redis

【version 6.0】

# linux下安装

redis官网：redis.io

中文官网：<https://www.redis.net.cn/>

reids下载地址：<https://redis.io/download/#redis-downloads>

下载后需要上传至opt目录下

## 解压缩redis安装包

tar -zxvf redis安装包

## 进入到redis的加压缩目录进行安装

安装前需要先安装gcc-c++

安装指令 yum -y install gcc-c++

在redis安装包解压缩目录下执行make进行安装即可

\*\* 如果是先执行了make 发现找不到gcc 在执行安装gcc 那么此时需要先执行make distclean清理调之前的安装

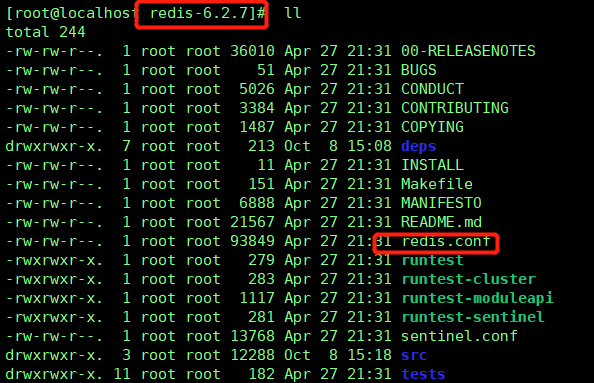
最后执行make install 如果一切正常则表示安装成功

跳过run make test 因为如果要运行redis的测试程序还需要tcl的组件

# 修改配置文件

在redis的解压缩目录下有一个redis.conf的文件 其就是redis的核心配置文件

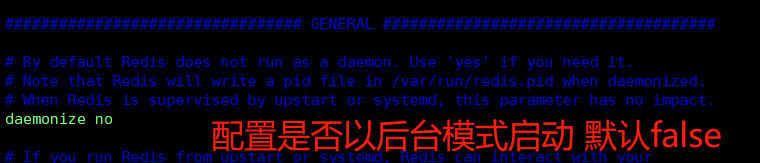
修改前建议备份



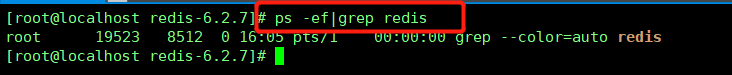
备份指令 cp redis.conf /myredis/

## 修改内容

将daemonize no改为也是yes 表示以后台程序运行



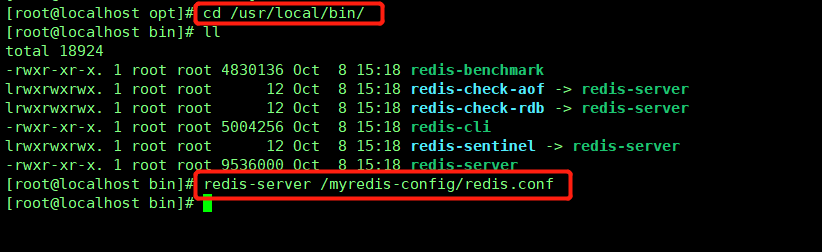
## 查看redis是否启动

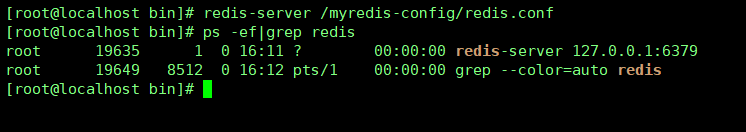


# 启动redis

在/usr/local/bin下 执行redis-server /myredis/redis.conf

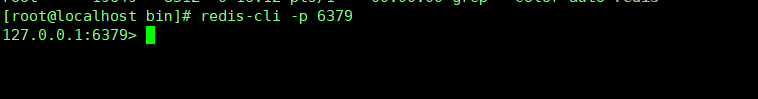
\*\*启动时可以指定启动哪个redis配置文件的redis 就是说redis的配置文件可以有多份 每份内容不同 当我们启动时可以自由选择对哪个配置文件的redis进行启动



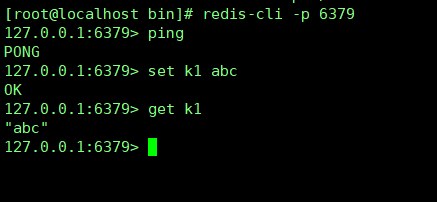


# 登录redis

在usr/local/bin下执行redis-cli -p 6379



## 简单测试



# 关闭redis

关闭redis服务 在客户端中使用shutdown指令

关闭redis客户端 在客户端中使用exit指令



# redis杂项知识

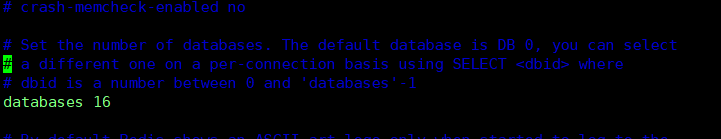
## 模式

redis采用单进程模式处理客户端请求，对于读写事件的响应是通过对epoll函数封装来实现的 实际上redis的处理速度完全依赖于主进程的执行效率。

* epoll函数是Linux内核中处理大批量文件的描述符 在Linux中是多路复用的增强版

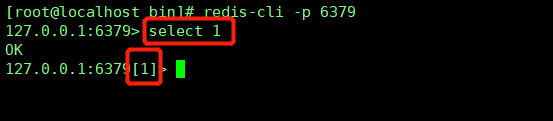
## redis库

redis数据库的个数是在redis.conf中进行配置的 默认16个 从下标0到下标15 默认进入的数据库是下标为0的库，可以通过select+数据库下标方式进行选择不同的数据库



### 切换不同的数据库

select+数据库下标



## redis中Key操作

### 查看当前库中所有的key

首先我们要知道 redis采用的是KV模式来存储数据的，那么每个Key对应的就是一条数据。

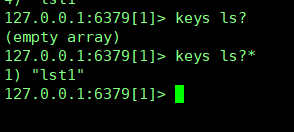
keys \* 可以查看当前库中所有的key

### 模糊查询匹配的key

使用?占位符 一个?表示一个位 \*表示全部 可以组合使用

例如模糊查询lst1这个key

keys ls?\*

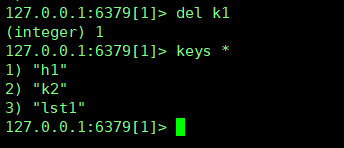


### 删除key

删除key就相当于删除了这条数据

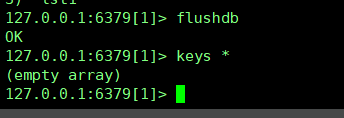
#### 删除当前的key

del +keyName



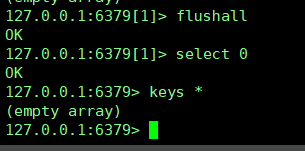
#### 删除当前库中的所有key

flushdb



#### 删除全部库的key

flushall



### redis的索引

redis的索引是从下标0开始

### redis的端口

redis的端口默认6379 可以在redis.conf中进行修改默认端口

### redis安全管理

redis是可以配置通过密码形式访问，但不建议这样做

因为redis是运行在linux系统中的 本身就是一个很好的屏障，如果设置了访问密码 则每次访问都需要进行密码验证 这对程序来说无疑增加了开销

#### 设置密码

#### 通过密码访问

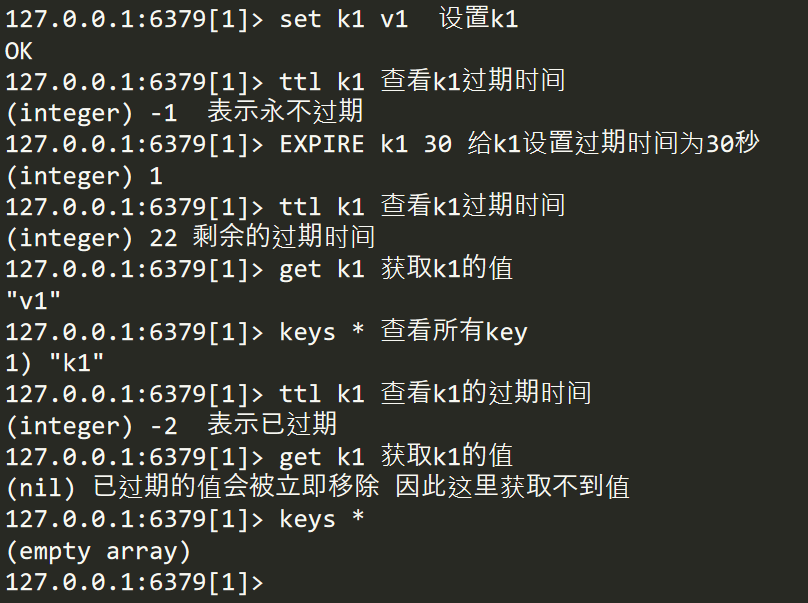
#### 取消密码

# key的操作

redis中对于key的操作

在redis中key表示键值对中的键 我们可以通过对其进行一些简单的操作来获取我们需要的信息 例如获取当前key的类型，过期时间等。

|  |  |
| --- | --- |
| 关键字 | 说明 |
| keys \* | 查看当前库中所有的key |
| exists keyName | 判断某个key是否存在 存在返回1 否则0 |
| move keyName dbIndex | 移动某个key到其它的库中 dbIndex是数据库的下标 |
| expire keyName seconds | 给key设置过期时间 seconds是过期的秒数 0表示立即过期 |
| ttl keyName | 查看当前的key剩余过期时间 |
| type keyName | 查看key的数据类型 |
| del keyName | 删除key |



# 数据类型

redis中常用的数据类型有五种 分别如下

string,list,set,zset,hash

这五种数据类型共同点是 都是KV模式存储数据

除了string外 其它四种数据类型都是对string的一种细化

例如 list 可以直接将元素插入到集合的头部或尾部，hash则使用了类似Map嵌套的原理 -- key对应的是一个键值对等等

参考文档：http://redisdoc.com/

## string

string类型是redis中最常用的数据类型

是单值 单value形式存储数据的

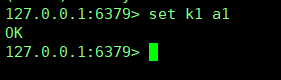
常用方法

### set 设置值

set 用来给string类型设置值

语法格式

set keyName value

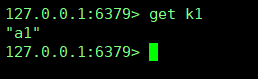


### get 获取值

get 用来获取string类型中的值

语法格式

get keyName

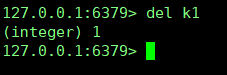


### del 删除key

删除指定的key

语法格式

del keyName



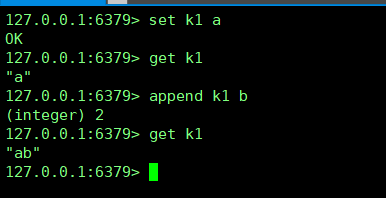
### append 追加内容

向string类型的key中追加内容

语法格式

append keyName appendValue

appendValue是要追加的内容



### strlen 获取长度

获取key中元素个数

语法格式

strlen keyName

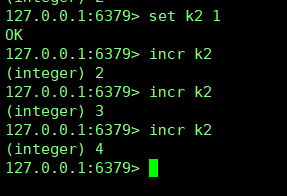


### incr 递增1

incr必须是对数字操作

语法格式

incr keyName

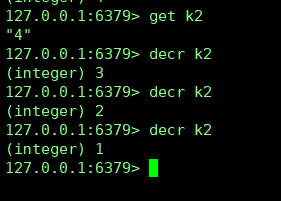


### decr 递减1

decr必须对数字操作

语法格式

decr keyName

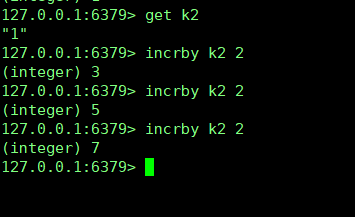


### incrby 指定递增大小

incrby必须对数字操作

语法格式

incrby keyName

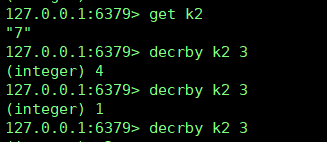


### decrby 指定递减大小

decrby必须对数字操作

语法格式

decrby keyName

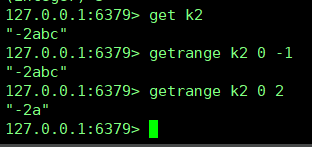


### getrange 获取范围内的值

类似oracle的between and 用来获取指定范围内的值 这个范围指的是下标索引的范围,0到-1表示全部范围

语法格式

getrange keyName 0 -1



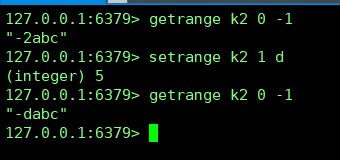
### setrange 元素添加到指定位置

替换指定下标的元素

语法格式

setrange keyName index value

index表示元素的下标索引值



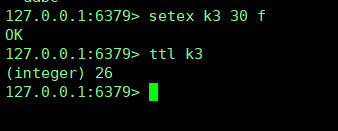
### setex 指定过期时间

已存在的key指定过期时间 使用expire keyName seconds

新增key的时候直接指定过期时间 setex

语法格式

setex keyName seconds value

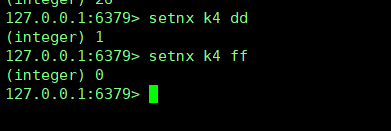


### setnx 不存在则添加

当key不存在时则进行添加 nx => not exists

语法格式

setnx keyName value

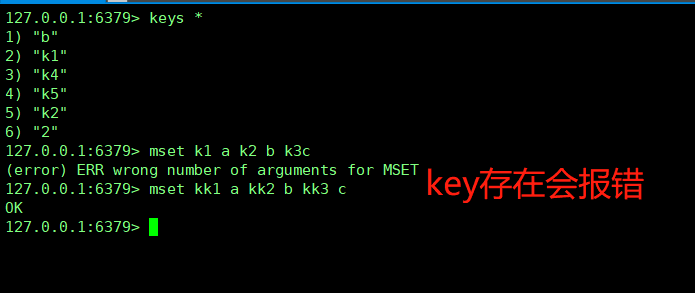


### mset 同时添加多个KV

mset => more set 可以同时添加多个key value到string中

语法格式

mset key1 value1 key2 value2 key3 value3 …



### mget 同时获取多个key

同时获取多个key的值

语法格式

mget key1 key2 key3…

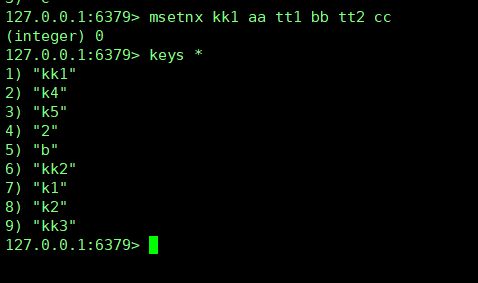


### msetnx 不存在则添加

同时设置多个KV 但设置的key不能存在 如果部分key存在 部分不存在 则不抛出异常 但也不会添加到key中(所有都不会被添加到key)

语法格式

msetnx key1 value1 key2 value2…



## list

list中存放的数据是单值 多value

多个value之间使用空格隔开(list,set,zset通用的模式)

### 性能

list数据类型是一个字符串链表 左右都可以操作数据

list中的数据是有序且可以重复的(按照添加的顺序读取)

无论是操作头或尾效率都很高，但操作中间数据效率就较低了

常用方法

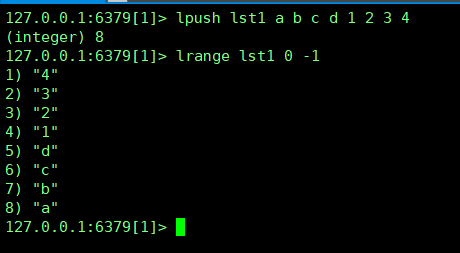
### lpush 左侧开始添加元素

在左侧开始添加内容到list集合中

**先进后出原则** 先添加的数据最后弹出

语法格式

lpush keyName value1 value2 value3 …



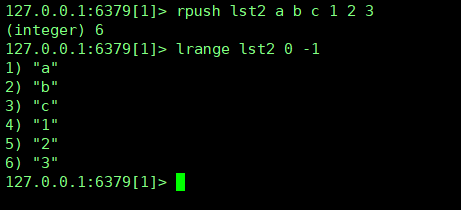
### rpush 右侧开始添加元素

在右侧向list集合中添加元素

**先进先出的原则**

语法格式

rpush keyName value1 value2 value3 …



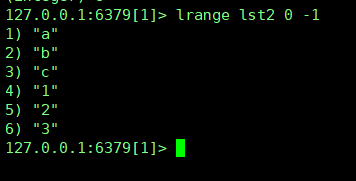
### lrange 遍历list 带范围

遍历list集合 0到-1表示全部获取

范围指的是数据的下标索引值的范围

语法格式

lrange keyName start stop



注意 没有rrange的

### lpop 左侧第一个元素弹出

list顶部元素弹出一个

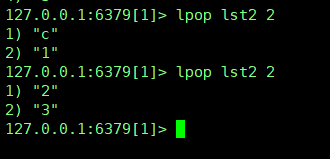
语法格式

lpop keyName



* 扩展

lpop keyName count 每次弹出指定数量的元素



**如果元素都被弹出后 那么key也就不存在了**

### rpop 右侧第一个元素弹出

list底部元素弹出一个

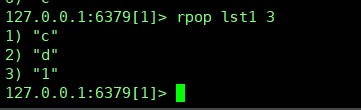
语法格式

rpop keyName



* 扩展

ropo keyName count 指定弹出元素的个数

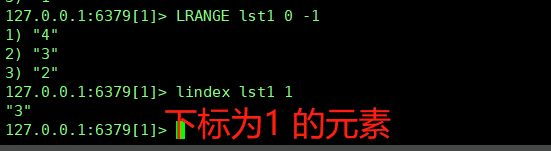


### lindex 获取下标对应的元素

查看下标对应的元素

语法格式

lindex keyName index

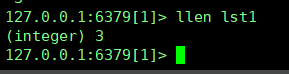


### llen 获取元素个数

获取list元素个数

语法格式

llen keyName



### lrem 指定删除多个元素

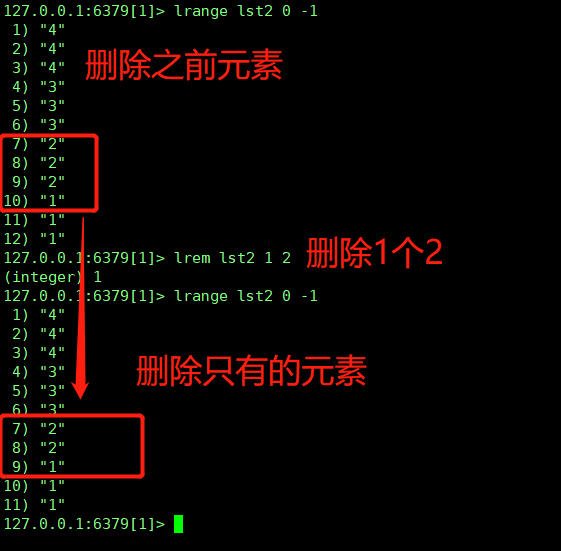
删除N个元素

语法格式

lrem keyName count element

count是要删除的元素个数

element是要进行删除的元素



如果要删除的元素个数大于原有的元素个数 则会全部删除



### ltrim 截取一部分覆盖到当前list中

从当前key中截取指定的元素 然后重新赋值给当前的key

语法格式

ltrim keyName start stop



### rpoplpush 移动一个元素到另个list

弹出一个元素到另一个list中

语法格式

rpoplpush key1 key2

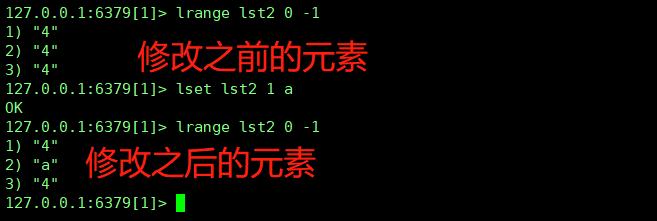


### lset 修改指定下标的值

修改指定下标的值

语法格式

lset keyName index value



### linsert 插入值

在某个值之前或之后插入元素

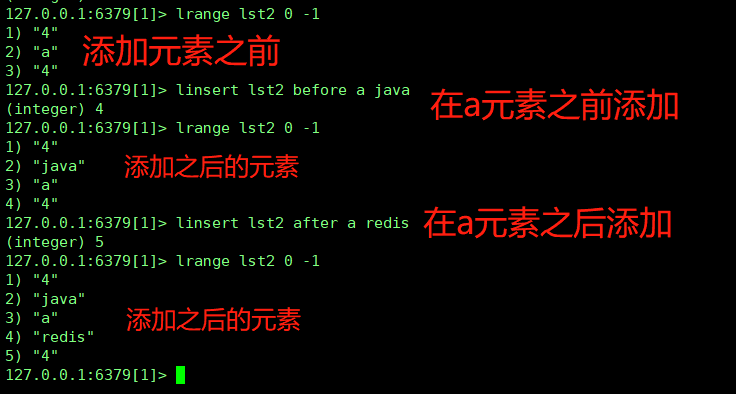
语法格式

linsert keyName before|after pivot value

before|after 前或后

pivot 指代的是元素中的某个值

value 要插入的元素



## set

set类型是单值 多value

set中的值是无序且不重复的

常用方法

### sadd 添加元素

添加元素到set中

语法格式

sadd keyName value1 value2 value3 …

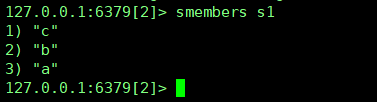


### smembers 遍历元素

遍历set集合 注意 不需要指定范围

语法格式

smembers keyName

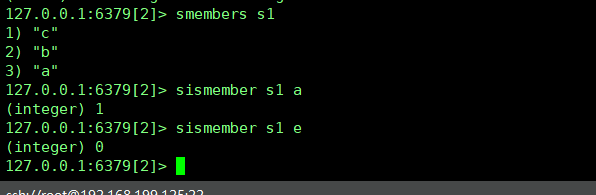


### sismember 判断是否存在某个元素

判断是否存在某个值 存在返回1 否则0

语法格式

sismember keyName element

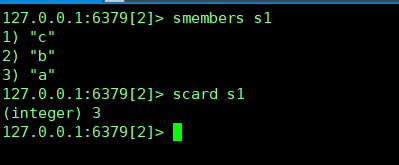


### scard 获取元素个数

获取set集合中的元素个数

语法格式

scard keyName

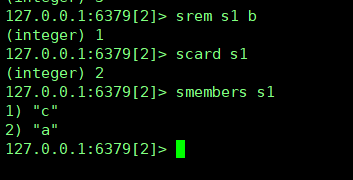


### srem 删除指定的元素

删除指定的元素 注意 是**根据value进行删除**的

语法格式

srem keyName deltetElement



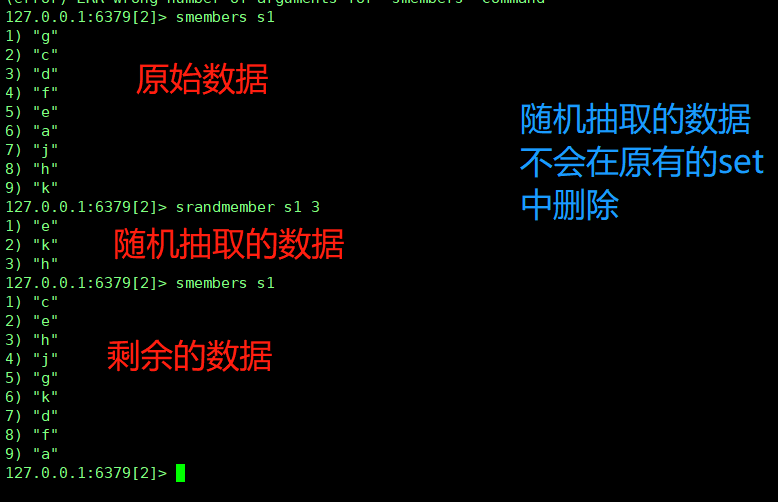
### srandmember 随机抽取元素

随机抽取指定个数的元素

注意 不会在原来的set集合中删除抽取出来的元素

语法格式

srandmember keyName count



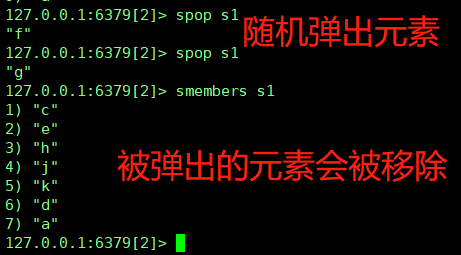
### spop 随机弹出元素

随机出栈

元素会随机的弹出一个 注意 **弹出的元素会在set集合中被移除**

语法格式

spop keyName



* 扩展

spop keyName count

count是指定每次随机弹出的个数

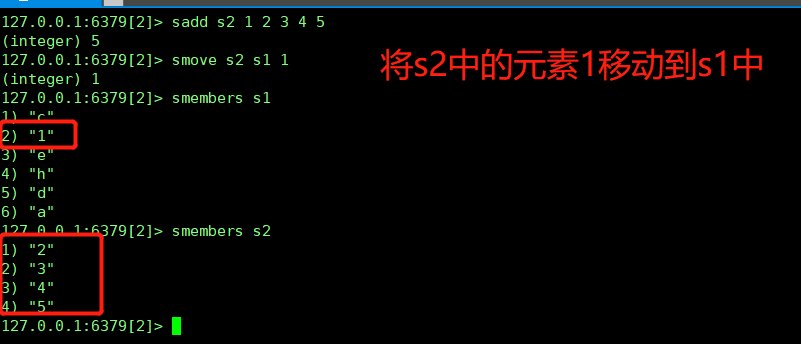


### smove 移动元素到另个set集合

将一个set中指定的值移动到另个set中

语法格式

smove key1 key2 key1Value

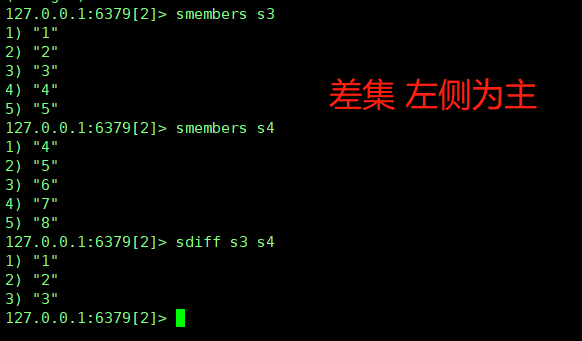


### sdiff 差集

取两个set集合的差集 左侧集合为主

语法格式

sdiff key1 key2

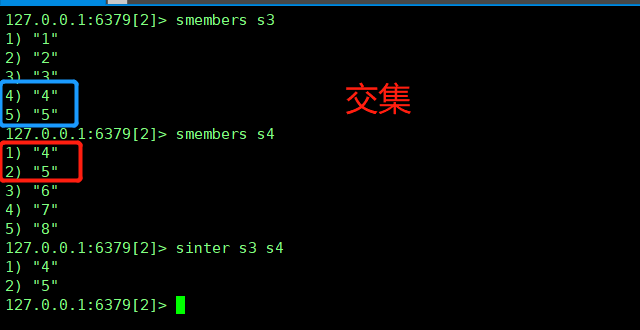


### sinter 交集

交集 取两个set集合相交部分的数据

语法格式

sinter key1 key2



### sunion 全集

全集 合并两个set集合



## zset

排序的set,我们都知道set类型是无序且唯一的 zset是在set的基础上增加了一个score用于排序

目的就是用于对set集合进行排序

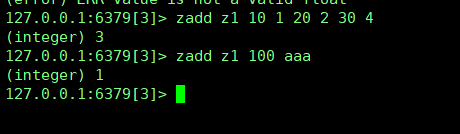
常用方法

### zadd 添加元素到set

添加元素到zset中 需要注意的是**zadd时 score只能是数字**

语法格式

zadd keyName score1 value1 score2 value2 …

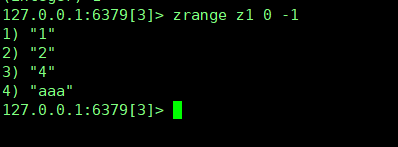


### zrange 遍历for

遍历zset 带范围 0到-1表示全部

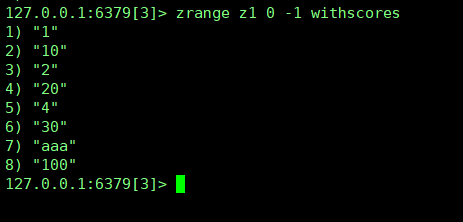
语法格式

zset keyName start stop



* 扩展

zset keyName start stop withscores 连同score一并获取到

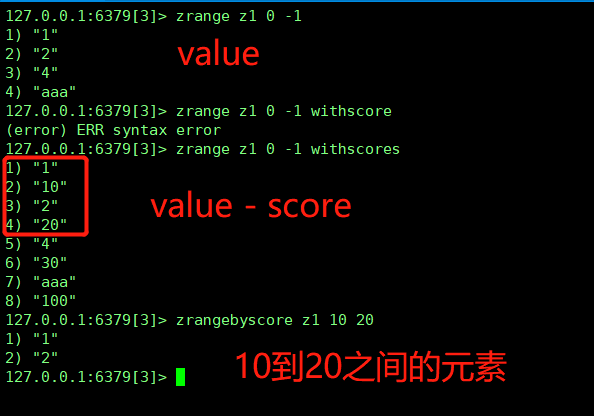


### zrangebyscore 根据score遍历set

根据score范围进行查找元素

语法格式

zrangebyscore keyName startSocre stopScore



### zrem 根据value删除

删除元素 根据value进行删除

语法格式

zrem keyName deleteValue

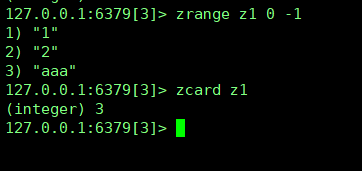


### zcard 获取元素个数

获取元素个数

语法格式

zcard keyName



### zcount 根据score统计元素个数

根据score统计元素个数

语法格式

zcount keyName startScore stopScore



### zrevrank 获取元素下标

根据值获取元素的下标索引

语法格式

zrevrank keyName value



### zscore 根据value获取score

根据value值获取score

语法格式

zscore keyName value



### zrevrange 倒序遍历

倒序遍历zset

语法格式

zrevrange keyName start end



### zrevrangebyscore 根据score倒序遍历

根据score进行倒序遍历

语法格式

zrevrangebyscore **endScore startScore**

注意 **先结束的score 再开始的score**



## hash

KV模式不变 但V存放的是一个键值对

类似Map<String,Map<String,Object>>

常用方法

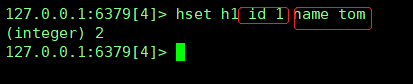
### hset 添加到hash

添加元素到hash集合中

新增的时候value可以多个同时传入

语法格式

hset keyName key1 value1 key2 value2 …

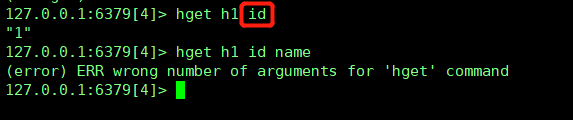


### hget 获取hash中的值

获取hash集合中的元素 根据value的key 只能单个的获取

语法格式

hget keyName key



### hmset 同时设置多个值

跟hset相同 同时添加多个值到hash中

语法格式

hmset keyName key1 value1 key2 value2…

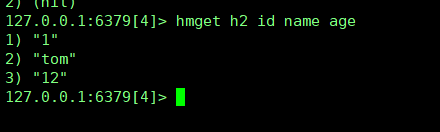


### hmget 同时获取多个值

同时获取多个value

语法格式

hmget keyName key1 key2 …

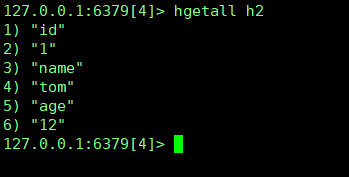


### hgetall 获取全部元素key 和value

获取当前hash中全部数据 k-v模式

语法格式

hgetall keyName



### hdel 根据key删除hash中的元素

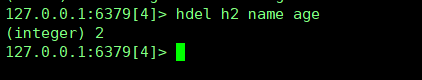
删除hash集合

需要注意 **不能直接删除整个hash集合中的数据** 需要根据key进行删除

语法格式

hdel keyName key1 key2 …



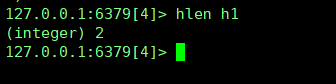


### hlen 获取hash中元素个数

获取hash集合中元素个数

语法格式

hlen keyName

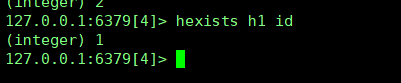


### hexists 判断是否存在

判断hash中是否存在某个key

语法格式

hexists keyName key

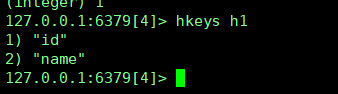


### hkeys 获取全部的key

获取hash中全部的key

语法格式

hkeys keyName

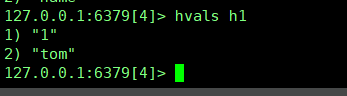


### hvals 获取全部的value

获取hash中全部的value

语法格式

hvals keyName

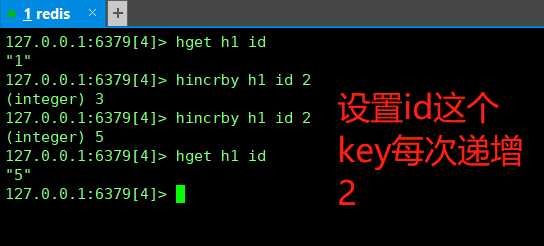


### hincrby 整数递增

给value中的key指定递增 可以指定递增数量 只能用于数字上边

语法格式

hincrbt keyName key imcrement

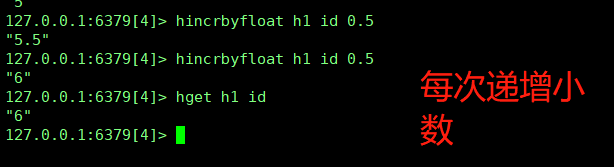


### hincrbyfloat 小数递增

给hash中的value的key设置每次递增小数

语法格式

hincrbyfloat keyName key 0.5



### hsetnx 不存在添加 存在舍弃

添加时判断value中的key是否存在 不存在则添加 存在则不添加

这样做的目的就可以有效避免key相同 值被覆盖的情况

语法格式

hsetnx keyName key value



# 配置文件

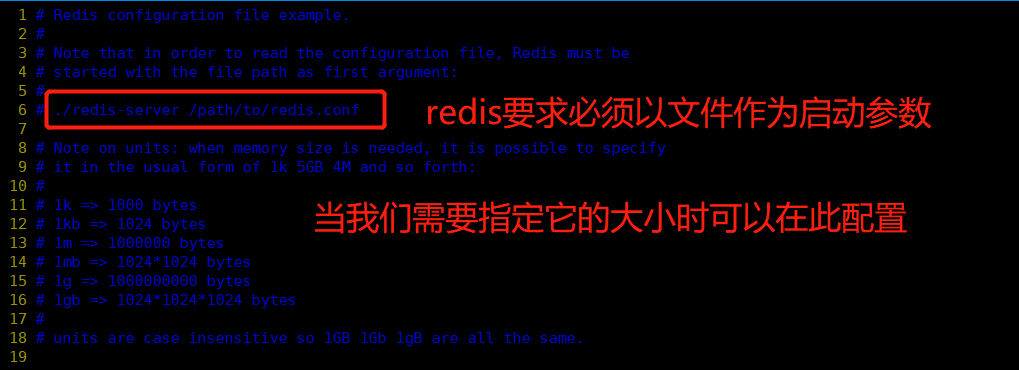
## 配置文件所在位置

redis配置文件的位置在redis的安装目录下(即 执行make指令的目录)会有一个redis.conf的文件 该文件就是redis的配置文件

## redis的单位配置

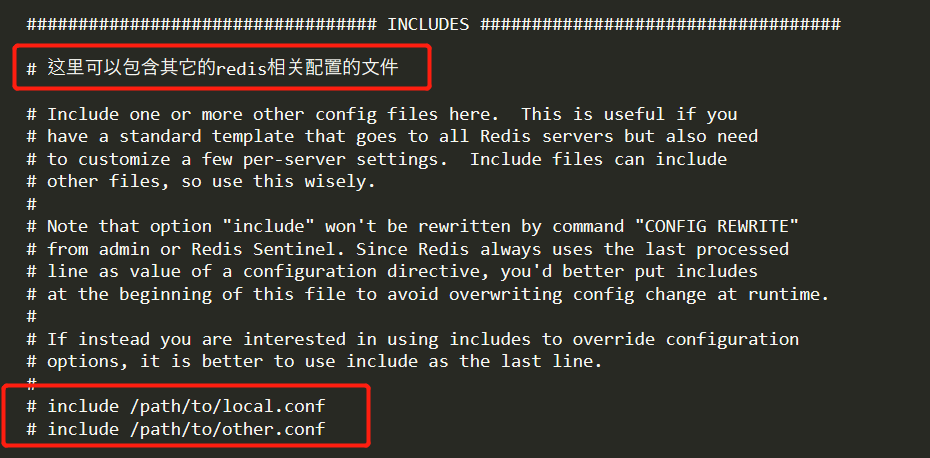
用来指定redis的内存大小 默认与系统内存相同

对于内存单位大小的配置 仅支持bytes 不支持bit 另外对大小写是不敏感的 即 1gb 与 1GB是相同的



## INCLUDES 包含配置

可以通过includes将其它redis相关的配置文件包含进来

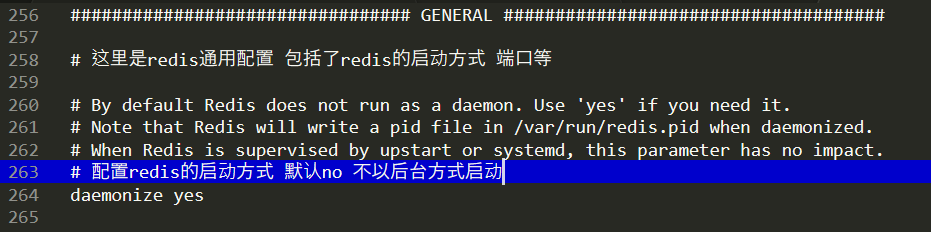


## GENERAL 通用配置

general是redis的通用配置 包括了redis的启动方式，端口号配置等

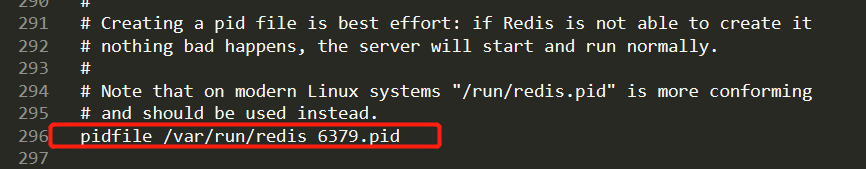
### daemonize

配置启动方式 默认no 关闭窗口时程序也会随之关闭



### pidfile

redis的进程文件路径 默认在/var/run/redis\_6379.pid



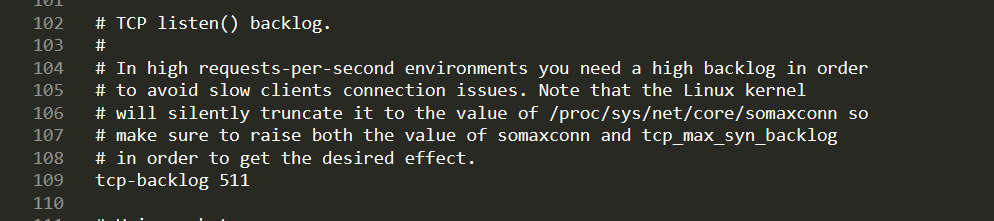
### tcp-backlog

**6.0版本迁移到network配置选项中了**

用来配置在高并发环境下减少慢客户端连接等待时间 需要通过计算backlog队列总和

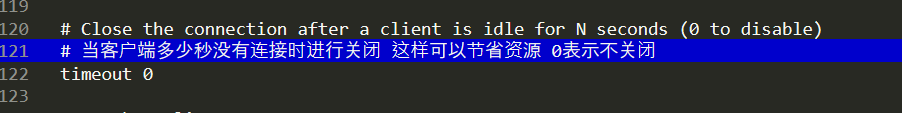
backlog总和=未完成的三次握手队列+已完成的三次握手队列

Linux内核会将这个值减小到/proc/sys/net/core/somaxconn值 我们在学习阶段是足够使用的 但生产环境下如果涉及到高并发需要进行修改



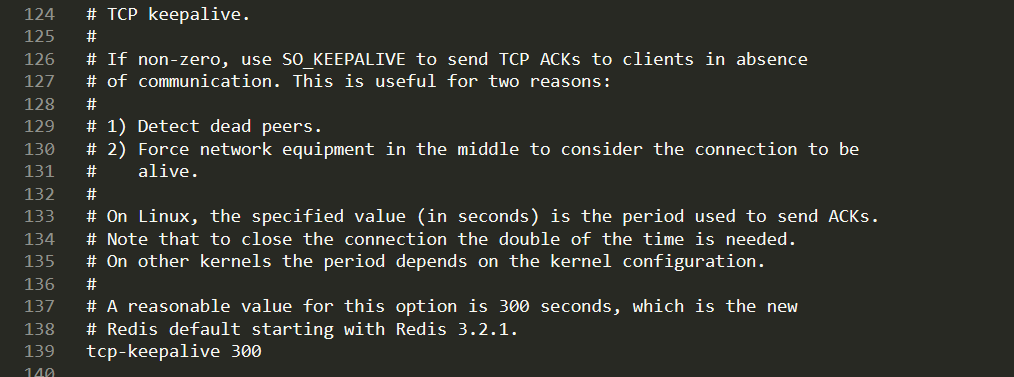
### timeout

配置客户端连接空闲关闭等待时间



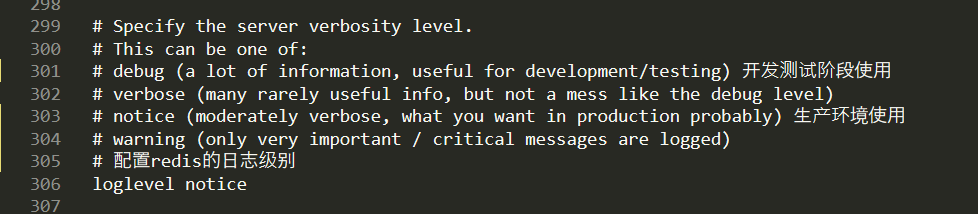
### tcp-keepalive

定时检测连接的存活状态 默认300秒检测一次



### loglevel

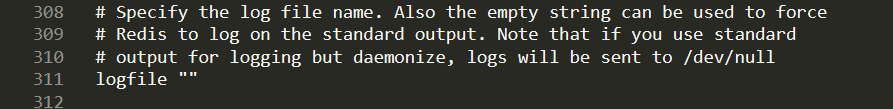
配置redis的日志级别



### logfile

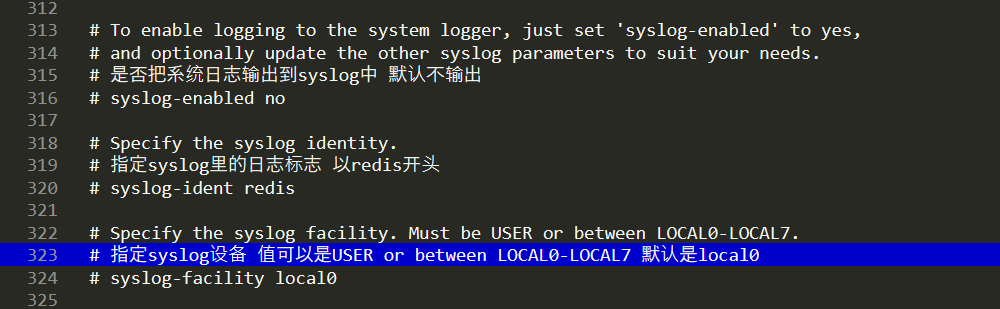
配置日志输出的文件

默认为空 表示控制台输出



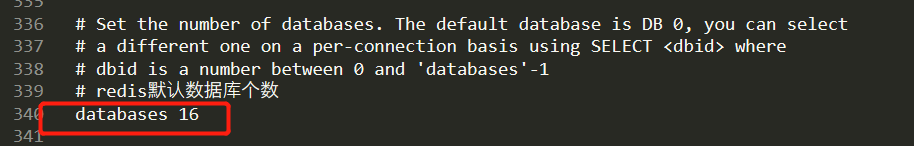
### syslog

系统日志相关配置



### databases

配置redis数据库个数 默认是16个



## SNAPSHOTTING 快照

### save

是用来配置RDB(持久化)的

提供了三种模式 可以配合使用

save 900 1 900秒内有一次更新操作

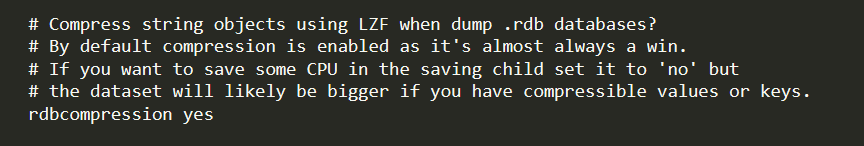
save 300 10 300秒内有10次更新操作

save 60 10000 60秒内有10000次更新操作

如果不想等到时间到了在进行备份操作 可以手动的通过save进行备份操作 指令就是save

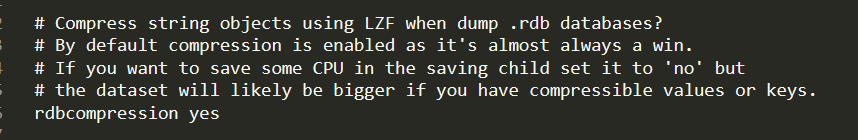
### stop-writes-on-bgsave-error

表示出现异常时是否立即备份 改为no 表示当redis出现异常时不启用备份不关心数据是否一致 默认是yes 但redis出现异常时不会进行备份 redis不会把异常的数据备份到rdb中



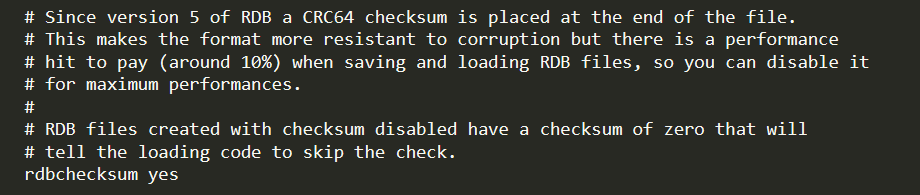
### rdbcompression

是否启用LZF压缩rdb文件 如果考虑到CPU消耗的话 可以关闭 但RDB文件会变得很大



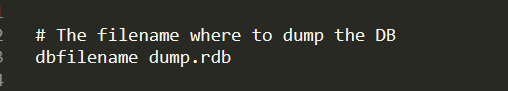
### rdbchecksum

在存储快照后是否进行CRC64算法对数据校验 如果进行数据校验会增加大约10%的性能消耗 考虑到性能可以关闭



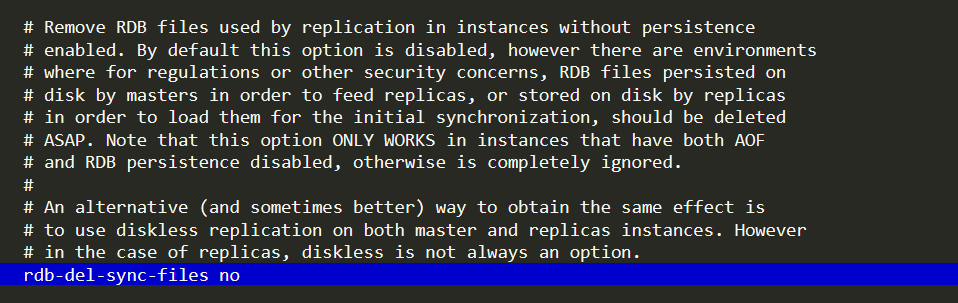
### dbfilename

指定要备份的rdb文件名称



### rdb-del-sync-files

配置是否删除同步锁 默认no



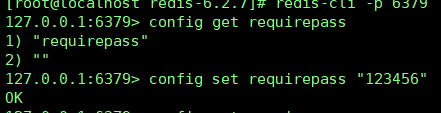
## REPLICATION 复制

## SECURITY 安全

redis默认是不需要使用密码来访问的，但遇到特殊情况 可以设置访问密码

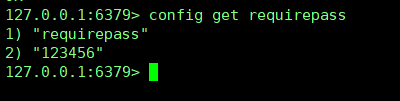
### 设置密码

config set requirepass "需要设置的密码"



### 查看密码

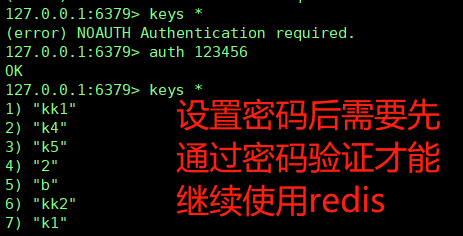
config get requirepass



### 使用密码

auth + 设置的密码

注意 在使用任何redis命令之前执行



### 关闭密码

关闭使用密码 直接改为空即可



## LIMITS限制(缓存过期策略)

### maxclients

最大连接的客户端数量

默认关闭

### maxmemory

配置最大内存 单位bytes

默认关闭

### maxmemory-policy

配置内存过期策略 默认永不过期 noeviction

**LRU算法 最近最少使用**

**LFU算法 最近最不经常使用**

random 随机

volatile-lru 使用最近最少使用算法移除key 只针对设置了过期时间的key (在key没有到期之前 如果符合了LRU算法也会被移除)

allkeys-lru 使用最近最少使用的算法移除所有key

volatile-lfu 使用lfu算法移除key 针对设置了过期时间的健

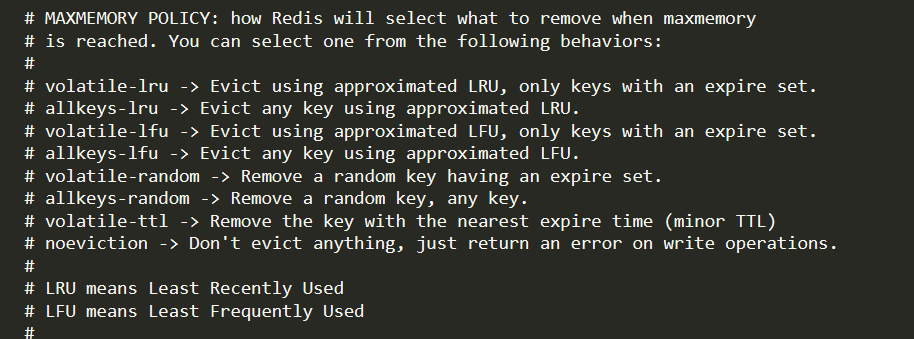
allkeys-lfu 采用lfu算法移除所有key

volatile-random 随机移除过期时间的key

allkeys-random 随机移除所有设置了过期时间的key

volatile-ttl 移除最近要过期时间的key

noeviction 不进行移除 针对写操作 只返回错误信息

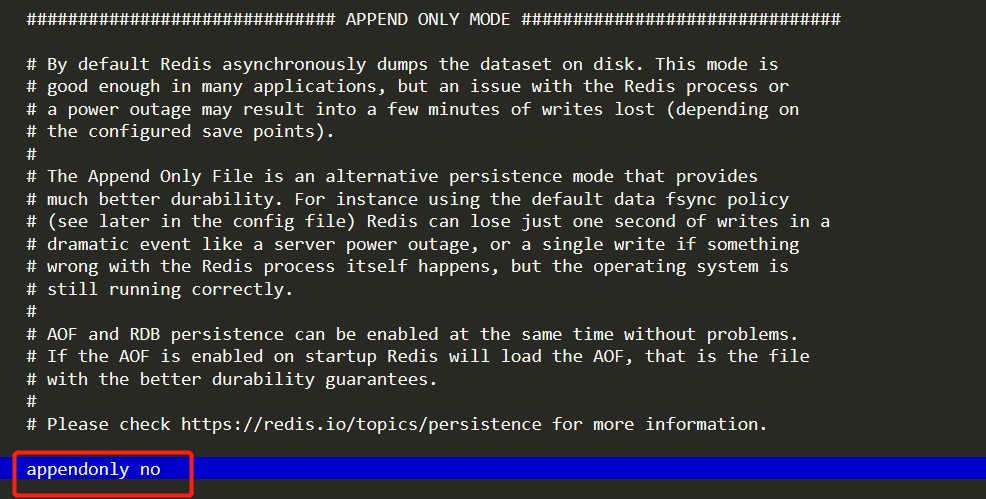


### maxmemory-samples

redis会随机抽取这么多key的样例对其进行LRU或LFU算法 默认是5个

## APPEND ONLY MODE

用来配置aof备份文件相关信息 默认不启用



### appendfilename

配置aof文件名称 默认appendonly.aof

### appendfsync

aof的持久化策略

提供了三种模式

**always** 同步持久化 每次发生数据改变都会立即被记录到磁盘 性能低 但数据完整性较高

**everysec** **默认配置** 异步操作 每秒记录一次 如果宕机会损失一秒钟的数据

**no** 不启用

### no-appendfsync-on-rewrite

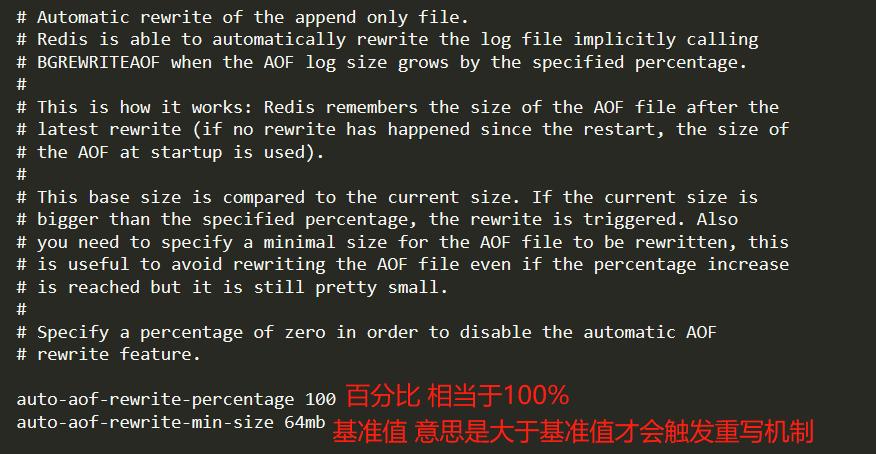
重写aof文件时是否可以运用appendfsync 默认no 不使用 保证数据安全

### auto-aof-rewrite-min-size

设置重写的基准值

### auto-aof-rewrite-precentage

设置重写的百分比



## 总结

redis.conf配置项如下：

1. redis默认不是以守护进程启动 可以通过修改**daemonize**使其使用守护进程方式启动
2. 当redis以守护进程启动时 会默认把pid进程文件写到默认的配置路径 通过**pidfile**来指定新的存放路径
3. 默认端口6379 可以通过**port**来指定其它端口
4. 绑定主机地址 通过**bind**修改要绑定的主机地址 默认本机127.0.0.1
5. 当客户端闲置多少事件后关闭连接 0表示关闭该配置 即永不关闭 可通过**timeout**来指定空闲时间
6. 指定redis的日志记录级 默认是verbose  
   redis支持四种级别 debug ,verbose ,notice,warning
7. 日志记录方式 如果设置了守护进程方式启动redis 则需要配置redis的日志记录方式 默认输出到控制台 通过logfile指定日志输出位置
8. databases 设置数据库的数量 默认16个 可以通过select+数据库下标选择不同的数据库
9. 指定在多少时间内对数据有多少次更新操作就将其实例化到文件中 可以多个条件配合  
   save <seconds> <changes>  
   save 900 1 表示在900秒内有一次更新操作  
   save 300 10 表示在300秒内有10次更新操作  
   save 60 1000 表示在60秒内有1000次更新操作

10.指定存储到本地时是否压缩数据，默认yes redis采用LZF压缩 如果为了节省CPU时间 可以关闭该选项 但会导致数据库文件变得很大

11.指定本地数据库文件名 默认dump.rdb  
通过dbfilename指定

12 指定本地数据库的存放目录 默认是当前目录  
dir ./

13.当本机为slave(从机)时 设置master(主机)的ip地址及端口 在redis启动时会自动从master进行同步数据

14.当master设置了密码保护时 需要设置slave连接master的密码 slaveof <masterip><masterport>

15.设置redis连接密码 可以在客户端(redis-cli)中通过config set requirpass设置 也可以在配置文件中设置

配置文件中设置requirepass +密码 默认是关闭的

16.设置同一时间最大客户端的连接数 处理高并发访问

maxclients 128 默认无限制 即10000

17.指定redis最大内存限制，在redis启动时会先把数据加载到内存中 达到最大内存后会先尝试清除已到期或即将到期的key 达到最大内存时将无法进行写入操作 但可以继续读取操作

通过maxmemory进行指定

18.指定是否每次更新操作后记录日志 默认no 通过appendonly修改

19.指定更新日志文件名 默认appendonly.aof 通过appendfilename指定

20.指定更新日志的条件 共有三种

no 表示等操作系统进行数据缓存同步到磁盘时

always 每次更新操作后需要手动调用fsync()将数据写到磁盘(安全但效率慢)

everysec 每秒同步一次(默认 ，折中值)

21.虚拟机是否启用虚拟内存机制 默认no,VM将数据分页存放 有redis将访问量较小的页即冷数据swap到磁盘上，访问多的页由磁盘自动换出到内存中

vm-enabled来配置

22.虚拟机文件路径 注意 不可以多个redis实例共享

vm-swap-file /tmp/redis.swap

23.指定包含其它配置文件 可以在同一主机上多个redis实例之间使用同一份配置文件 而同时各个实例又拥有自己的特定配置文件

include来指定

# 持久化

redis持久化分两种 RDB和AOF

## RDB

### 是什么

Redis Database

RDB 指的是在指定的时间间隔内将内存中的数据快照写入到磁盘 也就是我们常说的snapshot快照。

redis会单独创建(fork)一个子进程来进行持久化，先将数据写入到一个临时文件中 带持久化的过程都结束了在用这个临时文件替换上次持久化好的文件 整个过程中 主进程不参与任何IO操作，这就确保了极高的性能，如果需要大规模的数据恢复且对数据的完整性要求不高的话那么RDB方式比AOF更加高效 RDB的缺点是可能会丢失最后一次持久化的数据。

### Fork

fork的作用是复制一个与当前一样的进程来进行持久化操作，新进程的所有数据都跟原进程完全一致 但是是一个新的进程并作为原进程的子进程 这个进程主要负责RDB的持久化操作。

### RDB保存的文件

RDB保存的文件是dump.rdb

可以通过dir指定文件保存路径

通过dbfilename指定rdb文件名称 默认是dump.rdb

### RDB的相关配置

可以参照[SNAPSHOTTING快照](#_SNAPSHOTTING_快照)

save时只管保存 不管其它 全部阻塞

bgsave后台异步的进行保存 同时不会影响客户端的请求和响应 可以通过lastsave获取最后一次执行成功的快照时间

执行save或flushall ,flushdb,shutdown时都会触发立即备份rdb flushall ,flushdb触发的备份都是空 无意义

### 优缺点

优点

适合大规模的数据恢复，对数据的完整性和一致性要求不高

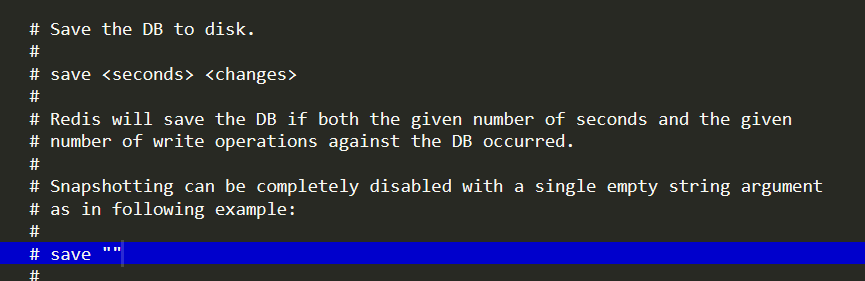
缺点

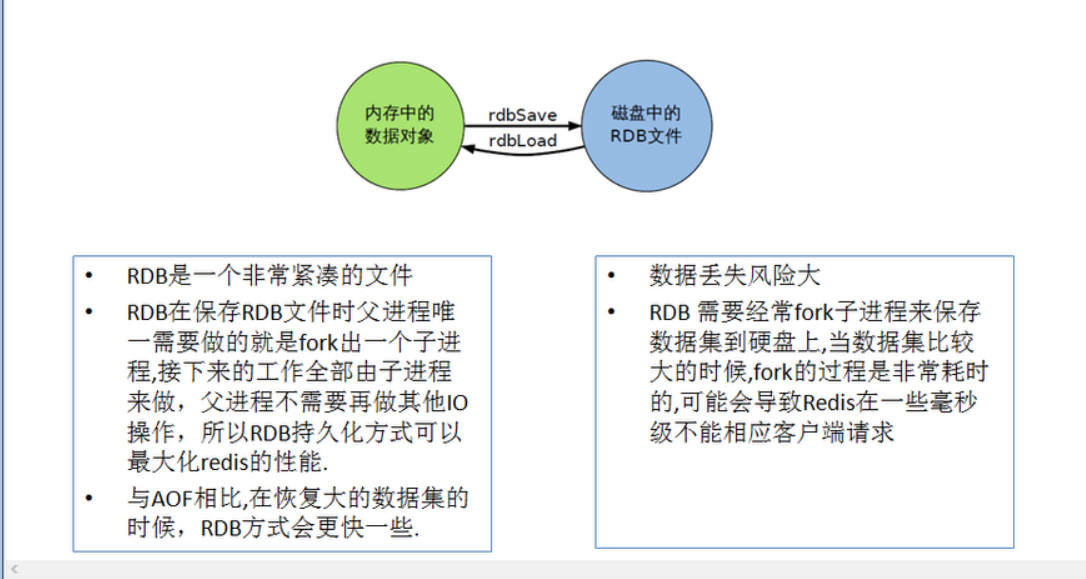
rdb采用的是在一定时间内做一次备份 这样就有可能出现丢失最后一次数据的备份情况，另外fork子进程的时候 内存中的数据被clone了一份 相当于两倍的数据膨胀

### 停止rdb备份

通常不建议停用rdb备份

可以在配置文件中修改save备份模式为空即可





## AOF

### 是什么

AOF => Append Only File

AOF是以日志形式记录每个写操作，将redis所有的写的操作指令记录下只许追加文件 但不允许改写文件 redis启动时会读取该文件重新构建数据（需要在配置文件中打开aof的配置）

### 配置文件位置

redis.conf文件

修改内容参考[APPEND ONLY MODE](#_APPEND_ONLY_MODE)

### AOF启动/修复/恢复

均在redis.conf中的APPEND ONLY MODE节点下修改

#### AOF启动

配置文件中 appendonly no 改为yes

#### AOF修复/恢复

当redis非正常退出时 需要将备份的aof文件复制到对应的启动目录中(config get dir 查看当前启动的目录 redis-cli中执行)

AOF在启动时会自动加载该文件

#### AOF异常恢复

当AOF文件本身出现异常 那么程序是无法启动的 无论采用aof的那种策略 都不可避免数据丢失 当机器宕机刹那间 可能对于aof文件中的记录操作只进行了一部分 因此也就可能导致aof文件中内容记录不完整 从而导致文件异常，这种情况在redis启动时加载aof文件会报错 导致redis无法启动

修复方式

在usr/local/bin下有redis-ckeck-aof修复向 直接执行即可

redis-check-aof --fix appendonly.aof

修复RDB同理 都在usr/local/bin下存在对应的修复工具

另外：redis启动时会先去加载aof文件

### rewrite

aof采用文件追加的方式 ，那么文件就会越来越大 为了避免出现此种情况 aof新增了重写机制 当aof的文件大小达到设定的阙值时 redis就会启动aof的内容压缩 只保留可以恢复数据的最小指令集

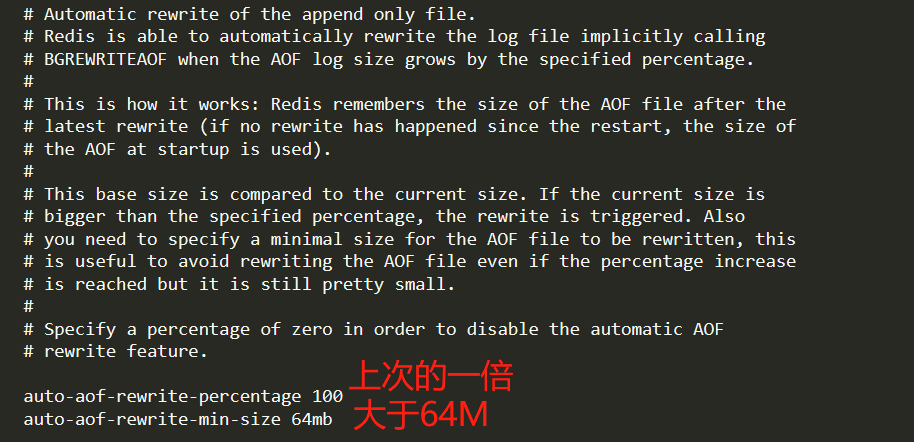
#### 重写原理

aof文件会因持续增加而过大时会fork新进程将文件进行重写(先写临时文件最后在rename) 然后遍历新进程中的数据 每条记录有一条set，重写aof文件的操作不会读取旧的aof文件 而是将整个内存中的数据用命令的方式重新写了一个aof文件 与快照类似

#### 触发机制

redis会记录上次重写aof时的大小，默认配置是当aof文件大小是上次重写大小的一倍且文件大于64M时触发重写机制

我们可以通过指定文件大小的阙值来触发重写机制



### 优缺点

优势：

每秒同步： (appendfsync:always) 同步持久化 每次发生数据改变都会立即记录到磁盘 性能低但数据安全性高

每修改同步(appendfsync:everysec) 异步操作 每秒同步一次 如果宕机 可能会丢失一秒内的数据

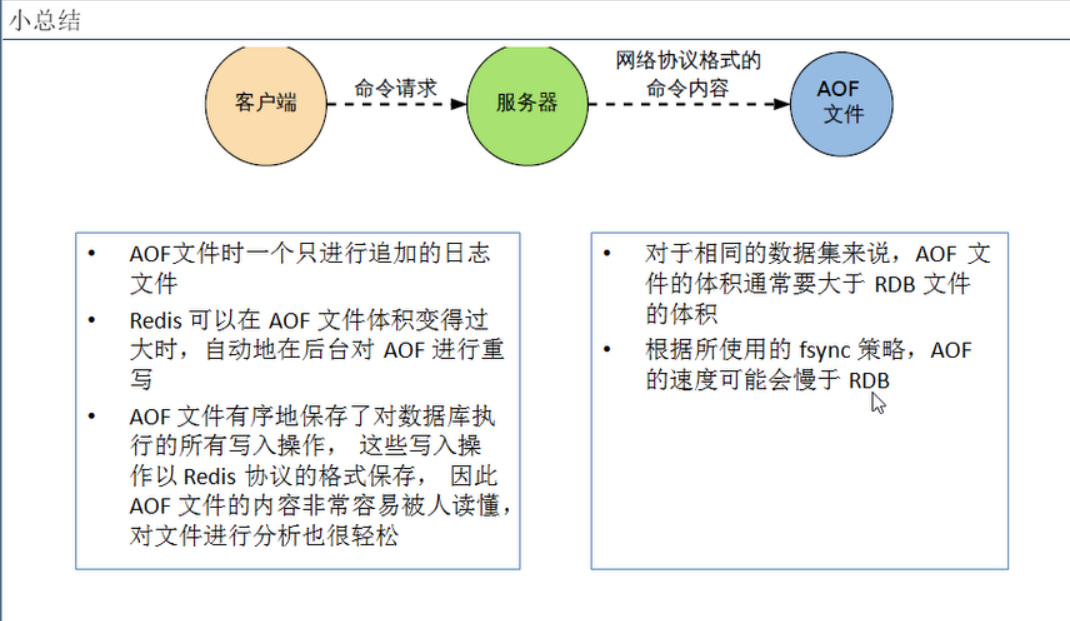
不同步(appendfsync:no) 从不同步

劣势：

相同的数据集 aof的文件要远大于rdb文件 恢复速度也会慢于rdb

aof的运行效率 实时记录慢于rdb,每秒同步略低于rdb ，不同步与rdb效率相同 但不同步就不需要启用aof功能了

### 总结



## 持久化总结

RDB持久化方式能够在指定时间间隔内对数据进行快照存储

AOF持久化是采用日志记录方式来记录每次对数据写操作

当redis重启时会先加载aof文件将数据恢复到内存中

redis对aof文件设置了重写机制 避免导致aof文件过于庞大

应用场景：

RDB适合大数据恢复并且对于数据完整性要求不高的情况下

AOF则满足对于数据要求略高的情况下进行备份恢复

两者同时存在：RDB和AOF同时存在redis会先加载AOF 当AOF文件异常时 redis无法启动成功 可以通过redis-check-aof --fix appendonly.aof方式进行对aof文件的修复

性能建议：

因为RDB文件只用作备份用途 建议旨在slave上持久化RDB 一般15分钟一次备份就可以了即保留save 900 1这项配置即可

如果开启AOF 好处是在恶劣的情况下最多只会丢失两秒内的数据，redis启动时只加载自己的aof文件就可以了，代价是 1带来了持续的IO，2 AOF的重写机制最后在重写过程中产生的新数据写道新文件造成的阻塞几乎是不可避免的，只要硬盘允许应该尽量减少AOF的重写频率，AOF重写的基准大小默认是64M 可以设置到3G以上 默认超过大小100%(1倍)时触发重写机制

如果不开启AOF 可以节省很多性能开支 可以使用主从复制方式实现reids的备份

# 事务

## 概述

一个队列中 一次性的 顺序的排他性的执行一系列命令

redis的事务本质就是一组命令的集合，一个事务中的所有命令都会被序列化，按顺序串行的执行而不会被其它命令插入。

换言之 redis的事务就是批处理操作。

## 常用命令

discard 放弃当前事务

exec 执行当前事务

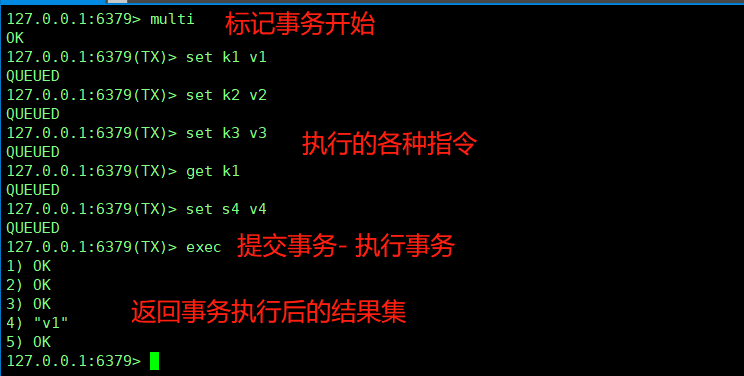
multi 标记事务的开始

watch 监控一个或多个事务

unwatch 取消监控事务

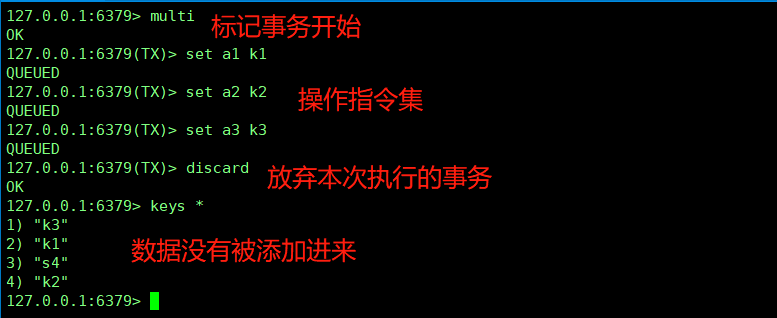
### 正常执行

先使用multi标记事务，然后执行各种指令 最后exec提交事务



### 放弃事务

先multi标记事务 在执行各种指令集 最后执行distcard放弃本次事务



### 全体连坐

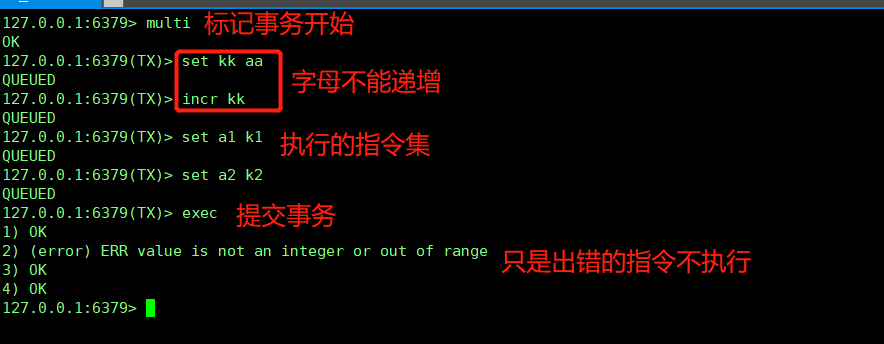
一组指令中出现错误的指令时本次事务会全部执行失败

在标记事务后 执行事务之前(exec) 如果指令集中出现错误或异常 那么这种情况exec会执行失败 指令集中的指令也不会被执行了



### 冤头债主

指令集在加入队列时没有出错 在执行事务时才发现出错 类似运行时异常 此时哪个指令出的错 那么那条指令不会被执行 但不会牵连到其它指令的执行



### watch监控

如果事务提交之前 监控的key被其它线程改动了 那么事务将被打断

#### 悲观锁

类似mysql中的表锁

操作之前先上锁 预防其它线程把当前要操作的数据修改了导致数据不一致，此方式极高的保证了数据的安全 但效率也是最低的 因为当你在操作的时候其它线程只能排队等待

#### 乐观锁

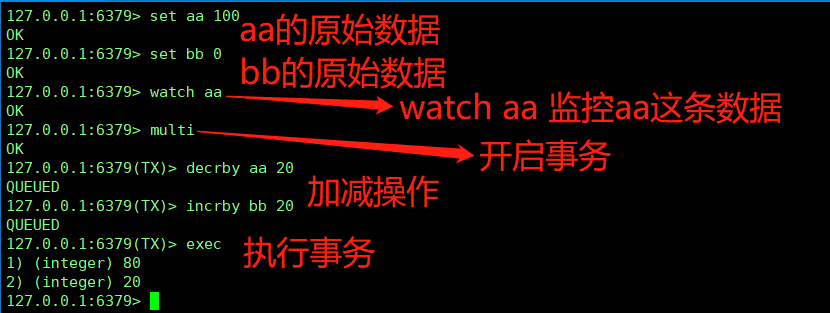
类似mysql中的行锁

在每条数据上添加一个隐士的version 当有多个线程同时修改了同一条数据时才会抛出异常 如果非同时修改的则不会抛出异常(雷同于git的版本冲突)

乐观锁的策略：提交数据时版本必须大于当前版本才能执行更新操作

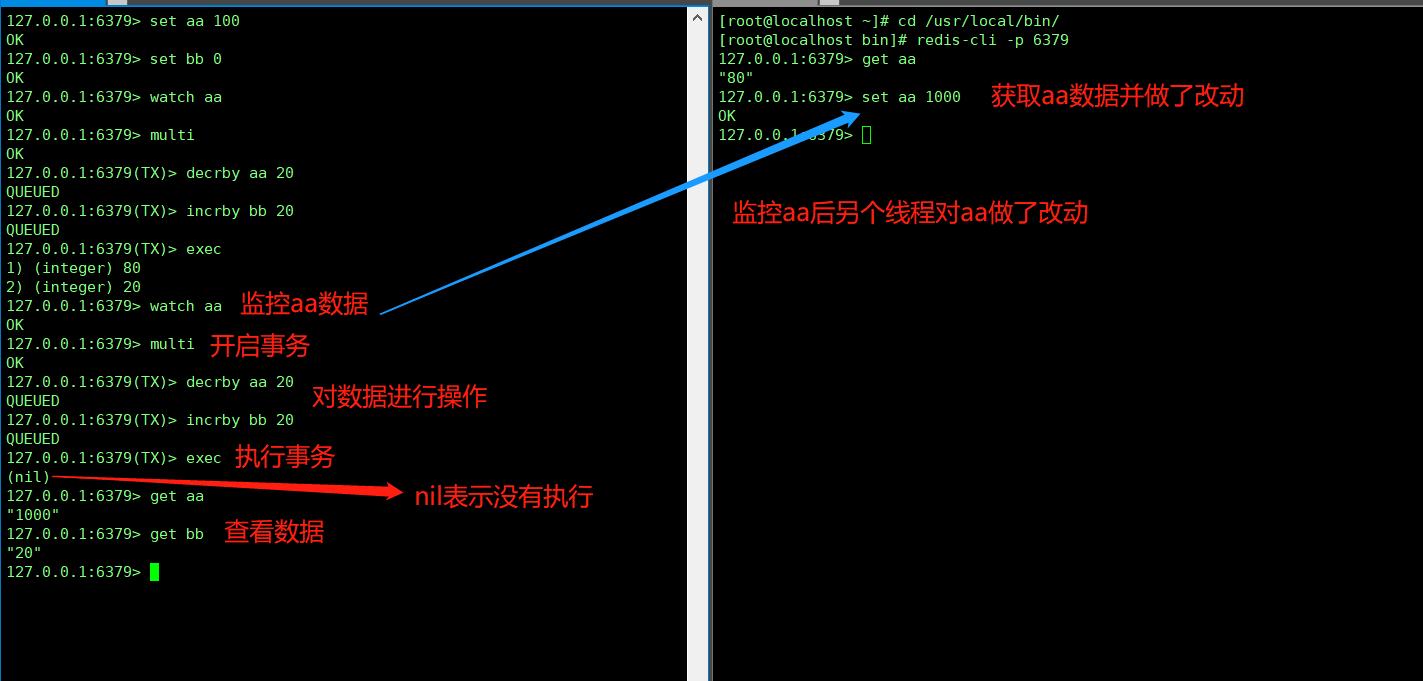
#### 案例1

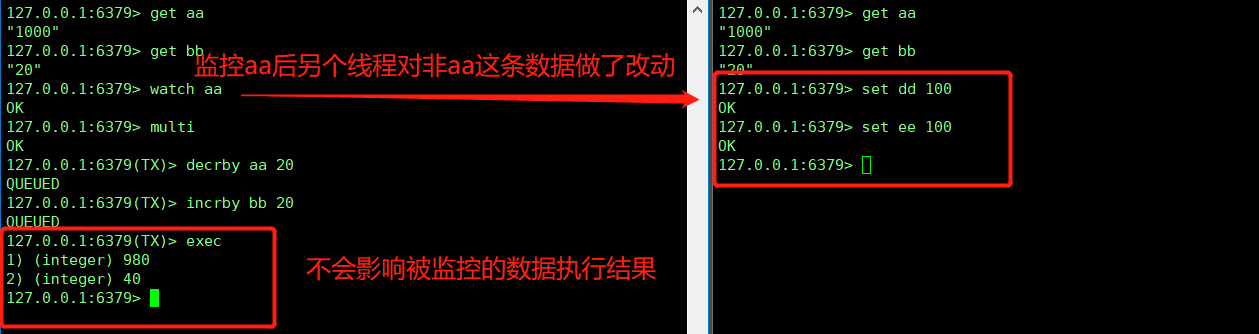
两条数据 一条增加 一条减少 在同一个事务中 执行事务之前没有其它线程改动这两条数据中的任意一条



#### 案例2

如果被监控的数据在提交事务之前做了改动 则提交成功但监控的数据不会被执行





#### watch总结

watch命令类似乐观锁 事务提交时 如果key的值已经被别的线程改动过了 整个事务队列不会被执行 同时会返回Nullmutli-bulk通知调用者事务执行失败

### unwatch

放弃当前的数据监控

注意 放弃监控时是对所有监控的key都放弃监控

## 事务的三个阶段

开启事务 multi

入队 将多个执行命令加入到事务中 接收到这些命令后不会被立即执行 而是等到exec提交事务时才会去执行

执行 由exec触发事务执行

## 事务的特性

redis事务的特性大致分为三个

1. 单独的隔离操作：事务中的所有命令都会先保存到队列中(序列化) 然后按顺序执行 在事务执行过程中不会被其它线程发送的命令打扰
2. redis的事务没有隔离级别的概念，队列中的命令在没有提交事务之前都不会实际被执行。
3. redis的事务不保证数据的原子性，在同一个事务中 如果有一条指令执行失败 其它的指令仍然会被执行 没有回滚。

# 消息订阅发布

## 概述

redis的发布订阅是进程间的一种消息模式，由发送者(pub)发送消息，订阅者(sub)接收消息

## 命令



## 案例

发布消息 publish a1 xxxx

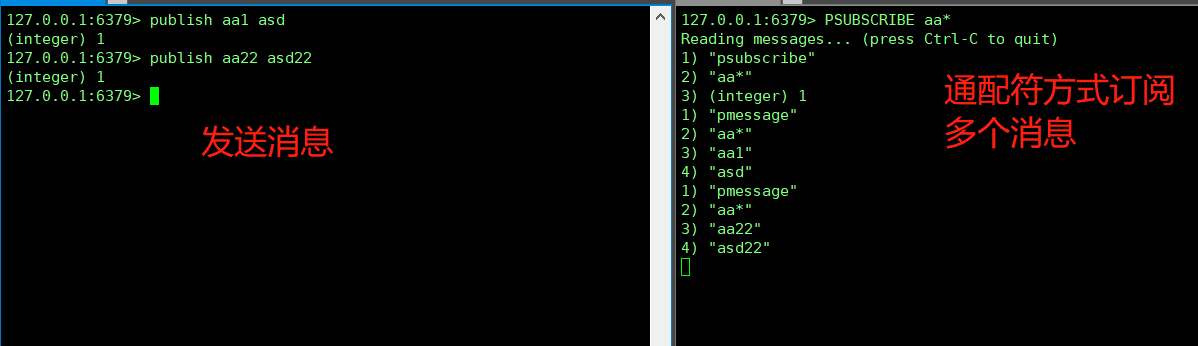
一次性订阅多个消息 subscribe a1 a2 a3



通配符方式接收订阅

接收端 psubscribe 订阅的频道 \*表示任意字符 例如aa\* 表示以aa开头的都被订阅

发布者 publish aa1 msg1



# 主从复制

## 概述

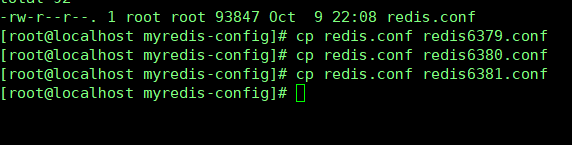
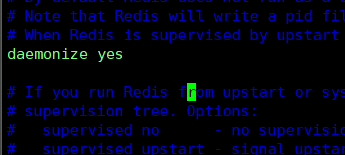
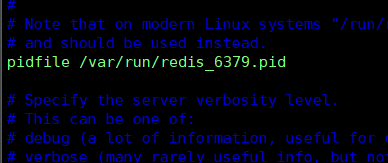
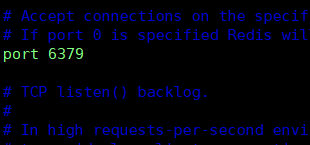
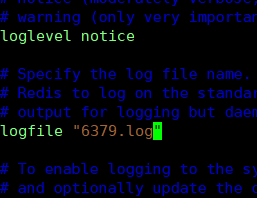
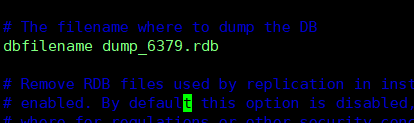
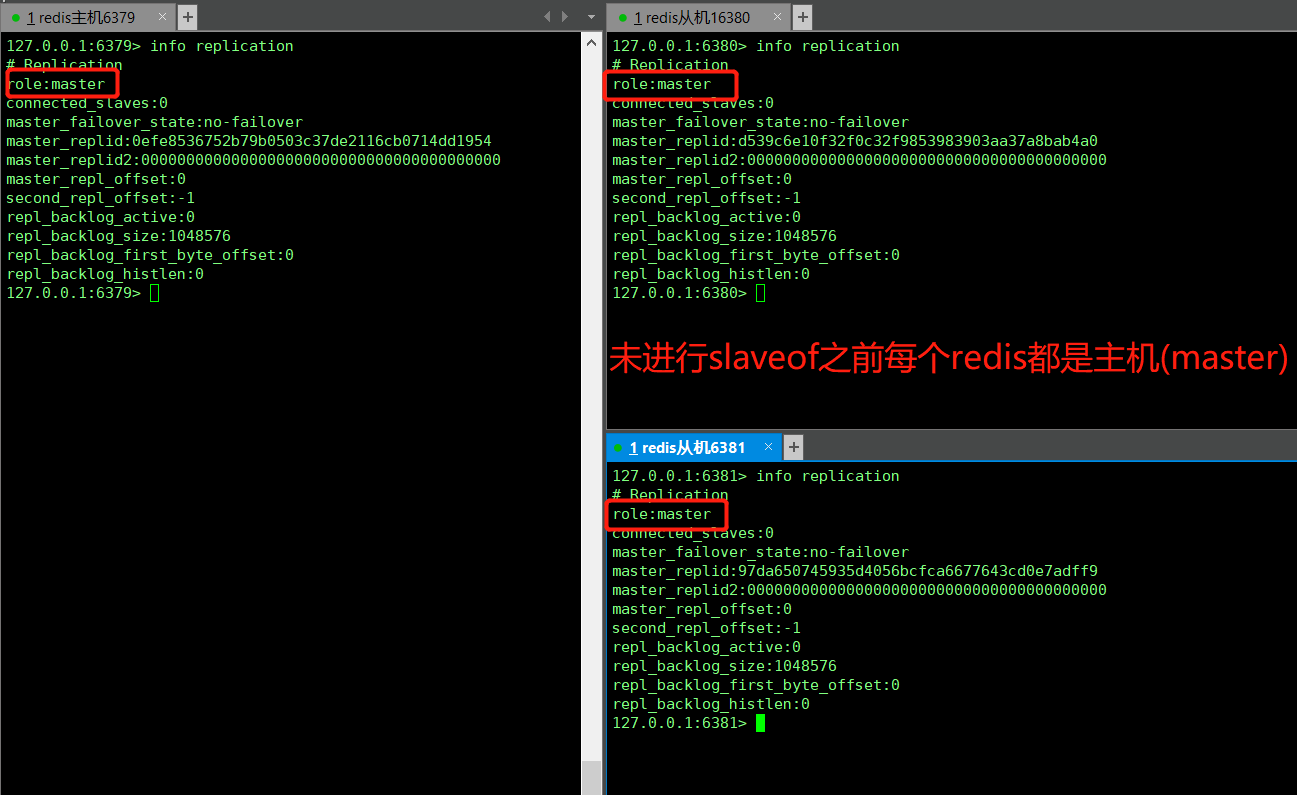
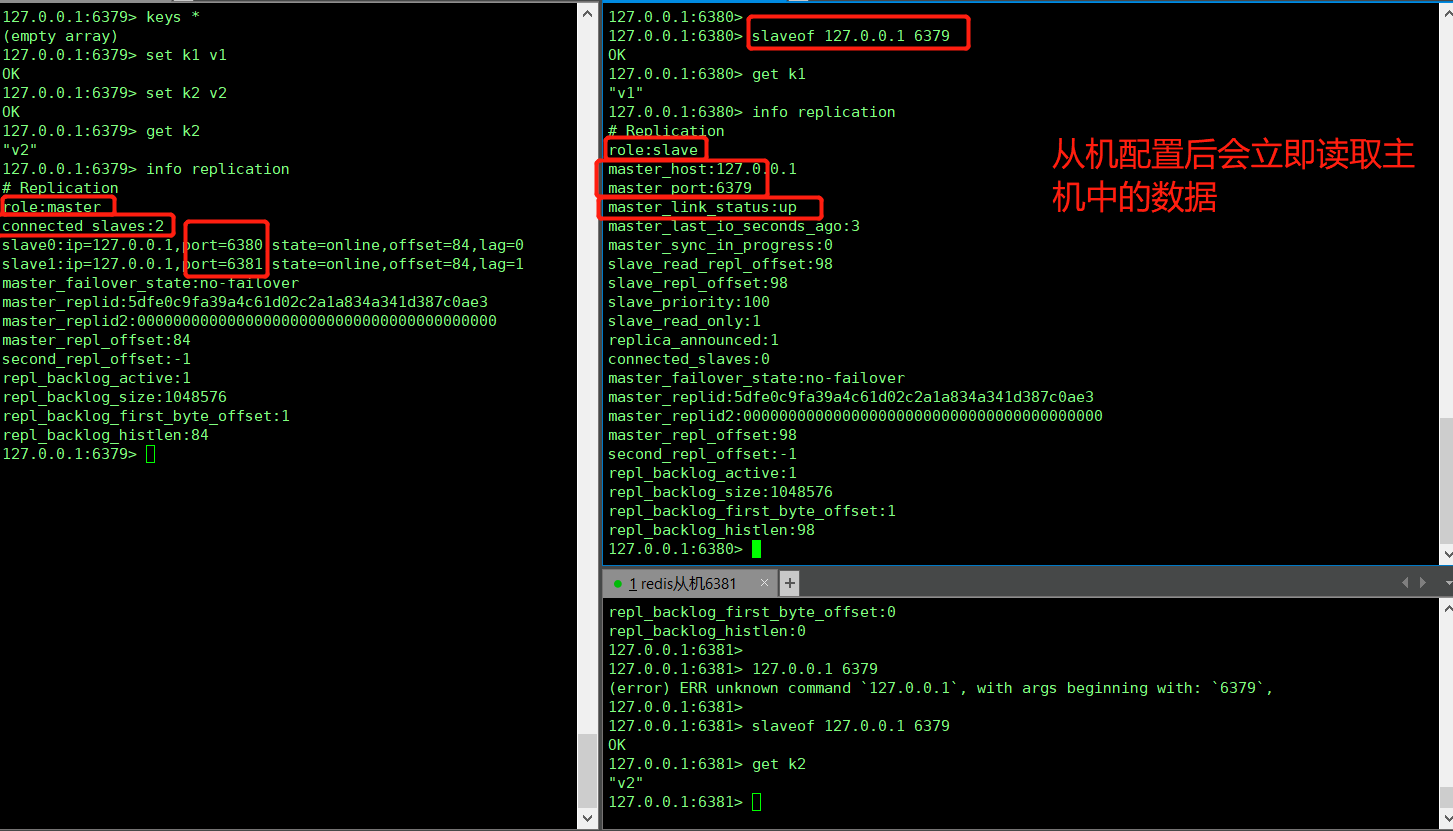
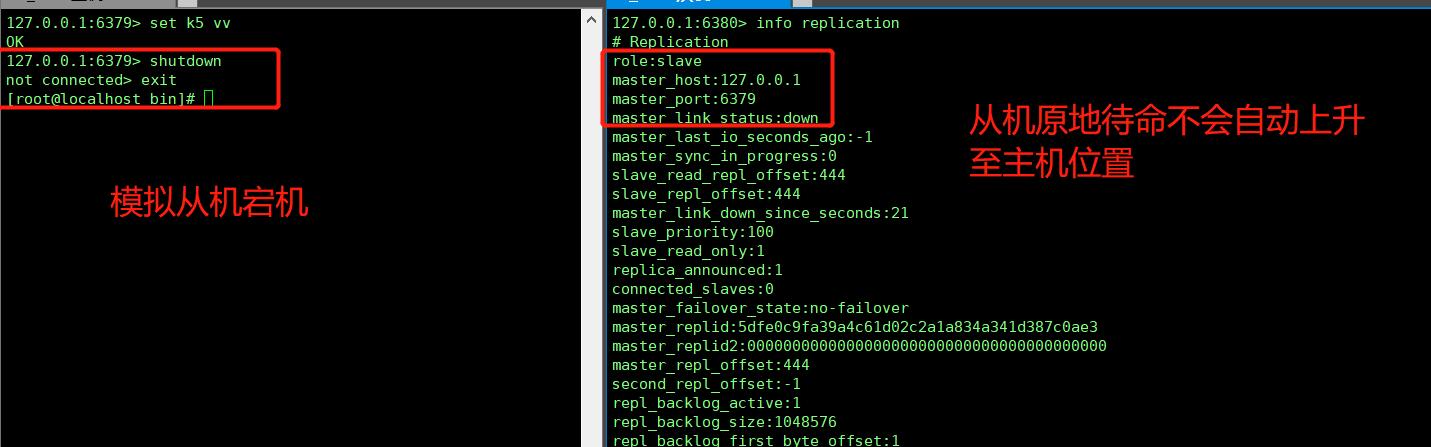
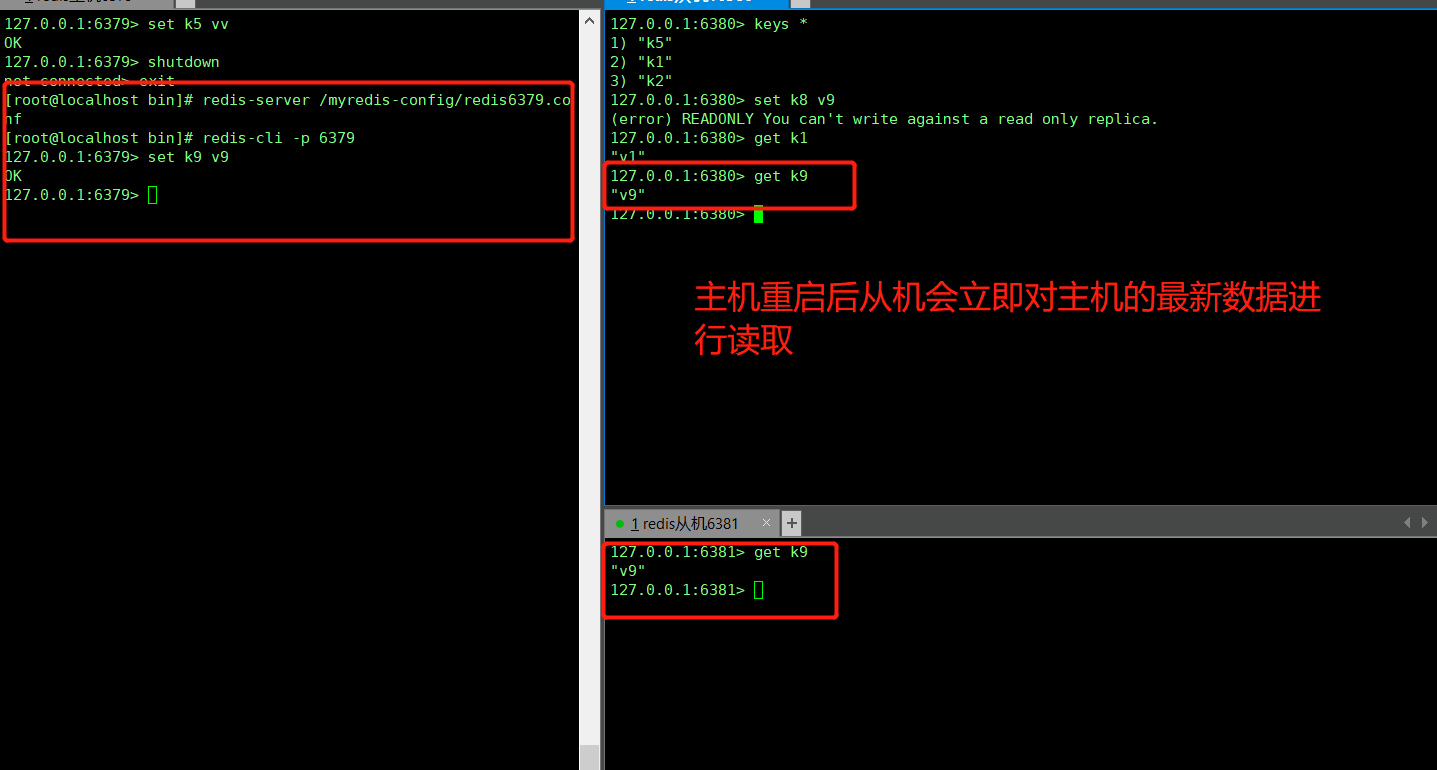
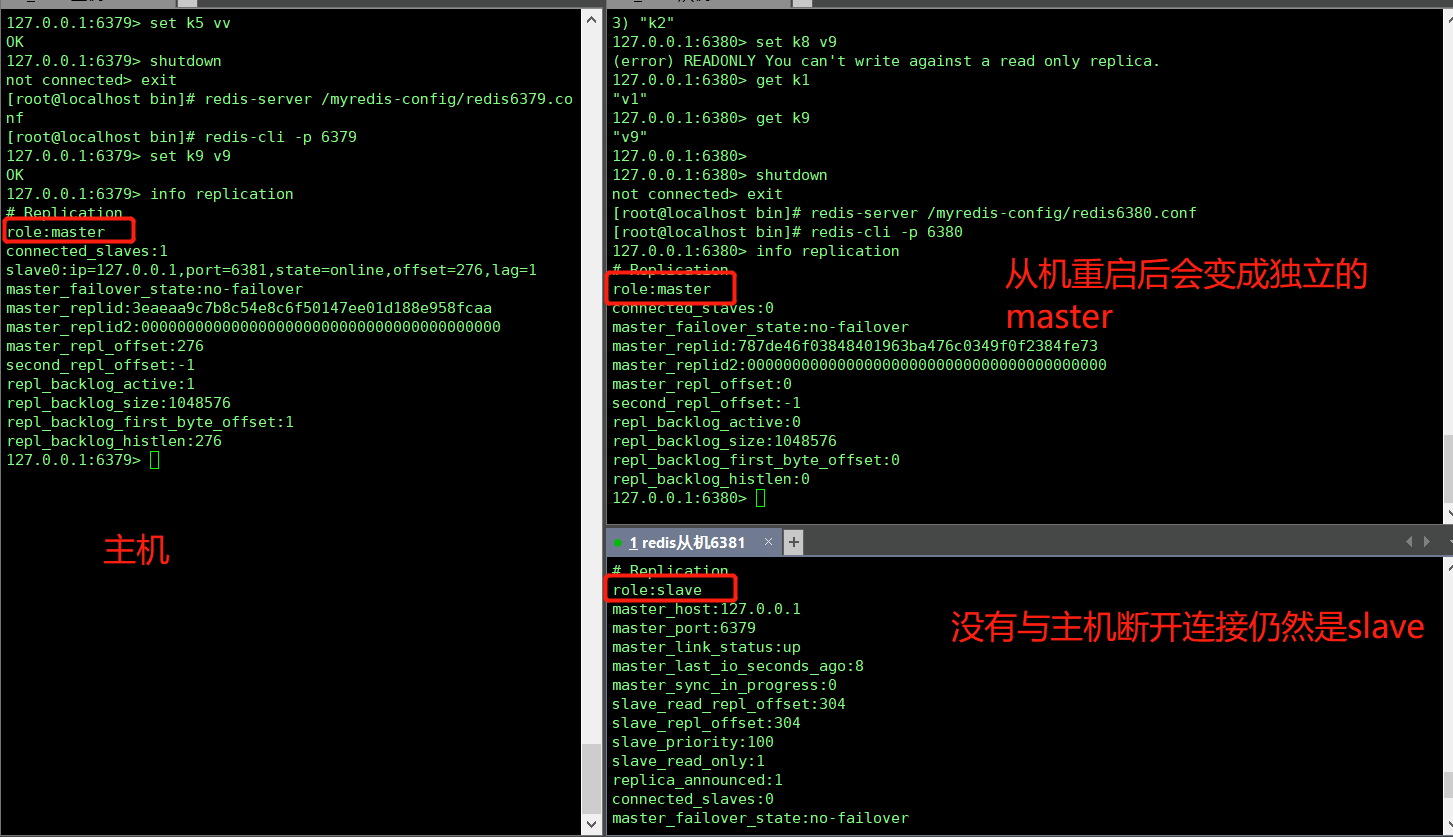
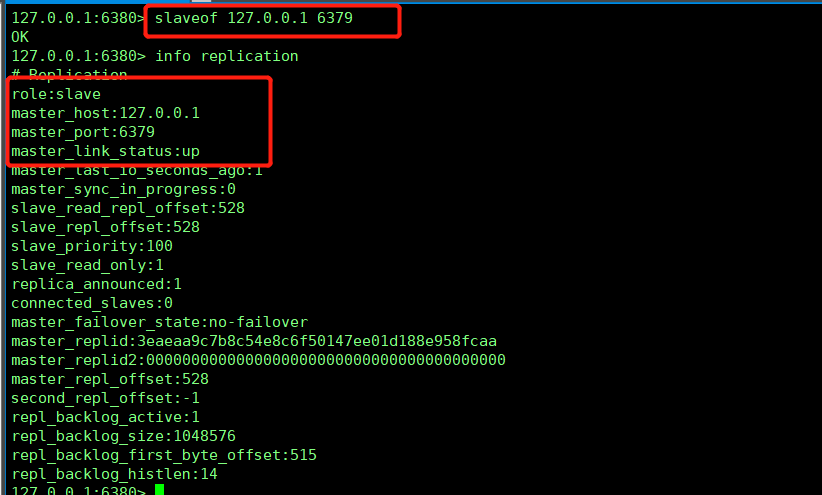
多台redis协同完成数据缓存，此时我们会设置一台主机 剩余的都作为从机，主机主要用于写操作，从机主要用于读操作，当主机更新数据后会通过配置自动同步到从机。

主从复制可以实现读写分离，容灾备份 相比单机的redis 对于备份而言 数据会更加近似于完整性

## 配置主从复制

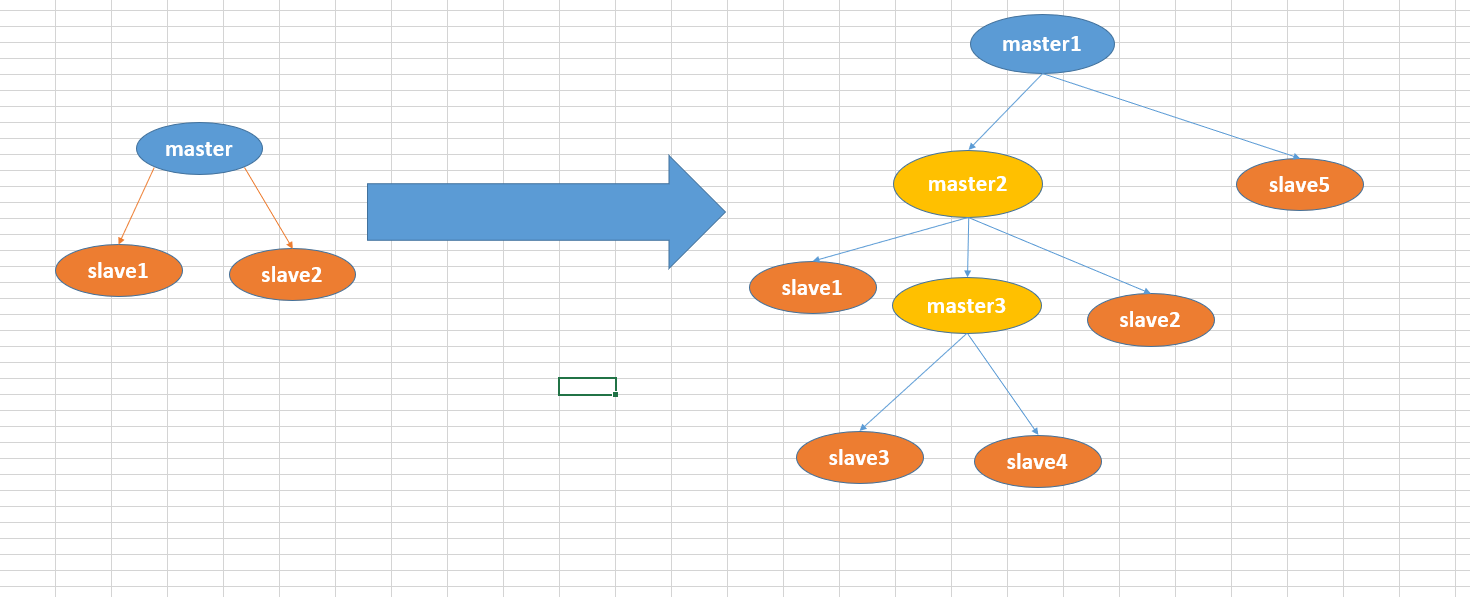
### 方式1 控制台配置

#### 一主多从

1. 配从机不配主机 - 所有配置都在从机的客户端进行配置
2. 从机配置：slaveof主机IP和主机端口  
   这种方式当从机每次与主机断开连接时都需要重新slaveof主机IP和端口 否则从机就会变成独立的主机(master) 可以通过对从机的配置文件修改完善这块问题
3. 修改配置文件细节
   1. 模拟情况下 copy多个redis配置文件分别修改端口 代表是不同的机器上运行的redis  
      
   2. 全部开启daemonize yes 后台运行  
      
   3. pid进程文件名字需要改为对应的  
      
   4. 指定每台机器的端口  
      
   5. log的文件名需要配置  
      
   6. dump.rdb备份文件名字需要配置  
      
   7. 分别启动三个redis
   8. 在客户端通过info replication查看各个redis状态  
      
   9. 在从机的客户端配置主机的ip地址和端口号
   10. 再次通过info replication查看各个redis的状态  
       
   11. 主机添加数据时 从机会立即进行备份
   12. 从机只能进行读取操作 不能对数据有任何的写操作  
       
   13. 主机宕机 从机会原地待命  
       
   14. 主机宕机时从机仍然不可以进行写操作  
       
   15. 主机恢复后 从机立即从主机读取最新的数据  
       
   16. 从机宕机 恢复后会变成独立的master 需要再次通过slaveof方式配置其主机IP和端口  
         
       

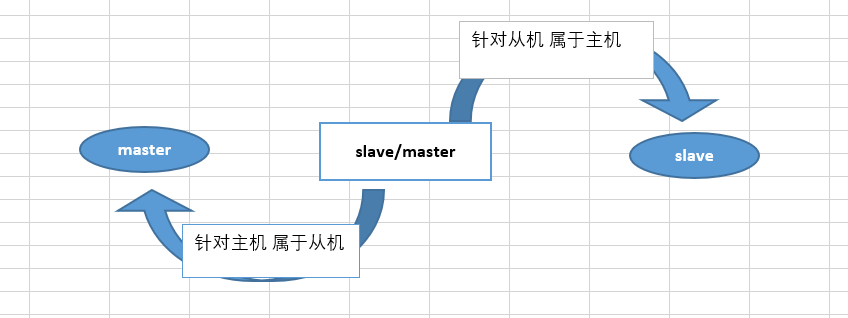
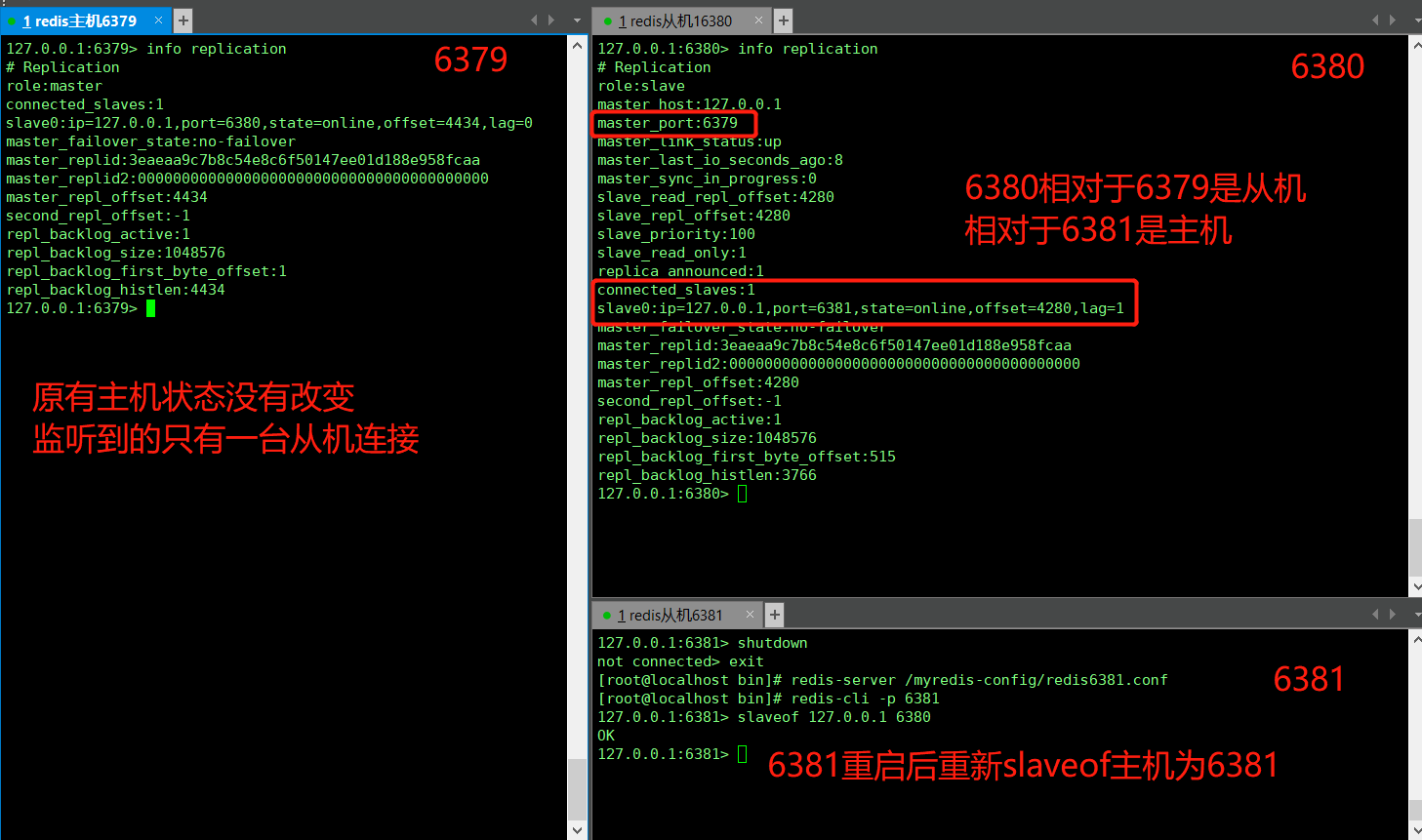
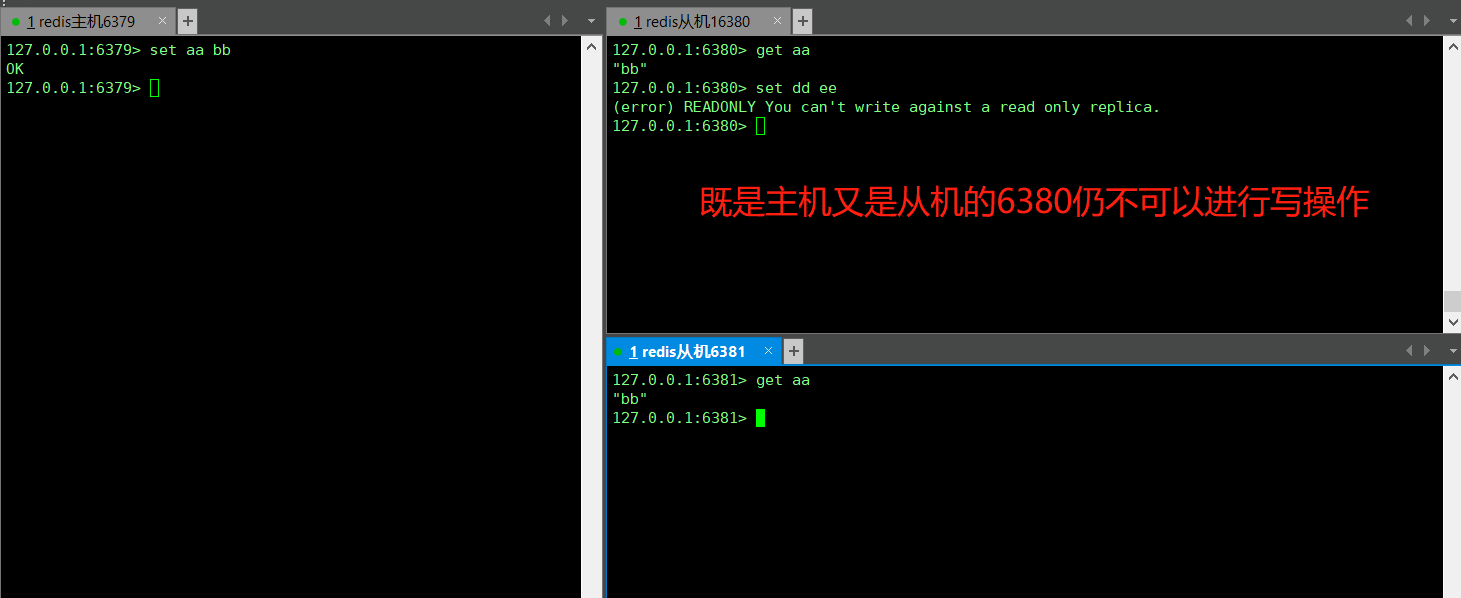
#### 多主多从

上一个主机的从机可以是下一个从机的主机 该从机同样可以接收其它从机的同步请求 这样就实现了去中心化 减轻了一台主机多台从机中主机的压力



如果中途转向 会清除之前的数据 然后重新拉取最新数据

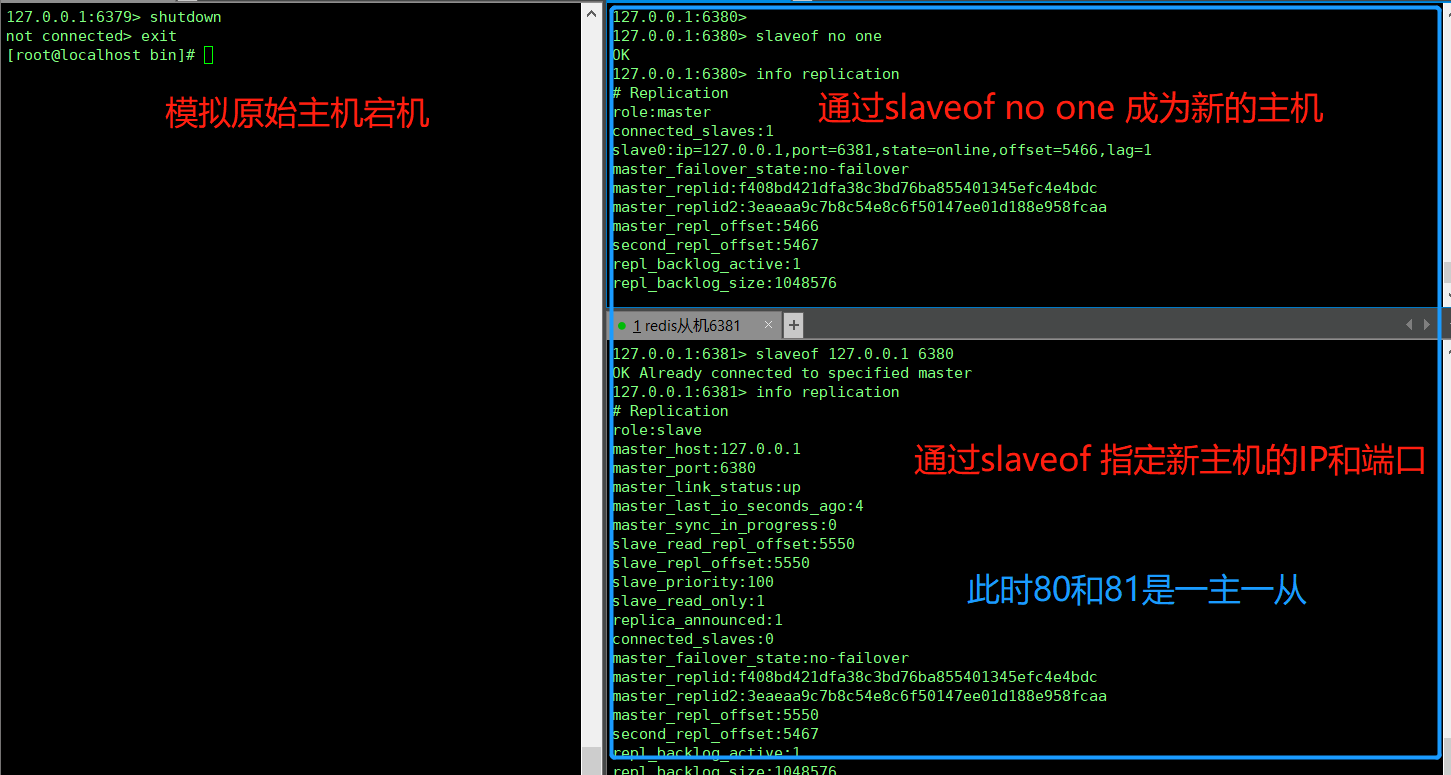
命令还是slaveof 要作为主机的ip和端口

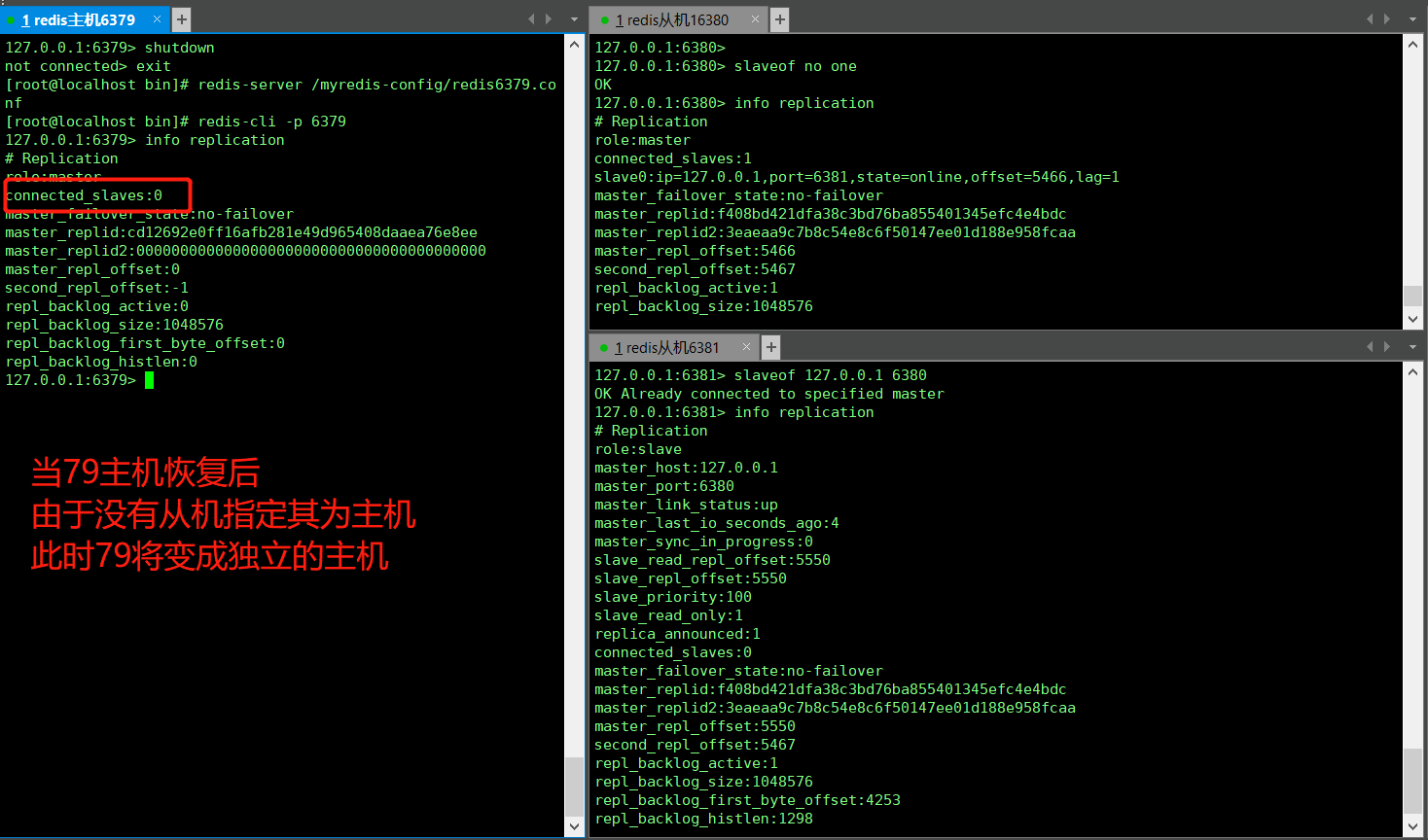
1. 多主多从下状态  
     
   
2. 多主机多从机模式下从机获取数据 既是主机又是从机仍不可以进行写操作  
   

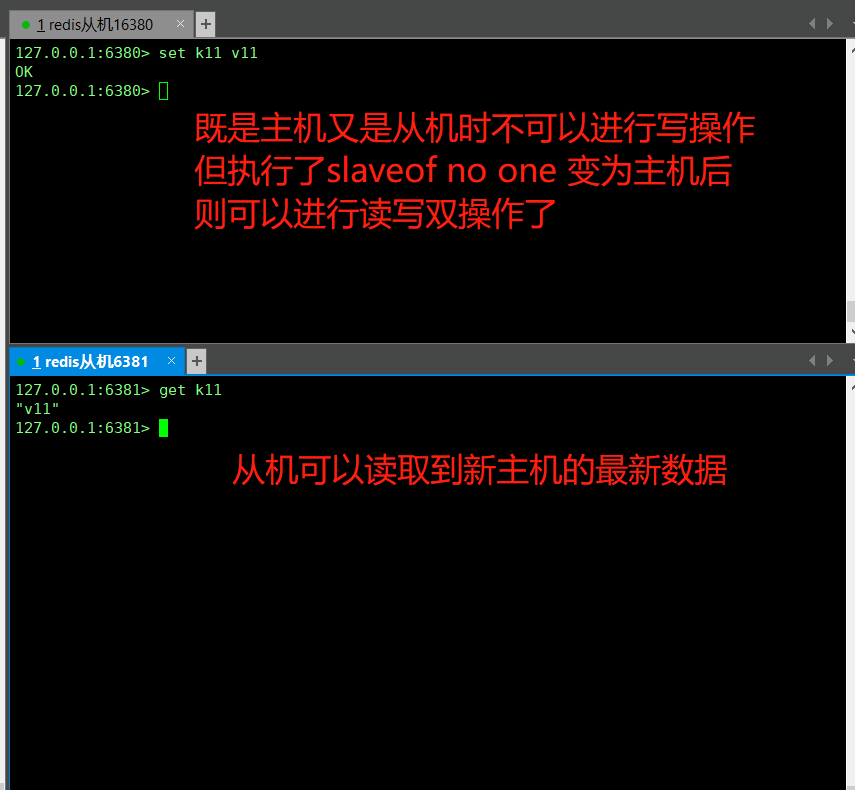
#### 反客为主

即当主机宕机后从机通过slaveof no one 指令成为新的主机 但之前的其它从机需要重新slaveof 新主机的IP和端口

当之前宕机的主机恢复后则会变成独立的master 不在挂载其它从机了(前提是所有从机重新制定了新的主机 原地待命的从机仍热挂在到原有主机上)







### 复制的原理



从机启动成功连接到主机时会发送一个sync命令 主机接收到命令后会启动后台的存盘进程 同时手机所有接收到的用于修改数据的命令集合在后台进行执行完毕后 主机将传送整个数据文件到从机 以完成第一次的数据同步

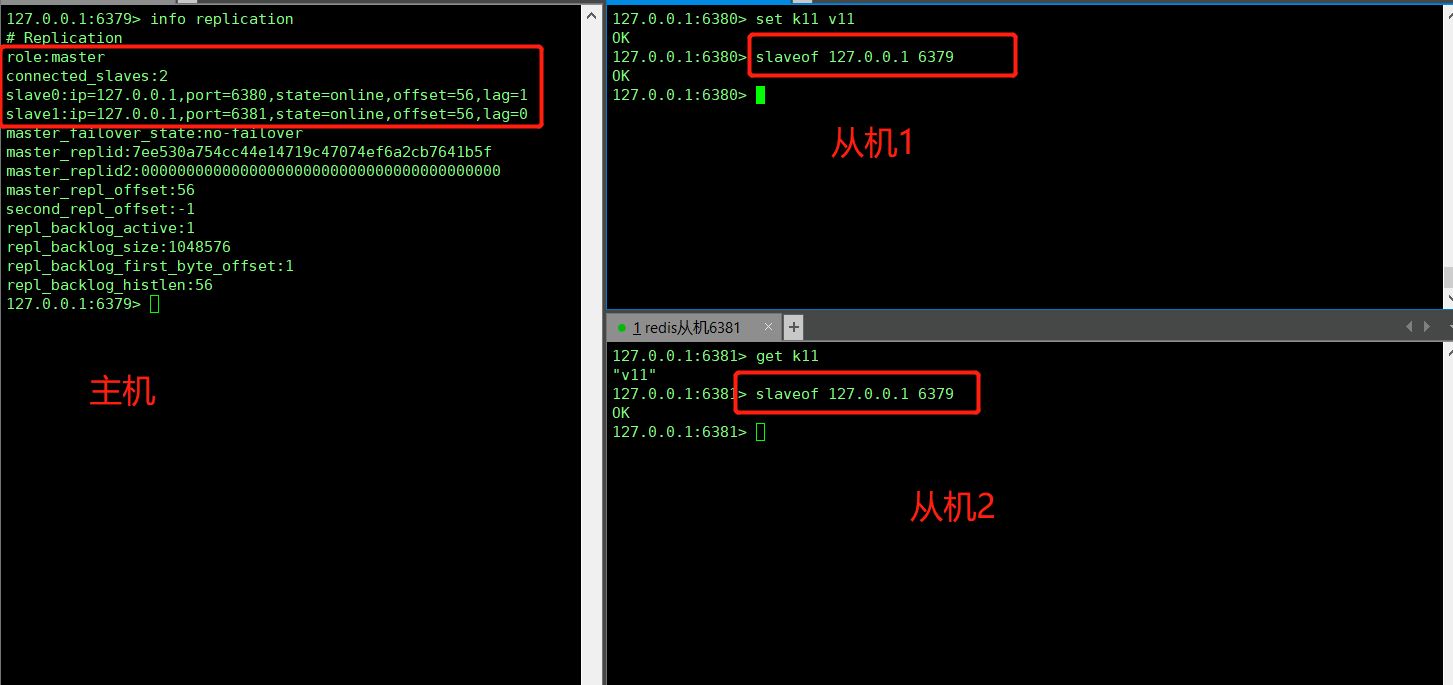
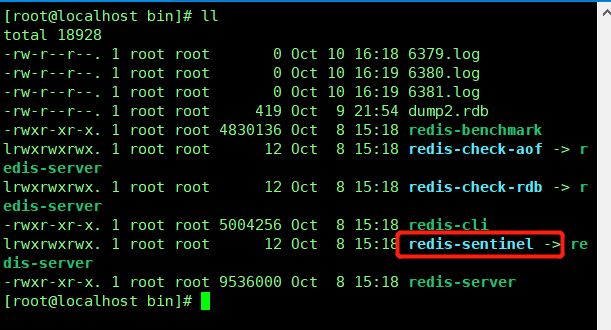
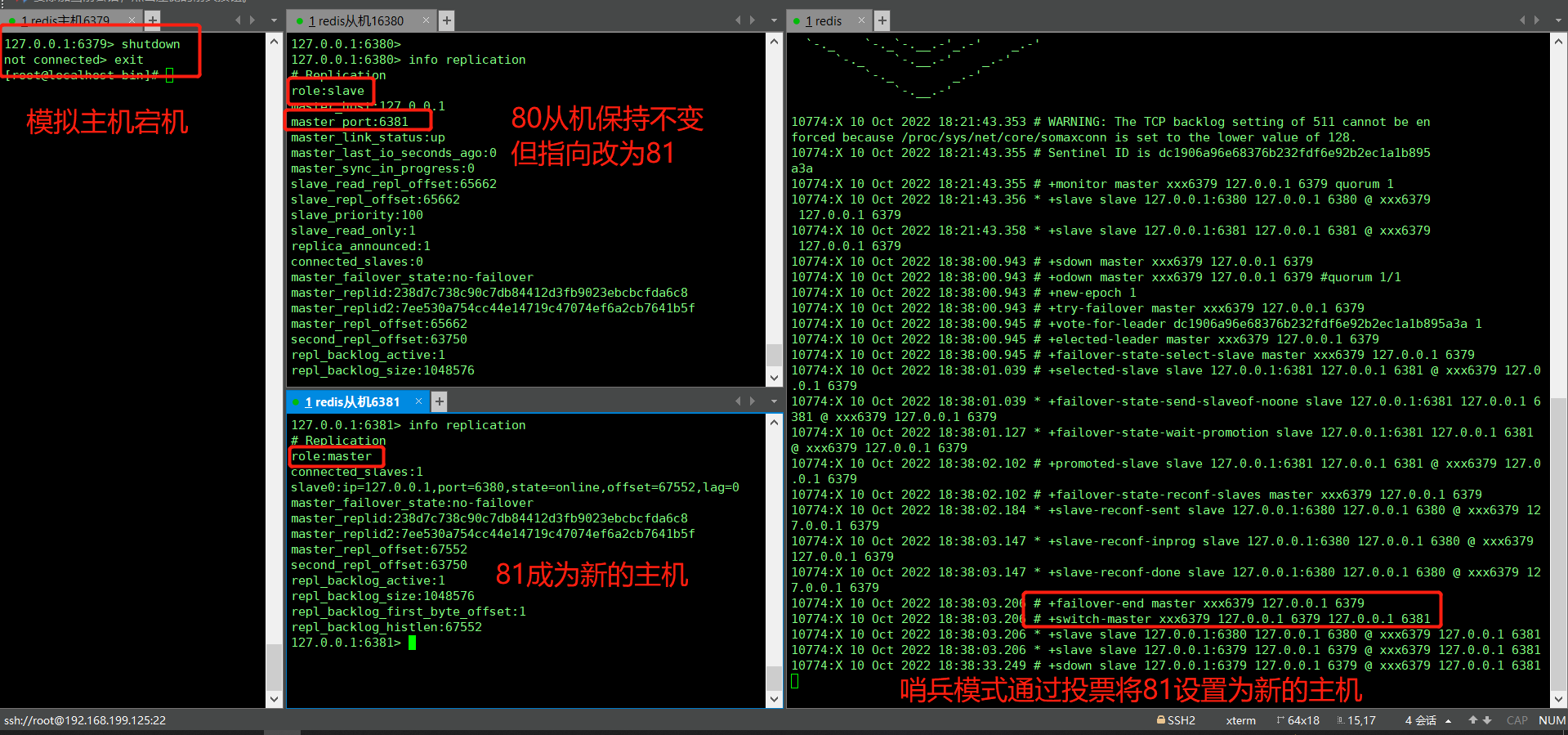
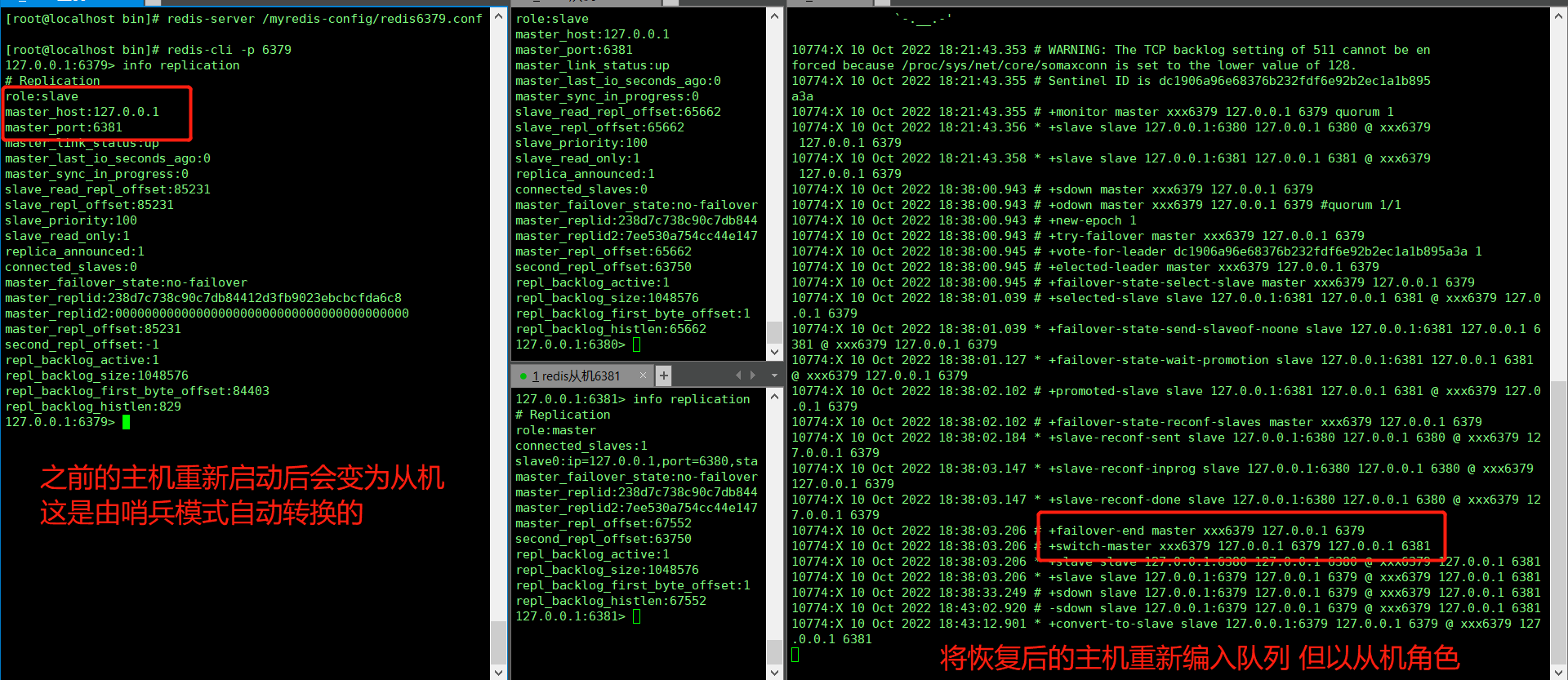
**从机连接到主机后第一次是全量复制主机的数据 从第一次之后则每次复制改为增量模式 如果重新连接到主机 则又会从第一次开始**

### 哨兵模式(自动监控模式)

反客为主的自动版，反客为主模式是需要我们手动执行slaveof no one成为新的主机 并手动调整其它从机的指向 而哨兵模式则是通过计算结果自动选出新的主机(主机宕机后才会触发哨兵模式)

**注意 即便配置了哨兵模式 如果我们手动干预选择主机(指向slaveof 主机的IP和端口) 哨兵模式会自动将其它从机挂载到指定的主机上**

#### 演示

1. 调整结构为一主二从  
   
2. 自定义sentinel.conf文件并添加如下内容(注意 文件名不能随意)  
   **sentinel monitor xxx6379 127.0.0.1 6379 1**  
   解释：sentinel monitor 被监控的数据库名称(可随意写) 被监控的主机IP和端口 最后的1表示当监控的主机宕机后 让从机进行投票 谁的票数多 谁就是新的主机
3. 启动哨兵模式  
   **redis-sentinel** + sentinel.conf文件所在的目录即该文件  
   例如 redis-sentinel /myredis/sentinel.conf  
     
   
4. 模拟主机宕机 查看从机状态  
   
5. 恢复之前宕机的主机 查看其状态  
   
6. 主从的优缺点  
   优点  
   可以实现读写分离 可以通过配置哨兵模式自动监听和调整主从分配，可以实现最大化的近似数据完整(数据备份)  
   缺点  
   所有的写操作只能在主机上进行 然后同步到从机，所以会有一定的延时 当遇到系统繁忙或网络异常时 这种情况会更加严重 另外从机数量的增加也会使得这个问题更加严重

### 方式2 redis.conf中配置主从

在从机的配置文件中REPLICATION节点下添加天边内容

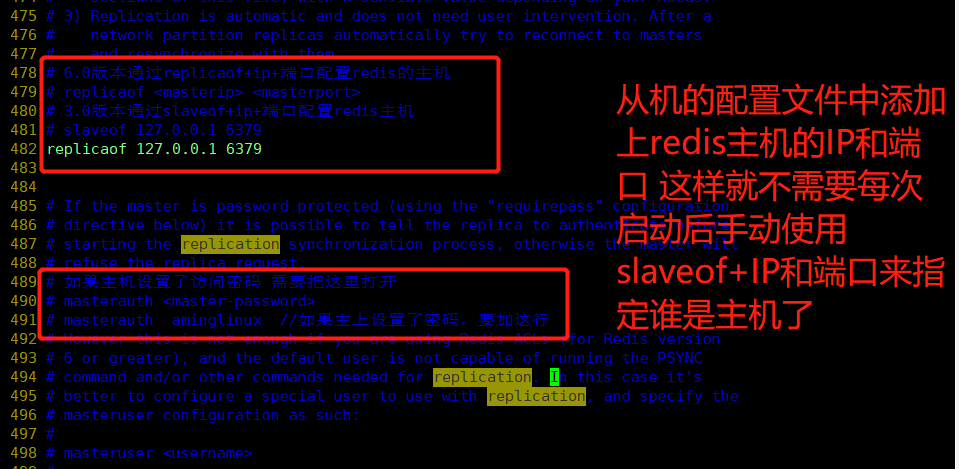
replicaof 127.0.0.1 6379

如果redis是3.0 则需要使用slaveof +主机的IP和端口

另外如果主机配置了访问密码 需要在配置文件中添加下边内容

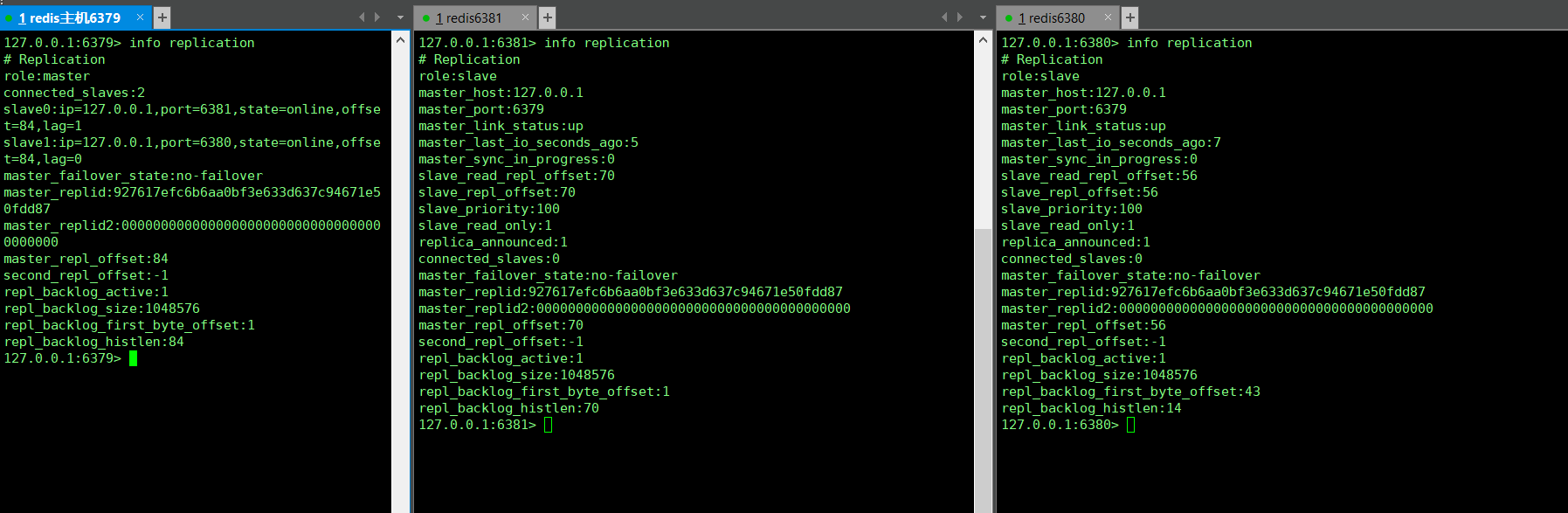
masterauth <master-password>

masterauth  访问密码  //如果主上设置了密码，要加这行



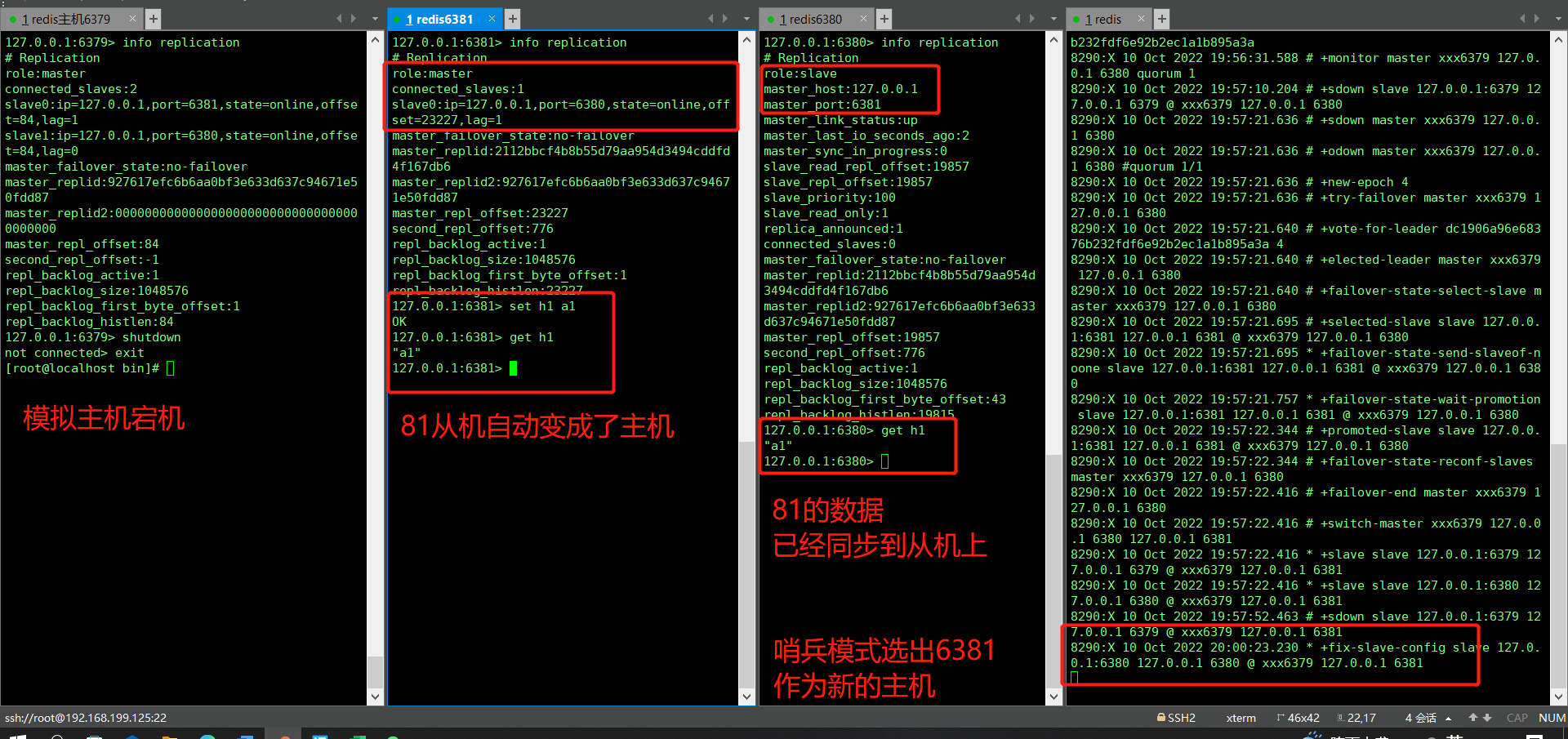
重启所有redis进行测试

可以看到从机会自动挂在到6379的主机上

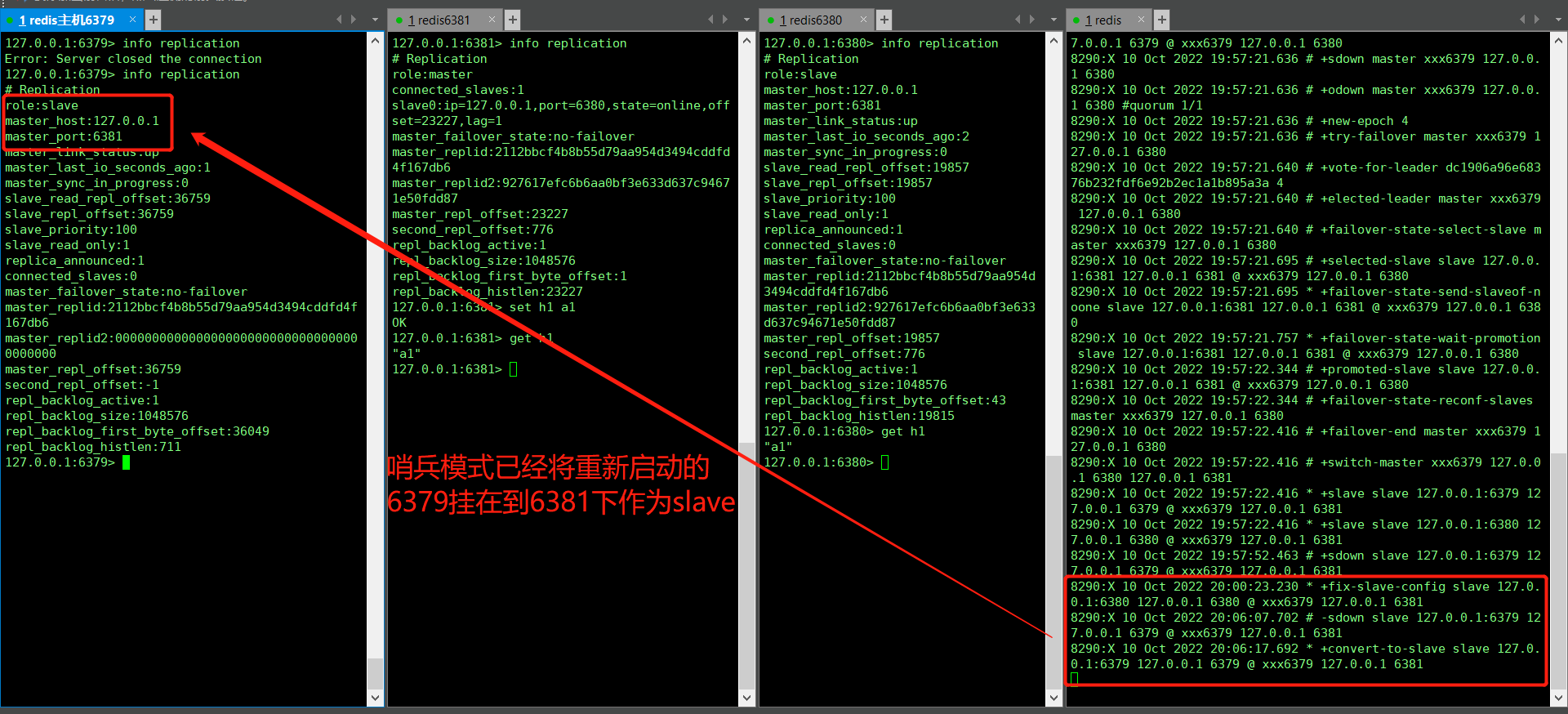


测试哨兵模式下主从

当主机宕机后 哨兵模式会自动选出适合的从机作为新的主机 然后将另一台从机挂在到新的主机上



原来的主机重新启动后哨兵模式会将原来的主机改为从机并挂在到新的主机上



# Jedis

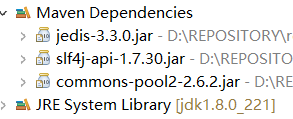
## 概述

jedis是用于Java连接redis的组件包，通过该组件可以使用Java来操作redis,封装了redis中所有数据类型的操作命令对应的方法

## 组件包下载

会直接将需要的commons-pool一块依赖进来

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>3.3.0</version>  </dependency> |

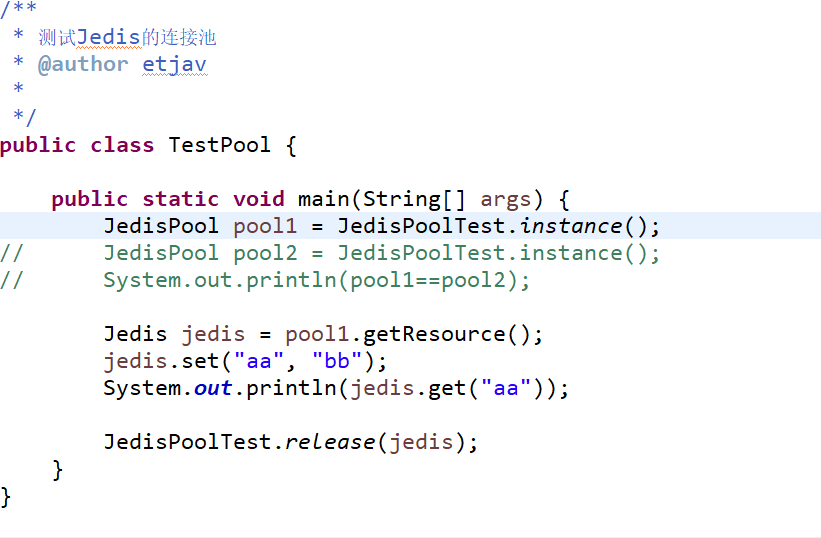


## Jedis连接池封装

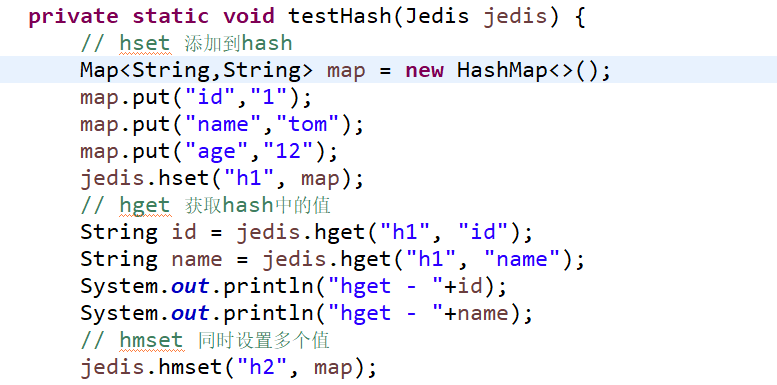
采用单例模式封装各种池子 这里也不例外

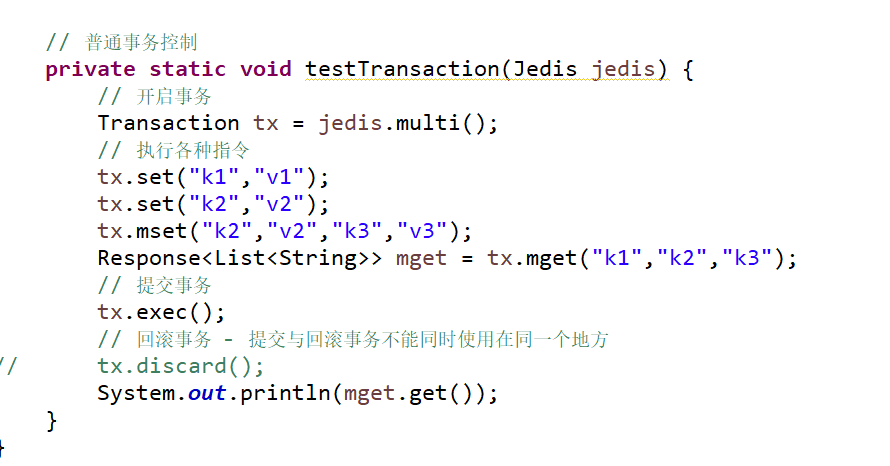
|  |
| --- |
| **package** com.etjava.jedis;  **import** redis.clients.jedis.Jedis;  **import** redis.clients.jedis.JedisPool;  **import** redis.clients.jedis.JedisPoolConfig;  /\*\*  \* 封装jedis连接池  \* **@author** etjav  \*  \*/  **public** **class** JedisPoolTest {  p**rivate** JedisPoolTest() {}  // volatile 关键字保证可见性 但不保证原子性  **private** **static** JedisPool *jedisPoll* = **null**;  // 单例模式创建jedis连接池  **public** **static** JedisPool instance() {  **if**(*jedisPoll*==**null**) {  **synchronized** (JedisPoolTest.**class**) {  **if**(*jedisPoll*==**null**) {  // jedis连接池的配置  JedisPoolConfig config = **new** JedisPoolConfig();  config.setMaxTotal(1000);// 最大连接数  config.setMaxIdle(32); // 最大空闲  config.setMaxWaitMillis(3000);// 最大等待时间  *jedisPoll* = **new** JedisPool(config,"192.168.199.125",6379);  }  }  }  **return** *jedisPoll*;  }  // 用完连接后在放回连接池中  /\*  \* jedis2.9以上的版本的close方法把 jedispool.returnBrokenResource();jedispool.returnResource()包括了，  \* 所以直接jedis.close 就可以释放连接了,  \* 但如果你在使用redis的时侯，想要close，就不能释放，所以你先要退出client，  \* 就要用jedis.quit(),这样就完事了  public void close(jedis){  jedis.quit();  jedis.disconnect();  }  \*/  **public** **static** **void** release(Jedis jedis) {  **if**(jedis!=**null**) {  jedis.quit();  jedis.disconnect();  jedis.close();  }  }  } |

测试连接池



## Jedis简单测试





# redis配置远程连接

redis.conf文件 network节点下 注释掉绑定的本地地址及protected-mode

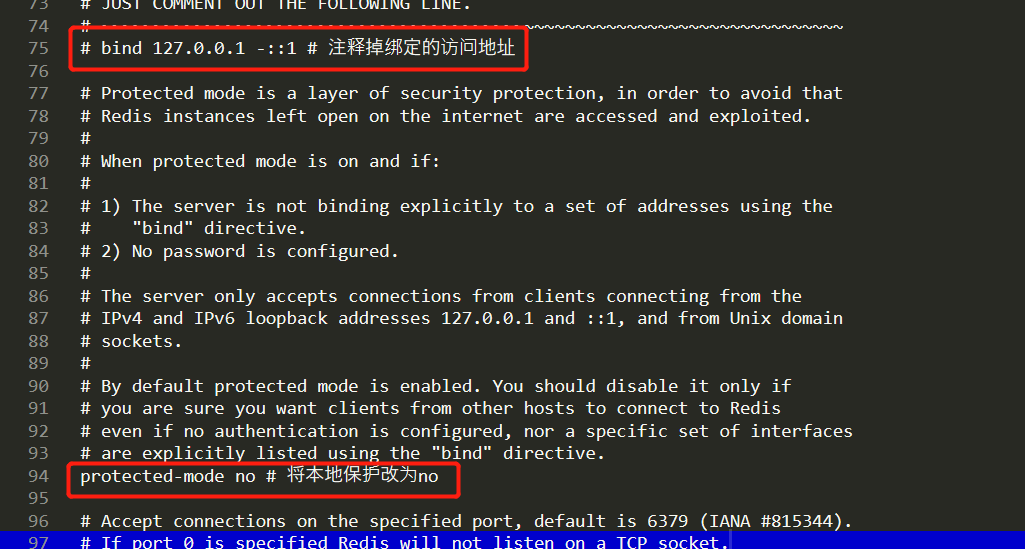
给防火墙添加6379端口

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

firewall-cmd --reload 重启防火墙

如果是自己测试可以直接将防火墙关闭

|  |
| --- |
| systemctl stop firewalld.service #停止firewall  systemctl disable firewalld.service #禁止firewall开机启动  firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running） |



# redis集群配置