# Redis学习笔记

# 五大常用数据类型

连接redis：/usr/local/bin/redis-cli

## Redis键(key)

keys\*查看当前所有key{匹配：key\*1}

exists key 判断某个key是否存在

type key 查看你的key是什么类型

del key 删除指定的key数据

unlink key根据value选择非阻塞删除

仅将keys从keyspace元数据删除，真正的删除会在后续异步操作

expire key 10 10秒钟：为给定的key设置过期时间

ttl key查看还有多少秒过期，-1表示永不过期，-2表示已过期·

select命令切换数据库

dbsize查看当前数据库的key的数量

flushdb清空当前库

flushall通杀全部库

## Redis字符串（String）

### 简介

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类型是一个二进制安全的。意味着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

### 常用命令

set <key> <value>添加键值对

\*NX：当数据库中key不存在时，可以将key-value添加数据库

\*XX：当数据库key存在时，可以将key-value添加数据库，与NX参数互斥

\*EX：key的超时秒数

\*PX：key的超时毫秒数，与EX互斥

get <key>查询对应键值

append <key> <value> 将给定的<value>追加到原值的末尾

strlen <key>获得值的长度

setnx <key> <value> 只有在key不存在时，设置key的值

incr <key>

将key中存储的数字值增1

只能对数字值操作，如果为空，新增值为1

decr <key>

将key中存储的数字值减1

只能对数字值操作，如果为空，新增值为-1

Incrby/decrby <key><步长>将key中存储的数字值增减。自定义步长

原子性

所谓原子操作是指不会被线程调度机制打断的操作：

这种操作一旦开始，就会一直运行到结束，中间不会有任何context switch（切换到另一个线程）

1. 在单线程中，能够在单条指令中完成的操作都可以认为是原子操作。因为中断只能发生于指令之间
2. 在多线程中，不能被其他进程（线程）打断的操作就叫做原子操作

Redis单命令的原子性主要得益于Redis的单线程

mset <key1> <value1> <k2><value2>……

同时设置一个或者多个key-value对

mget <key1><key2><key3>……

同时获取一个或者多个value

msetnx <key1><value1><key2><value2>……

同时设置一个或多个key-value对，当且仅当所有给定key都不存在。

原子性，有一个失效则失败

getrange <key> <起始位置><结束位置>

获取值的范围，类似java中的substring，前包，后包

setrange <key><起始位置><value>

用<value>覆写<key>所储存的字符串值，从<起始位置>开始（索引从0开始）

setex <key><过期时间><value>

设置键值的同时，设置过期时间，单位秒

getset <key> <value>

以旧换新，设置了心智同时获得旧值

Linuxps指令：ps -ef | grep 进程关键字 查找进程关键字

## Redis列表（List）

### 简介

单键多值

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到列表的头部（左边）或者是尾部（右边）

它的底层是个双向链表，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

### 常用命令：

lpush/rpush <key><value1><value2><value3>…从左边/右边插入一个或者多个值。

lpop/rpop <key>从左边/右边吐出一个值。值在键在，值光键亡.

Rpoplpush <key1><key2>从<key1>列表右边吐出一个值，插到<key2>列表左边

lrange<key><start><stop>

按照索引下标获得元素（从左到右）

lrange mylist 0 -1 0左边第一个，-1右边第一个，（0-1表示获取所有）

lindex <key> <index>按照索引下标获得元素（从左到右）

lien <key> 获得列表长度

linsert <key> before <value> <newvalue> 在<value>的后面插入<newvalue>插入值

lrem<key><n><value> 从左边删除n个value（从左到右）

lset <key> <index><value> 将列表key下标为index的值替换成value

### 数据结构

List的数据结构为快速链表quickList

首先在列表元素较少的情况下会使用一块连续额的内存存储，这个结构是ziplist,也即是压缩列表。

它将所有的元素紧挨着一起存储，分配的是一块连续的内存

当数据量比较多额的时候才会改成quicklist

因为普通的链表需要的附加指针空间太大，会比较浪费时间。比如这个链表里存的知识int类型的数据，结构山还需要两个额外的指针prew和next

Redis将链表和ziplist结合起来组成了quicklist。也就是将多个ziplist使用双向指针串起来使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

## Redis集合（Set）

### 简介

Redis set对外提供的功能与list类似是一个链表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个链表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是1list所不能提供的。

Redis的set是string类型的无序集合他底层其实是一个value为null的hash表，所以添加，删除，查找的复杂度都是O（1）

一个算法，随着数据的增加，执行时间的长短，如果是O（1），数据增加，查找数据的时间不变

### 常用指令：

sadd <key> <value1><value2>……

将一个或多个member元素加入到集合key中，已经存在的member元素将被忽略

smembers <key>取出该集合的所有值

sismember <key><value>判断集合<key>是否为含有的该<value>值，有1

，没有0

scard <key> 返回该集合的元素个数

srem <key><value1><value2>……删除集合中的某个元素

spop <key> 随机从该集合中吐出一个值

srandmember <key> <n>随机从该集合中取出n个值。不会从集合中删除

smove <source><destination>value把集合中一个值从一个集合移动到另一个集合

sinter <key1> <key2>返回两个集合的交集元素

sunion <key1> <key2>返回两个结合额的并集元素

sdiff <key1><key2> 返回两个集合的差集元素（key1中的，不包含key2中的）

## Redis哈希（Hash）

hset <key><field><value> 给<key>集合中的<field>键赋值<value>

hget <key1><field>从<key1>集合<field>取出value

hmset <key1><field><value1><field><value2>…批量设置hash的值

hexists<key1><field>查看哈希表key中，给定域field是否存在。

hkeys <key>列出该hash集合的所有field

hvals <key> 列出hash集合的所有value

hincrby <key> <field> <increment>为哈希表key中的域field的值加上增量1 -1

hsetnx <key><field><value>将哈希表key中的域field的值设置为value，当且仅当域field不存在

## Redis有序集合Zset（sorted set）

### 常用命令

zadd <key><score1><value1><score2><value2>……

将一个或多个member元素及其score值加入到有序集key当中

zrange <key><start><stop>[WITHSCORES]

返回有序集key中，下标在<start><stop>之间的元素

带WITHSCORES，可以让分数一起和值返回到结果集

zrangebyscore key min max[withscores][limit offset count]

返回有序集key中，所有score值介于min和max之间（包括等于min或max）的成员。

有序集成员按score值递增（从小到大）次序排列

zrevrangebyscore key maxmin [withscores][limit offset count]

同上，改为从大到小排序

zincrby <key><increment><value>为元素的score加上增量

zrem <key><value>删除该集合下，指定的元素

zcount <key> <min><max>统计该集合，分数区间内的元素个数

zrank <key><value> 返回该值在集合的排名，从0开始

## Redis的发布和订阅

### 什么是发布和订阅

Redis发布订阅（pub/sub）是一种消息通信模式：发送者（pub）发送消息，订阅者

（sub）接收消息

Redis客户端可以订阅任意数量的频道

## 新数据类型

### Bitmaps

1. Bitmaps本身不是一种数据类型，实际上它就是字符串（key-value），但是它可以对字符串的位进行操作
2. Bitmaps单独提供了一套命令，所以在Redis中使用Bitmaps和使用字符串的方法不太相同。可以把Bitmaps想象成一个以位为单位的数组，数组的每个单元只能存储

0和1，数组的下标在Bitmaps中叫做偏移量

### 命令

setbit

1. 格式  
   setbit<key><offset><value>设置Bitmaps中某个偏移量的值（0或1）

\*offset：偏移量从0开始

（2）bitcount

统计字符串被设置为1的bit数。一般情况下，给定的整个字符串都会被进行计数，通过指定额外的start或end参数，可以让计数只在特定的位上进行。Start和end参数的设置，都可以使用负数：比如-1表示最后一个位，而-2表示倒数第二个位，start、end是指bit组的字节的下标数，二者皆包含

HyperLogLog

1. pfadd
2. 格式

pfadd <key> <element>[element…]添加指定元素到HyperLogLog中

1. pfcount

格式

pfcount <key>[key…]计算HLL的近似基数，可以计算多个HLL，比如用HLL存储每天的UV，计算一周的UV可以使用7天的UV合并计算即可

1. pfmerge

格式

pfmerge <destkey><sourcekey>[sourcekey……] 将一个或多个HLL合并后的结果存储在另一个HLL中

1. Geospatal
2. 格式

geoadd <key><longitude><latitude><member>[longitude latitude member…]添加地理位置（经度，，维度，名称）

geopos

geopos <key> <member>[member……]获取指定地区的坐标值

geodist

格式  
geodist <key><member1><member2>[m|km|ft|mi]获取两个位置之间的直线距离

georadius  
(1)格式

georadius < key><longitude><latitude>radius m|km|ft|mi 以给定的经纬度为中心，找出某一半径内的元素

# Jedis

## Redis事务

Redis事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断

Redis事务的主要作用就是串联多个命令防止别的命令插队

## Multi、Exec、discard

从输入Multi命令开始，输入的命令都会依次进出命令队列中，但不会执行，直到输入Exec后，Redis将之前的命令队列中的命令依次执行

组队的过程中可以通过discard来放弃组队

### 事务的错误处理

组队中某个事务出现了报告错误，执行时整个的所有队列都会被取消

如果执行阶段某个命令报出了错误，则只报错的命令不会被执行，而其他命令都会执行，不会回滚

### 悲观锁

顾名思义：就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就会用到很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁

### 乐观锁

顾名思义：就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁多适用于读的应用类型，这样可以提高吞吐量。Redis就是利用这种check-and-set机制实现事务的。

### WATCH key[key…]

在执行multi之前，先执行watch key1 [key2]，可以监视一个（或多个）key，如果在事务执行之前这个（或这些）key被其他命令所改动，那么是事务将被打断

### UNWATCH

取消WATCH命令第所有key的监视。

如果在执行WATCH命令之后，EXEC命令或DISCARD命令先被执行了的话，那么久不需要再执行UNWATCH了。

## Redis事务三特性

单独的隔离操作

事务中的所有命令都会被序列化，按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断

没有隔离级别的概念

队列中的命令没有提交之前都不会实际被执行，因为事务提交前任何指令都不会被实际执行

不保证原子性

事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

## Redis持久化

### Redis提供了两种不同形式的持久化方式

* RDB（Redis DataBase）
* AOF（Append Of File）

### RDB是什么

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，他恢复时是将快照文件直接读到内存里

### 备份是如何执行的

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，即RDB方式要比AOF方式更加的高校，RDB的缺点是最后一次持久化后的数据可能丢失

### Fork

* Fork的作用是复制一个与当前进程一样的进程新进程的所有数据（变量，环境变量、程序计数器等）数值和原进程一致，但是是一个全新的进程，并作为原进程的子进程
* 在linux程序中，fork（）产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用，处于效率考虑，linux中引入了“写时复制技术”
* 一般情况父进程和子进程会共用同一段物理内存，只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程

stop-writes-on-bgsave-error yes

当Redis无法写入磁盘党的话，直接关掉Redis的写操作、推荐yes

rdbcompression yes

对于存储到磁盘中的快照，可以设置是否进行压缩存储。如果是的话，redis会采用LZF算法进行压缩

如果你不想消耗CPU来进行压缩的话，可以设置为关闭此功能。推荐yes

补充：linux下的mv操作

Linux mv（英文全拼：move file）命令用来为文件或目录改名、或将文件或目录移入其它位置。

### Redis持久化之AOF

#### 什么是AOF

以日志形式来记录每个写操作（增量保存），将Redis执行过的所有指令记录下来（读操作不记录），只追加文件但是不可以改写文件，redis启动之初会读取改文件重新构建数据，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和RDB同时开启，Redis听谁的

AOF和RDB同时开启，系统默认取AOF的数据（数据不会存在丢失）

#### 异常恢复

修改啊默认的appendonly no，改为yes

如遇到AOF文件损坏，通过/usr/local/bin/redis-check-aof—fix appendonly.aof进行恢复

备份被写坏的AOF文件

恢复：重启redis，然后重新加载

#### 同步频率的设置

Appendfsync always

始终同步，每次Redis的写入都会立刻记入日志：性能比较差但数据完整性比较好

Appendfsync everysec

每秒同步，每秒传入日志一次，如果宕机，本秒的数据可能丢失

Appendfsync no

Redis不主动进行同步，把同步时机交给操作系统

#### Redis\_主从复制

##### 什么是主从复制

主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，slave以读为主

##### 反客为主

当一个master宕机后，后面的slave可以立刻升级为master，其后面的slave不用做任何修改

用slaveof no one将丛机变为主机

### 哨兵模式

#### 什么是哨兵模式

反客为主的自动版，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库

sentinel monitor mymaster 127.0.0.1 6379 1

其中mymaster为监控对象起的服务器名称，1为至少有多少个哨兵同意迁移的数量

#### 当主机挂掉，从机选举产生新的主机

那个从机会被选举为主机呢？根据优先级别：slave-priority

原主机重启后会变从机

replica-priority 100:值越小，优先级越高

1. 选择优先级靠前的
2. 选择偏移量最大的
3. 选择runid最小的从服务

优先级在redis.conf中默认：replica-priority 100，值越小优先级越高

偏移量是指在获得原主机数据最全的

每个redis实例启动后都会随机生成一个40位的runid

#### 复制延时

由于所有的写操作都是先在Master上操作，然后同步更新到slave上，所以从Master同步到Slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使这个问题更加严重

### Redis集群

#### 什么是集群

Redis集群实现了对Redis的水平扩容，即启动N个redis节点，将整个数据库分布存储在这N个节点中，每个节点存储总数据的1/N

Redis集群通过分区来提供一定程度的可用性：即使集群中有一部分节点失效或者无法进行通讯，集群也可以继续处理命令请求

#### Redis cluster 配置修改

Cluster-enabled yes 打开集群模式

Cluster-config-file nodes-6379.conf 设定节点配置文件名

Cluster-node-timeout 15000 设定节点失联时间，超过该时间（毫秒），集群自动进行主从切换

#### Linux指令的补充

|  |
| --- |
| :s/name/title/ #替换当前行第一个 name 为 title  :s/name/title/g #替换当前行所有 name 为 title  :n,$s/name/title/ #替换第 n 行开始到最后一行中每一行的第一个 name 为 title  :n,$s/name/title/g #替换第 n 行开始到最后一行中每一行所有 name 为 title  #（n 为数字，若 n 为 .，表示从当前行开始到最后一行）  :%s/name/title/ #（等同于 :g/name/s//title/） 替换每一行的第一个 name 为 title  :%s/name/title/g #（等同于 :g/name/s//title/g） 替换每一行中所有 name 为 title  可以使用 #或+ 作为分隔符，此时中间出现的 / 不会作为分隔符  :s#name/#title/# 替换当前行第一个 name/ 为 title/  :%s+/oradata/apras/+/user01/apras1+ （  使用+ 来 替换 / ）： /oradata/apras/替换成/user01/apras1 |
|  |

#### Redis将6个集群合成一个集群

redis-cli --cluster create --cluster-replicas 1 192.168.139.128:6379 192.168.139.128:6380 192.168.139.128:6381 192.168.139.128:6389 192.168.139.128:6390 192.168.139.128:6391

16384 slots covered.

#### -c采用集群策略连接，设置数据会自动切换到相应的写主机

|  |
| --- |
| [root@hadoop1 src]# redis-cli -c -p 6379  127.0.0.1:6379> |

通过clusetr nodes命令查看集群信息

|  |
| --- |
| 127.0.0.1:6379> cluster nodes  4a45fc72ae9a01d1f5407a9c479f3dd61fa86590 192.168.139.128:6389@16389 slave fb94558b79c4ac25121500709b52c5cb040f4445 0 1636283679000 3 connected  fb94558b79c4ac25121500709b52c5cb040f4445 192.168.139.128:6381@16381 master - 0 1636283679797 3 connected 10923-16383  5186b161d72fc429295f3236ef6f4bf1da5ff990 192.168.139.128:6379@16379 myself,master - 0 1636283681000 1 connected 0-5460  841b04de21977fe79fe0338b4829ab7094c9ac16 192.168.139.128:6391@16391 slave 6439dd29f21f384e05340c7cf12601a2d523a0f6 0 1636283681000 2 connected  6439dd29f21f384e05340c7cf12601a2d523a0f6 192.168.139.128:6380@16380 master - 0 1636283682000 2 connected 5461-10922  093e439fafa7a413f566c77f0c3b96113157e730 192.168.139.128:6390@16390 slave 5186b161d72fc429295f3236ef6f4bf1da5ff990 0 1636283682816 1 connected  127.0.0.1:6379> |

#### 查询集群中的值

CLUSTER GETKEYSINSLOT <slot><count>返回count个slot槽中的值

Cluster countkeysinslot 插槽值（查看当前插槽中的key的数量），只能查看自己插槽范围内的。

如果某一段插槽的主从都挂掉，而cluster-require-full-coverage为yes，那么，整个集群都挂掉

如果某一段插槽的从主都挂掉，而cluster-require-full-coverage为no，那么，该插槽数据全都不能使用，也无法存储

Redis.conf中的参数cluster-require-full-cover

#### Redis集群提供了一下好处

实现扩容

分摊压力

无中心配置相对简单

#### Redis集群的不足

多键操作是不被支持的

多键的Redis事务是不被支持的。Lua脚本不被支持

由于集群方案出现较晚，很多公司已经采取了其他的集群方案，而代理或者客户端分片的方案想要迁移至redis cluster，需要整体迁移而不是逐步过渡，复杂度较大

### 应用问题解决—缓存穿透

一个一定不存在缓存及查询不到的数据，由于缓存是不命中时被动写的，并且处于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义

解决方案：

1. 对空值缓存：如果一个查询返回的数据为空（不管数据是否不存在），我们仍然把这个空结果（null）进行缓存，设置空结果的过期时间会很短，最长不超过五分钟
2. 设置可访问的名单（白名单）

使用bitmaps类型定义一个可以访问的名单，名单id作为bitmaps的偏移量，每次访问和bitmap里面的id进行比较，如果访问id不在bitmaps里面，进行拦截，不允许访问·。

1. 采用布隆过滤器：（布隆过滤器（Bloom Filter））是1970年由布隆提出的。它实际是一个很长的二进制向量（位图）和一系列随机映射函数（哈希函数）。
2. 进行实时监控：当发现Redis的命中率开始急速降低，需要排查访问对象和访问的数据，和运维人员配合，可以设置黑名单限制服务

### 缓存击穿

1数据库访问压力顺价增加

2.redis里面没有出现大量key过期

3.redis正常运行

Redis某个key过期了，大量访问使用这个key

解决方案：

Key可能会在某些事件点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿的问题”

解决问题：

1. 预先设置热门数据：在redis高峰访问之前，把一些热门数据提前存入到redis里面，加大这些热门key的时长
2. 实时调整：现场监控那那些数据热门，实时调整key的过期时间
3. 使用锁：  
   （1）就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db
4. 先使用缓存工具的某些带成功操作返回值的操作（比如redis的SETNX）

### 应用解决问题—缓存雪崩

1.数据库压力变大服务器崩溃

1.在极少时间段，查询大量key的集中过期情况

缓存失效的雪崩效应对底层系统的冲击非常可怕

解决方案：  
（1）构建多级缓存构架：nginx缓存+redis缓存+其他缓存（ehcache）等

1. 使用锁或队列：
2. 用枷锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。不适用高并发情况
3. 设置过期标志更新缓存  
   记录缓存是否过期（设置提前量），如果过期会触发通知另外的线程在后台去更新实际key的缓存
4. 将缓存失效时间分散开  
   比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件
5. 使用setnx上锁，通过del释放锁
6. 锁一直没有释放，设置key过期时间，自动释放
7. 上锁之后突然出现异常，无法设置过期时间了
8. \*上锁时候同时设置过期时间就可以

|  |
| --- |
| set users 10 nx ex 12表示即上锁又设置过期时间 |

应用问题解决—分布式锁（UUID）

第一步：uuid表示不同的操作

第二步：释放锁时候，首先判断当前uuid和要释放锁uuid是否一样

为了保证分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

互斥性：在任意时刻，只有一个客户端能持有锁

不会发生死锁：即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能枷锁

解铃还需系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了

加锁和解锁必须具有原子性