

Redis浅析—part0(源码)


徐潇
2020.08

Redis简介

Redis: 全称REmote DIctionary Server, 是一个由Salvatore Sanfilippo写的高性能**key-value**非关系型内存数据库, 其完全开源免费, 遵守BSD协议。

Value类型: 字符串string、列表list、集合set、有序集合zset、哈希表hash

Redis为何这么快?

1. 使用语言: C语言, 可直接编译成可执行二进制代码
 2. 内存, 不需要考虑磁盘转速/扇区等等——mysql存储最小单元
 3. 数据结构简单
 4. 底层使用的数据结构符合redis设计特点: 字符串SDS、双向链表、字典、跳表等等
 5. IO多路复用, epoll
 6. 使用单线程?
- 

Redis常见用途

- 数据缓存——memcache
- pub/sub发布订阅队列（类MQ）——rabbitmq/kafka
- 分布式锁（setnx(key, value)）——ZooKeeper/RedLock
- 计数器/限速器(incr)
- 点赞数、评论数、点击数等(hash)
- 帖子的标题、摘要、作者等信息（hash）
- 关注列表、收藏列表（zset）
- top10热帖（zset）

源码读的方法

1. 自底向上：从耦合关系最小的模块开始读，然后逐渐过度到关系紧密的模块。就好像写程序的测试一样，先从单元测试开始，然后才到功能测试。
2. 自顶向下：从程序的 `main()` 函数，或者某个特别大的调用者函数为入口，以深度优先或者广度优先的方式阅读它的源码。
3. 从功能入手：通过文件名（模块名）和函数名，快速定位到一个功能的具体实现，然后追踪整个实现的运作流程，从而了解该功能的实现方式。

文件一览 (src文件夹)

adlist.c	C 文件	crc16.c	C 文件	redis-benchmark.c	C 文件	sort.c	C 文件
adlist.h	H 文件	crc64.c	C 文件	redis-check-aof.c	C 文件	sparkline.c	C 文件
ae.c	C 文件	crc64.h	H 文件	redis-check-dump.c	C 文件	sparkline.h	H 文件
ae.h	H 文件	db.c	C 文件	redis-cli.c	C 文件	syncio.c	C 文件
ae_epoll.c	C 文件	debug.c	C 文件	redis-trib.rb	RB 文件	t_hash.c	C 文件
ae_evport.c	C 文件	dict.c	C 文件	release.c	C 文件	t_list.c	C 文件
ae_kqueue.c	C 文件	dict.h	H 文件	replication.c	C 文件	t_set.c	C 文件
ae_select.c	C 文件	endianconv.c	C 文件	rio.c	C 文件	t_string.c	C 文件
anet.c	C 文件	endianconv.h	H 文件	rio.h	H 文件	t_zset.c	C 文件
anet.h	H 文件	fmacros.h	H 文件	scripting.c	C 文件	testhelp.h	H 文件
aof.c	C 文件	help.h	H 文件	sds.c	C 文件	util.c	C 文件
asciilogo.h	H 文件	hyperloglog.c	C 文件	sds.h	H 文件	util.h	H 文件
bio.c	C 文件	intset.c	C 文件	sentinel.c	C 文件	valgrind.sup	SUP 文件
bio.h	H 文件	intset.h	H 文件	setproctitle.c	C 文件	version.h	H 文件
bitops.c	C 文件	latency.c	C 文件	sha1.c	C 文件	ziplist.c	C 文件
blocked.c	C 文件	latency.h	H 文件	sha1.h	H 文件	ziplist.h	H 文件
cluster.c	C 文件	lzf.h	H 文件	slowlog.c	C 文件	zipmap.c	C 文件
cluster.h	H 文件	lzf.c.c	C 文件	slowlog.h	H 文件	zipmap.h	H 文件
config.c	C 文件	lzf_d.c	C 文件	solarisfixes.h	H 文件	zmalloc.c	C 文件
config.h	H 文件	lzfP.h	H 文件	sort.c	C 文件	zmalloc.h	H 文件
		Makefile	文件				

Redis数据结构底层实现

Redis数据结构

字符串对象：String

使用**整数值**实现的字符串对象

使用**embstr**编码的**动态字符串**实现的字符串对象

动态字符串实现的字符串对象

列表对象：List

压缩列表实现的列表对象

双端链表实现的列表对象

哈希对象：Hash

压缩列表实现的哈希对象

字典实现的哈希对象

集合对象：Set

整数集合实现的集合对象

字典实现的集合对象

有序集合对象：ZSet

压缩列表实现的有序集合对象

跳跃表和**字典**实现的有序集合对象

数据结构

SDS字符串

双向链表

字典

跳跃表

Simple dynamic string 简单动态字符串

位置: src/sds.c src/sds.h

len已使用字节
free未使用字节
buf字节数组

```
#include <stdint.h>

typedef char *sds;

struct sdshdr {
    unsigned int len;
    unsigned int free;
    char buf[];
};
```

Simple dynamic string 简单动态字符串

```
/* Create an empty (zero length) sds string. Even in this case the string
 * always has an implicit null term. */
sds sdsempty(void) {
    return sdsnewlen("", 0);
}

/* Create a new sds string starting from a null terminated C string. */
sds sdsnew(const char *init) {
    size_t initlen = (init == NULL) ? 0 : strlen(init);
    return sdsnewlen(init, initlen);
}

/* Duplicate an sds string. */
sds sdsdup(const sds s) {
    return sdsnewlen(s, sdslen(s));
}

/* Free an sds string. No operation is performed if 's' is NULL. */
void sdsfree(sds s) {
    if (s == NULL) return;
    zfree(s - sizeof(struct sdshdr));
}
```

Simple dynamic string 简单动态字符串

相对于C原生字符串

- 获取字符串长度——常数复杂度 $O(1)$
- 内存重分配次数减少(free)
- 杜绝缓冲区溢出
- 二进制安全
- 兼容部分C字符串函数

Simple dynamic string 简单动态字符串

获取字符串长度

Key value 长度限制均为512MB

确保执行STRLEN命令不会成为性能瓶颈

Simple dynamic string 简单动态字符串

内存重分配次数减少

空间预分配

若len<1MB, 则分配free=len

若len>=1MB, 则分配free=1MB

```
8  * by sdslen(), but only the free buffer space we have. */
9  sds sdsMakeRoomFor(sds s, size_t addlen) {
10     struct sdshdr *sh, *newsh;
11     size_t free = sdsavail(s);
12     size_t len, newlen;
13
14     if (free >= addlen) return s;
15     len = sdslen(s);
16     sh = (void*) (s-(sizeof(struct sdshdr)));
17     newlen = (len+addlen);
18     if (newlen < SDS_MAX_PREALLOC)
19         newlen *= 2;
20     else
21         newlen += SDS_MAX_PREALLOC;
22     newsh = zrealloc(sh, sizeof(struct sdshdr)+newlen+1);
23     if (newsh == NULL) return NULL;
24
25     newsh->free = newlen - len;
26     return newsh->buf;
27 }
```

Simple dynamic string 简单动态字符串

内存重分配次数减少——惰性释放

```

    *
    * Output will be just "Hello World".
    */
- sds sdstrim(sds s, const char *cset) {
    struct sdshdr *sh = (void*) (s-(sizeof(struct sdshdr)));
    char *start, *end, *sp, *ep;
    size_t len;

    sp = start = s;
    ep = end = s+sdslen(s)-1;
    while(sp <= end && strchr(cset, *sp)) sp++;
    while(ep > start && strchr(cset, *ep)) ep--;
    len = (sp > ep) ? 0 : ((ep-sp)+1);
    if (sh->buf != sp) memmove(sh->buf, sp, len);
    sh->buf[len] = '\0';
    sh->free = sh->free+(sh->len-len);
    sh->len = len;
    return s;
}
```

Simple dynamic string 简单动态字符串

杜绝缓冲区溢出

C字符串不记录自身长度

Malloc & free

溢出 & 泄露

二进制安全

C字符串靠\0结束符，无法存储图片、视频等二进制文件

SDS通过len来记录

Simple dynamic string 简单动态字符串

兼容部分C字符串函数

sds字符串总会以\0结束，分配空间时多分配一个字节

复用C库函数，如strcat(c_string, sds_buf)进行追加

双向链表

链表定义

```
/* Node, list, and iterator are the only c  
- typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
- } listNode;  
  
- typedef struct listIter {  
    listNode *next;  
    int direction;  
- } listIter;  
  
- typedef struct list {  
    listNode *head;  
    listNode *tail;  
    void *(*dup)(void *ptr);  
    void (*free)(void *ptr);  
    int (*match)(void *ptr, void *key);  
    unsigned long len;  
- } list;
```

- 前置节点与尾节点都指向NULL;
- 双端，可前后两个方向进行迭代;
- 长度计数，无需计算节点个数;
- 多态，使用void*指针保存节点

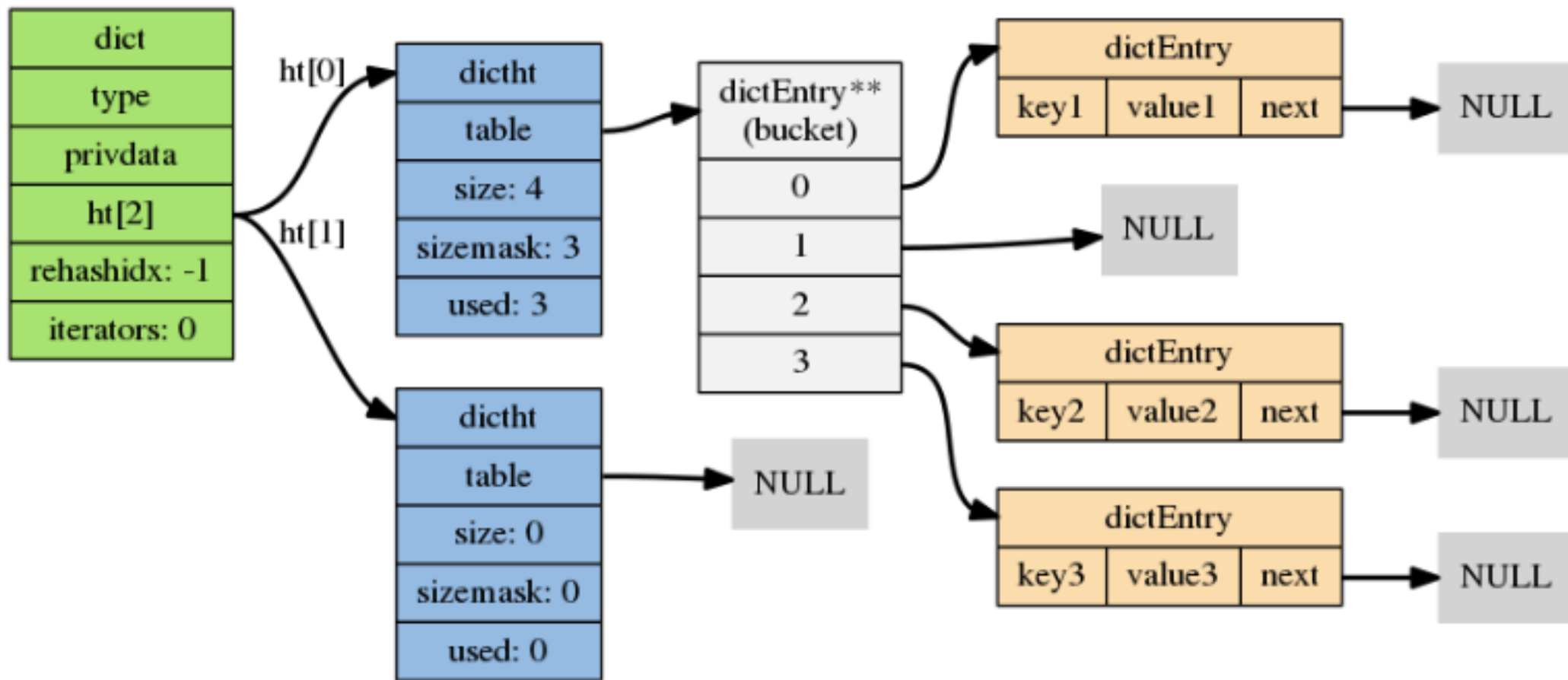
dup: 复制链接节点的值

free: 释放链表节点的值

match: 比对值是否相等

字典

字典结构



字典

字典结构

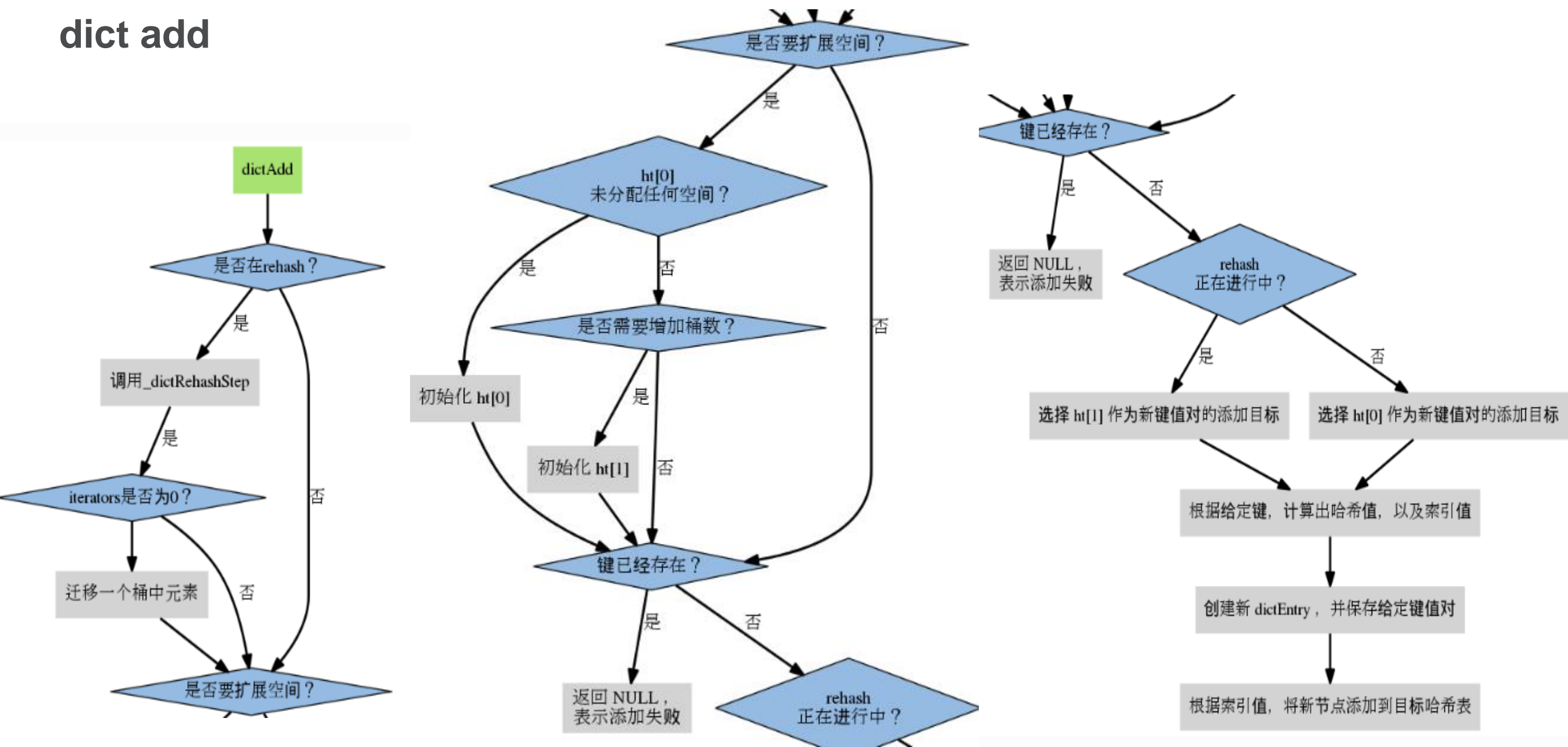
```
typedef struct dict {
    dictType *type; /* 类型特定函数（多态） */
    void *privdata; /* 私有数据（多态） */
    dictht ht[2]; /* 保存的两个哈希表，ht[0]是真正使用的，ht[1]会在rehash时使用 */
    long rehashidx; /* rehash进度，如果不等于-1，说明还在进行rehash */
    int iterators; /* 运行中的迭代器个数 */
} dict;

typedef struct dictht { /* 哈希表 */
    dictEntry **table; /* 节点数组 */
    unsigned long size; /* 大小 */
    unsigned long sizemask; /* 哈希表大小掩码，用于计算哈希表的索引值，大小总是dictht.size - 1 */
    unsigned long used; /* 哈希表已经使用的节点数量 */
} dictht;

typedef struct dictEntry { /* 哈希表节点 */
    void *key; /* 键名 */
    union {
        void *val;      uint64_t u64;      int64_t s64;      double d;
    } v; /* 值 */
    struct dictEntry *next; /* 指向下一个节点，（链地址法，解决key冲突） */
} dictEntry;
```

字典

dict add



rehash—重新散列

目的：保持哈希表的键值对数量保持一个合理范围内

扩展/收缩触发条件：

- 服务器目前没有在执行 BGSAVE 命令或者 BGREWRITEAOF 命令，并且哈希表的负载因子大于等于 1；
- 服务器目前正在执行 BGSAVE 命令或者 BGREWRITEAOF 命令，并且哈希表的负载因子大于等于 5；
- 当哈希表的负载因子小于 0.1 时，程序自动开始对哈希表执行收缩操作

$$\text{load_factor} = \text{ht}[0].\text{used} / \text{ht}[0].\text{size}$$

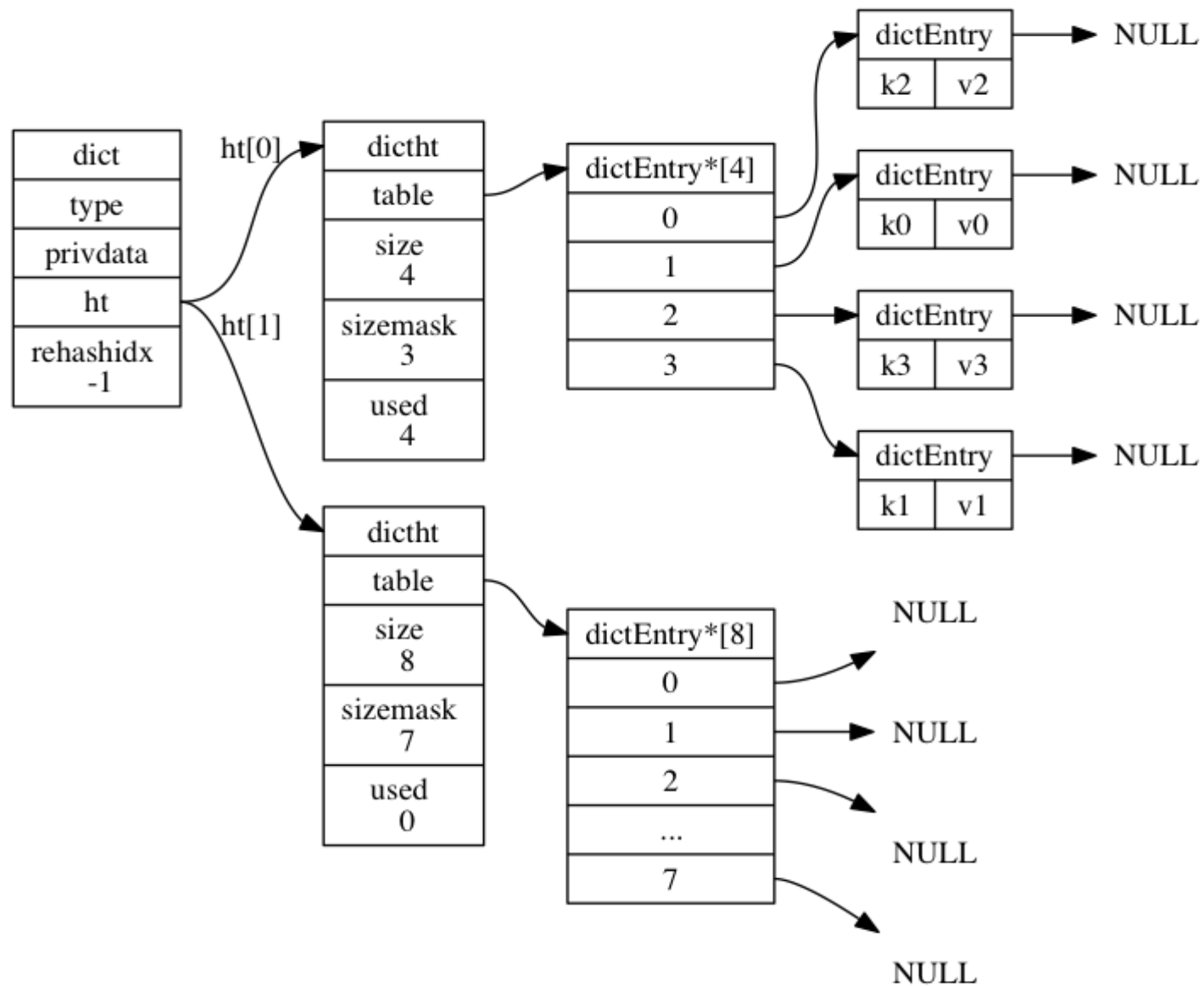
1. 为字典的 ht[1] 哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及 ht[0] 当前包含的键值对数量（也即是 ht[0].used 属性的值）：
 - 如果执行的是扩展操作，那么 ht[1] 的大小为第一个大于等于 ht[0].used * 2 的 2^n （2 的 n 次方幂）；
 - 如果执行的是收缩操作，那么 ht[1] 的大小为第一个大于等于 ht[0].used 的 2^n 。
2. 将保存在 ht[0] 中的所有键值对 **rehash** 到 ht[1] 上面：rehash 指的是重新计算键的哈希值和索引值，然后将键值对放置到 ht[1] 哈希表的指定位置上。
3. 当 ht[0] 包含的所有键值对都迁移到了 ht[1] 之后（ht[0] 变为空表），释放 ht[0]，将 ht[1] 设置为 ht[0]，并在 ht[1] 新创建一个空白哈希表，为下一次 rehash 做准备。

渐进式rehash

1. 为 `ht[1]` 分配空间，让字典同时持有 `ht[0]` 和 `ht[1]` 两个哈希表。
2. 在字典中维持一个索引计数器变量 `rehashidx`，并将它的值设置为 0，表示 rehash 工作正式开始。
3. 在 rehash 进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将 `ht[0]` 哈希表在 `rehashidx` 索引上的所有键值对 rehash 到 `ht[1]`，当 rehash 工作完成之后，程序将 `rehashidx` 属性的值增一。
4. 随着字典操作的不断执行，最终在某个时间点上，`ht[0]` 的所有键值对都会被 rehash 至 `ht[1]`，这时程序将 `rehashidx` 属性的值设为 -1，表示 rehash 操作已完成。

字典

渐进式rehash

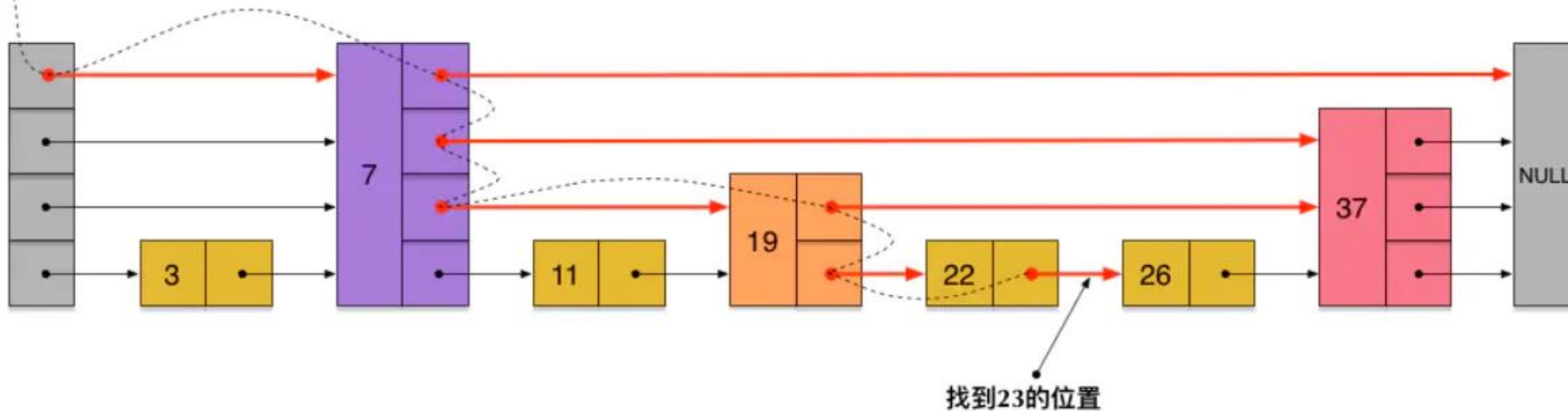


跳跃表

跳跃表是什么？

跳跃表是一种有序的数据结构，它通过在每个节点中维持多个指向其他的几点指针，从而达到快速访问队尾目的。

从这里开始查找23



跳跃表

skiplist与平衡树、哈希表的比较

- skiplist和各种平衡树（如AVL、红黑树等）的元素是**有序**的，而哈希表不是。在哈希表上只能做单个key的查找，不适宜做范围查找。所谓范围查找，指的是查找那些大小在指定的两个值之间的所有节点。
- 在做**范围查找**的时候，平衡树比skiplist操作要复杂。在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在skiplist上进行范围查找就非常简单，只需要在找到小值之后，对第1层链表进行若干步的遍历就可以实现。
- 平衡树的**插入和删除**操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。
- 从**内存占用**上来说，skiplist比平衡树更灵活一些。一般来说，平衡树每个节点包含2个指针（分别指向左右子树），而skiplist每个节点包含的指针数目平均为 $1/(1-p)$ ，具体取决于参数p的大小。如果像Redis里的实现一样，取 $p=1/4$ ，那么平均每个节点包含1.33个指针，比平衡树更有优势。
- 查找单个key，skiplist和平衡树的时间复杂度都为 $O(\log n)$ ；而哈希表在保持较低的哈希值冲突概率的前提下，查找时间复杂度接近 $O(1)$ ，性能更高一些。所以我们平常使用的各种Map或dictionary结构，大都是基于哈希表实现的。
- 从算法**实现难度**上来比较，skiplist比平衡树要简单得多。

redis应用部分

常见后悔瞬间

- 错用命令: keys * flushdb/flushall
- 缓存穿透、缓存雪崩和缓存击穿
- 无效/低命中率数据过多导致内存打满
- 等等...

参考信息

[Redis官网](#)

[如何阅读Redis源码?](#)

[Redis中文文档](#)

[redis源码注释](#)

后续计划

redis架构:

集群: 主从、哨兵、cluster模式

持久化: RDB、AOF

新特性: 多线程

管道技术pipeline

应用分析:

分布式锁

内存管理、缓存过期LRU&LFU

双写一致性问题

排行榜、计数器等