1. 底层数据结构, 与Redis Value Type之间的关系

对于Redis的使用者来说, Redis作为Key-Value型的内存数据库, 其Value有多种类型.

0. String
1. Hash
2. List
3. Set
4. ZSet

这些Value的类型, 只是"Redis的用户认为的, Value存储数据的方式". 而在具体实现上, 各个Type的Value到底如何存储, 这对于Redis的使用者来说是不公开的.

举个栗子: 使用下面的命令创建一个Key-Value

```
$ SET "Hello" "World"
```

对于Redis的使用者来说, `Hello` 这个Key, 对应的Value是String类型, 其值为五个ASCII字符组成的二进制数据. 但具体在底层实现上, 这五个字节是如何存储的, 是不对用户公开的. 即, Value的Type, 只是表象, 具体数据在内存中以何种数据结构存放, 这对于用户来说是不必要了解的.

Redis对使用者暴露了五种Value Type, 其底层实现的数据结构有8种, 分别是:

0. SDS - simple synamic string - 支持自动动态扩容的字节数组
1. list - 平平无奇的链表
2. dict - 使用双哈希表实现的, 支持平滑扩容的字典
3. zskiplist - 附加了后向指针的跳跃表
4. intset - 用于存储整数数值集合的自有结构
5. ziplist - 一种实现上类似于TLV, 但比TLV复杂的, 用于存储任意数据的有序序列的数据结构
6. quicklist - 一种以ziplist作为结点的双链表结构, 实现的非常苟
7. zipmap - 一种用于在小规模场合使用的轻量级字典结构

而衔接"底层数据结构"与"Value Type"的桥梁的, 则是Redis实现的另外一种数据结构: `redisObject` . Redis中的Key与Value在表层都是一个 `redisObject` 实例, 故该结构有所谓的"类型", 即是 `ValueType` . 对于每一种 `Value Type` 类型的 `redisObject` , 其底层至少支持两种不同的底层数据结构来实现. 以应对在不同的应用场景中, Redis的运行效率, 或内存占用.

2. 底层数据结构

2.1 SDS - simple dynamic string

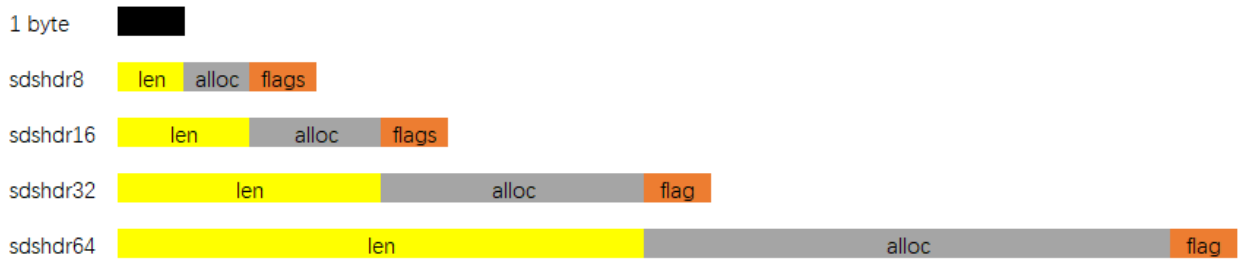
这是一种用于存储二进制数据的一种结构, 具有动态扩容的特点. 其实现位于 `src/sds.h` 与 `src/sds.c` 中, 其关键定义如下:

```
typedef char *sds;

/* 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[];
};
```

SDS的总体概览如下图:

其中 `sdshdr` 是头部, `buf` 是真实存储用户数据的地方. 另外注意, 从命名上能看出来, 这个数据结构除了能存储二进制数据, 显然是用于设计作为字符串使用的, 所以在 `buf` 中, 用户数据后总跟着一个 `\0` . 即图中 "数据" + "\0" 是为所谓的 `buf`

SDS有五种不同的头部. 其中 `sdshdr5` 实际并未使用到. 所以实际上有四种不同的头部, 分别如下:



0. `len` 分别以 `uint8` , `uint16` , `uint32` , `uint64` 表示用户数据的长度(不包括末尾的 `\0`)
1. `alloc` 分别以 `uint8` , `uint16` , `uint32` , `uint64` 表示整个SDS, 除过头部与末尾的 `\0` , 剩余的字节数.
2. `flag` 始终为一字节, 以低三位标示着头部的类型, 高5位未使用.

当在程序中持有一个SDS实例时, 直接持有的是数据区的头指针, 这样做的用意是: 通过这个指针, 向前偏一个字节, 就能取到 `flag` , 通过判断flag低三位的值, 能迅速判断: 头部的类型, 已用字节数, 总字节数, 剩余字节数. 这也是为什么 `sds` 类型即是 `char *` 指针类型别名的原因.

创建一个SDS实例有三个接口, 分别是:

```
// 创建一个不含数据的sds:
//  头部    3字节 sdshdr8
//  数据区  0字节
//  末尾    \0 占一字节
sds sdsempty(void);
// 带数据创建一个sds:
//  头部    按initlen的值, 选择最小的头部类型
//  数据区  从入参指针init处开始, 拷贝initlen个字节
//  末尾    \0 占一字节
sds sdsnewlen(const void *init, size_t initlen);
// 带数据创建一个sds:
//  头部    按strlen(init)的值, 选择最小的头部类型
//  数据区  入参指向的字符串中的所有字符, 不包括末尾 \0
//  末尾    \0 占一字节
sds sdsnew(const char *init);
```

0. 所有创建sds实例的接口, 都不会额外分配预留内存空间
1. `sdsnewlen` 用于带二进制数据创建sds实例, `sdsnew` 用于带字符串创建sds实例. 接口返回的sds可以直接传入libc中的字符串输出函数中进行操作, 由于无论其中存储的是用户的二进制数据, 还是字符串, 其末尾都带一个\0, 所以至少调用libc中的字符串输出函数是安全的.

在对SDS中的数据进行修改时, 若剩余空间不足, 会调用 `sdsMakeRoomFor` 函数用于扩容空间, 这是一个很低级的API, 通常情况下不应当由SDS的使用者直接调用. 其实现中核心的几行如下:

```
sds sdsMakeRoomFor(sds s, size_t addlen) {
    ...
    /* Return ASAP if there is enough space left. */
    if (avail >= addlen) return s;
```

```
    len = sdslen(s);
    sh = (char*)s-sdsHdrSize(oldtype);
    newlen = (len+addlen);
    if (newlen < SDS_MAX_PREALLOC)
        newlen *= 2;
    else
        newlen += SDS_MAX_PREALLOC;
    ...
}
```

可以看到, 在扩充空间时

0. 先保证至少有 `addlen` 可用
1. 然后再进一步扩充, 在总体占用空间不超过阈值 `SDS_MAC_PREALLOC` 时, 申请空间再翻一倍. 若总体空间已经超过了阈值, 则步进增长 `SDS_MAC_PREALLOC` . 这个阈值的默认值为 `1024 * 1024`

SDS也提供了接口用于移除所有未使用的内存空间. `sdsRemoveFreeSpace` , 该接口没有间接的被任何SDS其它接口调用, 即默认情况下, SDS不会自动回收预留空间. 在SDS的使用者需要节省内存时, 由使用者自行调用:

```
sds sdsRemoveFreeSpace(sds s);
```

总结:

0. SDS除了是某些Value Type的底层实现, 也被大量使用在Redis内部, 用于替代C-Style字符串. 所以默认的创建SDS实例接口, 不分配额外的预留空间. 因为多数字符串在程序运行期间是不变的. 而对于变更数据区的API, 其内部则是调用了 `sdsMakeRoomFor` , 每一次扩充空间, 都会预留大量的空间. 这样做的考量是: 如果一个SDS实例中的数据被变更了, 那么很有可能会在后续发生多次变更.
1. SDS的API内部不负责清除未使用的闲置内存空间, 因为内部API无法判断这样做的合适时机. 即便是在操作数据区的时候导致数据区占用内存减少时, 内部API也不会清除闲置内在空间. 清除闲置内存空间责任应当由SDS的使用者自行担当.
2. 用SDS替代C-Style字符串时, 由于其头部额外存储了数据区的长度信息, 所以字符串的求长操作时间复杂度为O(1)

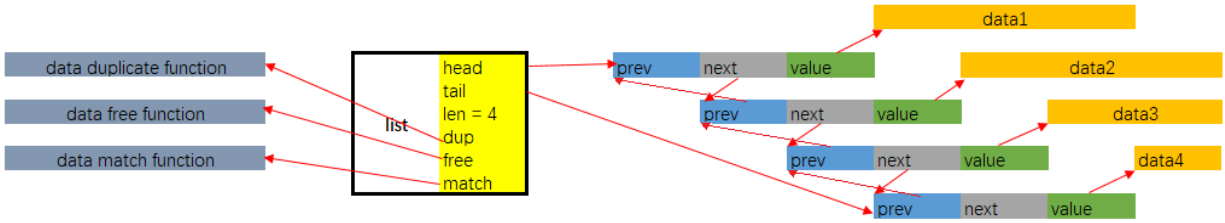
2.2 list

这是普通的链表实现, 链表结点不直接持有数据, 而是通过 `void *` 指针来间接的指向数据. 其实现位于 `src/adlist.h` 与 `src/adlist.c` 中, 关键定义如下:

```
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;
```

其内存布局如下图所示:



这是一个平平无奇的链表的实现. `list` 在Redis除了作为一些Value Type的底层实现外, 还广泛用于Redis的其它功能实现中, 作为一种数据结构工具使用. 在 `list` 的实现中, 除了基本的链表定义外, 还额外增加了:

0. 迭代器 `listIter` 的定义, 与相关接口的实现.
1. 由于 `list` 中的链表结点本身并不直接持有数据, 而是通过 `value` 字段, 以 `void *` 指针的形式间接持有, 所以数据的生命周期并不完全与链表及其结点一致. 这给了 `list` 的使用者相当大的灵活性. 比如可以多个结点持有同一份数据的地址. 但与此同时, 在对链表进行销毁, 结点复制以及查找匹配时, 就需要 `list` 的使用者将相关的函数指针赋值于 `list.dup` , `list.free` , `list.match` 字段.

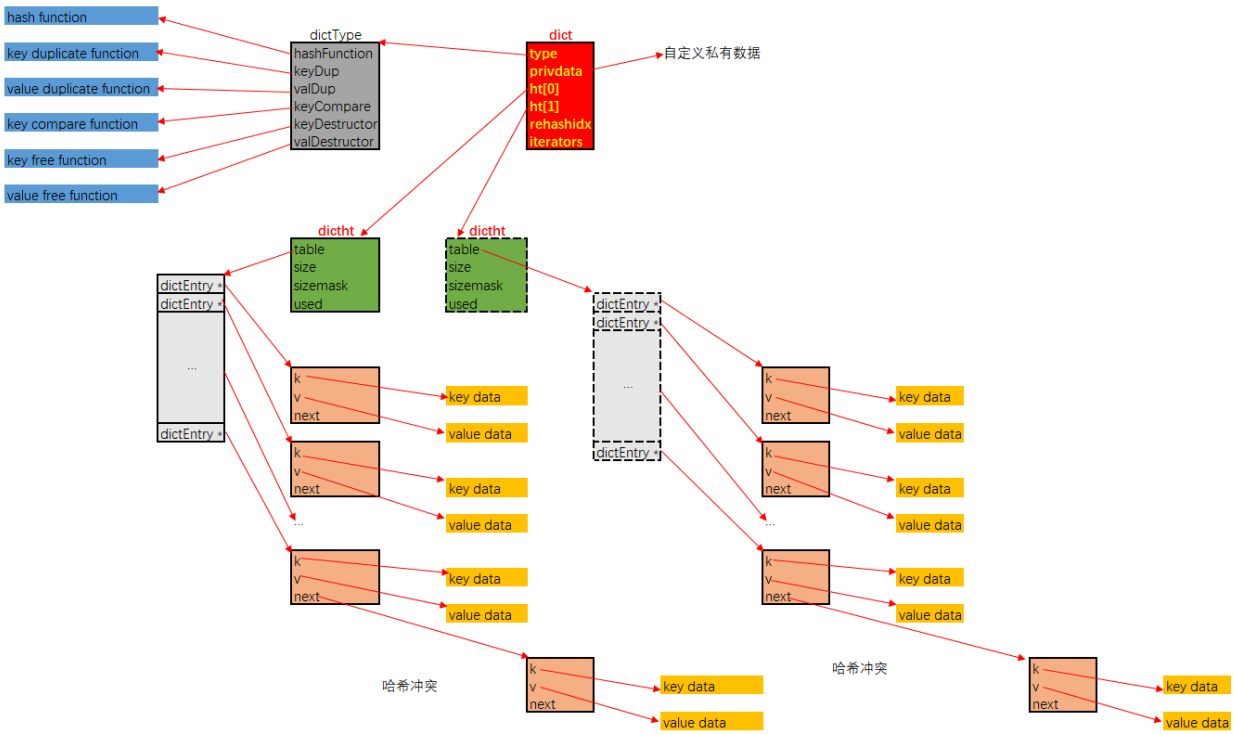
2.3 dict

`dict` 是Redis底层数据结构中实现最为复杂的一个数据结构, 其功能类似于C++标准库中的 `std::unordered_map` , 其实现位于 `src/dict.h` 与 `src/dict.c` 中, 其关键定义如下:

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

```
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;

/* 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; /* number of iterators currently running */
} dict;

/* If safe is set to 1 this is a safe iterator, that means, you can call
 * dictAdd, dictFind, and other functions against the dictionary even while
 * iterating. Otherwise it is a non safe iterator, and only dictNext()
 * should be called while iterating. */
typedef struct dictIterator {
    dict *d;
    long index;
    int table, safe;
    dictEntry *entry, *nextEntry;
    /* unsafe iterator fingerprint for misuse detection. */
    long long fingerprint;
} dictIterator;
```

其内存布局如下所示:



0. `dict` 中存储的键值对, 是通过 `dictEntry` 这个结构间接持有的, `k` 通过指针间接持有键, `v` 通过指针间接持有值. 注意, 若值是整数值的话, 是直接存储在v字段中的, 而不是间接持有. 同时 `next` 指针用于指向, 在bucket索引值冲突时, 以链式方式解决冲突, 指向同索引的下一个 `dictEntry` 结构.
1. 传统的哈希表实现, 是一块连续空间的顺序表, 表中元素即是结点. 在 `dictht.table` 中, 结点本身是散布在内存中的, 顺序表中存储的是 `dictEntry` 的指针
2. 哈希表即是 `dictht` 结构, 其通过 `table` 字段间接的持有顺序表形式的bucket, bucket的容量存储在 `size` 字段中, 为了加速将散列值转化为bucket中的数组索引, 引入了 `sizemask` 字段, 计算指定键在哈希表中的索引时, 执行的操作类似于 `dict->type->hashFunction(键) & dict->ht[x].sizemask` . 从这里也可以看出来, bucket的容量适宜于为2的幂次, 这样计算出的索引值能覆盖到所有bucket索引位.
3. `dict` 即为字典. 其中 `type` 字段中存储的是本字典使用到的各种函数指针, 包括散列函数, 键与值的复制函数, 释放函数, 以及键的比较函数. `privdata` 是用于存储用户自定义数据. 这样, 字典的使用者可以最大化的自定义字典的实现, 通过自定义各种函数实现, 以及可以附带私有数据, 保证了字典有很大的调优空间.
4. 字典为了支持平滑扩容, 定义了 `ht[2]` 这个数组字段. 其用意是这样的:
   0. 一般情况下, 字典 `dict` 仅持有一个哈希表 `dictht` 的实例, 即整个字典由一个bucket实现.
   0. 随着插入操作, bucket中出现冲突的概率会越来越大, 当字典中存储的结点数目, 与bucket数组长度的比值达到一个阈值(1:1)时, 字典为了缓解性能下降, 就需要扩容
   0. 扩容的操作是平滑的, 即在扩容时, 字典会持有两个 `dictht` 的实例, `ht[0]` 指向旧哈希表, `ht[1]` 指向扩容后的新哈希表. 平滑扩容的重点在于两个策略:
   0. 后续每一次的插入, 替换, 查找操作, 都插入到 `ht[1]` 指向的哈希表中
   0. 每一次插入, 替换, 查找操作执行时, 会将旧表 `ht[0]` 中的一个bucket索引位持有的结点链表, 迁移到 `ht[1]` 中去. 迁移的进度保存在 `rehashidx` 这个字段中.在旧表中由于冲突而被链接在同一索引位上的结点, 迁移到新表后, 可能会散布在多个新表索引中去.

0. 当迁移完成后， `ht[0]` 指向的旧表会被释放，之后会将新表的持有权转交给 `ht[0]` ，再重置 `ht[1]` 指向 `NULL`

5. 这种平滑扩容的优点有两个：

0. 平滑扩容过程中，所有结点的实际数据，即 `dict->ht[0]->table[rehashindex]->k` 与 `dict->ht[0]->table[rehashindex]->v` 分别指向的实际数据，内存地址都不会变化. 没有发生键数据与值数据的拷贝或移动，扩容整个过程仅是各种指针的操作. 速度非常快

0. 扩容操作是步进式的，这保证任何一次插入操作都是顺畅的， `dict` 的使用者是无感知的. 若扩容是一次性的，当新旧bucket容量特别大时，迁移所有结点必然会导致耗时陡增.

除了字典本身的实现外，其中还顺带实现了一个迭代器，这个迭代器中有字段 `safe` 以标示该迭代器是"安全迭代器"还是"非安全迭代器"，所谓的安全与否，指是的这种场景：
设想在运行迭代器的过程中，字典正处于平滑扩容的过程中. 在平滑扩容的过程中时，旧表一个索引位上的，由冲突而链起来的多个结点，迁移到新表后，可能会散布到新表的多个索引位上. 且新的索引位的值可能比旧的索引位要低.

遍历操作的重点是，保证在迭代器遍历操作开始时，字典中持有的所有结点，都会被遍历到. 而若在遍历过程中，一个未遍历的结点，从旧表迁移到新表后，索引值减小了，那么就可能会导致这个结点在遍历过程中被遗漏.

所以，所谓的"安全"迭代器，其在内部实现时：在迭代过程中，若字典正处于平滑扩容过程，则暂停结点迁移，直至迭代器运行结束. 这样虽然不能保证在迭代过程中插入的结点会被遍历到，但至少保证在迭代起始时，字典中持有的所有结点都会被遍历到.

这也是为什么 `dict` 结构中有一个 `iterators` 字段的原因：该字段记录了运行于该字典上的安全迭代器的数目. 若该数目不为0，字典是不会继续进行结点迁移平滑扩容的.

下面是字典的扩容操作中的核心代码，我们以插入操作引起的扩容为例：

先是插入操作的外部逻辑：
0. 如果插入时，字典正处于平滑扩容过程中，那么无论本次插入是否成功，先迁移一个bucket索引中的结点至新表
0. 在计算新插入结点键的bucket索引值时，内部会探测哈希表是否需要扩容(若当前不在平滑扩容过程中)

```
int dictAdd(dict *d, void *key, void *val)
{
    dictEntry *entry = dictAddRaw(d,key,NULL);          // 调用dictAddRaw

    if (!entry) return DICT_ERR;
    dictSetVal(d, entry, val);
    return DICT_OK;
}

dictEntry *dictAddRaw(dict *d, void *key, dictEntry **existing)
{
    long index;
    dictEntry *entry;
    dictht *ht;
```

```
    if (dictIsRehashing(d)) _dictRehashStep(d); // 若在平滑扩容过程中，先步进迁移一个bucket索引

    /* Get the index of the new element, or -1 if
     * the element already exists. */

    // 在计算键在bucket中的索引值时，内部会检查是否需要扩容
    if ((index = _dictKeyIndex(d, key, dictHashKey(d,key), existing)) == -1)
        return NULL;

    /* Allocate the memory and store the new entry.
     * Insert the element in top, with the assumption that in a database
     * system it is more likely that recently added entries are accessed
     * more frequently. */
    ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
    entry = zmalloc(sizeof(*entry));
    entry->next = ht->table[index];
    ht->table[index] = entry;
    ht->used++;

    /* Set the hash entry fields. */
    dictSetKey(d, entry, key);
    return entry;
}
```

下面是计算bucket索引值的函数，内部会探测该哈希表是否需要扩容，如果需要扩容(结点数目与bucket数组长度比例达到1:1)，就使字典进入平滑扩容过程：

```
static long _dictKeyIndex(dict *d, const void *key, uint64_t hash, dictEntry **existing)
{
    unsigned long idx, table;
    dictEntry *he;
    if (existing) *existing = NULL;

    /* Expand the hash table if needed */
    if (_dictExpandIfNeeded(d) == DICT_ERR) // 探测是否需要扩容，如果需要，则开始扩容
        return -1;
    for (table = 0; table <= 1; table++) {
        idx = hash & d->ht[table].sizemask;
        /* Search if this slot does not already contain the given key */
        he = d->ht[table].table[idx];
        while(he) {
            if (key==he->key || dictCompareKeys(d, key, he->key)) {
                if (existing) *existing = he;
                return -1;
            }
            he = he->next;
        }
```

```
        if (!dictIsRehashing(d)) break;
    }
    return idx;
}

/* Expand the hash table if needed */
static int _dictExpandIfNeeded(dict *d)
{
    /* Incremental rehashing already in progress. Return. */
    if (dictIsRehashing(d)) return DICT_OK; // 如果正在扩容过程中，则什么也不做

    /* If the hash table is empty expand it to the initial size. */
    // 若字典中本无元素，则初始化字典，初始化时的bucket数组长度为4
    if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_SIZE);

    /* If we reached the 1:1 ratio, and we are allowed to resize the hash
     * table (global setting) or we should avoid it but the ratio between
     * elements/buckets is over the "safe" threshold, we resize doubling
     * the number of buckets. */
    // 若字典中元素的个数与bucket数组长度比值大于1:1时，则调用dictExpand进入平滑扩容状态
    if (d->ht[0].used >= d->ht[0].size &&
        (dict_can_resize ||
         d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
    {
        return dictExpand(d, d->ht[0].used*2);
    }
    return DICT_OK;
}

int dictExpand(dict *d, unsigned long size)
{
    dictht n; /* the new hash table */  // 新建一个dictht结构
    unsigned long realsize = _dictNextPower(size);

    /* the size is invalid if it is smaller than the number of
     * elements already inside the hash table */
    if (dictIsRehashing(d) || d->ht[0].used > size)
        return DICT_ERR;

    /* Rehashing to the same table size is not useful. */
    if (realsize == d->ht[0].size) return DICT_ERR;

    /* Allocate the new hash table and initialize all pointers to NULL */
    n.size = realsize;
    n.sizemask = realsize-1;
    n.table = zcalloc(realsize*sizeof(dictEntry*));// 初始化dictht下的table，即bucket
数组
    n.used = 0;

    /* Is this the first initialization? If so it's not really a rehashing
     * we just set the first hash table so that it can accept keys. */
```

```
    // 若是新字典初始化，直接把dictht结构挂在ht[0]中
    if (d->ht[0].table == NULL) {
        d->ht[0] = n;
        return DICT_OK;
    }

    // 否则，把新dictht结构挂在ht[1]中，并开启平滑扩容(置rehashidx为0，字典处于非扩容状态时，该字段值为-1)
    /* Prepare a second hash table for incremental rehashing */
    d->ht[1] = n;
    d->rehashidx = 0;
    return DICT_OK;
}
```

下面是平滑扩容的实现：

```
static void _dictRehashStep(dict *d) {
    // 若字典上还运行着安全迭代器，则不迁移结点
    // 否则每次迁移一个旧bucket索引上的所有结点
    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);
        // 在旧bucket中，找到下一个非空的索引位
        while(d->ht[0].table[d->rehashidx] == NULL) {
            d->rehashidx++;
            if (--empty_visits == 0) return 1;
        }
        // 取出该索引位上的结点链表
        de = d->ht[0].table[d->rehashidx];
        /* Move all the keys in this bucket from the old to the new hash HT */
        // 把所有结点迁移到新bucket中去
        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... */
    // 检查是否旧表中的所有结点都被迁移到了新表
    // 如果是，则置先释放原旧bucket数组，再置ht[1]为ht[0]
    // 最后再置rehashidx=-1，以示字典不处于平滑扩容状态
    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;
}
```

总结:

0. 字典的实现很复杂, 主要是实现了平滑扩容逻辑
1. 用户数据均是以指针形式间接由 `dictEntry` 结构持有, 故在平滑扩容过程中, 不涉及用户数据的拷贝
2. 有安全迭代器可用, 安全迭代器保证, 在迭代起始时, 字典中的所有结点, 都会被迭代到, 即使在迭代过程中对字典有插入操作
3. 字典内部使用的默认散列函数其实也非常有讲究, 不过限于篇幅, 这里不展开讲. 并且字典的实现给了使用者非常大的灵活性(`dictType` 结构与 `dict.privdata` 字段), 对于一些特定场合使用的键数据, 用户可以自行选择更高效更特定化的散列函数

2.4 zskiplist

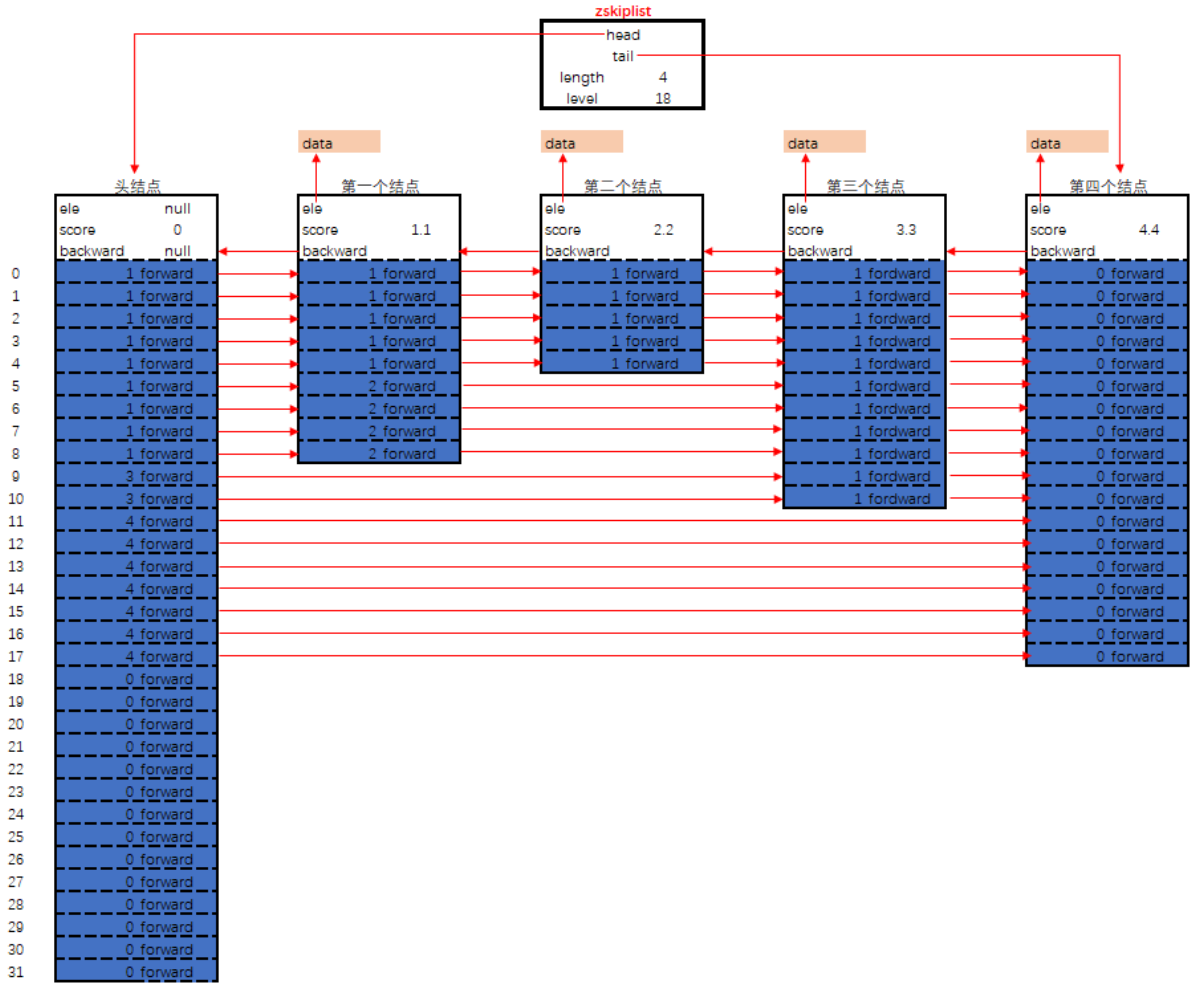
`zskiplist` 是Redis实现的一种特殊的跳跃表. 跳跃表是一种基于线性表实现简单的搜索结构, 其最大的特点就是: 实现简单, 性能能逼近各种搜索树结构. 血统纯正的跳跃表的介绍在维基百科中即可查阅. 在Redis中, 在原版跳跃表的基础上, 进行了一些小改动, 即是现在要介绍的 `zskiplis` t结构.

其定义在 `src/server.h` 中, 如下:

```
/* ZSETs use a specialized version of Skiplists */
typedef struct zskiplistNode {
    sds ele;
    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;
```

其内存布局如下图:



`zskiplist` 的核心设计要点为:

0. 头结点不持有任何数据, 且其 `level[]` 的长度为32

0. 每个结点, 除了持有数据的 `ele` 字段, 还有一个字段 `score` , 其标示着结点的得分, 结点之间凭借得分来判断先后顺序, 跳跃表中的结点按结点的得分升序排列.

0. 每个结点持有一个 `backward` 指针, 这是原版跳跃表中所没有的. 该指针指向结点的前一个紧邻结点.

0. 每个结点中最多持有32个 `zskiplistLevel` 结构. 实际数量在结点创建时, 按幂次定律随机生成(不超过32). 每个 `zskiplistLevel` 中有两个字段.

0. `forward` 字段指向比自己得分高的某个结点(不一定是紧邻的), 并且, 若当前 `zskiplistLevel` 实例在 `level[]` 中的索引为 `X` , 则其 `forward` 字段指向的结点, 其 `level[]` 字段的容量至少是

`X+1` . 这也是上图中, 为什么 `forward` 指针总是画的水平的原因.

0. `span` 字段代表 `forward` 字段指向的结点, 距离当前结点的距离. 紧邻的两个结点之间的距离定义为1.

0. `zskiplist` 中持有字段 `level` , 用以记录所有结点(除过头结点外), `level[]` 数组最长的长度.

跳跃表主要用于, 在给定一个分值的情况下, 查找与该分值最接近的结点. 搜索时, 伪代码如下:

```
int level = zskiplist->level - 1;
zskiplistNode p = zskiplist->head;

while(1 && p)
{
    zskiplistNode q = (p->level)[level]->forward:
    if(q->score > 分值)
    {
        if(level > 0)
        {
            level--;
        }
        else
        {
            return :
                q为整个跳跃表中, 分值大于指定分值的第一个结点
                q->backward为整个跳跃表中, 分值小于或等于指定分值的最后一个结点
        }
    }
    else
    {
        p = q;
    }
}
```

跳跃表的实现比较简单, 最复杂的操作即是插入与删除结点, 需要仔细处理邻近结点的所有 `level[]` 中的所有 `zskiplistLevel` 结点中的 `forward` 与 `span` 的值的变更.

另外, 关于新创建的结点, 其 `level[]` 数组长度的随机算法, 在接口 `zslInsert` 的实现中, 核心代码片断如下:

```
zskiplistNode *zslInsert(zskiplist *zsl, double score, sds ele) {
    //...

    level = zslRandomLevel();    // 随机生成新结点的, level[]数组的长度
        if (level > zsl->level) {
        // 若生成的新结点的level[]数组的长度比当前表中所有结点的level[]的长度都大
        // 那么头结点中需要新增几个指向该结点的指针
        // 并刷新ziplist中的level字段
        for (i = zsl->level; i < level; i++) {
            rank[i] = 0;
            update[i] = zsl->header;
            update[i]->level[i].span = zsl->length;
        }
        zsl->level = level;
```

```
    }
    x = zslCreateNode(level,score,ele); // 创建新结点
    //... 执行插入操作
}

// 按幂次定律生成小于32的随机数的函数
// 宏 ZSKIPLIST_MAXLEVEL 的定义为32, 宏 ZSKIPLIST_P 被设定为 0.25
// 即
//      level == 1的概率为 75%
//      level == 2的概率为 75% * 25%
//      level == 3的概率为 75% * 25% * 25%
//      ...
//      level == 31的概率为 0.75 * 0.25^30
//      而
//      level == 32的概率为 0.75 * sum(i = 31 ~ +INF){ 0.25^i }
int zslRandomLevel(void) {
    int level = 1;
    while ((random()&0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
        level += 1;
    return (level<ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
}
```

2.5 intset

这是一个用于存储在序的整数的数据结构, 也底层数据结构中最简单的一个, 其定义与实现在 `src/intest.h` 与 `src/intset.c` 中, 关键定义如下:

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

#define INTSET_ENC_INT16 (sizeof(int16_t))
#define INTSET_ENC_INT32 (sizeof(int32_t))
#define INTSET_ENC_INT64 (sizeof(int64_t))
```

`inset` 结构中的 `encoding` 的取值有三个, 分别是宏 `INTSET_ENC_INT16` , `INTSET_ENC_INT32` , `INTSET_ENC_INT64` . `length` 代表其中存储的整数的个数, `contents` 指向实际存储数值的连续内存区域. 其内存布局如下图所示:

0. `intset` 中各字段, 包括 `contents` 中存储的数值, 都是以主机序(小端字节序)存储的. 这意味着Redis若运行在PPC这样的大端字节序的机器上时, 存取数据都会有额外的字节序转换开销

1. 当 `encoding == INTSET_ENC_INT16` 时， `contents` 中以 `int16_t` 的形式存储着数值. 类似的, 当 `encoding == INTSET_ENC_INT32` 时， `contents` 中以 `int32_t` 的形式存储着数值.
2. 但凡有一个数值元素的值超过了 `int32_t` 的取值范围, 整个 `intset` 都要进行升级, 即所有的数值都需要以 `int64_t` 的形式存储. 显然升级的开销是很大的.
3. `intset` 中的数值是以升序排列存储的, 插入与删除的复杂度均为O(n). 查找使用二分法, 复杂度为O(log_2(n))
4. `intset` 的代码实现中, 不预留空间, 即每一次插入操作都会调用 `zrealloc` 接口重新分配内存. 每一次删除也会调用 `zrealloc` 接口缩减占用的内存. 省是省了, 但内存操作的时间开销上升了.
5. `intset` 的编码方式一经升级, 不会再降级.

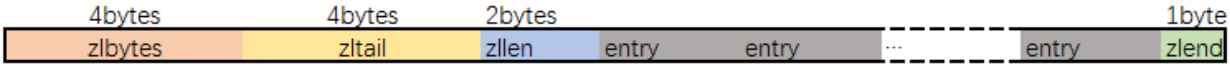
总之， `intset` 适合于如下数据的存储:

0. 所有数据都位于一个稳定的取值范围中. 比如均位于 `int16_t` 或 `int32_t` 的取值范围中
1. 数据稳定, 插入删除操作不频繁. 能接受O(lgn)级别的查找开销

2.6 ziplist

`ziplist` 是Redis底层数据结构中, 最苟的一个结构. 它的设计宗旨就是: 省内存, 从牙缝里省内存. 设计思路和TLV一致, 但为了从牙缝里节省内存, 做了很多额外工作.

`ziplist` 的内存布局与 `intset` 一样: 就是一块连续的内存空间. 但区域划分比较复杂, 概览如下图:



0. 和 `intset` 一样， `ziplist` 中的所有值都是以小端序存储的
1. `zlbytes` 字段的类型是 `uint32_t` , 这个字段中存储的是整个 `ziplist` 所占用的内存的字节数
2. `zltail` 字段的类型是 `uint32_t` , 它指的是 `ziplist` 中最后一个 `entry` 的偏移量. 用于快速定位最后一个 `entry` , 以快速完成 `pop` 等操作
3. `zllen` 字段的类型是 `uint16_t` , 它指的是整个 `ziplit` 中 `entry` 的数量. 这个值只占16位, 所以蛋疼的地方就来了: 如果 `ziplist` 中 `entry` 的数目小于65535, 那么该字段中存储的就是实际 `entry` 的值. 若等于或超过65535, 那么该字段的值固定为65535, 但实际数量需要一个个 `entry` 的去遍历所有 `entry` 才能得到.
4. `zlend` 是一个终止字节, 其值为全F, 即 `0xff` . `ziplist` 保证任何情况下, 一个 `entry` 的首字节都不会是 `255`

在画图展示 `entry` 的内存布局之前, 先讲一下 `entry` 中都存储了哪些信息:

0. 每个 `entry` 中存储了它前一个 `entry` 所占用的字节数. 这样支持 `ziplist` 反向遍历.
1. 每个 `entry` 用单独的一块区域, 存储着当前结点的类型: 所谓的类型, 包括当前结点存储的数据是什么(二进制, 还是数值), 如何编码(如果是数值, 数值如何存储, 如果是二进制数据, 二进制数据的长度)
2. 最后就是真实的数据了

`entry` 的内存布局如下所示:

`prevlen` 即是"前一个entry所占用的字节数", 它本身是一个变长字段, 规约如下:

0. 若前一个 `entry` 占用的字节数小于 254, 则 `prevlen` 字段占一字节
1. 若前一个 `entry` 占用的字节数等于或大于 254, 则 `prevlen` 字段占五字节: 第一个字节值为 254, 即 `0xfe` , 另外四个字节, 以 `uint32_t` 存储着值.

`encoding` 字段的规约就复杂了许多

0. 若数据是二进制数据, 且二进制数据长度小于64字节(不包括64), 那么 `encoding` 占一字节. 在这一字节中, 高两位值固定为0, 低六位值以无符号整数的形式存储着二进制数据的长度. 即 `00xxxxxx` , 其中低六位bit `xxxxxx` 是用二进制保存的数据长度.
1. 若数据是二进制数据, 且二进制数据长度大于或等于64字节, 但小于16384(不包括16384)字节, 那么 `encoding` 占用两个字节. 在这两个字节16位中, 第一个字节的高两位固定为 `01` , 剩余的14个位, 以小端序无符号整数的形式存储着二进制数据的长度, 即 `01xxxxxx, yyyyyyyy` , 其中 `yyyyyyyy` 是高八位, `xxxxxx` 是低六位.
2. 若数据是二进制数据, 且二进制数据的长度大于或等于16384字节, 但小于2^32-1字节, 则 `encoding` 占用五个字节. 第一个字节是固定值 `10000000` , 剩余四个字节, 按小端序 `uint32_t` 的形式存储着二进制数据的长度. 这也是 `ziplist` 能存储的二进制数据的最大长度, 超过 `2^32-1` 字节的二进制数据, `ziplist` 无法存储.
3. 若数据是整数值, 则 `encoding` 和 `data` 的规约如下:
    0. 首先, 所有存储数值的 `entry` , 其 `encoding` 都仅占用一个字节. 并且最高两位均是 `11`
    0. 若数值取值范围位于 `[0, 12]` 中, 则 `encoding` 和 `data` 挤在同一个字节中. 即为 `1111 0001` ~ `1111 1101` , 高四位是固定值, 低四位的值从 `0001` 至 `1101` , 分别代表 0 ~ 12这十五个数值
    0. 若数值取值范围位于 `[-128, -1] [13, 127]` 中, 则 `encoding == 0b 1111 1110` . 数值存储在紧邻的下一个字节, 以 `int8_t` 形式编码
    0. 若数值取值范围位于 `[-32768, -129] [128, 32767]` 中, 则 `encoding == 0b 1100 0000` . 数值存储在紧邻的后两个字节中, 以小端序 `int16_t` 形式编码
    0. 若数值取值范围位于 `[-8388608, -32769] [32768, 8388607]` 中, 则 `encoding == 0b 1111 0000` . 数值存储在紧邻的后三个字节中, 以小端序存储, 占用三个字节.
    0. 若数值取值范围位于 `[-2^31, -8388609] [8388608, 2^31 - 1]` 中, 则 `encoding == 0b 1101 0000.` 数值存储在紧邻的后四个字节中, 以小端序 `int32_t` 形式编码
    0. 若数值取值均不在上述范围, 但位于 `int64_t` 所能表达的范围内, 则 `encoding == 0b 1110 0000` , 数值存储在紧邻的后八个字节中, 以小端序 `int64_t` 形式编码

在大规模数值存储中, `ziplist` 几乎不浪费内存空间, 其苟的程序到达了字节级别, 甚至对于 `[0, 12]` 区间的数值, 连 `data` 里的那一个字节也要省下来. 显然, `ziplist` 是一种特别节省内存的数据结构, 但它的缺点也十分明显:

0. 和 `intset` 一样, `ziplist` 也不预留内存空间, 并且在移除结点后, 也是立即缩容, 这代表每次写操作都会进行内存分配操作.

1. `ziplist` 最蛋疼的一个问题是: 结点如果扩容, 导致结点占用的内存增长, 并且超过254字节的话, 可能会导致链式反应: 其后一个结点的 `entry.prevlen` 需要从一字节扩容至五字节. 最坏情况下, 第一个结点的扩容, 会导致整个 `ziplist` 表中的后续所有结点的 `entry.prevlen` 字段扩容. 虽然这个内存重分配的操作依然只会发生一次, 但代码中的时间复杂度是o(N)级别, 因为链式扩容只能一步一步的计算. 但这种情况的概率十分的小, 一般情况下链式扩容能连锁反映五六次就很不幸了. 之所以说这是一个蛋疼问题, 是因为, 这样的坏场景下, 其实时间复杂度并不高: 依次计算每个 `entry` 新的空间占用, 也就是o(N), 总体占用计算出来后, 只执行一次内存重分配, 与对应的 `memmove` 操作, 就可以了. 蛋疼说的是: 代码特别难写, 难读. 下面放一段处理插入结点时处理链式反应的代码片断, 大家自行感受一下:

```
unsigned char *__ziplistInsert(unsigned char *zl, unsigned char *p, unsigned char
*s, unsigned int slen) {
    size_t curlen = intrev32ifbe(ZIPLIST_BYTES(zl)), reqlen;
    unsigned int prevlensize, prevlen = 0;
    size_t offset;
    int nextdiff = 0;
    unsigned char encoding = 0;
    long long value = 123456789; /* initialized to avoid warning. Using a value
                                    that is easy to see if for some reason
                                    we use it uninitialized. */
    zlentry tail;

    /* Find out prevlen for the entry that is inserted. */
    if (p[0] != ZIP_END) {
        ZIP_DECODE_PREVLEN(p, prevlensize, prevlen);
    } else {
        unsigned char *ptail = ZIPLIST_ENTRY_TAIL(zl);
        if (ptail[0] != ZIP_END) {
            prevlen = zipRawEntryLength(ptail);
        }
    }

    /* See if the entry can be encoded */
    if (zipTryEncoding(s,slen,&value,&encoding)) {
        /* 'encoding' is set to the appropriate integer encoding */
        reqlen = zipIntSize(encoding);
    } else {
        /* 'encoding' is untouched, however zipStoreEntryEncoding will use the
         * string length to figure out how to encode it. */
        reqlen = slen;
    }
    /* We need space for both the length of the previous entry and
     * the length of the payload. */
    reqlen += zipStorePrevEntryLength(NULL,prevlen);
    reqlen += zipStoreEntryEncoding(NULL,encoding,slen);

    /* When the insert position is not equal to the tail, we need to
     * make sure that the next entry can hold this entry's length in
```

```
     * its prevlen field. */
    int forcelarge = 0;
    nextdiff = (p[0] != ZIP_END) ? zipPrevLenByteDiff(p,reqlen) : 0;
    if (nextdiff == -4 && reqlen < 4) {
        nextdiff = 0;
        forcelarge = 1;
    }

    /* Store offset because a realloc may change the address of zl. */
    offset = p-zl;
    zl = ziplistResize(zl,curlen+reqlen+nextdiff);
    p = zl+offset;

    /* Apply memory move when necessary and update tail offset. */
    if (p[0] != ZIP_END) {
        /* Subtract one because of the ZIP_END bytes */
        memmove(p+reqlen,p-nextdiff,curlen-offset-1+nextdiff);

        /* Encode this entry's raw length in the next entry. */
        if (forcelarge)
            zipStorePrevEntryLengthLarge(p+reqlen,reqlen);
        else
            zipStorePrevEntryLength(p+reqlen,reqlen);

        /* Update offset for tail */
        ZIPLIST_TAIL_OFFSET(zl) =
            intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+reqlen);

        /* When the tail contains more than one entry, we need to take
         * "nextdiff" in account as well. Otherwise, a change in the
         * size of prevlen doesn't have an effect on the *tail* offset. */
        zipEntry(p+reqlen, &tail);
        if (p[reqlen+tail.headersize+tail.len] != ZIP_END) {
            ZIPLIST_TAIL_OFFSET(zl) =
                intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+nextdiff);
        }
    } else {
        /* This element will be the new tail. */
        ZIPLIST_TAIL_OFFSET(zl) = intrev32ifbe(p-zl);
    }

    /* When nextdiff != 0, the raw length of the next entry has changed, so
     * we need to cascade the update throughout the ziplist */
    if (nextdiff != 0) {
        offset = p-zl;
        zl = __ziplistCascadeUpdate(zl,p+reqlen);
        p = zl+offset;
    }

    /* Write the entry */
    p += zipStorePrevEntryLength(p,prevlen);
    p += zipStoreEntryEncoding(p,encoding,slen);
```

```
    if (ZIP_IS_STR(encoding)) {
        memcpy(p,s,slen);
    } else {
        zipSaveInteger(p,value,encoding);
    }
    ZIPLIST_INCR_LENGTH(zl,1);
    return zl;
}

unsigned char *__ziplistCascadeUpdate(unsigned char *zl, unsigned char *p) {
    size_t curlen = intrev32ifbe(ZIPLIST_BYTES(zl)), rawlen, rawlensize;
    size_t offset, noffset, extra;
    unsigned char *np;
    zlentry cur, next;

    while (p[0] != ZIP_END) {
        zipEntry(p, &cur);
        rawlen = cur.headersize + cur.len;
        rawlensize = zipStorePrevEntryLength(NULL,rawlen);

        /* Abort if there is no next entry. */
        if (p[rawlen] == ZIP_END) break;
        zipEntry(p+rawlen, &next);

        /* Abort when "prevlen" has not changed. */
        if (next.prevrawlen == rawlen) break;

        if (next.prevrawlensize < rawlensize) {
            /* The "prevlen" field of "next" needs more bytes to hold
             * the raw length of "cur". */
            offset = p-zl;
            extra = rawlensize-next.prevrawlensize;
            zl = ziplistResize(zl,curlen+extra);
            p = zl+offset;

            /* Current pointer and offset for next element. */
            np = p+rawlen;
            noffset = np-zl;

            /* Update tail offset when next element is not the tail element. */
            if ((zl+intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))) != np) {
                ZIPLIST_TAIL_OFFSET(zl) =
                    intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+extra);
            }

            /* Move the tail to the back. */
            memmove(np+rawlensize,
                np+next.prevrawlensize,
                curlen-noffset-next.prevrawlensize-1);
            zipStorePrevEntryLength(np,rawlen);

            /* Advance the cursor */
            p += rawlen;
```

```
            curlen += extra;
        } else {
            if (next.prevrawlensize > rawlensize) {
                /* This would result in shrinking, which we want to avoid.
                 * So, set "rawlen" in the available bytes. */
                zipStorePrevEntryLengthLarge(p+rawlen,rawlen);
            } else {
                zipStorePrevEntryLength(p+rawlen,rawlen);
            }

            /* Stop here, as the raw length of "next" has not changed. */
            break;
        }
    }
    return zl;
}
```

这种代码的特点就是: 最好由作者去维护, 最好一次性写对. 因为读起来真的费劲, 改起来也很费劲.

2.7 quicklist

如果说 `ziplist` 是整个Redis中为了节省内存, 而写的最苟的数据结构, 那么称 `quicklist` 就是在最苟的基础上, 再苟了一层. 这个结构是Redis在3.2版本后新加的, 在3.2版本之前, 我们可以讲, `dict` 是最复杂的底层数据结构, `ziplist` 是最苟的底层数据结构. 在3.2版本之后, 这两个记录被双双刷新了.

这是一种, 以 `ziplist` 为结点的, 双端链表结构. 宏观上, `quicklist` 是一个链表, 微观上, 链表中的每个结点都是一个 `ziplist` .

它的定义与实现分别在 `src/quicklist.h` 与 `src/quicklist.c` 中, 其中关键定义如下:

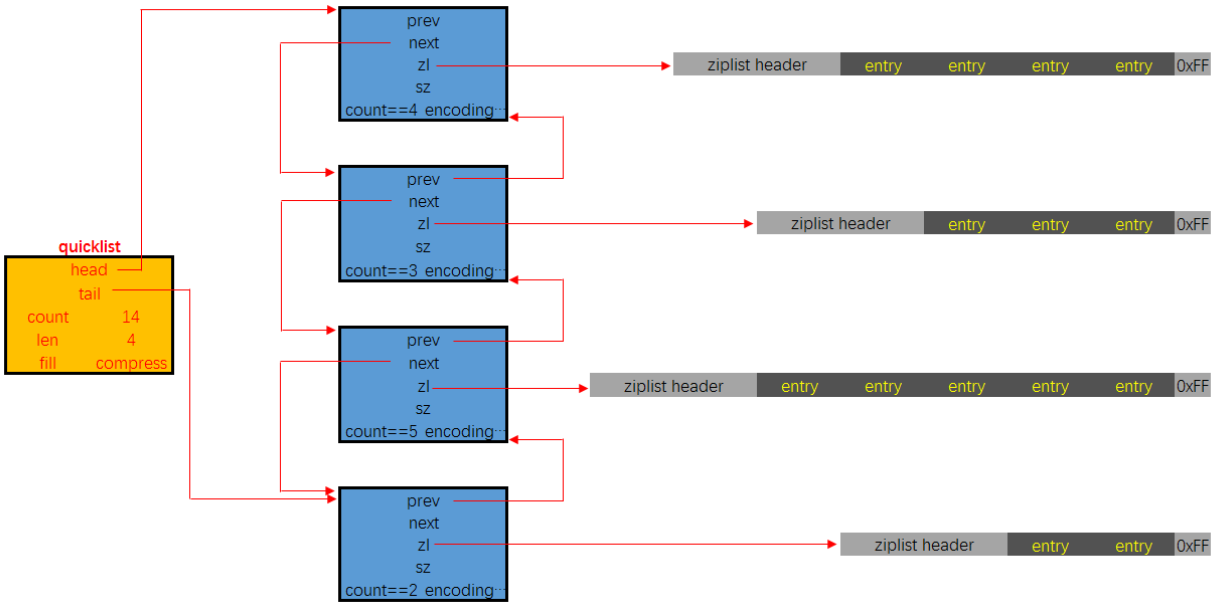
```
/* Node, quicklist, and Iterator are the only data structures used currently. */

/* quicklistNode is a 32 byte struct describing a ziplist for a quicklist.
 * We use bit fields keep the quicklistNode at 32 bytes.
 * count: 16 bits, max 65536 (max zl bytes is 65k, so max count actually < 32k).
 * encoding: 2 bits, RAW=1, LZF=2.
 * container: 2 bits, NONE=1, ZIPLIST=2.
 * recompress: 1 bit, bool, true if node is temporarry decompressed for usage.
 * attempted_compress: 1 bit, boolean, used for verifying during testing.
 * extra: 12 bits, free for future use; pads out the remainder of 32 bits */
typedef struct quicklistNode {
    struct quicklistNode *prev;
    struct quicklistNode *next;
    unsigned char *zl;
    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;

/* quicklistLZF is a 4+N byte struct holding 'sz' followed by 'compressed'.
 * 'sz' is byte length of 'compressed' field.
 * 'compressed' is LZF data with total (compressed) length 'sz'
 * NOTE: uncompressed length is stored in quicklistNode->sz.
 * When quicklistNode->zl is compressed, node->zl points to a quicklistLZF */
typedef struct quicklistLZF {
    unsigned int sz; /* LZF size in bytes*/
    char compressed[];
} quicklistLZF;

/* quicklist is a 40 byte struct (on 64-bit systems) describing a quicklist.
 * 'count' is the number of total entries.
 * 'len' is the number of quicklist nodes.
 * 'compress' is: -1 if compression disabled, otherwise it's the number
 *                of quicklistNodes to leave uncompressed at ends of quicklist.
 * 'fill' is the user-requested (or default) fill factor. */
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;

typedef struct quicklistIter {
    const quicklist *quicklist;
    quicklistNode *current;
    unsigned char *zi;
    long offset; /* offset in current ziplist */
    int direction;
} quicklistIter;

typedef struct quicklistEntry {
    const quicklist *quicklist;
    quicklistNode *node;
    unsigned char *zi;
    unsigned char *value;
    long long longval;
    unsigned int sz;
    int offset;
} quicklistEntry;
```

这里定义了五个结构体：

0. `quicklistNode` , 宏观上, `quicklist` 是一个链表, 这个结构描述的就是链表中的结点. 它通过 `zl` 字段持有底层的 `ziplist` . 简单来讲, 它描述了一个 `ziplist` 实例
1. `quicklistLZF` , `ziplist` 是一段连续的内存, 用LZ4算法压缩后, 就可以包装成一个 `quicklistLZF` 结构. 是否压缩 `quicklist` 中的每个 `ziplist` 实例是一个可配置项. 若这个配置项是开启的, 那么 `quicklistNode.zl` 字段指向的就不是一个 `ziplist` 实例, 而是一个压缩后的 `quicklistLZF` 实例
2. `quicklist` . 这就是一个双链表的定义. `head, tail` 分别指向头尾指针. `len` 代表链表中的结点. `count` 指的是整个 `quicklist` 中的所有 `ziplist` 中的 `entry` 的数目. `fill` 字段影响着每个链表结点中 `ziplist` 的最大占用空间, `compress` 影响着是否要对每个 `ziplist` 以LZ4算法进行进一步压缩以更节省内存空间.
3. `quicklistIter` 是一个迭代器
4. `quicklistEntry` 是对 `ziplist` 中的 `entry` 概念的封装. `quicklist` 作为一个封装良好的数据结构, 不希望使用者感知到其内部的实现, 所以需要把 `ziplist.entry` 的概念重新包装一下.

`quicklist` 的内存布局图如下所示:



下面是有关 `quicklist` 的更多额外信息:

0. `quicklist.fill` 的值影响着每个链表结点中, `ziplist` 的长度.
   0. 当数值为负数时, 代表以字节数限制单个 `ziplist` 的最大长度. 具体为:
   0. `-1` 不超过4kb
   0. `-2` 不超过 8kb
   0. `-3` 不超过 16kb
   0. `-4` 不超过 32kb
   0. `-5` 不超过 64kb
   0. 当数值为正数时, 代表以 `entry` 数目限制单个 `ziplist` 的长度. 值即为数目. 由于该字段仅占16位, 所以以 `entry` 数目限制 `ziplist` 的容量时, 最大值为2^15个

1. `quicklist.compress` 的值影响着 `quicklistNode.zl` 字段指向的是原生的 `ziplist` , 还是经过压缩包装后的 `quicklistLZF`
    0. `0` 表示不压缩, `zl` 字段直接指向 `ziplist`
    0. `1` 表示 `quicklist` 的链表头尾结点不压缩, 其余结点的 `zl` 字段指向的是经过压缩后的 `quicklistLZF`
    0. `2` 表示 `quicklist` 的链表头两个, 与末两个结点不压缩, 其余结点的 `zl` 字段指向的是经过压缩后的 `quicklistLZF`
    0. 以此类推, 最大值为 `2^16`
2. `quicklistNode.encoding` 字段, 以指示本链表结点所持有的 `ziplist` 是否经过了压缩. `1` 代表未压缩, 持有的是原生的 `ziplist` , `2` 代表压缩过
3. `quicklistNode.container` 字段指示的是每个链表结点所持有的数据类型是什么. 默认的实现是 `ziplist` , 对应的该字段的值是 `2` , 目前Redis没有提供其它实现. 所以实际上, 该字段的值恒为2
4. `quicklistNode.recompress` 字段指示的是当前结点所持有的 `ziplist` 是否经过了解压. 如果该字段为 `1` 即代表之前被解压过, 且需要在下一次操作时重新压缩.

`quicklist` 的具体实现代码篇幅很长, 这里就不贴代码片断了, 从内存布局上也能看出来, 由于每个结点持有的 `ziplist` 是有上限长度的, 所以在与操作时要考虑的分支情况比较多. 想想都蛋疼.

`quicklist` 有自己的优点, 也有缺点, 对于使用者来说, 其使用体验类似于线性数据结构, `list` 作为最传统的双链表, 结点通过指针持有数据, 指针字段会耗费大量内存. `ziplist` 解决了耗费内存这个问题. 但引入了新的问题: 每次写操作整个 `ziplist` 的内存都需要重分配. `quicklist` 在两者之间做了一个平衡. 并且使用者可以通过自定义 `quicklist.fill` , 根据实际业务情况, 经验主义调参.

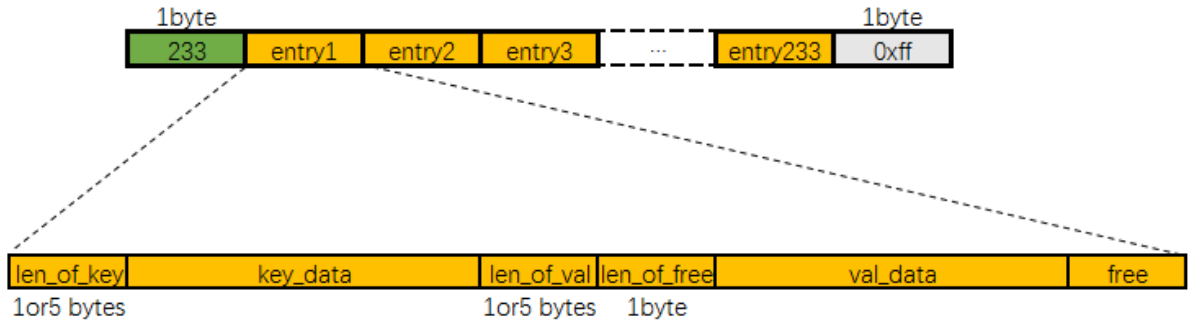
2.8 zipmap

`dict` 作为字典结构, 优点很多, 扩展性强悍, 支持平滑扩容等等, 但对于字典中的键值均为二进制数据, 且长度都很小时, `dict` 的中的一坨指针会浪费不少内存, 因此Redis又实现了一个轻量级的字典, 即为 `zipmap` .

`zipmap` 适合使用的场合是:

0. 键值对量不大, 单个键, 单个值长度小
1. 键值均是二进制数据, 而不是复合结构或复杂结构. `dict` 支持各种嵌套, 字典本身并不持有数据, 而仅持有数据的指针. 但 `zipmap` 是直接持有数据的.

`zipmap` 的定义与实现在 `src/zipmap.h` 与 `src/zipmap.c` 两个文件中, 其定义与实现均未定义任何struct结构体, 因为 `zipmap` 的内存布局就是一块连续的内存空间. 其内存布局如下所示:



0. `zipmap` 起始的第一个字节存储的是 `zipmap` 中键值对的个数. 如果键值对的个数大于254的话, 那么这个字节的值就是固定值254, 真实的键值对个数需要遍历才能获得.
1. `zipmap` 的最后一个字节是固定值 `0xFF`
2. `zipmap` 中的每一个键值对, 称为一个 `entry` , 其内存占用如上图, 分别六部分:
   0. `len_of_key` , 一字节或五字节. 存储的是键的二进制长度. 如果长度小于254, 则用1字节存储, 否则用五个字节存储, 第一个字节的值固定为 `0xFE` , 后四个字节以小端序 `uint32_t` 类型存储着键的二进制长度.
   0. `key_data` 为键的数据
   0. `len_of_val` , 一字节或五字节, 存储的是值的二进制长度. 编码方式同 `len_of_key`
   0. `len_of_free` , 固定值1字节, 存储的是 `entry` 中未使用的空间的字节数. 未使用的空间即为图中的 `free` , 它一般是由于键值对中的值被替换发生的. 比如, 键值对 `hello <-> word` 被修改为 `hello <-> w` 后, 就空了四个字节的闲置空间
   0. `val_data` , 为值的数据
   0. `free` , 为闲置空间. 由于 `len_of_free` 的值最大只能是254, 所以如果值的变更导致闲置空间大于254的话, `zipmap` 就会回收内存空间.

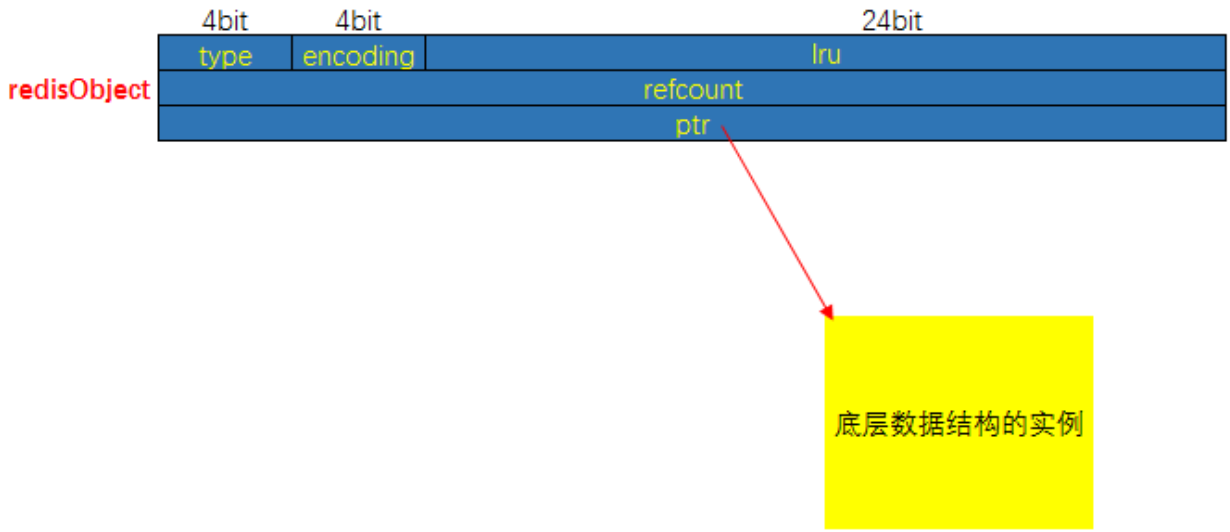
3. 胶水层 redisObject

衔接底层数据结构, 与五种Value Type之间的桥梁就是 `redisObject` 这个结构. 该结构的关键定义如下(位于 `src/server.h` 中):

```
/*-----------------------------------------------------------------------------
 * Data types
 *----------------------------------------------------------------------------*/

/* A redis object, that is a type able to hold a string / list / set */

/* The actual Redis Object */
#define OBJ_STRING 0
#define OBJ_LIST 1
#define OBJ_SET 2
#define OBJ_ZSET 3
#define OBJ_HASH 4

/* 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 LRU_BITS 24
#define LRU_CLOCK_MAX ((1<<LRU_BITS)-1) /* Max value of obj->lru */
#define LRU_CLOCK_RESOLUTION 1000 /* LRU clock resolution in ms */

#define OBJ_SHARED_REFCOUNT INT_MAX
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    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;
    void *ptr;
} robj;
```

`redisObject` 的内存布局如下:



从定义上来看, `redisObject` 有:

0. 与Value Type一致的Object Type, 即 `type` 字段
0. 特定的Object Encoding, 即 `encoding` 字段, 表明对象底层使用的数据结构类型
0. 记录最末一次访问时间的 `lru` 字段

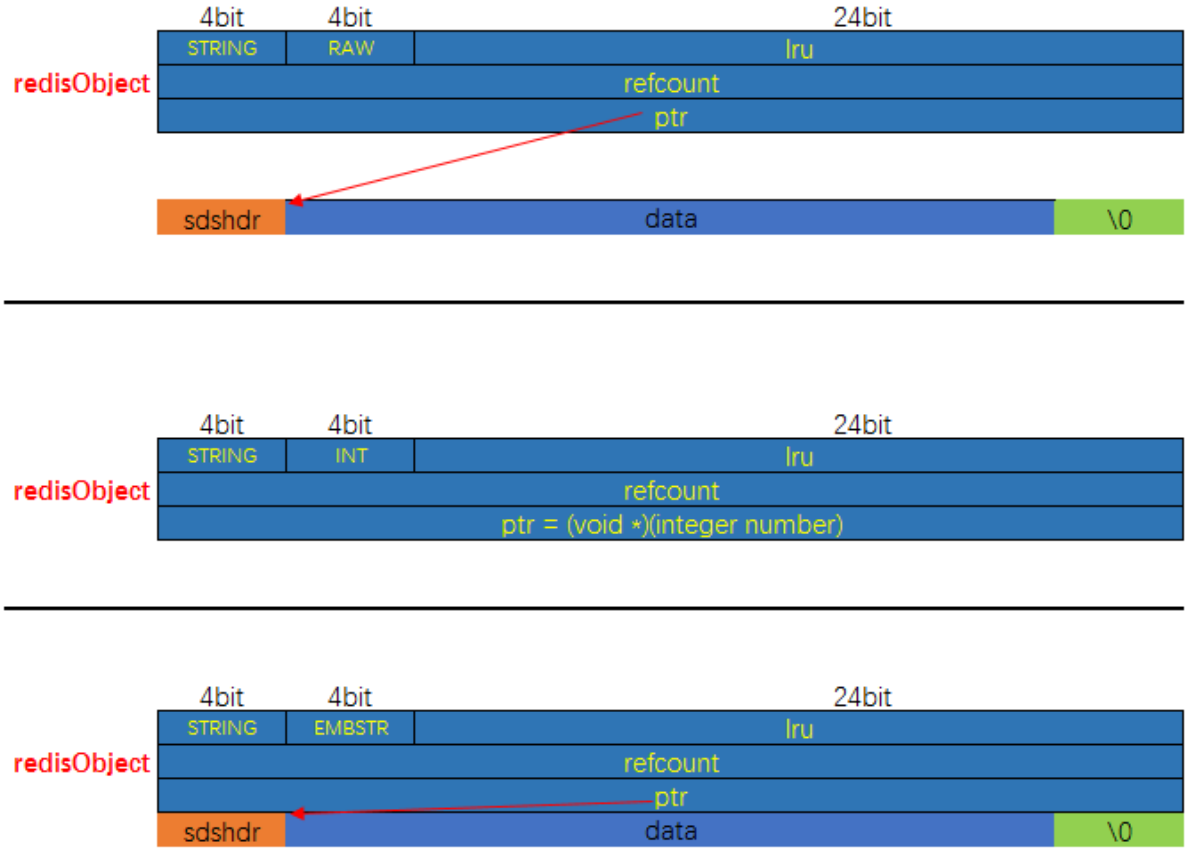
0. 引用计数 `refcount`
0. 指向底层数据结构实例的 `ptr` 字段

`redisObject` 的通用操作API如下:

API	功能
char *strEncoding(int encoding)	返回各种编码的可读字符串表达
void decrRefCount(robj *o);	引用计数-1. 若减后引用计数会降为0, 则会自动调用freeXXXObject函数释放对象
void decrRefCountVoid(void *o);	功能同decrRefCount, 只不过接收的是void * 型参数
void incrRefCount(robj *o);	引用计数+1
robj *makeObjectShared(robj *o);	将对象置为"全局共享对象", 所谓的"全局只读共享对象", 有以下特征 0. 内部引用计数为 INT_MAX 0. 引用计数操作函数对其不起作用 0. 多纯种共享读是安全的, 不需要加锁 0. 禁止写操作
robj *resetRefCount(robj *obj);	将引用计数置为0, 但不会调用freeXXXObject函数释放对象
robj *createObject(int type, void *ptr);	创建一个对象, 对象类型由参数指定, 对象底层编码指定为RAW, 底层数据由参数提供, 对象引用计数为1. 并初始化lru字段. 若服务器采用LRU算法, 则置该字段的值为当前分钟级别的一个时间戳. 若服务器采用LFU算法, 则置为一个计数值.
unsigned long long estimateObjectIdleTime(robj *o)	获取一个对象未被访问的时间, 单位为毫秒. 由于redisObject中lru字段有24位, 并不是无限长, 所以有循环溢出的风险, 当发生循环溢出时(即当前LRU时钟计数比对象中的lru字段小), 那么该函数始终保守认为循环溢出只发生了一次

3.1 字符串对象

字符串对象支持三种编码方式： `INT` ， `RAW` ， `EMBSTR` ，三种方式的内存布局分别如下：



字符串对象的相关接口如下：

分类	API名	功能
创建接口	robj *createEmbeddedStringObject(const char *ptr,size_t len)	创建一个编码为EMBSTR的字符串对象. 即底层使用SDS, 且SDS与RedisObject位于同一块连续内存上
--	robj *createRawStringObject(const char *ptr,size_t len)	创建一个编码为RAW的字符串对象. 即底层使用SDS, 且SDS由RedisObject间接持有 内部是先用入参创建一个SDS, 然后用这个SDS再去调用createObject
--	robj *createStringObject(const char *ptr,size_t len)	创建一个字符串对象. 当len参数的值小于或等于OBJ_ENCODING_EMBSTR_SIZE_LIMIT时, 编码方式为EMBSTR, 否则为RAW 内部是通过调用createRawStringObject与

分类	API名	功能
		createEmbeddedStringObject来创建不同编码的字符串对象的
--	robj *createStringObjectFromLongLong(long long value)	根据整数值, 创建一个字符串对象. 若可复用全局共享字符串对象池中的对象, 则会尽量复用. 否则以最节省内存的原则, 来决定对象的编码
--	robj *createStringObjectFromLongDouble(long double value,int humanfriendly)	根据浮点数值, 创建一个字符串对象 其中参数humanfriendly不为0, 则字符串以小数形式表达. 否则以exp计数法表达.根据字符串表达的长短, 编码可能是RAW, 或EMBSTR
释放接口	void freeStringObject(robj *o)	释放字符串对象. 若字符串对象底层使用SDS, 则调用sdsfree释放这个SDS. 否则什么也不做
读写接口	robj *dupStringObject(const robj *o)	创建一个字符串对象的深拷贝副本. 不影响原字符串对象的引用计数. 创建的副本与原字符串毫无关联
--	int isSdsRepresentableAsLongLong(sds s,long long *llval)	判断SDS字符串是否是一个取值在long long数值范围内的数值的字符串表达. 如果是, 就把相应的数值置在出参中 内部调用的是string2ll来判断 严格来讲这不应该算是RedisObject的接口函数, 而应当算是SDS的接口函数"
--	int isObjectRepresentableAsLongLong(robj *o,long long *llval)	判断字符串对象是否是一个取值在long long数值范围内的数值的字符串表达. 如果是, 就把相应的数值置在出参中.
--	robj *tryObjectEncoding(robj *o)	尝试缩减这个字符串对象的内存占用. 策略为: 如果字符串对象代表的是一个位于long取值范围内的数值, 则尝试返回全局共享字符串对象池里的等价对象. 若由于服务器配置等原因不成功, 则尝试将对象编码改为INT

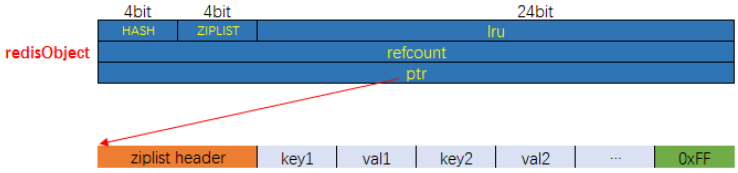
分类	API名	功能
		如果以上都不成功, 则尝试将对象的编码改为EMBSTR 若以上都不成功, 则在对象的编码为RAW的状态下, 至少调用sdsRemoveFreeSpace来移除掉内部SDS中, 闲置的内存空间
--	robj *getDecodedObject(robj *o)	返回字符串对象的一个浅拷贝. 在编码为RAW或EMBSTR时, 底层数据引用计数+1, 返回一个共享句柄 在编码为INT时, 返回一个编码为RAW或EMBSTR的新副本的句柄. 新旧对象之间无关
--	size_t stringObjectLen(robj *o)	返回字符串对象中的字符个数
--	int getDoubleFromObject(const robj *o,double *target)	从字符串对象中解析出数值, 兼容整数值
--	int getLongLongFromObject(robj *o,long long *target)	从字符串对象中解析出整数值, 不兼容浮点数值
--	int getLongDoubleFromObject(robj *o,long double *target)	从字符串对象中解析出数值, 兼容整数值
--	int compareStringObjects(robj *a, robj *b)	二进制比较两个字符串对象. 若有字符串对象使用的是INT编码, 则先会把ptr中的数值转化为字符串表达, 然后再去比较
--	int collateStringObjects(robj *a, robj *b)	底层调用strcoll去比较两个字符串对象. 比较的大小结果受LC_LOCALE的影响
--	int equalStringObjects(robj *a, robj *b)	字符串判等

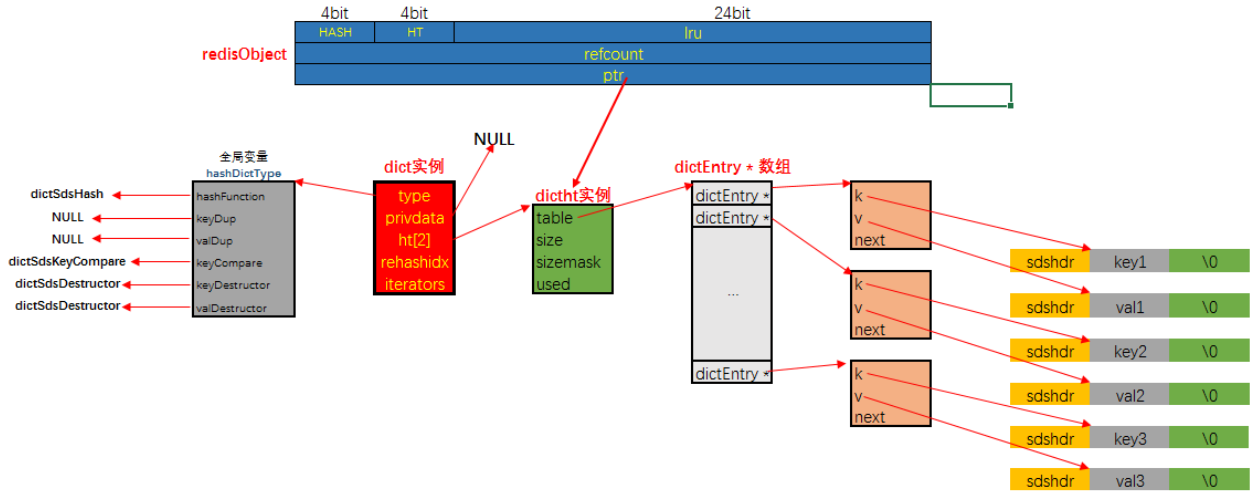
分类	API名	功能
--	#define sdsEncodedObject(objptr)	宏, 判断字符串对象的内部是否为SDS实现. 即编码为RAW或EMBSTR

3.2 哈希对象

哈希对象的底层实现有两种, 一种是 `dict` , 一种是 `ziplist` . 分别对应编码 `HT` 与 `ZIPLIST` . 而之前介绍的 `zipmap` 这种结构, 虽然也是一种轻量级的字典结构, 且纵使在源代码中有相应的编码宏值, 但遗憾的是, 至Redis 4.0.10, 目前哈希对象的底层编码仍然只有 `ziplist` 与 `dict` 两种

`dict` 自不必说, 本身就是字典类型, 存储键值对的. 用 `ziplist` 作为底层数据结构时, 是将键值对以 `<key1><value1><key2><value2>...<keyn><valuen>` 这样的形式存储在 `ziplist` 中的. 两种编码内存布局分别如下:





上图中不严谨的地方有:

0. `ziplist` 中每个entry, 除了键与值本身的二进制数据, 还包括其它字段, 图中没有画出来
0. `dict` 底层可能持有两个 `dictht` 实例
0. 没有画出 `dict` 的哈希冲突

需要注意的是: 当采用 `HT` 编码, 即使用 `dict` 作为哈希对象的底层数据结构时, 键与值均是以sds的形式存储的.

哈希对象的相关接口如下:

分类	API名	功能
创建接口	robj *createHashObject(void)	创建一个空哈希对象 底层编码使用ZIPLIST, 即底层使用ziplist
释放接口	void freeHashObject(robj *o)	释放哈希对象 若哈希对象底层使用的是dict, 则调用dictRelease释放这个dict 若哈希对象底层使用的是ziplist, 则直接释放掉这个ziplist占用的连续内存空间
编码转换接口	void hashTypeConvertZiplist(robj *o, int enc)	将哈希对象的编码从ZIPLIST转换为HT, 即底层实现从ziplist转为dict
--	void hashTypeConvert(robj *o, int enc)	转换哈希对象的编码. 虽然接口设计的好像可以在底层编码之间互相转换, 但实际上这个接口的实现, 目前仅支持从ZIPLIST转向HT
--	void hashTypeTryConversion(robj *o,robj **argv,int start,int end)	o是一个哈希对象. argv是其它对象的数组.(最好是字符串对象, 且为SDS实现) 这个函数会检查argv数组中, 从start到end之间的所有对象, 如果这些对象中, 但凡有一个对象是字符串对象, 且长度超过了用ziplist实现哈希对象时, ziplist的限长 那么o这个哈希对象的编码就会从ZIPLIST转为HT
读写接口	int hashTypeSet(robj *o,sds field,sds value,int flags)	向哈希对象写入一个键值对. 在底层编码为HT时, flag将影响插入键值对时的具体行为. flag可有标志位HASH_SET_TAKE_VALUE与HASH_SET_TAKE_FIELD, 若对应位置1, 代表键与值直接引用参数值. 否则代表要调用sdsdup接口拷贝键与值. 在底层编码为ZIPLIST时, 键与值必然会被拷贝
--	int hashTypeExists(robj *o, sds field)	查询指定键在哈希对象中是否存在

分类	API名	功能
--	unsigned long hashTypeLength(const robj *o)	查询哈希对象中的键值对总数
--	int hashTypeGetFromZiplist(robj *o, sds field,unsigned char **vstr,unsigned int *vlen,long long *vll)	从编码为ZIPLIST的哈希对象中, 取出一个键对应的值. 键从field传入, 当值为数值类型时, 值以*vll传出, 当值为二进制类型时, 值以*vstr与*vlen传出
--	sds hashTypeGetFromHashTable(robj *o, sds field)	从编码为HT的哈希对象中, 取出一个键对应的值. 键从field传入, 值以返回值传出. 若值不存在, 返回NULL"
--	"int hashTypeGetValue(robj *o,sds field,unsigned char **vstr,unsigned int *vlen,long long *vll)	取出哈希对象中指定键对应的值. 若值是数值类型, 则以*vll传出, 否则以*vstr与*vlen传出
--	robj *hashTypeGetValueObject(robj *o, sds field)	取出哈希对象中指定键对应的值, 并包装成RedisObject返回. 返回的对象为字符串对象
--	size_t hashTypeGetValueLength(robj *o, sds field)	取出哈希对象中指定键对应的值的长度
--	int hashTypeDelete(robj *o, sds field)	删除哈希对象中的一个键值对. 键不存在时返回0, 成功删除返回1
迭代器接口	hashTypeIterator *hashTypeInitIterator(robj *subject)	在指定哈希对象上创建一个迭代器
--	void hashTypeReleaseIterator(hashTypeIterator *hi)	释放哈希对象的迭代器
--	int hashTypeNext(hashTypeIterator *hi)	让哈希迭代器步进一步

分类	API名	功能
--	void hashTypeCurrentFromZiplist(hashTypeIterator *hi,int what,unsigned char **vstr,unsigned int *vlen,long long *vll)	取出哈希对象迭代器当前指向的键 或值. 当what传入OBJ_HASH_KEY时, 取的是键, 否则取的是值. 注意, 该函数仅在哈希对象的编码为ZIPLIST时才能正确运行
--	sds hashTypeCurrentFromHashTable(hashTypeIterator *hi,int what)	取出哈希对象迭代器当前指向的键 或值. 当what传入OBJ_HASH_KEY时, 取的是键, 否则取的是值. 注意, 该函数仅在哈希对象的编码为HT时才能正确运行
--	void hashTypeCurrentObject(hashTypeIterator *hi,int what,unsigned char **vstr,unsigned int *vlen,long long *vll)	取出哈希对象迭代器当前指向的键或值. 当what传入OBJ_HASH_KEY时, 取的是键, 否则取的是值.
--	sds hashTypeCurrentObjectNewSds(hashTypeIterator *hi,int what)	取出哈希对象迭代器当前指向的键或值. 且把键或值以一个全新的SDS字符串返回. 当what传入OBJ_HASH_KEY时, 取的是键, 否则取的是值.

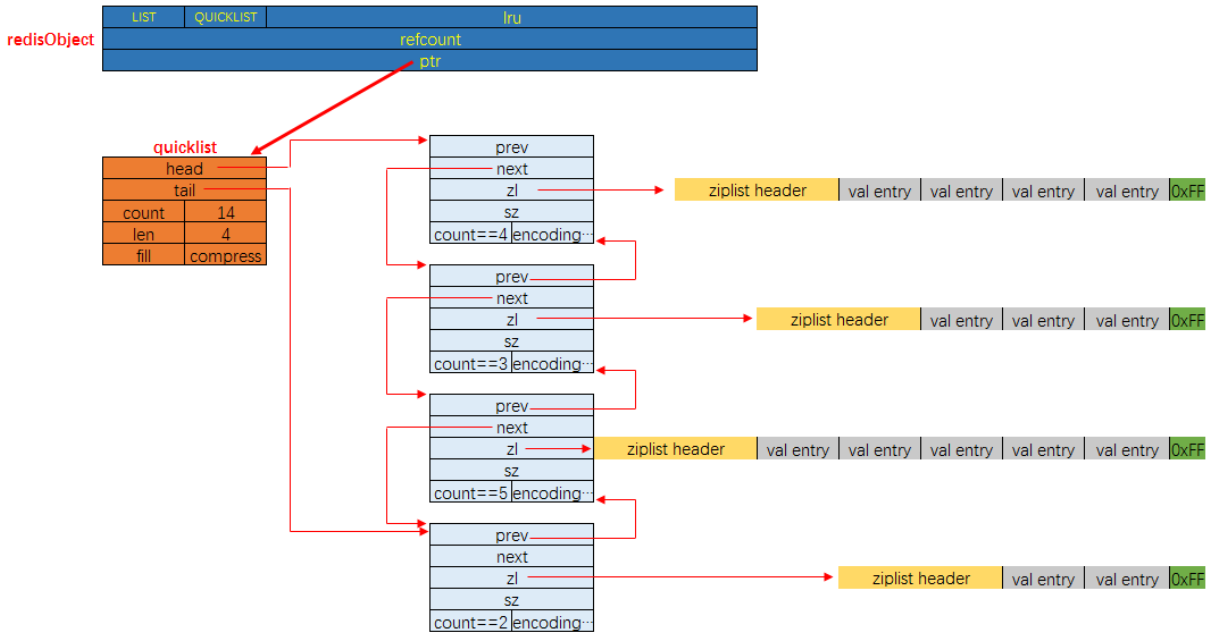
3.3 列表对象

列表对象的底层实现, 历史上是有两种的, 分别是 `ziplist` 与 `list` , 但截止Redis 4.0.10版本, 所有的列表对象API都不再支持除去 `quicklist` 之外的任何底层实现. 也就是说, 目前(Redis 4.0.10), 列表对象支持的底层实现实质上只有一种, 即是 `quicklist` .

列表对象的创建API依然支持从 `ziplist` 的实例创建一个列表对象, 即你可以创建一个底层编码为 `ZIPLIST` 的列表对象, 但如果用该列表对象去调用任何其它列表对象的API, 都会导致panic. 在使用之前, 你只能再次调用相关的底层编码转换接口, 将这个列表对象的底层编码转换为 `QUICKLIST` .

并且遗憾的是, `LINKEDLIST` 这种编码, 即底层为 `list` 的列表, 被彻底淘汰了. 也就是说, 截止目前(Redis 4.0.10), Redis定义的10个对象编码方式宏名中, 有两个被完全闲置了, 分别是: `OBJ_ENCODING_ZIPMAP` 与 `OBJ_ENCODING_LINKEDLIST` . 从Redis的演进历史上来看, 前者是后续可能会得到支持的编码值, 后者则应该是被彻底淘汰了.

列表对象的内存布局如下图所示:



列表对象的API接口如下:

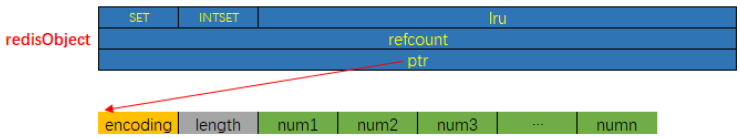
分类	API名	功能
创建接口	robj *createQuicklistObject(void)	创建一个列表对象. 内部编码为QUICKLIST 即内部使用quicklist实现的列表对象
--	robj *createZiplistObject(void)	创建一个列表对象. 内部编码为ZIPLIST 即内部使用ziplist实现的列表对象
释放接口	void freeListObject(robj *o)	释放一个列表对象
编码转换接口	void listTypeConvert(robj *subject, int enc)	转换列表对象的内部编码. 虽然接口设计的好你可以在底层编码之间互相转换, 但实际上这个接口的实现, 目前仅支持从ZIPLIST转换为QUICKLIST 并且蛋疼的是, 4.0.10这个版本中, 所有的列表对象操作API内部实现都仅支持编码方式为QUICKLIST的列表对象, 其它编码方式会panic. 所以目前为止, 这个API的唯一作用, 就是配合createZiplistObject接口, 来使用一个ziplist创建一个内部编码为QUICKLIST的列表对象.
读写接口	void listTypePush(robj *subject,robj *value,int where)	向列表对象中添加一个数据. 由where参数的值控制是在头部添加, 还是

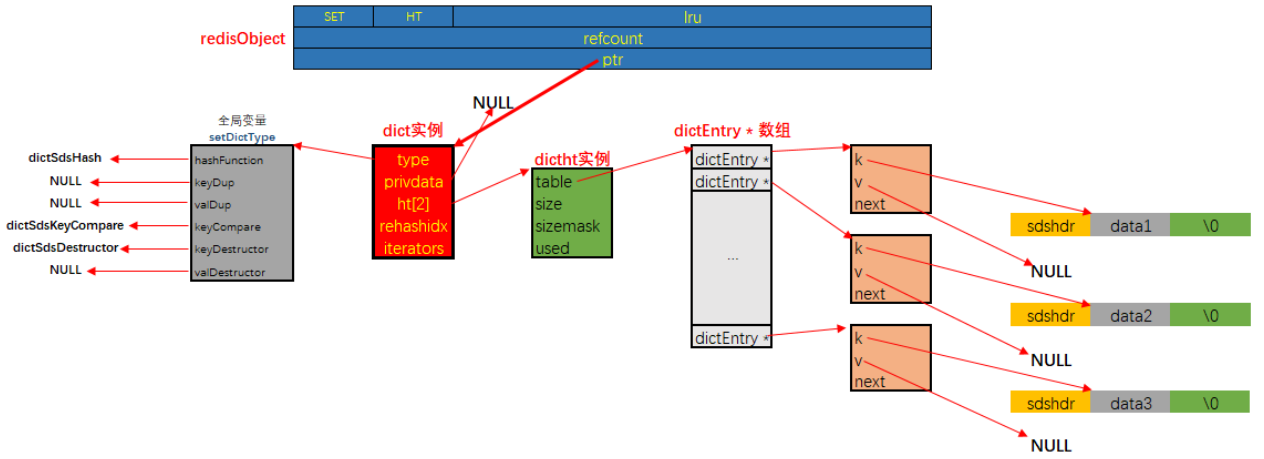
分类	API名	功能
		尾部添加. where可选的值为LIST_HEAD, LIST_TAIL
--	robj *listTypePop(robj *subject,int where)	从列表对象的头部或尾部取出一个数据. 取出的数据通过被包装成字符串对象后返回. 具体取出位置通过参数where控制
--	unsigned long listTypeLength(const robj *subject)	获取列表对象中保存的数据的个数
--	void listTypeInsert(listTypeEntry *entry,robj *value, int where)	将字符串对象中的数据插入到列表对象的头部或尾部. 插入过程中不会拷贝字符串对象持有的数据本身. 但会缩减字符串对象的引用计数.
--	int listTypeEqual(listTypeEntry *entry, robj *o)	判断字符串对象o与列表对象中指定位置上存储的数据是否相同.
--	robj *listTypeGet(listTypeEntry *entry)	获取列表对象中指定位置的数据. 位置信息通过entry传入, 这是一个入参. 数据将拷贝一份后通过SDS形式返回
迭代器接口	listTypeIterator *listTypeInitIterator(robj *subject,long index,unsigned char direction)	创建一个列表对象迭代器
--	void listTypeReleaseIterator(listTypeIterator *li)	释放一个列表对象迭代器
--	int listTypeNext(listTypeIterator *li, listTypeEntry *entry)	让列表对象迭代器步进一步, 并将步进之前迭代器所指向的数据保存在entry中
--	void listTypeDelete(listTypeIterator *iter, listTypeEntry *entry)	删除列表迭代器当前指向的列表对象中存储的数据. 被删除的数据通过entry返回

3.4 集合对象

集合对象的底层实现有两种, 分别是 `intset` 和 `dict` . 分别对应编码宏中的 `INTSET` 和 `HT` . 显然当使用 `intset` 作为底层实现的数据结构时, 集合中存储的只能是数值数据, 且必须是整数. 而当使用 `dict` 作为集合对象的底层实现时, 是将数据全部存储于 `dict` 的键中, 值字段闲置不用.

集合对象的内存布局如下图所示:





集合对象的API接口如下:

分类	API名	功能
创建接口	robj *createSetObject(void)	创建一个空集合对象. 底层编码使用HT, 即底层使用dict
--	robj *createIntsetObject(void)	创建一个空集合对象. 底层编码使用INTSET, 即底层使用intset
--	robj *setTypeCreate(sds value)	创建一个空集合对象. 注意入参虽然携带了一个数据, 但这个数据并不会存储在集合中 这个数据只起到决定编码方式的作用, 若这个数据是数值的字符串表达, 则底层编码则为INTSET, 否则为HT
释放接口	void freeSetObject(robj *o)	释放集合对象. 若集合对象底层使用的是dict, 则调用dictRelease释放这个dict 若集合对象底层使用的是intset, 则直接释放这个intset占用的连续内存

分类	API名	功能
编码转换接口	void setTypeConvert(robj *setobj, int enc)	转换集合对象的内部编码 虽然接口设计的好你可以在底层编码之间互相转换, 但实际上这个接口的实现, 目前仅支持从INTSET转换为HT
读写接口	int setTypeAdd(robj *subject, sds value)	向集合对象中写入一个数据
--	int setTypeRemove(robj *setobj, sds value)	删除集合对象中的一个数据
--	int setTypeIsMember(robj *subject, sds value)	判断指定数据是否在集合对象中
--	int setTypeRandomElement(robj *setobj, sds *sdsele, int64_t *llele)	从集合对象中, 随机选出一个数据, 将其数据通过出参返回. 若数据是数值类型, 则从*llele返回, 否则, 从*sdsele返回. 注意该接口若取得二进制数据, 则*sdsele是直接引用集合内的数据, 而不是拷贝一份
--	unsigned long setTypeSize(const robj *subject)	返回集合中数据的个数
迭代器接口	setTypeIterator *setTypeInitIterator(robj *subject)	创建一个集合对象迭代器
--	void setTypeReleaseIterator(setTypeIterator *si)	释放集合对象迭代器
--	int setTypeNext(setTypeIterator *si, sds *sdsele, int64_t *llele)	让集合迭代器步进一步, 并从出参中返回步进前迭代器所指向的数据. 若数据是数值类型, 则从*llele返回, 否则, 从*sdsele返回 注意该接口若取得二进制数据, 则*sdsele是直接引用集合内的数据, 而不是拷贝一份
--	sds setTypeNextObject(setTypeIt	让集合迭代器步进一步, 并把步进前所指向的数据, 拷贝一份, 构造成一个新的SDS, 作

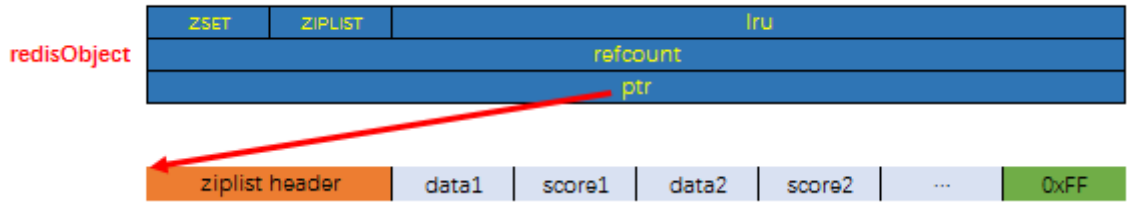
分类	API名	功能
	erator *si)	为返回值返回

3.5 有序集合对象

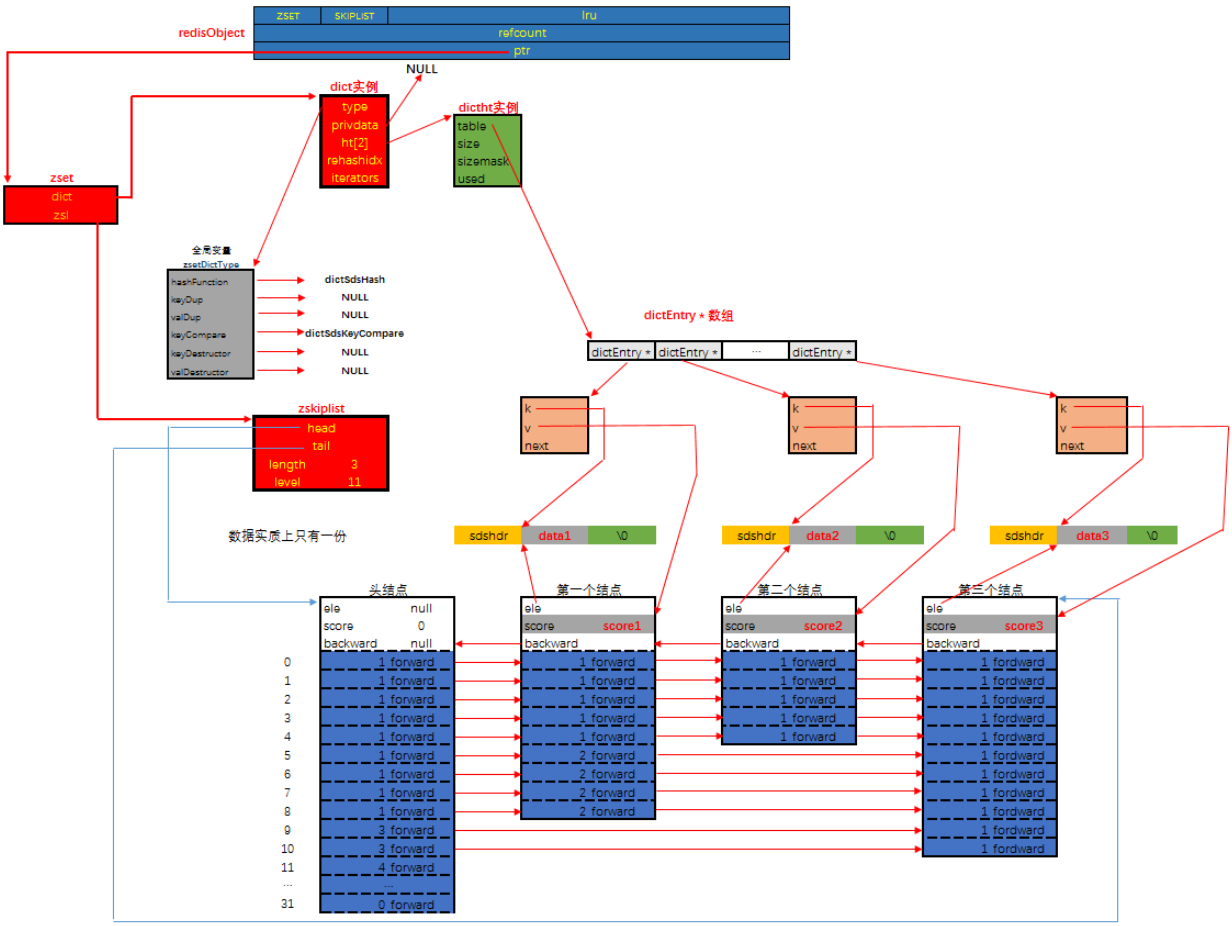
有序集合的底层实现依然有两种，一种是使用 `ziplist` 作为底层实现，另外一种比较特殊，底层使用了两种数据结构： `dict` 与 `skiplist` . 前者对应的编码值宏为 `ZIPLIST` ，后者对应的编码值宏为 `SKIPLIST`

使用 `ziplist` 来实现在序集合很容易理解，只需要在 `ziplist` 这个数据结构的基础上做好排序与去重就可以了. 使用 `zskiplist` 来实现有序集合也很容易理解，Redis中实现的这个跳跃表似乎天然就是为了实现有序集合对象而实现的，那么为什么还要辅助一个 `dict` 实例呢？我们先看来有序集合对象在这两种编码方式下的内存布局，然后再做解释：

首先是编码为 `ZIPLIST` 时，有序集合的内存布局如下：



然后是编码为 `SKIPLIST` 时，有序集合的内存布局如下：



在使用 `dict` 与 `skiplist` 实现有序集合时, 跳跃表负责按分数索引, 字典负责按数据索引. 跳跃表按分数来索引, 查找时间复杂度为O(lgn). 字典按数据索引时, 查找时间复杂度为O(1). 设想如果没有字典, 如果想按数据查分数, 就必须进行遍历. 两套底层数据结构均只作为索引使用, 即不直接持有数据本身. 数据被封装在SDS中, 由跳跃表与字典共同持有. 而数据的分数则由跳跃表结点直接持有(double类型数据), 由字典间接持有.

有序集合对象的API接口如下:

分类	API名	功能
创建接口	robj *createZsetObject(void)	创建一个有序集合对象 默认内部编码为SKIPLIST, 即内部使用zskiplist与dict来实现有序集合
--	robj *createZsetZiplistObject(void)	创建一个有序集合对象 指定内部编码为ZIPLIST, 即内部使用ziplist来实现有序集合
释放接口	void freeZsetObject(robj *o)	释放一个有序集合对象
编码转换接口	void zsetConvert(robj *zobj, int encoding)	转换有序集合对象的内部编码 可以在ZIPLIST与SKIPLIST两种编码间转换
--	void zsetConvertToZiplistIfNeeded(robj *zobj, size_t maxelelen)	判断当前有序集合对象是否有必要将底层编码转换为ZIPLIST, 如果有必要, 就执行转换
读写接口	int zsetScore(robj *zobj, sds member, double *score)	获取有序集合中, 指定数据的得分. 数据由member参数携带, 通过二进制判等的方式匹配
--	int zsetAdd(robj *zobj, double score, sds ele, int *flags, double *newscore)	向有序集合中添加数据, 或更新已存在的数据的得分. flag是一个in-out参数, 其作为入参, 控制函数的具体行为, 其作为出参, 报告函数执行的结果. 作为入参时, *flags的语义如下: ZADD_INCR 递增已存在的数据的得分. 如果数据不存在, 则添加数据, 并设置得分. 且若newscore != NULL, 执行操作后, 数据的得分还会赋值给*newscore ZADD_NX 仅当数据不存在时, 执行添加数据并

分类	API名	功能
		设置得分, 否则什么也不做 ZADD_XX 仅当数据存在时, 执行重置数据得分. 否则什么也不做 作为出参, *flags的语义如下: ZADD_NAN 数据的得分不是一个数值, 代表内部出现的异常 ZADD_ADDED 新数据已经添加至集合中 ZADD_UPDATED 数据的得分已经更新 ZADD_NOP 函数什么也没做
--	int zsetDel(robj *zobj, sds ele)	从有序集合中移除一个数据
--	long zsetRank(robj *zobj, sds ele, int reverse)	获取有序集合中, 指定数据的排名. 若reverse==0, 排名以得分升序排列. 否则排名以得分降序排列. 第一个数据的排名为0, 而不是1
--	unsigned int zsetLength(const robj *zobj)	获取有序集合对象中存储的数据个数

分类: Redis

标签: Redis

张浮生
粉丝 - 23 关注 - 1

« 上一篇: ZooKeeper: 简介, 配置及运维指南
» 下一篇: Redis中单机数据库的实现

posted @ 2018-09-10 18:47 张浮生 阅读(18884) 评论(2) 编辑 收藏 举报

编辑推荐：

· 现代图片性能优化及体验优化指南 - 响应式图片方案
· SQLSERVER 语句交错引发的死锁研究
· 这些「误区」99%的研发都踩过
· 由小见大！不规则造型按钮解决方案
· 小公司需要使用微服务架构吗?

阅读排行：

· 快学会这个技能-.NET API拦截技法
· 记一次 .NET 某医保平台 CPU 爆高分析
· 推荐一款.Net Core开发的后台管理系统YiShaAdmin
· .NET 8 预览版 1：NativeAOT 升级和新的Blazor United (团结)
· 现代图片性能优化及体验优化指南 - 响应式图片方案