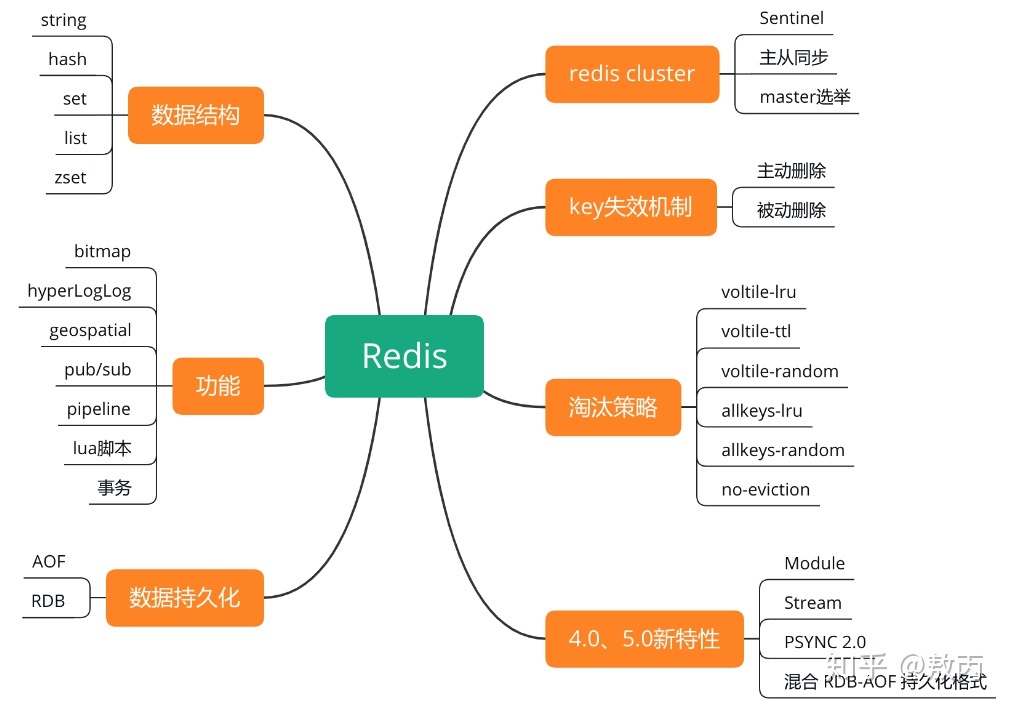
# Redis





## 简介

NoSql:（Not only Sql, non-relational）指的是非关系型数据库;

优点：易于扩展：数据之间无关系，这就非常容易扩展

大数据量，高性能，NoSql数据库具有非常高的读写性能，这得益于数据库的简单结构

Redis是一个开源的，基于键值对的NoSql数据库；

## Redis的值可以是String(字符串)，hash（哈希），list（列表），set（集合），zset（有序集合），Bitmaps(位图),HyperLogLog,GEO(地理信息定位)等多种数据结构和算法组成

Redis可以满足很多应用场景；所有数据都是放在内存中，所以它的读写能力非常惊人

## Redis的特性

1. 速度快：官方给出的读写性能可以达到10万/秒（也取决于机器性能）

Redis所有的数据都是放在内存中的，把数据放在内存中是redis把速度快的主要原因

Redis是用c语言实现的，离操作系统进，所以执行速度相对会更快

Redis是单线程架构，预防了多线程可能产生的竞争问题

1. 基于键值对的数据结构服务器，便于应用在许多的开发场景，同时也提升了开发效率
2. 丰富的功能

提供了键过期功能，可以用来实现缓存

提供了订阅发布功能，可以实现消息系统

支持lua脚本功能，可以利用lua创建出新的redis命令

提供简单的事务，能在一定程度上保证事务特性

提供了流水线(Pipeline)功能，这样客户端能将一批命令一次性传递到redis，减少网络开销

1. 简单稳定

Redis源码少

Redis使用单线程模型

Redis不需要依赖操作系统的类库

1. 支持的客户端语言多，几乎涵盖了主流的编程语言，例如java，PHP，c,c++,nodejs
2. 持久化:redis提供了2中持久化方式：RDB和AOF,也就是可以使用这2种方式来将内存中的数据保存到本地磁盘
3. 主从复制：redis提供复制功能，实现多个相同数据的redis副本，复制功能是分布式redis的基础
4. 高可用和分布式：redis cluser ,它是redis真正的分布式实现，提供了高可用，读写和容量的扩展性

## Redis的应用场景

Redis可以做什么

1. 缓存
2. 排行榜系统
3. 计数器应用
4. 社交网络
5. 消息队列系统

Redis键值队中键的一些常用命令

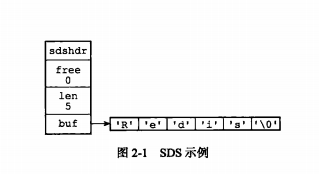
1. 查看所有键 ：keys \*
2. 键总数 dbsize
3. 检查键是否存在：exists key
4. 删除键：del key 删除键
5. 键过期：expire key seconds 设置键过期时间
6. 键的数据结构类型 ： type key 返回值的数据结构类型

# Redis的数据结构

## 字符串string

字符串类型是redis的最基础数据结构；redis的键都是字符串，而且其他几种数据结构都是在字符串类型基础上构建的，字符串类型的值可以是字符串，数字，甚至是二进制，最大不能超过512M

Redis不是采用C语言中的字符串表示的，而是自己构建了一个名为简单动态字符串（SDS）的抽象类型,并将SDS作为默认的字符串表示



### 内部编码

字符串类型的内部编码有3中

1. int：8个字节的长整型
2. embstr:小于等于39个字节的字符串
3. raw:大于39字节的字符串

返回字符串类型：object encoding key

### 数据结构SDS

SDS （simple dynamic string：简单动态字符串）定义

**Sds有两个版本，在redis3.2之前使用第一个版本结构如下**

struct sdshdr{

     //记录buf数组中已使用字节的数量

//等于 SDS 保存字符串的长度

     int len;

     //记录 buf 数组中未使用字节的数量

     int free;

     //字节数组，用于保存字符串

     char buf[];

}

len保存了SDS存放字符串的长度

buf[]数组用来保存字符串的每个元素

free记录buf数组中未使用的字节数量

Redis 3.2 版本中，对数据结构做出了修改，针对不同的长度范围定义了不同的结构，如下，这是目前的结构

typedef char \*sds;

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr5 { // 对应的字符串长度小于 1<<5

unsigned char flags; /\* 3 lsb of type, and 5 msb of string length \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr8 { // 对应的字符串长度小于 1<<8

uint8\_t len; /\* used \*/ //目前字符创的长度

uint8\_t alloc; //已经分配的总长度

unsigned char flags; //flag用3bit来标明类型，类型后续解释，其余5bit目前没有使用

char buf[]; //柔性数组，以'\0'结尾

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr16 { // 对应的字符串长度小于 1<<16

uint16\_t len; /\* used \*/

uint16\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr32 { // 对应的字符串长度小于 1<<32

uint32\_t len; /\* used \*/

uint32\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr64 { // 对应的字符串长度小于 1<<64

uint64\_t len; /\* used \*/

uint64\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

### 与c语言的字符串类型比较

1. **、常数复杂度获取字符串长度**

由于 len 属性的存在，我们获取 SDS 字符串的长度只需要读取 len 属性，时间复杂度为 O(1)。而对于 C 语言，获取字符串的长度通常是经过遍历计数来实现的，时间复杂度为 O(n)。通过 strlen key 命令可以获取 key 的字符串长度。

**②、杜绝缓冲区溢出**

我们知道在 C 语言中使用 strcat  函数来进行两个字符串的拼接，一旦没有分配足够长度的内存空间，就会造成缓冲区溢出。而对于 SDS 数据类型，在进行字符修改的时候，会首先根据记录的 len 属性检查内存空间是否满足需求，如果不满足，会进行相应的空间扩展，然后在进行修改操作，所以不会出现缓冲区溢出。

**③、减少修改字符串的内存重新分配次数**

C语言由于不记录字符串的长度，所以如果要修改字符串，必须要重新分配内存（先释放再申请），因为如果没有重新分配，字符串长度增大时会造成内存缓冲区溢出，字符串长度减小时会造成内存泄露。

　　而对于SDS，由于len属性和free属性的存在，对于修改字符串SDS实现了空间预分配和惰性空间释放两种策略：

　　1、空间预分配：对字符串进行空间扩展的时候，扩展的内存比实际需要的多，这样可以减少连续执行字符串增长操作所需的内存重分配次数。

　　2、惰性空间释放：对字符串进行缩短操作时，程序不立即使用内存重新分配来回收缩短后多余的字节，而是使用 free 属性将这些字节的数量记录下来，等待后续使用。（当然SDS也提供了相应的API，当我们有需要时，也可以手动释放这些未使用的空间。）

**④、二进制安全**

　　因为C字符串以空字符作为字符串结束的标识，而对于一些二进制文件（如图片等），内容可能包括空字符串，因此C字符串无法正确存取；而所有 SDS 的API 都是以处理二进制的方式来处理 buf 里面的元素，并且 SDS 不是以空字符串来判断是否结束，而是以 len 属性表示的长度来判断字符串是否结束。

**⑤、兼容部分 C 字符串函数**

　　虽然 SDS 是二进制安全的，但是一样遵从每个字符串都是以空字符串结尾的惯例，这样可以重用 C 语言库<string.h> 中的一部分函数。

### 常用命令

**设置值**

set key value [ex seconds] [px milliseconds] [nx|xx]

set hello world --设置key为hello value为world

其他选项

ex seconds:为key设置秒级过期时间

px milliseconds:为key设置毫秒级过期时间

nx:键必须不存在才能设置成功，用于添加

xx:与nx相反，键必须存在才能设置成功，用于更新

setex key seconds value

setnx key value

这两个命令的作用和ex,nx选项一样

set setnx setxx的区别

当键（以hello为key）不存在的时候

使用exists命令： exists hello 返回值为0

设置键为hello ：set hello value 返回ok

键hello已经存在，使用setnx设置键hello时会失败，返回0

使用set xx能成功：set hello redis xx 返回ok

setnx和setxx的使用场景

setnx:由于redis的单线程命令处理机制，如果多客户端同时设置setnx key value,根据setnx的特性只有一个客户端能设置成功，setnx可以作为分布式锁的一种实现方案

**获取值**

get key

获取hello的值:get hello 返回value值

如果获取的键不存在，则返回nil（空）

**批量设置值**

mset key value [key value …]

一次设置3个键值队

mset k1 v1 k2 v2 k3 v3

**批量获取值**

mget key [key…]

批量获取k1 k2 k3:mget k1 k2 k3

批量操作有助于提高业务效率，但是每次批量操作发送的命令数不是无节制的，如果数量过多可能造成redis阻塞或者网络阻塞

**计数**

incr key

incr命令用于对值做自增操作，返回结果分为三种情况

1. 值不是整数，直接返回错误
2. 值是整数，返回自增后的结果
3. 键不存在，按照值为0自增，返回结果1

除了自增还有decr自减，incrby自增指定数字，decrby自减指定数字，incrbyfloat自增浮点数

Redis是单线程架构，任何命令到redis服务端都要顺序执行

**追加值**

append key value:命令向字符串尾部追加值

**字符串长度**

strlen key

strlen redis 结果为5

中文占用3个字节

Set hello 世界

strlen hello 返回结果为6

**设置并返回原值**

getset key value

getset和set一样会设置值，不同的是它同时会返回键原来的值

getset hello world

返回:nil

getset hello redis

返回:world

**设置指定位置的字符**

setrange key offset value

offset字符串指定位置

**获取部分字符串**

getrange key start end

start和end分别是开始和结束的偏移量

### 应用场景

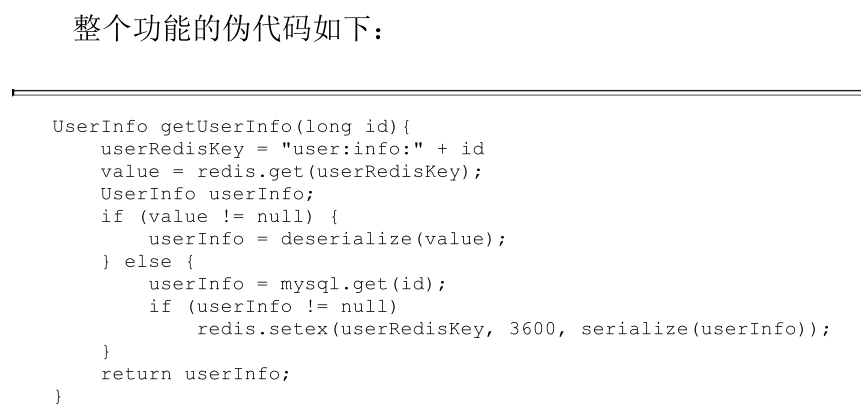
1. 缓存功能

缓存使用场景，其中redis作为缓存层，mysql作为存储层，绝大部分请求的数据都是从redis中获取，redis具有支持高并发的特性，所以缓存通常能够起到加速读写和降低后端压力的作用

设计合理的键名：有利于防止键冲突和项目可维护性，比较推荐的方式是使用**业务名:对象名:id:[属性]**作为键名；

例如mysql的数据库名为vs,用户名为user，那么对应的键可以用vs:user:1,vs:user:1:name来表示，如果当前redis只被一个业务使用，那么就可以去掉vs

获取用户信息，，首先从redis获取用户信息，如果没有从redis获取到用户信息，需要从mysql中进行获取，并将结果回写到redis，添加1小时的过期时间



1. 计数

使用redis作为计数的基础工具，它可以使用实现快速计数，查询缓存的功能，同时数据可以异步落地到其他数据源

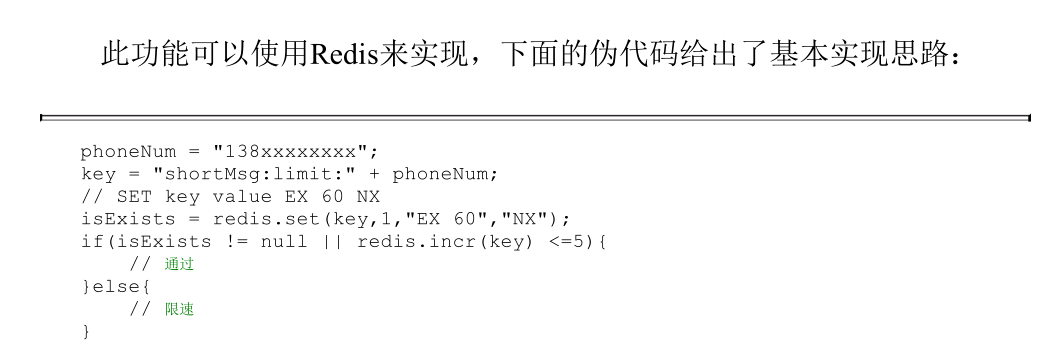
1. 共享session

分布式web服务中，使用redis将用户的session进行集群管理，每次用户更新或查询登陆信息都直接从redis中集中获取

1. 限速

场景：应用在每次登陆时，让用户输入手机验证码，从而确定是否是用户本人，但是为了短信接口不被频繁访问，会限制用户每分钟获取验证码的频率，例如每分钟不能超过5次

设置手机号码对应的key,value为1，将key保存到redis中，设置过期时间为60秒；



Redis其他限速功能，例如网站限制一个ip地址不能在一秒钟之内访问超过n次也可以采用类似的思路

## 列表list

### 内部编码

Ziplist(压缩列表)：当列表的元素个数小于list\_max\_ziplist\_entries配置（默认是512），同时列表中每个元素的值都是小于list\_max\_ziplist\_value配置（默认是64），redis会使用ziplist来减少内存的使用

Linkedlist：当列表类型无法满足ziplist的条件，redis就使用linkedlist；当列表中的数据量比较大的时候，需要通过双向循环链表来实现，因为双向链表占用的内存比压缩列表要多， 所以当创建新的列表键时， 列表会优先考虑使用压缩列表， 并且在有需要的时候， 才从压缩列表实现转换到双向链表实现。

**压缩列表转换为双向链表的条件**

创建新列表时redis默认使用redis\_encoding\_ziplist编码，单满足以下任意一个条件时，就会被转换为redis\_endcoding\_linkedlist:试图往列表新添加一个字符串值，且这个字符串的长度超过 、list\_max\_ziplist\_value （默认值为 64 ）。ziplist 包含的节点超过 list\_max\_ziplist\_entries （默认值为 512 ）。

注意：这两个条件是可以修改的，在 redis.conf 中：

list-max-ziplist-value 64

list-max-ziplist-entries 512

### 数据结构

#### 双向链表linkedlist

linkedlist是标准的双向链表，Node节点包含prev和next指针，可以进行双向遍历；还保存了 head 和 tail 两个指针，因此，对链表的表头和表尾进行插入的复杂度都为 (1) —— 这是高效实现 LPUSH 、 RPOP、 RPOPLPUSH 等命令的关键。

这个链表结构定义在adlist.h文件中

ListNode节点数据结构：

typedef struct listNode{

//前置节点

struct listNode \*prev;

//后置节点

struct listNode \*next;

//节点的值

void \*value;

}listNode

typedef struct listIter {

listNode \*next;

int direction;

} listIter;

链表数据结构

typedef struct list{

//表头节点

listNode \*head;

//表尾节点

listNode \*tail;

//链表所包含的节点数量

unsigned long len;

//节点值复制函数

void (\*free) (void \*ptr);

//节点值释放函数

void (\*free) (void \*ptr);

//节点值对比函数

int (\*match) (void \*ptr,void \*key);

}list;

#### 压缩列表Ziplist

压缩列表是redis为了节约内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型数据结构，一个压缩列表包含任意多个节点，每个节点可以保存一个字节数组或者一个整数类型的值

Ziplist不是一个类名，压缩列表的各个组成部分

<zlbytes> <zltail> <zllen> <entry> <entry> ... <entry> <zlend>

Zlbytes:类型为uint32\_t,4字节；记录压缩列表占用的内存字节数，在对压缩列表进行内存重分配，或者计算zlend时使用

Zltail：类型uint32\_t,4字节；记录压缩列表尾节点距离列表起始位置地址有多少字节；通过这个偏移量，可以直接知道链表尾节点的地址

Zllen：类型为uint16\_t,2字节，记录压缩列表的节点数量，

Zlend：用来标记压缩列表的末端

Entry：列表节点

Zlentry就是ziplist的节点

typedef struct zlentry {

unsigned int prevrawlensize; prevrawlensize是指prevrawlen的大小，有1字节和5字节两种

unsigned int prevrawlen; prevrawlen是前一个节点的长度

unsigned int lensize; lensize为编码len所需的字节大小

unsigned int len; len为当前节点长度

unsigned int headersize; 当前节点的header大小

unsigned char encoding; 节点的编码方式

unsigned char \*p; 指向节点的指针

} zlentry;   
ziplist 将表中每一项存放在前后连续的地址空间内，每一项因占用的空间不同，而采用变长编码。当元素个数较少时，Redis 用 ziplist 来存储数据，当元素个数超过某个值时，链表键中会把 ziplist 转化为 linkedlist，字典键中会把 ziplist 转化为 hashtable。由于内存是连续分配的，所以遍历速度很快。在3.2之后，ziplist被quicklist替代。但是仍然是zset底层实现之一。

Ziplist是一个特殊的双向链表，它没有维护双向指针，而是存储上一个entry的长度，通过长度推算出下一个元素在什么地方牺牲读取性能，获取高效的存储空间，因为存储指针比存储长度更消耗内存，这是典型的时间换空间

**ziplist使用局限性**：字段、值比较小，才会用ziplist。

**如何通过一个节点向前跳转到另一个节点？**  
用指向当前节点的指针 e ， 减去 前一个 entry的长度， 得出的结果就是指向前一个节点的地址 p

**prevrawlen是变长编码，有两种表示方法：**

如果前一节点的长度小于 254 字节，则使用1字节(uint8\_t)来存储prevrawlen；

如果前一节点的长度大于等于 254 字节，那么将第 1 个字节的值设为 254 ，然后用接下来的 4 个字节保存实际长度。

**Ziplist连锁更新问题**

因为在ziplist中，每个zlentry都存储着前一个节点所占的字节数，而这个数值又是**变长编码**的。假设存在一个压缩列表，其包含e1、e2、e3、e4…..，e1节点的大小为253字节，那么e2.prevrawlen的大小为1字节，如果此时在e2与e1之间插入了一个新节点e\_new，e\_new编码后的整体长度（包含e1的长度）为**254字节**，此时e2.prevrawlen就需要扩充为**5字节**；如果e2的整体长度变化又引起了e3.prevrawlen的存储长度变化，那么e3也需要扩…….如此递归直到表尾节点或者某一个节点的prevrawlen本身长度可以容纳前一个节点的变化。**其中每一次扩充都需要进行空间再分配操作**。删除节点亦是如此，只要引起了操作节点之后的节点的prevrawlen的变化，都可能引起连锁更新。

连锁更新在最坏情况下需要进行N次空间再分配，而每次空间再分配的最坏时间复杂度为O(N)，因此连锁更新的总体时间复杂度是O(N^2)。  
即使涉及连锁更新的时间复杂度这么高，但它能引起的性能问题的概率是极低的：需要列表中存在大量的节点长度接近254的zlentry。

ziplist的主要优点是节省内存，但它上面的查找操作只能按顺序查找（可以是从前往后、也可以从后往前）。  
ziplist将数据按照一定规则编码在一块连续的内存区域，目的是节省内存，这种结构并不擅长做修改操作。一旦数据发生改动，就会引发内存realloc，可能导致内存拷贝。

#### Redis3.2+ list的新实现quickList

Redis中的列表list，在版本3.2之前，列表底层的编码是ziplist和linkedlist实现的，但是在版本3.2之后，重新引入 quicklist，列表的底层都由quicklist实现。

在版本3.2之前，当列表对象中元素的长度比较小或者数量比较少的时候，采用ziplist来存储，当列表对象中元素的长度比较大或者数量比较多的时候，则会转而使用双向列表linkedlist来存储。

**这两种存储方式的优缺点**

* 双向链表linkedlist便于在表的两端进行push和pop操作，在插入节点上复杂度很低，但是它的内存开销比较大。首先，它在每个节点上除了要保存数据之外，还要额外保存两个指针；其次，双向链表的各个节点是单独的内存块，地址不连续，节点多了容易产生内存碎片。
* ziplist存储在一段连续的内存上，所以存储效率很高。但是，它不利于修改操作，插入和删除操作需要频繁的申请和释放内存。特别是当ziplist长度很长的时候，一次realloc可能会导致大批量的数据拷贝。

可以认为quickList，是ziplist和linkedlist二者的结合；quickList将二者的优点结合起来。

quickList是一个ziplist组成的双向链表。每个节点使用ziplist来保存数据。  
本质上来说，quicklist里面保存着一个一个小的ziplist

quickList就是一个标准的双向链表的配置，有head 有tail;  
每一个节点是一个quicklistNode，包含prev和next指针。  
每一个quicklistNode 包含 一个ziplist，\*zp 压缩链表里存储键值。  
所以quicklist是对ziplist进行一次封装，使用小块的ziplist来既保证了少使用内存，也保证了性能。

typedef struct quicklist {

quicklistNode \*head; //头结点

quicklistNode \*tail; //尾节点

unsigned long count; /\* total count of all entries in all ziplists \*/

unsigned int len; /\* number of quicklistNodes \*/

int fill : 16; /\* fill factor for individual nodes \*///负数代表级别，正数代表个数

unsigned int compress : 16; /\* depth of end nodes not to compress;0=off \*///压缩级别

} quicklist;

### 常用命令

**添加**

rpush:list右边插入 rpush key value [value…]

lpush:list左边插入 lpush key value [value…]

,linesert：向某个元素前或者后插入

linesert key before|after pivot value:从list中找到pivot元素，在前或者后插入新元素value

**查找**

lrange,lindex llen

lrange key start end获取列表指定索引范围所有的元素。

索引下标的特性：1.索引下标从左到右分别是0到N-1,但是从右到左是-1到-N

2.end选项包含自身

lindex key index:获取指定下标的元素

llen key获取列表的长度

**删除**

Lpop rpop lrem ltrim

lpop key:从列表左侧弹出元素

rpop key:从列表右侧弹出元素

删除指定元素

lrem key count value

从列表中找到等于value的元素进行删除

Count的作用：

1. count>0，从左到右，删除最多count个元素
2. count<0，从右到左，删除最多count绝对值个元素
3. count==0,删除所有

ltrim key start end 安装索引范围修剪列表

**修改**

lset key index newvalue

**阻塞操作**

Blpop和brpop是lpop/rpop的阻塞版，使用方式是相同的，

Blpop命令包含两个参数 key [key…]多个列表的键

Timeout：阻塞时间

当列表为空时，如果设置timeout=3，那么客户端要等待3秒后返回，如果timeout=0，那么客户端一致阻塞下去

列表不为空，客户端会立即返回

### 应用场景

消息队列

Redis的lpush+brpop命令组合可以实现阻塞队列，生产者客户端使用lrpush从列表左侧插入元素，多个消费者客户端使用brpop命令阻塞式的抢列表尾部的元素

列表分页

Lpush+lpop=栈

Lpush+rpop=Queue队列

Lpsh+ltrim=有序集合

Lpush+brpop=消息队列

## 哈希hash

在redis中哈希类型中的值又是一个价值队结构；使用hash表作为底层实现，一个hash表里面可以存放多个hash表节点，每个hash节点保存了字典中的一个键值对

### 数据结构

#### 字典dict

typedef struct dict {

dictType \*type;

void \*privdata;

dictht ht[2]; 两个hash表 rehash时使用

long rehashidx; rehash的索引，-1表示没有进行

unsigned long iterators; 正在迭代的迭代器数量

} dict;

type和privdata针对不同类型的键值对，为创建多态字典而设置的；

type是一个指向dictType结构的指针，每个dictType结构保存了许多用于操作特定类型的键值对的函数，redis会为用途不同的字典设置不同的类型特定函数

privdata保存需要传给特点函数的参数

ht属性是一个包含两个项的数组，数组中的每个项都是一个dictht哈希表，字典使用ht[0]，ht[1]在对ht[0]进行rehash的时候使用

ht[2]，两个hash表

rehashidx，这是一个辅助变量，用于记录rehash过程的进度，以及是否正在进行rehash等信息，当此值为-1时，表示该dict此时没有rehash过程

iterators，记录此时dict有几个迭代器正在进行遍历过程

#### 哈希表dictht

typedf struct dictht{

dictEntry \*\*table;//存储数据的数组 二维

unsigned long size;//数组的大小

unsigned long sizemask;//哈希表的大小的掩码，用于计算索引值，总是等于

//size-1

unsigned long used;//// 哈希表中中元素个数

}dictht;

Table属性是一个数组，数组中每个元素都是一个指向dictEntry结构的指针，每个dictEntry结构保存一个键值对

Size属性记录哈希表的大小，也就是table数组的大小

Used属性记录目前hash表已经节点的数量

Sizemask属性的值是size-1,这个属性和哈希值一起决定一个键应该被放到table数组的哪个索引上

#### 哈希表节点dictEntry

每个dictEntry保存一个键值对

typedef struct dictEntry {

void \*key;

union {

void \*val;

uint64\_t u64;

int64\_t s64;

double d;

} v;

struct dictEntry \*next;

} dictEntry;

Key保存键值对中的键，

V保存值，值可以是一个指针，或者unit64\_t整数，或者是int64\_t整数

Next属性表示指向下个哈希节点dictEntry,这个指针可以将多个哈希值相同的键值对连接在一起，以此来解决键冲突的问题

根据键计算出哈希值和数组索引，然后根据索引值，将包含新键值对的哈希表节点放到dictht中数组table的对应位置

#### 解决键冲突

当两个或以上的键被分配到哈希表数组的同一个索引位置时，键就发生冲突

Redis中的hash表使用链地址法来解决键冲突

每个哈希节点都是一个next指针，第一个哈希表节点可以构成一个单向链表，被分配到同一个索引上的节点可以用这个单向链表连接起来，这就解决了键冲突

dictEntry节点组成的链表没有指向链表表尾的指针，所有将新增的节点放在链表的头部

#### rehash

当操作不断增加，哈希表保存的键值对会不断的增加或减少，让哈希表的负载因子维持在一个合理的范围内，当哈希表保存的键值对数量太多或者太少时，需要对哈希表进项扩展或者收缩，这个扩展和收缩就是rehash操作来实现的

执行rehash的步骤

1. 为ht[1]分配空间，这个哈希表的空间大小取决于要执行的操作，以及ht[0]当前包含的键值对数量

如果是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used\*2的2的n次方

收缩操作，那么ht[1]的大小为第一个大于等于ht[0].used 的2的n次方

1. 将保存在ht[0]中的键值对rehash到ht[1]上，rehash是指重新计算的哈希值和索引值，然后将键值保存到ht[1]上，
2. 当ht[0]全部迁移到ht[1]上后，释放ht[0],将ht[1]设置为ht[0],在ht[1]上新建一个空白的哈希表为下次rehash做准备

哈希表的扩展和收缩

自动开始对哈希表的扩展操作

1. 服务器目前没有在执行bgsave或者bgrewriteaof命令,并且哈希表的负载因子大于等于1
2. 服务器目前在执行bgsave或者bgrewriteaof命令，哈希表的负载因子大于等于5

当负载因子小于0.1时，自动开始收缩操作

负载因子的计算公式：load\_factor=ht[0].used/ht[0].size

负载因子=哈希表以保存的节点数量 / 哈希表大小

在进行rehash操作过程中，对hash表的操作（删改查）会在两个表上进行，添加操作只在ht[1]上进行

### 命令

**设置值**

hset key field value

hset key name tom

hset key age 1

设置成功返回1，失败返回0；

Redis提供hsetnx命令，作用和setnx类型，只不过作用域由键变为field

**获取值**

hget key field

hget key name

如果field不存在，会返回nil

**删除filed**

hdel key field[field]…删除1个或者多个field，返回结果为成功删除field的个数

删除的filed不存在，返回0

hdel user name 🡪(integer)1

hdel user name 🡪(integer)0

**计算field个数**

hlen key

**批量设置或获取field-value**

获取 hmget key field [field]…

设置 hmset key field value [field value]…

**判断field是否存在**

hexsits key field,存在返回1，不存在返回0

**获取所有的field**

hkeys key

**获取所有的value**

hvals key

**获取所有的field-value**

hgetall key

**计算value的字符串长度**

Hstrlen key field

### 内部编码

### 应用场景

将表中整行的信息保存为哈希结构， 相比于使用字符串序列化的缓存用户信息，哈希类型变的更加直观，并且更新操作上会更加便捷；哈希是稀疏的，关系型数据是完全结构化的

缓存用户的3中方式

1. 使用原生的字符串类型：每个属性一个键

优点，简单直观，每个属性都支持

缺点，占用过多的键，内存占用较大，同时用户信息内聚性较差

1. 序列化字符串类型

优点：简化编程，如果合理的使用序列化可以提高内存的使用率

缺点：序列化和反序列化有一定的开销，同时每次更新都需要把全部数据取出来进行反序列化，更新后在序列化保存到redis中；

1. 哈希类型

每个用户属性使用一对field-value，但是只用一个键保存

优点：简单直观，可以减少内存的使用

缺点：Hashtable会消耗更多的内存，要控制哈希在ziplist和hashtable之间的转换

## 集合set

**set是一个无序的、自动去重的集合数据类型，Set底层用两种数据结构存储，一个是hashtable，一个是inset。**

### 内部编码

集合类型的内部编码有两种

1. intset
2. hashtable（哈希表）

### 数据结构

1. 整数集合intset

整数集合是集合的底层实现之一，当元素都是整数，并且元素个数小于512时使用这个数据结构作为set的底层数据结构

元素个数不少于默认值512可以配置set-max-inset-entries 512

typedef struct intset {

uint32\_t encoding;编码方式

uint32\_t length;集合包含的元素数量

int8\_t contents[];保存元素的数组

} intset;

Contents数组：整数集合的每个元素都是放在这个数组中，每个元素都是从大到小有序排列的，并且不包含重复的数据

Length：记录整数集合包含的元素数量，也就是contents的长度

Encoding：决定数组的类型

如果属性值为inset\_enc\_int16,那么contents就是一个int16\_t类型的数组，数组每个值必须是int16\_t类型的整数值（最小为-32768，最大32767）

如果属性值为inset\_enc\_int32,那么contents就是一个int32\_t类型的数组，数组每个值必须是int32\_t类型的整数值（最小为-2147483648，最大2147483647）

如果属性值为inset\_enc\_int64,那么contents就是一个int64\_t类型的数组，数组每个值必须是int64\_t类型的整数值（最小为-9223372036854775808，最大9223372036854775807）

**整数集合的升级**

当新增元素的类型比当前集合所有元素的类型都要长，就需要对整数集合进行升级，然后才能将元素添加到数组中

步骤

1. 根据新元素的类型，扩展整数集合底层数组的空间大小，并为新元素分配空间
2. 将数组的元素转换为新的元素类型，然后放到正确的位置
3. 将新元素放到数组中

升级可以提升整数集合的灵活性，也可以尽量的节省内存

***整数集合不支持降级操作***

1. Hashtable

Hashtable也就是hash表，不过value为null，key为set中的元素值

### 命令

#### 集合内操作

**添加元素**

sadd key element[element…]

sadd myset a b c

集合中没有重复的元素，添加成功返回3

sadd myset b c 元素已经存在了，返回0

**删除元素**

srem key element[element]

sadd myset a c 返回2

删除的元素不存在，返回0

**计算元素个数**

scard key

scard myset

**判断元素是否存在**

sismember key element:在集合中返回1，不在返回0

**随机从集合中返回指定个数的元素**

srandmember key [count] :count是可选参数，如果不写默认为1

**从集合中数据弹出元素**

spop key 从集合中随机弹出一个元素

redis从3.2版本开始，spop也支持[count]参数

**获取所有元素**

smembers key 获取所有元素,并且返回结果是无序的

#### 集合间操作

**求多个集合的交集**

sinter key[key…]

**求多个集合的并集**

sunion key[key…]

**求多个集合的差集**

sdiff key [key…] 在第一个集合中不在第二个集合中

**将交集，并集，差集的结果保存**

sinterstore destination key [key …] 保存交集

sunionstore destination key [key …] 保存并集

sdiffstore destination key [key …] 保存差集

### 使用场景

集合类型比较典型的使用场景是标签；例如一个用户可能对娱乐、体育比较感兴趣，另一个用户可能对历史、新闻比较感兴趣，这些兴趣点就是标签

Sadd = tagging标签

Spop/srandmember =生成随机数，比如抽奖

Sadd+sinter=社交需求，共同爱好或者好友

## 有序集合zset

zset为**有序**（有限score排序，score相同则元素字典序），**自动去重**的集合数据类型

### 内部编码

Ziplist（压缩列表）和skiplist（跳表）

Ziplist：当有序集合个数小于zset-max-ziplist-entries配置（默认为128），同时每个元素的值都小于zset-max-ziplist-value（默认64字节）时，使用ziplist作为内部数据结构，ziplist可以有效减少内存的使用

Skiplist:当不满足ziplist时，有序集合会使用ziplist，因为此时使用ziplist读写效率就会下降

### 数据结构

其底层实现为 字典（dict） + 跳表（skiplist），当数据比较少的时候用ziplist编码结构存储。

同时满足以下两个条件采用ziplist存储：

有序集合保存的元素数量小于默认值128个

有序集合保存的所有元素的长度小于默认值64字节

#### 字典dict

就是hash的数据结构

#### 跳表skiplist

跳表是一个有序的数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问的目的

跳表支持平均o(logN),最坏o(n)复杂度的节点查找，还可以通过顺序操作批量处理节点

跳表节点zskiplistNode(在server.h文件中)

typedef struct zskiplistNode {

sds ele; //数据域

double score; //分值

struct zskiplistNode \*backward; //后向指针，使得跳表第一层组织为双向链表

struct zskiplistLevel { //每一个结点的层级

struct zskiplistNode \*forward; //某一层的前向结点

unsigned int span; //某一层距离下一个结点的跨度

} level[]; //level本身是一个柔性数组，最大值为32，由 ZSKIPLIST\_MAXLEVEL 定义

} zskiplistNode;

层级：跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其他节点的指针，通过这个层来加快访问其他节点，层数越多访问其他节点的速度越快

前进指针：

跨度

后退指针

分值和成员

typedef struct zskiplist {

struct zskiplistNode \*header; //头部

struct zskiplistNode \*tail; //尾部

unsigned long length; //长度，即一共有多少个元素

int level; //最大层级，即跳表目前的最大层级

} zskiplist;

typedef struct zset {

dict \*dict;

zskiplist \*zsl;

} zset;

### 常用命令

#### 集合内

1. 添加成员

zadd key score member [score member…]

1. 计算成员个数

Zcard key

1. 计算某个成员的分数

Zscore key member

1. 计算成员的排名

Zrank key member:分数从低到高排名

Zrevrank key member：分数从高到低排名

1. 删除成员

Zrem key member

1. 增加成员的分数

zincrby key increment member

1. 返回指定排名范围的成员

zrank key start end[withscores] 返回从低到高

zrevrank key start end[withscores] 返回从高到低

1. 返回指定分数范围的成员

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

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

1. 返回指定分数返回成员的个数

zcount key min max

1. 删除指定排名的升序元素

Zremrangebyrank key start end

1. 删除指定分数的范围的成员

Zremrangebyscore key min max

#### 集合间操作

1. 交集

zinterstore destination numkeys key [key…] [weights weight [weight…]] aggregate sum|min|max

destination:交集结果保存到这个键

numkeys:需要做交集的键的个数

key [key…]需要做交集的键

[weights weight [weight…]]每个键的权重，交集计算时，每个键的值会乘以这个权重，默认为1

aggregate sum|min|max:计算交集后，可以按照sum min max这几个公式求出交集的结果，默认为sum

1. 并集

zunionstore destination numkeys key [key…] [weights weight [weight…]] aggregate sum|min|max

### 应用场景

排行榜，点赞

# redis的小功能

## redis慢查询

系统在执行命令前后计算出命令执行的时间，当超过了预设的阈值，就将这条信息记录下来，redis提供了类似的功能

客户端命令执行的步骤

1. 发送命令
2. 命令排队
3. 命令执行
4. 返回结果

慢查询只统计步骤3的时间

### 慢查询的参数配置

慢查询中预设阈值怎么设置，慢查询记录存放在哪里

Riedis提供slowlog-log-slower-than 和slowlog-max-len这两个参数来解决问题

slowlog-log-slower-than就是预设阈值，单位是微妙，默认是10000，，如果一个命令执行时间超过了10000微妙，那么他就会被记录在慢查询日志中；如果slowllog-log-slower-than=0,那么所有的命令都会被记录下来；如果配置slowllog-log-slower-than<0，那么任何命令都不会被记录

slowlog-max-len只是说明慢查询日志最多存储多少条，并没有说明存放在哪里；redis实际上是通过一个列表来存储日志的，slowlog-max-len就是列表的最大长度。当慢查询日志列表中已经到达最大长度时，最早插入的记录就会被移除，

配置修改的方式：

1修改配置文件；

2使用config set命令动态修改，例如 config set slowlog-log-slower-than 200

Config set slowlog-log-len 1000;如果需要就修改持久化到本地，那么就使用config write命令

其他的命令

1. 获取慢查询日志 slowlog-log [n] n表示获取的条数
2. 获取慢查询日志列表的长度 slowlog len
3. 慢查询日志重置 slowlog reset

最佳实践

1. slowlog-max-len:尽量调大慢查询列表，记录满查询是redis会对命令做截断操作，并不会占用大量的内存，增大列表可以减缓慢查询被剔除的可能，比如可以设置为1000
2. slowlog-log-slower-than：默认为10毫秒，需要根据redis并发量调整该值，由于redis是单线程的响应命令，对于高流量场景，如果命令执行时间超过1毫秒，那么redis最大支持ops不到1000，因此对高ops场景的redis建议设置为1毫秒
3. 慢查询日志是一个先进先出的队列，那么在慢查询比较多的情况下会丢失部分慢查询命令，那么就需要将慢查询日志持久化到其他的存储结构中，，然后可以制作可视化的界面进行查询

## Pipeline

Redis客户端执行命令的过程分为4步：

1. 发送命令
2. 命令排序
3. 命令执行
4. 返回结果

第一步和第四步是往返时间round trip time RTT

Redis提供批量操作命令，可以有效的节省rtt,但是大部分命令都不支持批量操作，要执行n次命令，那么就需要消耗n次rtt;

Pipeline机制能改善这类问题，他能将一组redis命令进行组装，通过一次rtt传输给redis，在将这组命令的执行结果按顺序返回客户端

### Java客户端Jedis中使用pipeline

### 原生批量命令和pipeline

区别：

1. 原生批量命令是原子的，pipeline是非原子的
2. 原生批量命令是一个命令对应多个key,pipeline支持多个命令
3. 原生批量命令是redis服务端支持实现的，而pipeline需要服务端和客户端的共同实现

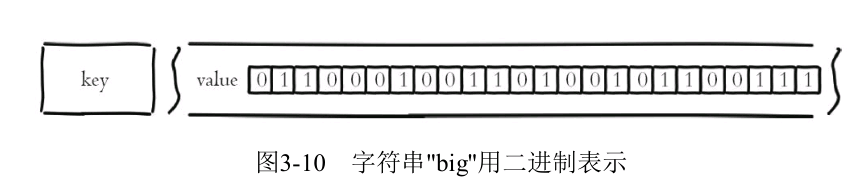
### 最佳实践

当pipeline一次组装过大时，一方面会增加客户端的等待时间，另一方面会造成一定的网络阻塞，可以将一次包含大量命令的pipeline才分为多个较小的pipeline

## Bitmaps

Bitmaps本身不是一种数据结构，它实际上是字符串，但是它可以对**字符串的位**进行操作；通过操作二进制字符串来操作二进制

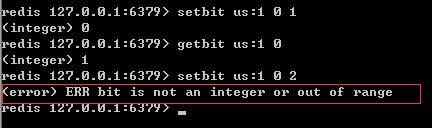
Bitmaps提供了一套命令，可以把bitmaps看作是一个数组，数组的元素就是0和1，数组的下标在bitmaps中叫做偏移量



### 命令

#### 设置值

setbit key offset value 设置键的第office个位的值（从0算起），value只能为0或者1，不是的话就会报错



#### 获取值

getbit key offset:获取键的第offset为的值（从0开始），返回0表示对应位置上没有数据

#### 获取bitmap指定范围1的个数

Bitcount [start] [end]:start和end代表起始和结束字节数

#### Bitmaps间的运算

bitop op destkey key[key…]

bitop是一种复核操作，可以做多个bitmaps的and（交集）,or（并集）,not（非）,xor（异或）操作并将结果保存在destkey（目标key中）

#### 计算bitmaps中第一个targetBit（也就是1）的偏移量

bitop key targetBit [start] [end]

### 应用场景

在超大数量的情况下使用

比如1亿用户的网站，每天独立访问的用户有5千万，使用bitmaps存储活跃的用户比集合类型更加节省空间

但是当用户访问量很少时就不太适合用bitmaps

## HyperLoglog

HyperLogLog并不是新的数据结构（实际类型为字符串），而是一种基数算法；实际类型就是字符串；通过HyperLogLog可以利用极小的空间完成独立总数的统计，数据集可以是IP,Email,ID等

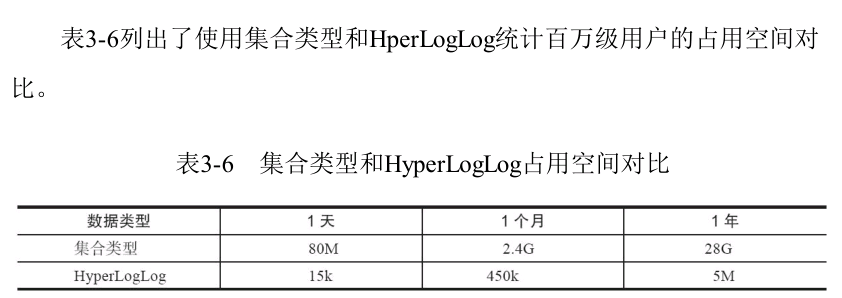
### 命令

添加

pfadd key element[element…]:向HyperLogLog中添加元素，添加成功返回

计算独立用户数(不重复数据)

pfcount key[key…]：计算一个或多个HyperLogLog的独立总数



HyperLogLog内存占用量非常小，但是用如此小的空间来估算巨大的数据，必然不是100%正确的，会有误差存在，redis官方给出的误差率是0.81%

合并

pfmerge destkey sourcekey [sourcekey…]:求出多个hyperLogLog的并集并赋值给destkey

HyperLogLog内存占用量非常小，但是存在错误率，开发者进行数据结构选型时只需要确认如下2条即可：

只为了计算独立总数，不需要获取单条数据

可以容忍一定误差率，毕竟HyperLogLog在内存的占用上有很大的优势

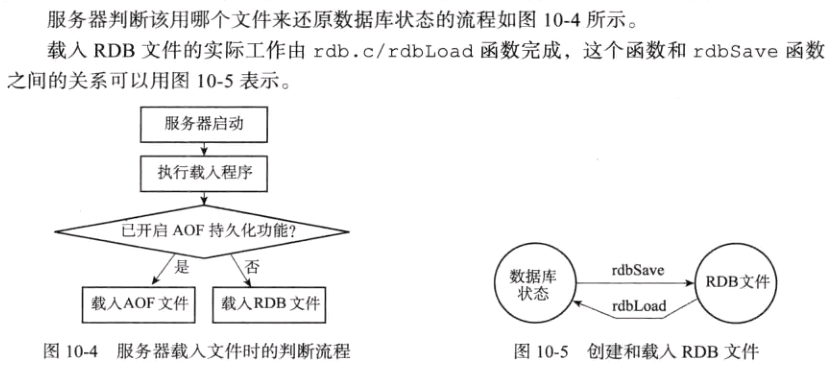
## LUA

# 持久化

Aof文件的更新频率通常比rdb高，所以：

如果服务器开启了aof持久化功能，那么服务器会优先使用aof文件来还原数据

只有在aof功能关闭的时候，服务器才会使用rdb文件来还原数据



## RDB

Rdb持久化是把当前进程数据生成快照保存到硬盘的过程，触发rdb持久化过程分为手动触发和自动触发（根据服务器配置的选项定期执行）；rdb文件是一个经过压缩的二进制文件，通过该文件可以还原生成rdb是的数据

### 触发机制

#### 手动触发

手动触发分别对应save和bgsave命令

Save命令：阻塞当前redis服务器，直到rdb过程完成为止，对于内存教大的实例会阻塞教长时间，线上环境不建议使用

Bgsave命令：redis进程执行fork操作创建子进程，rdb持久化过程由子进程负者，完成后自动释放。阻塞只会出现在fork阶段，一般时间很短

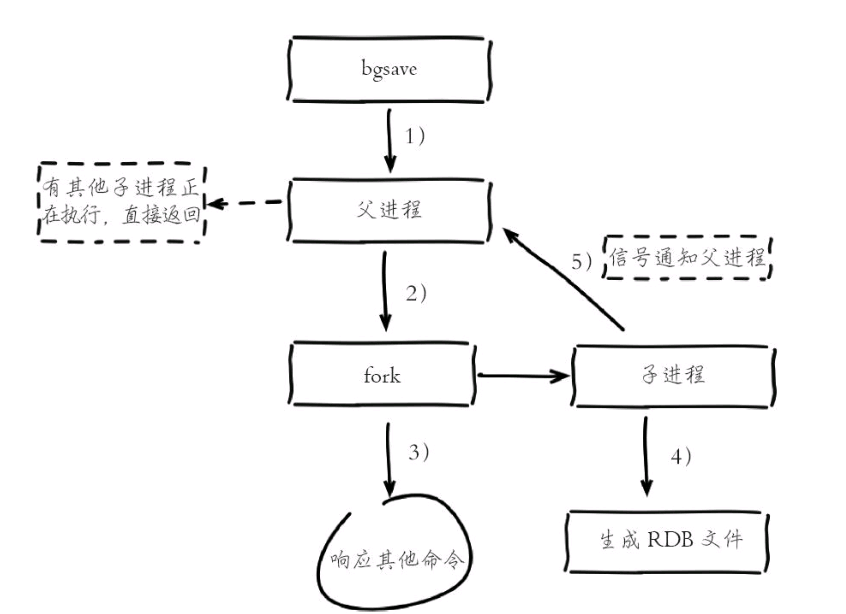
Bgsave是针对save阻塞问题的优化方式，因此redis内存所有的设计rdb操作都采用bgsave的方式，而save命令已经废弃

#### 自动触发

Redis内存存在自动触发rdb的方式

1. 在配置文件中使用save相关的配置比如save m n；m表示在m秒内数据集存在n次修改时，自动触发bgsave
2. 如果从节点执行全量复制，主节点自动执行bgsave生成rdb文件并发送给从节点
3. 执行debug reload命令重新加载redis时，也会自动触发save操作
4. 默认情况下执行shutdown命令时，如果没有开启aof则 自动执行bgsave命令

### Rdb的过程



1. 执行bgsave命令，redis父进程会去判断当前是否存在正在执行的子进程，比如rdb/aof子进程，如果存在bgsave命令直接返回
2. 父进程执行fork操作创建子进程，fork操作会阻塞父进程，通过info stats命令查看lastest\_fork\_usec选项，可以获取最近一个fork操作的耗时，单位是微妙
3. 父进程fork完成后，bgsave命令返回background saving started信息，并且不再阻塞父进程，可以执行其他命令
4. 子进程创建rdb文件，根据父进程生成临时快照文件，快照文件生成后对源文件进行替换。执行lastsave命令可以获取最后一次生成rdb的时间，对应info统计的rdb\_last\_save\_time选项
5. 子进程发送信号给父进程表示完成rdb过程，父进程更新统计信息

底层实现

自动执行的时候

Redis服务器会周期性操作函数serverCrontab默认是100毫秒执行异常，这个函数的作用是对正在运行的服务器进行维护，它的一项工作就是检查save的条件是否已经满足，如果满足就执行bgsave命令

Redis将save的配置的条件保存在redisServer结构的saveparams属性中

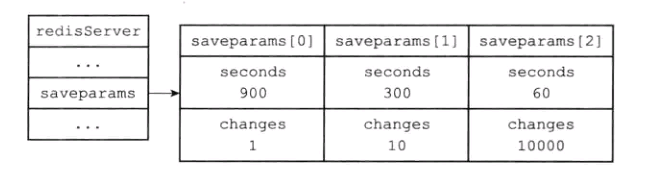
Saveparams是一个数组，每个saveparams的结构都保存了一个save条件

Strut saveparam{

Time\_t seconds 秒数

Int changes 修改次数

}



除了saveparams之外，服务器还维护了一个dirty计数器，已经一个lastsave属性

Dirty计数器：记录距离上一次成功执行save或bgsave命令之后，服务器对数据库状态进行修改的次数

Lastsave属性是unix时间戳,记录上一次服务器成功执行save或bgsave命令的时间

### Rdb文件的处理

保存：rdb文件保存在dir配置指定的目录下，文件名通过dbfilename配置指定；可以通过执行config set dir {newDir}和config set dbfilename{flename}运行期动态执行

当磁盘坏了或者满了，可以通过config set dir {newDir}在线修改文件路径到可以用的磁盘路径下

压缩：redis采用lzf算法对生成的rdb文件做压缩处理，压缩后的文件大小远小于内存大小，默认是开启的，可以通过参数config set rdbcompression{yes|no}修改

压缩会消耗cpu,但是可大幅度降低文件的大下，方便保存可传送，线上建议开启

### Rdb文件结构

完整的rbd文件所包含的各个部分

**redis db\_version databases eof check\_num**

**redis**是rdb文件最开头的部分，长度为5字节，保存着redis这5个字符，通过这5个字符，程序可以在载入文件时，快速检测文件是否是rdb

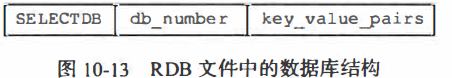
**db\_version** 长度为4字节，值是整数字符串，记录rdb文件的版本号

**database**包含零个或任意个数据库，以及各个数据的键值对数据

**eof**常量的长度为1字节，这个常量标志着rdb文件正文内容的结束，当读入程序遇到这个值的时候，所有的键值对都已加载完毕

**check\_num**是一个8字节长的无符号整数，保存着一个校验和，这个校验和是程序通过对redis db\_version databases eof四个部分的内容进行计算得到的；根据这个值来校验rdb文件是否出错或损坏

**databases部分**

一个RDB文件的databases部分可以保存任意多个**非空数据库**。每个非空数据库在RDB文件中都可以保存为SELECTDB、db number、key value pairs三个部分， 如图10-13所示。  


SELECTDB常量的长度为1字节， 当读入程序遇到这个值的时候， 它知道接下来要读入的将是一个数据库号码。

db number保存着一个数据库号码， 根据号码的大小不同， 这个部分的长度可以是1字节、 2字节或者5字节。 当程序读人db number部分之后， 服务器会调用 SELECT 命令， 根据读人的数据库号码进行数据库切换， 使得之后读入的键值对可以载入到正确的数据库中。

key value pairs部分保存了数据库中的所有键值对数据， 如果键值对带有过期时间， 那么过期时间也会和键值对保存在一起。 根据键值对的数量、类型、 内容以及是否有过期时间等条件的不同， key\_value\_pairs部分的长度也会有所不同

### Rdb的优缺点

优点：

1. Rdb是一个紧凑压缩的二进制文件，代表redis在某个时间点上的数据快照，非常适合·用于备份，全量复制等场景；redis加载恢复数据远快于aof方式
2. Redis加载rdb恢复数据远远快于aof

缺点：

1. rdb不能做到实时的持久化或者秒级持久化，因为bgsave每次运行时要执行fork操作创建子进程，属于重量级操作，频繁执行成本过高，
2. 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

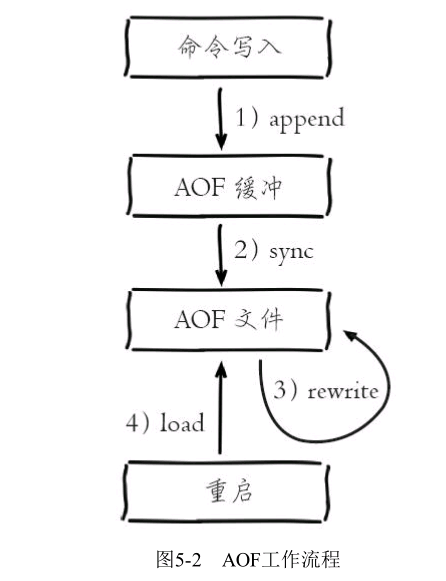
## AOF

Aof持久化：以独立日志的方式记录每次写命令，重启时在重新执行aof文件中的命令达到恢复数据的目的；aof的主要作用就是为了解决数据持久化的实时性

Aof功能默认是不开启的，开始aof功能需要在配置文件中配置参数appendonly yes.aof文件名通过appendfilename配置来设置的，默认文件名appendonly.aof

### 工作流程

命令写入—》文件同步—》文件重写—》重启加载



1. 所有的写入命令会追加到aof\_buf中
2. Aof缓冲区根据对应的策略向硬盘做同步操作
3. 随着aof文件越来越大，需要定期对aof文件进行重写，达到压缩的目的
4. 当redis服务器重启时，可以加载aof文件进行数据恢复

### 命令写入

aof命令写入的内容直接是文本协议格式，将文本追加到aof缓冲区

采用文本协议的好处

1. 文本协议具有很好的兼容性
2. 开启aof后，所有的写入命令都包含在追加操作，直接采用协议格式，避免二次处理开销
3. 文本协议具有可读性，方便直接修改和处理

Aof为什么把命令追加到aof\_buf中

如果每次写aof文件命令都直接追加到硬盘中，那么性能完全取决于当前硬盘负载，先写入缓冲区aof\_buf中，redis可以提供多种缓冲区同步硬盘的策略，在性能和安全上方面做出平衡

### 文件同步

Redis提供多种aof缓冲区同步文件策略，由参赛appendfsync控制，不同的值对应含义如下：

1. always:命令写入aof\_buf后调用fsync操作同步到aof文件，fsync完成后线程返回；每次写入都要同步aof文件，这个和redis的高性能背道而驰，不建议使用
2. everysec:命令写入aof\_buf后调用write操作，write操作完成后线程返回，fsync同步文件操作由专门线程每秒调用一次；建议的同步策略，也是默认配置，做到兼顾性能和数据安全性（理论上在系统宕机的情况下丢失1秒的数据）
3. no:命令写入aof\_buf后调用write操作，不对aof文件进行fsync同步，同步操作由操作系统负者，通常同步周期为30秒;由于操作系统每次同步aof文件的周期不可控，而且会加大每次同步硬盘的数据量，虽然提升了性能，但是不能保证数据的安全性

write操作会触发延迟写机制，write操作在写入缓冲区后直接返回，同步硬盘操作依赖于系统调度机制；在同步之前，如果此时系统出现故障宕机，缓冲区的数据就会丢失

fsync操作：针对单个文件操作，做强制硬盘同步，fsync将阻塞直到写入硬盘完成后返回，保证了数据持久化

### 文件重写

随着命令不断写入aof，文件会越来越大，为了解决这个问题，redis引入aof重写机制来压缩文件体积；

Aof文件重写是把redis进程中的数据转换为写命令同步到新aof文件的过程；就是redis将内存中的数据以命令的形式保存到aof文件中；

Aof文件变小的原因

1. 内存中超时的数据不会被写入文件
2. 旧的aof中含有无效命令，重写使用进程内存中的数据，这样新的aof文件只保存最终的数据的写入命令
3. 多条命令合并为一个，比如：lpush list a,lpush list b,lpush list c会被转换为lpush list a b c;

Aof文件重写降低文件的占用空间，而且更小的aof文件会更快的被redis加载

Aof重写过程可以是手动触发和自动触发

手动触发:bgrewriteaof命令

自动触发：auto-aof-rewrite-min-size和auto-aof-rewrite-percentage参数确定自动触发时机

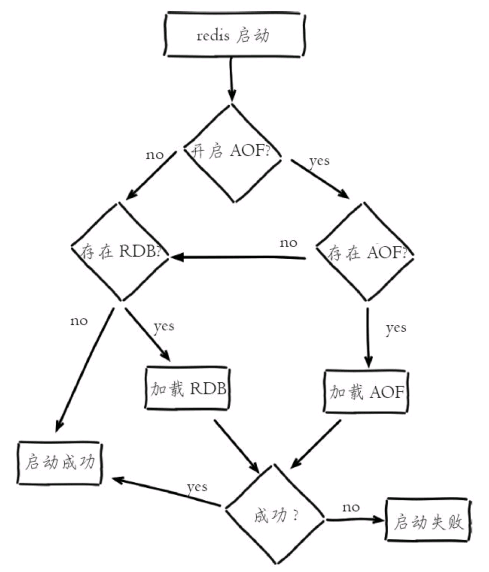
Auto-aof-rewrite-min-size表示运行文件最小体积，默认最小是64MB

auto-aof-rewrite-percentage：当期aof文件空间（aof\_current\_size）和上一次重写aof文件空间(aof\_base\_size)的比值

自动触发的时机:aof\_current\_size>auto\_aof\_rewrite\_min-size&&( aof\_current\_size- aof\_base\_size)/ aof\_base\_size>= auto-aof-rewrite-percentage

### 重启加载

重启恢复数据，redis持久化加载流程



流程说明：

1. aof持久化开启并且存在aof文件，优先加载aof
2. aof关闭或者不存在时，加载rdb文件
3. 加载aof或者rdb文件成功后，redis启动成功
4. Aof/rdb存在错误时，redis启动失败并打印错误日志

文件校验

加载损坏的aof文件时redis会拒绝启动

## 问题定位

常见的持久化问题

### fort操作

当Redis做**RDB或AOF重写时**，一个必不可少的操作就是执行**fork操作创建子进程**，对于大多是系统来说**fork是一个重量级操作**。

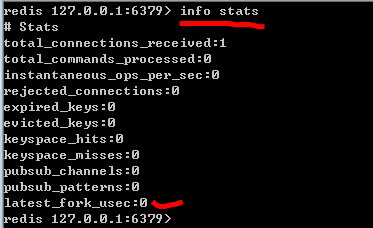
虽然fork创建的子进程不需要拷贝父进程的物理内存空间，但是会复制父进程的空间内存页表。

例如对于10GB的Redis进程，需要复制大约20MB的内存页表，因此fork操作耗时跟进程总内存息息相关，

如果使用虚拟化技术，特别是Xen虚拟机，fork操作会更耗时。

fork耗时问题定位：对于高流量的Redis实例OPS可达5万以上，如果fork操作耗时在秒级别将拖慢Redis几万条命令执行，对线上应用延迟影响非常明显。

正常情况下，fork耗时应该是每GB消耗20毫秒。可以在info stats统计中查latest\_fork\_usec指标获取最近一次fork操作耗时，单位微秒。



如何改善fork操作的耗时：

1）优先使用物理机或者高效支持fork操作的虚拟机技术，**避免使用Xen**。

2）**控制Redis实例最大可用内存**，fork耗时跟内存量成正比，线上建议每个Redis实例内存控制在10GB以内。

3）合理**配置Linux内存分配策略**，避免物理内存不足导致fork失败。

4）**降低fork操作的频率**，如适度放宽AOF自动触发时机，避免不必要的全量复制等

### 子进程开销监控和优化

子进程负责AOF或者RDB文件的重写，它的运行过程主要涉及CPU、内存、硬盘三部分的消耗。

**1.CPU**

（1）CPU开销分析

子进程负责把进程内的数据分批写入文件，这个过程属于CPU密集操作，通常子进程对单核CPU利用率接近90%。

（2）CPU消耗。

Redis是CPU密集型操作，不要做绑定单核CPU的操作。由于子进程非常消耗CPU，会和父进程产生单核资源竞争。

不要和其他CPU密集型服务部署在一起，造成CPU过度竞争。

如果部署多个Redis实例，尽量保证同一时刻只有一个子进程执行重写工作。

**2.内存**

**（1）内存消耗分析**

子进程通过fork操作产生，占用内存大小等同于父进程，理论上需要两倍的内存来完成持久化的操作，但是Linux有写时复制机制。

父进程会共享相同的物理内存页，当父进程处理写请求时会把要修改的页创建副本，而子进程在fork操作过程中共享整个父进程内存快照。

**（2）内存消耗监控**

如果重写过程中存在内存页操作，父进程负责创建所修改内存页的副本，从日志中可以看出这部分的内存消耗。

父进程维护页副本消耗同RDB重写过程类似，不同之处在于AOF重写需要AOF重写缓冲区。

提示：编写shell脚本根据Redis日志可快速定位子进程重写期间内存过度消耗的情况。

**（3）内存消耗优化**

* 同CPU优化一样，如果部署多个Redis实例，尽量保证同一时刻只有一个子进程在工作。
* 避免大量写入时做子进程重写操作，这样将导致父进程维护大量页副本，造成内存消耗。

在Linux kernel在2.6.38内核增加了Transparent Huge pages（THP），支持huge page（2MB）的页分配，默认开启。

当开启时可以降低fork创建子进程的速度，但执行fork之后，如果开启HTP，复制页单位会从原来4KB变为2MB，会大幅增加重写期间父进程的内存消耗。

**3.硬盘**

（1）硬盘开销分析

子进程主要职责是把AOF或者RDB文件写入硬盘持久化，势必会造成硬盘写入压力，

根据Redis重写AOF/RDB的数据量，结合系统工具如sar、iotop等，可分析出重写期间硬盘负载情况。

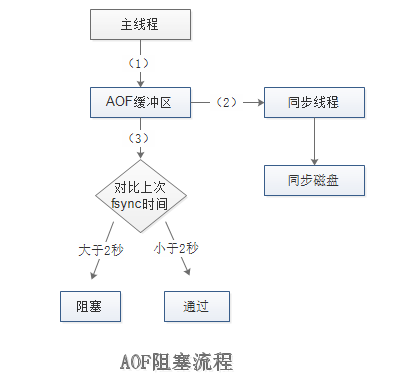
（2）硬盘开销优化

* 不要和其它高硬盘负载的服务部署在一起。如：存储服务、消息队列服务。
* AOF重写时会消耗大量硬盘IO，可以开启配置no-appendfsync-on-rewrite，默认关闭。表示在AOF重写期间不做fsync操作。
* 当开启AOF功能的Redis用于高流量写入场景时，如果使用普通机械磁盘，写入吞吐一般在100MB/s左右，这是瓶颈主要在AOF同步硬盘上。
* 对于单机配置多个Redis实例的情况，可以配置不同实例分盘存储AOF文件，分摊硬盘写入压力。

### AOF追加阻塞

当开启持久化时，常用的同步硬盘的策略是everysec，用于平衡性能和数据安全性。

对于这种方式，Redis使用另一条线程每秒执行fsync同步硬盘。当系统硬盘资源繁忙时，会造成Redis主线程阻塞。



**1.阻塞流程分析：**

（1）主线程负责写入AOF缓冲区。

（2）AOF线程负责每秒执行一次同步操作，并记录最近一次同步时间。

（3）主线程负责每秒执行一次同步磁盘操作，并记录最近一次同步时间。

* 如果距上次同步成功时间在2秒内，主线程直接返回。
* 如果距上次同步时间超过两秒，主线程将会阻塞，直到同步操作完成。

通过分析AOF阻塞流程可以发现两个问题：

（1）everysec配置最多可能丢失2秒数据，不是1秒。

（2）如果系统fsync缓慢，将会导致Redis主线程阻塞影响效率。

**2.AOF阻塞问题定位**

（1）发生AOF阻塞问题，Redis会在日志中记录其行为。

（2）每当发生AOF追加阻塞事件，在info Persistence统计中，aof\_delayed\_fsync指标会累加，查看这个指标方便定位AOF阻塞问题。

（3）AOF同步最多允许2秒延迟，当延迟发生时说明硬盘存在高负载问题，可以通过监控工具如iotop，定位消耗硬盘IO资源的进程。

# 过期策略和淘汰机制

## 过期策略

### Redis设置过期时间

Expire key time（一秒为单位设置过期时间） --最常用的方法

setex(String key, int seconds, String value)--字符串独有的方式**定时删除**

除了字符串自己独有设置过期时间的方法外，其他方法都需要依靠expire方法来设置时间

如果没有设置时间，那缓存就是永不过期

如果设置了过期时间，之后又想让缓存永不过期，使用persist key

### 定时删除

在设置key的过期时间的同时，为key创建一个定时器，定时器在key的过期时间来临时，对key进行删除

优点：能够保存内存被尽快释放

缺点：若过期key很多，**删除这些key会占用很多的CPU时间**，在CPU时间紧张的情况下，CPU不能把所有的时间用来做要紧的事儿，还需要去花时间删除这些key定时器的创建耗时，若**为每一个设置过期时间的key创建一个定时器（将会有大量的定时器产生），性能影响严重**

### 懒汉式删除

Key过期的时候不删除，在通过key获取值的时候去检查时候过期，如果过期就删除，返回null（用的时候判断是否到期，到期就删除）

优点：删除操作只发生在获取值的时候，而且只删除当前key,所以对cpu时间的占用很少；而且此时的删除是已经到了非做不可的地步（如果此时还不删除的话，我们就会获取到了已经过期的key了）

缺点：若**大量的key在超出超时时间**后，很久一段时间内，都**没有被获取过，那么可能发生内存泄露（无用的垃圾占用了大量的内存）**

### 定期删除

每隔一段时间执行一次删除过期key操作

优点：**通过限制删除操作的时长和频率，来减少删除操作对CPU时间的占用--处理"定时删除"的缺点定期删除过期key--处理"懒汉式删除"的缺点**

缺点：**在内存友好方面，不如"定时删除"（会造成一定的内存占用，但是没有懒汉式那么占用内存）在CPU时间友好方面，不如"懒汉式删除"（会定期的去进行比较和删除操作，cpu方面不如懒汉式，但是比定时好）**

难点：**合理设置删除操作的执行时长（每次删除执行多长时间）和执行频率（每隔多长时间做一次删除）（这个要根据服务器运行情况来定了），每次执行时间太长，或者执行频率太高对cpu都是一种压力**。每次**进行定期删除操作执行之后**，需要**记录遍历循环到了哪个标志位**，以便下一次定期时间来时，从上次位置开始进行循环遍历淘汰机制

### Redis采用的过期策略

懒汉式删除+定期删除

懒汉式删除流程：在进行get或setnx等操作时，先检查key是否过期；若过期，删除key，然后执行相应操作；若没过期，直接执行相应操作；

定期删除流程（简单而言，对指定个数的每一个库随机删除小于等于指定个数个过期key）：遍历每个数据库（就是redis.conf中配置的"database"数量，默认为16）检查当前库中的指定个数个key（默认是每个库检查20个key，注意相当于该循环执行20次，循环体是下边的描述）如果当前库中没有一个key设置了过期时间，直接执行下一个库的遍历随机获取一个设置了过期时间的key，检查该key是否过期，如果过期，删除key判断定期删除操作是否已经达到指定时长，若已经达到，直接退出定期删除。

对于**定期删除**，在程序中**有一个全局变量current\_db来记录下一个将要遍历的库**，假设有16个库，我们这一次定期删除遍历了10个，那此时的current\_db就是11，下一次定期删除就从第11个库开始遍历，假设current\_db等于15了，那么之后遍历就再从0号库开始（此时current\_db==0）

### Rdb对过期key的处理

#### 过期key对RDB没有任何影响

从内存数据库持久化数据到RDB文件

* + 持久化key之前，会检查是否过期，过期的key不进入RDB文件

从RDB文件恢复数据到内存数据库

* + 数据载入数据库之前，会对key先进行过期检查，如果过期，不导入数据库（主库情况）

#### ****AOF对过期key的处理****

过期key对AOF没有任何影响

* 从内存数据库持久化数据到AOF文件：
  + 当key过期后，还没有被删除，此时进行执行持久化操作（该key是不会进入aof文件的，因为没有发生修改命令）
  + 当key过期后，在发生删除操作时，程序会向aof文件追加一条del命令（在将来的以aof文件恢复数据的时候该过期的键就会被删掉）
* AOF重写
  + 重写时，会先判断key是否过期，已过期的key不会重写到aof文件

## 淘汰策略

1. **noeviction**：当内存使用超过配置的时候会返回错误，不会驱逐任何键

2. **allkeys-lru**：加入键的时候，如果过限，首先通过LRU算法驱逐最久没有使用的键

3. **volatile-lru**：加入键的时候如果过限，首先从设置了过期时间的键集合中驱逐最久没有使用的键

4. **allkeys-random**：加入键的时候如果过限，从所有key随机删除

5. **volatile-random**：加入键的时候如果过限，从过期键的集合中随机驱逐

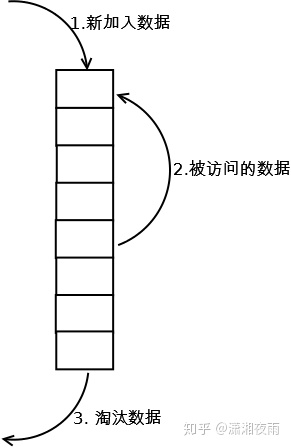
6. **volatile-ttl**：从配置了过期时间的键中驱逐马上就要过期的键

7. **volatile-lfu**：从所有配置了过期时间的键中驱逐**使用频率最少的键**

8. **allkeys-lfu**：从所有键中驱逐使用频率最少的键

### Lru算法

1. 新增key value的时候首先在链表结尾添加Node节点，如果超过LRU设置的阈值就淘汰队头的节点并删除掉HashMap中对应的节点。
2. 修改key对应的值的时候先修改对应的Node中的值，然后把Node节点移动队尾。
3. 访问key对应的值的时候把访问的Node节点移动到队尾即可。



#### Redis中lru的实现

Redis维护了一个24位时钟，可以简单理解为当前系统的时间戳，每隔一定时间会更新这个时钟。每个key对象内部同样维护了一个24位的时钟，当新增key对象的时候会把系统的时钟赋值到这个内部对象时钟。比如我现在要进行LRU，那么**首先拿到当前的全局时钟，然后再找到内部时钟与全局时钟距离时间最久的（差最大）进行淘汰**，这里值得注意的是全局时钟只有24位，按秒为单位来表示才能存储194天，所以可能会出现key的时钟大于全局时钟的情况(全局时钟达到最大值后归0，从新来计时)，如果这种情况出现那么就两个相加而不是相减来求最久的key。

Redis中的LRU与常规的LRU实现并不相同，常规LRU会准确的淘汰掉队头的元素，但是Redis的LRU并不维护队列，只是根据配置的策略要么从所有的key中随机选择N个（N可以配置）要么从所有的设置了过期时间的key中选出N个键，然后再从这N个键中选出最久没有使用的一个key进行淘汰。

#### 为什么要使用近似LRU

1、性能问题，由于近似LRU算法只是最多随机采样N个key并对其进行排序，如果精准需要对所有key进行排序，这样近似LRU性能更高

2、内存占用问题，redis对内存要求很高，会尽量降低内存使用率，如果是抽样排序可以有效降低内存的占用

3、实际效果基本相等，如果请求符合长尾法则，那么真实LRU与Redis LRU之间表现基本无差异

4、在近似情况下提供可自配置的取样率来提升精准度，例如通过 CONFIG SET maxmemory-samples <count> 指令可以设置取样数，取样数越高越精准，如果你的CPU和内存有足够，可以提高取样数看命中率来探测最佳的采样比例。

### Lfu算法

LFU是在Redis4.0后出现的，LRU的最近最少使用实际上并不精确，考虑下面的情况，如果在|处删除，那么A距离的时间最久，但实际上A的使用频率要比B频繁，所以合理的淘汰策略应该是淘汰B。LFU就是为应对这种情况而生的。

A~~A~~A~~A~~A~~A~~A~~A~~A~~A~~~|

B~~~~~B~~~~~B~~~~~B~~~~~~~~~~~~B|

LFU把原来的key对象的内部时钟的24位分成两部分，前16位还代表时钟，后8位代表一个计数器。16位的情况下如果还按照秒为单位就会导致不够用，所以一般这里以时钟为单位。而后8位表示当前key对象的访问频率，8位只能代表255，但是redis并没有采用线性上升的方式，而是通过一个复杂的公式，通过配置如下两个参数来调整数据的递增速度。

lfu-log-factor 可以调整计数器counter的增长速度，lfu-log-factor越大，counter增长的越慢。

lfu-decay-time 是一个以分钟为单位的数值，可以调整counter的减少速度。

所以这两个因素就对应到了LFU的Counter减少策略和增长策略，它们实现逻辑分别如下。

LFU把原来的key对象的内部时钟的24位分成两部分，前16位还代表时钟，后8位代表一个计数器。16位的情况下如果还按照秒为单位就会导致不够用，所以一般这里以时钟为单位。而后8位表示当前key对象的访问频率，8位只能代表255，但是redis并没有采用线性上升的方式，而是通过一个复杂的公式，通过配置如下两个参数来调整数据的递增速度。

lfu-log-factor 可以调整计数器counter的增长速度，lfu-log-factor越大，counter增长的越慢。

lfu-decay-time 是一个以分钟为单位的数值，可以调整counter的减少速度。

所以这两个因素就对应到了LFU的Counter减少策略和增长策略，它们实现逻辑分别如下。

### 降低LFUDecrAndReturn

1、先从高16位获取最近的降低时间ldt以及低8位的计数器counter值

2、计算当前时间now与ldt的差值（now-ldt），当ldt大于now时，那说明是过了一个周期，按照65535-ldt+now计算（16位一个周期最大65535）

3、使用第2步计算的差值除以lfu\_decay\_time，即LFUTimeElapsed(ldt) / server.lfu\_decay\_time，已过去n个lfu\_decay\_time，则将counter减少n。

### 增长LFULogIncr

1、获取0-1的随机数r

2、计算0-1之间的控制因子p，它的计算逻辑如下

*//LFU\_INIT\_VAL默认为5*

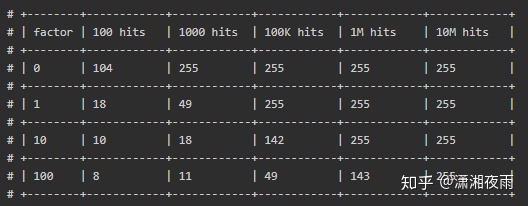
baseval **=** counter **-** LFU\_INIT\_VAL;

*//计算控制因子*

p **=** 1.0**/**(baseval**\***lfu\_log\_factor**+**1);

3、如果r小于p，counter增长1

p取决于当前counter值与lfu\_log\_factor因子，counter值与lfu\_log\_factor因子越大，p越小，r小于p的概率也越小，counter增长的概率也就越小。增长情况如下图：



从左到右表示key的命中次数，从上到下表示影响因子，在影响因子为100的条件下，经过10M次命中才能把后8位值加满到255.

### 新生KEY策略

另外一个问题是，当创建新对象的时候，对象的counter如果为0，很容易就会被淘汰掉，还需要为新生key设置一个初始counter。counter会被初始化为LFU\_INIT\_VAL，默认5。

# 集群

## 主从复制

在分布式系统中，把数据复制到多个副本中部署到其他的机器，来实现负载均衡和故障恢复等需求；redis提供了复制的功能，实现了相同数据的多个副本，复制是redis高可用的基础

### 主从复制的配置

#### 建立主从连接

参与redis复制的实例称为主节点和从节点，默认情况下所有的redis实例都是主节点；每个从节点只要一个主节点，一个主节点可以有多个从节点；复制数据是单向的，只能由主节点复制到从节点

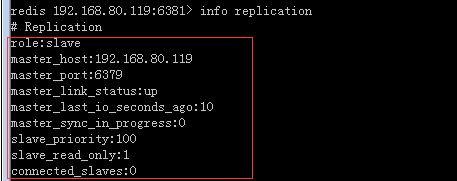
配置主从复制的方式有3种：

1. 在配置文件中加入配置slaveof <masterip> <masterport> redis启动后生效
2. 在redis-server启动后加-- slaveof <masterip> <masterport>命令生效
3. 直接使用slaveof <masterip> <masterport>命令生效

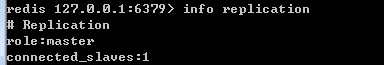
Slaveof可以在运行期间配置，也可以在配置文件中提前写好，最终的效果都是一样的；

Slaveof命令本身是异步命令，执行slaveof命令时，节点只保存主节点信息后返回，后续复制流程在节点内部异步执行，主从节点建立连接后，可以使用info replication命令查看复制相关的状态

从节点的信息



主节点的信息



Windows下启动

cmd下：

redis-server redis.windows.conf

连接命令操作窗口:

redis-cli.exe -h 127.0.0.2 -p 6379

#### 断开主从连接

使用命令slaveof no one

断开复制流程

1. 断开与主节点的主从关系
2. 从节点升级为主节点

从节点不会抛弃复制过来的数据，只是不会再起复制原主节点的数据‘

对于数据比较重要的节点，主节点应该通过设置参数requirepass进行密码验证，客户端必须使用auth命令实行检验

默认情况下从节点使用slave-read-only=yes配置为只读模式；因为从节点的修改主节点是无法感知的，如果修改从节点就会造成数据主从不一致。

传送延迟

Redis提供了repl-disable-tcp-nodelay参数用于控制是否关闭TCP\_NODELAY,默认是关闭的；

当关闭的时候，主节点产生的命令数据无论大小都会及时发送给从节点，这样主从之间的延迟就会减小，但是会增加网络带宽的消耗，适用于网络环境良好的场景，如同机房或同机架部署

当开启时主节点合并较小的TCP数据包从而节省宽带，默认发送时间取决于linux的内存，一般为40毫秒；适用于主从网络环境复制或者带宽紧张的场景，如跨机方部署

部署主从节点时需要考虑网络延迟、宽度使用率，防火级别等因素，如果要求**低延迟**，建议同机架或同机房部署并关闭repl-disable-tcp-nodelay；如果考虑到高容灾，那么可以跨机房部署并且开启repl-disable-tcp-nodelay；

### 主从拓扑结构

**一主一从**

最简单的复制拓扑结构，用于主节点宕机时从节点提供故障转移支持

**一主多从**

使得应用端可以利用多个从节点实现读写分离，对于读占比较大的场景，可以把读命令发送到从节点来分担主节点的压力

树状主从结构

**从节点不但可以复制主节点数据，同时可以作为其他的从节点的主节点继承向下复制**；通过引入复制中间层，可以有效降低主节点负载和需要传送给从节点的数据量；当主节点需要挂载多个从节点时为了避免对主节点的性能干扰，可以采用树状主从结构降低主节点压力

### 复制原理

在从节点执行slaveof命令后，复制过程便开始运作

#### 复制步骤

1. 保存主节点信息

执行slaveof后从节点只保存主节点的地址信息（ip和端口）直接返回，这时建立复制流程还没有开始，

1. 和主节点建立网络连接

从节点内存通过**定时任务来维护复制相关的逻辑**，当定时任务发现存在新的主节点后，会尝试与该节点建立网络连接（socket连接）；从节点会建立socket套接字，专门用于接收主节点发送的复制命令，如果从节点无法建立连接，定时任务会无限重试直到连接成功或者执行slave no one中断复制命令

连接失败时在从节点执行info replication查看master\_link\_down\_since\_seconds指标,它会记录和主节点连接失败的系统时间

1. 连接建立后，发送ping命令（检测主从之间网络套接字是否可用，检测主节点当前是否可以接收处理命令）

如果发送的ping命令后，从节点没有收到主节点的回复或者超时，从节点会断开复制连接，下次定时任务时发起重连

1. 权限验证

如果主节点设置了requirepass参数，则需要密码验证

1. 同步数据集

主从复制连接正常通信后，对于**首次建立复制**的场景，主节点会把所有的数据全部发送给从节点，这部分的操作耗时最长

同步分为全量同步和部分同步

1. 命令持续复制

当主节点把当前的数据同步给从节点后，便完成了复制的建立流程。接下来主节点会持续把命令写入到从节点，保证主从节点的数据一致性

#### 数据同步

Redis在2.8及以后版本使用psync命令来完成主从同步数据，同步分为全量同步和部分同步

**全量同步**

一般用于初次复制场景，把主节点的数据全部一次性的发送给从节点，当数据量较大的时候会对主从节点和网络造成很大的开销；redis早期只支持全量同步

**部分同步**

用于处理主从复制中因**网络中断等原因造成的数据丢失场景**，当从节点再次连上主节点后，如果条件允许，**主节点会补发丢失数据给从节点**；部分复制是对老版本复制的重大优化，有效的避免了不必要的全量复制操作，

Psync命令运行需要以下组件的支持

1. 主节点各自复制偏移量
2. 主节点复制积压缓冲区
3. 主节点运行id

#### 全量复制

全量复制是Redis最早支持的复制方式，也是主从复制建立时必须经历的阶段；触发全量复制的命令是sync和psync

在redis2.8版本之前使用的是sync命令，在redis2.8之后（包括2.8）使用的是psync命令；

Psync的流程

1. 发送psync命令进行数据同步，由于是第一次进行复制，从节点没有复制偏移量和主节点的运行id，所以发送psync-1
2. 主节点根据psync-1知道当前是全量复制，就回复+fullresync响应,
3. 从节点接收到响应数据后保存运行id和偏移量
4. 主节点执行bgsave保存rdb文件到本地，保存完成后发送rdb文件给从节点，从节点接收到文件后保存在本地并且把它作为数据文件
5. 对于从节点接收rdb文件开始到接收这段时间，主节点仍会继续响应读写命令；因此主节点会把这期间写命令数据保存在复制客户端缓冲区内，当节点加载完rdb文件之后，主节点在把缓存区的数据发送给从节点，保证主从数据一致性

如果主节点创建和传送的rdb文件时间过长，对于高流量的并发场景非常容易造成缓冲区溢出。默认配置为client-output-buffer-Limite slave256mb64mb60,如果60秒内缓冲区消耗持续大于64mb或者直接超过256mb.主节点会直接关闭复制客户端连接，造成全量同步失败

1. 从节点接收完主节点传送的数据后，清空自身旧数据，清空后开始加载rbd文件，对于较大的rdb文件，这一步操作会比较耗时
2. 加载完rdb后，如果当前节点开启了aof持久化功能，它会立刻做bgrewriteaof操作（aof重写）

全量复制流程中消耗时间操作

1. 主节点bgsave的时间，也就是将数据写入到rbd这个过程
2. Rbd文件网络传输
3. 从节点清空数据时间
4. 从节点加载rdb文件的时间
5. 可能的重写aof时间

#### 部分复制

部分复制主要针对全量复制一些过大开销作出的一些优化，使用psync{runid}{offset}命令实现；当从节点正在复制主节点时，如果网络出现闪断或者命令丢失等情况，从节点回向主节点发送补发丢失的命令，如果主节点的复制积压缓存区内部存在这部分数据则直接发送给从节点，这样就可以保证主从节点的数据一致性，补发的数据一般远小于全量数据，所以开销很小

部分复制流程

1. 主从节点出现网络中断，如果超时repl-timeout时间，主节点会认为从节点故障并中断连接
2. 主从连接中断期间主节点依然响应命令，但是连接中断命令无法发送给从节点，主节点内部存在的复制积压缓冲区，依然可以保存最近一段时间的写命令数据，默认最大缓存1MB
3. 主从节点网络恢复后，从节点再次连接到主节点，由于从节点保存了自身已复制的偏移量和主节点的运行Id.因此会把他们当作psync参数发送给主节点，要求进行部分复制操作
4. 主节点接收到psync命令后会首先核对参数runid是否与自身一致，如果一直说明之前复制的是当前主节点；之后根据参数offset在自身复制积压缓冲区查找，如果偏移量之后的数据在缓冲区，则对从节点发送+Continue响应，表示可以进行部分复制
5. 主节点根据偏移量把复制积压缓冲区中的数据发送给从节点保证主从复制进入正常状态

#### 心跳

主从节点建立连接之后，他们之间会维护长连接并彼此发送心跳命令，主从心跳判断机制

1） 主从节点彼此都有心跳检测机制， 各自模拟成对方的客户端进行通信， 通过client list命令查看复制相关客户端信息， 主节点的连接状态为flags=M， 从节点连接状态为flags=S。

2） 主节点默认每隔10秒对从节点发送ping命令，**判断从节点的存活性和连接状态**。 可通过参数repl-ping-slave-period控制发送频率。

3） 从节点在主线程中每隔1秒发送replconf ack{offset}命令， **给主节点上报自身当前的复制偏移量**。

replconf命令主要作用如下

**·**实时监测主从节点网络状态。  
**·**上报自身复制偏移量， 检查复制数据是否丢失，**如果从节点数据丢失， 再从主节点的复制缓冲区中拉取丢失数据**。  
**·**实现保证从节点的数量和延迟性功能， 通过min-slaves-to-write、 minslaves-max-lag参数配置定义。

主节点根据replconf命令判断从节点超时时间， 体现在info replication统计中的lag信息中， lag表示与从节点最后一次通信延迟的秒数， 正常延迟应该在0和1之间。 如果超过repl-timeout配置的值（默认60秒） ， 则判定从节点下线并断开复制客户端连接。 即使主节点判定从节点下线后， 如果从节点重新恢复， 心跳检测会继续进行。

为了降低主从延迟， 一般把Redis主从节点部署在相同的机房/同城机房， 避免网络延迟和网络分区造成的心跳中断等情况。

异步复制

**主节点不但负责数据读写， 还负责把写命令同步给从节点**。 写命令的发送过程是异步完成， 也就是说主节点自身处理完写命令后直接返回给客户端， 并不等待从节点复制完成

主节点复制流程：  
1） 主节点接收处理命令。  
2） 命令处理完之后返回响应结果。  
3） 对于修改命令异步发送给从节点， 从节点在主线程中执行复制的命令。由于主从复制过程是异步的， 就会造成从节点的数据相对主节点存在延迟。 具体延迟多少字节， 我们可以在主节点执行info replication命令查看相关指标获得。

**Redis的复制速度取决于主从之间网络环境， repl-disabletcp-nodelay， 命令处理速度等。 正常情况下， 延迟在1秒以内。**

**Redis实现读写分离时，从节点响应读请求时，可能会遇到的问题：**

**复制数据延迟**

**复制数据的延迟是由于异步复制的特性造成的无法避免，延迟取决于网络带宽和命令阻塞情况；需要业务场景允许短时间内的数据延迟。对于无法容忍大量延迟的场景，可以通过外部监控程序监听注册节点的复制偏移量，当延迟较大时触发报警或者通知客户端避免读取延迟较高的从节点**

**读到过期数据**

**当主节点设置大量超时的数据时，Redis内部要维护过期数据删除策略，删除的方式有2中：惰性删除和定时删除**

**惰性删除：在主节点每次处理读取命令时，都会检查键是否超时，如果超时则执行del命令删除键对象，之后del命令也会异步同步到从节点，因为为了保持数据的一致性，从节点是不会主动去删除超时数据的**

**定时删除：Redis主节点在内部定时任务会定时判断键是否过期，过期就执行del命令，之后再同步到从节点**

**当主节点判断键过期的速度并且主节点没有读取过期键，那么从节点将无法收到del命令，那么访问从节点就会读取到过期数据**

**Redis3.2版本解决了这个问题，从节点读取数据时会去判断是否过期来决定是否返回数据**

**从节点故障**

**可以使用哨兵机制来监控节点是否存活；或者使用Rediscluster**

## 哨兵机制

哨兵模式是建立在主从复制模式下的，主从模式的弊端是不具有高可用性，当master挂掉后，redis就不能提供写功能，需要人工干预进行故障转移，无论是对Redis的应用还是运维都会带来很大的不变

### 作用

当主节点出现故障，Redis sentinel能自动完成故障发现和故障转移，并通知应用方，从而实现高可用

主要作用

1. 监控：sentinel会定期监控Redis数据节点和其他的sentinel节点是否可用
2. 通知：sentinel节点会将故障转移的结果通知给应用方
3. 主节点故障转移：选举出一个主节点，实现从节点晋升为主节点并维护主从关系
4. 配置提供者：客户端在初始化的时候连接点的是sentinel节点，从中获取主节点信息

Sentinel自身也是Redis的节点，只不过它不存数据，只存部分命令

哨兵(sentinel)是一个分布式系统,你可以在一个架构中运行多个哨兵(sentinel)进程,这些进程使用流言协议(gossipprotocols)来接收关于Master是否下线的信息,并使用投票协议(agreement protocols)来决定是否执行自动故障迁移,以及选择哪个Slave作为新的Master.

每个哨兵(sentinel) 会向其它哨兵(sentinel)、master、slave定时发送消息,以确认对方是否”活”着,如果发现对方在指定时间(可配置)内未回应,

则暂时认为对方已挂(所谓的”主观认为宕机” Subjective Down,简称sdown).若“哨兵群”中的多数sentinel,都报告某一master没响应,系统才认为该master"彻底死亡"(即:客观上的真正down机,Objective Down,简称odown),通过一定的vote算法,从剩下的slave节点中,选一台提升为master,然后自动修改相关配置.

虽然哨兵(sentinel) 释出为一个单独的可执行文件 redis-sentinel ,但实际上它只是一个运行在特殊模式下的 Redis 服务器，

你可以在启动一个普通 Redis 服务器时通过给定 --sentinel 选项来启动哨兵(sentinel).

### 工作流程

\* 每个sentinel以每秒钟一次的频率向它所知的master，slave以及其他sentinel实例发送一个 PING 命令

\* 如果一个实例距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被sentinel标记为主观下线。

\* 如果一个master被标记为主观下线，则正在监视这个master的所有sentinel要以每秒一次的频率确认master的确进入了主观下线状态

\* 当有足够数量的sentinel（大于等于配置文件指定的值）在指定的时间范围内确认master的确进入了主观下线状态， 则master会被标记为客观下线

\* 在一般情况下， 每个sentinel会以每 10 秒一次的频率向它已知的所有master，slave发送 INFO 命令

\* 当master被sentinel标记为客观下线时，sentinel向下线的master的所有slave发送 INFO 命令的频率会从 10 秒一次改为 1 秒一次

\* 若没有足够数量的sentinel同意master已经下线，master的客观下线状态就会被移除；

若master重新向sentinel的 PING 命令返回有效回复，master的主观下线状态就会被移除

当使用sentinel模式的时候，客户端就不要直接连接Redis，而是连接sentinel的ip和port，由sentinel来提供具体的可提供服务的Redis实现，

这样当master节点挂掉以后，sentinel就会感知并将新的master节点提供给使用者。

### 配置详解

配置sentinel节点

redis-sentinel.conf文件中配置

port 26379 // 当前Sentinel服务运行的端口

daemonize yes

logfile “log.log”

dir /xx/xx

sentinel monitor mymaster 127.0.0.1 6379 2 配置监控的节点就是127.0.0.1这个机器上端口6379这Redis的实例，2代表判断主节点失败至少需要2个sentinel节点同意，mymaster主节点的别名

sentinel down-after-milliseconds mymaster 5000

sentinel parallel-syncs mymaster 1

sentinel failover-timeout mymaster 15000

port和dir分别代表sentinel节点的端口和工作目录

#### sentinel monitor

**sentinel monitor <master-name> <ip> <port> <quorum>**:配置监控的节点，

<master-name>主节点的别名，ip主节点的ip地址，port主节点的端口号，

<quorum>判定主节点不可达所需要的票数，比如2代表判断主节点失败至少需要2个sentinel节点同意

同时<quorum>还与sentinel节点的领导者选择有关，至少要有max（quorum，num（sentinels）/2+）个sentinel节点参与选举，才能选出领导者

sentinel down-after-milliseconds <master-name> <times>:配置发送ping命令有效回复的时间，超过这个时间没有有效的恢复，那么就判定为节点不可达，times就是超时时间

down-after-milliseconds越大那么sentinel节点对于监控节点的不可达的条件越宽松，反之越严格

#### parallel-syncs

**sentinel parallel-syncs <master-name> <nums>**

故障转移后选出新的主节点，原来的从节点就会向新节点发送复制操作，**parallel-syncs就是用来限制在一次故障之后，每次向新主节点发起复制操作的节点的个数**

#### failover-timeout

sentinel failover-timeout <master-name> <times>

failover-timeout通常被解释成故障转移超时时间，但实际上它作用于故障转移的各个阶段：  
a）选出合适从节点。  
b）晋升选出的从节点为主节点。  
c）命令其余从节点复制新的主节点。  
d）等待原主节点恢复后命令它去复制新的主节点。

 failover-timeout的作用具体体现在四个方面：

1）如果Redis Sentinel对一个主节点故障转移失败，那么下次再对该主节点做故障转移的起始时间是failover-timeout的2倍。

2）在b）阶段时，如果Sentinel节点向a）阶段选出来的从节点执行slaveof no one一直失败（例如该从节点此时出现故障），当此过程超过failover-timeout时，则故障转移失败。

3）在b）阶段如果执行成功，Sentinel节点还会执行info命令来确认a）阶段选出来的节点确实晋升为主节点，如果此过程执行时间超过failovertimeout时，则故障转移失败。

 4）如果c）阶段执行时间超过了failover-timeout（不包含复制时间），则故障转移失败。注意即使超过了这个时间，Sentinel节点也会最终配置从节点去同步最新的主节点。

#### sentinel auth-pass

sentinel auth-pass <master-name> <password>

如果Sentinel监控的主节点配置了密码，sentinel auth-pass配置通过添加主节点的密码，防止Sentinel节点对主节点无法监控。

#### sentinel notification-script

sentinel notification-script <master-name> <script-path>

 sentinel notification-script的作用是在故障转移期间，当一些警告级别的Sentinel事件发生（指重要事件，例如-sdown：客观下线、-odown：主观下线）时，会触发对应路径的脚本，并向脚本发送相应的事件参数。

例如在/opt/redis/scripts/下配置了notification.sh，该脚本会接收每个Sentinel节点传过来的事件参数，可以利用这些参数作为邮件或者短信报警依据：

1. #!/bin/sh
2. *#获取所有参数*
3. msg=$\*
4. *#报警脚本或者接口，将msg作为参数*
5. exit 0

如果需要该功能，就可以在Sentinel节点添加如下配置（<mastername >= mymaster）

sentinel notification-script mymaster /opt/redis/scripts/notification.sh

例如下面就是某个Sentinel节点对主节点做了主观下线后脚本收到的参数：

#### sentinel client-reconfig-script

sentinel client-reconfig-script <master-name> <script-path>

 sentinel client-reconfig-script的作用是在故障转移结束后，会触发对应路径的脚本，并向脚本发送故障转移结果的相关参数。和notification-script类似，可以在/opt/redis/scripts/下配置了client-reconfig.sh，该脚本会接收每个Sentinel节点传过来的故障转移结果参数，并触发类似短信和邮件报警：

1. #!/bin/sh
2. *#获取所有参数*
3. msg=$\*
4. *#报警脚本或者接口，将msg作为参数*
5. exit 0

 如果需要该功能，就可以在Sentinel节点添加如下配置（<mastername>= mymaster）

sentinel client-reconfig-script mymaster /opt/redis/scripts/client-reconfig.sh

当故障转移结束，每个Sentinel节点会将故障转移的结果发送给对应的脚本，具体参数如下

<master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>

·<master-name>：主节点名。  
·<role>：Sentinel节点的角色，分别是leader和observer，leader代表当前 Sentinel节点是领导者，是它进行的故障转移；observer是其余Sentinel节点。  
·<from-ip>：原主节点的ip地址。  
·<from-port>：原主节点的端口。  
·<to-ip>：新主节点的ip地址。  
·<to-port>：新主节点的端口。

例如以下内容分别是三个Sentinel节点发送给脚本的，其中一个是leader，另外两个是observer：

1. mymaster leader start 127.0.0.1 6379 127.0.0.1 6380
2. mymaster observer start 127.0.0.1 6379 127.0.0.1 6380
3. mymaster observer start 127.0.0.1 6379 127.0.0.1 6380

 有关sentinel notification-script和sentinel client-reconfig-script有几点需要  
注意：  
·<script-path>必须有可执行权限。  
·<script-path>开头必须包含shell脚本头（例如#！/bin/sh），否则事件发  
生时Redis将无法执行脚本产生如下错误：

-script-error /opt/sentinel/notification.sh 0 2

·Redis规定脚本的最大执行时间不能超过60秒，超过后脚本将被杀掉。  
·如果shell脚本以exit 1结束，那么脚本稍后重试执行。如果以exit 2或者更高的值结束，那么脚本不会重试。正常返回值是exit 0。  
·如果需要运维的Redis Sentinel比较多，建议不要使用这种脚本的形式来进行通知，这样会增加部署的成本。

### 启动

## redis cluster

redis cluster在Redis3.0版本推出。当遇到单机内存，并发，流量等瓶颈时，可以使用cluster架构方案达到负载均衡的目的

数据分布

分布式数据库首先要解决的是把整个数据集按照分区规则映射到多个节点的问题，即把数据集划分到多个节点上，每个节点负责整体数据的一个子集。

数据分区规则有哈希分区和顺序分区

哈希分区：离散度好，数据分布和业务无关，无法顺序访问

顺序分区：离散度易倾斜，数据分布和业务相关，可以顺序访问

Redis cluster采用的就是哈希分区

哈希分区的几种规则

1. 节点取余分区

使用特定的数据，例如redis的键或者用户id，在根据节点数量N使用公式hash（key）%N计算出哈希值（通过哈希函数计算出哈希值，然后和节点数取余），用来决定数据映射到那个节点

1. 一致性哈希分区

系统为每个节点分配一个token，范围一般在0到2的23次方，这些token构造一个哈希环。查找需要操作的节点时，根据key计算hash值，然后顺时针找到第一个大于等于哈希值的token节点就是要操作的节点；

这种方式相比于节点取余最大的好处就是加入和删除节点值影响哈希环中相邻的节点，对其他节点没有影响

但是也存在几个问题

加减节点会造成哈希环中部分数据无法命中，需要手动处理或者忽略这部分数据，因此一致性哈希分区常用于缓存场景

当使用少量节点时，节点变化将大范围影响哈希环中的数据映射，这种方式不适合少量数据节点的分布式方案

普通的哈希一致性分区在增减节点时需要增加一倍或者减去一半节点才能保证数据和负载的均衡

1. 虚拟槽分区

使用分散度良好的哈希函数把所有数据映射到一个固定范围的整数集合中，整数定义为槽（slot）;这个范围一般大于节点数，redis cluster槽范围是0~16383；槽是集群内数据管理和迁移的基本单位，采用大范围槽的主要目的就是为了方便数据拆分和集群扩展;每个节点会负责一定数量的槽

Redis采用的是虚拟槽分区，所有的键根据哈希函数映射到0-16383整数槽内，计算公式就是CRC16(key)&16383,将键映射到槽上，每个节点负责维护一部分槽已经槽所映射的键值数据

集群功能的限制

Redis集群相对单机功能存在一些限制

1. key批量操作支持有限
2. key事务操作支持有限
3. key作为数据分区的最小粒度，因此不能将一个大的键值对象如hash，list等映射到不同的节点
4. 不支持多数据库空间，单机下的Redis可以支持16个数据库，集群模式下只能使用一个数据库
5. 复制结构只支持一层，从节点只能复制主节点

### 搭建集群

#### 集群配置

#### 节点间的通信

#### 分配槽

#### 集群伸缩

节点增加节点减少

#### 请求路由

# redis的阻塞

## Redis自身导致阻塞的原因

1. APi或数据结构使用不合理
2. Cpu饱和
3. 持久化相关的阻塞

### APi或数据结构使用不合理

通常Redis执行命令速度非常快，但是也存在例外，比如对一个包含上万个元素的hash结构执行hgetall操作，由于数据量比较大且命令的算法的复杂度是O(n)，这条命令必然执行速度很慢

如何发现慢查询

Redis是提供了慢查询统计的功能，执行slowlog get n命令可以获取最近的n条满查询记录，发现慢查询后需要作出调整，可以从2个方向去调整

1. 修改为低算法度的命令：入hgetall改为hmget,禁用key\* sort等命令
2. 调整大对象，将大对象数据或大对象拆分为小对象防止一条命令操作过多数据

如何发现大对象

Redis命令Redis-cli-h {ip}-p{port} bigkeys.内部原理采用分段进行scan操作，把历史扫描过的最大对象统计出来

### Cpu饱和

Redis把单核cpu使用率跑到了接近100%，Top命令可以查看Redis进程的 cpu使用率；cpu饱和将会导致Redis无法处理更多的命令，严重影响吞吐量和应用的稳定性；

判断Redis的并发量是否到达了极限，使用命令 Redis-cli-h{ip}-p{port}—start获取当前Redis的使用情况；**当Redis的并发达到极限的时候就需要做集群来水平话扩展分摊ops压力**

### 持久化阻塞

持久化引起主线程阻塞的操作主要有：fork阻塞，aof刷盘阻塞，hugePage写操作阻塞

1. fork阻塞

在rdb和aof持久化时，Redis主线程会调用fork操作产生共享内存的子进程，由子进程完成持久化操作；**fork操作自身耗时过长就会导致主线程阻塞**

优先使用物理机或者高效支持fork操作的虚拟机技术，**避免使用Xen**。

**控制Redis实例最大可用内存**，fork耗时跟内存量成正比，线上建议每个Redis实例内存控制在10GB以内。

合理**配置Linux内存分配策略**，避免物理内存不足导致fork失败。

**降低fork操作的频率**，如适度放宽AOF自动触发时机，避免不必要的全量复制等

1. aof刷盘阻塞

使用aof持久化功能时，文件刷盘的方式一般是每秒一次，后天线程每秒对aof文件做fsync操作，当硬盘压力过大时，fsync操作需要等待，直到写入完成。如果主线程发现距离上一次fsync成功超过2秒，就会阻塞自己直到执行完fsync操作，这主要是硬盘压力引起的

优化硬盘负载

* 不要和其它高硬盘负载的服务部署在一起。如：存储服务、消息队列服务。
* AOF重写时会消耗大量硬盘IO，可以开启配置no-appendfsync-on-rewrite，默认关闭。表示在AOF重写期间不做fsync操作。
* 当开启AOF功能的Redis用于高流量写入场景时，如果使用普通机械磁盘，写入吞吐一般在100MB/s左右，这是瓶颈主要在AOF同步硬盘上。
* 对于单机配置多个Redis实例的情况，可以配置不同实例分盘存储AOF文件，分摊硬盘写入压力。

1. hugePage写操作阻塞

子进程在执行重写期间利用Linux写时复制技术降低内存开销， 因此只有写操作时Redis才复制要修改的内存页。 对于开启Transparent HugePages的操作系统， 每次写命令引起的复制内存页单位由4K变为2MB， 放大了512倍， 会拖慢写操作的执行时间， 导致大量写操作慢查询。 例如简单的incr命令也会出现在慢查询中。

## 阻塞的外在原因

### Cpu竞争

进程竞争：Redis是cpu密集型应用，不建议和其他多核cpu密集型服务器部署在一起，当进程过多消耗cup时，会影响吞吐量

绑定cpu：部署Redis时可以利用多核cpu，通常一台机器上部署多个实例；常见的优化方式就是将Redis进程绑定到cpu上，用于降低cpu频繁切换上下文的开销。

开启了持久化或参与复制的主节点不建议绑定cpu:当父进程创建子进程进行rdb或者aof重写时，如果做了cpu绑定，会和父进程共有一个cup，父进程和子进程将产生cpu竞争，极大的影响Redis的稳定性

### 内存交换

操作系统将Redis使用的部分内存换出到硬盘，内存和硬盘的读写速度差几个数量级，会导致发送交换后的Redis性能急剧下降

识别Redis内存交换的方法：

* 1. 查询Redis进程号
  2. 根据进程号查询内存交换信息

预防内存交换的方法：

保证机器的内存充足

确保Redis的实例设置了最大可用内存，防止极端情况下Redis内存不可控的增长

降级系统使用swap的优先级

### 网络问题

#### 连接拒绝

网络闪断：

Redis连接拒绝：Redis通过参数maxclients设置最大的连接数量，默认是10000；当连接数量超过maxclients时会拒绝新的连接

连接溢出

#### 网络延迟

客户端到Redis服务器之间的网络环境，主要包括**物理拓扑和带宽占用**；

物理拓扑按网络延迟由低到高可分为：同物理机<同机架<跨机架<同机房<同城机房<异地机房

网络延迟一般出现在跨机房的部署结构上，如果延迟较大，就需要调整拓扑结构；比如把客户端和Redis服务部署到同机房或同城机房

带宽瓶颈：

机器网卡带宽

机架交换机带宽

机房之间专线带宽

#### 网卡软中断

单个网卡队列只能使用一个cpu，高并发网卡数据交互都集中在一个cpu上，导致无法充分利用多核cpu，一般出现在网络高流量吞吐量的场景

由软件系统本身发给操作系统内核的中断信号，称之为软中断。通常是由硬中断处理程序或进程调度程序对操作系统 内核的中断，也就是我们常说的系统调用(System Call)

# 缓存设计

## 缓存收益和代价

### 收益

加快读写速度：缓存通常是全内存的，而存储层通过读写性能不够强悍，通过缓存可以有效的加快读写，优化用户体验

降低后端负载：帮助减少后端的访问量和复杂计算，降低后端的负载

### 代价

数据不一致性：缓存层和存储层的数据存在一定的时间不一致

代码维护成本：加入缓存层后，需要处理缓存层和存储层的逻辑

运维成本：

## 缓存的使用场景

开销大的复杂运算：比如在mysql中，存在大量联表操作，一些分组计算，如果不加入缓存，既不能满足高并发，同时也会给mysql带来极大的压力

加速请求响应：

## 缓存更新策略

缓存中的数据通常具有生命周期，需要在指定时间之后删除，保证缓存空间在一个可控的范围内，缓存中的数据和数据源中的数据有一段时间的不一致，需要利用某些策略来进行更新

### Lru/lfu/fifo算出剔除策略

使用场景，缓存空间使用超过了预设值；例如Redis使用maxmemory-policy这个配置作为内存的最大值后对数据剔除

一致性：要清除哪些数据是由算法本身决定的，开发人员只是决定使用那种算法，数据一致性最差

维护成本：一般通过配置即可实现

### 超时剔除

使用场景：给缓存的数据设置了过期时间才会使用，让其过期后自动删除，例如Redis的命令expire

一致性：一段时间窗口内（取决于数据过期时间的长短）存在一致性，

维护成本：维护成本不高，设置过期时间即可

### 主动更新

使用场景：对数据一致性要求高时，需要在真实数据更新后立即缓存数据

一致性：数据一致性高，真实数据更新完成后立即更新缓存

维护成本：比较高，开发者需要自己来完成更新，并保证操作的一致性

### 最佳实践

低一致性业务建议使用配置最大内存和淘汰策略的方式

高一致性业务可以结合超时剔除和主动更新，主动更新出了问题，也能保证数据过期后删除脏数据

## .缓存雪崩

在一个时间点上，大量的key过期失效，同时有大量的请求访问进来，这些请求没有在缓存中获取到数据就直接去访问数据库，导致数据库压力

剧增而崩溃

与缓存击穿的区别在于缓存雪崩针对很多key缓存，缓存击穿则是某一个key。

解决方法

让key过期时间不要在同一个时间点上，把失效时间分散一点

热点数据不过期

## 缓存击穿

当某个key过期失效后，大量请求同时访问这个key时，没有获取到数据就去访问数据库，导致数据库压力剧增而崩溃

key可能会在某些时间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。

解决方法

热点数据不过期

使用互斥锁（用细粒度的锁，锁住对这个key访问的线程）

## 缓存穿透

大量请求同时访问缓存中不存在的key时，没有不能获取数据，从而直接访问数据库

导致数据库压力剧增而崩溃

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，并且出于容错考虑，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。

解决方法

1.校验数据，如果key不在规定的范围内就直接返回；增加校验，比如用户鉴权校验，

参数做校验，不合法的参数直接代码Return，比如：id 做基础校验，id <=0的直接拦截等。

2.给这个key的value设置为null，保存在缓存中（使用细粒度的互斥来实现）

从缓存取不到的数据，在数据库中也没有取到，这时也可以将对应Key的Value对写为null、

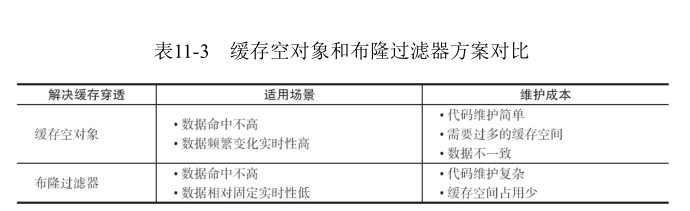
位置错误、稍后重试这样的值具体取啥问产品，或者看具体的场景，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。

这样可以防止攻击用户反复用同一个id暴力攻击，

3.布隆过滤器：

他的原理也很简单就是利用高效的数据结构和算法快速判断出你这个Key是否在数据库中存在，

不存在你return就好了，存在你就去查了DB刷新KV再return



## 高并发场景需要考虑的因素

当访问量确实超过了一定量时，就需要有考虑使用，限流、降级、熔断等措施来避免数据库崩溃

事前：Redis 高可用，主从+哨兵，Redis cluster，避免全盘崩溃。

事中：本地 ehcache 缓存 +\*\* Hystrix\*\* 限流+降级，避免MySQL被打死。

事后：Redis 持久化 RDB+AOF，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据Redis数据一致性

# 应用

## 分布式锁redisson

### 加锁机制

线程去获取锁，**获取成功**: 执行lua脚本，保存数据到redis数据库。

线程去获取锁，**获取失败:** 一直通过while循环尝试获取锁，获取成功后，执行lua脚本，保存数据到redis数据库。

### watch dog自动延期机制

在一个分布式环境下，假如一个线程获得锁后，突然服务器宕机了，那么这个时候在一定时间后这个锁会自动释放，你也可以设置锁的有效时间(不设置默认30秒），这样的目的主要是防止死锁的发生。

### 可重入锁

### 释放锁

### Redis分布式锁的缺点

其实上面那种方案最大的问题，就是如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。

但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，**导致各种脏数据的产生**。

所以这个就是redis cluster，或者是redis master-slave架构的**主从异步复制**导致的redis分布式锁的最大缺陷：在redis master实例宕机的时候，可能导致多个客户端同时完成加锁。

缓存查询等

读写分离

消息队列

布隆过滤器