Redis Data-Structures

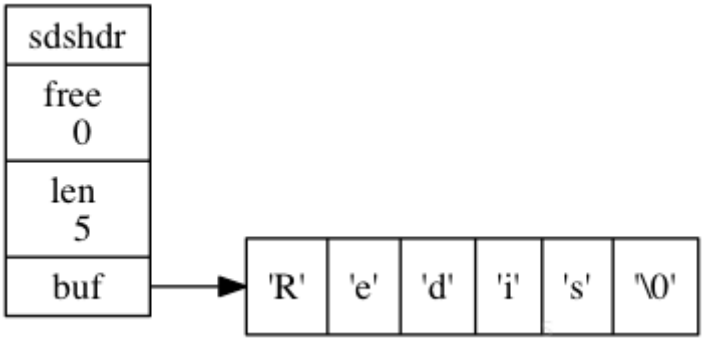
1. Simple Dynamic String (SDS)

- WHAT

  普通动态字符串类型，redis中几乎所有的模块都用到了sds类型。

  数据结构：

  ```
  struct {
      // sds字符串缓存中已经占用的长度
      int len;
      // sds字符串缓存中剩余可分匹配的长度
      int free;
      // 字符串缓存
      char buf[];
  }
  ```



- WHY

  不使用c语言原始字符串(char*)有以下考虑：

  1. 求字符串长度的时间复杂度

     char * => O(n)

     sds => O(1)

  2. Append操作(假设现在append N次)

     char * 需要重分配内存N次

     sds通过内存预分配，可以有效减少内存重分配次数

  3. 二进制安全

     所有 sds API 都会以处理二进制的方式来处理 SDS 存放在 `buf` 数组里的数据， 程序不会对其中的数据做任何限制、过滤、或者假设，标准C语言字符串的'\0'就不是二进制安全的。

- HOW

  如何预分配内存？

```
// 假设现在需要append addlen字节
// 如果sds剩余空间大于addlen，直接在buf中append即可
if (avail >= addlen) return s;
...
// 否则计算新的字符串长度
newlen = (len+addlen);
    // 如果新的长度小于 1MB，就预分配一个同等大小的空闲空间
    if (newlen < SDS_MAX_PREALLOC)
        newlen *= 2;
    // 如果新的长度大于 1MB，就预分配一个1MB大小的空闲空间
    else
        newlen += SDS_MAX_PREALLOC;
```

注意： 根据以上与分配策略可以发现，sds方案实际是以空间换取了时间，如果对redis服务器的append操作过多，同时字符串长度较大的情况下，我们要适时释放预分配空间。

2. List

2.1 双端链表

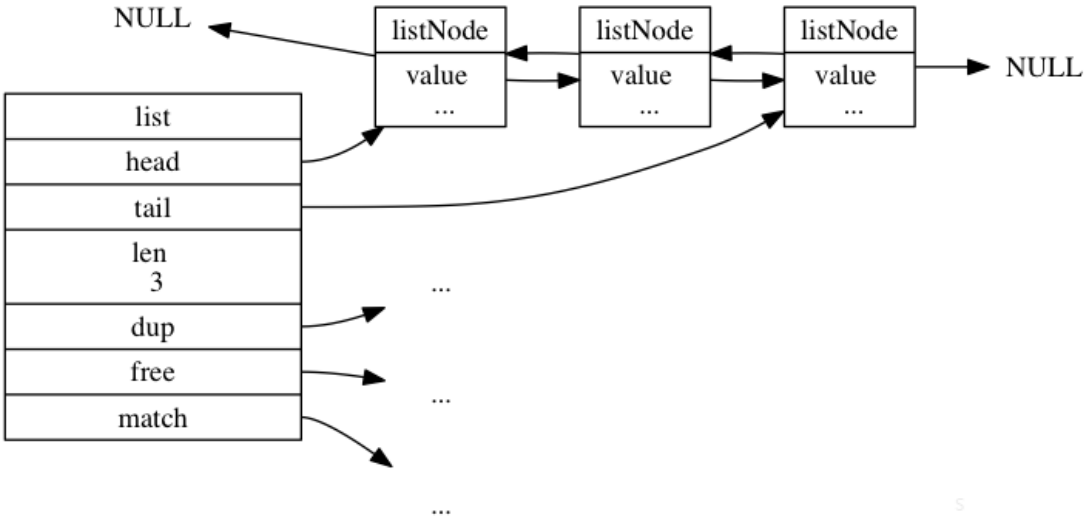
- WHAT

  链表节点结构：

```
typedef struct listNode {
    // 前置节点
    struct listNode *prev;
    // 后置节点
    struct listNode *next;
    // 节点的值
    void *value;
} listNode;
```

  链表整体结构：

```
typedef struct list {
    // 表头节点
    listNode *head;
    // 表尾节点
    listNode *tail;
    // 链表所包含的节点数量
    unsigned long len;
    // 节点值复制函数
    void *(*dup)(void *ptr);
    // 节点值释放函数
    void (*free)(void *ptr);
    // 节点值对比函数
    int (*match)(void *ptr, void *key);
} list;
```



双端链表常规操作的时间复杂度：

insert => O(1)

delete => O(1)

iterate => O(n)

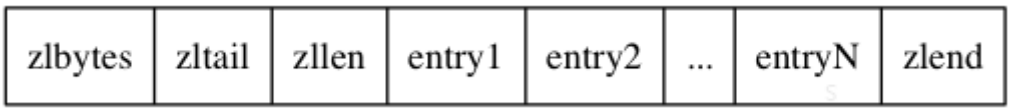
getLen => O(1)

具体参考JAVA中`LinkedList`容器的时间复杂度。

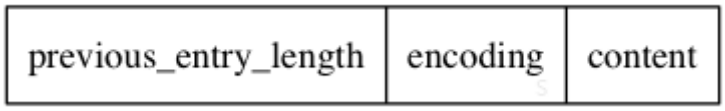
2.2 压缩链表

- WHAT

  压缩列表是 Redis 为了节约内存而开发的， 由一系列特殊编码的连续内存块组成的顺序型（ sequential ）数据结构。一个压缩列表可以包含任意多个节点（ entry ）， 每个节点可以保存一个字节数组或者一个整数值。



每个entry的内容如下图：



previous_entry_length => 上一个节点的长度

encoding => content属性所保存数据的类型以及长度

content => 节点的值

3. Dictionary

- WHAT

  Redis 的字典使用哈希表作为底层实现， 一个哈希表里面可以有多个哈希表节点， 而每个哈希表节点就保存了字典中的一个键值对。

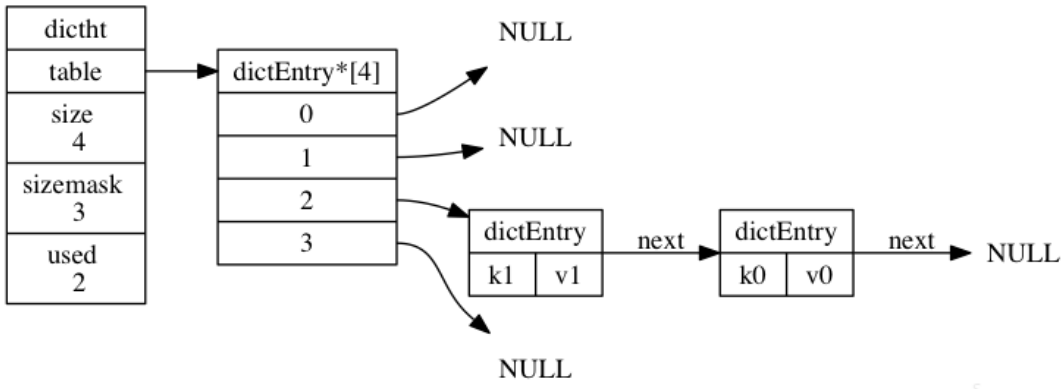
  哈希表节点数据结构：

```
typedef struct dictht {
    // 哈希表数组
    dictEntry **table;
    // 哈希表大小
    unsigned long size;
    // 哈希表大小掩码，用于计算索引值
    // 总是等于 size - 1
    unsigned long sizemask;
    // 该哈希表已有节点的数量
    unsigned long used;
} dictht;
```

哈希表数据结构：

```
typedef struct dict {
    // 类型特定函数
    dictType *type;
    // 私有数据
    void *privdata;
    // 哈希表
    dictht ht[2];
    // rehash 索引
    // 当 rehash 不在进行时，值为 -1
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */

} dict;
```



以上，可知redis的字典使用的哈希表采用的是数组-链表结构，采用拉链法解决哈希冲突。

再来看**字典**的实际结构，在redis中，每一个字典实际上维护着两个哈希表，具体来看代码实现：

```
typedef struct dict {
    // 类型特定函数
    dictType *type;
    // 私有数据
    void *privdata;
    // 哈希表
    dictht ht[2];
    // rehash 索引
    // 当 rehash 不在进行时，值为 -1
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */

} dict;
```

dictType结构体中维护着各种操作函数，这里就类似于object-oriented语言中的成员函数，privdata 属性则保存了函数的可选参数：

```
typedef struct dictType {
    // 计算哈希值的函数
    unsigned int (*hashFunction)(const void *key);
    // 复制键的函数
    void *(*keyDup)(void *privdata, const void *key);
    // 复制值的函数
    void *(*valDup)(void *privdata, const void *obj);
    // 对比键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);
    // 销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);
    // 销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);

} dictType;
```

- REHASH

  哈希表是一个对自身大小很敏感的数据结构，为了维持哈希表较好的性能，如java的哈希容器一样，redis的哈希表也会适时地调整哈希表的大小，dict数据结构中的两张哈希表的第二张（ht[1]）就是用来进行rehash操作的。

  rehash的时机：

  - 服务器目前没有在执行 BGSAVE 命令或者 BGREWRITEAOF 命令， 并且哈希表的负载因子大于等于 `1` ，执行扩容；
  - 服务器目前正在执行 BGSAVE 命令或者 BGREWRITEAOF 命令， 并且哈希表的负载因子大于等于 `5` ，执行扩容；
  - 当哈希表的负载因子小于 `0.1` 时， 执行收缩。

  > 负载因子 = 哈希表已保存节点数量 / 哈希表大小
  >
  > load_factor = ht[0].used / ht[0].size

  在rehash开始时，会给ht[1]分配内存，策略如下：

  - 如果执行的是扩展操作， 那么 `ht[1]` 的大小为第一个大于等于 `ht[0].used * 2` 的 2^n （ `2` 的 `n` 次方幂）；
  - 如果执行的是收缩操作， 那么 `ht[1]` 的大小为第一个大于等于 `ht[0].used` 的 2^n 。

  rehash过程是渐进式的：

- 在 rehash 进行期间， 每次对字典执行添加、删除、查找或者更新操作时， 程序除了执行指定的操作以外， 还会顺带将 `ht[0]` 哈希表在 `rehashidx` 索引上的所有键值对 rehash 到 `ht[1]` ， 当 rehash 工作完成之后， 程序将 `rehashidx` 属性的值+1。
- 在进行渐进式 rehash 的过程中， 字典会同时使用 `ht[0]` 和 `ht[1]` 两个哈希表， 所以在渐进式 rehash 进行期间， 字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行。

字典操作的时间复杂度

put => O(1)

get => O(1)

delete => O(1)

虽然都是Hash表结构，但是HashMap和redis的字典还是有很大的不同，这里附上之前写的一篇文章：HashMap源码解读

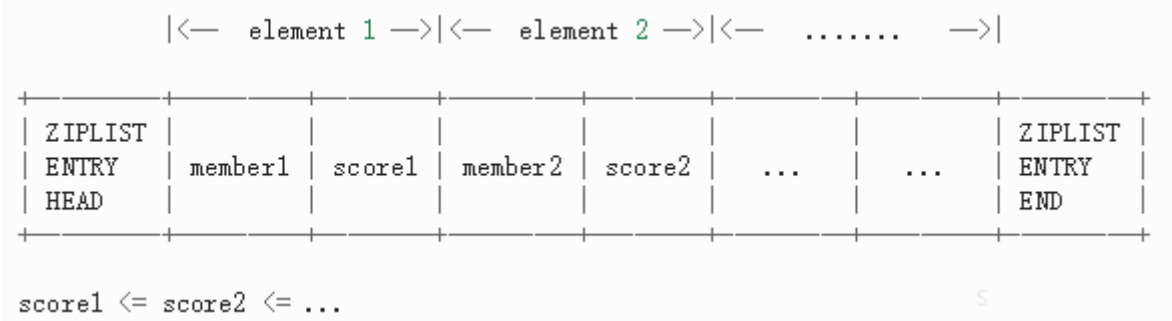
4. Set

set数据结构类似于`HashSet`，底层结构同样使用的是hash表，只不过value部分是一个固定值，我们只需关注key部分。哈希表已经在字典部分解读过，在此部分，我们详述**ZSET**

ZSET

ziplist编码

每个有序集元素以两个相邻的 `ziplist` 节点表示， 第一个节点保存元素的 `member` 域， 第二个元素保存元素的 `score` 域。多个元素之间按 `score` 值从小到大排序， 如果两个元素的 `score` 相同， 那么按字典序对 `member` 进行对比， 决定那个元素排在前面， 那个元素排在后面。



时间复杂度：添加/删除/更新操作都需要执行一次查找元素的操作， 因此复杂度都为O(N)

skiplist编码

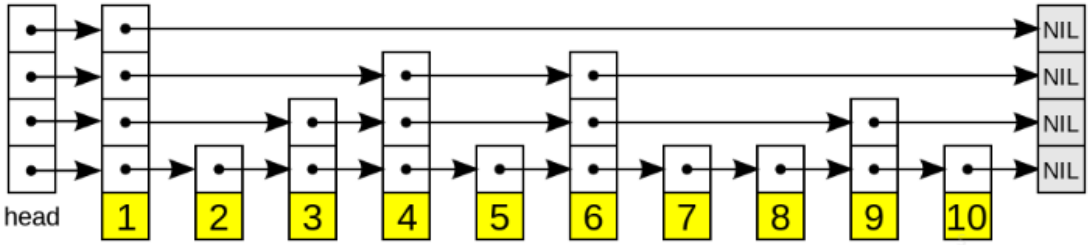
为了维护有序状态，redis并没有使用常见的平衡搜索树，而是使用了跳表（SKIP LIST）这一数据结构，然后结合dict，共同实现了ZSET的需求：

- dict用来存储元素和score的映射，以O(1)的时间复杂度执行ZSCORE命令，get到元素的score
- 跳表用来按照score的大小顺序存储元素，以O(lgn)的平均时间复杂度来搜索元素，最坏情况下为O(n)

数据结构如下：

```
typedef struct zset {
    // 指向跳表的指针
    zskiplist *zsl;
    // 指向字典的指针
    dict *dict;
} zset;
```

- 跳表



跳表主要由以下部分构成：

  - 表头（head）：负责维护跳表的节点指针。
  - 跳表节点：保存着元素值，以及多个层。
  - 层：保存着指向其他元素的指针。高层的指针越过的元素数量大于等于低层的指针，为了提高查找的效率，程序总是从高层先开始访问，然后随着元素值范围的缩小，慢慢降低层次。
  - 表尾：全部由 `NULL` 组成，表示跳跃表的末尾。

时间复杂度：ZSCORE => O(1)，添加/删除/更新操作都是跳表的查找时间复杂度， 因此复杂度都为O(N)