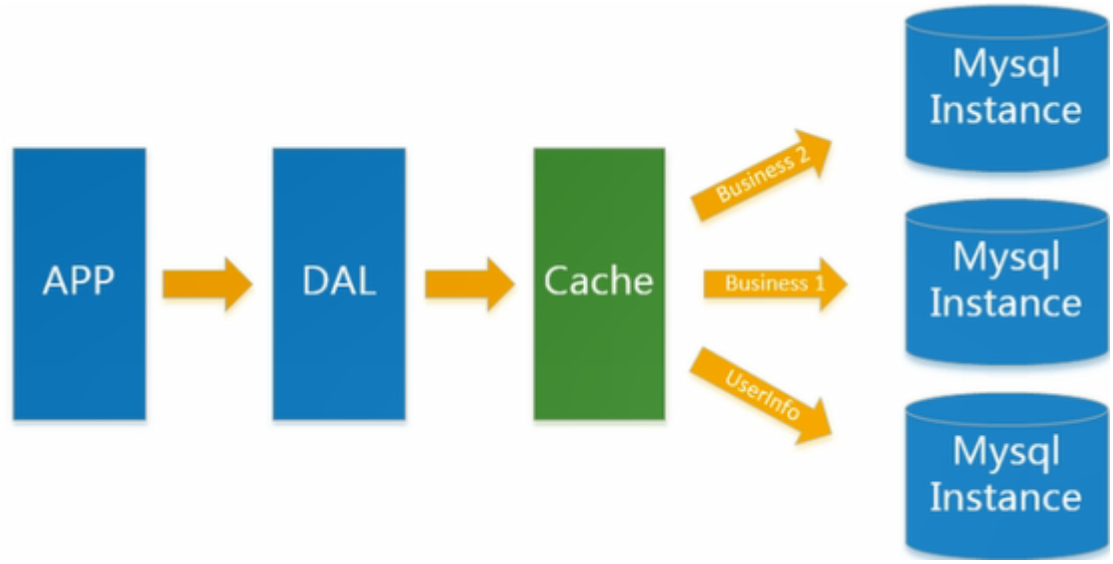
前言

为什么在传统的SQL数据库的背景下会出现NoSql数据库

在90年代，一个网站的访问路一般都不大，用单个数据库完全可以轻松应付。

但随着互联网时代的发展，简单的系统架构已无力承受大流量的访问，网站架构也日益复杂，目的就是解决大流量的访问。

在《我在淘宝这十年》一书中提到，早期的网站的架构也是从简单到LAMP架构模式演变到使用**Oracle、搜索引擎、分库分表，主从库**来减缓对数据库的压力，要知道即使是在牛逼的数据库面对每日几千万的访问量也是难以支撑的，所以避免多次对数据库进行访问是提高系统性能的手段之一。



并且有些业务当中往往在需要重复在访问数据库中的同个字段数据，因为这种情况，那么我们是否可以将这个字段数据提取出来放在缓存里，访问时不必再次去从数据库中拿而是从缓存中拿。

NoSQL一般为内存数据库，其优点是**性能高、读写数据快**，正因为这个原因所以非常适合作为缓存，而其中Redis就是当中的明星角色。

NoSQL数据库的四大分类

- KV键值，代表产品有：Memecache、redis
- 文档型数据库，代表产品有：CouchDB、MongoDB
- 列存储数据库，代表产品有：Cassandra、HBase
- 图关系数据库，代表产品有：Neo4J、InfoGrid

根据数据类型或关系选择合适的NoSQL数据库。

Redis

简介

Redis:REmote DIctionary Server(远程字典服务器)。

是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的(key/value)分布式内存数据库，基于内 存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一,也被人们称为数据结构服务器。

Redis的优点

Redis 与其他 key - value 缓存产品有以下三个特点：

1. Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用
2. Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储
3. Redis支持数据的备份，即master-slave模式的数据备份

Linux环境下载Redis

由于企业里面做Redis开发，99%都是Linux版的运用和安装，几乎不会涉及到Windows版

注意

在make编译时之前需要Linux系统需要安装gcc环境，可通过yum联网下载配置`yum install gcc-c++` **下载地址：**http://redis.io/download，下载最新稳定版本。

1. **下载>>>解压>>>编译**

```
[root@localhost]# wget http://download.redis.io/releases/redis-2.8.17.tar.gz#联网下载
[root@localhost]# tar xzf redis-2.8.17.tar.gz                    #解压到当前目录
[root@localhost]# cd redis-2.8.17                                #进入解压目录
[root@localhost redis]# make                                     #编译
```

2. **添加到环境变量**

   为了使用方便，我们需要将这个几个文件加到/usr/local/bin目录下去。这个目录在Path下面的话，就可以直接执行这几个命令了。

```
  [root@localhost redis]# make install
  cd src && make install
```

3. **运行redis服务**

   make install完后/usr/local/bin目录下会出现编译后的redis服务程序redis-server,还有用于测试的客户端程序redis-cli,两个程序位于安装目录 src 目录下（也就是解压目录xi）：

- 使用默认配置文件

```
$ cd src
$ ./redis-server
```

- 使用指定配置文件

```
$ cd src
$ ./redis-server redis.conf配置文件路径
```

4. **启动Redis客户端**

   类似于启动Mysql服务后，需要有一个客户端与服务端进行交互。

```
$ cd src             #进入安装目录
$ ./redis-cli        #启动redis客户端
redis> set foo bar   #添加数据
OK                   #操作提示
redis> get foo       #获取数据
"bar"                #数据
```

   用法

   redis-cli [OPTIONS][cmd [arg [arg ...]]

   -h <主机ip>，默认是127.0.0.1

   -p <端口>，默认是6379

   -a <密码>，如果redis加锁，需要传递密码

   --help，显示帮助信息

5. **关闭服务**
   - 关闭单实例运行客户端连接的redis服务

```
$ redis-cli shutdown
```

   - 关闭指定端口上运行的客户端连接的redis服务

```
$ redis-cli -p 6379 shutdown
```

6. **退出**

```
exit
```

Redis安装目录

在Linux下默认的安装目录在usr/local/bin下，其中reids的安装目录主要有：

CPA理论

关系型数据库遵循的是ACID规则，即A（原子性）、C（一致性）、I（隔离性）、D（持久性）。

那么在非关系型数据库中当然也有自己的规则，那就是CAP，即C（强一致性）、P（可用性）、A（分布容错性）。 不过在关系型数据库中必须遵行ACID所有规则，但在非关系型数据库中却只能进行3选2，也就是只能满足CA、CP、AP。 CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的。所以我们只能在一致性和可用性之间进行权衡，没有NoSQL系统能同时保证这三点。

> 一些分布式系统对CPA的选择

CA： 传统Oracle数据库

AP：大多数网站架构的选择

CP：Redis、Mongodb

分析

分布式架构的时候必须做出取舍。一致性和可用性之间取一个平衡。多余大多数web应用，其实并不需要强一致性（实现强一致性必定会损耗系统整体性能）。因此牺牲C换取P，这是目前分布式数据库产品的方向。

而Redis往往作为缓存数据库，所以对于数据安全性比较低，从而导致在数据的一致性也相对较差一些。 但对于传统Oracle数据库而言，作为一款磁盘存储数据库，对于每一次写操作都保存在磁盘当中，所以数据的安全性和一致性比较高。 对于web2.0网站来说，关系数据库的很多主要特性却往往无用武之地。 比如数据库事务一致性需求。 很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低， 有些场合对写一致性要求并不高。允许实现最终一致性（对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之 后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的）。

> Base

BASE就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案。 BASE其实是下面三个术语的缩写：

1.基本可用（Basically Available） 2.软状态（Soft state） 3.最终一致（Eventually consistent） 它的思想是通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观。为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法

> 分布式和集群的概念

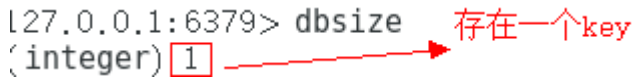
- 分布式：不同的多台服务器上面部署**不同的服务模块（工程）**，他们之间通过Rpc/Rmi之间通信和调用，对外提供服务和组内协作。
- 集群：不同的多台服务器上面部署**相同的服务模块**，通过分布式调度软件进行统一的调度，对外提供服务和访问。

基本知识

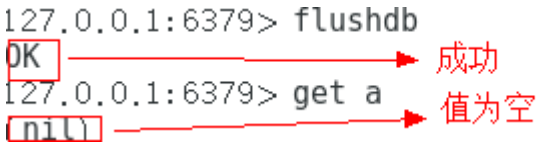
1. Redis默认有16个数据库，默认使用零号数据库
2. **select**命令切换数据库，如：

```
select 0            #切换到零号数据库
```

3. **dbsize**命令查看当前数据库的key的数量



4. **flushdb**命令清空当前库数据，包括key和value



5. **flushall**命令删除所有库数据，包括key和value，小心使用

```
127.0.0.1:6379> flush all
(error) ERR unknown command 'flush'
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> get a
(nil)
```

配置文件解析

units（单位）

```
# 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.
```

介绍配置的单位

1 . 配置大小单位,开头定义了一些基本的度量单位，只支持bytes，不支持bit。 2 . 对大小写不敏感，也就是1GB与1gb和1gB都一样。

INCLUDES（包含）

```
################################## INCLUDES ###################################

# Include one or more other config files here.  This is useful if you
# have a standard template that goes to all Redis server but also need
# to customize a few per-server settings.  Include files can include
# other files, so use this wisely.
#
# Notice 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.
#
# If instead you are interested in using includes to override configuration
# options, it is better to use include as the last line.
#
# include /path/to/local.conf
# include /path/to/other.conf
```

类似于在Spring中为主配置文件包含其它配置文件，通过 `include redis配置文件文件位置` 进行包含。

GENERAL（通用）

```
################################ 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.
daemonize no

# When running daemonized, Redis writes a pid file in /var/run/redis.pid by
# default. You can specify a custom pid file location here.
pidfile /var/run/redis.pid

# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket.
port 6379

# TCP listen() backlog.
#
# In high requests-per-second environments you need an high backlog in order
# to avoid slow clients connections issues. Note that the Linux kernel
# will silently truncate it to the value of /proc/sys/net/core/somaxconn so
# make sure to raise both the value of somaxconn and tcp_max_syn_backlog
# in order to get the desired effect.
tcp-backlog 511
```

主要配置通用设置有：

1. debug
2. verbose
3. notice
4. warning

SNAPSHOTTING（快照）

```
################################ SNAPSHOTTING  ################################
#
# Save the DB on disk:
#
#   save <seconds> <changes>
#
#   Will save the DB if both the given number of seconds and the given
#   number of write operations against the DB occurred.
#
#   In the example below the behaviour will be to save:
#   after 900 sec (15 min) if at least 1 key changed
#   after 300 sec (5 min) if at least 10 keys changed
#   after 60 sec if at least 10000 keys changed
#
#   Note: you can disable saving at all commenting all the "save" lines.
#
#   It is also possible to remove all the previously configured save
#   points by adding a save directive with a single empty string argument
#   like in the following example:
#
#   save ""

save 900 1
save 300 10
                                                              143,11        15%
```

RDB持久化方式的配置。

1. save 900 1 如果在900秒之内有1次操作的话，那么将在第900秒进行持久化保存
2. save 300 10 如果在300秒之内有10次操作的话，那么将在第300秒进行持久化保存
3. 60 10000 如果在60 秒之内有10000次操作的话，那么将在第60秒进行持久化保存

   若想关闭rdb持久化的话，save可写“”或不写save。

REPLICATION（复制）

SECURITY（安全）

```
################################## SECURITY ###################################

# Require clients to issue AUTH <PASSWORD> before processing any other
# commands.  This might be useful in environments in which you do not trust
# others with access to the host running redis-server.
#
# This should stay commented out for backward compatibility and because most
# people do not need auth (e.g. they run their own servers).
#
# Warning: since Redis is pretty fast an outside user can try up to
# 150k passwords per second against a good box. This means that you should
# use a very strong password otherwise it will be very easy to break.
#
# requirepass foobared

# Command renaming.
#
# It is possible to change the name of dangerous commands in a shared
# environment. For instance the CONFIG command may be renamed into something
# hard to guess so that it will still be available for internal-use tools
```

Redis的安全说明，若设置redis密码的话，那么在操作redis之前需要进行密码验证。

密码设置

```
config set requirepass "密码"
```

密码验证

```
auth 密码
```

```
127.0.0.1:6379> config set requirepass "123456"
OK
127.0.0.1:6379> ping
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth 123456
OK
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

设置密码

操作之前需要进行密码验证

密码验证

LIMITS（限制）

```
################################## LIMITS ####################################

# Set the max number of connected clients at the same time. By default
# this limit is set to 10000 clients, 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).
#
# Once the limit is reached Redis will close all the new connections sending
# an error 'max number of clients reached'.
#
# maxclients 10000

# Don't use more memory than the specified amount of bytes.
# When the memory limit is reached Redis will try to remove keys
# according to the eviction policy selected (see maxmemory-policy).
#
# If Redis can't remove keys according to the policy, or if the policy is
# set to 'noeviction', Redis will start to reply with errors to commands
# that would use more memory, like SET, LPUSH, and so on, and will continue
# to reply to read-only commands like GET.
#
                                                              366,37        48%
```

设置客户端连接的限制和Redis在内存中数据量的大小与内存数据移除的策略。。。主要有以下四种配置。

1. volatile-lru：使用LRU算法移除key，只对设置了过期时间的键
2. allkeys-lru：使用LRU算法移除key
3. volatile-random：在过期集合中移除随机的key，只对设置了过期时间的键
4. allkeys-random：移除随机的key
5. volatile-ttl：移除那些TTL值最小的key，即那些最近要过期的key
6. noeviction：不进行移除。针对写操作，只是返回错误信息

APPEND ONLY MODE（追加）

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

# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes in a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check http://redis.io/topics/persistence for more information.

appendonly no

# The name of the append only file (default: "appendonly.aof")

appendfilename "appendonly.aof"
                                                              462,31        57%
```

若开启AOF方式进行持久化，可修改默认的配置。

- Always：每次发生数据变更会被立即记录到磁盘，由于会多次进行IO操作，从而导致性能下降，但数据完整性比较好
- Everysec：出厂默认推荐，异步操作，每秒记录，如果一秒内出现问题，就会出现数据丢失。
- NO：不进行数据保存同步

数据类型操作

完整数据类型操作命令

http://redisdoc.com/

常见数据类型操作命令

Key（键）

> keys （查询当前库所有key名）

- 查询所有

```
keys *
```

```
127.0.0.1:6379> set key1 value1
OK
127.0.0.1:6379> set key2 value2
OK
127.0.0.1:6379> keys *
1) "key2"
2) "key1"
```

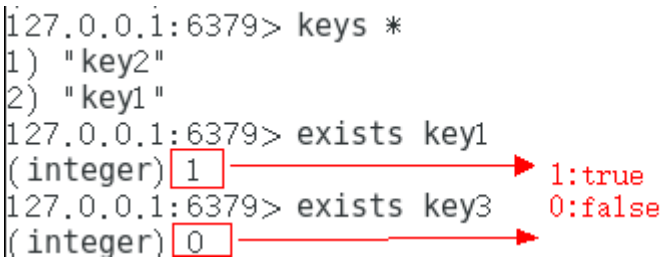
- 匹配查询（类似SQL查询中使用'?'占位符查询）

```
keys key?
```

```
127.0.0.1:6379> keys key?
1) "key2"
2) "key1"
```

> exists （判断是否存在某个key）

```
exists key名
```

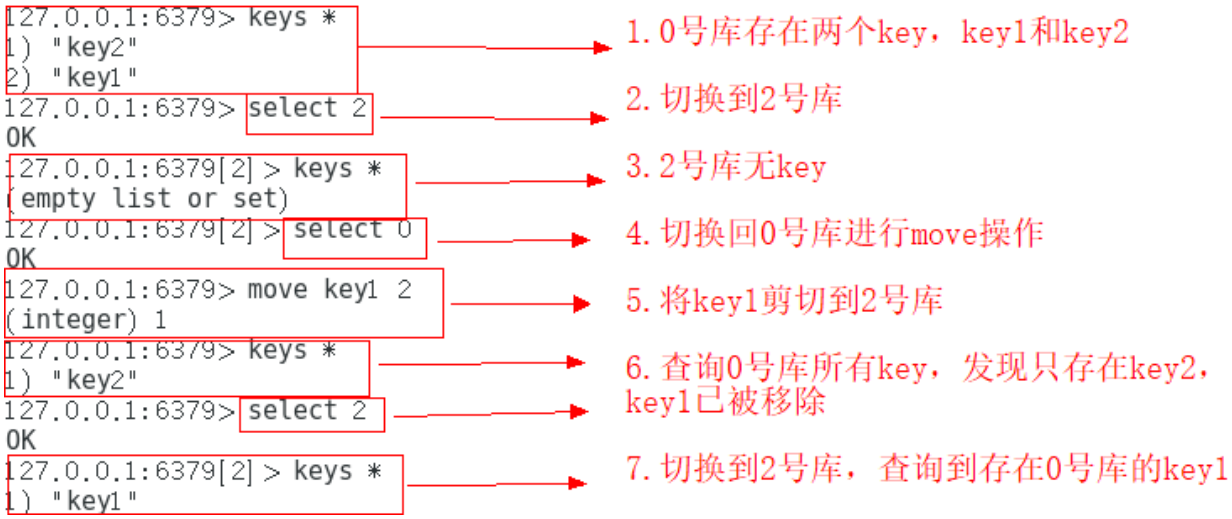


> del（删除指定key名的数据）

```
del key名
```

> move（将某个key和value剪切到指定库）

```
move key名 db索引
```



> type （查看key是什么类型）

```
type key名
```

```
127.0.0.1:6379[2]> type key1
string
```

> expire（给key设置过期时间）

```
expire key名 秒钟
```

```
127.0.0.1:6379[2]> expire key1 10
(integer) 1
```

到期后，key和value都会从内存中移除

> ttl（查看key还有多少秒过期）

```
ttl key名
```

```
127.0.0.1:6379[2]> ttl key1
(integer) -1
```

返回值表示

String（字符串）

说明

string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。 string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。 string类型是Redis最基本的数据类型，一个redis中字符串value最多可以是512M。

增删改查

- set（设值）

```
set key名 value值
```

若之前存在该key名，那么将覆盖原来值

- get（查询指定key名的value值）

```
get key名
```

- append（在指定key名对应的string数据后面追加数据，成功返回字符串长度）

```
append key名 追加数据
```

- strlen（查询指定key名数据的字符串长度）

```
strlen key名
```

数字字符加减

- incr（+1）

```
incr key名
```

- decr（-1）

```
decr key名
```

- incrby（+X）

```
incrby key名 加数
```

- decrby（-X）

```
decrby key名 减数
```

注意：一定得是数字字符串数据才能加减

范围操作数据

- getrange（获取指定范围字符串）

```
getrange key名 起始位置 结尾位置
```

```
127.0.0.1:6379[2]> getrange key2 0 1
"va"
127.0.0.1:6379[2]> get key2
"value"
127.0.0.1:6379[2]> getrange key2 0 1
"va"
```

- setrange（在指定位置开始插入数据）

```
setrange key名 插入位置 插入数据
```

```
127.0.0.1:6379[2]> get key2
"key2_va"
127.0.0.1:6379[2]> setrange key2 6 lue
(integer) 9
127.0.0.1:6379[2]> get key2
"key2_vlue"
```

多数据操作

- mset（同时设多个值）

```
mset key1名 value1值 key2名 value2值
```

```
127.0.0.1:6379[2]> mset key2 value2 key3 value3
OK
127.0.0.1:6379[2]> keys *
1) "key2"
2) "key3"
```

- mget（同时获取多个值）

```
mget key1名 key2名
```

```
127.0.0.1:6379[2]> mget key1 key2
1) (nil)
2) "value2"
```

多重操作

- setex（设值的同时设置过期时间）

```
setex key名 过期时间（秒） value值
```

```
127.0.0.1:6379[2] > setex key1 100 value1
OK
127.0.0.1:6379[2] > get key1
"value1"
127.0.0.1:6379[2] > ttl key1
(integer) 91
```

- setnx（设值之前会判断是否已存在该key，已存在将设值失败）

```
setnx key名 value值
```

- msetex（同理，设多值之前会先判断是否已存在key）

```
msetex key1名 value1值 key2名 value2值
```

先后操作

- getset（先get然后立即set）

```
getset key名 覆盖value值
```

```
127.0.0.1:6379[2] > getset key2 value
"value2"
127.0.0.1:6379[2] > get key2
"value"
```

List（列表）

说明

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。它的底层实际是个链表。

查

- lindex（查询指定索引的值）

```
lindex key名 索引
```

```
127.0.0.1:6379[2] > lrange key_list2 0 -1
1) "4"
2) "3"
3) "2"
127.0.0.1:6379[2] > lindex key_list2 0
"4"
```

- lrange（查询指定范围的列表数据）

```
lrange key名 起始位置 结尾位置
```

```
127.0.0.1:6379[2] > lrange list1 0 1
1) "1"
2) "2"
127.0.0.1:6379[2] > lrange list1 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

-1：表示无穷大，也就是0 -1将查询所有列表数据

- llen（查询列表数据量）

```
llen key名
```

```
127.0.0.1:6379[2] > lrange key_list2 0 -1
1) "4"
2) "3"
3) "2"
127.0.0.1:6379[2] > llen key_list2
(integer) 3
```

增

- lpush（向左追加数据）

```
127.0.0.1:6379[2] > lpush key_list2 1 2 3 4 5
(integer) 5
127.0.0.1:6379[2] > lrange key_list2 0 -1
1) "5"
2) "4"
3) "3"
4) "2"
5) "1"
```

- rpush（向右追加数据）

```
127.0.0.1:6379[2] > rpush key_list1 1 2 3 4 5
(integer) 5
127.0.0.1:6379[2] > lrange key_list1 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

- linsert（指定某数据前或后插入）

```
linsert key名 before|after 位置相对数据 待插入数据
```

```
127.0.0.1:6379[2]> lrange list 0 -1
1) "1"
2) "3"
127.0.0.1:6379[2]> linsert list  before 3 2
(integer) 3
127.0.0.1:6379[2]> lrange list 0 -1
1) "1"
2) "2"
3) "3"
```

注意对应方向：插入是从左向右，查询是从上往下

删

成功返回删除数据值

- lpop（删除左边第一个数据）

  ```
  lpop key名
  ```

  ```
  127.0.0.1:6379[2]> lrange key_list2 0 -1
  1) "5"
  2) "4"
  3) "3"
  4) "2"
  5) "1"
  127.0.0.1:6379[2]> lpop key_list2
  "5"
  127.0.0.1:6379[2]> lrange key_list2 0 -1
  1) "4"
  2) "3"
  3) "2"
  4) "1"
  ```

- rpop（删除右边第一个数据）

  ```
  127.0.0.1:6379[2]> lrange key_list2 0 -1
  1) "4"
  2) "3"
  3) "2"
  4) "1"
  127.0.0.1:6379[2]> rpop key_list2
  "1"
  127.0.0.1:6379[2]> lrange key_list2 0 -1
  1) "4"
  2) "3"
  3) "2"
  ```

- lrem （删除多个value）

  ```
  lrem key名 删除数量 待删除数据
  ```

```
127.0.0.1:6379[2]> rpush list 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
(integer) 15
127.0.0.1:6379[2]> lrange list 0 -1
 1) "1"
 2) "1"
 3) "1"
 4) "2"
 5) "2"
 6) "2"
 7) "3"
 8) "3"
 9) "3"
10) "4"
11) "4"
12) "4"
13) "5"
14) "5"
15) "5"
127.0.0.1:6379[2]> lrem list 3 1
(integer) 3
127.0.0.1:6379[2]> lrange list 0 -1
 1) "2"
 2) "2"
 3) "2"
 4) "3"
 5) "3"
```

改

- ltrim（截取指定范围数据）

```
ltrim key名 起始位置 结尾位置
```

```
127.0.0.1:6379[2]> lrange list 0 -1
 1) "2"
 2) "2"
 3) "2"
 4) "3"
 5) "3"
 6) "3"
 7) "4"
 8) "4"
 9) "4"
10) "5"
11) "5"
12) "5"
127.0.0.1:6379[2]> ltrim list 0 2
OK
127.0.0.1:6379[2]> lrange list 0 -1
1) "2"
2) "2"
3) "2"
```

- lset（指定位置修改数据）

```
lset key名 位置 新数据
```

```
127.0.0.1:6379[2]> lrange list 0 -1
1) "2"
2) "2"
3) "2"
127.0.0.1:6379[2]> lset list 0 1
OK
127.0.0.1:6379[2]> lrange list 0 -1
1) "1"
2) "2"
3) "2"
```

多重操作

- rpoplpush（移除列表的最后一个元素，并将该元素添加到另一个列表并返回）

```
127.0.0.1:6379[2]> lrange list 0 -1
1) "1"
2) "2"
127.0.0.1:6379[2]> lrange list1 0 -1
1) "3"
2) "1"
3) "2"
127.0.0.1:6379[2]> rpoplpush list1 list
"2"
127.0.0.1:6379[2]> lrange list 0 -1
1) "2"
2) "1"
3) "2"
127.0.0.1:6379[2]> lrange list1 0 -1
1) "3"
2) "1"
```

Set（无序不重复集合）

说明

Redis的Set是string类型的无序不重复集合。它是通过HashTable实现实现的。

查

- sismember（查询该元素是否存在指定key名中的集合中）

```
sismember key名 待查询元素
```

```
127.0.0.1:6379> smembers key_set
 1) "1"
 2) "2"
 3) "3"
 4) "4"
 5) "5"
 6) "6"
 7) "7"
 8) "8"
 9) "9"
10) "10"
127.0.0.1:6379> sismember key_set 1
(integer) 1
127.0.0.1:6379> sismember key_set 0
(integer) 0
```

存在返回1，不存在返回0

- scard（获取集合里面的元素个数）

```
scard key名
```

```
127.0.0.1:6379> smembers key_set
 1) "1"
 2) "2"
 3) "3"
 4) "4"
 5) "5"
 6) "6"
 7) "7"
 8) "8"
 9) "9"
10) "10"
127.0.0.1:6379> scard key_set
(integer) 10
```

- smembers（查询指定key集合中的所有元素）

```
smembers key名
```

```
127.0.0.1:6379> smembers key_set
 1) "1"
 2) "2"
 3) "3"
 4) "4"
 5) "5"
 6) "6"
 7) "7"
 8) "8"
 9) "9"
10) "10"
```

- srandmember（取出多少个随机元素）

```
srandmember key名 获取个数
```

```
127.0.0.1:6379> srandmember key_set 5
1) "2"
2) "7"
3) "9"
4) "6"
5) "8"
127.0.0.1:6379> srandmember key_set 5
1) "2"
2) "7"
3) "5"
4) "1"
5) "8"
127.0.0.1:6379> srandmember key_set 5
1) "2"
2) "7"
3) "1"
4) "4"
5) "8"
```

增

- sadd（添加元素到指定key中，成功返回添加元素个数）

```
sadd key名 value1 value2 value3
```

```
127.0.0.1:6379> sadd set 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
(integer) 5
127.0.0.1:6379> smembers set
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

注意set集合元素是不可以重复的

删

- srem（删除集合内的指定元素）

```
srem key名 待删除元素
```

```
127.0.0.1:6379> srem key_set 10
(integer) 1
127.0.0.1:6379> smembers key_set
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
```

- spop（随机删除某个元素）

```
spop key名
```

```
127.0.0.1:6379> smembers key_set
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
127.0.0.1:6379> spop key_set
"3"
127.0.0.1:6379> spop key_set
"2"
127.0.0.1:6379> smembers key_set
1) "1"
2) "4"
3) "5"
4) "6"
5) "7"
6) "8"
7) "9"
```

- smove（从当前集合剪切到其它集合）

```
smove 待剪切集合 目标集合 待剪切元素
```

```
127.0.0.1:6379> smembers key_set1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
127.0.0.1:6379> smembers key_set2
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
127.0.0.1:6379> smove key_set1 key_set2 6
(integer) 1
127.0.0.1:6379> smembers key_set1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
127.0.0.1:6379> smembers key_set2
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
```

数学逻辑运算

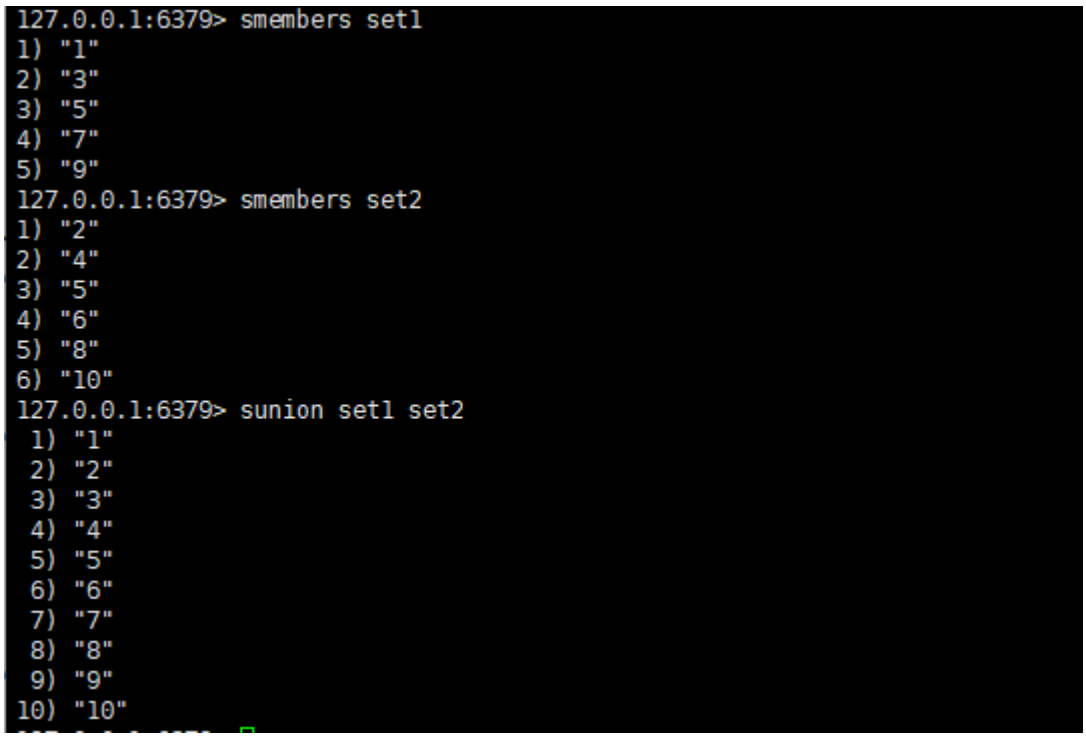
- sdiff（差集）

```
127.0.0.1:6379> smembers set1
1) "1"
2) "3"
3) "5"
4) "7"
5) "9"
127.0.0.1:6379> smembers set2
1) "2"
2) "4"
3) "5"
4) "6"
5) "8"
6) "10"
127.0.0.1:6379> sdiff set1 set2
1) "1"
2) "3"
3) "7"
4) "9"
127.0.0.1:6379> sdiff set2 set1
1) "2"
2) "4"
3) "6"
4) "8"
5) "10"
```

- sinter（交集）

```
127.0.0.1:6379> smembers set1
1) "1"
2) "3"
3) "5"
4) "7"
5) "9"
127.0.0.1:6379> smembers set2
1) "2"
2) "4"
3) "5"
4) "6"
5) "8"
6) "10"
127.0.0.1:6379> sinter set1 set2
1) "5"
```

- sunion（并集）



Hash（哈希）

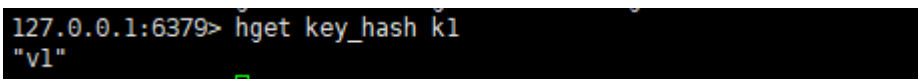
说明

Redis hash 是一个键值对集合。 Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。类似Java里面的Map<String,Object>

查

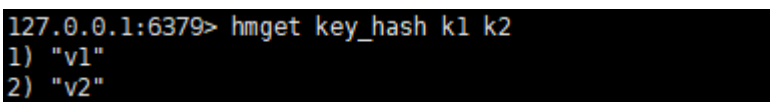
- hget（查询某个hash表的某个key的value）

```
hget redis缓存中的key 哈希表中的key
```



- hmget（查询多个hash表的某个key的value）

```
hget redis缓存中的key 哈希表中的key1 哈希表中的key2
```



- hgetall（获取全部kv（序号单数为K；双数有V））

```
hgetall redis缓存中的key
```

```
127.0.0.1:6379> hgetall key_hash
1) "k1"
2) "v1"
3) "k2"
4) "v2"
```

- hlen（查询集合元素个数）

```
hlen redis缓存中的key
```

```
127.0.0.1:6379> hgetall key_hash
1) "k1"
2) "v1"
3) "k2"
4) "v2"
127.0.0.1:6379> hlen key_hash
(integer) 2
```

- hexists（查询hash表中是否存在某个key）

```
hexists redis缓存中的key 待查询key
```

```
127.0.0.1:6379> hgetall key_hash
1) "k1"
2) "v1"
3) "k2"
4) "v2"
127.0.0.1:6379> hexists key_hash k1
(integer) 1
127.0.0.1:6379> hexists key_hash k3
(integer) 0
```

- hkeys（查询hash表内的所有key值）

```
hkeys redis缓存中的key
```

```
127.0.0.1:6379> hkeys key_hash
1) "k1"
2) "k2"
```

- hvals（查询hash表内的所有value值）

```
hvals redis缓存中的key
```

```
127.0.0.1:6379> hvals key_hash
1) "v1"
2) "v2"
```

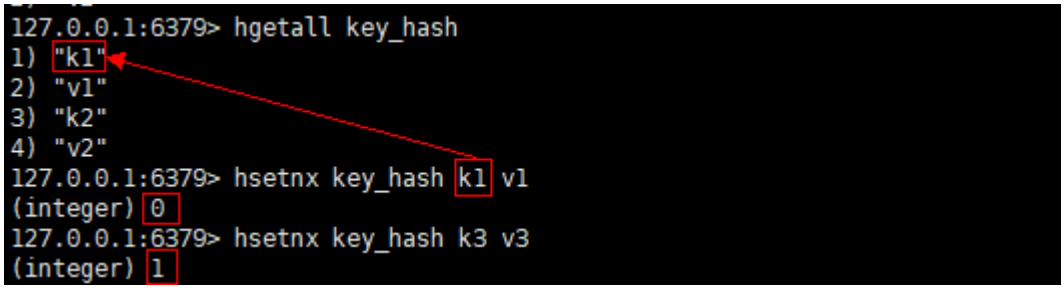
增

- hset（为指定缓存中某个key对应hash表添加一对KV数据，若该hash表中已存在key则会覆盖原先的value）

```
hset redis缓存中的key 哈希表中的key1 哈希表中的value1
```

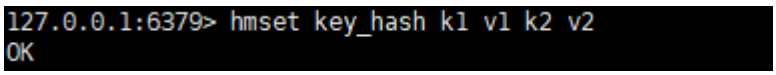
```
127.0.0.1:6379> hset key_hash k1 k2
(integer) 0
```

- hsetnx（同样为添加一对KV，但在添加之前会先判断是否已存在该key，存在的话将添加失败）



- hmset（为指定缓存中某个key对应hash表添加多对KV数据）

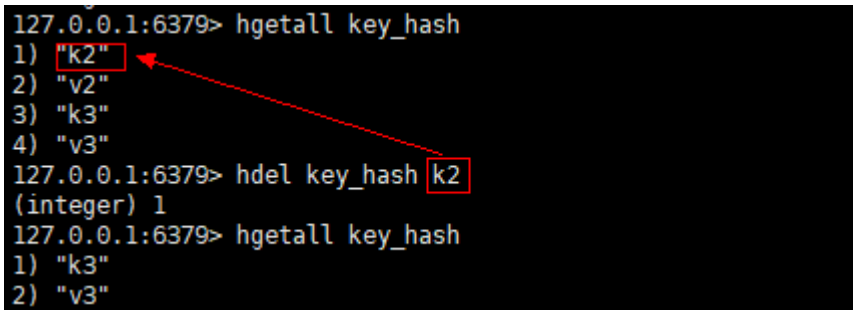
```
hset redis缓存中的key 哈希表中的key1 哈希表中的value1 哈希表中的key2 哈希表中的value2
```



删

- hdel（删除某对kV）

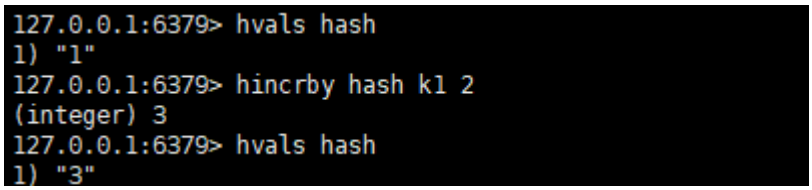
```
hdel redis缓存中的key 待删除hash表中的key
```



改

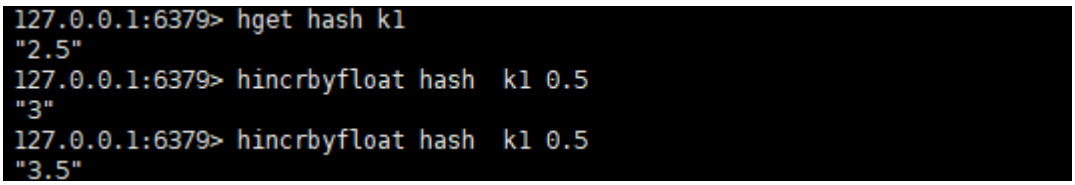
- hincrby（为某个整数类型的value加上int类型的整数值）

```
hincrby redis缓存中的key 待修改hash表中的key int类型加数值
```



- hincrybyfloat（为某个整数类型的value加上float类型的整数值）

```
hincrby redis缓存中的key 待修改hash表中的key float类型加数值
```



Zset（有序集合）

说明

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。 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的排序。zset的成员是唯一的,但分数(score)却可以重复。

查

- zrange（根据score升序后，取出指定下标范围元素）

```
zrange key名 起始下标 结束下标 【withscores】
```

```
127.0.0.1:6379> zadd zset 60 v1 70 v2 80 v3 90 v4 100 v5
(integer) 5
127.0.0.1:6379> zrange zset 0 2
1) "v1"
2) "v2"
3) "v3"
```

```
127.0.0.1:6379> zrange zset 0 -1
1) "v0"
2) "v1"
3) "v2"
4) "v3"
5) "v4"
6) "v5"
```

```
127.0.0.1:6379> zrange zset 0 2 withscores
1) "v1"
2) "60"
3) "v2"
4) "70"
5) "v3"
6) "80"
```

- zrevrange（根据score降序后，取出指定下标范围元素）

```
zrevrange key名 起始下标 结束下标 【withscores】
```

```
127.0.0.1:6379> zrevrange zset 0 -1 withscores
1) "v5"
2) "100"
3) "v4"
4) "90"
5) "v3"
6) "80"
7) "v2"
8) "70"
```

- zrangebyscore（根据对应的score进行升序后，按照指定分数范围查询元素）

```
zrangebyscore key名 开始score 结束score 【withscores】
```

```
127.0.0.1:6379> zrangebyscore zset 60 80
1) "v1"
2) "v2"
3) "v3"
```

```
127.0.0.1:6379> zrangebyscore zset 60 80 withscores
1) "v1"
2) "60"
3) "v2"
4) "70"
5) "v3"
6) "80"
```

```
127.0.0.1:6379> zrangebyscore zset (60 80
1) "v2"
2) "v3"
```

- zrevrangebyscore（根据对应的score进行降序后，按照指定分数范围查询元素）

```
zrevrangebyscore key名 开始score 结束score 【withscores】
```

```
127.0.0.1:6379> zrevrangebyscore zset 80 60
1) "v3"
2) "v2"
3) "v1"
```

注意：scroe范围值同样需按照降序，从大到小

- zrank（根据score升序后，获取某value在zset中的下标位置）

```
zrank key名 value值
```

```
127.0.0.1:6379> zrange zset 0 -1 withscores
1) "v2"
2) "70"
3) "v3"
4) "80"
5) "v4"
6) "90"
7) "v5"
8) "100"
127.0.0.1:6379> zrank zset v2
(integer) 0
```

- zrevrank（根据score降序后，获取某value在zset中的下标位置）

```
zrevrank key名 value值
```

```
127.0.0.1:6379> zrange zset 0 -1 withscores
1) "v2"
2) "70"
3) "v3"
4) "80"
5) "v4"
6) "90"
7) "v5"
8) "100"
127.0.0.1:6379> zrevrank zset v2
(integer) 3
```

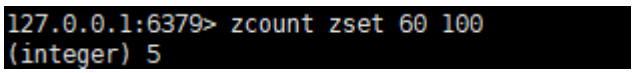
- zcard（获取集合中的元素个数）

```
zcard key名
```

```
127.0.0.1:6379> zcard zset
(integer) 5
```

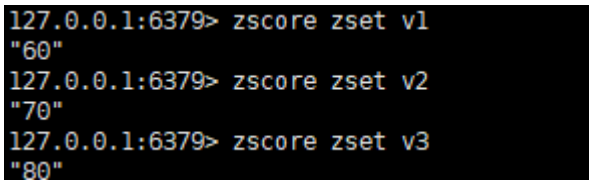
- zcount（获取分数区间内的元素个数）

```
zcount key名 60 100
```



- zscore（获取某value对应的score分数）

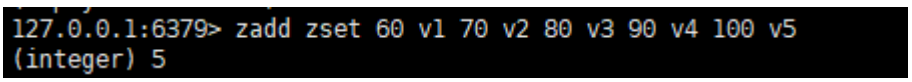
```
zscore key名 value值
```



增

- zadd（给集合添加一个或多个元素数据）

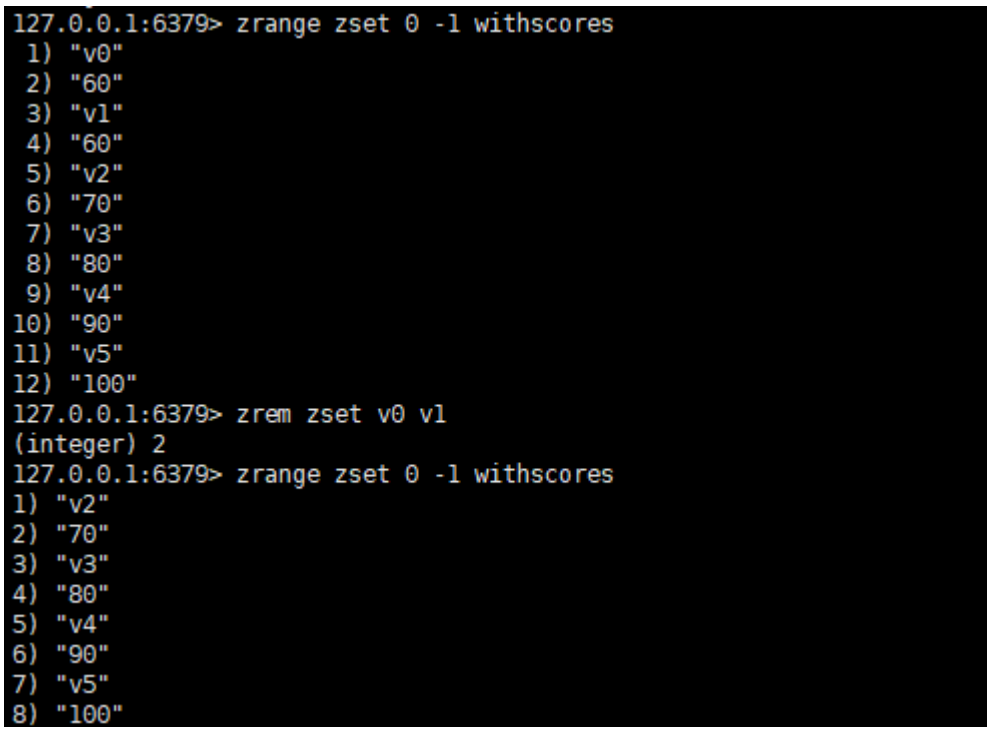
```
zadd key名 score1 value1 【score2 value2 】
```



删

- zrem（删除一个或多个元素）

```
zrem key名 value 【...value】
```



tip

持久化

前言

NoSql主要为内存数据库产品，既然数据存储再内存的话，那么就存在一定数据丢失风险（比如程序崩溃、服务器断电或关机等），所以及时对一些重要数据进行持久化保存到磁盘内是至关重要的。Redis根据对数据的一致性要求不同可采用两种持久化方式，RDB和AOF。

触发持久化条件

- 配置文件中配置的保存机制
- save命令或是bgsave命令
  - save时只管保存，其它不管，全部阻塞
  - bgsave：Redis会在后台异步进行快照操作， 快照同时还可以响应客户端请求。可以通过lastsave 命令获取最后一次成功执行快照的时间
- 执行flushal或flushdbl命令

如何恢复数据

将备份文件 (xxx.rdb或xxx.aof) 移动到 `SNAPSHOTTING` 中 `dir` 属性中配置备份文件读写和保存的目录并启动服务即可。

备份文件异常时处理

当备份文件出现某些原因导致格式不正确时，可根据不同格式的数据使用 `redis-check-aof` 和 `redis-check-dump` 两款工具进行矫正处理。如：

```
redis-check-aof appendfile.aof
```

RDB（Redis DataBase)

简介

在指定的时间间隔内将内存中的数据集快照写入磁盘， 也就是行话讲的Snapshot快照，它恢复时是将快照文件直接读到内存里。

原理

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。 整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。

rdb持久化保存的是dump.**rdb**文件。

评价

优势

REB的原理是fork出一个新进程来保存进程内的所有数据，所以在子进程中的IO操作不会影响到父进程的其它操作，相对来说性能很好的。

劣势

由于REB的保存机制是使用时间间隔来进行快照保存，所以可能存在最后一个周期时间出现某些原因（服务器关机或断电等）未能及时保存到的风险。

AOF（Append Only File)

简介

本身Redis已经觉得使用RDB进行数据的持久化保存的话就已经足够了，但不排除有些业务对于数据的安全性比较高的话，可以选择AOF这一种方式。

原理

以日志的形式来记录每个写操作（包括flushall和flushdb），将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件（也就是在之前备份文件保存的记录不会被下一次备份的时候进行改写，后续重新备份只会将新的日志记录追加到原来的后面），redis启动之初会读取该文件重新构建数据，换言之，redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。

Aof保存的是appendonly.**aof**文件。

rewrite（重写）

简介

AOF采用文件追加方式，文件会越来越大为避免出现此种情况，新增了重写机制,当AOF文件的大小超过所设定的阈值时，Redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集.可以使用命令bgrewriteaof。

触发原理

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(也是先写临时文件最后再rename)，遍历新进程的内存中数据，每条记录有一条的Set语句。重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似。

触发机制

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的多少倍且文件大于配置最小大小时触发

评价

优势

1. 由于数据记录的时间短，所以数据的安全性比较高。
2. 同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差。

劣势

1. 相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb。
2. aof运行效率要慢于rdb,每秒同步策略效率较好，不同步效率和rdb相同。

RDB与AOF总结对比

1. RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储
2. AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以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文件的体积不至于过大。
3. 只做缓存：如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式。
4. 同时使用两种持久化方式，在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。
5. RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢？作者建议不要，因为RDB更适合用于备份数据库(AOF在不断变化不好备份)，快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

性能建议

1. 因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留save 900 1这条规则。
2. 如果Enalbe AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了。代价一是带来了持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基础大小默认值64M太小了，可以设到5G以上。默认超过原大小100%大小时重写可以改到适当的数值。
3. 如果不Enable AOF ，仅靠Master-Slave Replication 实现高可用性也可以。能省掉一大笔IO也减少了rewrite时带来的系统波动。代价是如果Master/Slave同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个Master/Slave中的RDB文件，载入较新的那个。新浪微博就选用了这种架构。

事务

简介

事务管理是很多数据库都有的功能，而Redis的事务管理的是一串操作命令，比如：当我们想执行按顺序执行同时多个指令时。

但与传统的SQL型数据库不一样的是，前者具有原子性，也就是所有操作命令要么都成功，要么都失败。而Redis的事务确是一种**半原子状态的一种事务管理**，因为即使事务管理栈中的其中一个命令失败了，或许也不会影响到前面的操作。具体在下面的介绍会讲到。

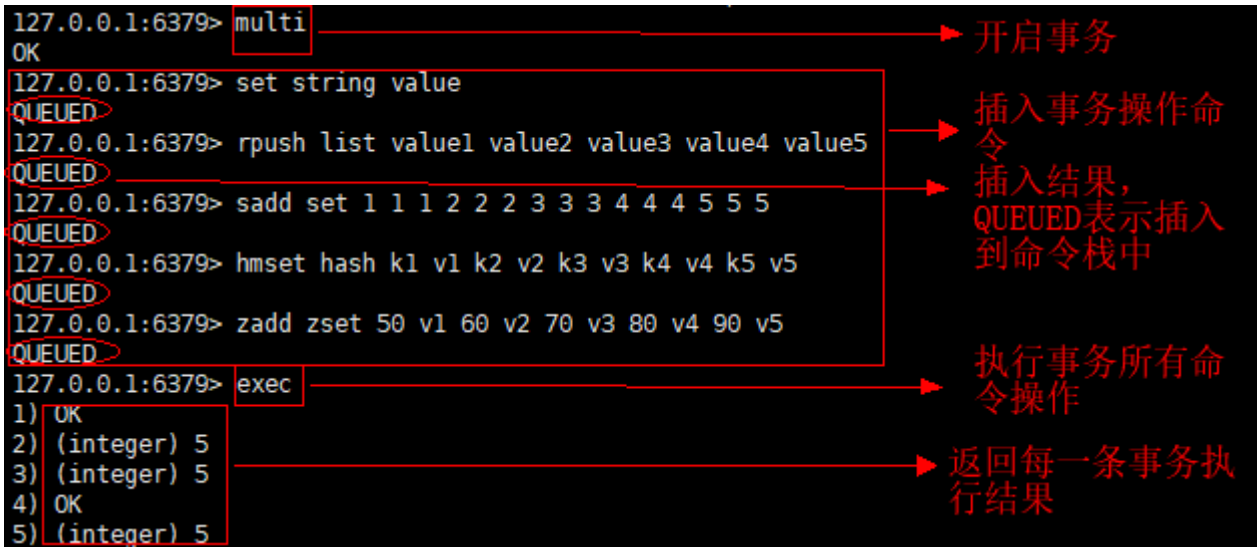
且Redis中的事务管理在提交事务之前，所有操作命令并不会实际执行。这点与Mysql中的事务管理是由区别的，在Mysql中，若你在事务进行时修改了某个数据，那么你可在结束事务之前，在本事务内可查询到你这次修改后的数据。而Redis只有在事务提交（exce命令）之后才会真正执行所有操作命令。所以，对于Redis来说，没有事务回滚概念。

事务命令

事务执行所有情况

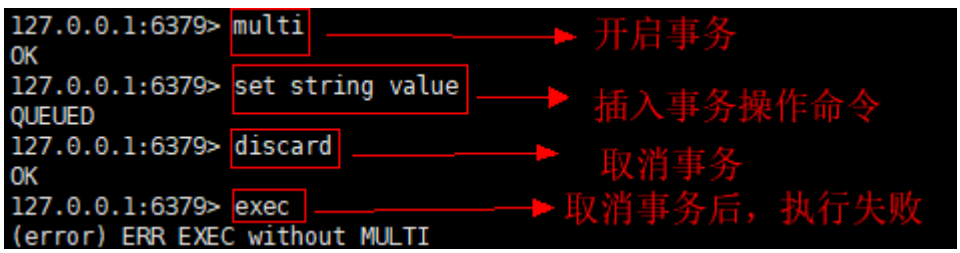
> 正常执行

开启事务后到执行所有操作命令的整个过程没有出现任何异常。



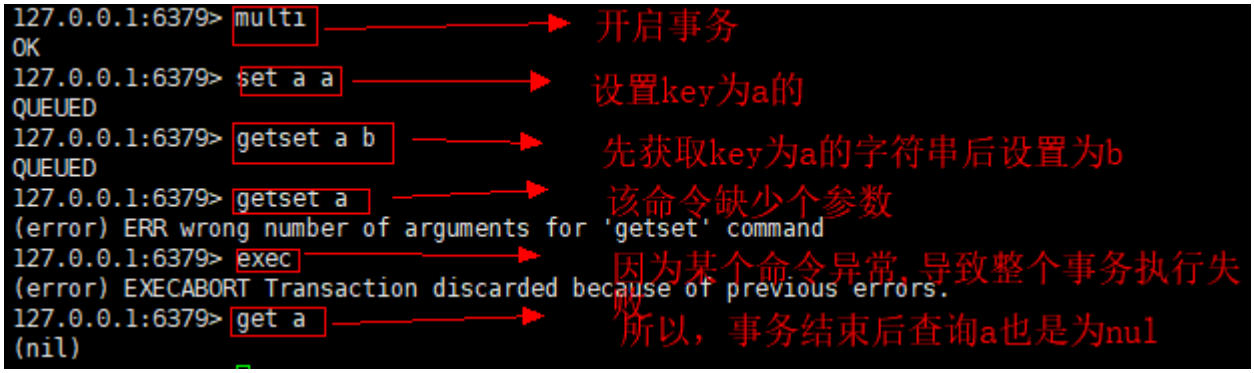
> 放弃执行

在开始事务后，中途终止事务的执行。



> 错误终结所有操作

在开启事务后，若插入的操作命令中存在错误，如下图中getset 中缺少一个参数，那么在事务提交执行所有操作命令时就会出现致命的错误，从而导致所有操作失败。

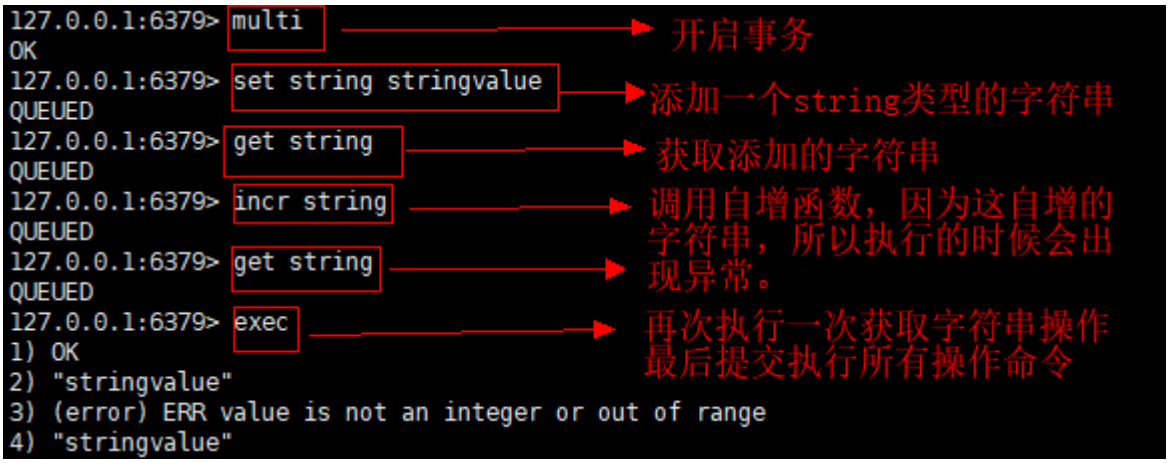


> 异常终结部分操作

这里与错误终止所有操作的区别在于，就好似前者是编译时异常时可能因为语法等（Java中出现编译时异常是连运行都不可以的），从而导致较为严重的错误，所以终止了所有命令操作。后者就好似运行时异常（Java的运行异常可以try-catch，不影响其它的代码执行），比如下面的自增时的参数异常，这种异常较为轻。

所以出现这种情况时，只会导致出现异常的操作命令不被执行而已，并不影响其它操作命令。

正是由于这种情况，所以说Redis的事务是一种半原子状态的。



悲观锁和乐观锁

> 悲观锁

与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。

就比如说有一个洗手间，里面有很多个马桶间可以给人上大号。这时候有一个人就想要上大号，但是他怕自己上大号被别人偷看，所以他进去洗手间之后就把洗手间的们给锁了，整个洗手间的这么多马桶间就只有他一个人在使用。那么，如果其他人也想要上大号的话，就只有等他上完出来后才能进去，老是这样的话，等着上大号的人就肯定会排老长队了。

另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。

说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。

> 乐观锁

乐观锁不是数据库自带的，需要我们自己去实现。乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。

通常实现是这样的：在表中的数据进行操作时(更新)，先给数据表加一个版本(version)字段，每操作一次，将那条记录的版本号加1。也就是先查询出那条记录，获取出version字段,如果要对那条记录进行操作(更新),则先判断此刻version的值是否与刚刚查询出来时的version的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将version字段的值加1；如果更新时发现此刻的version值与刚刚获取出来的version的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。

Watch监控

前面说到悲观锁和乐观锁，在Redis中Watch监控就有点类似于其中的乐观锁，为的就是防止多事务同时修改同个数据导致错乱的问题。

被Watch监控的key，在本次事务提交修改前，先会对比之前获取的version于当前key的version，如果小于当前缓存的version的话，那么执行该命令将失败，必须重写获取新的数据和该key的version，如果等于，则执行成功且将该key的version+1。

消息订阅

简介

举个例子：当我们在微信中订阅了某个公众号后，每当这个公众号给我发消息时，我们都能自动收取到这条消息。

消息中间件时JavaEE的一个规范，其实现的产品有很多，比如：RabbiMQ。而Redis主要的功能不在消息订阅，而是数据缓存上，所以对此章节，暂且浅入了解即可。

运用

1. **客户端1开始订阅**
    - subscribe（指定广播名订阅多个广播）

```
subscribe broadcast1 【broadcast2...】
```

```
127.0.0.1:6379> subscribe broadcast1 broadcast2 broadcast3
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "broadcast1"
3) (integer) 1
1) "subscribe"
2) "broadcast2"
3) (integer) 2
1) "subscribe"
2) "broadcast3"
3) (integer) 3
```

- *p*subscribe（使用通配符订阅多个广播）

```
psubscribe broadcast*
```

```
127.0.0.1:6379> psubscribe broadcast*
Reading messages... (press Ctrl-C to quit)
```

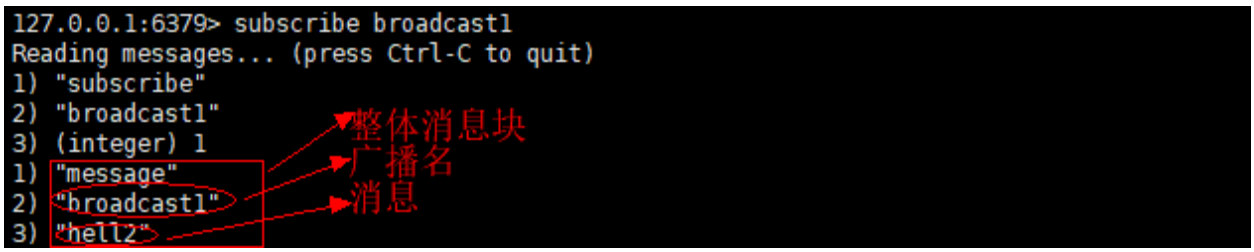
2. **客户端2发布消息**

```
publish 广播名 消息
```

```
127.0.0.1:6379> psubscribe broadcast*
Reading messages... (press Ctrl-C to quit)
```

3. **收取消息**

当客户端2发送消息到指定广播后，订阅了该广播的客户端就会自动收取到该消息。



主从复制