. 没有显式的bind一组ip
2. 没有提供requirepass

在protected-mode yes情况下，如果没有bind，但是有requirepass，则保护失效

如果有bind，没有requirepass，则保护失效

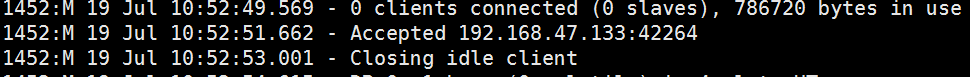
则使用非127.0.0.1连接redis会报错



### timeout

timeout 0 //关闭该功能

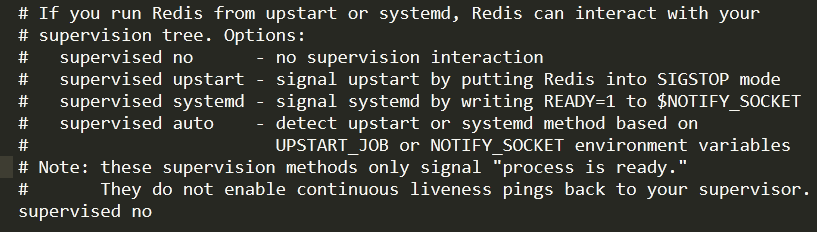
timeout 10 //在10秒内不操作就断开连接，再次操作不会报错



### supervision

supervision no

可以通过upstart或者systemd来交互redis



### loglevel

指定日志级别 loglevel notice

debug

会有大量信息

verbose

也会有大量信息，但是不会像debug那么混乱

notice

适度的信息，生产环境的选择

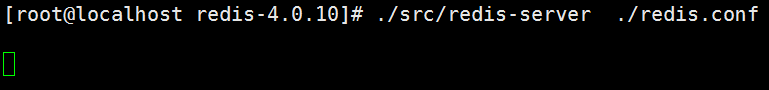
warning

只记录非常重要的错误信息

### logfile

logfile “” 空字符串将会不记录，如果使用daemonize然后又将日志记录到标准输出，则日志会写到/dev/null

如果指定了logile则不会将日志输出到标准输出



### database

设置数据库的数量

database 16

### maxclients

maxclients 2的话，则当连接第3个的时候，连接不会报错，但是操作会报错

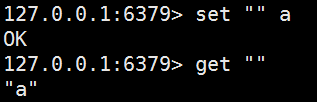
设置同时连接的最大数量，默认情况是10000个，一旦达到了则会关闭所有新连接并返回max number of clients reached

however if the Redis server is not able to configure the process file limit to allow for the specified limit the max number of allowed clients is set to the current file limit minus 32 (as Redis reserves a few file descriptors for internal uses).



## redis keys

key值是二进制安全的，这意味着可以用任何二进制序列作为key值，从形如”foo”的简单字符串到一个JPEG文件的内容都可以。空字符串也是有效key值



1. 太长的键值不是个好主意，例如1024字节的键值就不是个好主意，不仅因为消耗内存，而且在数据中查找这类键值的计算成本很高（redis是单线程的，操作都在一个线程队列里，其他操作要等待）
2. 最好坚持一种模式。例如：”object-type:id:field”就是个不错的注意，像这样”user:1000:password”

## redis数据库

redis.conf中database可以指定>=1，指定0会redis-server启动报错，可以指定0或者databases -1的数据

每个数据库是隔离的，不能共享

select dbnum

flushall会删除所有数据库的数据

flushdb会删除本库的

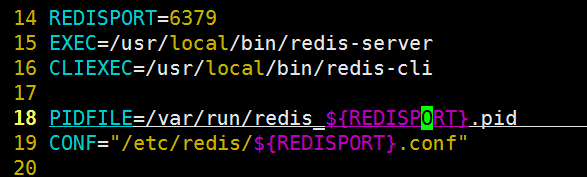
## redis开机自启

进入redis下载后的utils目录



vim redis\_init\_script

根据个性化自己修改



将redis\_init\_script拷贝到/etc/init.d，改名为redis

添加/etc/redis的x权限

将redis.conf的daemonize改为yes

systemctl stop redis.service

systemctl start redis.service

chkconfig redis on

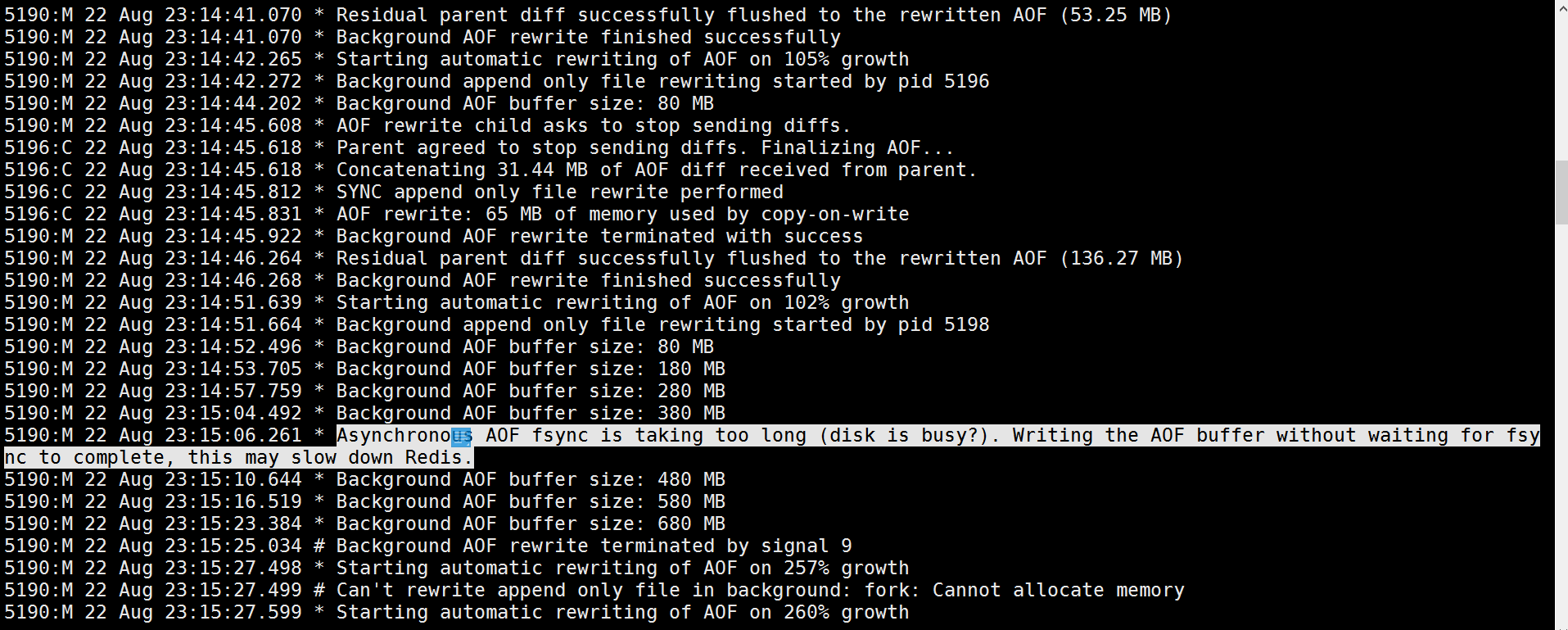
chkconfig –add redis

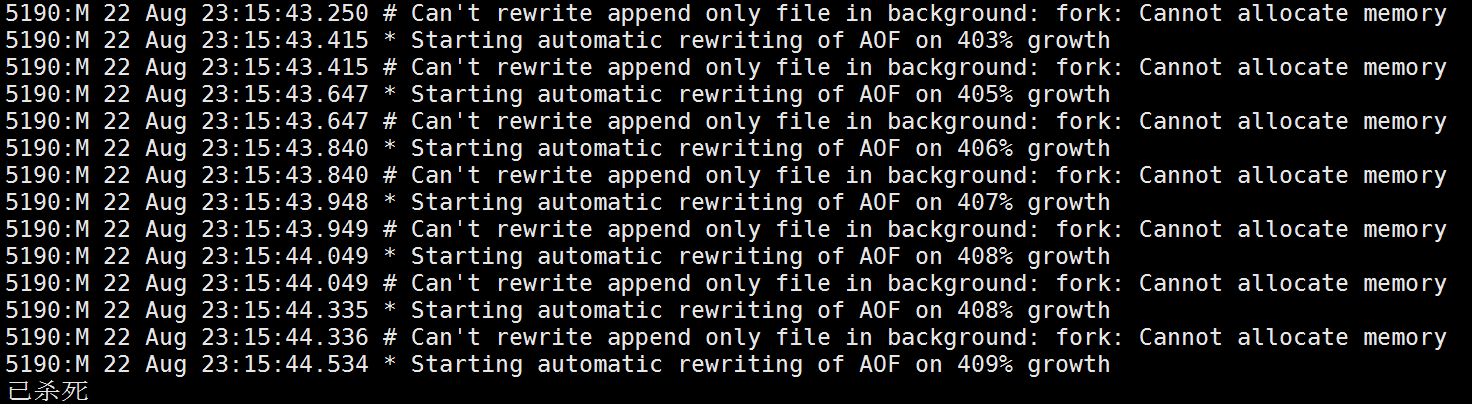
## redis持久化

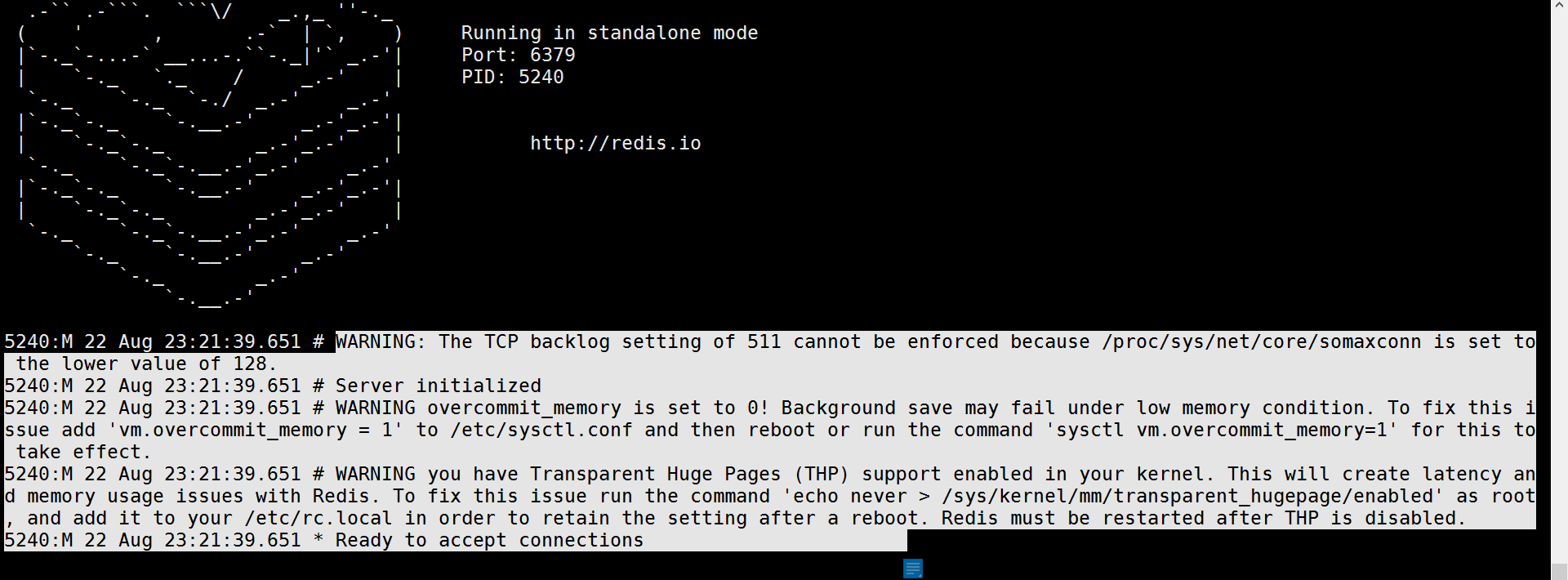
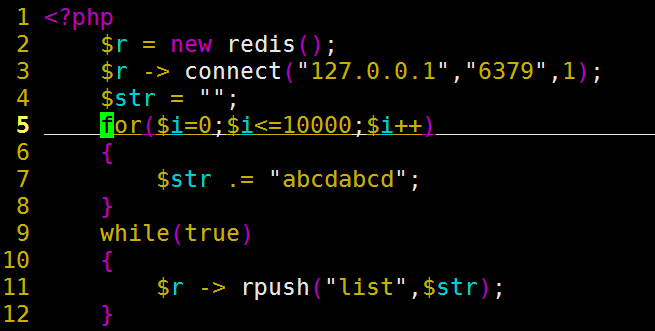
<https://www.cnblogs.com/cnmenglang/p/6225987.html>

在开启appendaof写入太多造成整个redis-server死掉，整个都不能连接了，kill了redis-server

Asynchronous AOF fsync is taking too long (disk is busy?). Writing the AOF buffer without waiting for fsync to complete, this may slow down Redis.







redis database //rdb

appendonly file //aof

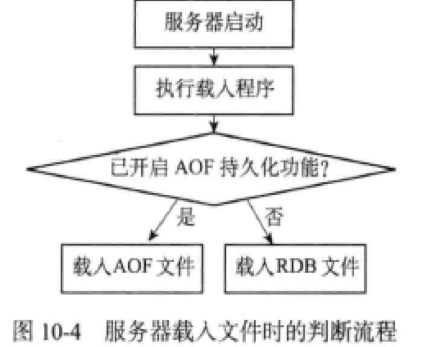
master通常使用AOF，slave使用snapshot

可以同时开启两种持久化，当redis重启的时候会优先载入AOF文件，因为在通常AOF保存的数据比RDB完整

RDB每次持久化内存都会翻倍，AOF持久化除了第一次启动的时候，剩下的内存使用量都很少，但是AOF文件很大

一般来说，应该同时使用两种持久化以达到数据安全，如果能承受几分钟内的数据丢失，应该使用RDB

有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快， 除此之外， 使用 RDB 还可以避免之前提到的 AOF 程序的 bug



### RDB

RDB持久化能够在指定时间间隔对你的数据进行快照存储

RDB文件是一个经过压缩的二进制文件

RDB是一个非常紧凑的文件,它保存了某个时间点得数据集,非常适用于数据集的备份,比如你可以在每个小时报保存一下过去24小时内的数据,同时每天保存过去30天的数据,这样即使出了问题你也可以根据需求恢复到不同版本的数据集

RDB是一个紧凑的单一文件,很方便传送到另一个远端数据中心

RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他IO操作，所以RDB持久化方式可以最大化redis的性能

RDB是将内存数据完整写入到磁盘一次，并不是是写入增量，如果数据量大的话，必然会引起大量的IO操作

与AOF相比,在恢复大的数据集的时候，RDB方式会更快一些

RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级内不能响应客户端的请求.如果数据集巨大并且CPU性能不是很好的情况下,这种情况会持续1秒,AOF也需要fork,但是你可以调节重写日志文件的频率来提高数据集的耐久度

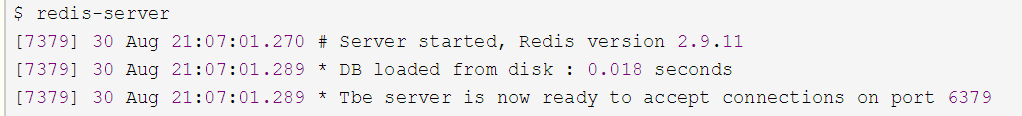
几种触发RDB的操作

1. 执行flushall
2. 执行save或者bgsave
3. 在指定的时间间隔内，执行指定次数的写操作
4. redis-server服务器正常退出(不管save是否生效)，如果save “” ，正常退出不会生成dump.rdb文件

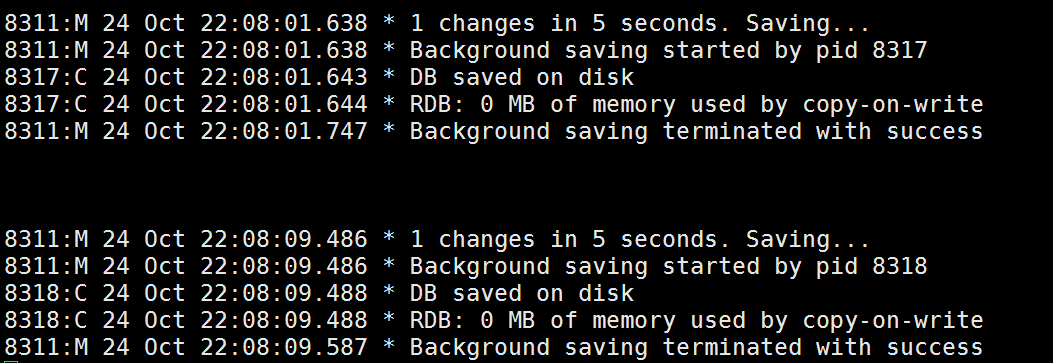
和使用SAVE命令或者BGSAVE命令创建RDB文件不同，RDB 文件的载入工作是在服务启动时自动执行的，所以Redis并没有专门用于载入RDB文件的命令，只要Redis服务器在启动时检测到RDB文件存在，它就会自动载入RDB文件

服务器在载入RDB文件期间，会一直处于阻塞状态

DB loaded from disk就是成功载入RDB文件



在redis.conf中save 5 1 save 100 600 就是5秒钟内有一个键值操作，100秒钟有600个键值操作，只要满足一个bgsave就会被自动执行



Redis.conf中

dbfilename dump.rdb //指定本地数据库文件名

dir ./ //本地数据库存放目录

rdbcompression yes //默认开启数据压缩，采用LZF压缩方式，占用了一点CPU时间

stop-write-on-bgsave-error yes //当snapshotting时出现错误无法继续时，是否阻塞客户端变更操作，错误可能是因为磁盘已经满了，磁盘故障，OS级别异常等

### AOF

AOF持久化记录每次对服务器写的操作，当服务器重启时候回重新执行这些命令来恢复原始数据，AOF以redis协议追加保存每次写的操作到文件末尾，REDIS还能对AOF文件进行后台重写，使得AOF文件体积不至于过大

是数据已经写入到文件或者即将写入，才进行实际的数据变更

AOF日志也不是完全按客户端的请求来生成日志的，比如命令 INCRBYFLOAT 在记AOF日志时就被记成一条SET记录，因为浮点数操作可能在不同的系统上会不同，所以为了避免同一份日志在不同的系统上生成不同的数据集，所以这里只将操作后的结果通过SET来记录

使用AOF 会让你的Redis更加耐久: 你可以使用不同的fsync策略：无fsync,每秒fsync,每次写的时候fsync.使用默认的每秒fsync策略,Redis的性能依然很好(fsync是由后台线程进行处理的,主线程会尽力处理客户端请求),一旦出现故障，你最多丢失1秒的数据

AOF文件是一个只进行追加的日志文件,所以不需要写入seek,即使由于某些原因(磁盘空间已满，写的过程中宕机等等)未执行完整的写入命令,你也也可使用redis-check-aof工具修复这些问题

AOF重写是重新生成一份AOF文件，新的AOF文件中一条记录的操作只会有一次，而不像一份老文件那样，可能记录了对同一个值的多次操作。其生成过程和RDB类似，也是fork一个进程，直接遍历数据，写入新的AOF临时文件。在写入新文件的过程中，所有的写操作日志还是会写到原来老的 AOF文件中，同时还会记录在内存缓冲区中。当重写操作完成后，会将所有缓冲区中的日志一次性写入到临时文件中。然后调用原子性的rename命令用新的 AOF文件取代老的AOF文件

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作

AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态

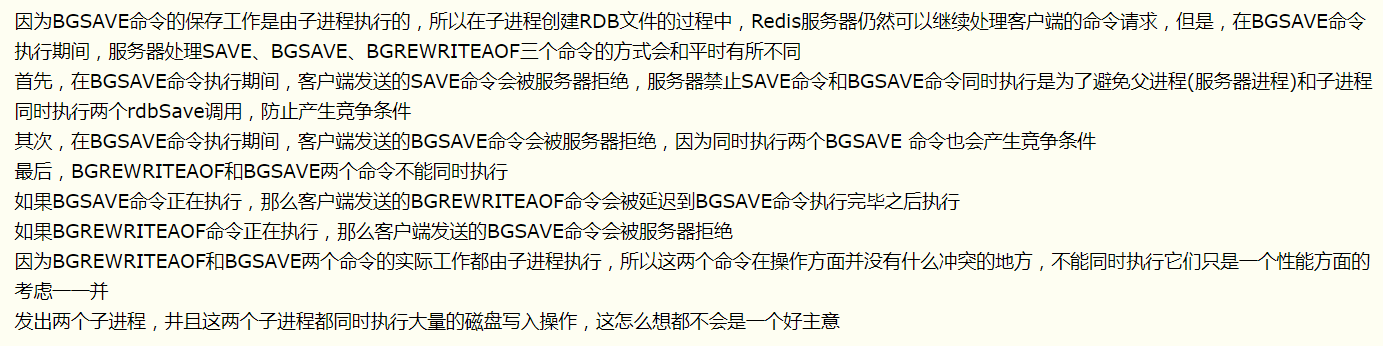
对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积

根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）

因为 AOF 的运作方式是不断地将命令追加到文件的末尾， 所以随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。举个例子， 如果你对一个计数器调用了 100 次 INCR ， 那么仅仅是为了保存这个计数器的当前值， AOF 文件就需要使用 100 条记录（entry）。然而在实际上， 只使用一条 SET 命令已经足以保存计数器的当前值了， 其余 99 条记录实际上都是多余的。

AOF的持久化是通过命令追加、文件写入和文件同步三个步骤实现的。当reids开启AOF后，服务端每执行一次写操作（如set、sadd、rpush）就会把该条命令追加到一个单独的AOF缓冲区的末尾，这就是命令追加；然后把AOF缓冲区的内容写入AOF文件里。看上去第二步就已经完成AOF持久化了那第三步是干什么的呢？这就需要从系统的文件写入机制说起：一般我们现在所使用的操作系统，为了提高文件的写入效率，都会有一个写入策略，即当你往硬盘写入数据时，操作系统不是实时的将数据写入硬盘，而是先把数据暂时的保存在一个内存缓冲区里，等到这个内存缓冲区的空间被填满或者是超过了设定的时限后才会真正的把缓冲区内的数据写入硬盘中。也就是说当redis进行到第二步文件写入的时候，从用户的角度看是已经把AOF缓冲区里的数据写入到AOF文件了，不过是把AOF缓冲区的内容放到了另一个但对系统而言只内存缓冲区里而已，之后redis还需要进行文件同步把该内存缓冲区里的数据真正写入硬盘上才算是完成了一次持久化

为了处理这种情况， Redis 支持一种有趣的特性： 可以在不打断服务客户端的情况下， 对 AOF 文件进行重建（rebuild）。执行 BGREWRITEAOF 命令， Redis 将生成一个新的 AOF 文件， 这个文件包含重建当前数据集所需的最少命令。Redis 2.2 需要自己手动执行 BGREWRITEAOF 命令； Redis 2.4 则可以自动触发 AOF 重写， 具体信息请查看 2.4 的示例配置文件



Redis.conf中

appendonly yes //默认关闭的

appendfilename “appendonly.aof”

#appendfsync always //同步持久化，每次发生数据变化会立刻写入到磁盘，性能较差但是数据保存完好

appendfsync everysec //默认配置，每秒异步记录一次。

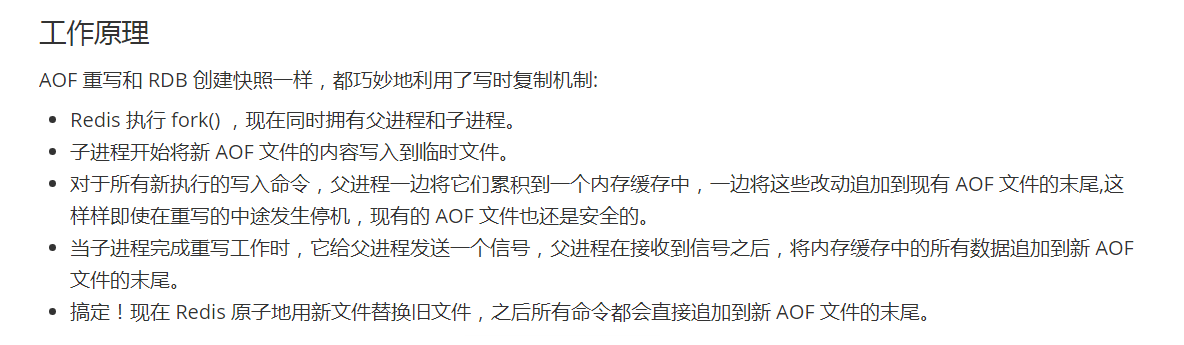
(存疑部分)fsync调用时长超过1秒时。Redis会采取延迟fsync的策略，再等一秒钟。也就是在两秒后再进行fsync，这一次的fsync就不管会执行多长时间都会进行。这时候由于在fsync时文件描述符会被阻塞，所以当前的写操作就会阻塞(存疑部分)

#appendfsync no //不同步，不会主动调用fsync，一切依赖于系统，对大多数linux，为30秒一次，或者缓冲区满了

Redis 打开 AOF 持久化功能后，Redis 在执行完一个写命令后，都会将执行的写命令追回到 Redis 内部的缓冲区的末尾。这个过程是命令的追加过程。 接下来，缓冲区的写命令会被写入到 AOF 文件，这一过程是文件写入过程。对于操作系统来说，调用write函数并不会立刻将数据写入到硬盘，为了将数据真正写入硬盘，还需要调用fsync函数，调用fsync函数即是文件同步的过程。只有经过文件同步过程，AOF 文件才在硬盘中真正保存了 Redis 的写命令

auto\_aofrewrite\_perc: 100 //aof文件大小超过旧aof百分之多少后触发bgrewriteaof。默认是100，意味着当前aof是旧aof大小的两倍时候触发bgrewriteaof。设置0可以禁用自动触发

auto\_aofrewrite\_min\_size: 64mb //当 AOF 文件需要达到多大体积时才进行 AOF 重写，后面的每次重写就不会根据这个变量了



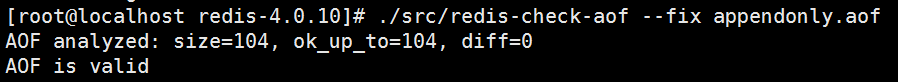
在重写的过程中，由于redis还会有新的写入，为了避免数据丢失，会开辟一块内存用于存放重写期间产生的写入操作，等到重写完毕后会将这块内存中的操作再追加到aof文件中

### AOF文件损坏

服务器可能在程序正在对aof文件写入时停止，如果停止造成了AOF文件出错，redis在重启时会拒绝载入这个aof文件，从而确保数据的一致性不会被破坏

修复

* + - 1. 为现有的AOF文件创建一个备份
      2. ./src/redis-check-aof –fix (原来的)filename



* + - 1. (可选)使用diff –u filename 备份aof文件来对比
      2. 重启redis服务器

### RDB切换为AOF

如果将conf的aof开启，然后重启服务器，所有数据都会丢失，因为开启aof服务会先读取aof文件，此时aof文件是空的

为最新的dump.rdb创建一个备份

将备份放到安全的地方

redis-cli config set appendonly yes

redis-cli config set save “”

执行第一个命令会开启aof功能，reids会阻塞知道初始aof文件创建完成为止，之后redis会继续处理命令请求，并开始将写入命令追加到aof文件末尾

然后更改conf的aof功能，将其开启，服务器重启之后， 之前通过 CONFIG SET 设置的配置就会被遗忘， 程序会按原来的配置来启动服务器

### RDB与AOF相互作用

在版本号大于等于 2.4 的 Redis 中， BGSAVE 执行的过程中， 不可以执行 BGREWRITEAOF 。 反过来说， 在 BGREWRITEAOF 执行的过程中， 也不可以执行 BGSAVE。这可以防止两个 Redis 后台进程同时对磁盘进行大量的 I/O 操作

如果 BGSAVE 正在执行， 并且用户显示地调用 BGREWRITEAOF 命令， 那么服务器将向用户回复一个 OK 状态， 并告知用户， BGREWRITEAOF 已经被预定执行： 一旦 BGSAVE 执行完毕， BGREWRITEAOF 就会正式开始。 当 Redis 启动时， 如果 RDB 持久化和 AOF 持久化都被打开了， 那么程序会优先使用 AOF 文件来恢复数据集， 因为 AOF 文件所保存的数据通常是最完整的

### 快照(snapshotting)

默认情况下，redis将快照保存在名字为dump.rdb的二进制文件中，可以对redis进行设置，让它在N秒内数据集至少有M个改动满足时，自动保存一次数据集。也可以通过save和bgsave操作手动让redis进行数据保存操作

save 60 100 //60秒内有100个键被改动

#### 工作方式

1. redis调用forks，同时拥有父进程和子进程
2. 子进程将数据写入到一个临时rdb文件
3. 当子进程完成对RDB文件的写入时，redis用新的rdb文件替换原来的rdb文件，并删除旧的rdb文件，这种工作方式可以从写时复制收益

### 追加文件Append-only file(AOF)

快照功能并不是非常耐久（dura ble）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。 从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化

你可以在配置文件中打开AOF方式

Appendony yes

可以配置redis多久才将数据fsync一次磁盘

1. 每次有新命令追加到AOF文件时就fsync一次，非常慢，也非常安全
2. 每秒fsync一次，和rdb持久化差不多，故障时候丢失1秒的数据，足够快
3. 从不fsync，将数据交给系统来处理，更快也更不安全

默认的也是推荐的是，每秒fsync一次

### redis持久化命令

#### Save

执行一个同步操作，以rdb文件的方式保存所有数据快照，很少在生产环境直接运行save，因为它会阻塞所有客户端的请求，知道快照恢复为止。应该用bgsave替代

#### Bgsave

后台保存DB。会立即返回 OK 状态码。 Redis forks, 父进程继续提供服务以供客户端调用，子进程将DB数据保存到磁盘然后退出。如果操作成功，可以通过客户端命令LASTSAVE来检查操作结果

会造成内存使用率大幅升高

#### Lastsave

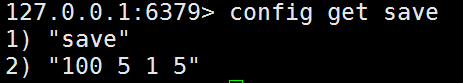
执行成功时返回UNIX时间戳。客户端执行 [BGSAVE](http://www.redis.cn/commands/bgsave.html) 命令时，可以通过每N秒发送一个 LASTSAVE 命令来查看[BGSAVE](http://www.redis.cn/commands/bgsave.html) 命令执行的结果，由 LASTSAVE 返回结果的变化可以判断执行结果

#### bgrewriteaof

cli模式主动重写aof文件，会创建一个子进程，然后由子进程负责对AOF文件的重写

#### Config get save

获取save配置



#### Config set save

设置save配置，即使生效，重启重新配置

Config set save 100 10

## LRU算法

当redis被当做缓存来使用，当你新增数据时，让它自动回收旧数据是件非常方便的事情

LRU是redis唯一支持的回收方法，redis的maxmemory指令用于将可用内存限制成一个固定大小，还包括了redis使用的LRU算法，这个实际上只是近似的LRU

maxmemory配置指令用于配置redis存储数据时指定限制的内存大小，通过redis.conf和config set可以配置

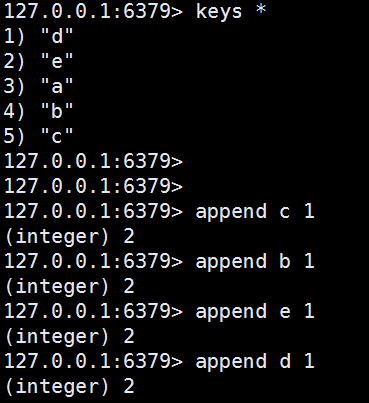
maxmemory 100mb

设置为0代表没有内存限制，对于64位系统这个是默认值，对于32位系统内存限制为3GB

当指定的内存限制大小达到时，需要选择不同的行为，也就是不同的策略。redis可以仅仅对命令返回错误，或者回收一些数据使得添加数据时可以避免内存限制

redis的maxmemory-policy

1. noeviction(默认)：返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令
2. allkeys-lru：尝试回收最少使用的键(LRU)，使得新添加的数据有空间存放



然后插入一个特别大的键，a被回收了

如果e是最后被访问的键，则e先被回收，然后插入一个特别大的键，但是还没有空间，则回收a，因为a最大

1. volatile-lru：尝试回收最少使用的键(LRU)，但仅限于从设置了过期时间的数据集中，使得新添加的数据有空间存放
2. allkeys-random:回收随机的键使得新添加的数据有存放空间
3. volatile-ttl:从设置了过期时间的数据集中，并且优先回收存活时间较短的键
4. volatile-random：从设置了过期时间的数据集中，随机选择一个数据进行释放

如果没有键满足回收的前提条件的话，策略volatile-lru, volatile-random以及volatile-ttl就和noeviction 差不多了。

选择正确的回收策略是非常重要的，这取决于你的应用的访问模式，不过你可以在运行时进行相关的策略调整，并且监控缓存命中率和没命中的次数，通过RedisINFO命令输出以便调优

一般的经验：

1. 使用allkeys-lru策略，部分的子集元素将比其他元素被访问的亘古多
2. allkeys-random，所有元素被访问的概率都差不多
3. volatile-ttl

redis的LRU算法并非完整的实现，这意味着redis并没办法选择最佳候选来回收，也就是最久未被访问的键。相反他会尝试运行一个近似的LRU算法，通过对少量keys进行取样，然后回收其中一个最好的key。从redis3.0算法已经改进为回收键的候选池子，改善了性能，使得更加近似真实的LRU算法

maxmemory-samples 5

每次回收时候检查的采样数量，以实现调整算法的精度

使用真实的LRU算法会占用很多内存

在所有key都是最近经常使用的，那么就需要选择allkeys-lru

如果所有的key访问概率都差不多，可以选用allkeys-random

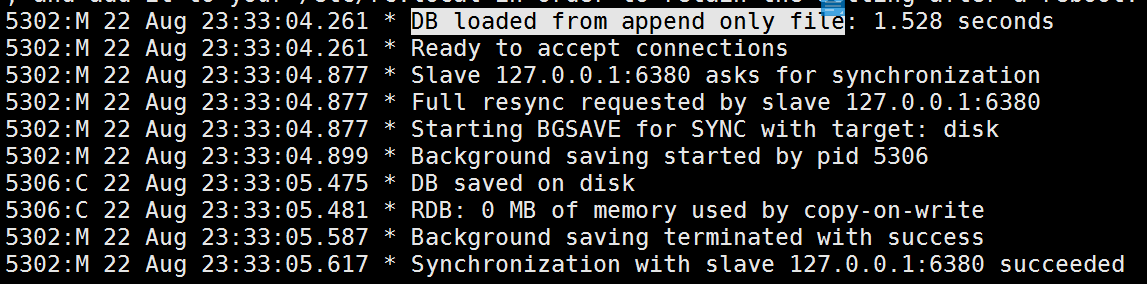
## 主从、复制(replication)

master写数据，如果slave有持久化，slave的持久化也是会生效的

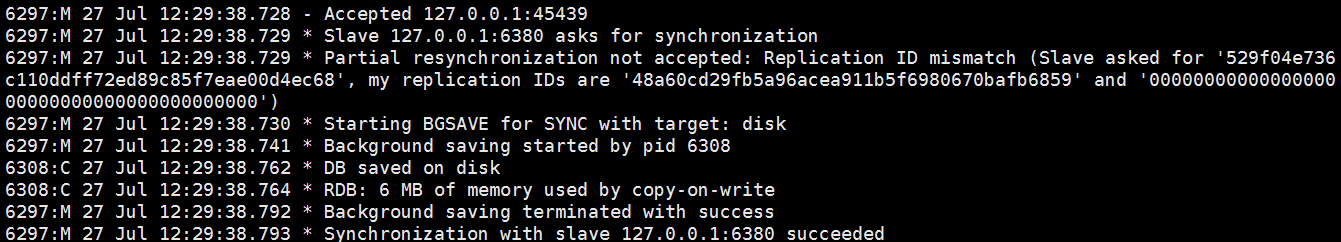
由于数据存储在一台服务器上，如果这台服务器出现硬盘故障，会导致数据丢失，为了避免单点故障，通常做法是将数据库复制多个副本以部署在不同的服务器上，一个master可以拥有多个slave，但是一个slave只能拥有一个master，一个slave可以接受其他的slave的连接

slave一般是只读，但是可以改为可写

当从服务器启动的时候，会向maste发送sync，主服务器收到sync后会在后台保存rdb，在保存rdb期间将收到的命令缓存起来，快照完成后，master会将快照和缓存的命令一起发送给slave，之后master每收到1个命令就同步发送给salve



**即使将主从的rdb都关闭掉，第一次复制也会产生主从的rdb文件**



从服务器slave需要精准的复制主redis服务器master的内容，每次当slave和master的连接断开，slave会自动重新连接到master上，slave都将尝试让自身成为master的精准副本

当一个master实例和一个slave连接正常，master会发送一连串的命令流来保持对slave的更新，以便于将自身数据的改变复制给slave

当 master 和 slave 之间的连接断开之后，因为网络问题、或者是主从意识到连接超时， slave 重新连接上 master 并会尝试进行部分重同步：这意味着它会尝试只获取在断开连接期间内丢失的命令流

当无法进行部分重同步时， slave 会请求进行全量重同步。这会涉及到一个更复杂的过程，例如 master 需要创建所有数据的快照，将之发送给 slave ，之后在数据集更改时持续发送命令流到 slave

Redis使用默认的异步复制，其特点是高延迟和高性能，是绝大多数 Redis 用例的自然复制模式。但是，从 Redis 服务器会异步地确认其从主 Redis 服务器周期接收到的数据量

redis使用异步复制，slave和master之间异步确认处理的数据量

redis复制在master侧是非阻塞的，意味着master在初次同步或者部分同步给slave时候时可以处理查询；在slave大部分也是非阻塞的，当slave进行初次同步时候，可以使用旧数据处理查询，不过需要配置。**在初次同步之后，旧数据必须被删除，同时加载新的数据（这个短暂的时间窗口内，会阻塞到来的连接请求）**，自 Redis 4.0 开始，可以配置 Redis 使删除旧数据集的操作在另一个不同的线程中进行，但是，加载新数据集的操作依然需要在主线程中进行并且会阻塞 slave

如果在master的db10中写数据，slave需要也在db10中才能看见

可以使用复制来避免 master 将全部数据集写入磁盘造成的开销：一种典型的技术是配置你的 master Redis.conf 以避免对磁盘进行持久化，然后连接一个 slave ，其配置为不定期保存或是启用 AOF。但是，这个设置必须小心处理，因为重新启动的 master 程序将从一个空数据集开始：如果一个 slave 试图与它同步，那么这个 slave 也会被清空

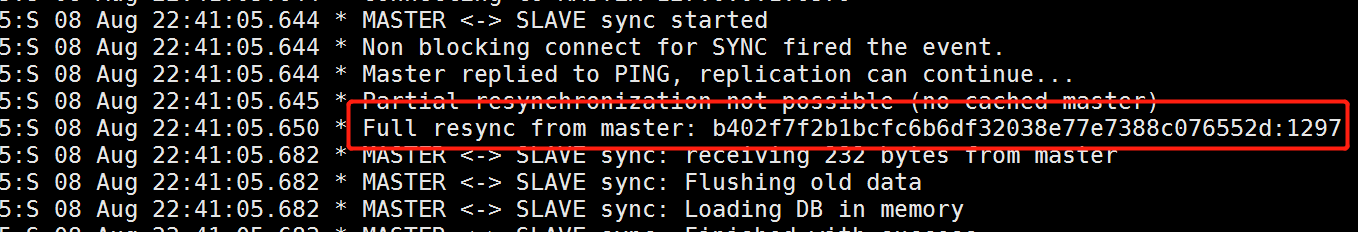
关闭master持久化，复制非常不安全。当master不可能启动持久化，应该配置实例来避免重置后自动重启

1. 我们设置节点 A 为 master 并关闭它的持久化设置，节点 B 和 C 从 节点 A 复制数据
2. 节点 A 崩溃，但是他有一些自动重启的系统可以重启进程。但是由于持久化被关闭了，节点重启后其数据集合为空
3. 节点 B 和 节点 C 会从节点 A 复制数据，但是节点 A 的数据集是空的，因此复制的结果是它们会销毁自身之前的数据副本

每一个redis master都有一个replication ID，这是一个较大的伪随机字符串，标记了一个给定的数据集。每个master也持有一个偏移量，master将自己产生的复制流发送给slave时，发送多少个字节的数据，自身的偏移量就会增加多少，目的是当有新操作修改自己的数据集时，它可以以此更新slave的状态。复制偏移量在没有一个slave连接到master时候，也会自增，基本上是每一个给定的

Replication ID，offset

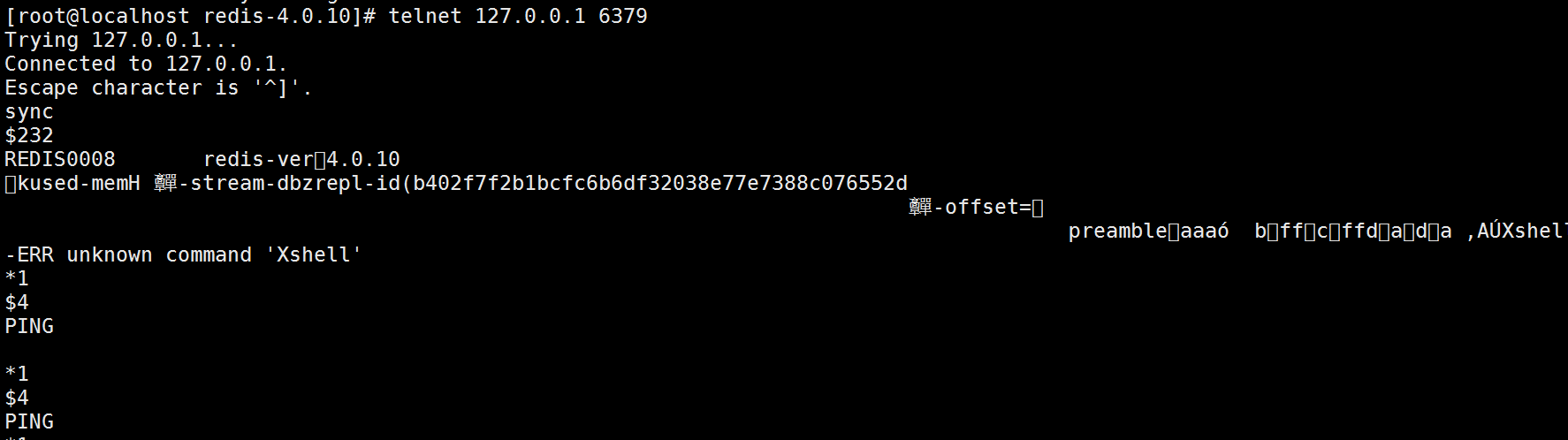
都会标识一个master数据集的确切版本



当slave连接到master时候，他们使用PSYNC命令发送他们记录的旧的master replication ID和他们至今为止处理的偏移量。通过这种方式，master能够仅发送slave所需的增量部分。但是如果master缓冲区没有足够的命令积压缓存记录，如果slave引用了不再知道的记录replication ID，则会转而进行一个全量冲同步，这时候slave会进行一个完整的同步，从头开始

master 开启一个**后台**保存进程，以便于生产一个 RDB 文件。同时它开始缓冲所有从客户端接收到的新的写入命令。当后台保存完成时， master 将数据集文件传输给 slave， slave将之保存在磁盘上，然后加载文件到内存。再然后 master 会发送所有缓冲的命令发给 slave。这个过程以指令流的形式完成并且和 Redis 协议本身的格式相同

可以用 telnet 自己进行尝试。在服务器正在做一些工作的同时连接到 Redis 端口并发出 [SYNC](https://redis.io/commands/sync) 命令。你将会看到一个批量传输，并且之后每一个 master 接收到的命令都将在 telnet 回话中被重新发出。事实上 SYNC 是一个旧协议，在新的 Redis 实例中已经不再被使用，但是其仍然向后兼容：但它不允许部分重同步，所以现在 PSYNC 被用来替代 SYNC



### 无磁盘复制

无需磁盘参与的复制，可以子进程直接发送RDB文件给slave，无需使用磁盘作为中间存储介质，redis2.8.18第一个支持无需磁盘复制的版本。正常情况下，一个全量重新同步要求在磁盘上创建一个RDB文件，然后将它从磁盘加载进内存，然后slave一次进行数据同步，如果磁盘性能很低的话，对master是一个很大的压力

redis.conf中

repl-diskless-sync

开启无磁盘复制

repl-diskless-sync-delay

可以延迟启动数据传输，目的是在第一个slave就绪后，等待更多的slave就绪

### 只读性质的slave

redis.conf中使用slave-read-only或者命令行config set控制slave只读

只读模式下slave会拒绝所有写入命令

在4.0版本之前，writeable slaves不能用生存时间来淘汰key，这意味着如果使用expire或者使用其他命令为key设置了ttl，将会在键值计算中看到这个key。4.0以后，writables slaves能够像master一样驱逐ttl的key了，但是DB编号大于63的key除外（没有模拟出来）

writeable slaves仅能本地，并且不会将数据传输到与该实例连接的sub-slave上，sub-slave将总是接受最顶层master向 intermediate slaves 发送的复制流相同的复制流

A->B->C

在节点B写数据，C是看不见的，C将拥有和master实例A相同的数据，在B写数据，A也看不见

如果A是master，B为A的slave，然后配置C为B的slave，B为C的master，在B写，C看不见，A写B、C能看见

### slave对master的验证

如果master通过requirepass设置了密码，要在运行的实例上运行次操作，可以使用

config set masterauth <password>

要永久设置的话，可以在redis.conf中

masterauth <password>

### 只允许写入N个附加的副本

在2.8开始，只有当至少N个slave连接到master时候，才有可能配置redis master接受写，但是由于redis使用异步复制，因此无法确保slave是否实际接受到给定的写命令，总有一个数据丢失窗口

redis slave每秒都会ping master，确认已处理的复制流的数量

redis master会记得上一次从每个slave都收到ping的时间

用户可以配置一个最小的slave数量，使得它滞后<=最大秒数

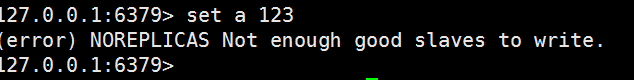
如果至少有N个slave，并且滞后小于M秒，则写入将被接受，如果不满足条件，master将会回复一个error并且写入不被接受

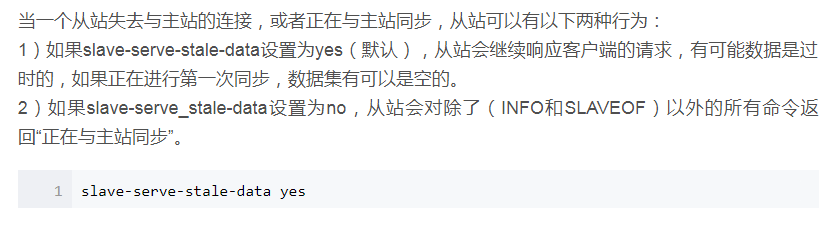
min-slaves-to-write <slave数量>

min-slaves-max-lag <秒数>

如果min-slaves-to-write和min-slaves-max-lag至少有一个是0，则该功能被禁止，默认min-slaves-to-write为0，min-slaves-max-lag为10

如果不满足配置，则master的写会报错





### 处理过期的key

redis slave需要正确的复制具有过期时间的key，redis不能依靠主从的时钟同步，因为这是一个无法解决race condition和数据集不一致问题，所以主要使用：

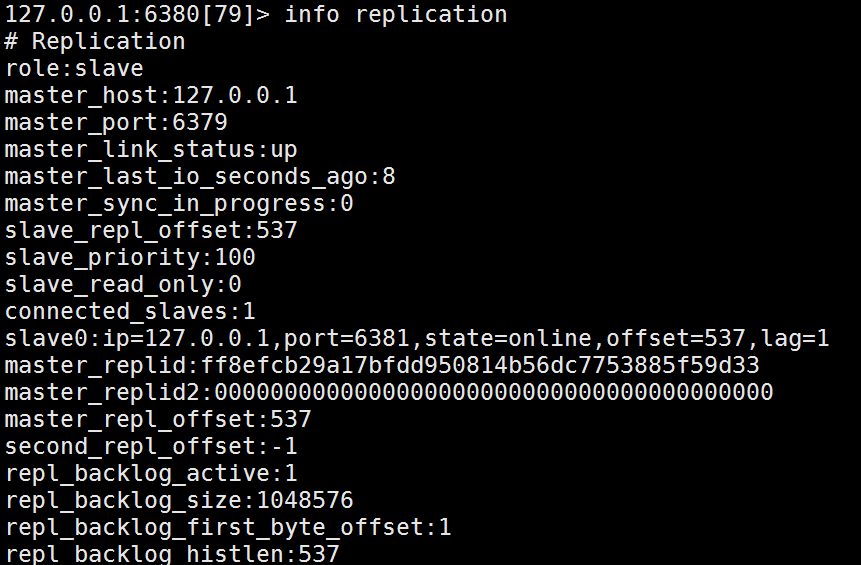
1.slave 不会让 key 过期，而是等待 master 让 key 过期。当一个 master 让一个 key 到期（或由于 LRU 算法将之驱逐）时，它会合成一个 DEL 命令并传输到所有的 slave

2.由于这是 master 驱动的 key 过期行为，master 无法及时提供 DEL 命令，所以有时候 slave 的内存中仍然可能存在在逻辑上已经过期的 key 。为了处理这个问题，slave 使用它的逻辑时钟以报告只有在不违反数据集的一致性的读取操作（从主机的新命令到达）中才存在 key。用这种方法，slave 避免报告逻辑过期的 key 仍然存在

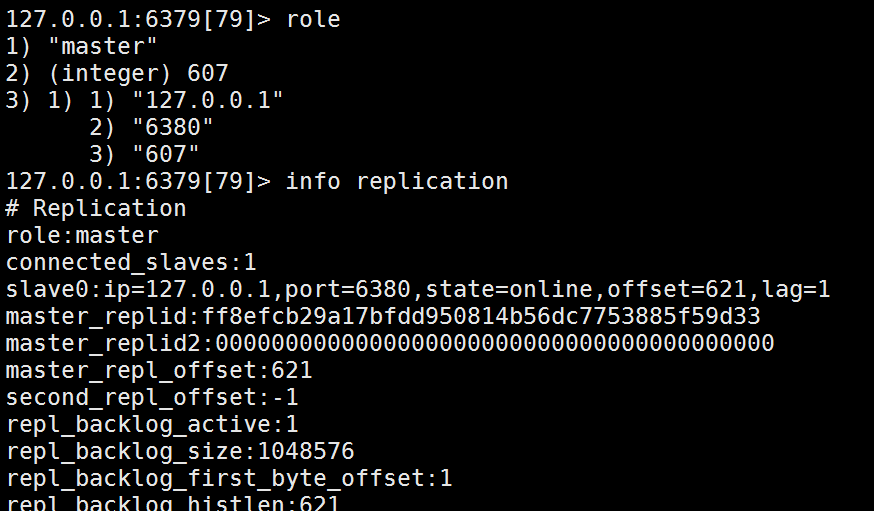
3.Lua脚本执行期间，不执行任何 key 过期操作。当一个Lua脚本运行时，从概念上讲，master 中的时间是被冻结的，这样脚本运行的时候，一个给定的键要么存在要么不存在。这可以防止 key 在脚本中间过期，保证将相同的脚本发送到 slave ，从而在二者的数据集中产生相同的效果

### INFO与ROLE

使用info replication可以查看主从配置



但是不够直观，可以使用role命令，其中第二个参数为sentinel



### 搭建

除了在redis.conf中配置slaveof，还可以在命令行中执行slaveof

cp redis.conf redis\_6379.conf(master)

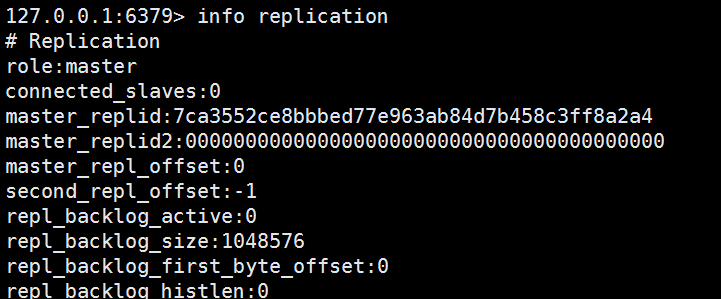
cp redis.conf redis\_6380.conf(slave)

将redis\_6379.conf和redis\_6380.conf的pidfile更改、port更改

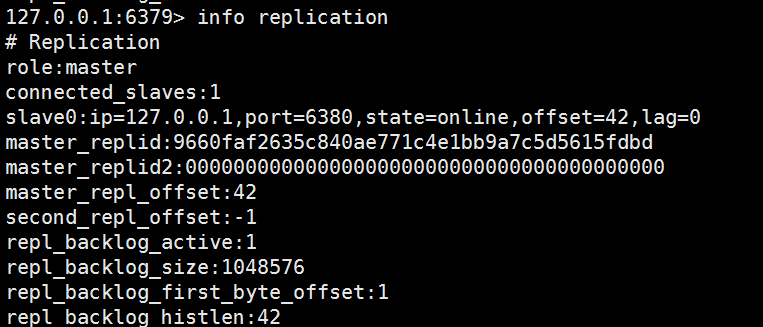
将redis\_6380.conf增加slaveof 127.0.0.1 6380

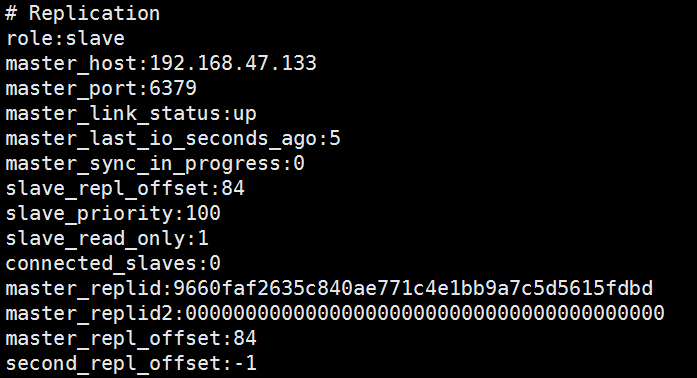
分别启动redis\_server 6380 6379

当只有6379启动的时候



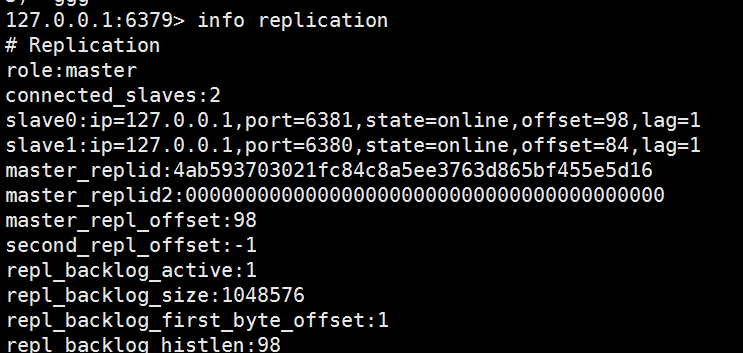
有6380启动的时候，查看6379的info





在slave写数据会报错



配置一主多从的时候，比如slave1为6380，slave2为6381，则更改redis\_6381.conf的slaveof就可以

## sentinel哨兵

用于管理多个redis服务器

1. 监控Monitoring，会不断的检查主服务器与从服务器是否正常运行
2. 提醒Notification，当被监控的redis出现问题，哨兵可以通过API向管理员或者其他应用发送通知
3. 自动故障迁移Automatic failover，当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会将失效主服务器的其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器

定期任务:

1. 每个sentinel以每秒种一次的频率向它所知的主服务器、从服务器以及其他的sentinel实例发送一个PING
2. 如果一个实例距离最后一次回复有效PING命令的时间超过down-after-millseconds所指定的值，那么这个实例会被sentinel标记为主观下线
3. 如果一个主服务器被标记为主观下线，那么正在监控这个主服务器的sentinel要以每秒一次的频率确认主服务器的确进入了主观下线
4. 如果一个主服务器被标记为主观下线，并且有足够数量的sentinel（至少要达到配置文件指定数量）在指定的时间范围内同意这一判断，那么这个主服务器被标记为客观下线
5. 一般情况下，每个sentinel会每10秒一次频率向已知的所有主服务器发送INFO命令，当一个主服务器被sentinel标记为客观下线，sentinel向下线主服务器的所有从服务器发送INFO命令的频率会从10秒一次改为1秒一次
6. 当没有足够数量的sentinel统一主服务器已经下线，主服务器的客观下线就会被移除，当主服务器重新向sentinel回复有效命令，主服务器的主观下线状态就会被移除

redis sentinel是一个分布式系统，可以在架构中运行多个sentinel进程，这些进程使用；流言协议gossip protocols来接受关于主服务器是否下线的信息，并且使用投票协议agreement protocols来决定是否执行自动故障迁移，以及从那个服务器作为新的主服务器

虽然 Redis Sentinel 释出为一个单独的可执行文件 redis-sentinel ， 但实际上它只是一个运行在特殊模式下的 Redis 服务器， 你可以在启动一个普通 Redis 服务器时通过给定 –sentinel 选项来启动 Redis Sentinel   
 启动：

1. ./redis-sentinel ./sentinel.conf
2. ./redis-server ./redis.conf ./sentinel.conf --sentinel

启动必须指定相应的配置文件，如果启动不指定或者配置文件不可写，那么sentinel会拒绝启动

### 主观、客观下线

1. 主观下线(Subjectvely Down)，简称SDOWN，指的是单个sentinel实例对服务器做出的判断
2. 客观下线(Objectively Down)，简称ODOWN，指的是多个sentinel在同一个服务器做出的SDOWN判断，并且通过sentinel的is-master-down-by-addr命令互相交流之后，得出服务器下线判断，一个sentinel可以通过向另一个sentinel发送SENTINEL is-master-down-by-addr来询问对方是否认定给定的服务器已下线

如果一个服务器没有在设定的sentinel down-after-milliseconds选项所指定的时间，对向他发送PING的sentinel返回一个有效回复，那么久标记为下线

服务器对PING命令的有效回复：（三种以外或者回复超时，则认为回复无效）

1. +PONG
2. -LOADING错误
3. -MASTERDOWN错误

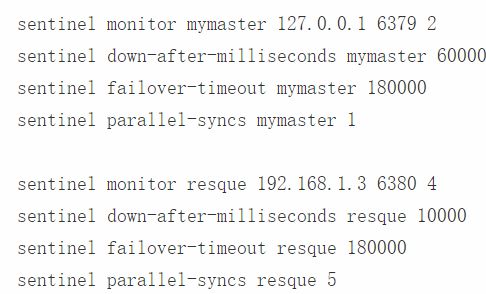
服务器必须在sentinel down-after-milliseconds选项所指定的时间内一直返回无效才会认为主观下线，如果设定了30000毫秒30秒，那么在29秒返回正确，这个服务器都会被认为正常

客观下线只适用于主服务器，对于任何其他类型的服务器，sentinel在判断为下线前不需要协商

只要有一个sentinel发现某个主服务器进入了客观下线，这个sentinel就可能会被其他sentinel推选出，并对失效的主服务器进行自动故障迁移

### 配置

运行一个sentinle需要的最少配置



**sentinel monitor mymaster 127.0.0.1 6379 2**

监控一个名字叫mymaster的主服务器，将这个主服务器判断为失效至少需要2个sentinel同意，只要同意sentinel的数量不达标，自动迁移就不会执行

不过要注意，无论你设置要多少个Sentinel同意才能判断一个服务器失效，一个 Sentinel都需要获得系统中多数（majority）Sentinel的支持，才能发起一次自动故障迁移，并预留一个给定的配置纪元（configuration Epoch 一个配置纪元就是一个新主服务器配置的版本号）

在只有少数（minority）Sentinel进程正常运作的情况下，Sentinel是不能执行自动故障迁移的

**sentinel down-after-milliseconds mymaster 6000**

如果服务器在给定的毫秒数之内，没有返回 Sentinel发送的 PING 命令的回复， 或者返回一个错误，那么 Sentinel将这个服务器标记为主观下线（subjectively down，简称 SDOWN ）不过只有一个Sentinel将服务器标记为主观下线并不一定会引起服务器的自动故障迁移：只有在足够数量的Sentinel都将一个服务器标记为主观下线之后，服务器才会被标记为客观下线（objectively down，简称 ODOWN），这时自动故障迁移才会执行

**sentinel parallel-syncs mymaster 1**

指定了在执行故障转移时，最多可以有多少个从服务器同时对新的主服务器进行同步，这个数字越小完成故障转移所需的时间就越长

如果服务器允许使用过期数据集，redis.conf中slave-server-stale-data，那么可能不希望所有从服务器都在同一时间向新的主服务器发送同步请求，因为从服务器在载入主服务器发过来的RDB文件，会阻塞请求，如果全部服务器一起对新的主服务器进行同步，那么就可能造成所有服务器在短时间内不能处理命令

## redis Strings

这是最简单Redis类型。如果你只用这种类型，Redis就像一个可以持久化的memcached服务器（注：memcache的数据仅保存在内存中，服务器重启后，数据将丢失）

值可以是任何种类的字符串（包括二进制数据），例如你可以在一个键下保存一副jpeg图片。值的长度不能超过512 MB

### set

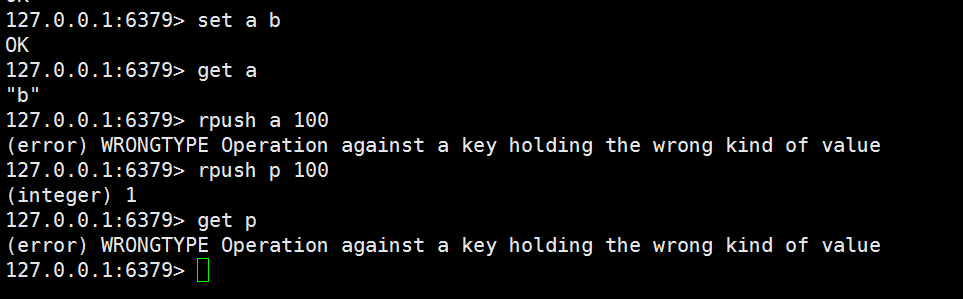
用于设定给定的key值，如果key已经存在，set就覆盖旧值，且无视类型

在设置成功时候返回OK

set a “abc”与set a abc

### get

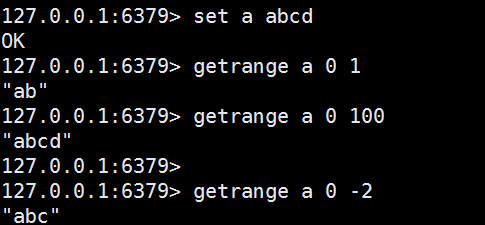
返回key的值，如果key不存在，返回nil。如果key不是字符串类型，返回一个错误



### getrange

用于获取存储在指定 key 中字符串的子字符串。字符串的截取范围由 start 和 end 两个偏移量决定(包括 start 和 end 在内)

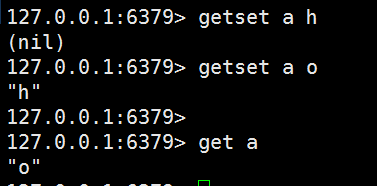
相当于返回从第start到end的字符，从0开始



### getset

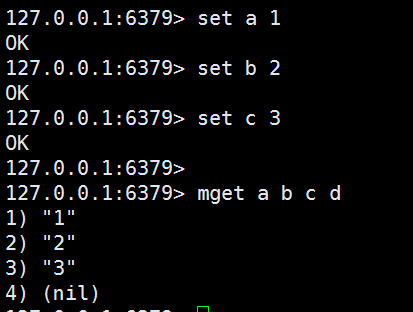
返回给定的key的旧值，当key没有旧值时，就是key不存在的时候返回nil，并且设定key-value

当key存在的时候，返回旧值，并设置新值



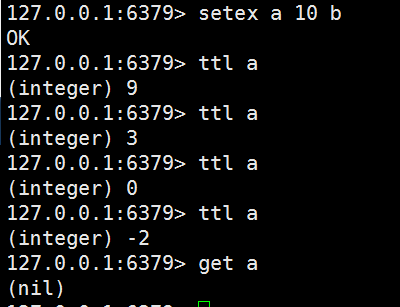
### mget

返回一个或者多个给定的key，如果给定的key有某个key不存在，则返回特殊的nil



### setex

为给定的key设置值和过期时间，过期时间为秒

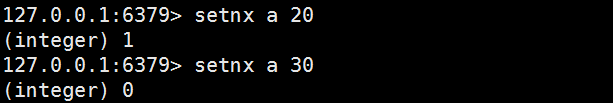


### setnx

set is not exists

在key不存在的时候，为key设置指定的值

设置成功返回1，失败返回0



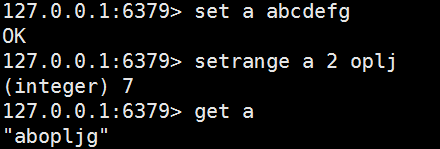
### setrange

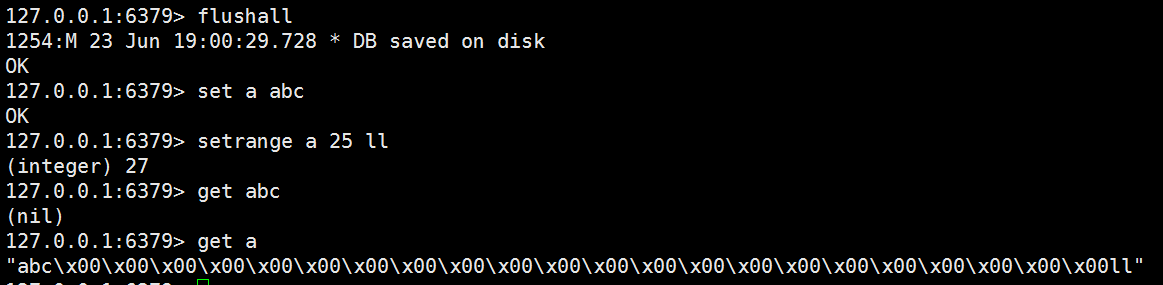
用指定的字符串覆盖给定key所存储的字符串值

返回修改后的字符串长度

没有的位数用\x00代替

当键不存在时候，也会创建





### strlen

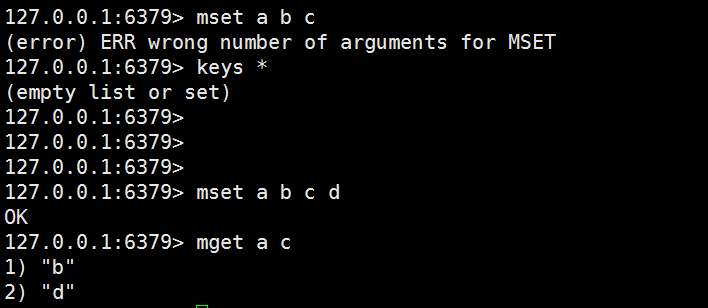
返回字符串的长度，key不存在时候返回0

当key存储的不是一个字符串时候，返回错误

### mset

同时设置多个键值

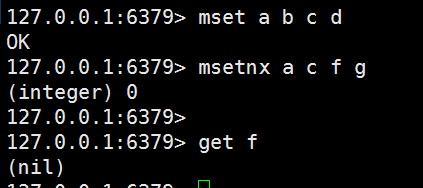
如果有一个设置失败，则返回0，全部插入失败



### msetnx

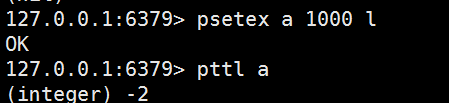
所有给定key都不存在时候，同时设置一个或者过个键值对

只要有一个设置失败，则全部设置失败



### psetex

以毫秒为单位设置key的生存周期



### incr

将key中存储的数字增1

如果key不存在，则先被初始化为0，然后再incr

如果值格式不对，则返回错误

值限制在64位bit有符号数字表示之内，strlen key <= 19

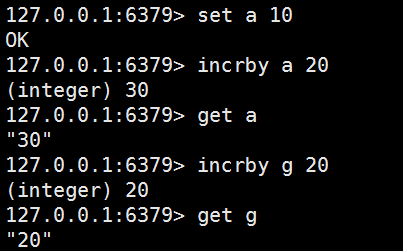
### incrby

将key中存储的数字增加指定的量

如果key不存在，则先被初始化为0，然后再incrby

如果值格式不对，则返回错误

值限制在64位bit有符号数字表示之内，strlen key <= 19



### incrbyfloat

增加浮点型数字

### decr

减，同incr

### decrby

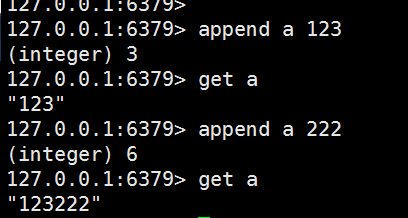
减，同incrby

### append

追加值

如果key不存在，则相当于set

返回执行后的字符串的长度



### setbit

现在还不知道啥意思

### getbit

现在还不知道啥意思

## redis Hash

不支持过期时间设置

hset a b 200

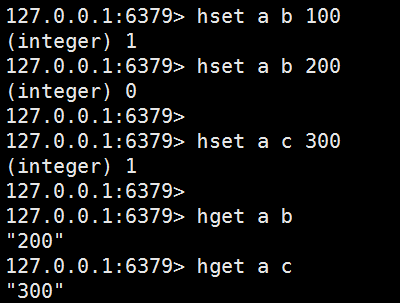
只能expire a 20

### hset

为哈希表中的字段赋值

如果哈希表不存在，一个新的哈希表被创建并进行hset，返回1

如果存在则旧值将被覆盖，返回0

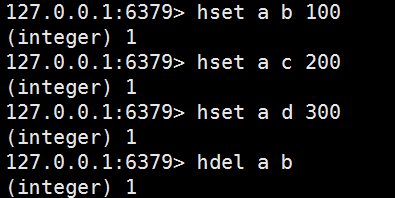


### hdel

该命令用于删除哈希表key中的一个或者多个指定字段，不存在的将被忽略

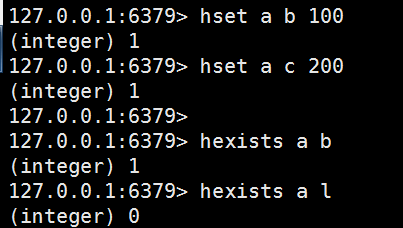
返回被成功删除字段的数量，不包含被忽略的字段

如果要删除整个hash，使用del



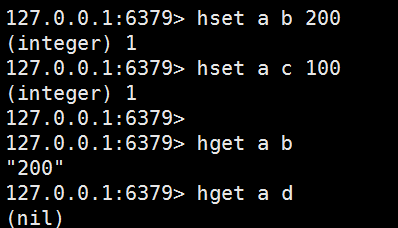
### hexists

存在返回1，不存在返回0



### hget

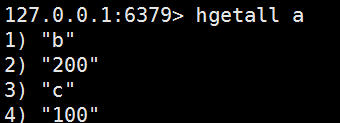
返回给定字段的值，不存在返回nil



### hgetall

获取hash表中所有字段和值

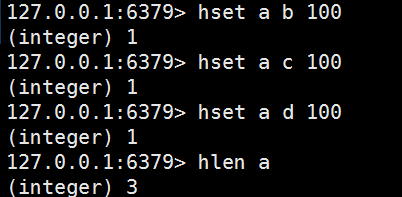
每个字段名之后是值，长度是哈希表的两倍



### hlen

获取hash表中字段的数量

不存在返回0



### hincby

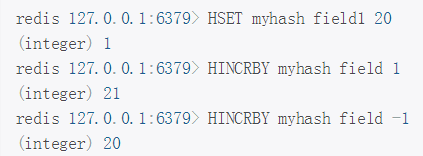
为hash表中字段的值加上指定的增量值

值可以为负数

如果不存在，一个新的值将会被创建

对字符串操作将会造成错误

值限制在64位bit有符号数字表示之内，strlen key <= 19

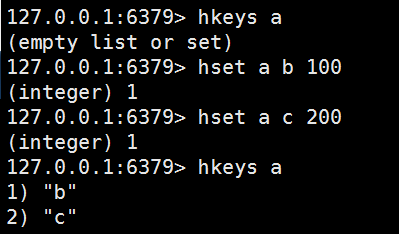


### hincrbyfloat

### hkeys

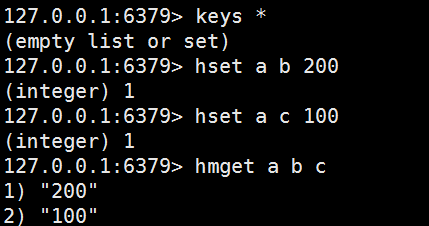
获取hash表中所有的字段名

不存在则返回空列表



### hmget

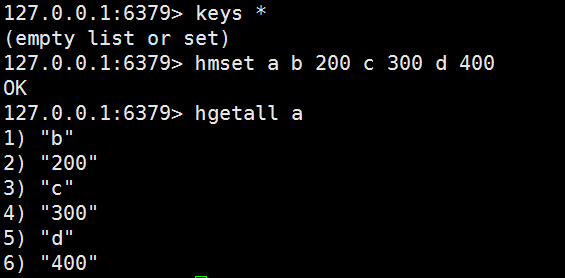
返回一个包含多个给定字段关联值的表



### hmset

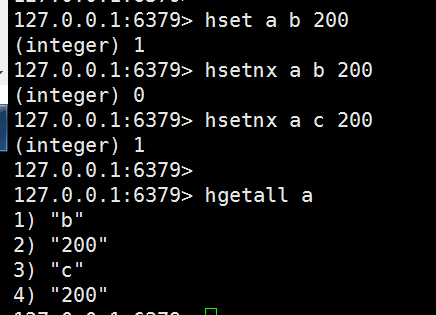
同时将多个值设置到hash表中

会覆盖已有的值

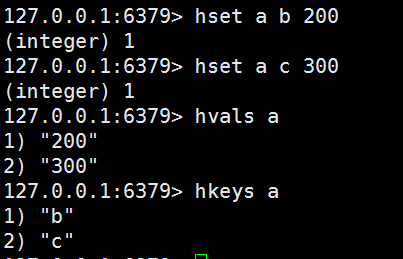


### hsetnx

为不存在的字段赋值



### hvals



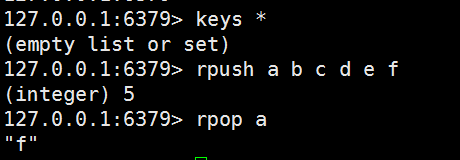
## redis list

### rpush

将一个或者多个值插入到列表最右面

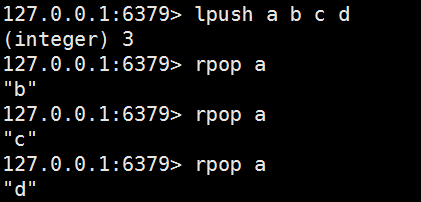
在redis2.4以前，只支持单个value值

返回操作后的list长度



### lpush

将一个或多个值插入到列表头部，如果key不存在则创建一个空列表



### lpushx

将一个或者多个值插入到已存在的列表头部，列表不存在则操作无效



### rpushx

插入一个或者多个，不存在则无效

### lpop

左面弹出，当列表不存在或者为空的时间返回nil

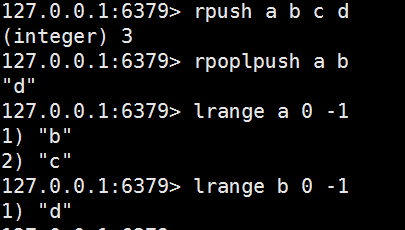
如果将列表都弹出，则列表被删除

### rpop

右面弹，当列表不存在或者为空的时间返回nil

### rpoplpush

没有lpoprpush

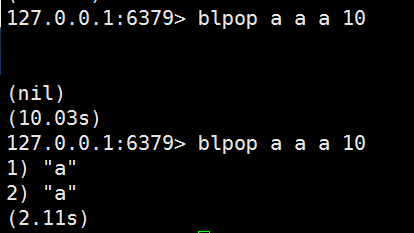


### blpop

移除并获取列表的第一个元素，如果列表没有元素会阻塞列表直到等待超时或者发现可弹出的元素位置

如果列表是空，就一直等待，直到超时。如果有值，就返回一个两个值，一个是键名，一个是值

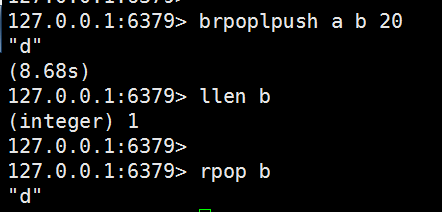
blpop a b c 10相当于在a、b、c中弹，10秒内如果没有值就取消阻塞



### brpop

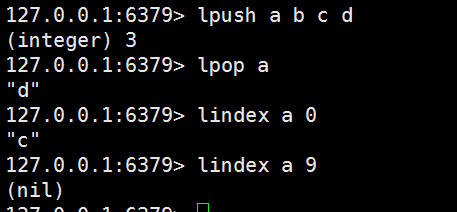
### brpoplpush

从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回他。如果没有元素会阻塞列表直到等待超时



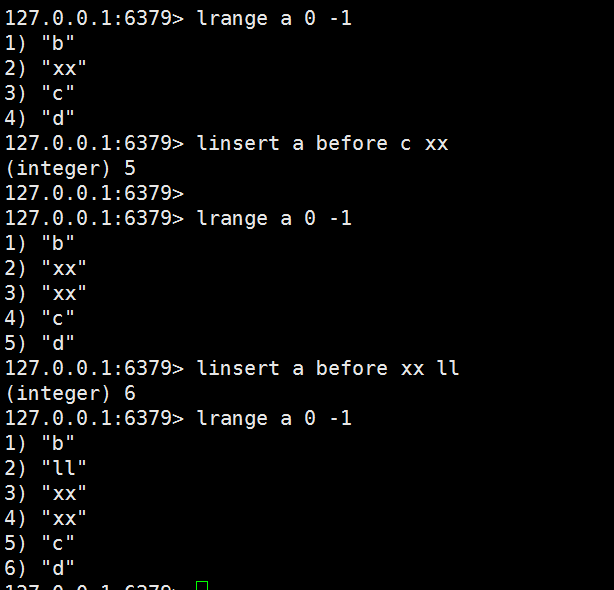
### lindex

通过索引获取列表中的元素，负数下标-1是最后一个元素，-2为倒数第二个



### linsert

用于在列表的元素前或者后面插入元素，当指定元素不存在时，不执行操作。当列表不存在，不执行操作



### llen

返回列表长度

如果列表不存在返回0

### lrange

1表示第二个元素，-1表示最后一个元素



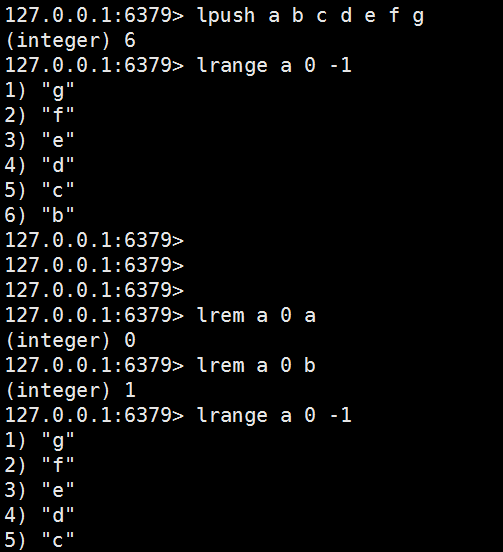
### lrem

根据参数count的值，移除列表中与参数value相等的元素

count>0，从表头开始向表尾部搜索，移除与value相等的元素，数量为count

count<0，从表尾开始向表头搜索，移除与value相等的元素，数量为count的绝对值

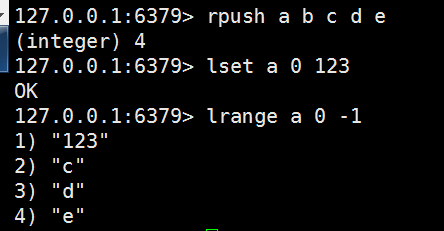
count=0，移除表中全部与value相等的值



### lset

通过索引设置列表元素的值

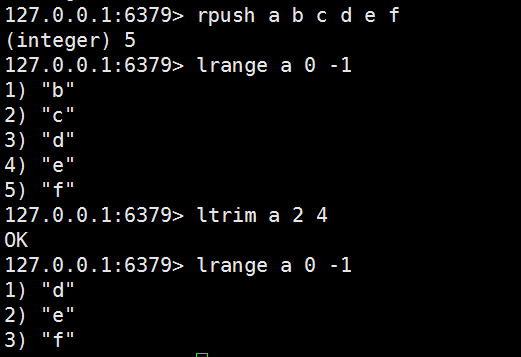
当索引超出范围或对一个空列表进行list时，返回一个错误



### ltrim

让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除

下标 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。 你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素



## redis set

### sadd

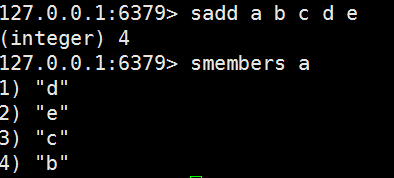
将一个或者多个元素加入到集合中，已经存在于集合的元素将会被忽略

无序的

假如集合不存在，则创建一个只包含添加元素的成员集合

在2.4版本以前，只能sadd单个元素

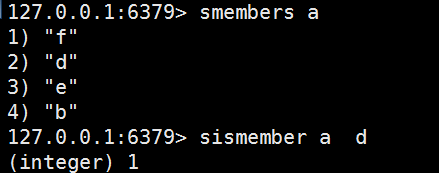
如果集合中已存在被添加的值，则添加失败



### smembers

返回集合中所有的成员

### sismember



### srandmember

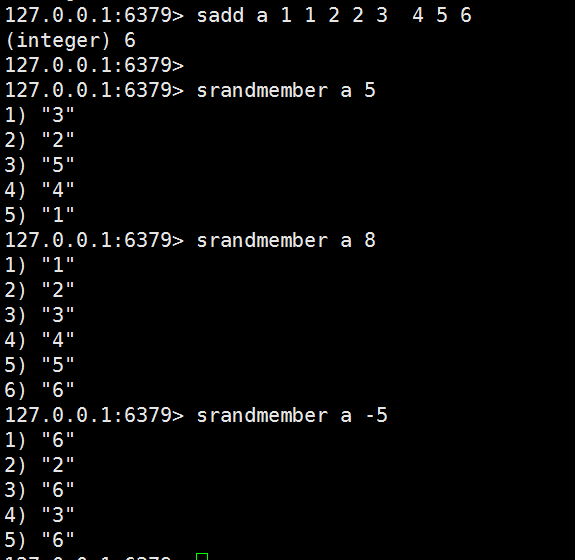
返回集合中一个或多个随机数

从 Redis 2.6 版本开始， Srandmember 命令接受可选的 count 参数

如果 count 为正数，且小于集合基数，那么命令返回一个包含 count 个元素的数组，数组中的元素各不相同。如果 count 大于等于集合基数，那么返回整个集合。

如果 count 为负数，那么命令返回一个数组，数组中的元素可能会重复出现多次，而数组的长度为 count 的绝对值。

该操作和 SPOP 相似，但 SPOP 将随机元素从集合中移除并返回，而 Srandmember 则仅仅返回随机元素，而不对集合进行任何改动。



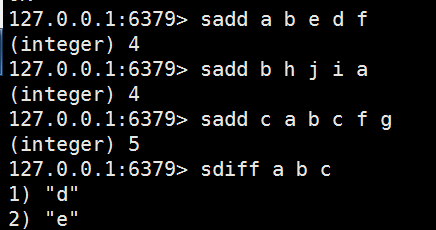
### scard

返回集合中元素的数量

### sdiff

返回给定所有集合的差集

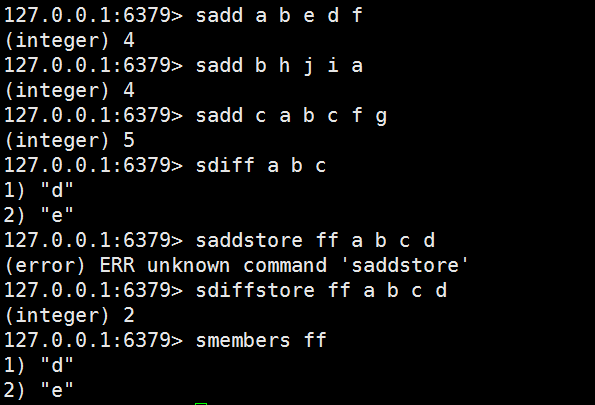
不存在的集合视为空集合



### sdiffstore

返回给定所有集合的差集并存储在一个集合中

如果指定存在在的集合存在，则会覆盖



### sunion

并集

### sunionstore

### sinter

交集

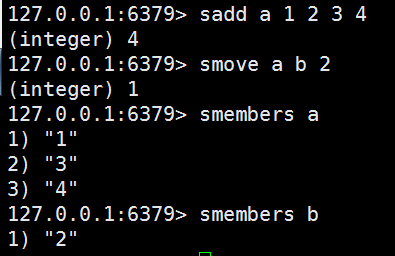
### sinterstore

交集并存储

### smove

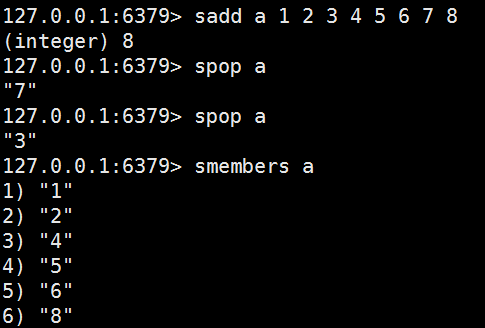
将指定成员 member 元素从 source 集合移动到 destination 集合

如果 source 集合不存在或不包含指定的 member 元素，则 SMOVE 命令不执行任何操作，仅返回 0 。否则， member 元素从 source 集合中被移除，并添加到 destination 集合中去



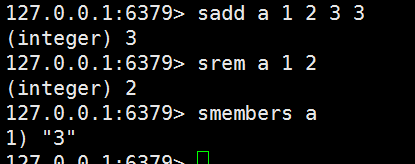
### spop

随机移除并返回集合中一个随机元素



### srem

移除集合中一个或者多个成员



### Sscan

## Redis sorted set

有序集合和集合一样，也是string类型元素的集合，且不允许重复元素

不同的是每个元素都会关联一个double类型的分数，redis正是通过分数来为集合中的成员进行从小到大排序

成员是唯一的，但是分数score却可以重复

每个集合可以存40多亿个元素

### Zadd

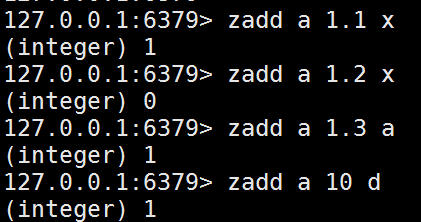
将一个或多个成员元素及其分数值填入到有序集合中

如果某个成员已经是有序集的成员，那么更新这个成员的分数值，并通过重新插入这个成员元素，来保证该成员在正确的位置上

分数值可以是整数值或双精度浮点数。

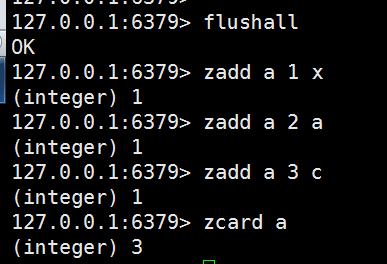
如果有序集合 key 不存在，则创建一个空的有序集并执行 ZADD 操作。

当 key 存在但不是有序集类型时，返回一个错误



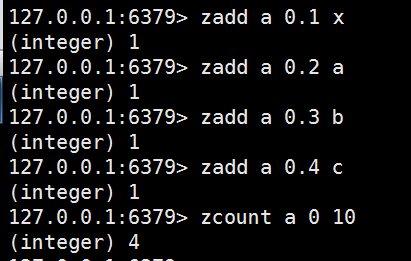
### zcard

用于计算集合中元素的数量



### zcount

计算有序集合中指定分数区间的成员数量



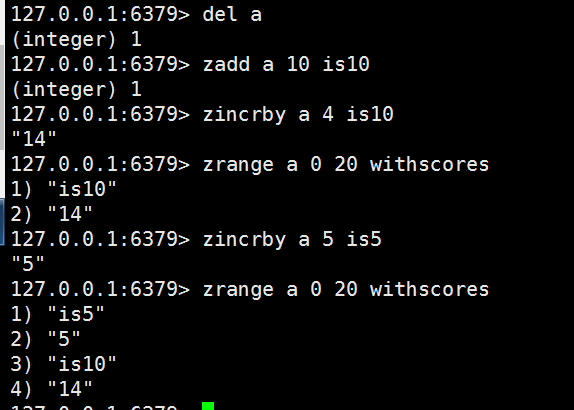
### zincrby

对有序集合中指定成员的分数加上增量increment

值可以为负数

zincrby key -5 member

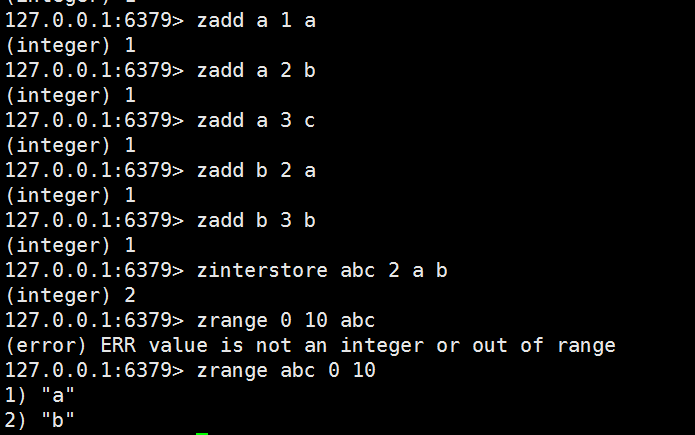
当key不存在，或member不存在，相当于zadd



### zinterstore

计算给定的一个或多个有序集合的交集，其中给定key的数量必须以numbers参数指定，并将交集存储到destination





### zlexcount

计算有序集合中指定字段区间内成员数量

