零声学院 Mark老师 QQ: 2548898954

主题

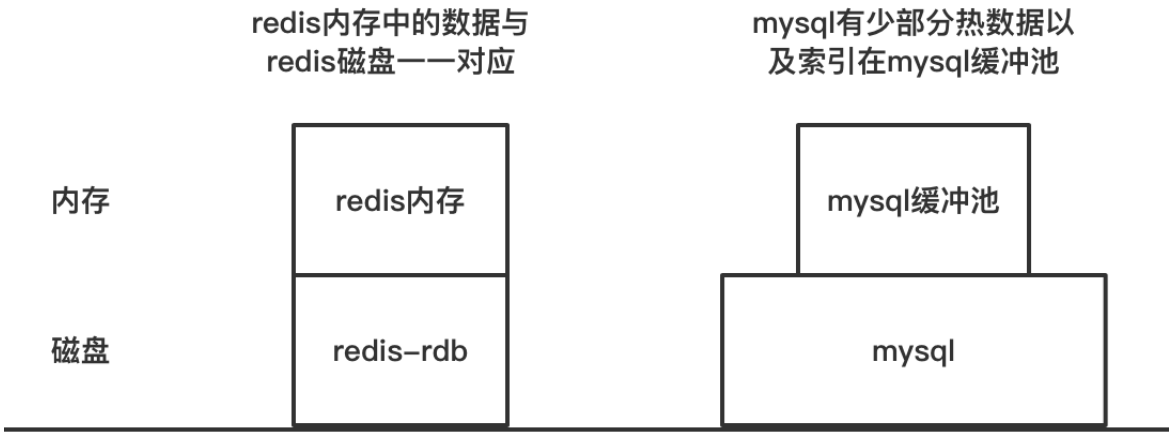
1. 相同数量级的数据，hash存储与zset存储内存占用情况?
2. 布隆过滤器如何解决缓存穿透的问题?
3. redis持久化有哪几种，适用场景有哪些?

如何来学习linux后台开发?

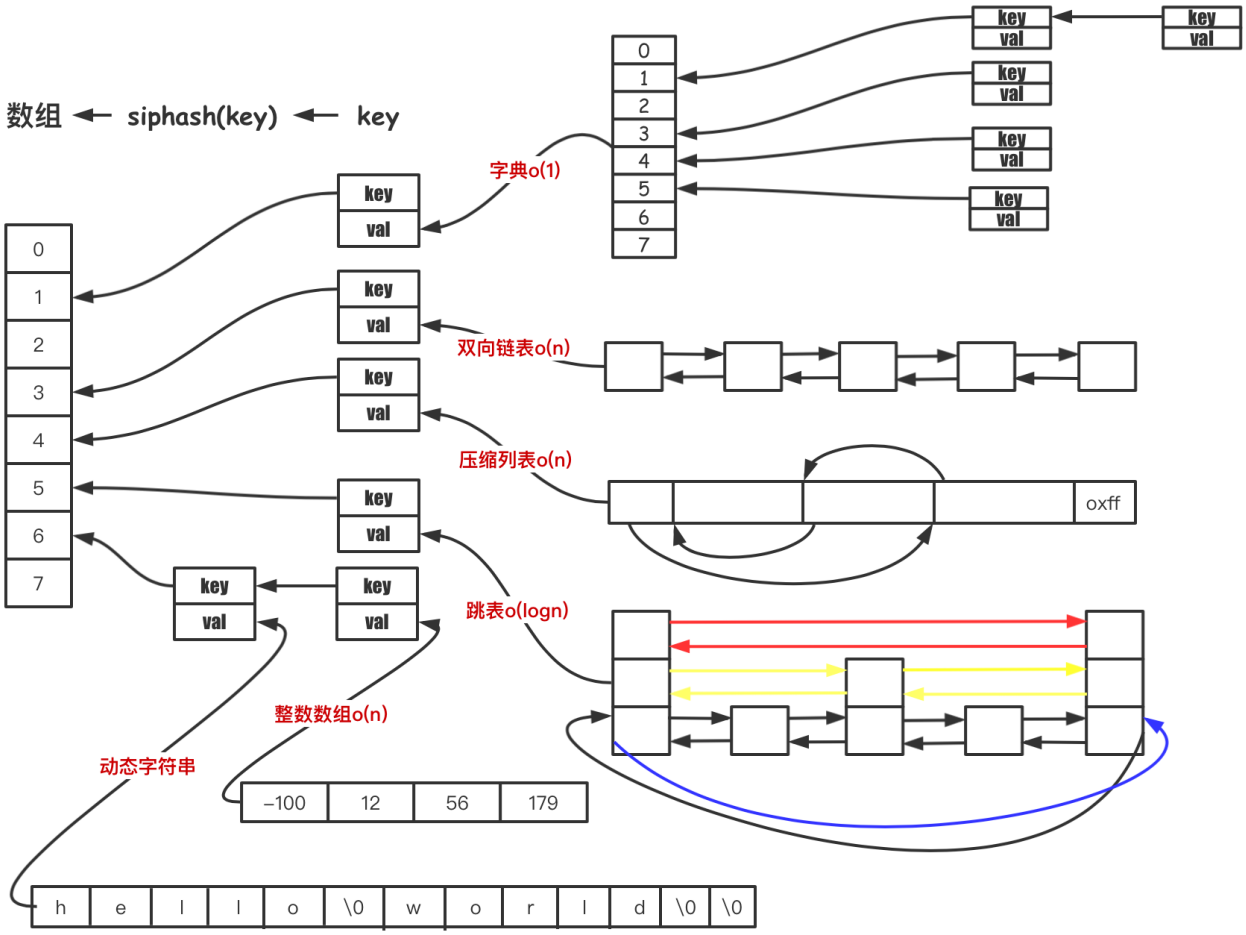
1. 框架、工具以及解决方案
2. 理论知识 知识体系的构建
3. 软实力 安全能力，代码能力，工程素养、架构能力、运营能力 DevOps

redis简介

> redis是内存数据库，kv数据库，数据结构数据库；
>
> 内存数据库，说明数据都在内存中；非内存数据库，比如mysql，数据主要在磁盘中，部分热点数据在缓冲层中；
>
> kv数据库，数据的组织方式，访问操作redis，通过key来索引value从而操作redis；
>
> 数据结构数据库，redis提供丰富的高效的数据结构供客户操作；redis中的value包含string，list，hash，set，zset等多种数据结构；

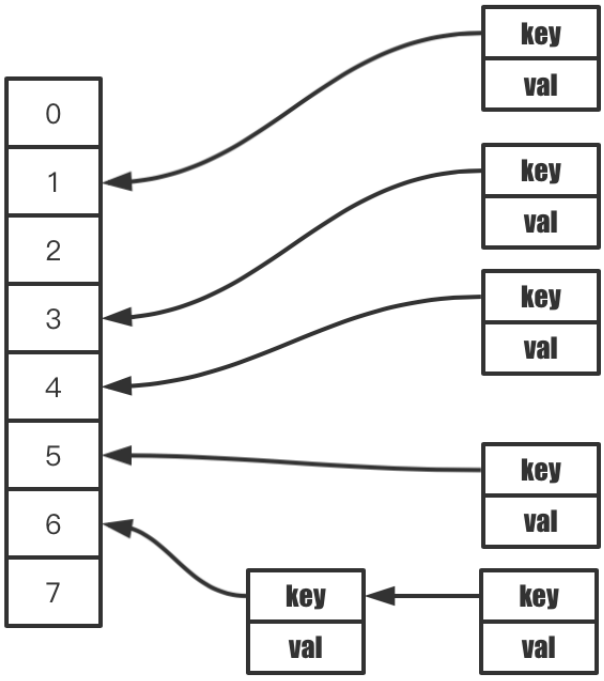


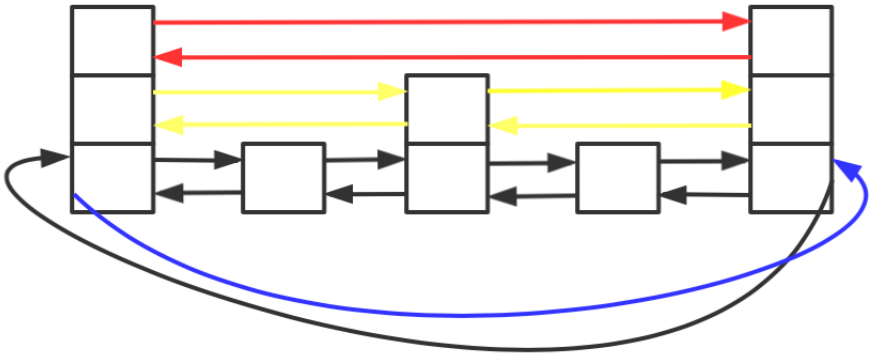
redis存储结构图



hashtable与跳表存储

数据结构

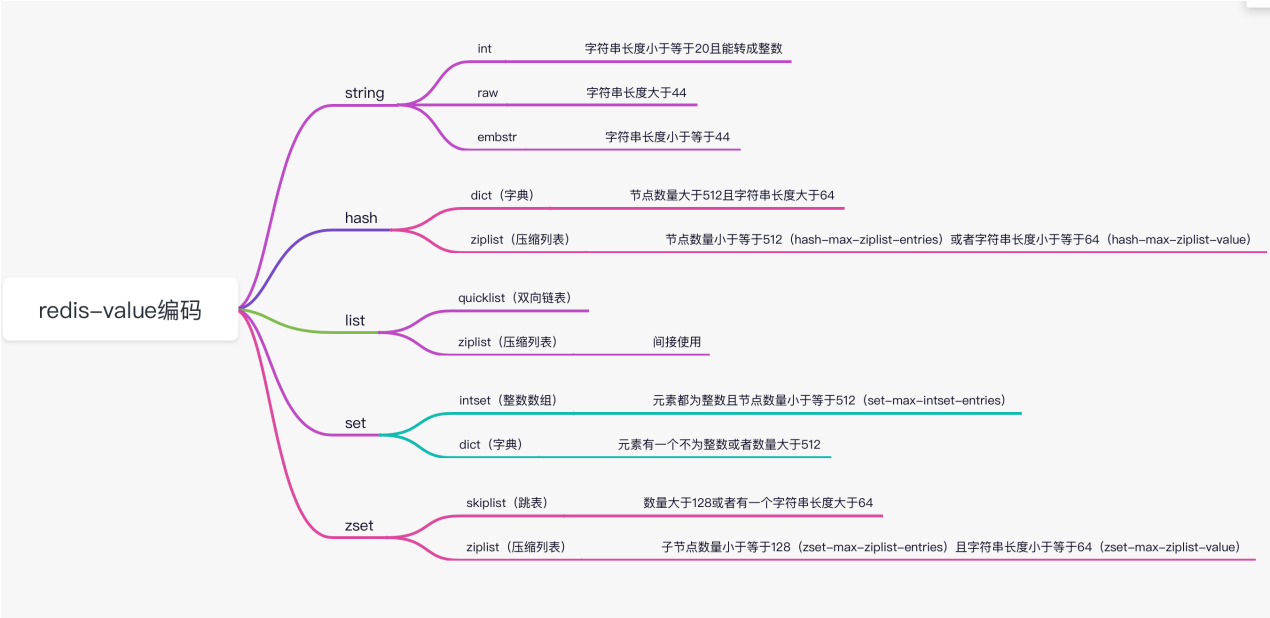




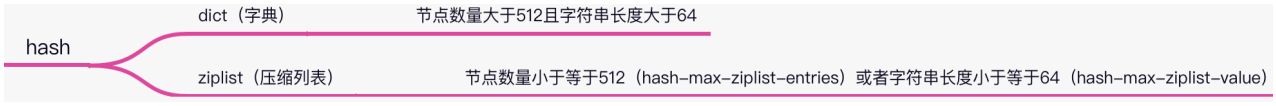
共同点

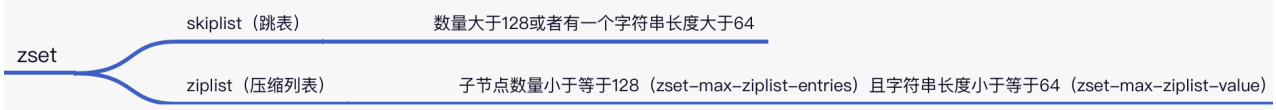
> hashtable和跳表都是空间换时间的数据结构；

redis-value编码



存储比较





代码

```c
typedef struct dictEntry {
    void *key;
    union {
        void *val; // 可以指向不同类型  string list hash stream set zset
        uint64_t u64; // 用于redis集群 哨兵模式 选举算法
```

```
        int64_t s64; // 记录过期时间
        double d;
    } v;
    struct dictEntry *next;
} dictEntry;
/* This is our hash table structure. Every dictionary has two of this as we
 * implement incremental rehashing, for the old to the new table. */
typedef struct dictht {
    dictEntry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;

typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    unsigned long iterators;
} dict;

/* Returns a random level for the new skiplist node we are going to create.
 * The return value of this function is between 1 and ZSKIPLIST_MAXLEVEL
 * (both inclusive), with a powerlaw-alike distribution where higher
 * levels are less likely to be returned. */
#define ZSKIPLIST_P 0.25      /* Skiplist P = 1/4 */
int zslRandomLevel(void) {
    int level = 1;
    while ((random()&0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
        level += 1;
    return (level<ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
}

/* ZSETs use a specialized version of Skiplists */
typedef struct zskiplistNode {
    sds ele;
    double score;
    struct zskiplistNode *backward;
    struct zskiplistLevel {
        struct zskiplistNode *forward;
        unsigned long span;
    } level[];
} zskiplistNode;

typedef struct zskiplist {
    struct zskiplistNode *header, *tail;
```

```
    unsigned long length;
    int level;
} zskiplist;
```

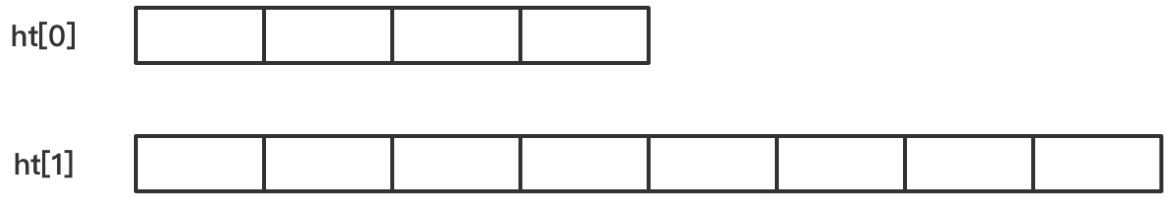
字典结构

- **负载因子**

> $factor = \frac{used}{size}$ used 为 hashtable 中包含的元素个数，size 为 hashtable 的数组长度；

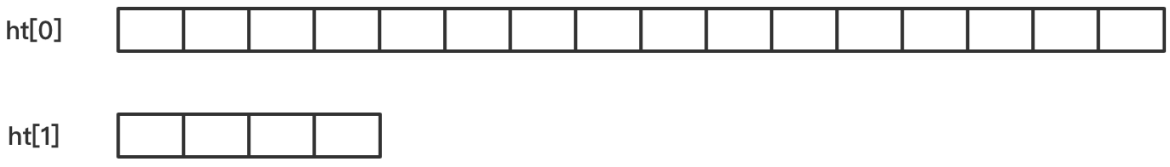
- **扩容**

> 在redis没有进行fork操作的时候（bgsave，bgrewriteaof），hashtable的负载因子大于等于1则进行扩容；
>
> 如果在redis进行fork操作的时候（bgsave，bgrewriteaof），只有hashtable的负载因子大于等于5才会进行扩容
>
> 扩容策略是翻倍；
>
> 数组初始长度为4；



- **缩容**

> 当hashtable的负载因子小于0.1，也就是当节点数量小于数组长度的$\frac{1}{10}$的时候，则进行缩容；
>
> 缩容策略是数组长度恰好包含节点数量的2^k；假设缩容后的节点数量为10个，则数组长度为$2^4 = 16$个；



- **渐进式rehash**

> 扩容和缩容的时候，hashtable需要将ht[0]里面的所有键值对rehash到ht[1]里面；但是这个rehash动作并不是一次性的、集中式的完成，而是分多次、渐进式完成的；
>
> 两种rehash方式：
>
> 1. 每次增删改查，rehash 1次（数组某个槽位中所有数据）；
> 2. 在redis没有进行fork操作的时候（bgsave，bgrewriteaof），定时执行，执行1ms，在这一毫秒中，每次执行100次（数组相邻100个槽位中所有数据）；

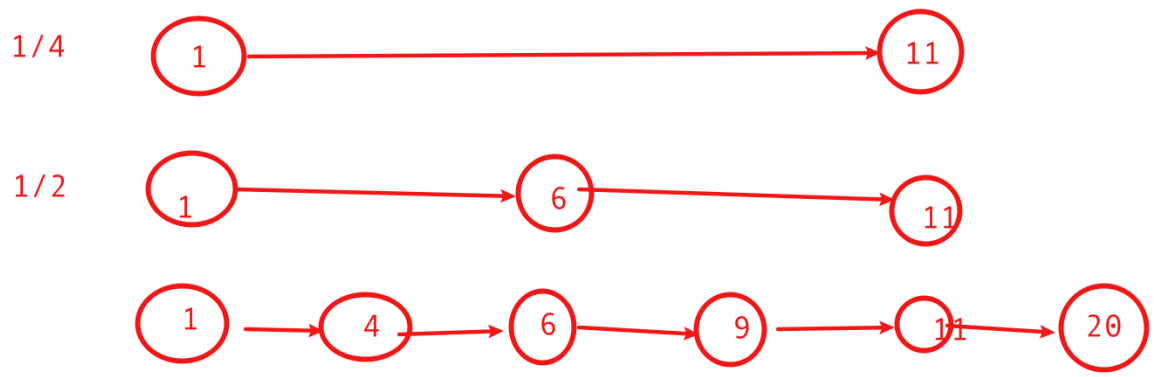
```
static void _dictRehashStep(dict *d) {
```

```
    if (d->iterators == 0) dictRehash(d,1);
}

int dictRehash(dict *d, int n) {
    int empty_visits = n*10; /* Max number of empty buckets to visit.
*/
    if (!dictIsRehashing(d)) return 0;

    while(n-- && d->ht[0].used != 0) {
        dictEntry *de, *nextde;

        /* Note that rehashidx can't overflow as we are sure there are
more
         * elements because ht[0].used != 0 */
        assert(d->ht[0].size > (unsigned long)d->rehashidx);
        while(d->ht[0].table[d->rehashidx] == NULL) {
            d->rehashidx++;
            if (--empty_visits == 0) return 1;
        }
        de = d->ht[0].table[d->rehashidx];
        /* Move all the keys in this bucket from the old to the new
hash HT */
        while(de) {
            uint64_t h;

            nextde = de->next;
            /* Get the index in the new hash table */
            h = dictHashKey(d, de->key) & d->ht[1].sizemask;
            de->next = d->ht[1].table[h];
            d->ht[1].table[h] = de;
            d->ht[0].used--;
            d->ht[1].used++;
            de = nextde;
        }
        d->ht[0].table[d->rehashidx] = NULL;
        d->rehashidx++;
    }

    /* Check if we already rehashed the whole table... */
    if (d->ht[0].used == 0) {
        zfree(d->ht[0].table);
        d->ht[0] = d->ht[1];
        _dictReset(&d->ht[1]);
        d->rehashidx = -1;
        return 0;
    }

    /* More to rehash... */
    return 1;
```

```c
}

int incrementallyRehash(int dbid) {
    /* Keys dictionary */
    if (dictIsRehashing(server.db[dbid].dict)) {
        dictRehashMilliseconds(server.db[dbid].dict,1);
        return 1; /* already used our millisecond for this loop... */
    }
    /* Expires */
    if (dictIsRehashing(server.db[dbid].expires)) {
        dictRehashMilliseconds(server.db[dbid].expires,1);
        return 1; /* already used our millisecond for this loop... */
    }
    return 0;
}

int dictRehashMilliseconds(dict *d, int ms) {
    long long start = timeInMilliseconds();
    int rehashes = 0;

    while(dictRehash(d,100)) {
        rehashes += 100;
        if (timeInMilliseconds()-start > ms) break;
    }
    return rehashes;
}
```

跳表结构

通过增加层级的方式，增加查找的效率 o(n)
理想跳表结构 有什么问题 增删改查
概率来解决 从一个节点出发 每次插入的时候 就给概率生成层级 1/2
数学期望来说 趋向于 理想跳表结构 二分查找 o(logn)
概率 大量数据（128）才会趋向稳定 o（logn）



> redis 跳表结构 插入一个节点概率为1/4增加一个层级 ， 最高层级为32层；

- **多层级有序链表**
- **概率性数据结构**

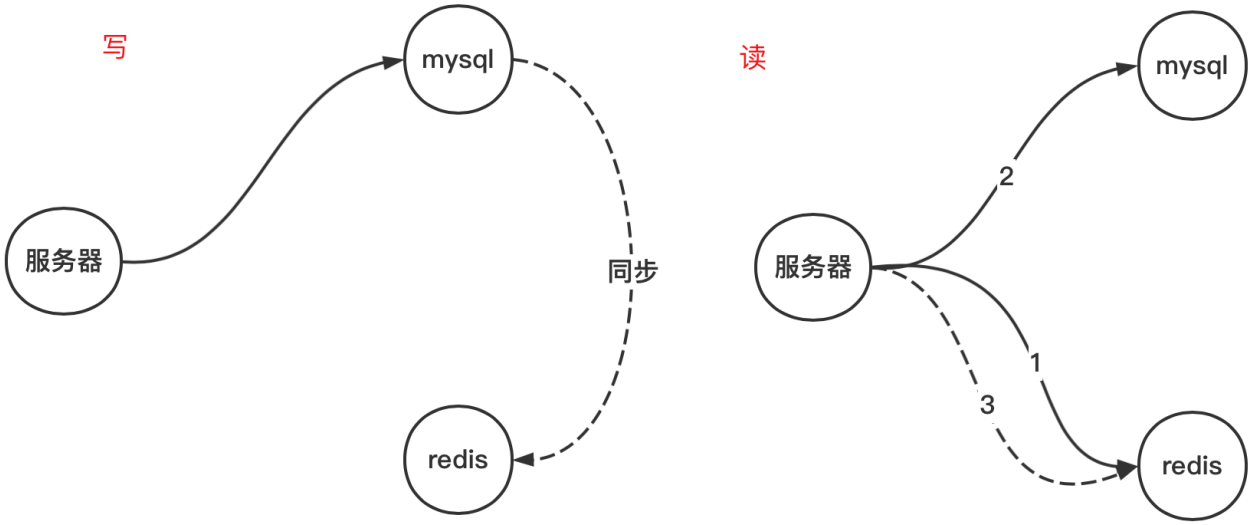
答案分析

> 跳表结构，通过增加搜索层级来提升搜索效率；
> 字典结构，需要说明，扩容，缩容，渐进式rehash，以及条件；
>
> 1. 当子节点数目 $0 < n <= 128$
>
>   此时hash和zset都采用压缩列表存储，内存占用相当
> 2. 当子节点数目 $128 < n <= 512$
>
>   hash采用压缩列表，zset采用跳表存储；此时hash占用内存小，zset占用内存大；
> 3. 当子节点数目 $n > 512$
>
>   此时hash采用字典实现，zset采用跳表存储；
>
>   跳表存储：$n + \frac{1}{4}n + \frac{1}{4^2}n + ... + \frac{1}{4^n}n < 1.5n$
>
>   字典存储：
>
>   - 没有扩容和缩容：最差的情况 5n，最好的情况n；
>   - 进行扩容：没有发生渐进式rehash，2n；发生渐进式rehash，3n；
>   - 进行缩容：90%空间浪费，10n；如果发生渐进式rehash，则更多；

缓存穿透

场景

> 1. 内存的访问速度是磁盘访问速度的10万倍；
> 2. mysql缓冲池不由用户来控制，也就是不能由用户来控制缓存具体数据；
> 3. redis存储的只是用户自定义的热点数据；数据的热度是相对的，所以需要给key添加过期时间；
> 4. 一般大部分项目读频次是写频次的10倍左右；
> 5. redis解决快速读问题，避免走mysql磁盘；



描述

> 假设某个数据redis不存在，mysql也不存在，而且一直尝试读怎么办？缓存穿透，数据最终压力依然堆积在mysql，可能造成mysql不堪重负而崩溃;

解决

> 1. 发现mysql不存在，将redis设置为 <key, nil> 下次访问key的时候 不再访问mysql 容易造成redis缓存很多无效数据；
> 2. 布隆过滤器，将mysql当中已经存在的key，写入布隆过滤器，不存在的直接pass掉；

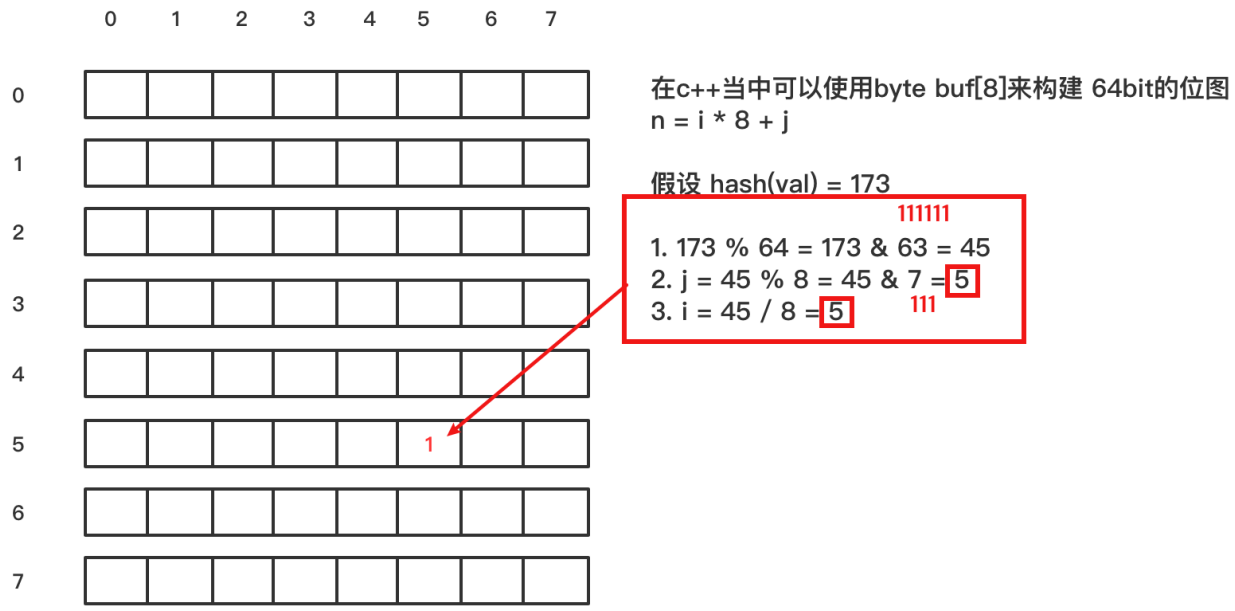
布隆过滤器

定义

> 布隆过滤器是一种**概率型**数据结构，它的特点是高效的**插入和查询**，能明确告知某个字符串**一定不存在**或者**可能存在**；

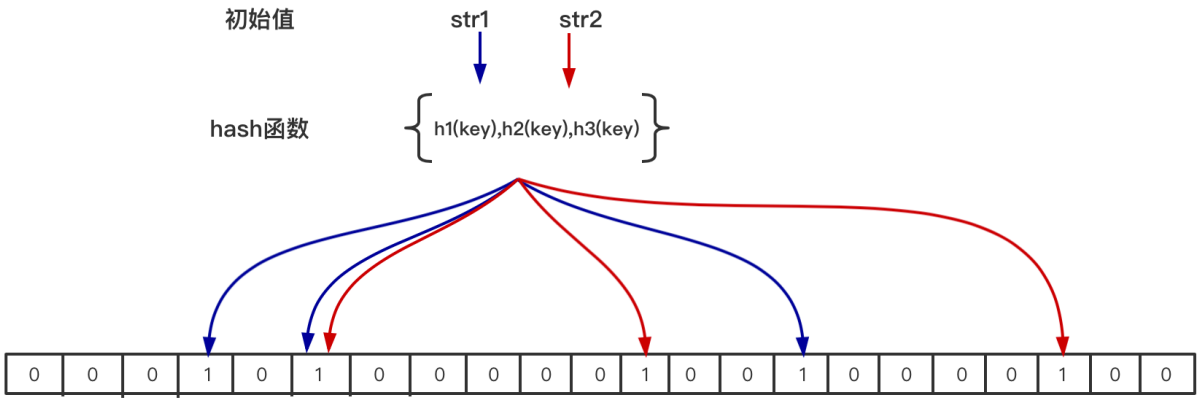
组成

> 位图（bit数组）+ n个hash函数



原理

> 当一个元素加入位图时，通过k个hash函数将这个元素映射到位图的k个点，并把它们置为1；当检索时，再通过k个hash函数运算检测位图的k个点是否都为1；如果有不为1的点，那么认为不存在；如果全部为1，则可能存在（存在误差）；



为什么不支持删除

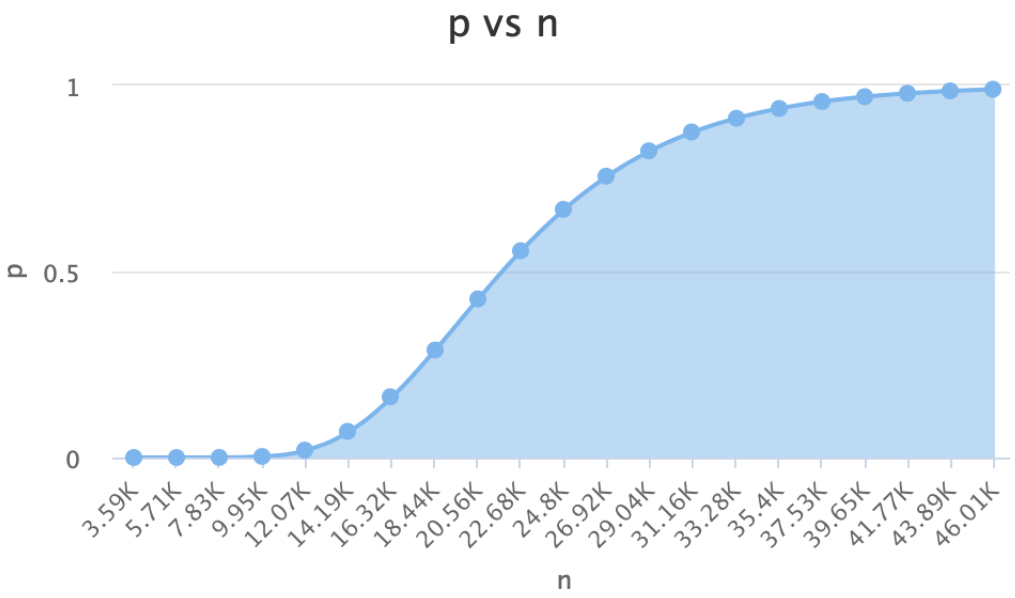
> 在位图中每个槽位只有两种状态（0或者1），一个槽位被设置为1状态，但不明确它被设置了多少次；也就是不知道**被多少个str1哈希映射**以及是**被哪个hash函数**映射过来的；所以不支持删除操作；

推导

```
n                -- 布隆过滤器中元素的个数，如上图 只有str1和str2 两个元素 那么 n=2
p                -- 假阳率，在0-1之间  0.000000
m                -- 位图所占空间
k                -- hash函数的个数

公式如下：
n = ceil(m / (-k / log(1 - exp(log(p) / k))))
p = pow(1 - exp(-k / (m / n)), k)
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));
k = round((m / n) * log(2));
```

应用

> 实际应用过程中，提供 n 和 p，通过公式算出 m 和 k；
>
> 比如：我们提供 n=4000 p = 0.000000001; 我们算出 m = 172532 k = 30



布隆过滤器部署

> 1. 应用层，实现布隆过滤器算法；
> 2. redis：git clone https://github.com/RedisBloom/RedisBloom.git

```
# 加载布隆过滤器
MODULE LOAD redisbloom.so
# 创建一个布隆过滤器
BF.RESERVE filter 0.0001 400
# 往布隆过滤器添加一个key
BF.ADD filter mark
# 往布隆过滤器添加多个key
BF.MADD filter mark mark1 mark2
# 检测布隆过滤器是否包含某key
BF.EXISTS filter mark
# 检测布隆过滤器是否包含这些key
BF.MEXISTS filter mark mark1 mark2
```

redis持久化

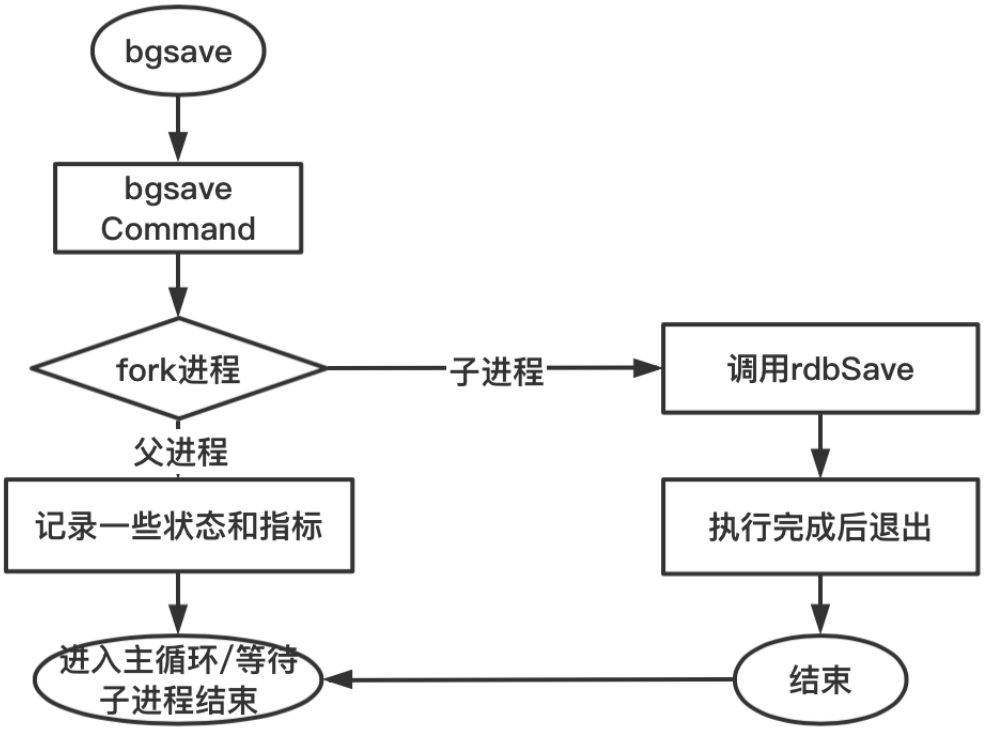
为什么需要持久化

> redis内存数据库，为了redis存储的数据重启后依然可用，需要将redis的数据落盘，重启后再加载回来。
>
> 集群的基础，主从复制，全量数据同步，rdb持久化，网络传输到从数据库，完成数据同步。

持久化的方式

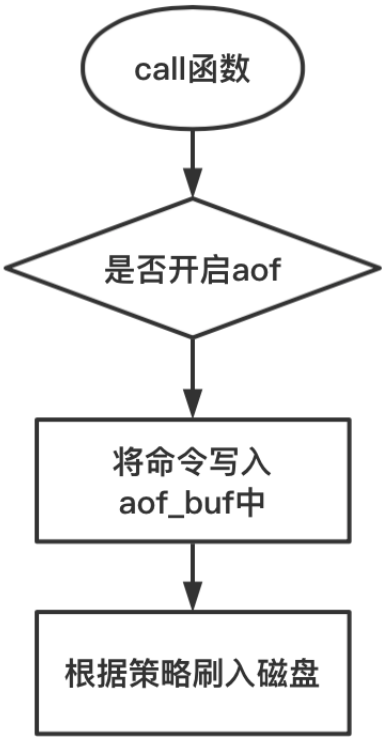
rdb

- 如何开启
    1. 脚本控制持久化策略：save ；save 900 1;如果900秒内有一次写操作，那么进行rdb持久化；
    2. 命令控制，save 阻塞 redis 进程直到 rdb 持久化完毕；bgsave 会fork出一个子进程，然后由子进程持久化；
- rdb持久化方式
    1. 数据的存储格式；
    2. rdb持久化就是根据内存中存储的数据格式进行持久化；数据量比较少；
    3. fork，内存拷贝，父进程继续对外提供服务；



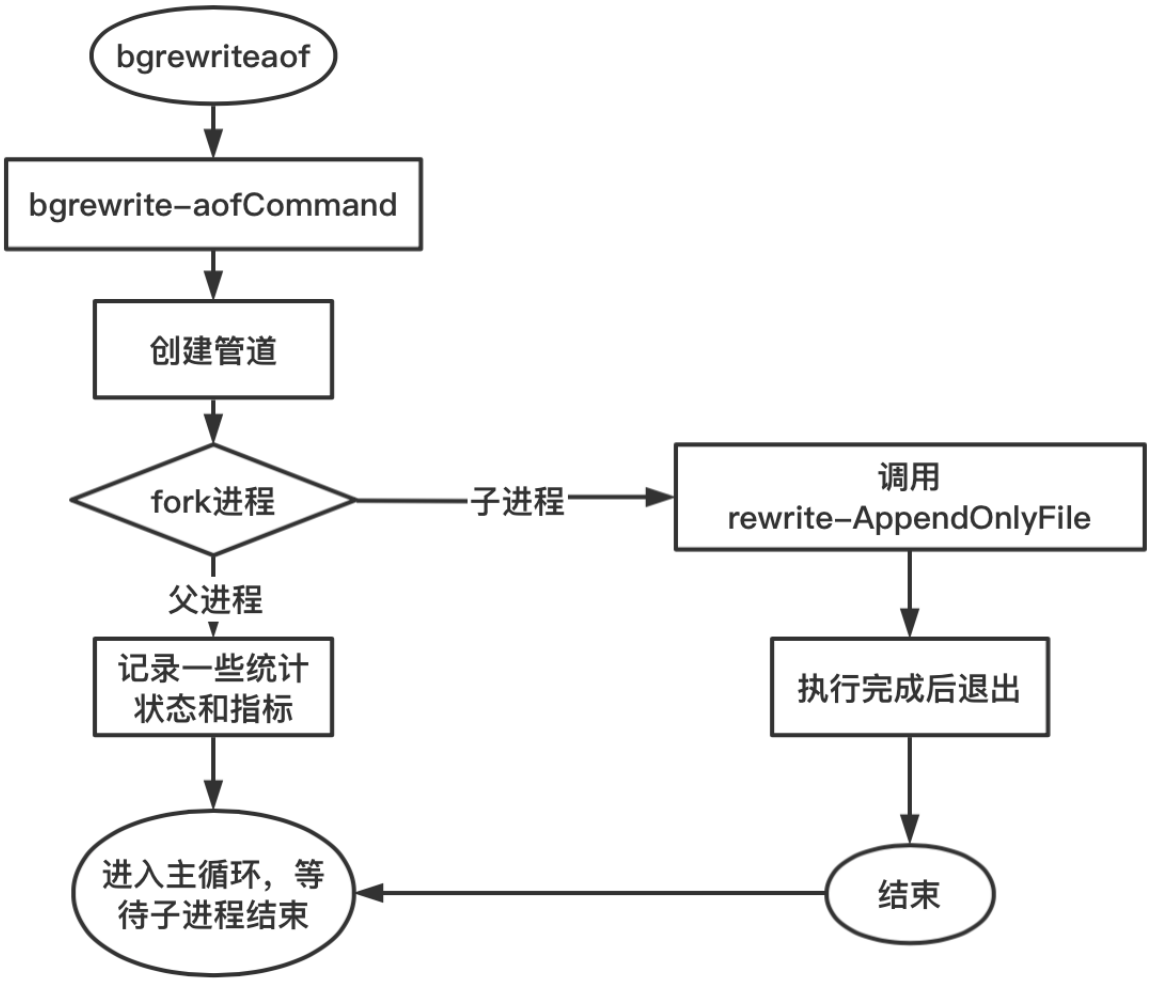
aof

- 如何开启
  1. 脚本控制持久化策略：appendonly yes；
  2. 刷磁盘策略：appendfsync everysec 每秒刷一次磁盘；appendfsync always 没写一次刷一次磁盘；appendfsync no 系统自己刷磁盘；
- aof持久化方式
  1. 存储格式为redis协议格式；
  2. 以**每次写**的协议格式进行存储；数据量比较大；
  3. 理论上最多只会丢一次写数据；假如开了appendfsync always；



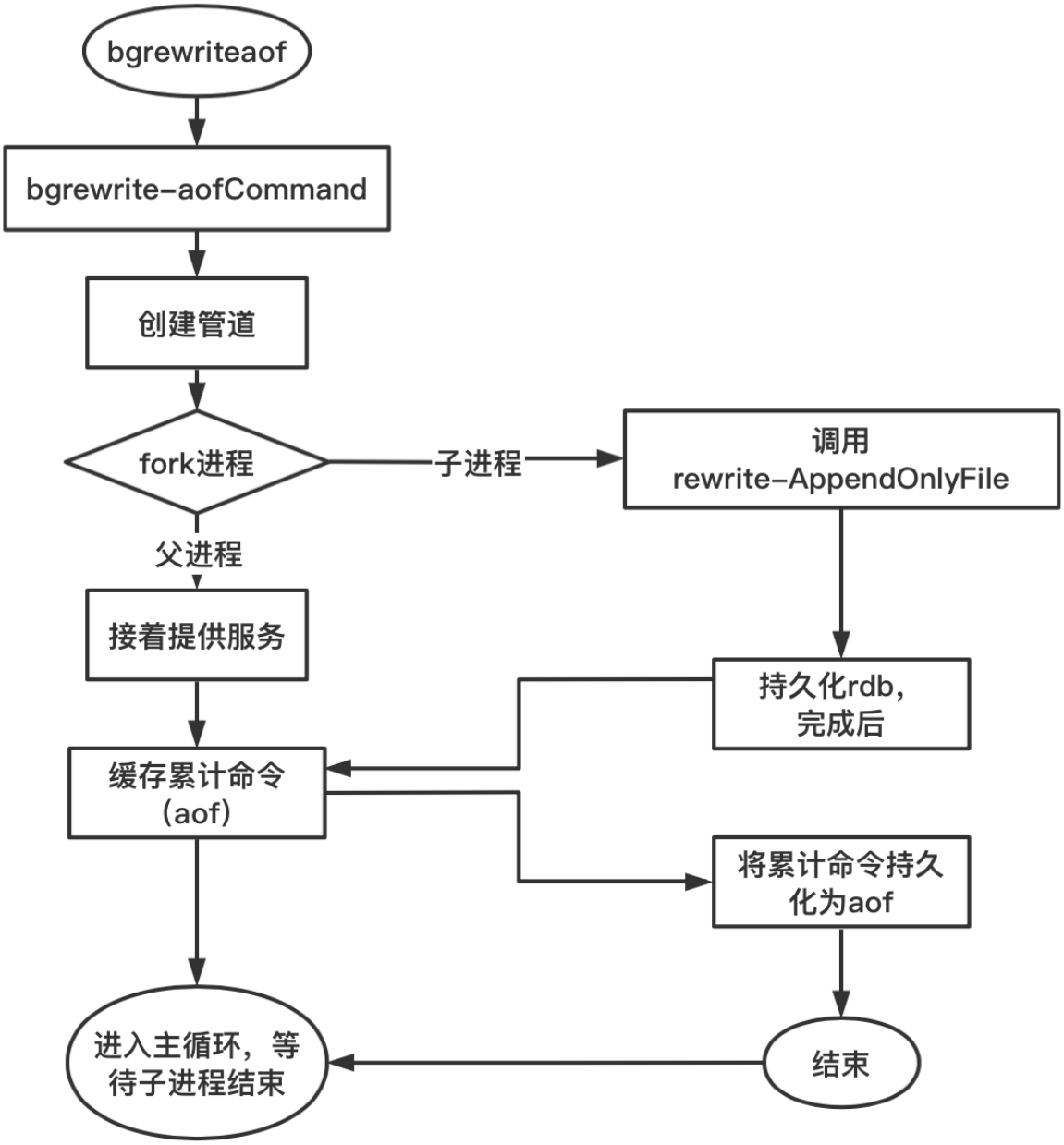
aof重写

- 如何开启
    1. 脚本控制持久化策略：auto-aof-rewrite-percentage 100；如果auto-aof-rewrite-percentage 0 则关闭aof重写；
    2. aof重写策略：auto-aof-rewrite-min-size 64mb；当aof数据大于64mb后进行aof重写；
    3. 命令控制：bgrewriteaof会派生出一个子进程，然后由子进程持久化；
- aof重写持久化方式
    1. 根据当前内存中的数据，生成aof格式（命令的协议格式）；这样的方式能较少aof的弊端，aof会产生很多中间操作步骤，而aof是以结果为导向直接生成语句；
    2. fork进程，由子进程来进行aof重写持久化；



混合持久化

- 如何开启
    1. 脚本控制：aof-use-rdb-preamble yes；
    2. 前提是需要**开启aof以及aof重写**；
    3. 进行aof重写的时候，将重写的数据以rdb的方式存储；后面缓冲区的数据以aof的方式存储；
    4. 存储在 appendonly.aof中；
- 混合持久化方式
    1. fork一个进程，根据当前内存中的数据，生成rdb格式数据；fork之后的写数据缓存起来，等子进程通知持久化结束，然后将这些数据以aof的方式存储起来；



2. 混合存储的文件格式

关于零声学院

- 构建**完整的后端开发**的知识体系，以此提升自身竞争力；
  - 框架、工具以及解决方案
  - 理论知识
  - 软实力

    - 安全能力
    - 代码能力
    - 工程素养
    - 架构能力
    - 运营能力
- 你可以自己学习，但也可考虑跟着我们的步伐来学习；

  > 零声学院第9代 linux C/C++ 后台架构开发成长体系

  https://www.0voice.com/uiwebsite/html/courses/v9.2.html
- 课程地址

  https://ke.qq.com/course/420945?tuin=97ed1c8a
- 不看广告，看疗效；

  https://www.yuque.com/lingshengxueyuan/0voice/rw8cgz
- 腾讯知识体系分享