原因

在《Redis开发与运维》的一书中介绍了Redis的数据结构与其编码。后因为接触到sds，觉得和书里讲得没有对上，所以去了解了一下相关内容。

以下的内容几乎全部来自于《Redis设计与实现》一书。 大佬可路过。

内容

先介绍Redis的键和值在redis程序中的**对象表示**

和对象的**类型、编码及其对应关系**

之后介绍**底层的数据结构**

最后是**编码和底层的数据结构的对应**

对象表示

Redis使用 **对象RedisObject** 来表示数据库中的键和值。当创建一个键值对的时候，至少会创建两个对象，一个为键对象，一个为值对象。

举个例子，当我们执行一条 `set name limeng` 的命令的时候，redis为我们创建了两个对象，一个包含了字符串“name”的对象，和一个包含了字符串“limeng”的对象。

redisObject

```
typedef struct redisObject {
    unsigned type:4; //对象的类型
    unsigned encoding:4; //对象的编码
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                            * LFU data (least significant 8 bits frequency
                            * and most significant 16 bits access time). */
    int refcount; //对象被引用的次数
    void *ptr; //指向底层的数据结构的指针
} robj;
```

type

```
/* The actual Redis Object */
#define OBJ_STRING 0    /* String object. */
#define OBJ_LIST 1      /* List object. */
#define OBJ_SET 2       /* Set object. */
#define OBJ_ZSET 3      /* Sorted set object. */
#define OBJ_HASH 4      /* Hash object. */
```

encoding

```
/* 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 OBJ_ENCODING_RAW 0     /* Raw representation */
#define OBJ_ENCODING_INT 1     /* Encoded as integer */
#define OBJ_ENCODING_HT 2      /* Encoded as hash table */
#define OBJ_ENCODING_ZIPMAP 3  /* Encoded as zipmap */
#define OBJ_ENCODING_LINKEDLIST 4 /* No longer used: old list encoding. */
#define OBJ_ENCODING_ZIPLIST 5 /* Encoded as ziplist */
#define OBJ_ENCODING_INTSET 6  /* Encoded as intset */
#define OBJ_ENCODING_SKIPLIST 7  /* Encoded as skiplist */
#define OBJ_ENCODING_EMBSTR 8  /* Embedded sds string encoding */
#define OBJ_ENCODING_QUICKLIST 9 /* Encoded as linked list of ziplists */
#define OBJ_ENCODING_STREAM 10 /* Encoded as a radix tree of listpacks */
```

type、encoding的对应关系

type	encoding
OBJ_STRING	OBJ_ENCODING _INT
OBJ_STRING	OBJ_ENCODING _EMBSTR
OBJ_STRING	OBJ_ENCODING _RAW
OBJ_LIST	OBJ_ENCODING _ZIPLIST
OBJ_LIST	OBJ_ENCODING _QUICKLIST
OBJ_SET	OBJ_ENCODING _INTSET
OBJ_SET	OBJ_ENCODING _HT
OBJ_HASH	OBJ_ENCODING _ZIPLIST
OBJ_HASH	OBJ_ENCODING _HT
OBJ_ZSET	OBJ_ENCODING _ZIPLIST
OBJ_ZSET	OBJ_ENCODING _SKIPLIST

为什么要用redisObject

从 redisObject 可以看出它支持这些功能：

- **类型检查**

当用户输入一个命令的时候，会根据key拿到对应的值对象，根据值对象的type属性可以判断这个命令是否可以应用在这种类型的值上。如*RPUSH*，*LPUSH* 等只能用于list类型。

- **值编码**

通过提供一个 `encoding` 和 `*ptr` 可以满足在不同应用场景下，为用户提供不同的编码方式，以获取空间或者时间上的平衡。

- **对象共享**

通过引用计数来进行对象共享和内存回收

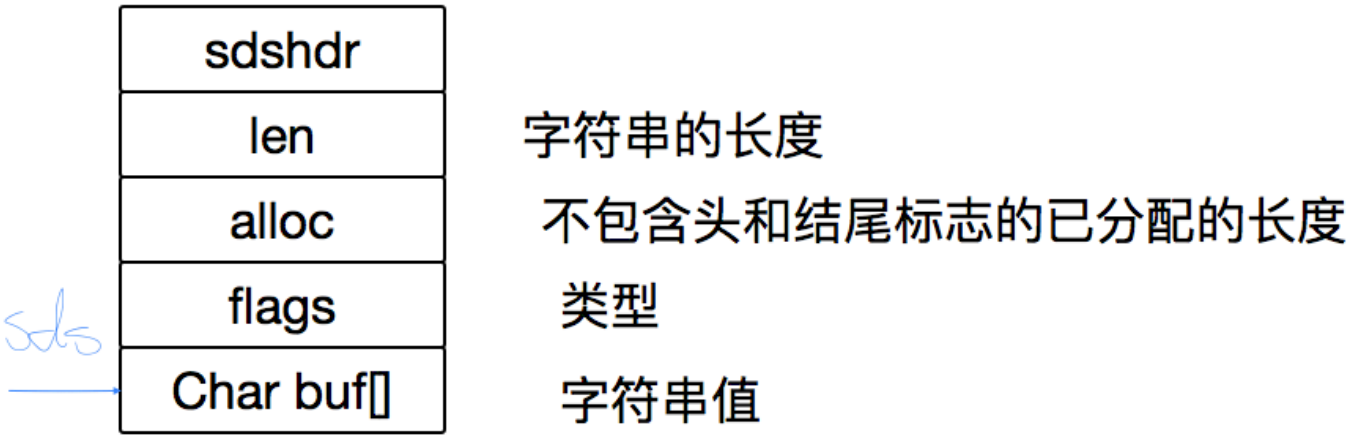
底层数据结构

sds

sds出现的原因

1. 传统的C语言字符串使用空字符作为结尾标志。只能满足一些文本类型的数据，对于音频图像或者视频的二进制编码难以支持
2. 每次增长或者缩短一个C字符串，程序总要对保存这个字符串的数组进行一次内存重分配的操作。而Redis作为内存数据库，数据可能被频繁修改，由此可能会对性能造成影响
3. C语言字符串不保存长度的信息，每次获取都是一个O(N)的操作 因此，redis自己设计了一种数据结构，用来保存键或者字符串的值。

sds的定义



```
//sds指针被直接指向字符串数组的起始位置
typedef char *sds;


//在创建一个sds字符串的时候，先根据长度找到对应的sdshdr结构。再分配空间和拷贝字符串
/* Note: sdshdr5 is never used, we just access the flags byte directly.
 * However is here to document the layout of type 5 SDS strings. */
struct __attribute__ ((__packed__)) sdshdr5 {
    unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr8 {
    uint8_t len; /* used */
    uint8_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr16 {
    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 {
    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 {
    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[];
};
```

使用多种sdshdr头结构可以节省内存空间的使用。

sds如何解决传统C语言字符串中的问题

存储二进制数据

以len属性来读取存在buf[]中的数据，而不是读到空字符就结束

内存重分配

redis为了避免内存重分配的开销，采用了减少内存重分配次数的策略，这是一种类似于ArrayList，hashMap

等扩容的一种操作。被称为**“空间预分配”**

当需要对sds进行修改的时候：

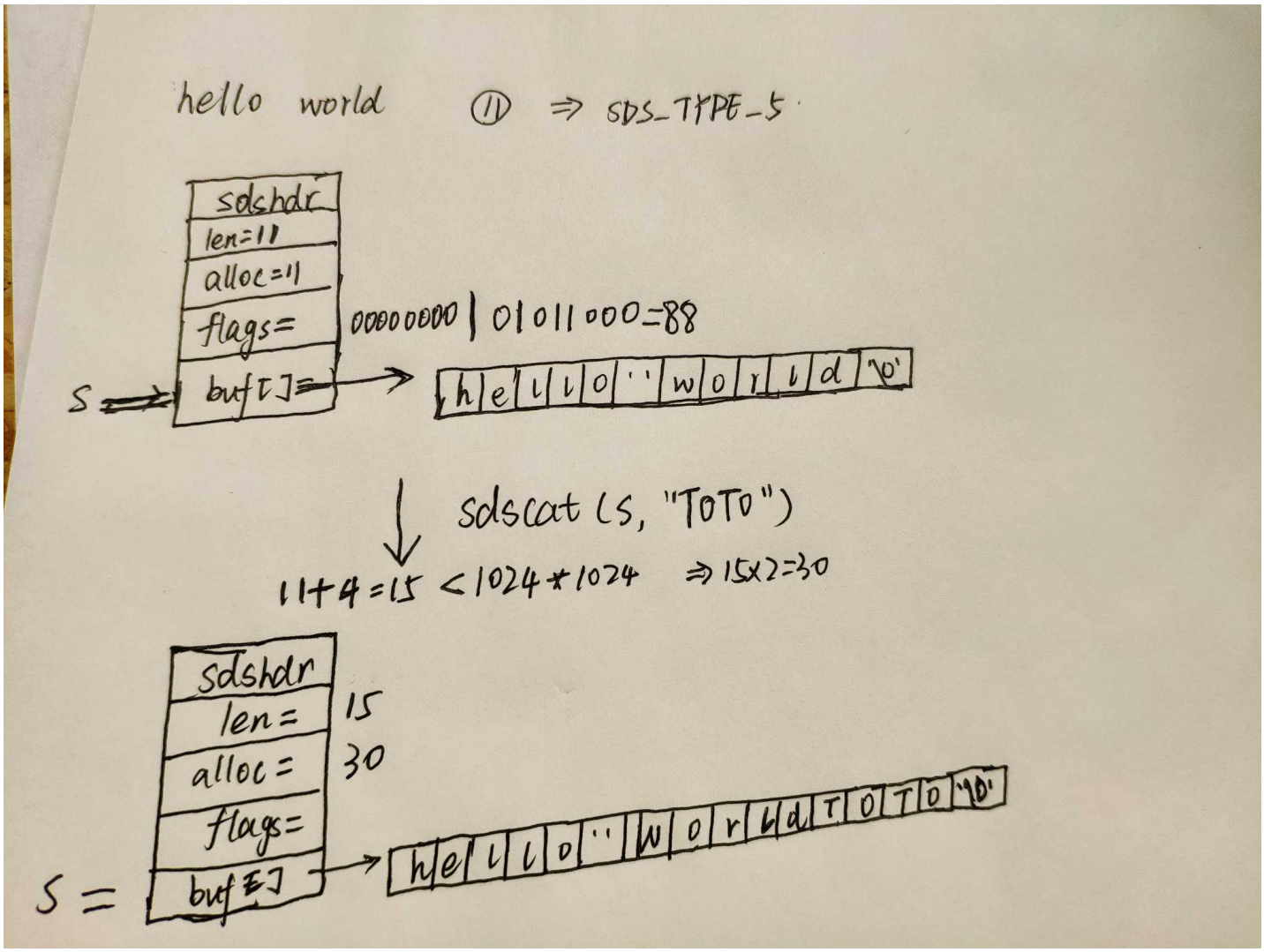
1. api会检查sds的空间是否满足大小
2. 如果不满足，会将sds的空间扩展值执行修改所需的大小。然后才执行修改的操作。

以sdscat为例介绍sds的空间预分配问题。

```
sds sdscatlen(sds s, const void *t, size_t len) {
    size_t curlen = sdslen(s);
    //为新的sds字符串分配空间
    s = sdsMakeRoomFor(s,len);
    if (s == NULL) return NULL;
    memcpy(s+curlen, t, len);
    sdssetlen(s, curlen+len);
    s[curlen+len] = '\0';
    return s;
}


sds sdsMakeRoomFor(sds s, size_t addlen) {
    void *sh, *newsh;
    //获取现有的sds字符串的可用的空间
    size_t avail = sdsavail(s);
    size_t len, newlen;
    char type, oldtype = s[-1] & SDS_TYPE_MASK;
    int hdrlen;

    /* Return ASAP if there is enough space left. */
    //如果空间足够，则直接返回。
    if (avail >= addlen) return s;

    len = sdslen(s);
    sh = (char*)s-sdsHdrSize(oldtype);
    newlen = (len+addlen);
    //如果修改后的字符串的长度小于 1M，则扩展两倍的空间
    if (newlen < SDS_MAX_PREALLOC)
        newlen *= 2;
    else
    //如果大于1M，则多扩展1M的空间
        newlen += SDS_MAX_PREALLOC;

    //确认新的sdshdr的类型
    type = sdsReqType(newlen);

    /* Don't use type 5: the user is appending to the string and type 5 is
     * not able to remember empty space, so sdsMakeRoomFor() must be called
```

```
     * at every appending operation. */
    if (type == SDS_TYPE_5) type = SDS_TYPE_8;

    hdrlen = sdsHdrSize(type);
    if (oldtype==type) {
        //分配空间
        newsh = s_realloc(sh, hdrlen+newlen+1);
        if (newsh == NULL) return NULL;
        s = (char*)newsh+hdrlen;
    } else {
        /* Since the header size changes, need to move the string forward,
         * and can't use realloc */
         * /
        //分配空间
        newsh = s_malloc(hdrlen+newlen+1);
        if (newsh == NULL) return NULL;
        memcpy((char*)newsh+hdrlen, s, len+1);
        s_free(sh);
        s = (char*)newsh+hdrlen;
        s[-1] = type;
        sdssetlen(s, len);
    }
    sdssetalloc(s, newlen);
    return s;
}
```

> 空间预分配
>
> 用于优化sds的字符串增长操作：当sds的api对一个sds进行修改的时候，并且需要对sds进行空间扩展的时候，程序不仅会为sds分配修改所需的空间，也会为sds分配额外的未使用空间
>
> 额外未使用的空间的数量由一下公式决定： 如果sds修改后，sds的长度将小于1M，则程序分配和len大小一样的未使用空间。 如果len>=1M，则程序分配1M的未使用空间
>
> 惰性空间释放
>
> 用来优化缩短字符串的操作。当缩短字符串时，不立即使用内存重分配来回收释放出来的空间。len=缩短后的长度，而alloc还是原来的已分配的空间。



链表和链表节点

Redis为list提供的结构与我们常见的链表是很相似的

已经不再使用

list与listnode的定义

```
typedef struct list {
    listNode *head; //表头
    listNode *tail; //表尾
    void *(*dup)(void *ptr); //节点复制函数
    void (*free)(void *ptr); //节点释放函数
    int (*match)(void *ptr, void *key); //节点值匹配函数
    unsigned long len;//节点数量
} list;

//双向链表
typedef struct listNode {
    struct listNode *prev;
    struct listNode *next;
    void *value;
} listNode;
```

Redis链表的特性

- 双端
- 无环
- 有表头和表尾
- 有长度
- 可以根据节点值实现多态

字典

字典存在的原由

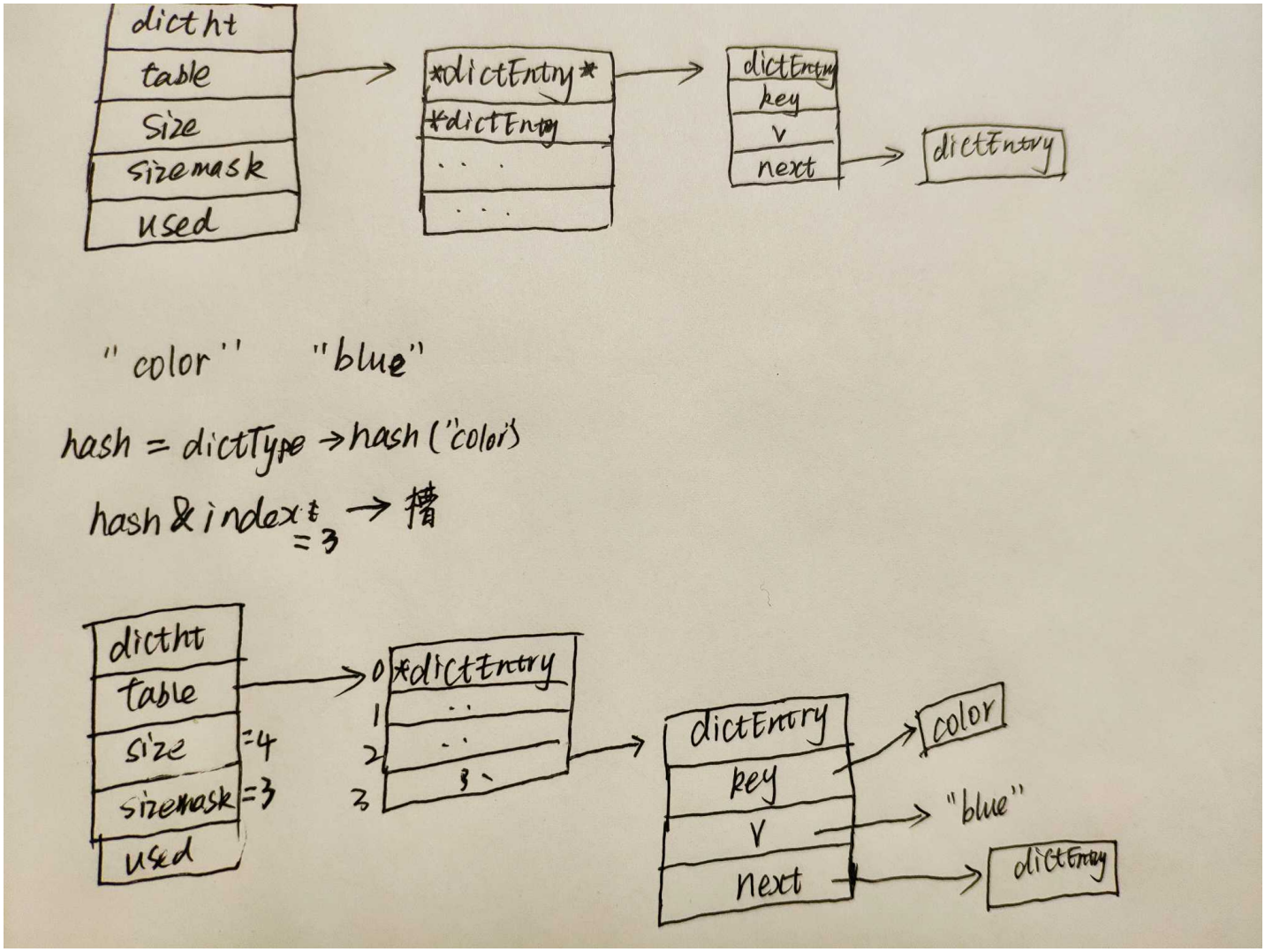
1. C语言没有内置键值对这种抽象数据结构
2. 被用来保存键和值的关系
3. 被用来保存hash键的值

字典的定义

```
/* This is our hash table structure. Every dictionary has two of this as we
 * implement incremental rehashing, for the old to the new table. */
//这是redis的hash结构，每个字典有两个 dictht ,用来做rehash操作
typedef struct dictht {
    dictEntry **table; //一个数组，每个元素都指向一个dictEntry.
    unsigned long size; //数组的大小
    unsigned long sizemask; //hash表大小掩码，用来计算索引值，总是等于size-1
    unsigned long used;//该hash表已有节点的数量
} dictht;

typedef struct dict {
    dictType *type; //类型特定函数
    void *privdata; //私有数据
    dictht ht[2]; //两个dictht
    long rehashidx;//rehash索引，不在rehash的时候为-1 /* rehashing not in progress if
 rehashidx == -1 */
    unsigned long iterators; /* number of iterators currently running */
} dict;


//类型特定函数
typedef struct dictType {
    uint64_t (*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;

//每一个条目
typedef struct dictEntry {
    void *key; //键
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v; //值
    struct dictEntry *next; //指向下一个节点
} dictEntry;
```

向dict添加一个元素

初始大小是0，当添加一个元素的时候，会扩展至4个桶。



dict的rehash

当添加一个元素的时候，如果负载因子大于一定的值，则进行扩展操作。

扩展后的桶的数量为 >= used * 2 的第一个2的n次幂。

并设置rehashidx = 0. 添加的元素将会添加到ht[1]中

每次添加元素时，都会判断rehashidx,如果不等于-1，则表明正在进行rehash操作。

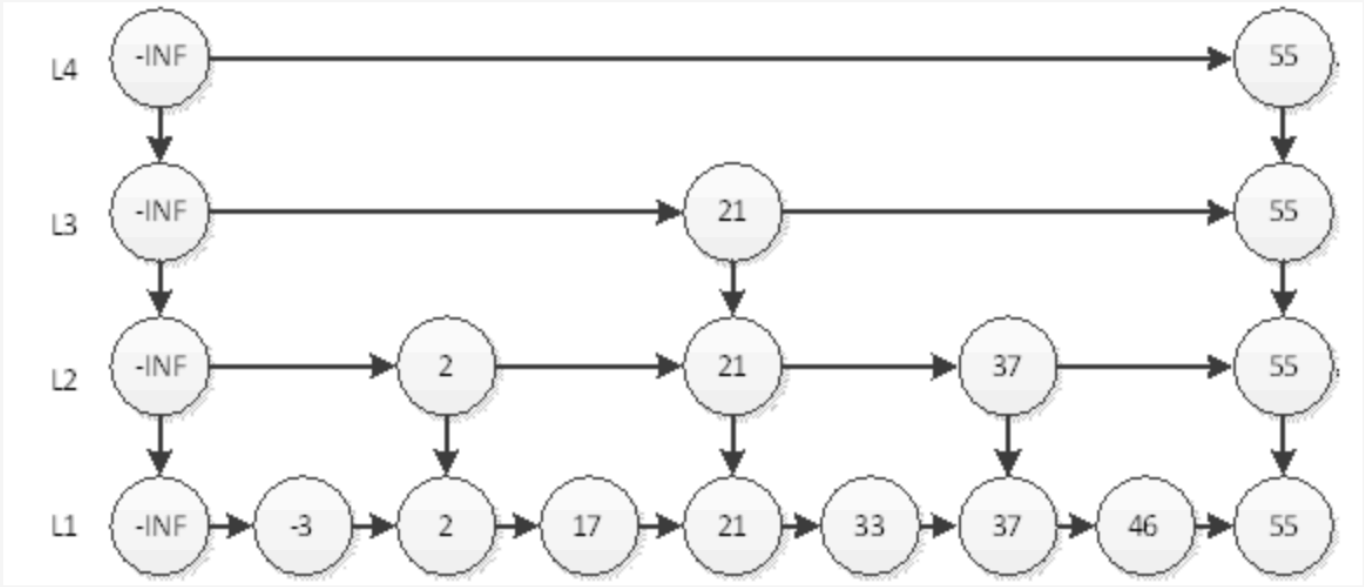
redis提供了一种**渐进式hash**，多次渐进地完成扩展或者收缩操作（每次移动部分桶），来避免大hashtable在rehash时造成的性能影响。这也是rehashidx属性的意义所在。

跳跃表

跳跃表是有序集合键的一种底层结构。

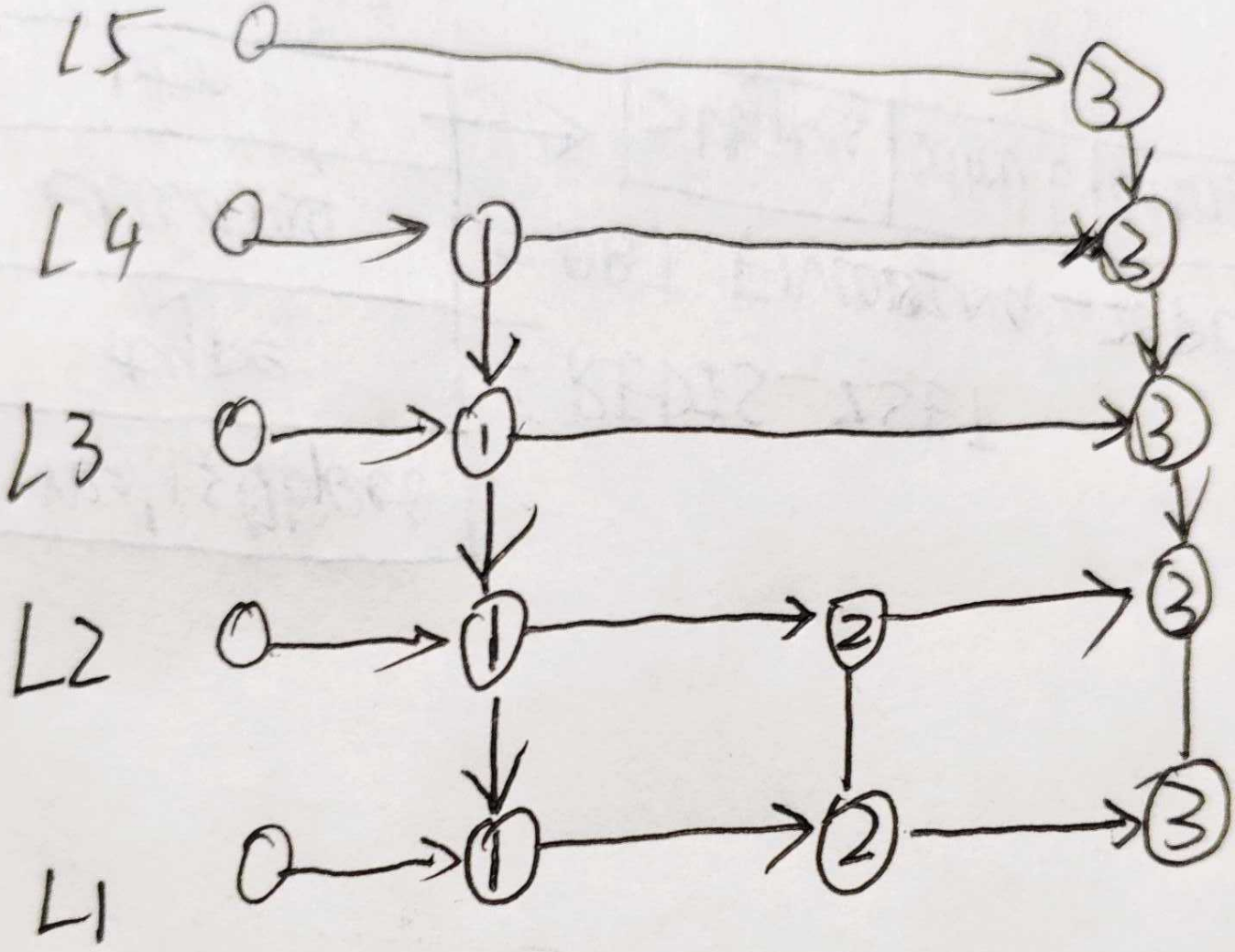
跳跃表结构简单介绍

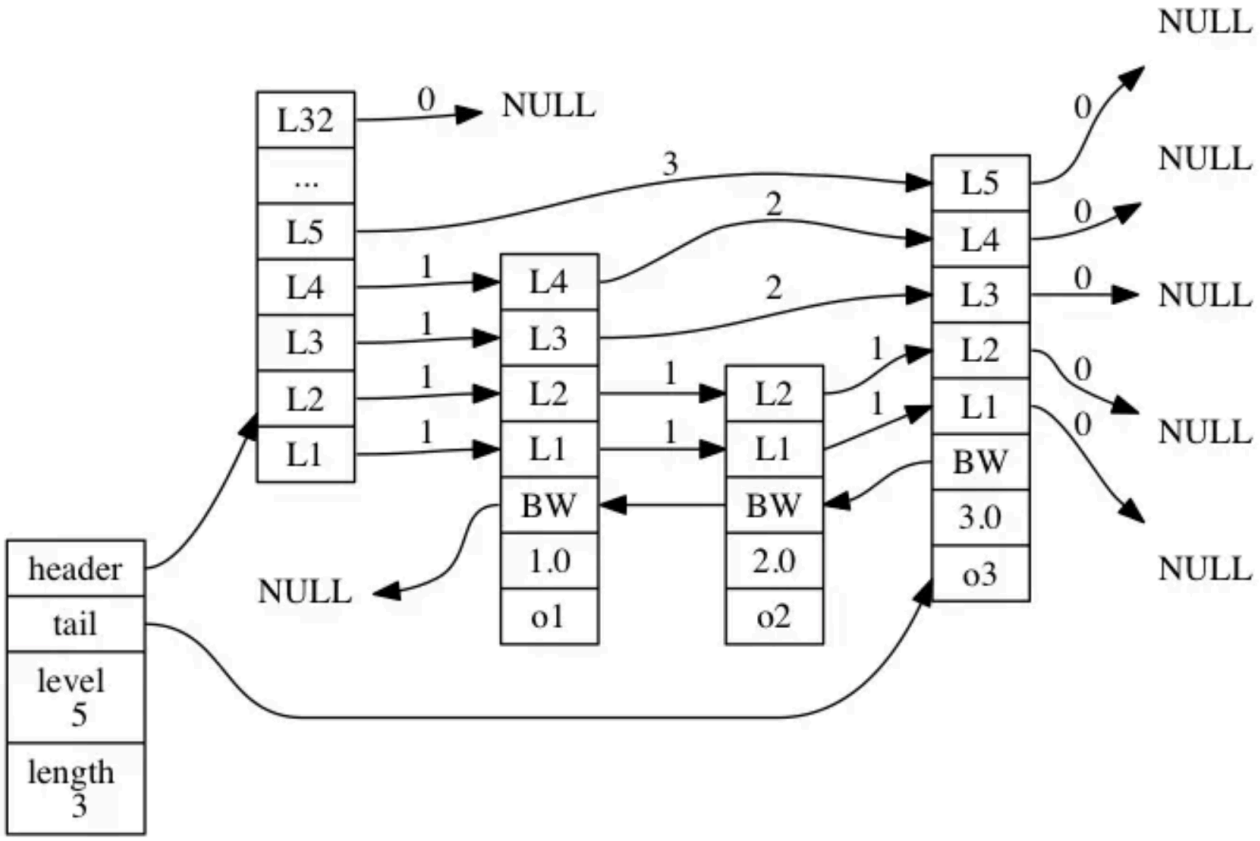
跳跃表：为了让查询、添加、删除元素更快而基于链表形成的一种结构。如下图



Redis中的skiplist定义

```
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;
```


L5
L4
L3
L2
L1
0
1
2
3



整数集

整数集intset是集合键的一种实现之一，当一个set只包含整数元素且元素数量不多时，使用intset作为底层实现。

intset的定义

```
typedef struct intset {
    uint32_t encoding;//编码
    uint32_t length; //长度
    int8_t contents[]; //内容
} intset;
```

虽然contents数组是int8类型的，但其存储的值的真正类型由uint32_t决定。
`INTSET_ENC_INT16 INTSET_ENC_INT32 INTSET_ENC_INT64`

升级

当将一个新的元素放入intset中时，新元素的类型比encoding所指示的要长，则整数集合需要升级操作。然后再将新元素添加到集合中。

升级分为三个步骤进行：

1. 扩充contents空间大小

2. 将原有元素转换类型，并移动到正确的位置，移动过程中保证有序性
3. 将新元素添加到底层数组里面。

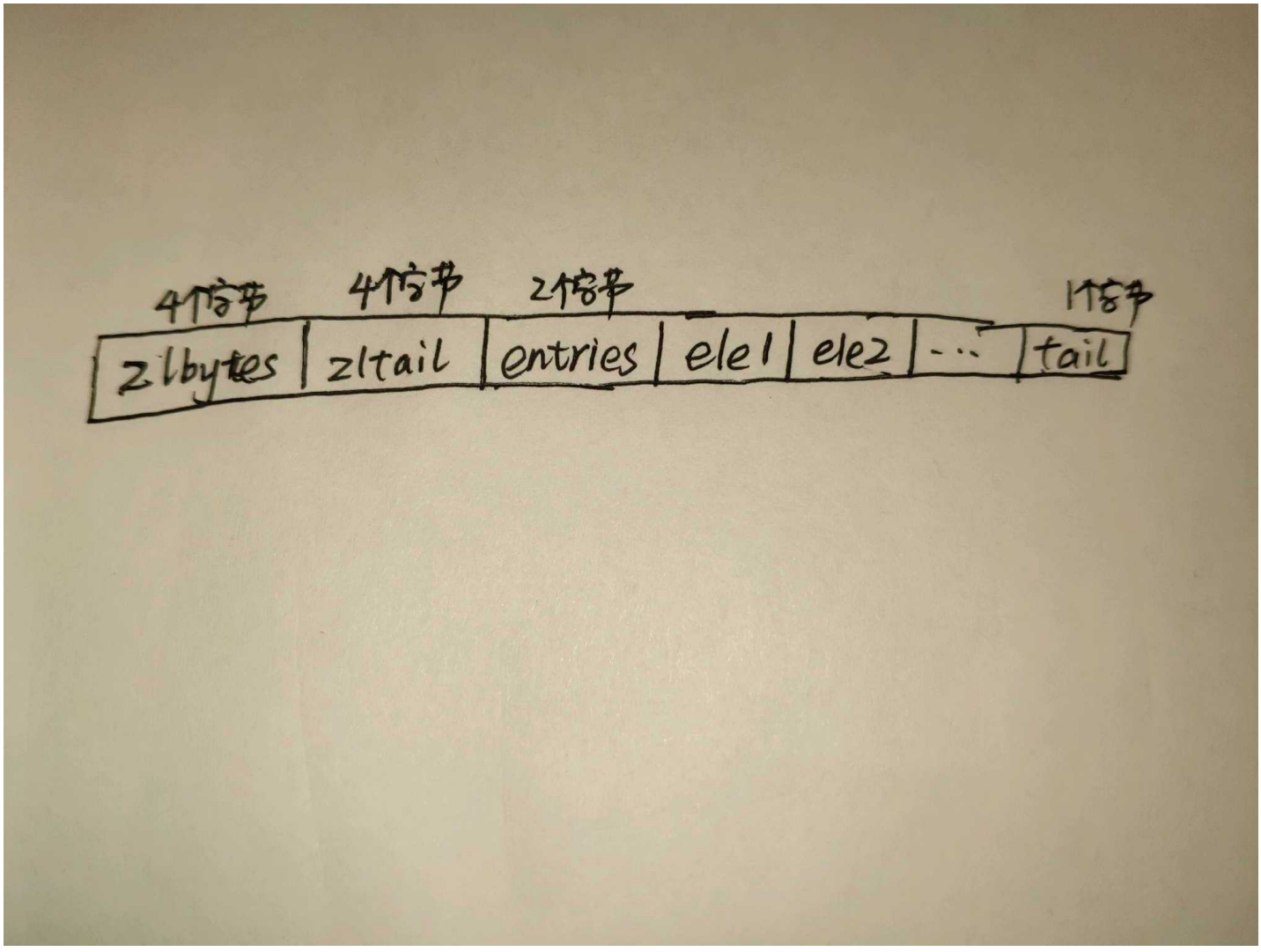
新的元素因为总是比原有所有元素要么大要么小，所以总是放在头部或者尾部。

升级带来的一个好处是**节省内存**。

压缩列表

ziplist是list键或者hash键的底层实现。当一个列表建的元素只有少量时，并且每个值元素时小整数或者比较短的字符串时，采用压缩列表作为底层实现。

ziplist的结构



zlbytes： 记录整个压缩列表占用的字节数

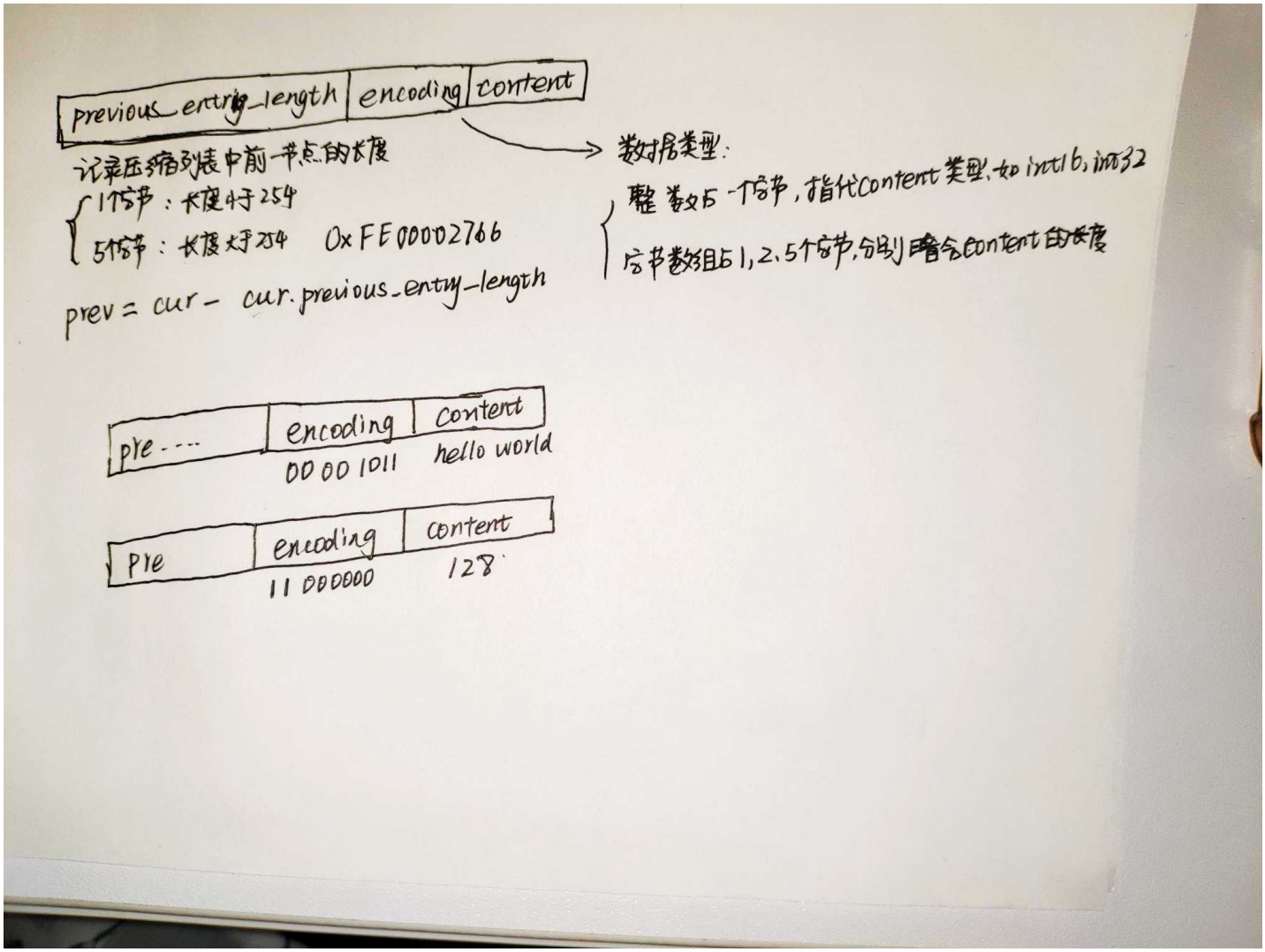
zltail : 记录压缩列表尾节点的偏移量。如ele10是尾节点，则ztail=ele10-ziplist

entries : 记录压缩列表的节点数

ele : 节点

tail : 0xFF 特殊值，用来标志压缩列表的结束。

zipentry 的结构



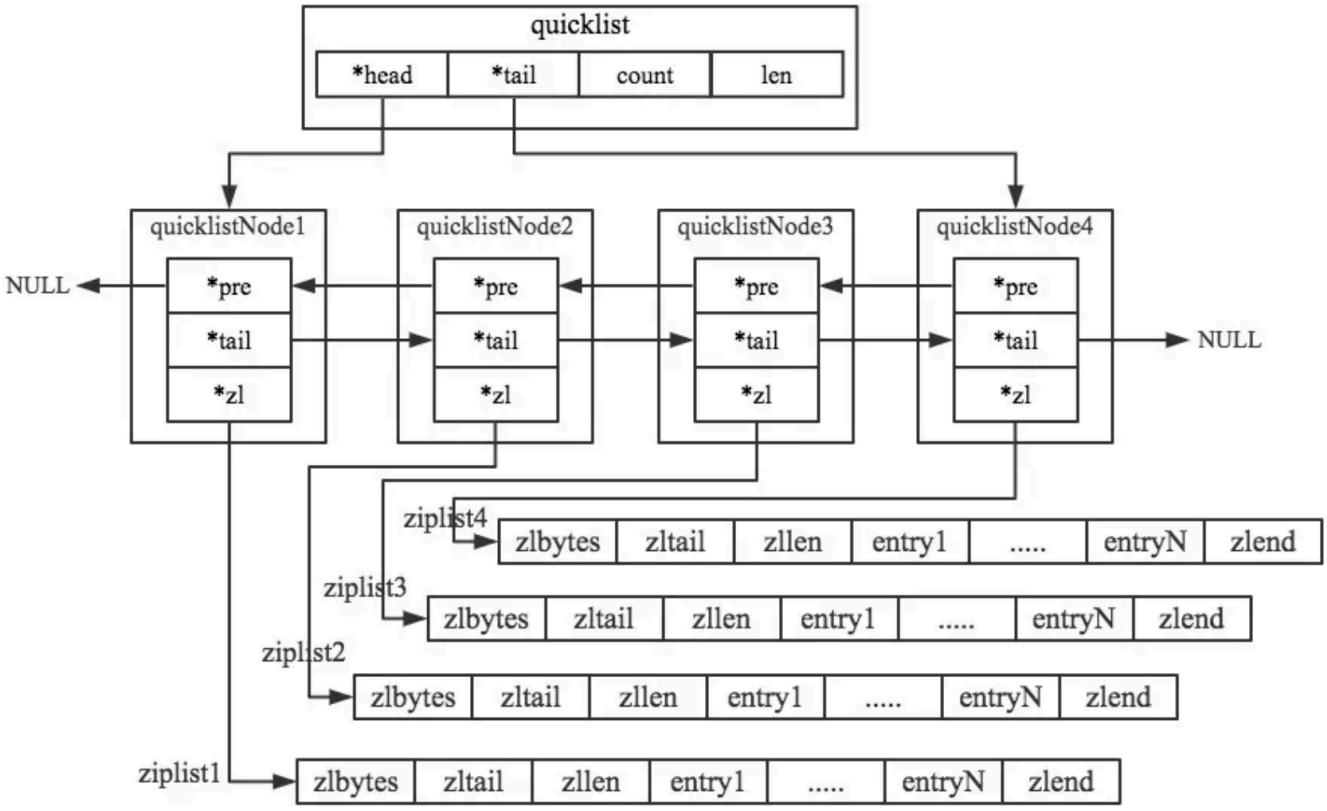
quicklist

quicklist是list的一种底层实现方式。

quicklist的定义

quicklist = 一个双向链表，链表中的每个元素都是一个ziplist

```
typedef struct quicklistNode {
    struct quicklistNode *prev;
    struct quicklistNode *next;
    unsigned char *zl;  //指向一个ziplist
    unsigned int sz;             /* ziplist size in bytes */
    unsigned int count : 16;     /* count of items in ziplist */
    unsigned int encoding : 2;   /* RAW==1 or LZF==2 */
    unsigned int container : 2;  /* NONE==1 or ZIPLIST==2 */
    unsigned int recompress : 1; /* was this node previous compressed? */
    unsigned int attempted_compress : 1; /* node can't compress; too small */
    unsigned int extra : 10; /* more bits to steal for future usage */
} quicklistNode;


typedef struct quicklist {
    quicklistNode *head;
    quicklistNode *tail;
    unsigned long count;        /* total count of all entries in all ziplists */
    unsigned long len;          /* number of quicklistNodes */
    int fill : 16;              /* fill factor for individual nodes */
    unsigned int compress : 16; /* depth of end nodes not to compress;0=off */
} quicklist;
```



为什么使用quicklist

quicklist出现之前，使用的是linkedlist。见链表。

ziplist的好处在于：内存连续且相对于linkedlist能够节省内存。但每次插入和删除都会修改previous*entry*length，极端情况下会引发连锁更新。

linkedlist的好处在于：插入和删除操作快，但是内存是不连续的，每个节点指向一个redisObject，相对于ziplist较耗内存。

quicklist的存在则结合两者的优点。使用ziplist来节省内存，使用双向链表来减小插入和删除的复杂度。

编码与其对应的底层数据结构

REDIS_STRING

OBJ_ENCODING _INT

如果一个字符串对象保存的是整数值，并且这个整数值可以用long类型来表示，那么，字符串对象会将整数值保存在redisObject的ptr属性里面，并将字符串对象的编码值设置为int。

```
//在执行set命令的时候，先尝试对对象进行编码，
//如果值<20位，且使用了maxmemory的话，则用创建一个整数对象，该整数对象如下：
 o->encoding = OBJ_ENCODING_INT;
 o->ptr = (void*) value;
```

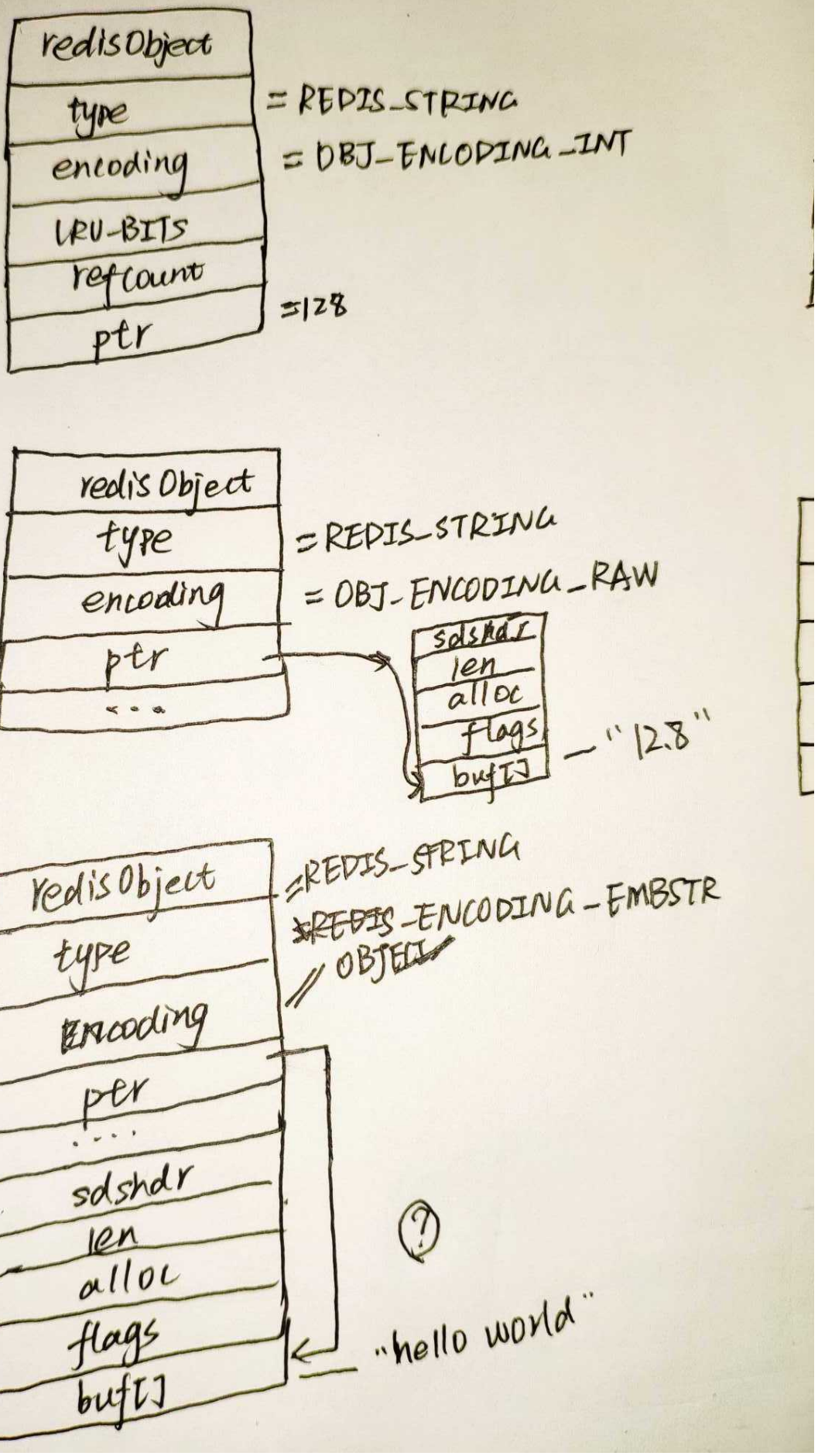
OBJ_ENCODING _RAW

如果字符串的长度大于44个字节，那么字符串将使用RAW编码的方式来保存这个字符串的值。

OBJ_ENCODING _EMBSTR

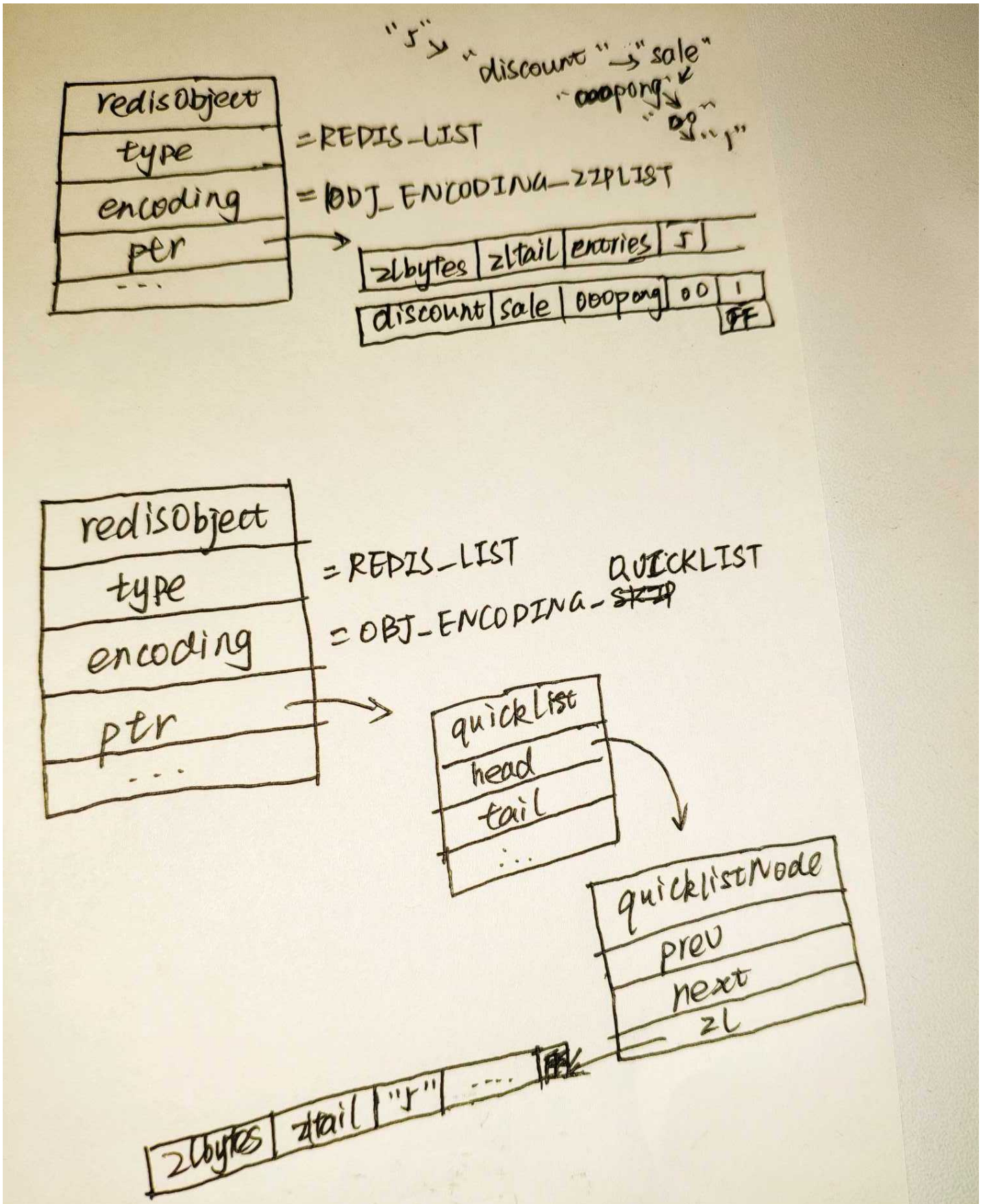
如果字符串对象保存的是一个字符串的值，并且这个字符串的长度小于等于44字节（书上说的是44个字节），那么字符串对象将使用embstr编码的方式来保存这个字符串的值。

embstr是一种专门用来保存短字符串的数据结构。和RAW编码一样，都用redisObject和sdshdr来表示字符串对象。但是embstr通过调用1次内存分配函数来分配一块连续的空间。**所以可以降低内存分配次数和内存释放次数**。

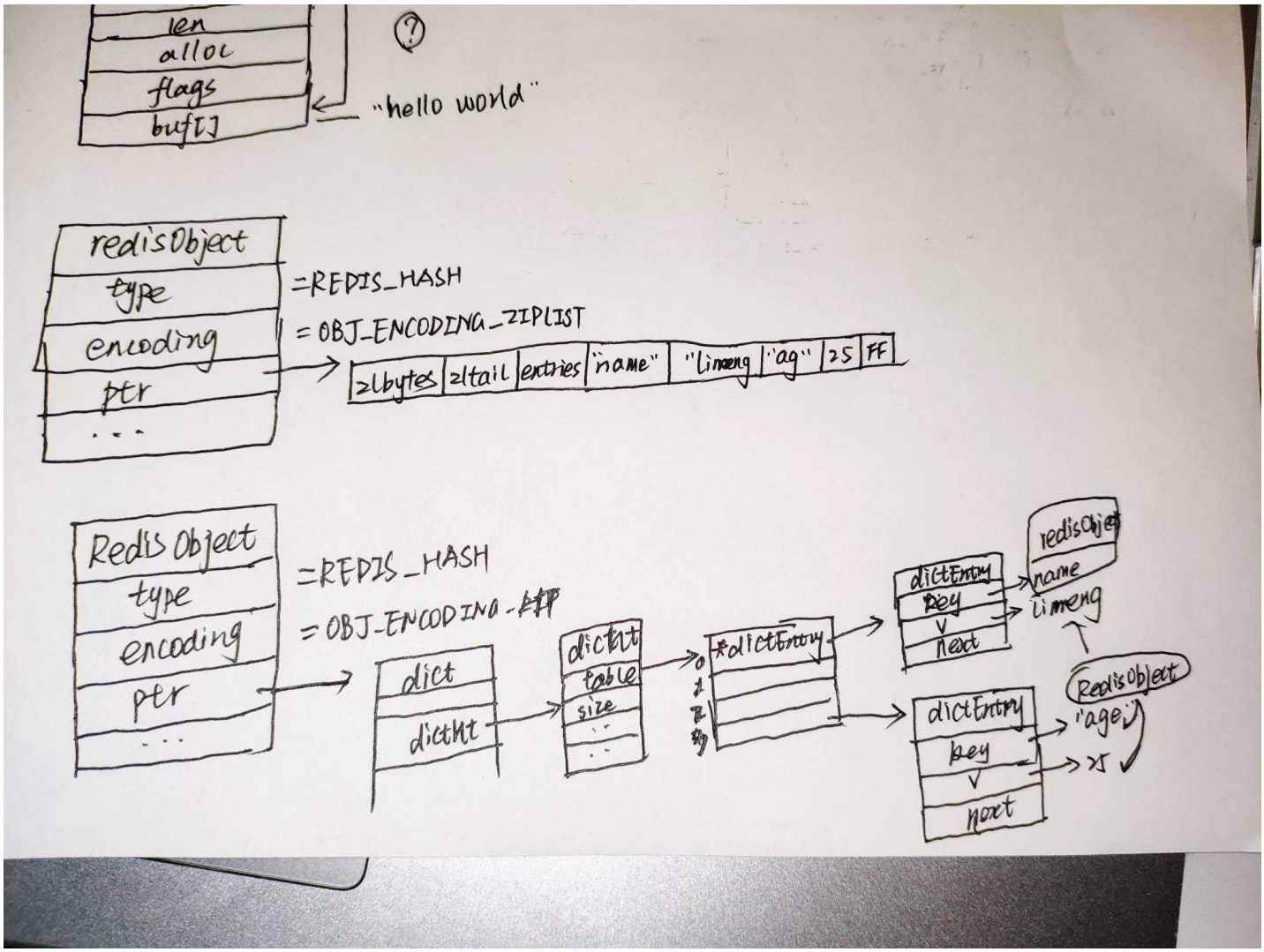

redisObject
type
encoding
LRU-BITS
refcount
ptr
=REDIS_STRING
=OBJ_ENCODING_INT
=128
redisObject
type
encoding
ptr
...
=REDIS_STRING
=OBJ_ENCODING_RAW
sdshdr
len
alloc
flags
buf[]
="12.8"
redisObject
type
Encoding
ptr
...
sdshdr
len
alloc
flags
buf[]
=REDIS_STRING
=REDIS_ENCODING_EMBSTR
// OBJECT
?
"hello world"

REDIS_LIST

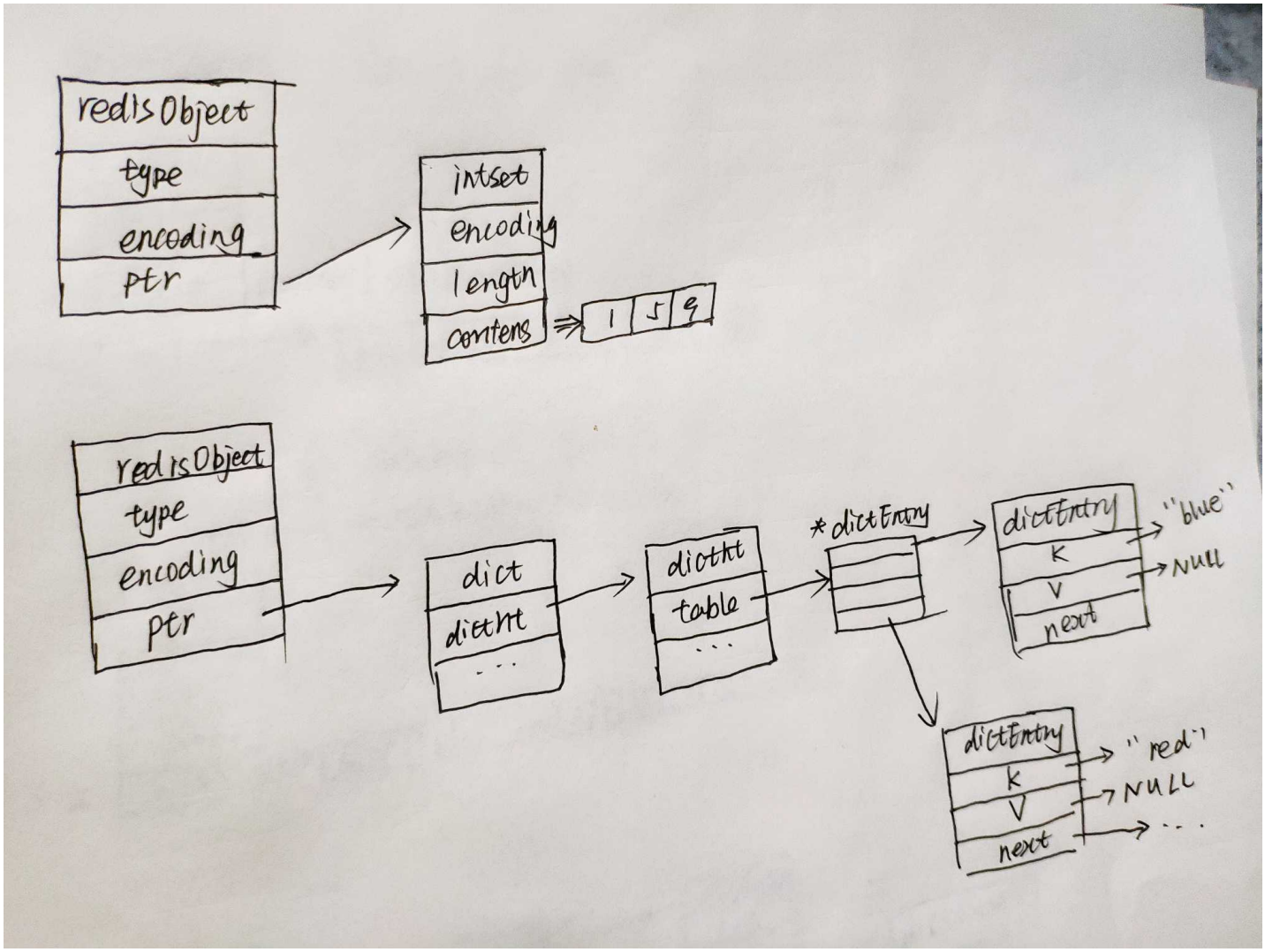
当list中元素数量较少或者长度较小的时候，使用ziplist编码，否则使用quicklist编码。


"5"
"discount" → "sale"
"ooopong"
"99"
"1"
redisObject
type
encoding
ptr
...
=REDIS_LIST
=OBJ_ENCODING_ZIPLIST
zlbytes
zltail
entries
5
discount
sale
ooopong
00
1
FF
redisObject
type
encoding
ptr
...
=REDIS_LIST
QUICKLIST
=OBJ_ENCODING_SKIP
quicklist
head
tail
...
quicklistNode
prev
next
zl
zlbytes
ztail
"5"
...
FF

REDIS_HASH

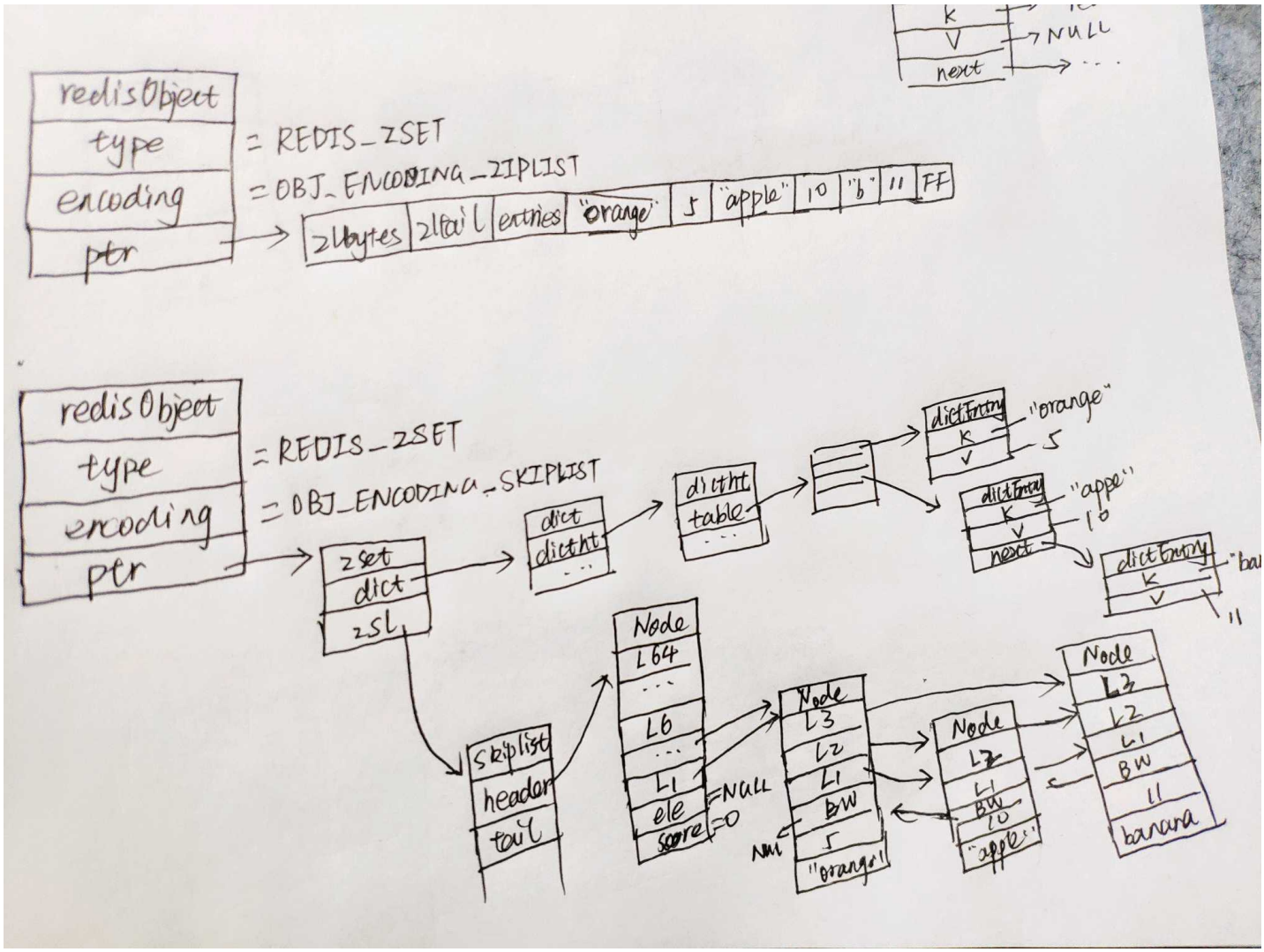


REDIS_SET



为什么不用ziplist呢？插入时复杂?

REDIS_ZSET



为什么要维护一个dict和一个zsl? 大概是因为想利用dict维护元素和分支的关系，以使ZSCORE 的复杂度为O(1)。由想利用skiplist的快速查找和有序性。 这种同时维护两个结构的做法并不会使用两倍的空间。因为dictEntry和skiplistNode都指向一个redisObject

总结

通过不同编码实现效率和空间的平衡。效率和空间都是Redis追求的目标. 当元素数量较少时，牺牲时间复杂度以减少空间。当元素数量较大时，选取能够快速存取的结构，尽管可能会浪费更多空间。