Redis 内存存储结构分析

五竹,20110418

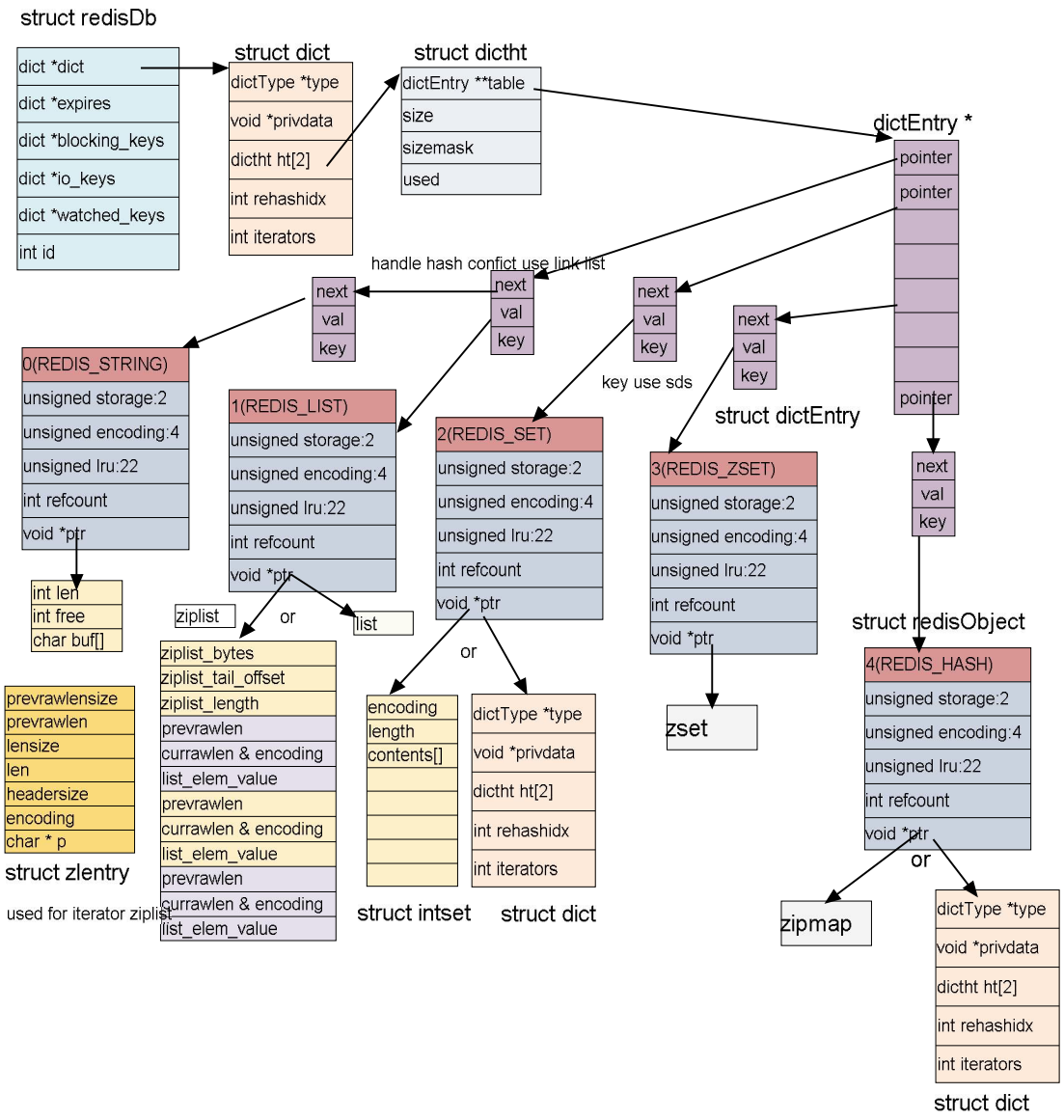
Redis: A persistent key-value database with built-in net interface written in ANSI-C for Posix systems

1 Redis 内存存储结构

本文是基于 Redis-v2.2.4 版本进行分析.

1.1 Redis 内存存储总体结构

Redis 是支持多 key-value 数据库(表)的,并用 RedisDb 来表示一个 key-value 数据库(表). redisServer 中有一个 redisDb *db; 成员变量, RedisServer 在初始化时,会根据配置文件的 db 数量来创建一个 redisDb 数组. 客户端在连接后,通过 SELECT 指令来选择一个 reidsDb,如果不指定,则缺省是 redisDb 数组的第 1 个(即下标是 0) redisDb. 一个客户端在选择 redisDb 后,其后续操作都是在此 redisDb 上进行的. 下面会详细介绍一下 redisDb 的内存结构.



redis 的内存存储结构示意图

redisDb 的定义:

```
typedef struct redisDb {
        dict *dict;                     /* The keyspace for this DB */
        dict *expires;                  /* Timeout of keys with a timeout set */
        dict *blocking_keys;        /* Keys with clients waiting for data (BLPOP) */
        dict *io_keys;                  /* Keys with clients waiting for VM I/O */
        dict *watched_keys;             /* WATCHED keys for MULTI/EXEC CAS */
        int id;
} redisDb;
```

struct redisDb 中 ,dict 成员是与实际存储数据相关的. dict 的定义如下:

```
typedef struct dictEntry
{
	void *key;
	void *val;
	struct dictEntry *next;
} dictEntry;

typedef struct dictType
{
	unsigned int (*hashFunction)(const void *key);
	void *(*keyDup)(void *privdata, const void *key);
	void *(*valDup)(void *privdata, const void *obj);
	int (*keyCompare)(void *privdata, const void *key1, const void *key2);
	void (*keyDestructor)(void *privdata, void *key);
	void (*valDestructor)(void *privdata, void *obj);
} dictType;

/* 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];
	int rehashidx; /* rehashing not in progress if rehashidx == -1 */
	int iterators; /* number of iterators currently running */
} dict;
```

dict 是主要是由 struct dictht 的 哈唏表构成的, 之所以定义成长度为 2 的 (dictht ht[2]) 哈唏表数组,是因为 redis 采用渐进的 rehash,即当需要 rehash 时, 每次像 hset,hget 等操作前,先执行 N 步 rehash. 这样就把原来一次性的 rehash 过程拆散到进行, 防止一次性 rehash 期间 redis 服务能力大幅下降. 这种渐进的 rehash 需要一个额外的 struct dictht 结构来保存.

struct dictht 主要是由一个 struct dictEntry 指针数组组成的, hash 表的冲突是通过链表法来解决的.

struct dictEntry 中的 key 指针指向用 sds 类型表示的 key 字符串, val 指针指向一个 struct redisObject 结构体, 其定义如下:

```
typedef struct redisObject
{
    unsigned type:4;
    unsigned storage:2;     /* REDIS_VM_MEMORY or REDIS_VM_SWAPPING */
    unsigned encoding:4;
    unsigned lru:22;              /* lru time (relative to server.lruclock) */
    int refcount;
    void *ptr;
    /* VM fields are only allocated if VM is active, otherwise the
     * object allocation function will just allocate
     * sizeof(redisObjct) minus sizeof(redisObjectVM), so using
     * Redis without VM active will not have any overhead. */
} robj;
```

- type 占 4 bit,用来表示 key-value 中 value 值的类型,目前 redis 支持: string, list, set,zset,hash 5 种类型的值.

```
/* Object types */
#define REDIS_STRING 0
#define REDIS_LIST 1
#define REDIS_SET 2
#define REDIS_ZSET 3
#define REDIS_HASH 4
#define REDIS_VMPOINTER 8
```

- storage 占 2 bit ,表示 此值是在 内存中,还是 swap 在硬盘上.
- encoding 占 4 bit ,表示值的编码类型,目前有 8 种类型:

```
/* Objects encoding. Some kind of objects like Strings and Hashes can be
 * internally represented in multiple ways. The 'encoding' field of the object
 * is set to one of this fields for this object. */
#define REDIS_ENCODING_RAW 0         /* Raw representation */
#define REDIS_ENCODING_INT 1         /* Encoded as integer */
#define REDIS_ENCODING_HT 2          /* Encoded as hash table */
#define REDIS_ENCODING_ZIPMAP 3    /* Encoded as zipmap */
#define REDIS_ENCODING_LINKEDLIST 4 /* Encoded as regular linked
list */
#define REDIS_ENCODING_ZIPLIST 5 /* Encoded as ziplist */
#define REDIS_ENCODING_INTSET 6    /* Encoded as intset */
#define REDIS_ENCODING_SKIPLIST 7    /* Encoded as skiplist */
```

  - ✧ 如 type 是 REDIS_STRING 类型的,则其值如果是数字,就可以编码成 REDIS_ENCODING_INT,以节约内存.
  - ✧ 如 type 是 REDIS_HASH 类型的,如果其 entry 小于配置值: hash-max-zipmap-entries 或 value 字符串的长度小于

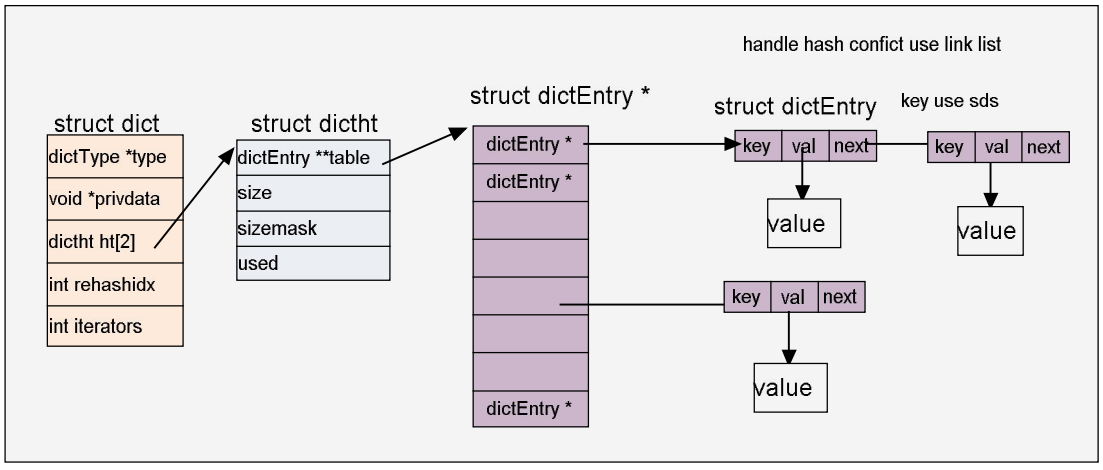
hash-max-zipmap-value, 则可以编码成 REDIS_ENCODING_ZIPMAP 类型存储,以节约内存. 否则采用 Dict 来存储.

- ✧ 如 type 是 REDIS_LIST 类型的,如果其 entry 小于配置值: list-max-ziplist-entries 或 value 字符串的长度小于 list-max-ziplist-value, 则可以编码成 REDIS_ENCODING_ZIPLIST 类型存储,以节约内存; 否则采用 REDIS_ENCODING_LINKEDLIST 来存储.
- ✧ 如 type 是 REDIS_SET 类型的,如果其值可以表示成数字类型且 entry 小于配置值 set-max-intset-entries, 则可以编码成 REDIS_ENCODING_INTSET 类型存储,以节约内存; 否则采用 Dict 类型来存储.

- lru: 是时间戳
- refcount: 引用次数
- void * ptr : 指向实际存储的 value 值内存块,其类型可以是 string, set, zset,list,hash ,编码方式可以是上述 encoding 表示的一种.

至于一个 key 的 value 采用哪种类型来保存,完全是由客户端的指令来决定的,如 hset ,则值是采用 REDIS_HASH 类型表示的,至于那种编码(encoding),则由 redis 根据配置自动决定.

1.2 Dict 结构



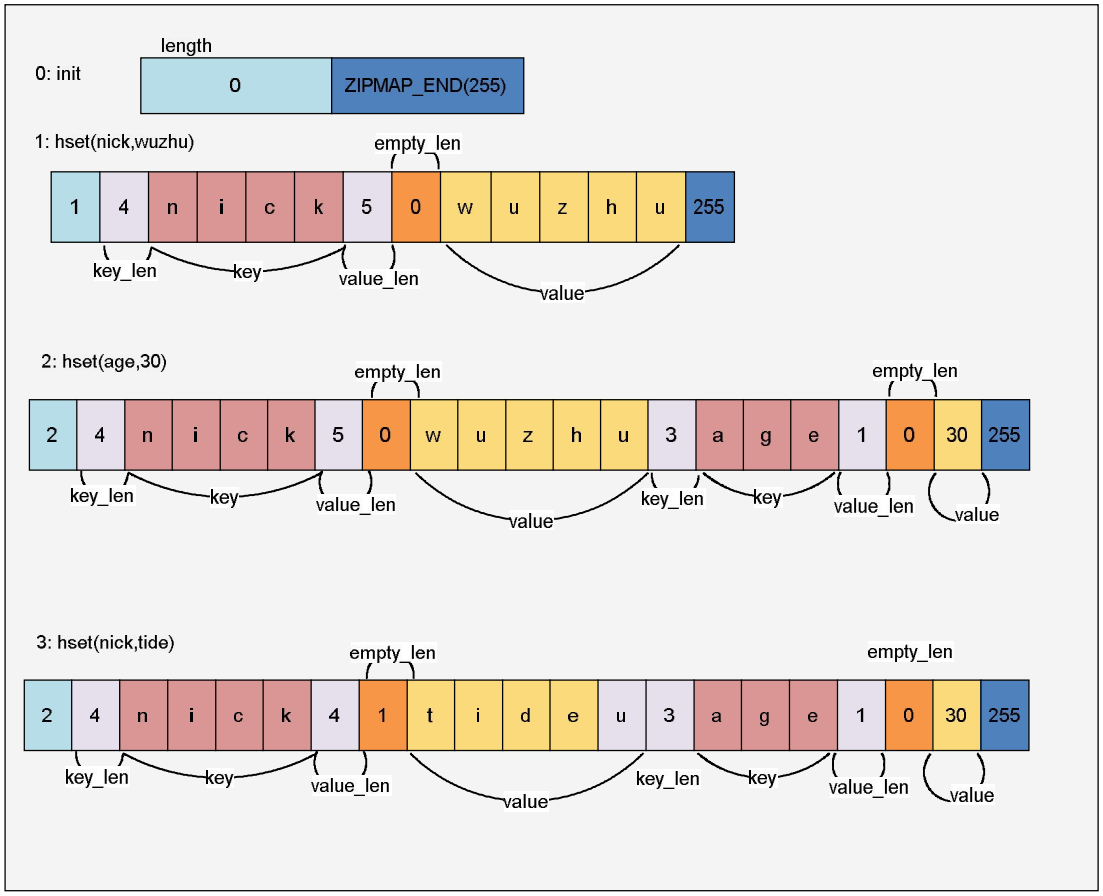
Dict 结构在< 1.1Redis 内存存储结构> 已经描述过了,这里不再累述.

1.3 zipmap 结构

如果 redisObject 的 type 成员值是 REDIS_HASH 类型的,则当该 hash 的

entry 小于配置值: hash-max-zipmap-entries 或者 value 字符串的长度小于 hash-max-zipmap-value, 则可以编码成 REDIS_ENCODING_ZIPMAP 类型存储, 以节约内存. 否则采用 Dict 来存储.

zipmap 其实质是用一个字符串数组来依次保存 key 和 value,查询时是依次遍列每个 key-value 对,直到查到为止. 其结构示意图如下:



为了节约内存,这里使用了一些小技巧来保存 key 和 value 的长度. 如果 key 或 value 的长度小于 ZIPMAP_BIGLEN(254),则用一个字节来表示,如果大于 ZIPMAP_BIGLEN(254), 则用 5 个字节保存, 第一个字节为保存 ZIPMAP_BIGLEN(254),后面 4 个字节保存 key 或 value 的长度.

1) 初始化时只有 2 个字节,第 1 个字节表示 zipmap 保存的 key-value 对的个数(如果 key-value 对的个数超过 254,则一直用 254 来表示, zipmap 中实际保存的 key-value 对个数可以通过 zipmapLen() 函数计算得到).
2) hset(nick,wuzhu) 后,
   - ✧ 第 1 个字节保存 key-value 对(即 zipmap 的 entry 数量)的数量 1
   - ✧ 第 2 个字节保存 key_len 值 4
   - ✧ 第 3~6 保存 key “nick”
   - ✧ 第 7 字节保存 value_len 值 5
   - ✧ 第 8 字节保存空闭的字节数 0 (当该 key 的值被重置时,其新值的长度与旧值的长度不一定相等,如果新值长度比旧值的长度大,则 realloc

扩大内存; 如果新值长度比旧值的长度小,且相差大于 4 bytes ,则 realloc 缩小内存,如果相差小于 4,则将值往前移,并用 empty_len 保存空闲的 byte 数)

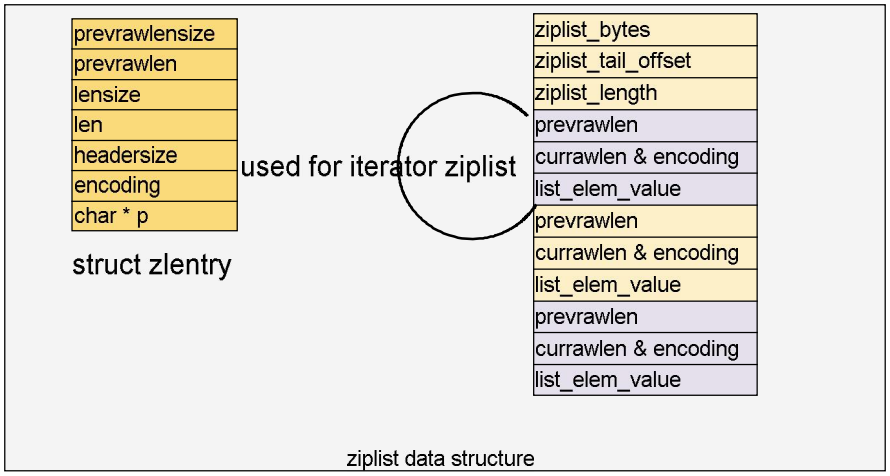
- ✧ 第 9~13 字节保存 value 值 “wuzhu”

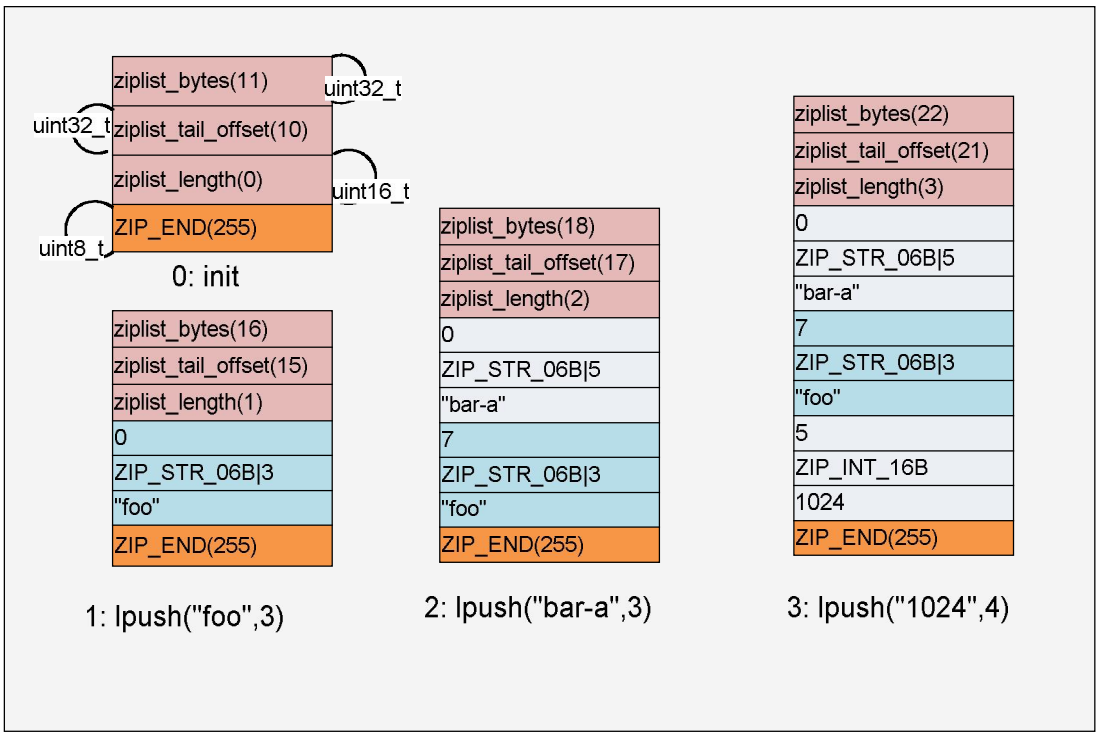
3) hset(age,30)
   插入 key-value 对 (“age”,30)
4) hset(nick,tide)
   插入 key-value 对 (“nick”,”tide”), 后可以看到 empty_len 为 1 ,

1.4 ziplist 结构

如果 redisObject 的 type 成员值是 REDIS_LIST 类型的,则当该 list 的 elem 数小于配置值: hash-max-ziplist-entries 或者 elem_value 字符串的长度小于 hash-max-ziplist-value, 则可以编码成 REDIS_ENCODING_ZIPLIST 类型存储, 以节约内存. 否则采用 Dict 来存储.

ziplist 其实质是用一个字符串数组形式的双向链表. 其结构示意图如下:





1) ziplist header 由 3 个字段组成:
   - ziplist_bytes: 用一个 uint32_t 来保存, 构成 ziplist 的字符串数组的总长度,包括 ziplist header,
   - ziplist_tail_offset: 用一个 uint32_t 来保存,记录 ziplist 的尾部偏移位置.
   - ziplist_length: 用一个 uint16_t 来保存,记录 ziplist 中 elem 的个数
2) ziplist node 也由 3 部分组成:
   - prevrawlen: 保存上一个 ziplist node 的占用的字节数,包括: 保存 prevarwlen,currawlen 的字节数和 elem value 的字节数.
   - currawlen&encoding: 当前 elem value 的 raw 形式存款所需的字节数及在 ziplist 中保存时的编码方式(例如,值可以转换成整数,如示意图中的"1024", raw_len 是 4 字节,但在 ziplist 保存时转换成 uint16_t 来保存,占 2 个字节).
   - (编码后的)value

可以通过 prevrawlen 和 currawlen&encoding 来遍列 ziplist.

ziplist 还能到一些小技巧来节约内存.

- ✧ len 的存储: 如果 len 小于 ZIP_BIGLEN(254),则用一个字节来保存; 否则需要 5 个字节来保存,第 1 个字节存 ZIP_BIGLEN,作为标识符.
- ✧ value 的存储: 如果 value 是数字类型的,则根据其值的范围转换成 ZIP_INT_16B, ZIP_INT_32B 或 ZIP_INT_64B 来保存,否则用 raw 形式保存.

1.5 adlist 结构

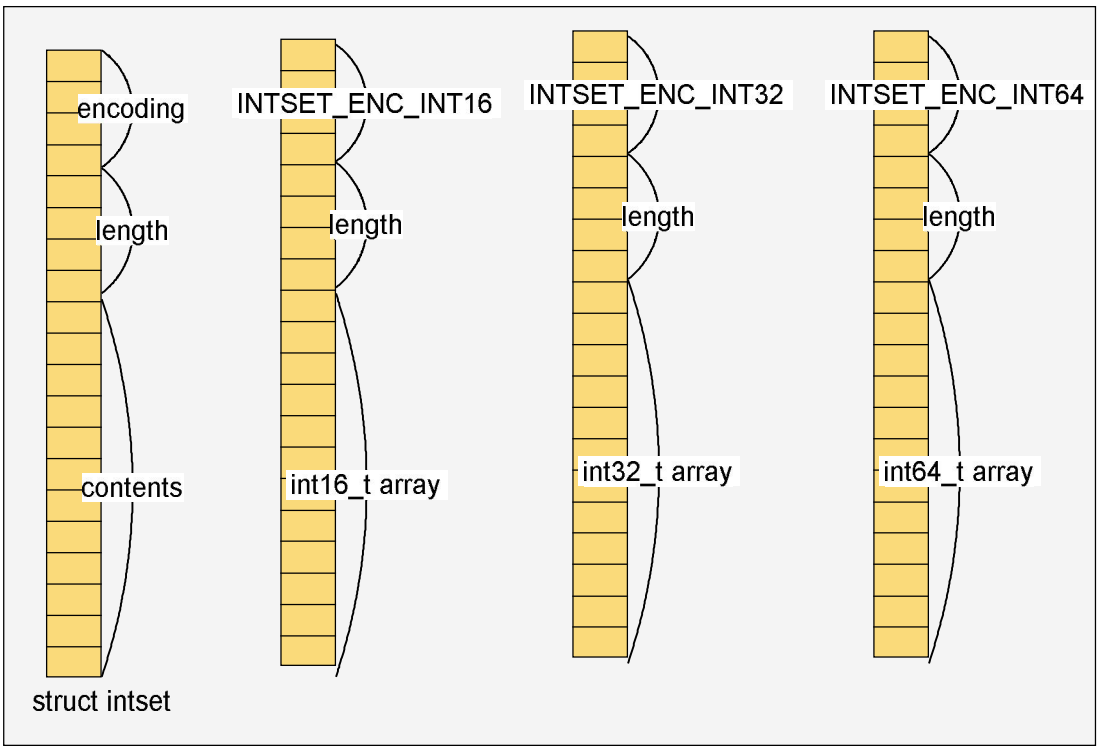
```
typedef struct listNode
{
    struct listNode *prev;
    struct listNode *next;
    void *value;
} listNode;

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 int len;
} list;
```

常见的双向链表,不作分析.

1.6 intset 结构



intset 是用一个有序的整数数组来实现集合(set). struct intset 的定义如下:

```
typedef struct intset
{
    uint32_t encoding;
    uint32_t length;
    int8_t contents[];
} intset;
```

- encoding: 来标识数组是 int16_t 类型, int32_t 类型还是 int64_t 类型的数组. 至于怎么先择是那种类型的数组,是根据其保存的值的取值范围来决定的, 初始化时是 int16_t, 根据 set 中的最大值在 [INT16_MIN, INT16_MAX] , [INT32_MIN, INT32_MAX], [INT64_MIN, INT64_MAX]的那个取值范围来动态确定整个数组的类型. 例如 set 一开始是 int16_t 类型,当一个取值范围在 [INT32_MIN, INT32_MAX]的值加入到 set 时,则将保存 set 的数组升级成 int32_t 的数组.
- length: 表示 set 中值的个数
- contents: 指向整数数组的指针

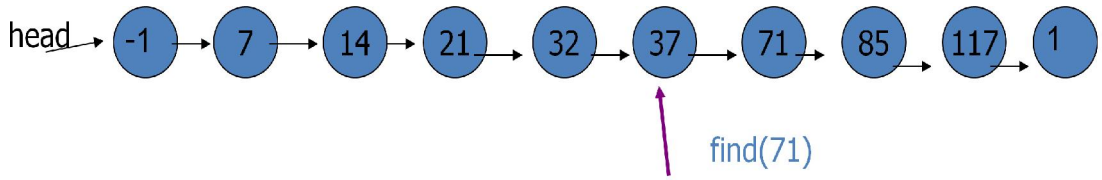
1.7 zset 结构

首先，介绍一下 skip list 的概念，然后再分析 zset 的实现.

1.7.1 Skip List 介绍

1.7.1.1 有序链表

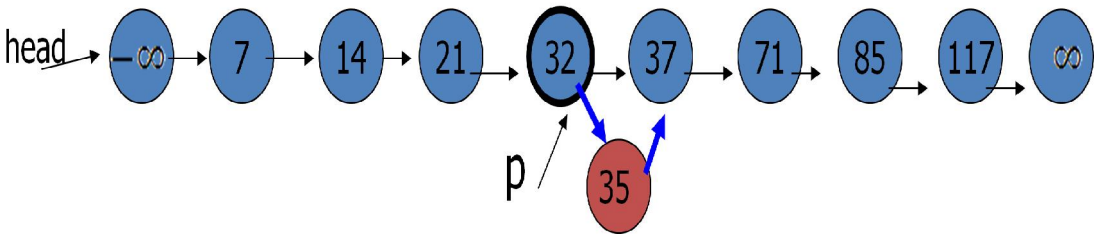
1) Searching a key in a Sorted linked list




```
//
c
w
r
```

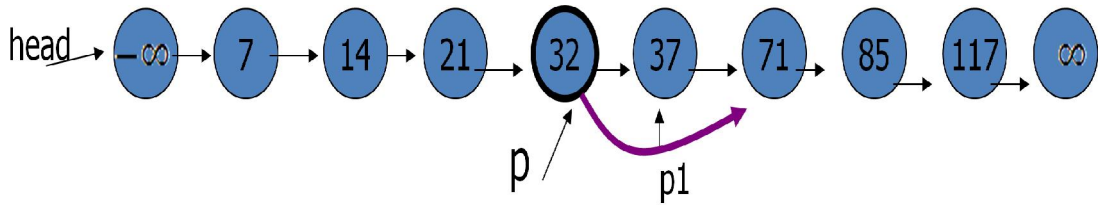
N
s

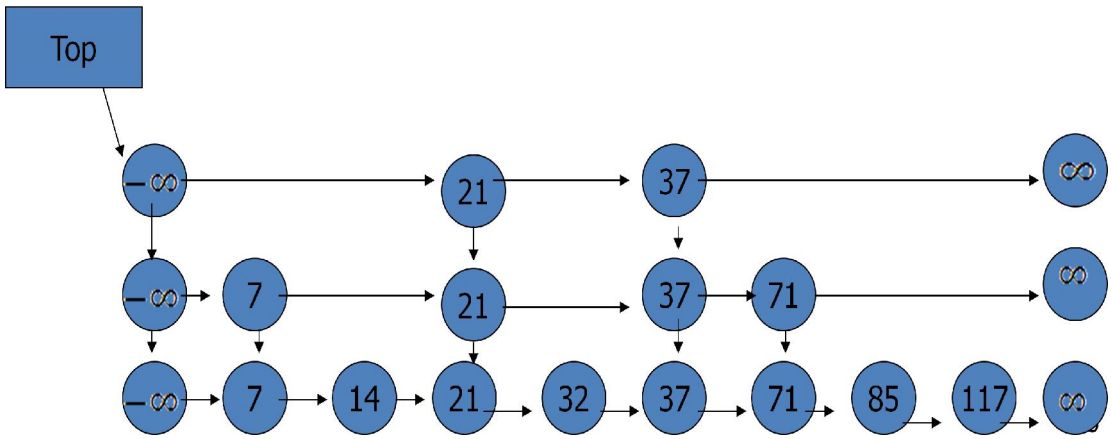
2



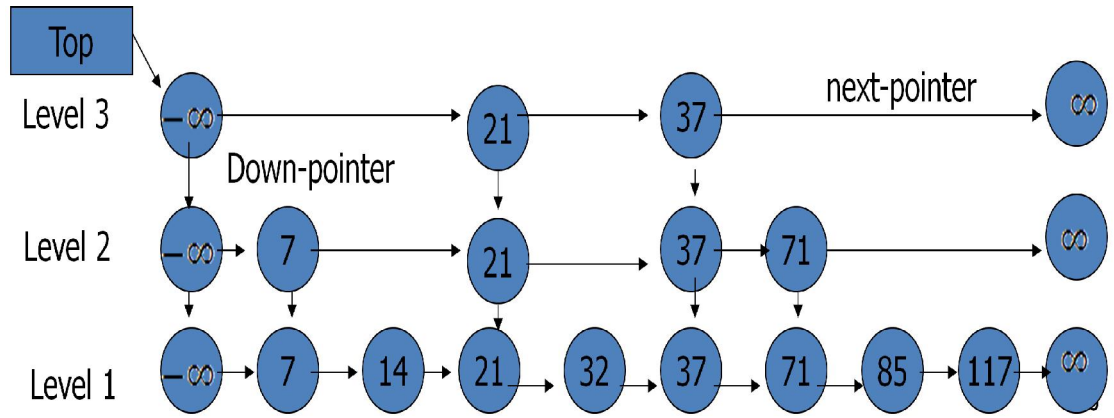

```
//To insert 35 -
p=find(35);
CELL *p1 = (CELL *) malloc(sizeof(CELL));
p1->key=35;
p1->next = p->next ;
p->next    = p1 ;
```

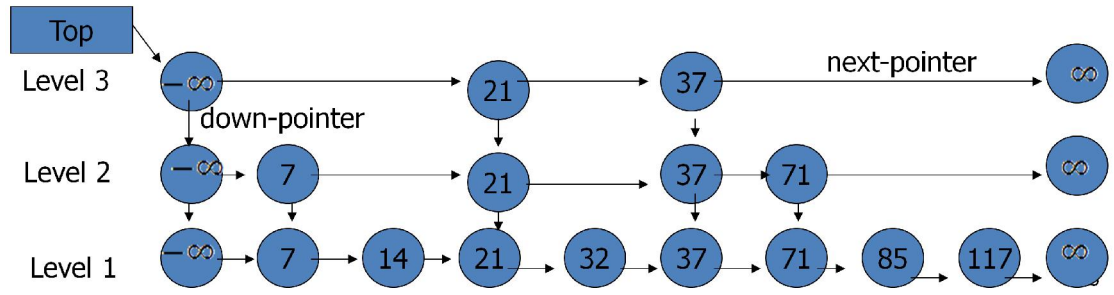
3) deleteing a key from a sorted list





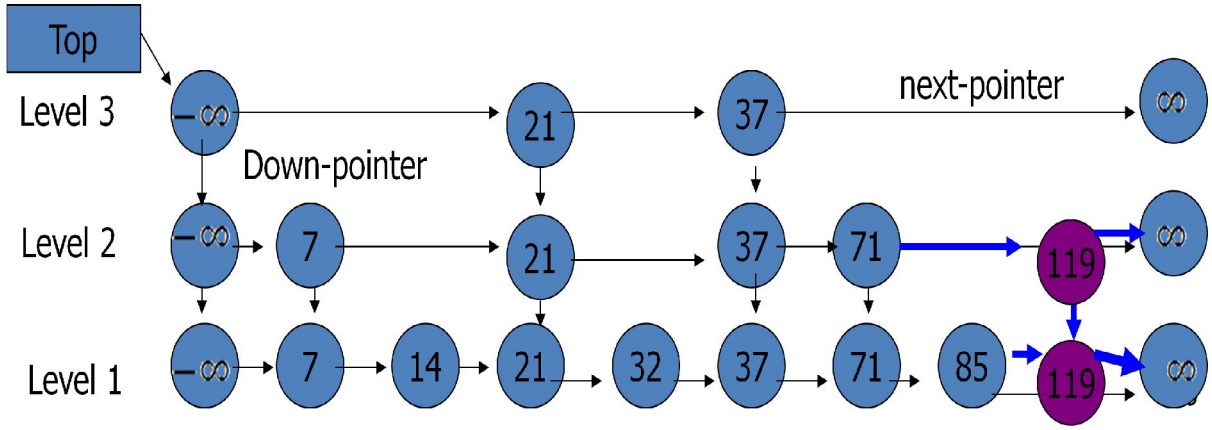
- An element in level *i* points (via down pointer) to the element with the same key in the level below.
- In each level the keys $-\infty$ and ∞ appear. (In our implementation, INT_MIN and INT_MAX
- Top points to the smallest element in the highest level.





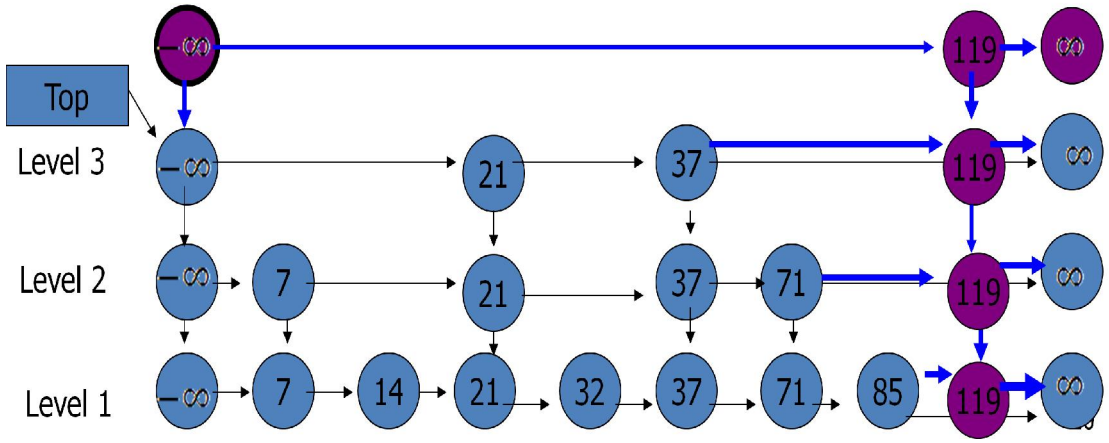
3) Inserting new element *X*

- ✧ Determine ***k*** the number of levels in which *x* participates (explained later)
- ✧ Do find(x), and insert x to the appropriate places in the lowest ***k*** levels. (after the elements at which the search path turns down or terminates)
- ✧ Example - inserting 119. ***k***=2



✧

✧

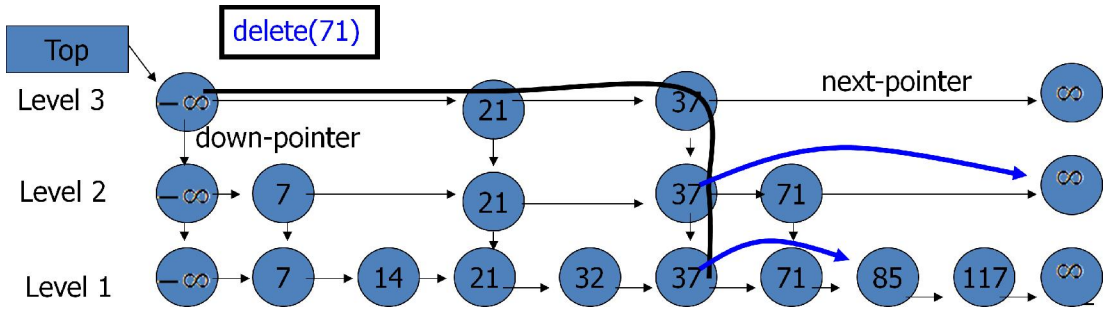


■ Determining *k*

- ✧ k - the number of levels at which an element x participate.
- ✧ Use a random function OurRnd() --- returns 1 or 0 (True/False) with equal probability.
- ✧ k=1 ;
- ✧ While(OurRnd()) k++ ;

■ Deleteing a key *x*

- ✧ Find x in all the levels it participates, and delete it using the standard 'delete from a linked list' method.
- ✧ If one or more of the upper levels are empty, remove them (and update top)



- Facts about SkipList
  - The expected number of levels is O(log *n*)
  - (here *n* is the numer of elements)
  - The expected time for insert/delete/find is O(log *n*)
  - The expected size (number of cells) is O(*n*)

1.7.2 redis SkipList 实现

```
/* ZSETs use a specialized version of Skiplists */
typedef struct zskiplistNode
{
        robj *obj;
        double score;
        struct zskiplistNode *backward;
        struct zskiplistLevel
        {
                struct zskiplistNode *forward;
                unsigned int span;
        } level[];
} zskiplistNode;

typedef struct zskiplist
{
        struct zskiplistNode *header, *tail;
        unsigned long length;
        int level;
} zskiplist;

typedef struct zset
{
        dict *dict;
        zskiplist *zsl;
} zset;
```

zset 的实现用到了 2 个数据结构: hash_table 和 skip list (跳跃表),其中 hash table 是使用 redis 的 dict 来实现的,主要是为了保证查询效率为 O(1),而 skip list (跳跃表) 是用来保证元素有序并能够保证 INSERT 和 REMOVE 操作是 O(logn)的复杂度。

1) zset 初始化状态

createZsetObject 函数来创建并初始化一个 zset

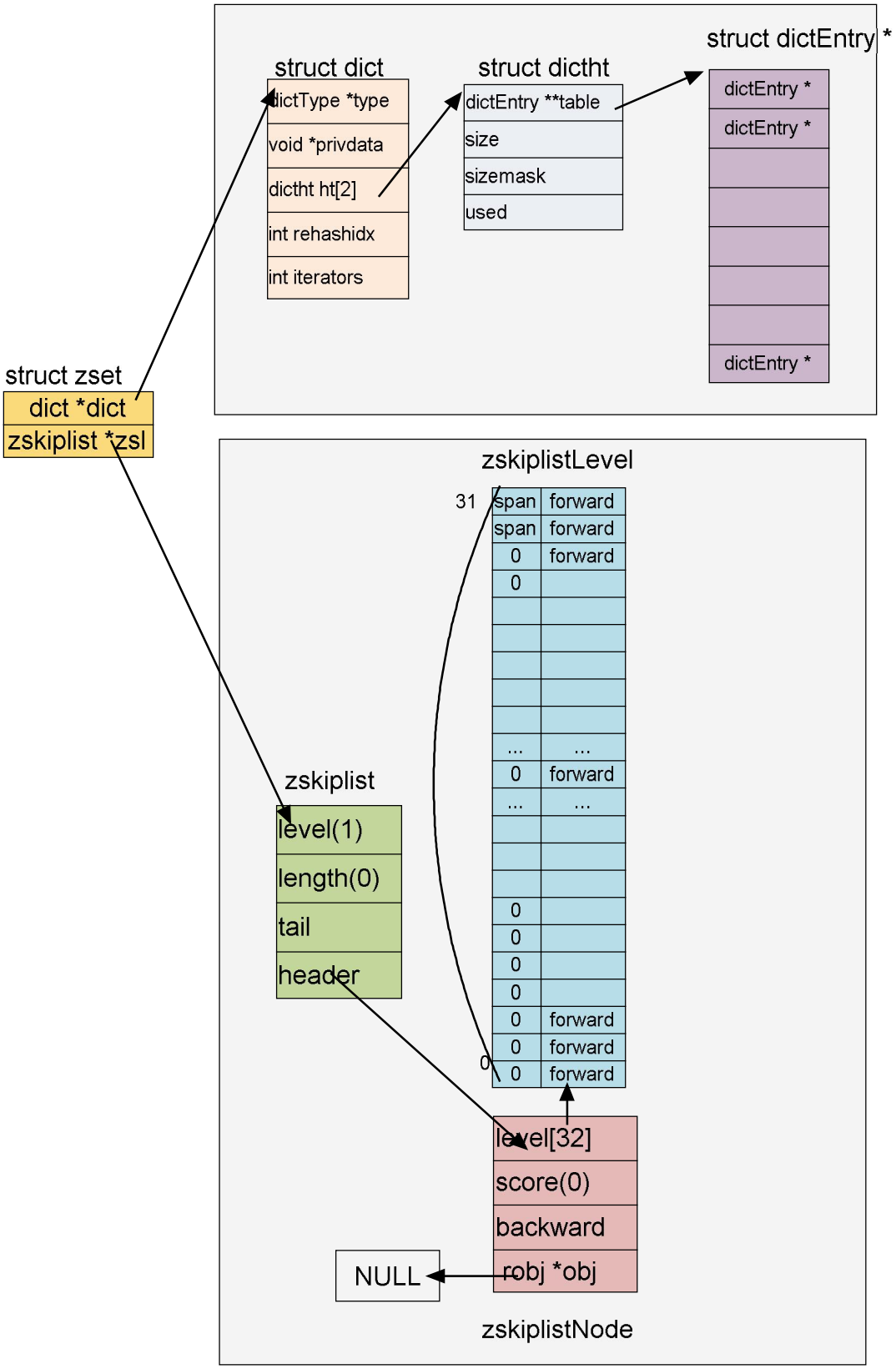
```
robj *createZsetObject(void)
{
	zset *zs = zmalloc(sizeof(*zs));
	robj *o;

	zs->dict = dictCreate(&zsetDictType,NULL);
	zs->zsl = zslCreate();
	o = createObject(REDIS_ZSET,zs);
	o->encoding = REDIS_ENCODING_SKIPLIST;
	return o;
}
```

zslCreate()函数用来创建并初如化一个 skiplist。 其中，skiplist 的 level 最大值为 ZSKIPLIST_MAXLEVEL=32 层。

```
zskiplist *zslCreate(void)
{
	int j;
	zskiplist *zsl;

	zsl = zmalloc(sizeof(*zsl));
	zsl->level = 1;
	zsl->length = 0;
	zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL,0,NULL);
	for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {
		zsl->header->level[j].forward = NULL;
		zsl->header->level[j].span = 0;
	}
	zsl->header->backward = NULL;
	zsl->tail = NULL;
	return zsl;
}
```

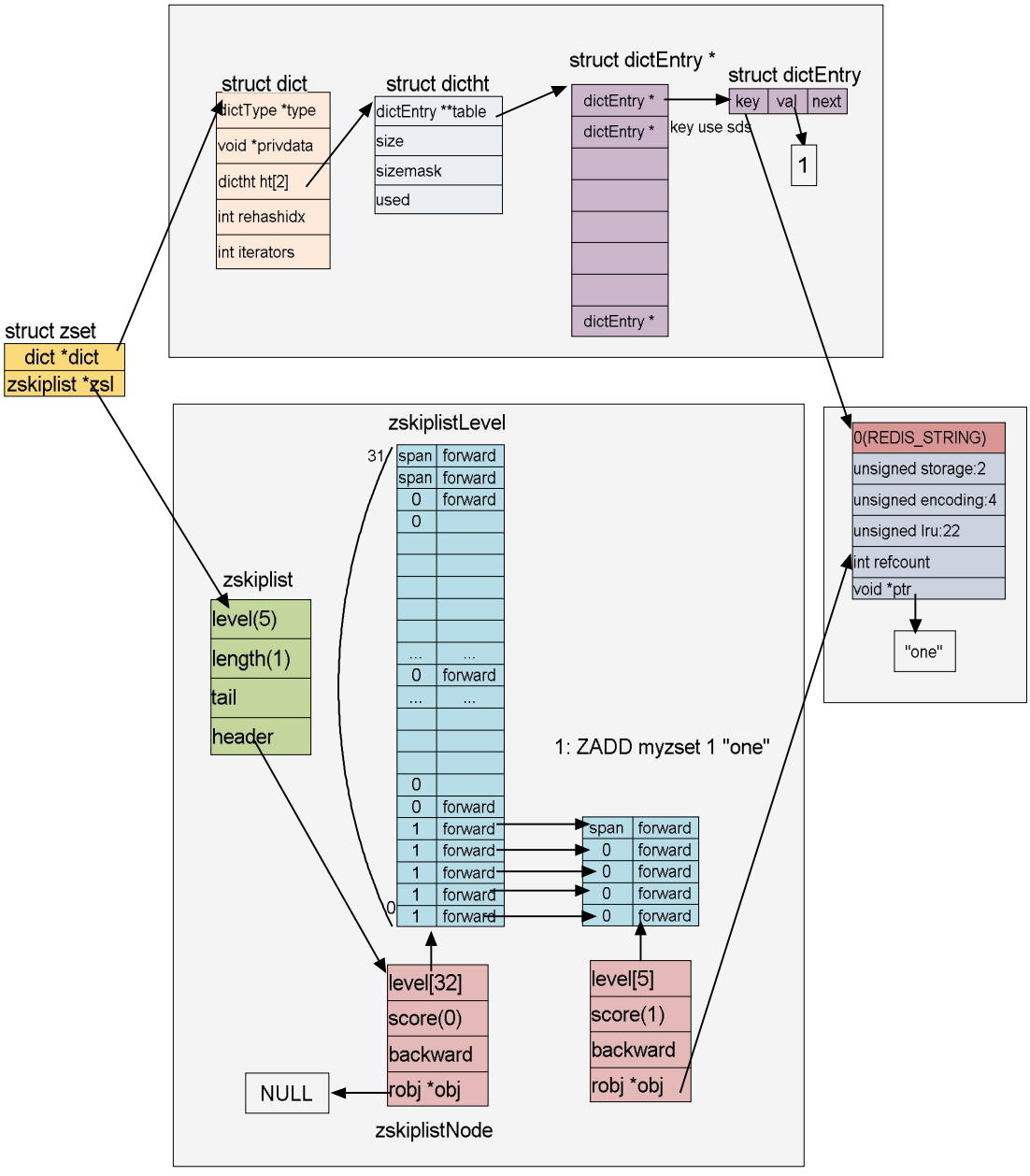


2) ZADD myzset 1 "one"

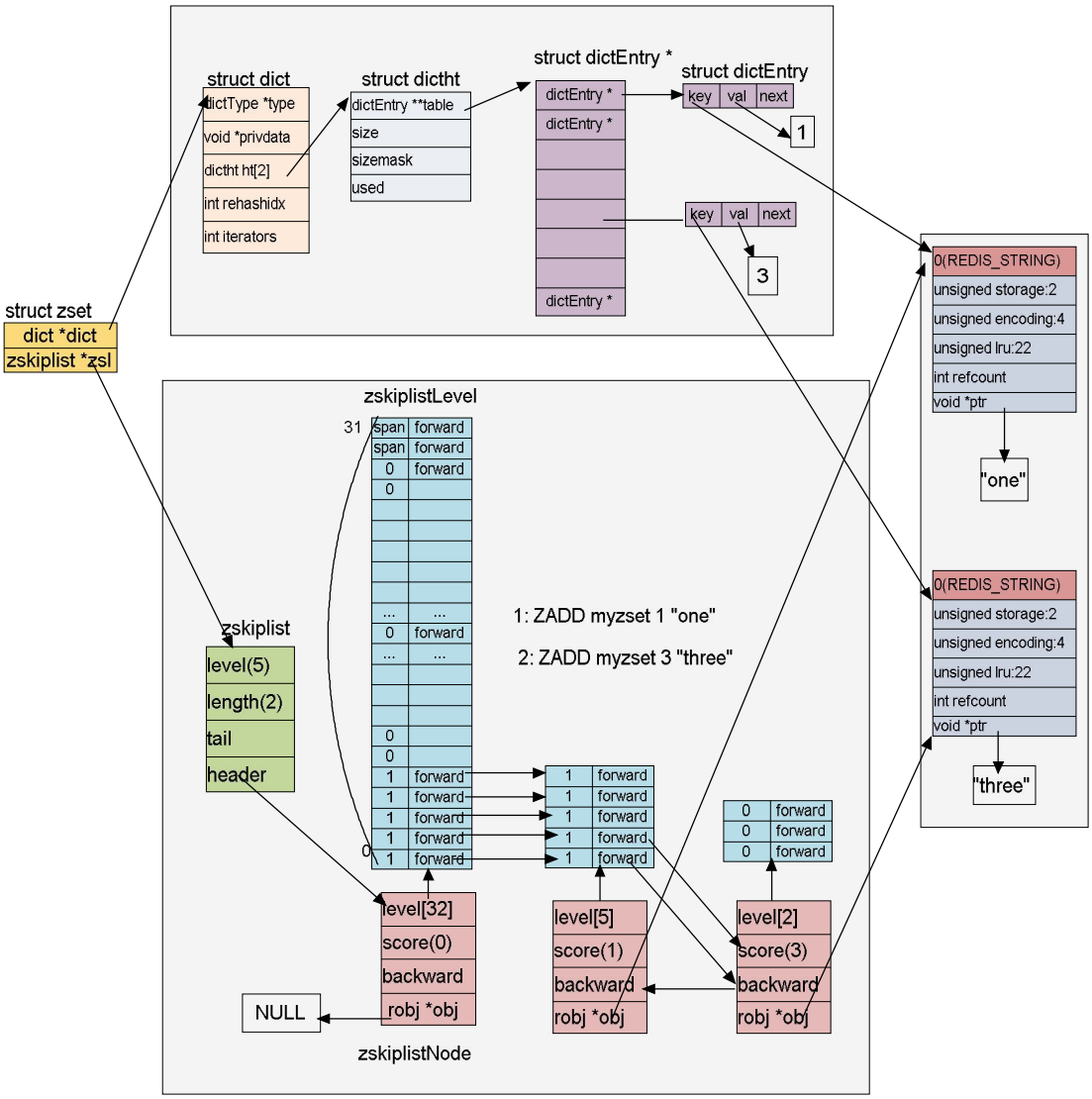
ZADD 命令格式：

ZADD key score member

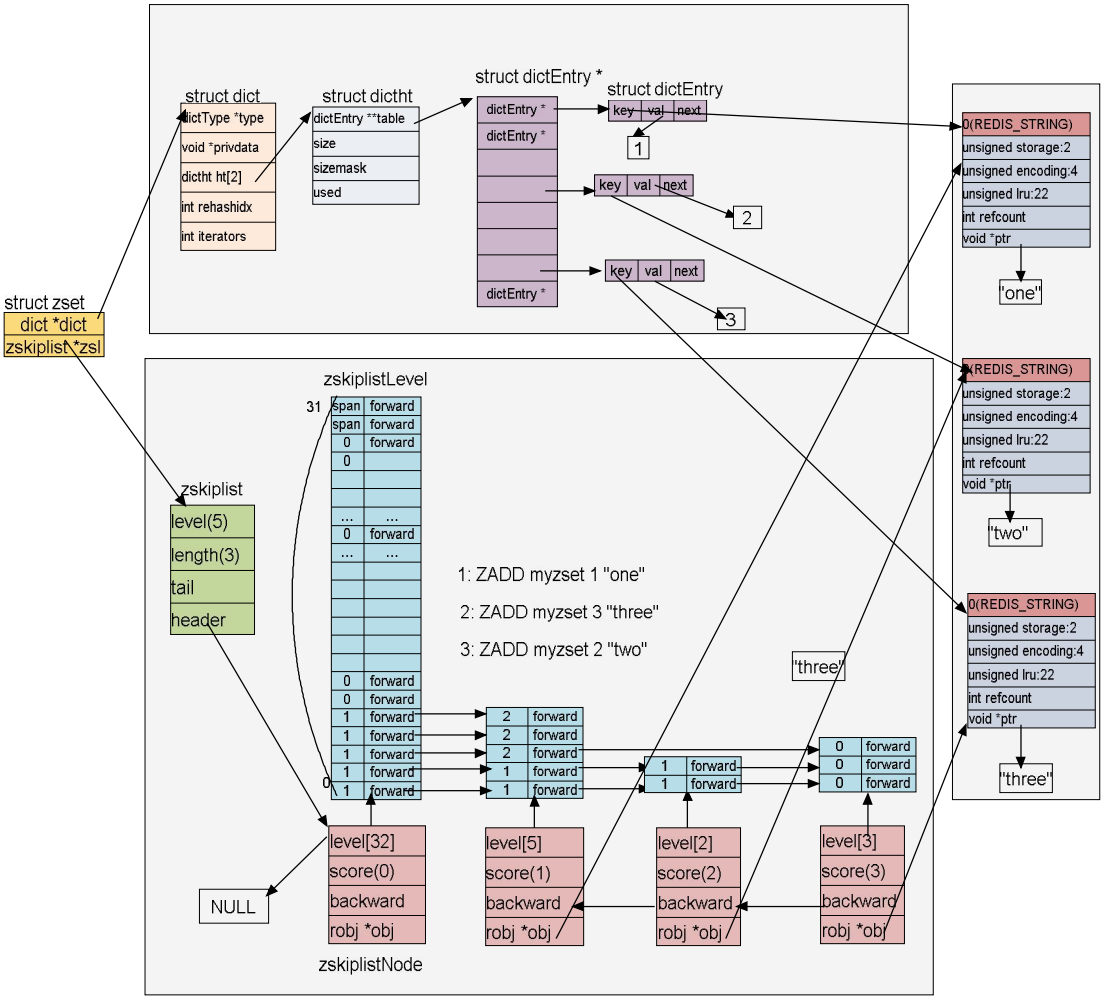
1. 根据 key 从 redisDb 进行查询，返回 zset 对象。
2. 以 member 作为 key,score 作为 value ，向 zset 的 dict 进行中插入;
3. 如果返回成功，表明 member 没有在 dict 中出现过，直接向 skiplist 进行插入。
4. 如果步骤返回失败，表明以 member 已经在 dict 中出现过，则需要先从 skiplist 中删除，然后以现在的 score 值重新插入。



3) ZADD myzset 3 "three"



4) ZADD myzset 2 "two"


struct zset
dict *dict
zskiplist *zsl
struct dict
dictType *type
void *privdata
dictht ht[2]
int rehashidx
int iterators
struct dictht
dictEntry **table
size
sizemask
used
struct dictEntry *
dictEntry *
dictEntry *
dictEntry *
struct dictEntry
key val next
1
2
3
0(REDIS_STRING)
unsigned storage:2
unsigned encoding:4
unsigned lru:22
int refcount
void *ptr
"one"
"two"
"three"
zskiplistLevel
31
span forward
0 forward
zskiplist
level(5)
length(3)
tail
header
1: ZADD myzset 1 "one"
2: ZADD myzset 3 "three"
3: ZADD myzset 2 "two"
level[32]
score(0)
backward
robj *obj
level[5]
score(1)
level[2]
score(2)
level[3]
score(3)
NULL
zskiplistNode