

# redis

2025.1.19

## 1.安装

redis安装后的一些文件默认是在/usr/local/bin目录下的：

```
总用量 21492
-rwxr-xr-x. 1 root root 5197848 12月 10 19:47 redis-benchmark
lrwxrwxrwx. 1 root root      12 12月 10 19:47 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 12月 10 19:47 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5411040 12月 10 19:47 redis-cli
lrwxrwxrwx. 1 root root      12 12月 10 19:47 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 11390192 12月 10 19:47 redis-server
```

自上而下分别是redis的测试工具、修复有问题的aof文件、修复有问题的rdb文件、客户端操作入口、Redis集群哨兵、redis服务器启动命令。

并且自带一个出厂默认的配置文件的 (/export/server/redis)：

```
[root@centos redis]# ll
总用量 264
-rw-rw-r--. 1 root root 34885 7月 18 2022 00-RELEASENOTES
-rw-rw-r--. 1 root root 51 7月 18 2022 BUGS
-rw-rw-r--. 1 root root 5027 7月 18 2022 CODE_OF_CONDUCT.md
-rw-rw-r--. 1 root root 2634 7月 18 2022 CONTRIBUTING.md
-rw-rw-r--. 1 root root 1487 7月 18 2022 COPYING
drwxrwxr-x. 7 root root 187 12月 10 19:41 deps
-rw-rw-r--. 1 root root 11 7月 18 2022 INSTALL
-rw-rw-r--. 1 root root 151 7月 18 2022 Makefile
-rw-rw-r--. 1 root root 6888 7月 18 2022 MANIFESTO
-rw-rw-r--. 1 root root 22441 7月 18 2022 README.md
-rw-rw-r--. 1 root root 106545 7月 18 2022 redis.conf
-rwxrwxr-x. 1 root root 279 7月 18 2022 runtest
-rwxrwxr-x. 1 root root 283 7月 18 2022 runtest-cluster
-rwxrwxr-x. 1 root root 1578 7月 18 2022 runtest-moduleapi
-rwxrwxr-x. 1 root root 285 7月 18 2022 runtest-sentinel
-rw-rw-r--. 1 root root 1695 7月 18 2022 SECURITY.md
-rw-rw-r--. 1 root root 14005 7月 18 2022 sentinel.conf
drwxrwxr-x. 4 root root 12288 12月 10 19:47 src
drwxrwxr-x. 11 root root 199 7月 18 2022 tests
-rw-rw-r--. 1 root root 3055 7月 18 2022 TLS.md
drwxrwxr-x. 8 root root 4096 7月 18 2022 utils
```

2025.1.20

## 2.怎么玩？

### 1.修改配置文件

redis的运行依赖于自定义的配置文件，这类配置文件在出厂时默认会自带一个，我们一般把默认的配置文件的进行复制，就是保留一个备份，这是个好习惯，这里我们保存在/export/server/myredis下：

```
[root@centos myredis]# ll
总用量 108
-rw-r--r--. 1 root root 106545 1月 19 10:27 redis.conf
```

然后修改我们的配置文件，修改后记得重启，因为不是实时生效

使用vim进入配置文件：

```
# Redis configuration file example.
#
# Note that in order to read the configuration file, Redis must be
# started with the file path as first argument:
#
# ./redis-server /path/to/redis.conf

# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.

##### INCLUDES #####

# Include one or more other config files here. This is useful if you
# have a standard template that goes to all Redis servers but also need
# to customize a few per-server settings. Include files can include
# other files, so use this wisely.
#
# Note that option "include" won't be rewritten by command "CONFIG REWRITE"
# from admin or Redis Sentinel. Since Redis always uses the last processed
# line as value of a configuration directive, you'd better put includes
# at the beginning of this file to avoid overwriting config change at runtime.
#
```

使用set nu显示行号

1.修改daemonize为true（如果是docker则不能设置为true）

使用/daemonize查找：

```
##### GENERAL #####

# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
# When Redis is supervised by upstart or systemd, this parameter has no impact.
daemonize no
```

这个是干什么的呢？我们尝试使用redis-server启动一下redis就会发现，虽然redis启动了，但是是以前台方式启动的，也就是说我们在启动后无法进行其他输入操作，画面会停在redis的logo界面，作为一个后端服务器我们希望他以后台方式启动，所以将其改为yes：

```
304 ##### GENERAL #####
305
306 # By default Redis does not run as a daemon. Use 'yes' if you need it.
307 # Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
308 # When Redis is supervised by upstart or systemd, this parameter has no impact.
309 daemonize yes
310
```

2.修改protected-mode yes为protected-mode no

```

101 # Protected mode is a layer of security protection, in order to avoid that
102 # Redis instances left open on the internet are accessed and exploited.
103 #
104 # When protected mode is on and the default user has no password, the server
105 # only accepts local connections from the IPv4 address (127.0.0.1), IPv6 address
106 # (:::1) or Unix domain sockets.
107 #
108 # By default protected mode is enabled. You should disable it only if
109 # you are sure you want clients from other hosts to connect to Redis
110 # even if no authentication is configured.
111 protected-mode yes

```

```

101 # Protected mode is a layer of security protection, in order to avoid that
102 # Redis instances left open on the internet are accessed and exploited.
103 #
104 # When protected mode is on and the default user has no password, the server
105 # only accepts local connections from the IPv4 address (127.0.0.1), IPv6 address
106 # (:::1) or Unix domain sockets.
107 #
108 # By default protected mode is enabled. You should disable it only if
109 # you are sure you want clients from other hosts to connect to Redis
110 # even if no authentication is configured.
111 protected-mode no

```

这个是保护模式的开关，在生产模式中不建议关闭，在开发模式中可以关闭，因为我们在开发中需要让我们的springboot或者springcloud微服务连接到我们的redis，所以需要把这个关闭。

### 3.修改我们的bind

先找到我们的bind：

```

73 # WARNING If the computer running Redis is directly exposed to the
74 # internet, binding to all the interfaces is dangerous and will expose the
75 # instance to everybody on the internet. So by default we uncomment the
76 # following bind directive, that will force Redis to listen only on the
77 # IPv4 and IPv6 (if available) loopback interface addresses (this means Redis
78 # will only be able to accept client connections from the same host that it is
79 # running on).
80 #
81 # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
82 # COMMENT OUT THE FOLLOWING LINE.
83 #
84 # You will also need to set a password unless you explicitly disable protected
85 # mode.
86 #
87 bind 127.0.0.1 -:::1

```

这个默认是只有127.0.0.1，也就是说只有本机才能访问我们的redis，这个在以后做微服务的时候是扩展性不高的，所以直接注释掉，让所有的ip都可以连接进我们的redis中。

```

87 #bind 127.0.0.1 -:::1

```

### 4.添加redis密码

查找我们的requirepass：

```

1027 # IMPORTANT NOTE: starting with Redis 6 "requirepass" is just a compatibility
1028 # layer on top of the new ACL system. The option effect will be just setting
1029 # the password for the default user. Clients will still authenticate using
1030 # AUTH <password> as usually, or more explicitly with AUTH default <password>
1031 # if they follow the new protocol: both will work.
1032 #
1033 # The requirepass is not compatible with aclfile option and the ACL LOAD
1034 # command, these will cause requirepass to be ignored.
1035 #
1036 # requirepass foobared

```

这里我们设置我们的redis访问密码为111

```
1037 requirepass 111
```

## 2.启动redis

使用redis-server 配置文件路径 来启动我们的redis:

```
[root@centos myredis]# redis-server redis7.conf
```

查看我们的redis的运行状态:

```
[root@centos myredis]# ps -ef|grep redis|grep -v grep
root      57617      1  0 16:31 ?        00:00:00 redis-server *:6379
```

可以看到我们的redis在我们的6379端口运行

## 3.以客户端模式连接我们的reids

使用reids-cli -a 密码 -p 端口号 来连接到我们对端口号的redis服务器 (如果不写-p就默认连接到6379)

如果我们不写-a, 我们也可以连接到redis-cli中, 但是我们不能进行操作, 需要使用

```
[root@centos myredis]# redis-cli -a 111 -p 6379
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
127.0.0.1:6379>
```

然后我们新开一个窗口查看redis的运行状况:

```
[root@centos ~]# ps -ef|grep redis
root      57617      1  0 16:31 ?        00:00:01 redis-server *:6379
root      64366    2755  0 16:36 pts/1    00:00:00 redis-cli -a 111 -p 6379
root      66026    65172  0 16:37 pts/3    00:00:00 grep --color=auto redis
[root@centos ~]#
```

可以发现除了我们的reids-server服务端外, 又多了一个redis-cli客户端

使用quit命令可以退出我们的reids客户端, 但仅仅是退出, 不会关闭我们的redis服务器。

关闭redis服务器可以用redis-cli -a 密码 shutdown (单实例关闭)

redis-cli -p 端口号 shutdown (多实例关闭)

## 3.一切始于helloworld

我们连接进redis-cli后:

使用redis的基础命令: set和get:

```
127.0.0.1:6379> set k1 helloworld
OK
127.0.0.1:6379> get k1
"helloworld"
```

## 4.redis十大数据类型

对于数据类型的操作命令, 可以去[Commands](#) | [Docs](#)查找, 也可以问ai

## 首先是对于redis中的key操作:

set k1 v1-设置k1为v1值, 类似于键值对。这里的k1是区分大小写的, 也就是q1和Q1是两个key

keys \*-查找所有的key

exists k1-查看k1是否存在

type k1-查看k1的数据类型

del k1-删除k1, 成功返回1, 失败返回0

unlink k1-非阻塞删除k1, 先给k1打上标记, 将k1与v1的连接切断, v1真正的删除工作由后续异步操作进行

ttl k1-查看当前k1还有多少秒过期, -1为永不过期, -2为已过期

expire k1 秒数-设置k1的过期时间

move k1 dbindex [0-15]-将当前的k1移动到数据库给定的db当中

这个是什么意思呢? 我们的redis数据库一共有16个库, 我们一般默认的直接访问就是0号库, 假如我们在0号库有k1:

```
127.0.0.1:6379> get k1
"hello world"
```

然后我们将其移动到3号库:

```
127.0.0.1:6379> move k1 3
(integer) 1
```

我们在0号库进行查询发现没有了:

```
127.0.0.1:6379> get k1
(nil)
```

我们切换到3号库:

```
127.0.0.1:6379> select 3
OK
```

查询:

```
127.0.0.1:6379[3]> get k1
"hello world"
```

## 关于数据库的操作:

使用select dbindex来切换到对应编号对的数据库, 数据库一共有16个, 这是由配置文件来决定的, 我们进入redis的配置文件, 然后查询database:

```
# Set the number of databases. The default database is DB 0, you can select
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16
```

可以发现默认就是16个数据库。

dbsize-查看数据库的key的数量



flushdb-清空当前库

flushall-通杀全部库

## 1.String

String是redis最基本的数据类型，一个key对应一个value

String是二进制安全的，意味着String可以储存任何数据，包括jpg图片或者被序列化的对象

一个value最大为512M

**set和get的进阶：**

**set的完整格式：**

set key value [NX|XX] [GET] [EX seconds|PX milliseconds|EXAT unix-time-seconds|PXAT unix-time-milliseconds|KEEPTTL]

set key value [NX|XX] [GET] [EX seconds|PX milliseconds|EXAT unix-time-seconds|PXAT unix-time-milliseconds|KEEPTTL]

SET命令有EX、PX、NX、XX以及KEEPTTL五个可选参数，其中KEEPTTL为6.0版本添加的可选参数，其它为2.6.12版本添加的可选参数。

- EX seconds: 以秒为单位设置过期时间
- PX milliseconds: 以毫秒为单位设置过期时间
- EXAT timestamp: 设置以秒为单位的UNIX时间戳所对应的时间为过期时间
- PXAT milliseconds-timestamp: 设置以毫秒为单位的UNIX时间戳所对应的时间为过期时间
- NX: 键不存在的时候设置键值
- XX: 键存在的时候设置键值
- KEEPTTL: 保留设置前指定键的生存时间
- GET: 返回指定键原本的值，若键不存在时返回nil

SET命令使用EX、PX、NX参数，其效果等同于SETEX、PSETEX、SETNX命令。根据官方文档的描述，未来版本中SETEX、PSETEX、SETNX命令可能会被淘汰。

EXAT、PXAT以及GET为Redis 6.2新增的可选参数。

### 返回值

设置成功则返回OK；返回nil为未执行SET命令，如不满足NX、XX条件等。

若使用GET参数，则返回该键原来的值，或在键不存在时返回nil。

我们一个一个参数看：

set key value NX-当key不存在的时候才会生效并创建key-value

set key value XX-当key存在的时候才会生效，覆盖原来的value值

set key value px 1000-设置key的过期时间为1秒

**注意！当使用px或者ex设置过期时间以后，如果再次使用set key value，会把原来的过期时间覆盖掉，变成永不过期，这时候就需要使用keepTTL来继承过期时间**

mset k1 v1 k2 v2....-一次性设置多个key-value键值对

mget k1 k2 k3...-一次性获取多个值

msetnx k1 v1 k2 v2...-一次性设置多个key的value，但是其中的key都需要不存在，只要有一个存在，整个语句全部失败

getrange key [left-right]-按照下标截取key中的value

```
127.0.0.1:6379> GETRANGE k1 0 -1
"abcde"
127.0.0.1:6379> GETRANGE k1 0 3
"abcd"
```

setrange key index xxxx...-用xxxx...来替换index之后的东西

```
127.0.0.1:6379> SETRANGE k1 1 xxxxxxxx
(integer) 9
127.0.0.1:6379> get k1
"axxxxxxxx"
```

incr key-当key的value是数字的时候才可以用，数值加一

incrby key number-数值增加number

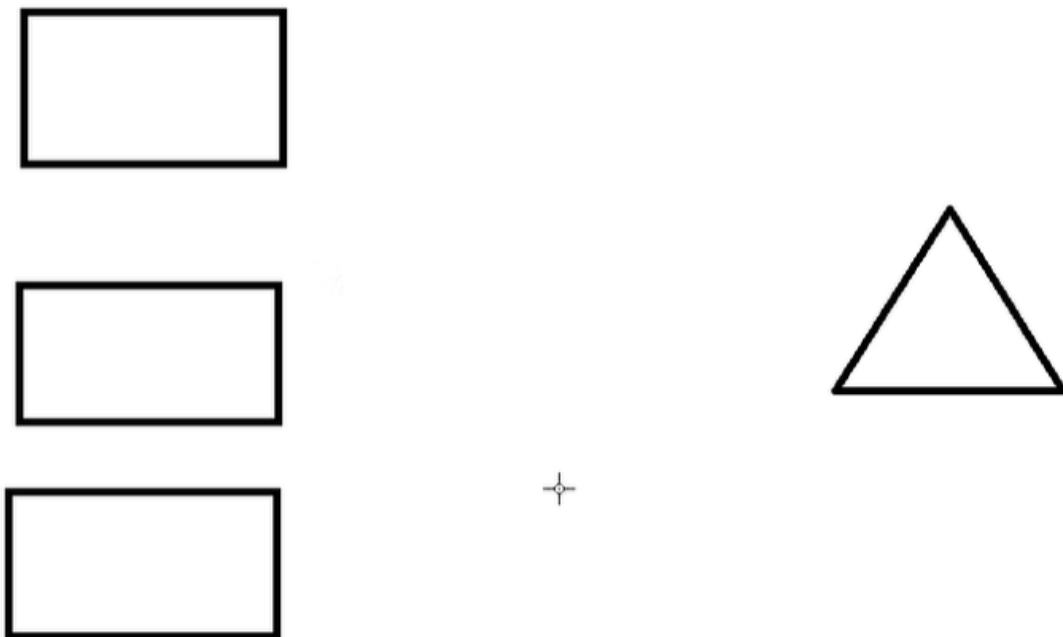
decr key-数值减一

decrby key number-数值减number

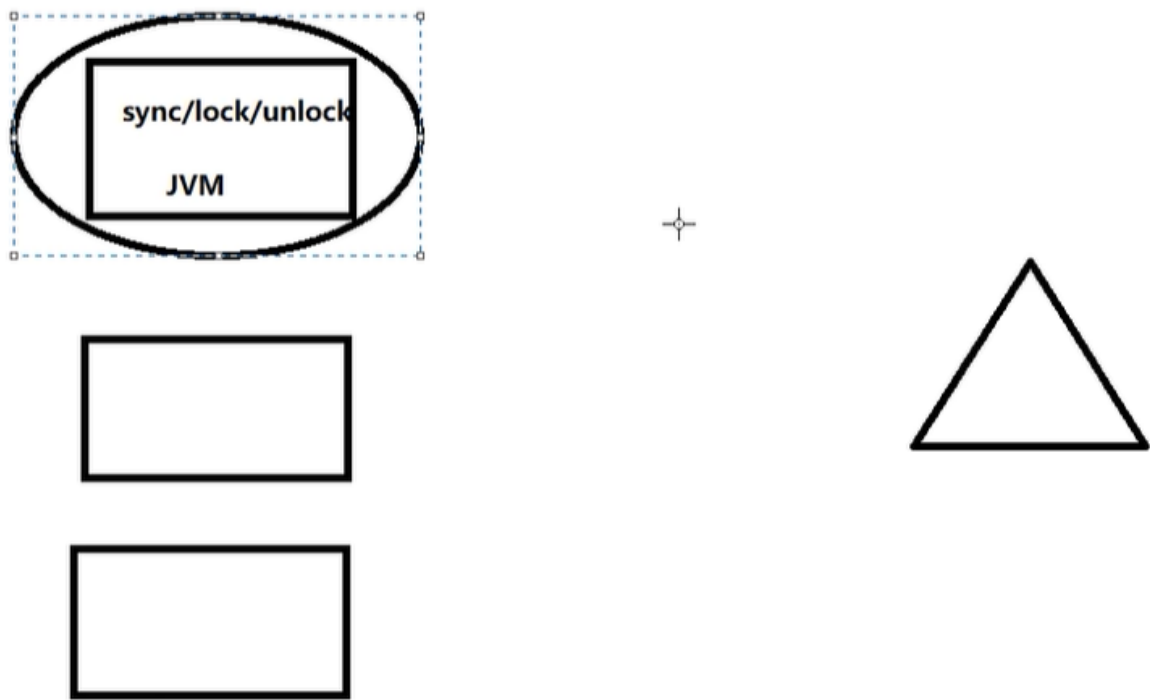
strlen key-获取key对应的value的字符串长度

append key string-内容追加

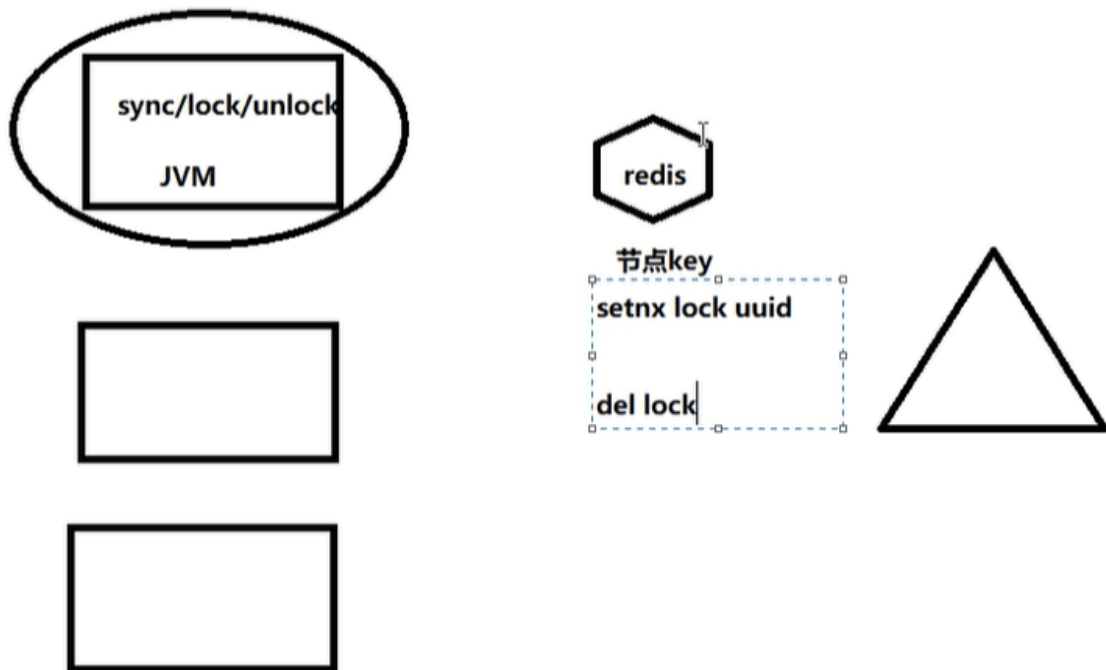
### 分布式锁



假如现在的微服务需要争夺一个资源，那我们很容易想到的就是加锁，但是我们常用的锁sync通常是只锁定一个微服务中的线程，对于其他微服务中的线程是无法管理的：



所以我们需要一个可以在微服务之间生效的锁，这就是分布式锁：



原理就类似于在redis中利用set lock threadname nx来创建一个只属于某个线程的key-value对，后来的线程使用自己的threadname进行比对，如果查询到值了，就进入，否则被拦截，当资源调用完成后再del lock。这样就实现了微服务之间的锁，只有一个微服务可以对资源进行操纵。

getset k1 value-先查询出k1的值并进行返回，然后用value进行覆盖

## 2.List

list的底层是一个双端链表的结构



一个双端链表的结构，容量是2的32次方减1个元素，大概40多亿，主要功能有push/pop等，一般用在栈、队列、消息队列等场景。

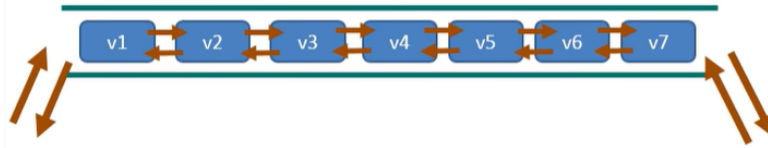
left、right都可以插入添加；

如果键不存在，创建新的链表；

如果键已存在，新增内容；

如果值全移除，对应的键也就消失了。

- 它的底层实际是个双向链表，对两端的操作性能很高，通过索引下标的操作中间的节点性能会较差。



lpush list v1 v2 v3...-从左段将值压入列表

rpush list v1 v2 v3...-从右侧将值压入列表

lrange list left right-从左向右依次读取下标为left到right的value

lpop list-弹出list中左侧第一个元素

rpop list-弹出list中右侧第一个元素

lindex list index-返回从左向右下标为index的元素

llen list-获取list的元素个数

lrem list number value-从list中删除number个value（redis的list中的东西是可以重复的）

ltrim list left right-只保留list中从左向右下标为left到right的值

rpoplpush 源列表 目的列表-将源列表中的右侧第一个元素添加到目的列表的左侧

linsert list before/after v1 v2-在list中v1前/后添加v2

### 3.Hash

换做java的结构就是：Map<String,Map<Object,Object>>

hset hash k1 v1 k2 v2....-创建哈希列表

```
127.0.0.1:6379> hset user:001 name zhangsan age 13
(integer) 2
```

hget hash k1-获取hash中k1所对应的值

```
127.0.0.1:6379> hget user:001 age
"13"
```

hmget hash k1 k2....-获取hash中多个key对应的值：

```
127.0.0.1:6379> hget user:001 age name
(error) ERR wrong number of arguments for 'hget' command
127.0.0.1:6379> hmget user:001 age name
1) "13"
2) "zhangsan"
```

hgetall hash-依次输出hash中所有的key-value

hdel hash key-删除hash中的key以及所对应的value

hexists hash key-查看hash中有没有key字段，有返回1，没有返回0

hkeys hash-显示出hash中的所有key

hvals hash-显示出hash中的所有value

hincrby hash key number-让hash中key对应的value加number

hincrbyfloat hash key number-让hash中key对应的value加小数number

hsetnx hash k1 v1 k2 v2...-当不存在时创建

使用案例：中小厂的购物车功能

## 4.集合Set

set集合中的元素会自动去重

sadd set v1 v2 v3...-创建一个集合并添加元素v1v2v3...

```
127.0.0.1:6379> sadd set1 1 1 2 2 3 3 4 4  
(integer) 4
```

smembers set-遍历set中的元素：

```
127.0.0.1:6379> smembers set1  
1) "1"  
2) "2"  
3) "3"  
4) "4"
```

sismember set number-判断set中有没有number，有返回1，没有返回0

```
127.0.0.1:6379> sismember set1 5  
(integer) 0  
127.0.0.1:6379> sismember set1 1  
(integer) 1
```

srem set v1 v2...-删除set中的v1v2....

scard set-获取set中的元素个数

randmember set number-从set中随机抽取number个元素展示，如果set的元素个数不足number则全部展示

spop set number-从set中随机弹出number个数字，并删除

smove set1 set2 number-将set1中存在的number移动到set2中

### 集合运算

我们先初始化集合：

```
127.0.0.1:6379> sadd A a b c 1 2  
(integer) 5  
127.0.0.1:6379> sadd B 1 2 3 a x  
(integer) 5
```

sdiff set1 set2-找出属于set1但不属于set2的元素

sunion set1 set2-set1和set2取并集并输出

sinter set1 set2-找出同时在set1和set2的元素

sintercard keynumbers k1 k2.... limit number-输出同时在keynumbers个key(k1,k2,k3.....)中的元素的个数，最大输出number。

**运用案例：**

社交关系

## 5.有序集合ZSet

zset就是在set基础上在每一个value之前增加一个score值，变为score1 v1 score2 v2...

zadd zset score1 v1 score2 v2.....-创建zset并添加权重和value

zrange zset left right-按照权重从小到大返回下标left到right中的元素

zrange zset left right withscores-按照权重从小到大返回下标left到right中的元素，同时返回分数，顺序为v1 score1 v2 score2.....

zrevrange-同zrange，只不过这个是按照权重从大到小

7.0新命令zrangebyscore zset min max [withscore] [limit start count]-获取zset中分数在min到max中的元素[带有分数] [起始下标为start，走count步，也就是输出count-start+1个元素]

zscore zset value-获取zset中value的对应权重

zcard zset-获取zset中元素的个数

zrem zset value-删除zset中的value以及权重

zincrby zset number value-将zset中的value对应的权重增加number

zcount zset min max-获取zset中分数在min到max中元素的个数

7.0新命令zmpop keynumber zset1 zset2.... min/max count number-从keynumber个zset中弹出最大/最小的number个元素

zrank zset value-获取value在zset中的下标

zrevrank zset value-获取value在zset中的逆序下标

**运用案例**

根据商品销售进行排序显示

## 6.地理空间GEO

为什么有这个呢？直接用数据库储存经纬度然后每次进行查询计算不行吗？不行的，在我们的下班高峰期，请求的数量是很恐怖的，如果一瞬间全部打到数据库里，数据库会崩溃的。

geoadd geo x1 y1 name1 x2 y2 name2.....-新建geo并添加多个经纬度地点

```
127.0.0.1:6379> GEOADD city 116.403963 39.915119 "天安门" 116.403414 39.924091 "故宫" 116.024067 40.362639 "长城"
(integer) 3
```

诶，我们看到geo的格式是不是类似于分数+value，那我们大胆猜测一下底层是什么？zset

```
127.0.0.1:6379> type city
zset
```

那我们就可以用zset的命令进行遍历，使用zrange:

```
127.0.0.1:6379> zrange city 0 -1
1) "\xe5\xa4\xa9\xe5\xae\x89\xe9\x97\xa8"
2) "\xe6\x95\x85\xe5\xae\xab"
3) "\xe9\x95\xbf\xe5\x9f\x8e"
```

我们发现全是乱码，怎么解决呢？

我们可以在进入redis客户端的时候加上--raw:

```
[root@centos myredis]# redis-cli -a 111 --raw
```

然后进行尝试:

```
127.0.0.1:6379> zrange city 0 -1
天安门
故宫
长城
```

拿下

geopos geo v1 v2...-返回geo中v1v2v3....的真实经纬度:

```
127.0.0.1:6379> geopos city 天安门 长城 故宫
116.40396326780319214
39.91511970338637383
116.02406591176986694
40.36263993239462167
116.40341609716415405
39.92409008156928252
```

经纬度小数位是不是太多了，眼花缭乱，我们希望显示的简单一点，那就将返回的坐标用geohash表示，用geohash算法生成base32编码值

geohash geo v1 v2 v3...-返回geo中v1v2v3...的32位编码值:

```
127.0.0.1:6379> geohash city 天安门 故宫 长城
wx4g0f6f2v0
wx4g0gfqsj0
wx4t85y1kt0
```

geodist geo v1 v2 m/km/ft/mi-返回geo中v1与v2之间的直线距离，单位是米/千米/英尺/英里

```
127.0.0.1:6379> geodist city 天安门 故宫 km
0.9988
```

georadius geo x y length m/km/ft/mi [withdist] [withcoord] count number [desc/asc]-返回geo中与xy经纬度相距不超过length+单位长度的number个地点，（withdist同时返回找到地点的经纬度），（withcoord同时返回找到地点的52位有符号数值），（desc按照距离降序排列，asc按照距离升序排列）

georadiusbymember与上面这个用处一样，只不过把x和y换成直接的名字，比如“天安门”。

## 7.基数统计HyperLogLog

去重统计功能的估计统计算法（误差大概0.81%），就是HyperLogLog

### Redis HyperLogLog

Redis 在 2.8.9 版本添加了 HyperLogLog 结构。

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。

在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近  $2^{64}$  个不同元素的基数。这和计算基数时，元素越多耗费内存就越多形成鲜明对比。

但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

pfadd pf v1 v2 v3...-创建并添加v1v2v3...到HyperLogLog

pfcount pf-返回去重基数统计结果

pfmerge pf1 pf2-将pf1和pf2合并成一个

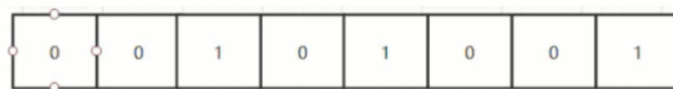
#### 运用案例：

统计一个网站的UV（独立访客数量，需要进行去重，通常一个ip被认为同一个访客）等的大规模的统计，比如天猫首页的亿级UV的redis统计方案。

## 8.位图bitmap

只能保存0和1的二进制bit数组

Bit arrays (or simply bitmaps. 我们可以称之为 位图)



一个字节(一个byte)=8位

上图由许许多多的小格子组成，每一个格子里面只能放1或者0，用它来判断Y/N状态说的专业点，每一个小格子就是一个bit

位图是利用String作为底层原理实现的结构，所以最大的容量为512M，而512M是 $2^{32}$ ，也就是说，使用位图仅仅512M就可以存储 $2^{32}$ 的字节信息。

setbit bitmap index 0/1-设置bitmap的下标为index的值为0/1

getbit bitmap index-获取bitmap中下标为index的值

strlen bitmap-查看bitmap占用多少字节

bitcount bitmap-统计bitmap中为1的个数

bitop and/or key bitmap1 bitmap2-将bitmap1和bitmap2的储存结果进行与运算/或运算，然后将结果赋值给key

#### 运用案例：

每日签到，任务点完成，钉钉打卡统计

## 9.位域bitfield

用的较少

## 10.流Stream

就是redis版本的MQ消息中间件+阻塞队列

sadd stream /id k1 v1 k2 v2.....-向stream队列中添加id为id(如果是就是系统自动生成，类似于mysql中的主键auto\_increment)的消息

```
127.0.0.1:6379> xadd mystream * sid 1 sname 张三
1737472662201-0
127.0.0.1:6379> xadd mystream * sid 2 sname 李四
1737472678124-0
127.0.0.1:6379> xadd mystream * sid 3 sname 王五
1737472689028-0
127.0.0.1:6379>
```

xrange stream - + [count number]-从stream中从最小值到最大值（取number个消息）显示出来

```
127.0.0.1:6379> xadd mystream * sid 1 sname 张三
1737472662201-0
127.0.0.1:6379> xadd mystream * sid 2 sname 李四
1737472678124-0
127.0.0.1:6379> xadd mystream * sid 3 sname 王五
1737472689028-0
127.0.0.1:6379>
```

```
127.0.0.1:6379> xrange mystream - + count 2
1737472662201-0
sid
1
sname
张三
1737472678124-0
sid
2
sname
李四
```

xrevrange stream + - [count number]-和上一个一样，只不过顺序相反

xdel stream id-删除stream中id为id的信息

xtrim stream maxlen number-从stream中截取id大的number个

xtrim stream minid number-将stream中id小于number的抛弃

xread [count number] streams stream 0-0 -从最小的id开始获取stream中的消息（0-0可以用0/00/000代替），（最多返回number个）

xread [count number] streams stream \$ -返回stream中大于最大id的消息（这个听起来有问题，但当队列为阻塞时，这个瞬间的id可能不是所有消息的最大id，可能还有更大的没有进入），\$表示当前消息中截取的最大消息id



xread [count number] block milliseconds streams stream \$/0 -以阻塞形式来读取消息，milliseconds是阻塞时间，如果为0则永远阻塞

```
127.0.0.1:6379> xread count 1 block 0 streams mystream $
```

这个意思就是监听比当前最大的id还要大的id，截取1个，block 0就代表一直监听，监听不到就耗着。

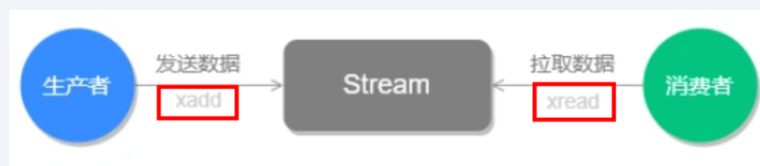
然后我们新开一个cli进行xadd操作

```
127.0.0.1:6379> xadd mystream * sid 6 sname 被监听者  
1737474292451-0
```

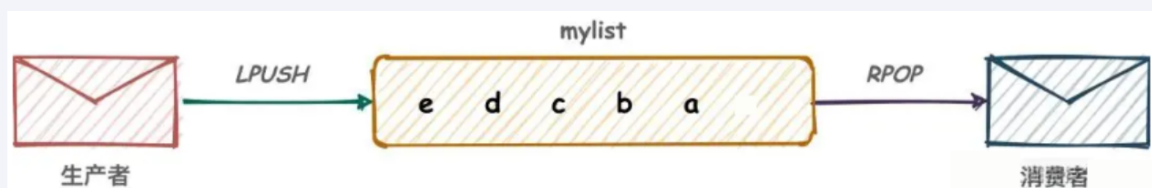
一瞬间，我们的另一个监听窗口弹出了结果：

```
127.0.0.1:6379> xread count 1 block 0 streams mystream $  
mystream  
1737474292451-0  
sid  
6  
sname  
被监听者
```

总结一下，生产者使用xadd发送消息到stream，消费者使用xread来拉取数据，实现简易版的消息队列



对比List结构



2025.1.22

## 5.redis持久化

持久化就是将我们内存中的数据写进磁盘当中

持久化有两种方式：

### RDB

又称为持久化快照，以指定时间来拍摄数据库的数据快照，就是每隔一段时间，就将数据库中的数据以文件的形式储存在磁盘上，这样发生宕机时也不会丢失大部分数据。

rdb保存在磁盘中的文件就是dump.rdb

关于快照的拍摄时间，redis6和redis7是不一样的。

rdb快照的触发分为两种：

## 自动触发

默认的条件：

```
426 # Unless specified otherwise, by default Redis will save the DB:
427 # * After 3600 seconds (an hour) if at least 1 change was performed
428 # * After 300 seconds (5 minutes) if at least 100 changes were performed
429 # * After 60 seconds if at least 10000 changes were performed
```

save a b的意思是：距离上次更新过了a时间，或者在a时间内更新了b次都会触发rdb保存

这里我们调整为距离上次更新过了5秒或者5秒内有2次修改就触发：

```
434 save 5 2
```

默认的储存地址：

```
497 # The working directory.
498 #
499 # The DB will be written inside this directory, with the filename specified
500 # above using the 'dbfilename' configuration directive.
501 #
502 # The Append Only File will also be created inside this directory.
503 #
504 # Note that you must specify a directory here, not a file name.
505 dir ./
```

我们指定一个位置（这个文件夹需要提前建好）：

```
499 # The DB will be written inside this directory, with the filename specified
500 # above using the 'dbfilename' configuration directive.
501 #
502 # The Append Only File will also be created inside this directory.
503 #
504 # Note that you must specify a directory here, not a file name.
505 dir /export/server/myredis/dumpfile
```

默认的rdb文件名：

```
482 dbfilename dump.rdb
```

我们修改为：

```
482 dbfilename dump6379.rdb
```

重启我们的redis

怎么判断我们的配置文件是否生效呢？我们可以使用config get xxx来获取配置文件中的某个信息：

```
127.0.0.1:6379> config get requirepass
requirepass
111
```

成功获取密码，则生效成功

在我们还没有进行操作时，这里面没有文件：

```
[root@centos dumpfile]# ll
总用量 0
```

然后我们进行一次修改，仍然没有文件：

```
127.0.0.1:6379> set k1 v1
OK
[root@centos dumpfile]# ll
总用量 0
```

然后我们5秒内修改两次，出现dump文件：

```
127.0.0.1:6379> set k1 v2
OK
127.0.0.1:6379> set k2 v1
OK
[root@centos dumpfile]# ll
总用量 4
```

那这个文件有什么用呢？可以恢复我们的数据。

我们先将原先的文件备份：

```
[root@centos dumpfile]# ll
总用量 4
-rw-r--r--. 1 root root 107 1月 22 14:42 dump6379.rdb.bf
```

然后模拟服务器宕机，使用flushdb

```
127.0.0.1:6379> FLUSHDB
OK
127.0.0.1:6379> keys *
总用量 8
-rw-r--r--. 1 root root 88 1月 22 15:00 dump6379.rdb
-rw-r--r--. 1 root root 107 1月 22 14:42 dump6379.rdb.bf
```

我们会发现重新生成了一个dump文件，这个文件是空的，没意义，我们删除这个，然后将原来的dump文件恢复回来

然后我们关闭redis，再重启发现：

```
127.0.0.1:6379> keys *
k3
k4
k1
k2
```

我们的数据可以根据rdb文件恢复，持久化成功

在我们使用redis的时候，产生的rdb文件一定要进行备份并分机隔离储存

## 手动触发

rdb提供了两个命令来实现手动储存：**save**和**bgsave**（默认）

## 禁止rdb

有时候可能因为快照文件过大，在重启的时候会导致严重io操作，很吃性能，储存的时候也很耗费空间，所以有时候我们选择禁用rdb操作。

**配置文件禁止：**修改save后面的字符串为空串

```
421 # Snapshotting can be completely disabled with a single empty string argument
422 # as in following example:
423 #
424 # save ""
```

**命令禁止：**redis-cli config set save ""

**RDB优化参数：**

```

445 # However if you have setup your proper monitoring of the Redis server
446 # and persistence, you may want to disable this feature so that Redis will
447 # continue to work as usual even if there are problems with disk,
448 # permissions, and so forth.
449 stop-writes-on-bgsave-error yes

```

这个的意思就是当写入出错时是否要停止写入，默认为yes

```

451 # Compress string objects using LZF when dump .rdb databases?
452 # By default compression is enabled as it's almost always a win.
453 # If you want to save some CPU in the saving child set it to 'no' but
454 # the dataset will likely be bigger if you have compressible values or keys.
455 rdbcompression yes

```

这个如果配置成yes，rdb写入磁盘时会使用LZF算法进行压缩，会占用一些CPU性能，默认为yes

```

462 # RDB files created with checksum disabled have a checksum of zero that will
463 # tell the loading code to skip the check.
464 rdbchecksum yes

```

这个如果配置成yes，会在储存快照后，再次利用CRC64算法进行数据校验，会损耗10%左右的CPU性能

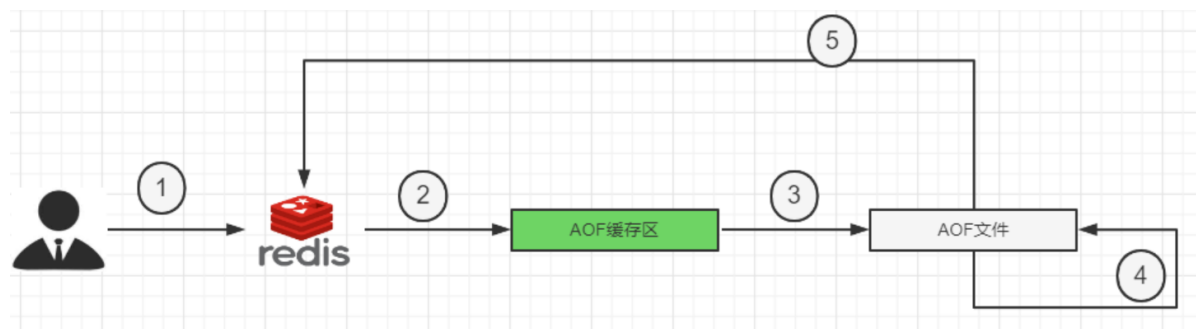
## AOF

又称为写操作复制

他会将所有写操作进行记录，当需要恢复数据的时候再从头到尾执行一遍

aof储存的文件是appendonly.aof

### AOF的工作流程



- 1.从客户端收到相应的写命令
- 2.将命令写到AOF的缓存区中，缓存区是位域内存中的，会等命令到达一定量后再写入文件中，这样可以避免大规模的io操作
- 3.根据缓存区的3种缓存策略将命令写入AOF文件
- 4.随着命令的增多，会根据规则进行AOF的重写，以达到压缩AOF文件的作用
- 5.当redis服务器重启的时候会利用AOF文件中的命令恢复redis数据

### AOF缓存的3中写会策略

#### always

同步写回，每个写命令执行完立刻同步地将日志写回磁盘

数据基本不丢失，但性能影响较大

#### everysec (默认)

每秒写回，每个写命令执行完后，先将命令写入缓存区，每隔1秒将缓存区的命令写入文件

性能适中，宕机时会丢失1秒的数据

**no**

将命令写进缓存区后就不管他了，让操作系统决定何时写进文件

性能好，但宕机时会丢失大量数据

## AOF正常恢复

### 1.开启AOF

AOF默认是关闭的，我们需要打开：

```
1367 # The Append Only File is an alternative persistence mode that provides
1368 # much better durability. For instance using the default data fsync policy
1369 # (see later in the config file) Redis can lose just one second of writes in a
1370 # dramatic event like a server power outage, or a single write if something
1371 # wrong with the Redis process itself happens, but the operating system is
1372 # still running correctly.
1373 #
1374 # AOF and RDB persistence can be enabled at the same time without problems.
1375 # If the AOF is enabled on startup Redis will load the AOF, that is the file
1376 # with the better durability guarantees.
1377 #
1378 # Please check https://redis.io/topics/persistence for more information.
1379
1380 appendonly no
```

### 2.确定写回策略

```
1438 # appendfsync always
1439 appendfsync everysec
1440 # appendfsync no
```

### 3.确定文件保存路径

在redis6，aof文件保存位置和rdb文件是一样的，都是通过配置文件修改dir配置

在redis7，aof会额外多一个配置appendonlyfilename "xxx"，同时在dir配置目录下新建一个名字叫xxx的文件夹，在文件夹中存放aof文件。

```
1407 appendfilename "appendonly.aof"
1408
1409 # For convenience, Redis stores all p
1410 # directory. The name of the director
1411 # configuration parameter.
1412
1413 appenddirname "appendonlydir"
```

所以我们修改为：

```
505 dir /export/server/myredis
506
```

### 4.aof文件的保存

redis6的aof文件是一整个，但是到了redis7，这个文件就被分成3个部分，这种模式被称为MP-AOF：

```
1403 # - appendonly.aof.1.base.rdb as a base file.
1404 # - appendonly.aof.1.incr.aof, appendonly.aof.2.incr.aof as incremental files.
1405 # - appendonly.aof.manifest as a manifest file.
```

这三个文件分别是base文件，incr文件，history文件。

incr文件是立即储存当前的所有写操作的文件。

base是将base中的写操作进行简化压缩后的写命令。

history这是redis用来储存历史版本的base和incr文件。

5.进行写操作测试：

```
127.0.0.1:6379> set k1 v1
OK
127.0.0.1:6379>
-rw-r--r--. 1 root root 88 1月 23 14:05 appendonly.aof.1.base.rdb
-rw-r--r--. 1 root root 52 1月 23 14:06 appendonly.aof.1.incr.aof
-rw-r--r--. 1 root root 88 1月 23 14:05 appendonly.aof.manifest
```

然后我们删除我们的rdb生成的文件（在flushdb和shutdown之后也要进行清除）：

```
[root@centos myredis]# ll
总用量 216
drwxr-xr-x. 2 root root    103 1月 23 14:05 appendonlydir
-rw-r--r--. 1 root root 106595 1月 23 14:05 redis7.conf
-rw-r--r--. 1 root root 106545 1月 19 10:27 redis.conf
```

备份一下我们的aof文件：

```
[root@centos myredis]# ll
总用量 216
drwxr-xr-x. 2 root root    103 1月 23 14:05 appendonlydir
drwxr-xr-x. 2 root root    103 1月 23 14:12 appendonlydir.bak
-rw-r--r--. 1 root root 106595 1月 23 14:05 redis7.conf
-rw-r--r--. 1 root root 106545 1月 19 10:27 redis.conf
```

然后我们重新启动之后发现是空的：

```
[root@centos myredis]# redis-server redis7.conf
[root@centos myredis]# redis-cli -a 111 --raw -p 6379
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
127.0.0.1:6379> keys *
127.0.0.1:6379>
```

因为我们的flushdb也是写操作，也就是我们的base.aof文件储存了flushdb这个命令并执行，那该怎么进行恢复呢？我们之前备份了一个aof文件，那个文件里面没有flushdb命令。这也就是为什么倡导我们要及时备份。

我们将备份文件恢复成appendonlydir之后再次启动发现：

```
[root@centos myredis]# redis-server redis7.conf
[root@centos myredis]# redis-cli -a 111 --raw -p 6379
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
127.0.0.1:6379> keys *
k1
```

则保存成功。

## 异常恢复

我们将生成的aof文件进行破坏：



```
FinalShell 4.5.12
1 redis001 x +
*2
$6
SELECT
$1
0
*3
$3
set
$2
k1
$2
v1
lahffioahiofhoawhdihalhsdholawhida
~
~
```

然后重启redis服务并尝试进入redis客户端:

```
[root@centos appendonlydir]# redis-cli -a 111
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
Could not connect to Redis at 127.0.0.1:6379: Connection refused
not connected>
```

还记得我们/usr/local/bin中有这样一个文件吗:

```
[root@centos ~]# cd /usr/local/bin/
[root@centos bin]# ll
总用量 21492
-rwxr-xr-x. 1 root root 5197848 12月 10 19:47 redis-benchmark
lrwxrwxrwx. 1 root root      12 12月 10 19:47 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 12月 10 19:47 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5411040 12月 10 19:47 redis-cli
lrwxrwxrwx. 1 root root      12 12月 10 19:47 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 11390192 12月 10 19:47 redis-server
```

这是aof的修复命令, 我们可以使用redis-check-aof aof文件名 来修复我们的文件:

```
[root@centos appendonlydir]# redis-check-aof appendonly.aof.1.incr.aof
Start checking Old-Style AOF
AOF appendonly.aof.1.incr.aof format error
AOF analyzed: filename=appendonly.aof.1.incr.aof, size=88, ok_up_to=52, ok_up_to_line=13, diff=36
AOF appendonly.aof.1.incr.aof is not valid. Use the --fix option to try fixing it.
```

别急, 我们再次打开intro文件:

```
*2
$6
SELECT
$1
0
*3
$3
set
$2
k1
$2
v1
lahffioahiofhoawhdihalhsdholawhida
```

这也妹修啊？因为我们没有加fix关键字，我们加上：

```
[root@centos appendonlydir]# redis-check-aof --fix appendonly.aof.1.incr.aof
Start checking Old-Style AOF
```

这样就将成功修复了aof文件。

### AOF的优势

- 1.有三种写入策略，一般使用的是每秒写入，宕机最差的结果就是丢失1s的数据，这是可以接受的
- 2.redis的incr文件会自动重写瘦身
- 3.aof文件是一个仅附加日志，不会出现管道问题，就算一个命令只写入了一半，也可以使用check命令来修复aof文件
- 4.aof文件的格式易于理解解析，同时可以轻松导出，就算不小心将flushdb等命令写进了aof文件，也可以使用vim进行手动去除。

### AOF的缺点

- 1.AOF文件即使瘦身，一般也比rdb文件的大小要大。
- 2.aof即使开启了每秒的写回策略，性能也是比较差的。

### AOF文件的重写机制

#### 自动触发

在redis的配置文件中写了关于redis的重写触发：

```
1480 auto-aof-rewrite-percentage 100
1481 auto-aof-rewrite-min-size 64mb
```

有两个条件：第一个就是触发aof储存后的新的文件对比上一次文件增加了100%，第二个就是aof文件的大小超过64mb。

重写前：

```
[root@zzyy appendonlydir]# ll
总用量 8
-rw-r--r-- 1 root root 0 11月 28 18:54 appendonly.aof.1.base.aof
-rw-r--r-- 1 root root 967 11月 28 18:57 appendonly.aof.1.incr.aof
-rw-r--r-- 1 root root 88 11月 28 18:54 appendonly.aof.manifest
```

incr文件的编号是1

重写后：

```
[root@zzyy appendonlydir]# ll
总用量 8
-rw-r--r-- 1 root root 126 11月 28 18:57 appendonly.aof.2.base.aof
-rw-r--r-- 1 root root 0 11月 28 18:57 appendonly.aof.2.incr.aof
-rw-r--r-- 1 root root 88 11月 28 18:57 appendonly.aof.manifest
```

incr文件内的命令重写进base文件，同时清除incr文件中的内容，随后将两个文件的编号变化为2

也就是对incr文件的重写其实不是对命令的重写，而是直接读取redis服务器中的数据键值，然后有一条命令进行储存，将这个命令储存起来，形成新的aof文件来代替原来那个aof文件。

### 手动触发

使用bgrewriteaof命令

## RDB与AOF同时开启

rdb是redis默认开启的，aof是默认关闭的，二者可以同时开启。

### 那同时开启了应该听谁的呢？

虽然rdb的性能高于aof，但是aof的数据一致性大于rdb，也就是rdb的数据更加新，所以还是以aof为主。

### 重启redis时数据读谁的呢？

当redis重启时，会优先读取aof文件中的数据，如果没有aof文件或aof文件损坏，会读取rdb文件数据，如果连rdb数据都没有，则启动失败。

### 那既然aof这么好，能不能只启用aof？

不建议，虽然aof的数据一致性高，但是aof是不断变化的，不利于备份数据库，如果要用aof备份数据库，那就要频繁地备份aof文件，很繁琐，而rdb快照文件是每隔一段时间才会更新一次，这个频率远小于aof文件更新频率。

### 怎么开启呢？

```
1507 # Redis can create append-only base files in either RDB or AOF formats. Using
1508 # the RDB format is always faster and more efficient, and disabling it is only
1509 # supported for backward compatibility purposes.
1510 aof-use-rdb-preamble yes
```

将这个配置开启就行（默认是开启）。

## 纯缓存模式

把rdb和aof同时关闭就行。

## 6.Redis事务

### redis的事务是什么？

redis事务是指一系列命令组成的有序命令组，里面的命令会被按顺序执行，同时其他命令既不会执行也不会插队。将所有命令放在一个队列中，一次性、顺序性、排他性执行。

#### redis事务与mysql事务的区别

1.redis事务执行过程中不会执行其他的命令，因为redis的架构是单线程的，也就没有事务的隔离级别，不存在脏读等问题，具有排他性。

2.redis事务不保证原子性，只保证是否开始执行全部命令的能力，不保证全部命令全部成功或全部失败，不具备执行一般进行回滚。

### redis事务常用命令

discard-放弃事务，取消事务中的所有指令

exec-执行所有事务块内的命令

multi-标记一个事务块的开始

unwatch-取消watch命令对key的监视

watch key1 key2.....监视一个或多个key，如果在事务开启之前这些key被其他命令改动过，则事务会被打断

### redis事务的运行

#### 1.正常运行

使用multi标记事务开头，使用exec标记并运行事务结尾：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> set k3 v3
QUEUED
127.0.0.1:6379(TX)> incr count
QUEUED
127.0.0.1:6379(TX)> exec
OK
OK
OK
1
```

使用multi后，会将后面的命令先放入一个队列中，先不执行，当收到exec命令后再一次性按顺序执行，然后按顺序返回结果。

## 2.放弃事务

使用multi开启事务后，中途使用discard来结束事务，不会执行事务中的任何语句：

```
127.0.0.1:6379> get count
1
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v2
QUEUED
127.0.0.1:6379(TX)> set k2 v3
QUEUED
127.0.0.1:6379(TX)> incr count
QUEUED
127.0.0.1:6379(TX)> discard
OK
127.0.0.1:6379> get count
1
```

我们可以看到，count的值没有进行改变

## 3.全体连坐

在multi开启事务后，写入错误的语句会导致所有事务无法执行：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v2
QUEUED
127.0.0.1:6379(TX)> set k2 v3
QUEUED
127.0.0.1:6379(TX)> set k3
ERR wrong number of arguments for 'set' command

127.0.0.1:6379(TX)> set k3 v4
QUEUED
127.0.0.1:6379(TX)> exec
EXECABORT Transaction discarded because of previous errors.
```

我们发现即使在后面跟着正常的对k3的操作语句，这个事务仍然无法执行。

这种错误在写入队列时就已经提示出来了，那如果有些命令在运行过程中才会出错的该怎么办呢？这就要提到冤头债主了。

## 4.冤头债主

在上一个事务案例中，出错后所有的事务都不会进行，但是当错误是RuntimeException时，redis在命令入队时不会进行提示，因为还没有运行，运行后其他正常的命令会执行，不正常的命令会出错，不会影响其他命令的进行，数据库也不会进行回滚。

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v111
QUEUED
127.0.0.1:6379(TX)> set k2 v222
QUEUED
127.0.0.1:6379(TX)> incr count
QUEUED
127.0.0.1:6379(TX)> incr email
QUEUED
127.0.0.1:6379(TX)> set k3 v333
QUEUED
127.0.0.1:6379(TX)> exec
OK
OK
2
ERR value is not an integer or out of range
OK
```

我们可以看到，只有错误的那一个命令对应的结果报错，我们尝试获取一下其他的key：

```
127.0.0.1:6379> get k1
v111
127.0.0.1:6379> get k2
v222
127.0.0.1:6379> get k3
v333
127.0.0.1:6379> get count
2
```

可以看到其他的命令是正常执行的。

## 5.watch监控

redis使用watch来提供乐观锁定，类似于CAS

那什么是乐观锁，什么是悲观锁，什么是CAS呢？

### 乐观锁

很乐观，每次拿数据都会认为别人没有修改数据，不会上锁，但每次更新数据时会判断数据有没有被别人更新过，当提交版本大于当前数据的版本时才会更新。redis就是用的这种思想。

### 悲观锁

顾名思义，很悲观，每次去拿数据都会以为别人会修改，所以每次去拿数据都会给数据加锁，将其他人对这个数据的操作block直到自己操作结束。这样数据安全了，但并发性能会下降，而redis是高速缓存，所以redis不会用这种方式

### CAS



全程check-and-set，就是在更新数据的时候会对数据版本进行判断，只有手上的数据版本大于服务器中数据的版本时才会执行更新，否则打断事务。

## 实战案例

### 正常情况

我们只开一个窗口，就不存在加塞修改等问题，我们目前有这两个键值：

```
127.0.0.1:6379> get k1
abc
127.0.0.1:6379> get balance
100
```

然后我们监控balance，开启事务：

```
127.0.0.1:6379> watch balance
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 abc2
QUEUED
127.0.0.1:6379(TX)> set balance 110
QUEUED
127.0.0.1:6379(TX)> exec
OK
OK
127.0.0.1:6379> get k1
abc2
127.0.0.1:6379> get balance
110
```

一切顺利

### 异常情况

我们开启两个串口，监听balance，在exec之前用另一个窗口将balance进行修改：

```
127.0.0.1:6379> watch balance
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 abc3
QUEUED
127.0.0.1:6379(TX)> set balance 150
QUEUED
127.0.0.1:6379(TX)> exec
127.0.0.1:6379> get k1
abc2
127.0.0.1:6379> get balance
120
127.0.0.1:6379> 
```

FinalShell 4.5.12

1 redis001 x +

```
127.0.0.1:6379> set balance 120
OK
127.0.0.1:6379> 
```

我么发现exec的输出结果是null，也就是事务被打断

这时候我们得到的k1不会进行变化，balance也不会根据事务命令进行变化，相当于这个事务不存在过。