零声教育 Mark 老师 QQ: 2548898954

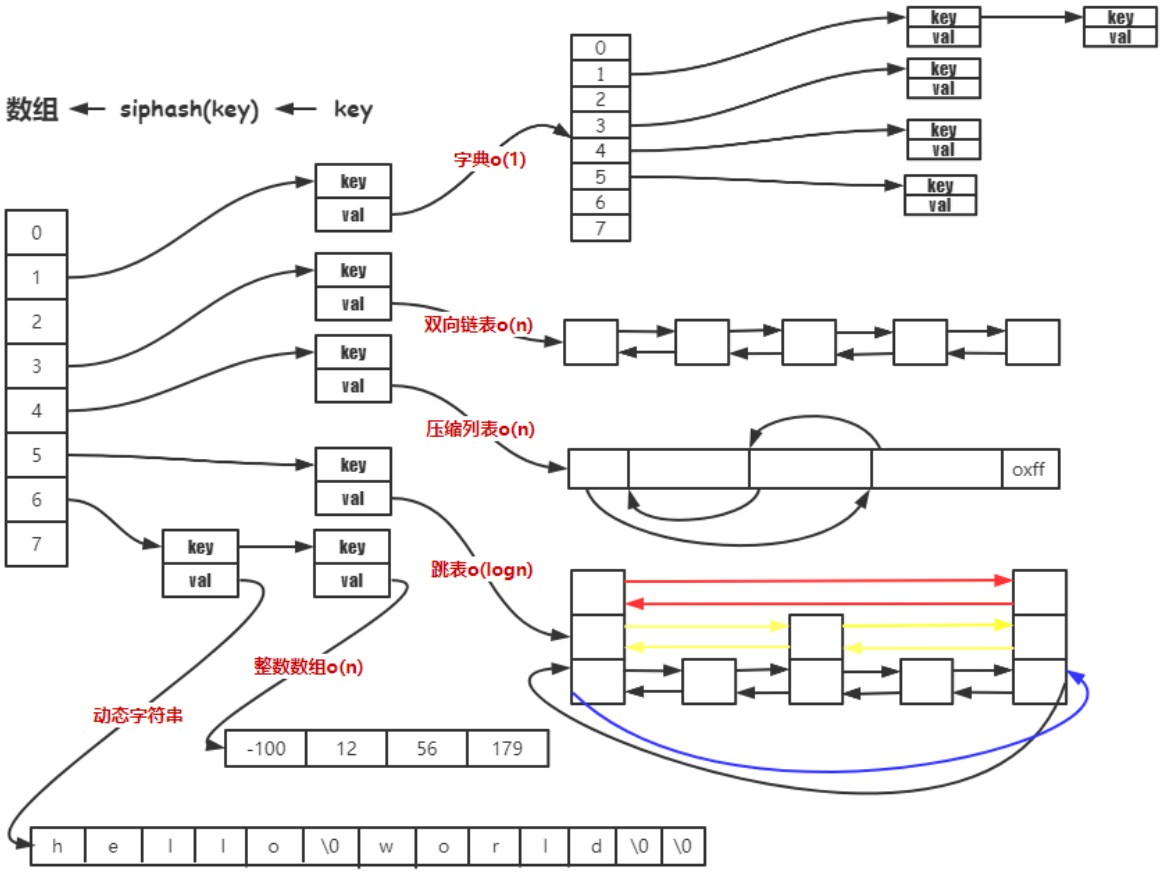
redis源码学习

建议学习方法

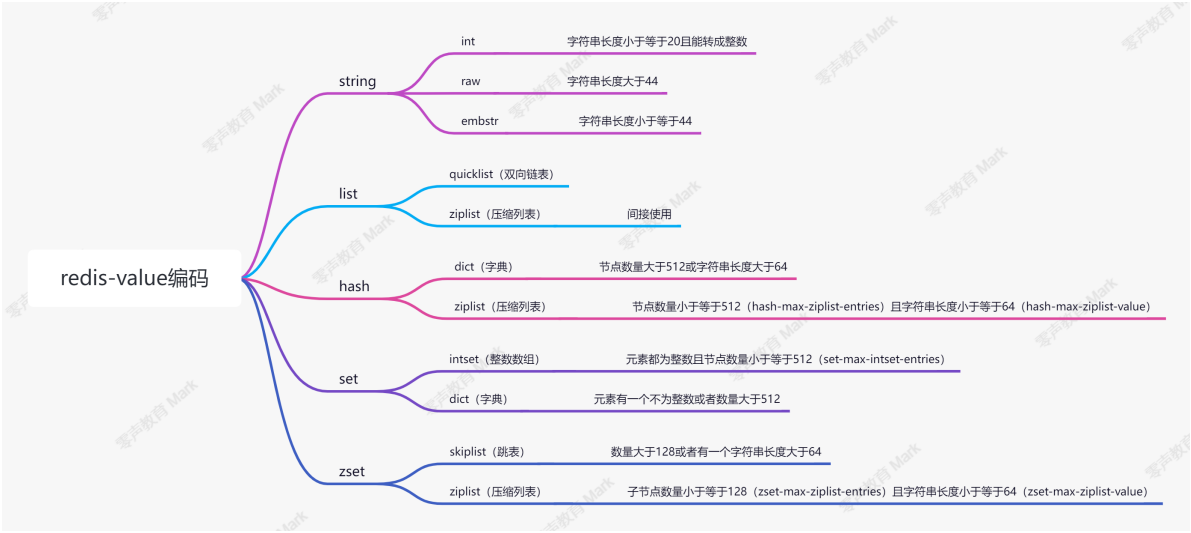
> 1. 首先定一个小的主题，预期要得到的效果；
> 2. 准备测试数据以及调试环境；
> 3. 查看流程，把每一个细支流程拷贝出来；并在旁边写上注释；
> 4. 得出结论；

redis 存储结构

存储结构



存储转换



字典实现

> *redis* 中 *KV* 组织是通过字典来实现的；*hash* 结构当节点超过 **512** 个或者单个字符串长度大于 **64** 时，*hash* 结构采用字典实现；

数据结构

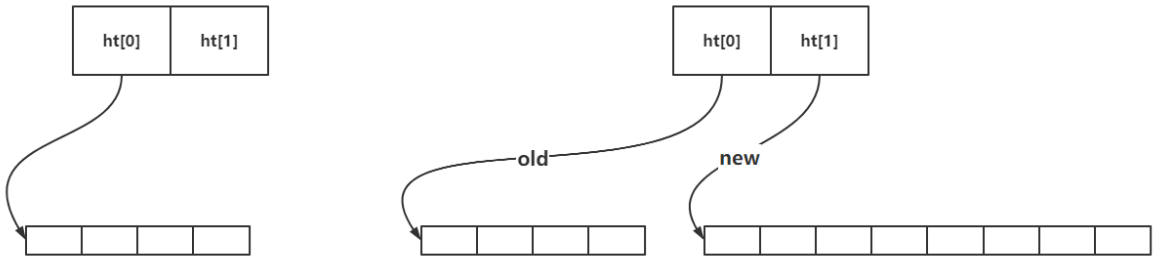
```
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;
} dictEntry;

typedef struct dictht {
    dictEntry **table;
    unsigned long size;// 数组长度
    unsigned long sizemask; //size-1
    unsigned long used;//当前数组当中包含的元素
```

```
} dictht;

typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    int16_t pauserehash; /* If >0 rehashing is paused (<0 indicates coding error) 用于安全遍历*/
} dict;
```

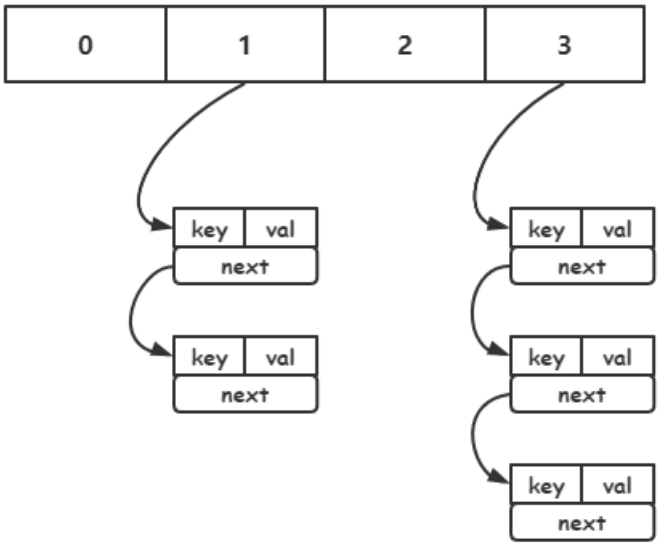
1. 字符串经过 *hash* 函数运算得到 64 位整数；
2. 相同字符串多次通过 *hash* 函数得到相同的64位整数；
3. 整数对 2^n 取余可以转化为位运算；
4. 抽屉原理 *n+1*个苹果放在 *n* 个抽屉中，苹果最多的那个抽屉至少有 2 个苹果；64位整数远大于数组的长度，比如数组长度为 4，那么 1、5、9、1+4n 都是映射到1号位数组；所以大概率会发生冲突；



冲突

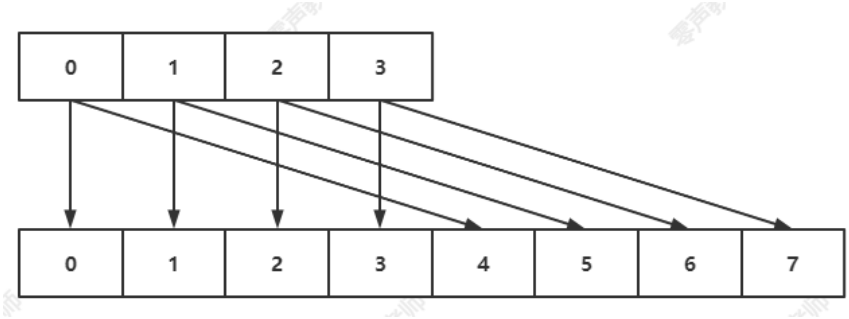
负载因子

> 负载因子 = `used` / `size`； `used` 是数组存储元素的个数，`size` 是数组的长度；
>
> 负载因子越小，冲突越小；负载因子越大，冲突越大；
>
> *redis* 的负载因子是 `1`；



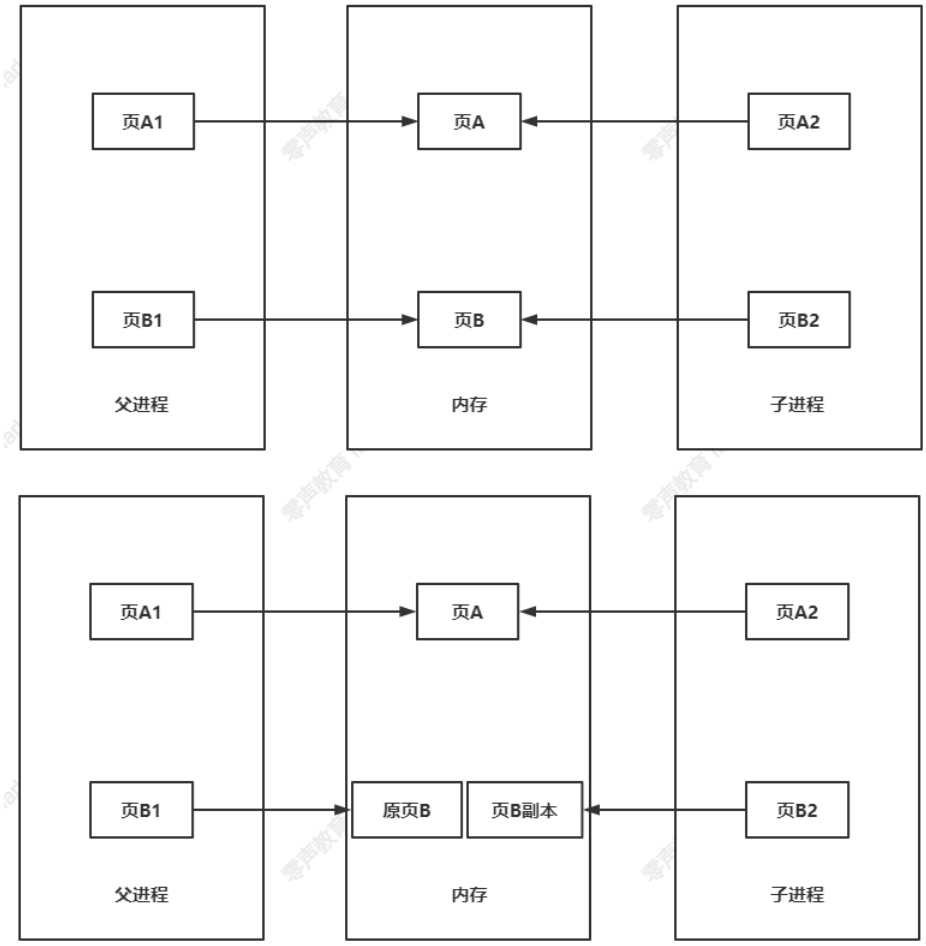
扩容

> 如果负载因子 `> 1`，则会发生扩容；扩容的规则是翻倍；
>
> 如果正在 `fork` （在 *rdb*、*aof* 复写以及 *rdb-aof* 混用情况下）时，会阻止扩容；但是此时若负载因子 `> 5`，索引效率大大降低， 则马上扩容；这里涉及到写时复制原理；



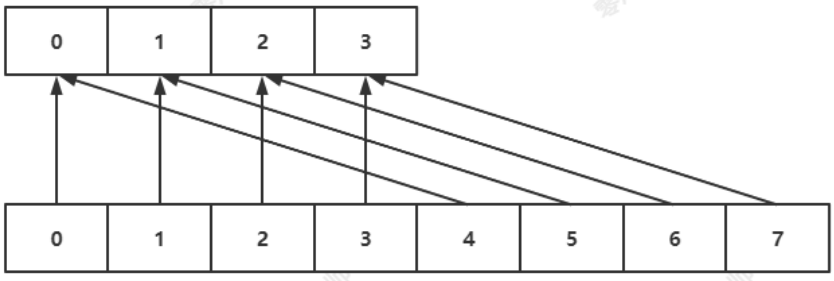
写时复制

> 写时复制核心思想：只有在不得不复制数据内容时才去复制数据内容；



缩容

> 如果负载因子 < `0.1`，则会发生缩容；缩容的规则是**恰好**包含 `used` 的 2^n；
>
> 恰好的理解：假如此时数组存储元素个数为 9，恰好包含该元素的就是 2^4，也就是 16；

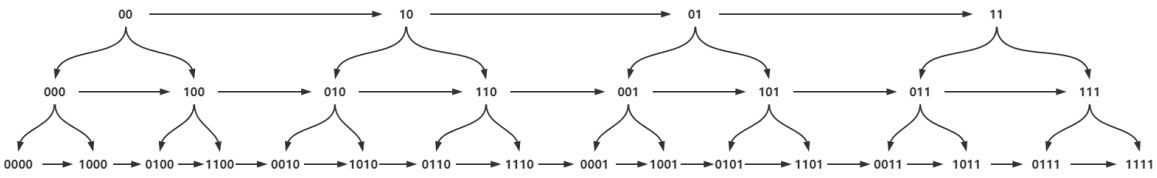


渐进式rehash

> 当 *hashtable* 中的元素过多的时候，不能一次性 *rehash* 到 `ht[1]`；这样会长期占用 *redis*，其他命令得不到响应；所以需要使用渐进式 *rehash*；
>
> **rehash步骤**：
>
> 将 `ht[0]` 中的元素重新经过 *hash* 函数生成 64 位整数，再对 `ht[1]` 长度进行取余，从而映射到 `ht[1]`；
>
> **渐进式规则**：
>
> 1. 分治的思想，将 `rehash` 分到之后的每步增删改查的操作当中；
> 2. 在定时器中，最大执行一毫秒 `rehash` ；每次步长 100 个数组槽位；
>
> **面试**：
>
> 处于渐进式 *rehash* 阶段时，是否会发生扩容缩容？不会！

scan

```
scan cursor [MATCH pattern] [COUNT count] [TYPE type]
```



> 采用高位进位加法的遍历顺序，`rehash` 后的槽位在遍历顺序上是相邻的；
>
> 遍历目标是：不重复，不遗漏；
>
> 会出现一种重复的情况：在scan过程当中，发生两次缩容的时候，会发生数据重复；

expire机制

```
# 只支持对最外层key过期;
expire key seconds
pexpire key milliseconds
ttl key
pttl key
```

惰性删除

> 分布在每一个命令操作时检查 *key* 是否过期；若过期删除 *key*，再进行命令操作；

定时删除

> 在定时器中检查库中指定个数（25）个 *key*;

```
#define ACTIVE_EXPIRE_CYCLE_KEYS_PER_LOOP 20 /* Keys for each DB loop. */
/*The default effort is 1, and the maximum configurable effort
     * is 10. */
config_keys_per_loop = ACTIVE_EXPIRE_CYCLE_KEYS_PER_LOOP +

ACTIVE_EXPIRE_CYCLE_KEYS_PER_LOOP/4*effort,
int activeExpireCycleTryExpire(redisDb *db, dictEntry *de, long long now);
```

大KEY

> 在 redis 实例中形成了很大的对象，比如一个很大的 hash 或很大的 zset，这样的对象在扩容的时候，会一次性申请更大的一块内存，这会导致卡顿；如果这个大 key 被删除，内存会一次性回收，卡顿现象会再次产生；

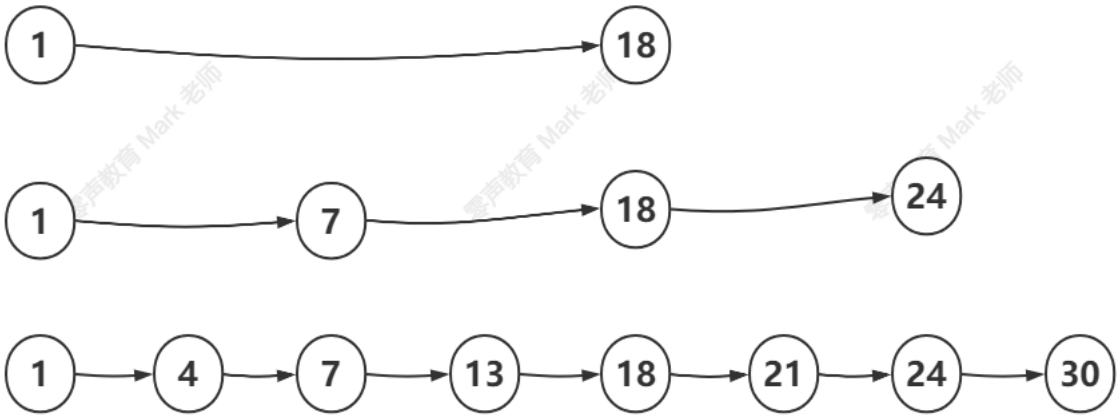
> 如果观察到 redis 的内存大起大落，极有可能因为大 key 导致的；

```
# 每隔0.1秒 执行100条scan命令
redis-cli -h 127.0.0.1 --bigkeys -i 0.1
```

跳表实现

> 跳表（多层级有序链表）结构用来实现有序集合；鉴于 *redis* 需要实现 `zrange` 以及 `zrevrange` 功能；需要节点间最好能直接相连并且增删改操作后结构依然有序；*B+* 树时间复杂度为 $h * O(log_2 n)$；鉴于 B+ 复杂的节点分裂操作；
>
> 时间复杂度：
>
> 有序数组通过二分查找能获得 $o(log_2 n)$ 时间复杂度；平衡二叉树也能获得 $o(log_2 n)$ 时间复杂度；

理想跳表

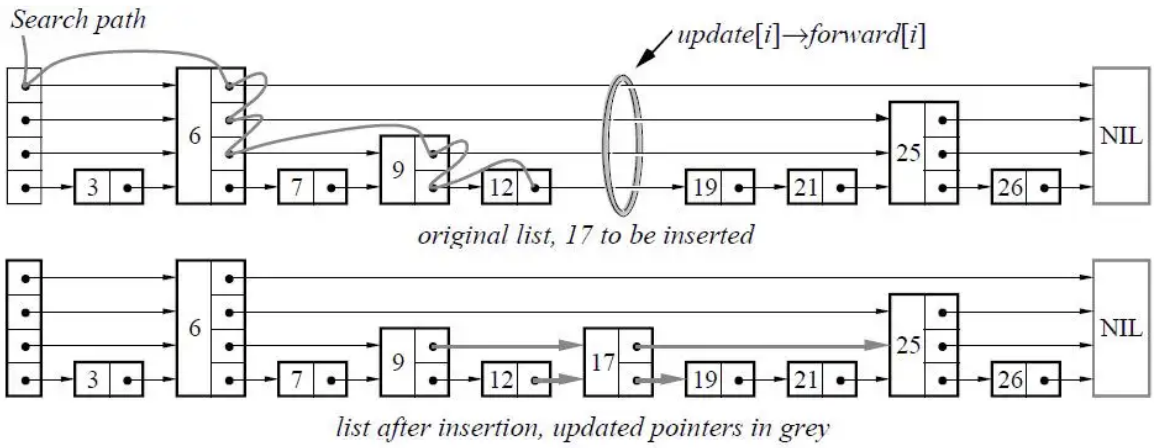


> 每隔一个节点生成一个层级节点；模拟二叉树结构，以此达到搜索时间复杂度为 $O(log_2 n)$；
>
> 空间换时间的结构；

> 但是如果对理想跳表结构进行删除增加操作，很有可能改变跳表结构；如果重构理想结构，将是巨大的运算；考虑用概率的方法来进行优化；从每一个节点出发，每**增加**一个节点都有 $\frac{1}{2}$ 的概率增加一个层级，$\frac{1}{4}$ 的概率增加两个层级，$\frac{1}{8}$ 的概率增加 3 个层级，以此类推；经过证明，当数据量足够大（256）时，通过概率构造的跳表趋向于理想跳表，并且此时如果删除节点，无需重构跳表结构，此时依然趋向于理想跳表；此时时间复杂度为 $(1-\frac{1}{n^c})*O(log_2n)$；

redis跳表

> 从节约内存出发，*redis* 考虑牺牲一点时间复杂度让跳表结构更加变扁平，就像二叉堆改成四叉堆结构；并且 *redis* 还限制了跳表的最高层级为 **32**；
>
> 节点数量大于 **128** 或者有一个字符串长度大于 **64**，则使用跳表（*skiplist*）；



数据结构

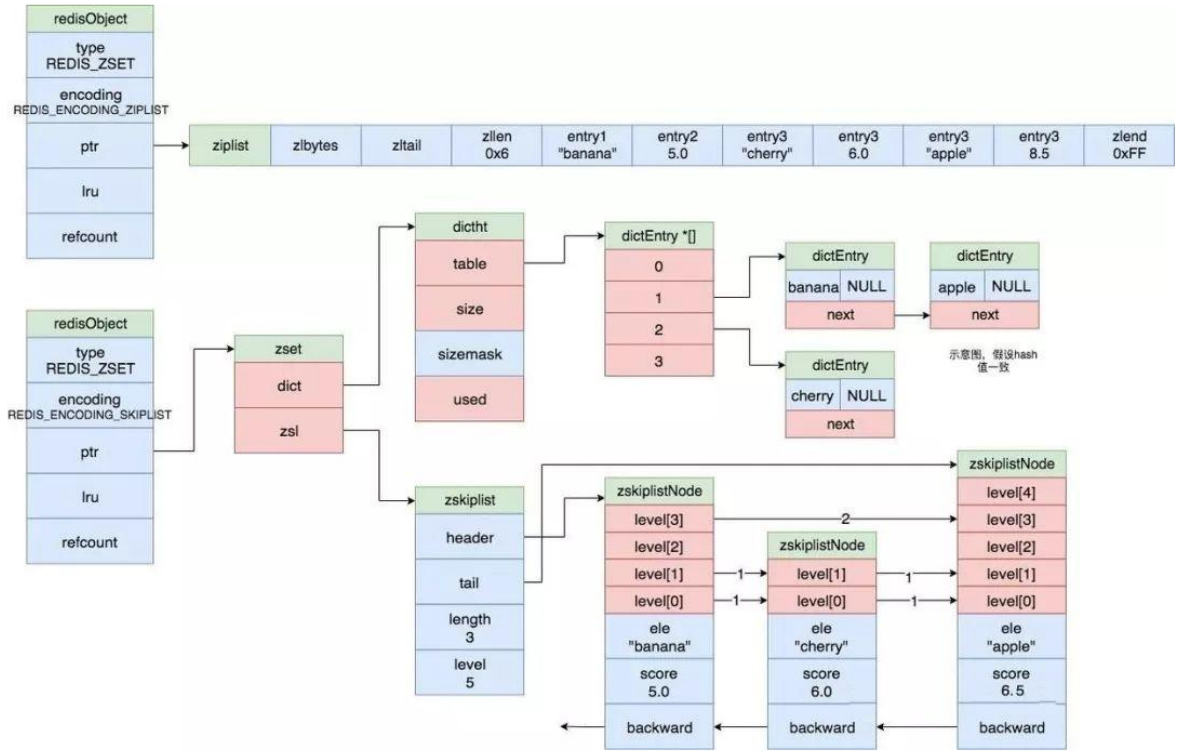
```
#define ZSKIPLIST_MAXLEVEL 32 /* Should be enough for 2^64 elements */
#define ZSKIPLIST_P 0.25      /* Skiplist P = 1/4 */
/* ZSETs use a specialized version of Skiplists */
typedef struct zskiplistNode {
```

```
    sds ele;
    double score; // WRN: score 只能是浮点数
    struct zskiplistNode *backward;
    struct zskiplistLevel {
        struct zskiplistNode *forward;
        unsigned long span; // 用于 zrank
    } level[];
} zskiplistNode;

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

typedef struct zset {
    dict *dict;     // 帮助快速索引到节点
    zskiplist *zsl;
} zset;
```

结构图



补充

redis io多线程