Redis Note

一、Redis 对象

```
typedef struct redisObject {
    // 类型
    unsigned type:4;
    // 编码
    unsigned encoding:4;
    // 指向底层实现数据结构的指针
    void *ptr;
    // ...
} robj;
```

1.1 Type类型

对象	对象 `type` 属性的值	TYPE 命令的输出
字符串对象	`REDIS_STRING`	`"string"`
列表对象	`REDIS_LIST`	`"list"`
哈希对象	`REDIS_HASH`	`"hash"`
集合对象	`REDIS_SET`	`"set"`
有序集合对象	`REDIS_ZSET`	`"zset"`

1.2 Encoding编码

对象所使用的底层数据结构	编码常量	OBJECT ENCODING 命令输出
整数	`REDIS_ENCODING_INT`	`"int"`
`embstr` 编码的简单动态字符串	`REDIS_ENCODING_EMBSTR`	`"embstr"`
简单动态字符串	`REDIS_ENCODING_RAW`	`"raw"`
字典	`REDIS_ENCODING_HT`	`"hashtable"`
双端链表	`REDIS_ENCODING_LINKEDLIST`	`"linkedlist"`
压缩列表	`REDIS_ENCODING_ZIPLIST`	`"ziplist"`
整数集合	`REDIS_ENCODING_INTSET`	`"intset"`
跳跃表和字典	`REDIS_ENCODING_SKIPLIST`	`"skiplist"

1.3 Type和Encoding对应关系

类型	编码	对象
`REDIS_STRING`	`REDIS_ENCODING_INT`	使用整数值实现的字符串对象。
`REDIS_STRING`	`REDIS_ENCODING_EMBSTR`	使用 `embstr` 编码的简单动态字符串实现的字符串对象。
`REDIS_STRING`	`REDIS_ENCODING_RAW`	使用简单动态字符串实现的字符串对象。
`REDIS_LIST`	`REDIS_ENCODING_ZIPLIST`	使用压缩列表实现的列表对象。
`REDIS_LIST`	`REDIS_ENCODING_LINKEDLIST`	使用双端链表实现的列表对象。
`REDIS_HASH`	`REDIS_ENCODING_ZIPLIST`	使用压缩列表实现的哈希对象。
`REDIS_HASH`	`REDIS_ENCODING_HT`	使用字典实现的哈希对象。
`REDIS_SET`	`REDIS_ENCODING_INTSET`	使用整数集合实现的集合对象。
`REDIS_SET`	`REDIS_ENCODING_HT`	使用字典实现的集合对象。
`REDIS_ZSET`	`REDIS_ENCODING_ZIPLIST`	使用压缩列表实现的有序集合对象。
`REDIS_ZSET`	`REDIS_ENCODING_SKIPLIST`	使用跳跃表和字典实现的有序集合对象。

二、对象底层分析

2.1 String

1. 使用整数值实现的字符串对象。
2. 使用 `embstr` 编码的简单动态字符串实现的字符串对象。
3. 使用简单动态字符串实现的字符串对象。

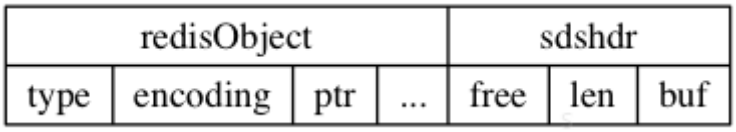
2.1.1 整数字符串

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字符串对象结构的 `ptr`属性里面（将 `void*` 转换成 `long` ）， 并将字符串对象的编码设置为 `int` 。

```
redis> SET number 10010
OK

redis> OBJECT ENCODING number
"int"
```

2.1.2 EMBSTR编码字符串

`embstr` 编码是专门用于保存短字符串的一种优化编码方式， 这种编码和 `raw` 编码一样， 都使用 `redisObject` 结构和 `sdshdr` 结构来表示字符串对象， 但 `raw` 编码会调用两次内存分配函数来分别创建 `redisObject` 结构和 `sdshdr` 结构， 而 `embstr` 编码则通过调用一 次内存分配函数来分配一块连续的空间， 空间中依次包含 `redisObject` 和 `sdshdr` 两个结构



总共64字节，buf段最大为44字节，所以redis中长度不大于44字节的字符串的编码都是`embstr`

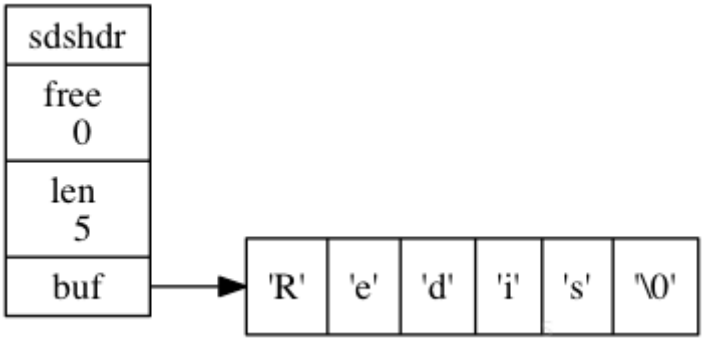
2.1.3 简单动态字符串（RAW）

- 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 sdshdr {
    // 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

1. 使用双端链表实现的列表对象。

   当一个列表键包含了数量比较多的元素， 又或者列表中包含的元素都是比较长的字符串时， Redis 就会使用链表作为列表键的底层实现。

2. 使用压缩列表实现的列表对象。

   当一个列表键只包含少量列表项， 并且每个列表项要么就是小整数值， 要么就是长度比较短的字符串， 那么 Redis 就会使用压缩列表来做列表键的底层实现。

2.1 双端链表

- WHAT

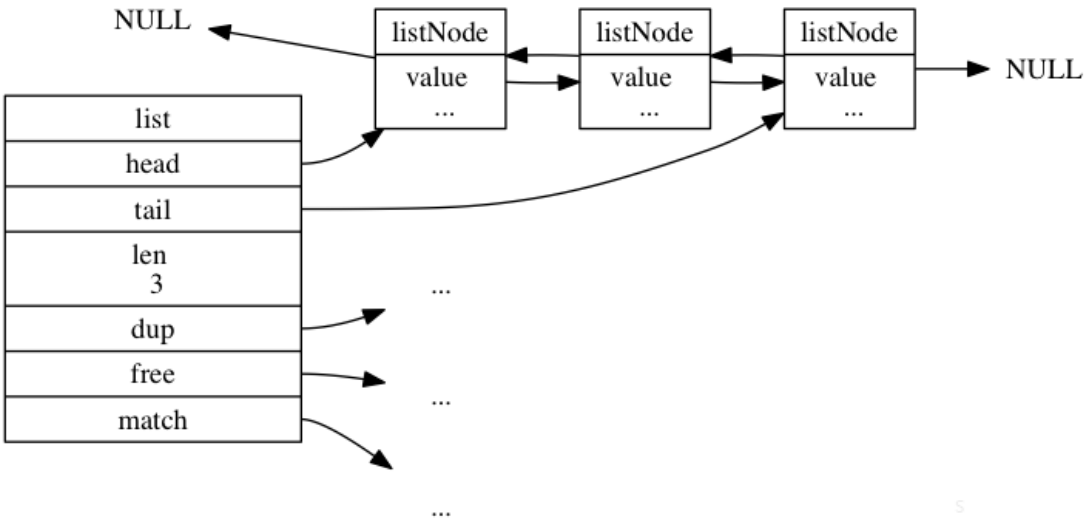
  链表节点结构：

```
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)

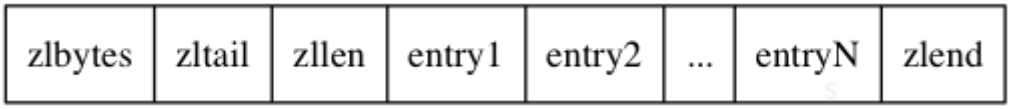
iterate => O(n)

getLen => O(1)

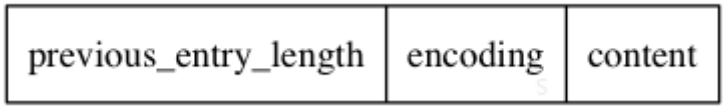
具体参考JAVA中`LinkedList`容器的时间复杂度。

2.2 压缩链表

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

注意：压缩链表中插入和删除元素的最坏复杂度为O(n^2)，因为存在连锁更新的问题

> 在某个/某些节点的前面添加新节点之后， 程序必须沿着路径挨个检查后续的节点，是否满足新长度的编码要求， 直到遇到一个能满足要求的节点（如果有一个能满足，则这个节点之后的其他节点也满足）， 或者到达 `ziplist` 的末端 `zlend` 为止， 这种检查操作的复杂度为 `O(N^2)` 。

但是连锁更新概率很小，所以平均复杂度是O(n)

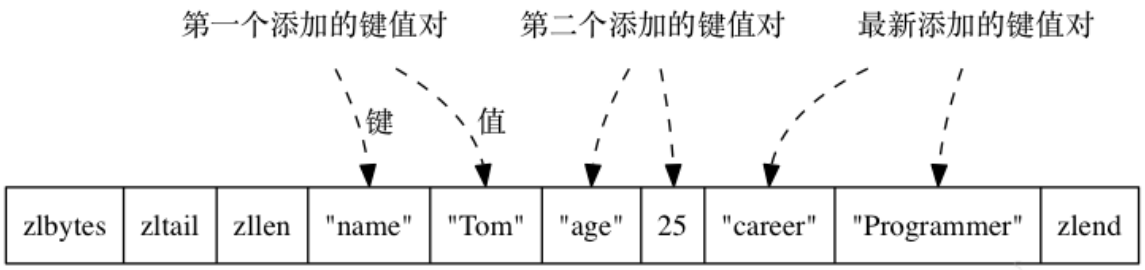
3. Dict

1. 使用压缩列表实现的dict。
2. 使用hash表实现的dict。

3.1 压缩列表

当哈希对象可以同时满足以下两个条件时， 哈希对象使用 `ziplist` 编码：

1. 哈希对象保存的所有键值对的键和值的字符串长度都小于 `64` 字节；
2. 哈希对象保存的键值对数量小于 `512` 个；



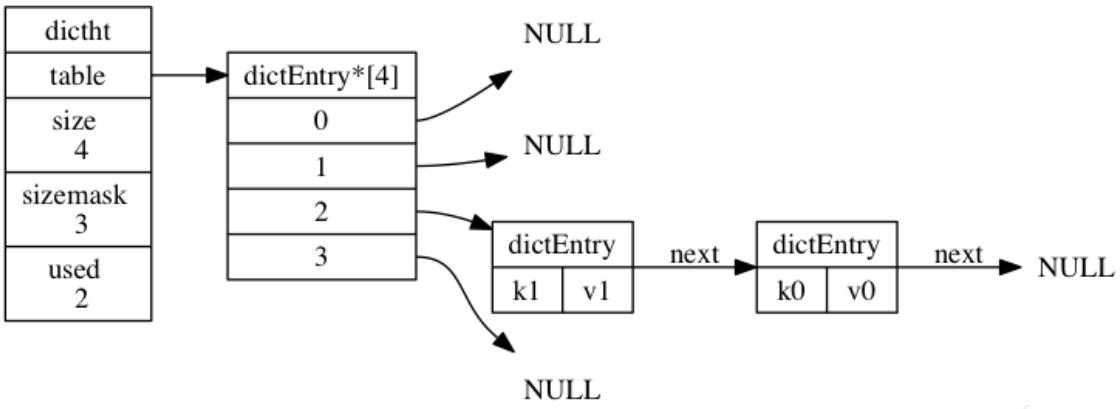
3.2 hash表

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

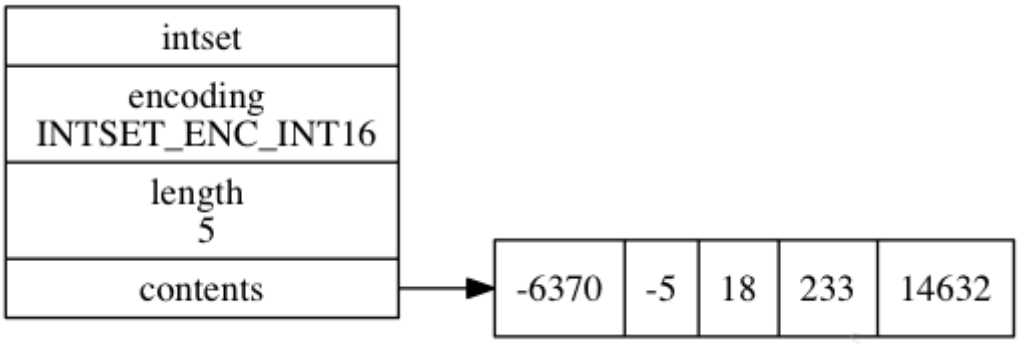
1. 使用整数集合实现的集合对象。
2. 使用hash表实现的集合对象。

4.1 整数集合

当集合对象可以同时满足以下两个条件时， 对象使用 `intset` 编码：

1. 集合对象保存的所有元素都是整数值；
2. 集合对象保存的元素数量不超过 `512` 个；

```
typedef struct intset {

    // 编码方式
    uint32_t encoding;

    // 集合包含的元素数量
    uint32_t length;

    // 保存元素的数组，此数组类型根据encoding字段决定
    int8_t contents[];

} intset;
```



4.2Hash表

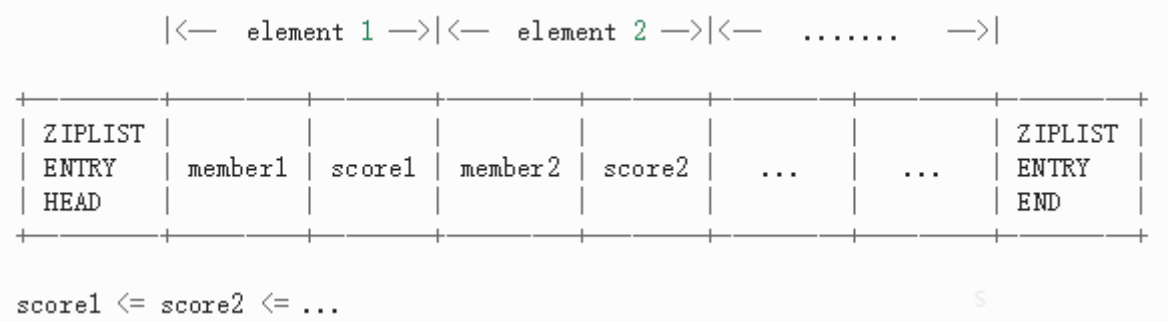
`hashtable` 编码的集合对象使用字典作为底层实现， 字典的每个键都是一个字符串对象， 每个字符串对象包含了一个集合元素， 而字典的值则全部被设置为 `NULL`

5. ZSET

5.1 ziplist编码

1. 有序集合保存的元素数量小于 `128` 个；
2. 有序集合保存的所有元素成员的长度都小于 `64` 字节；

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)

5.2 skiplist编码

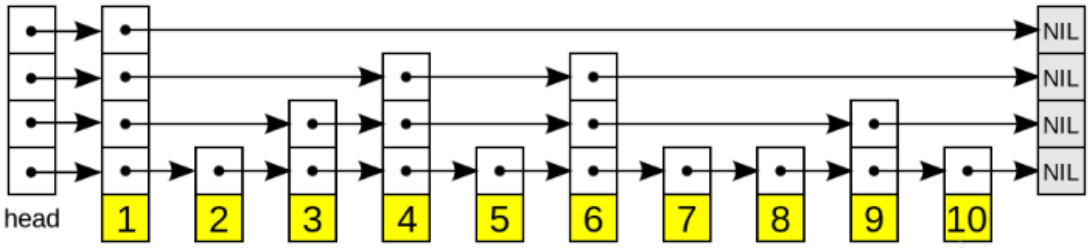
为了让有序集合的查找和范围型操作都尽可能快地执行， Redis 选择了同时使用字典和跳跃表两种数据结构来实现有序集合：

- 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(lgN)

- 具体实现

  注意字典和跳跃表会通过指针共享元素的成员和分值， 所以并不会造成任何数据重复， 也不会因此而浪费任何内存。

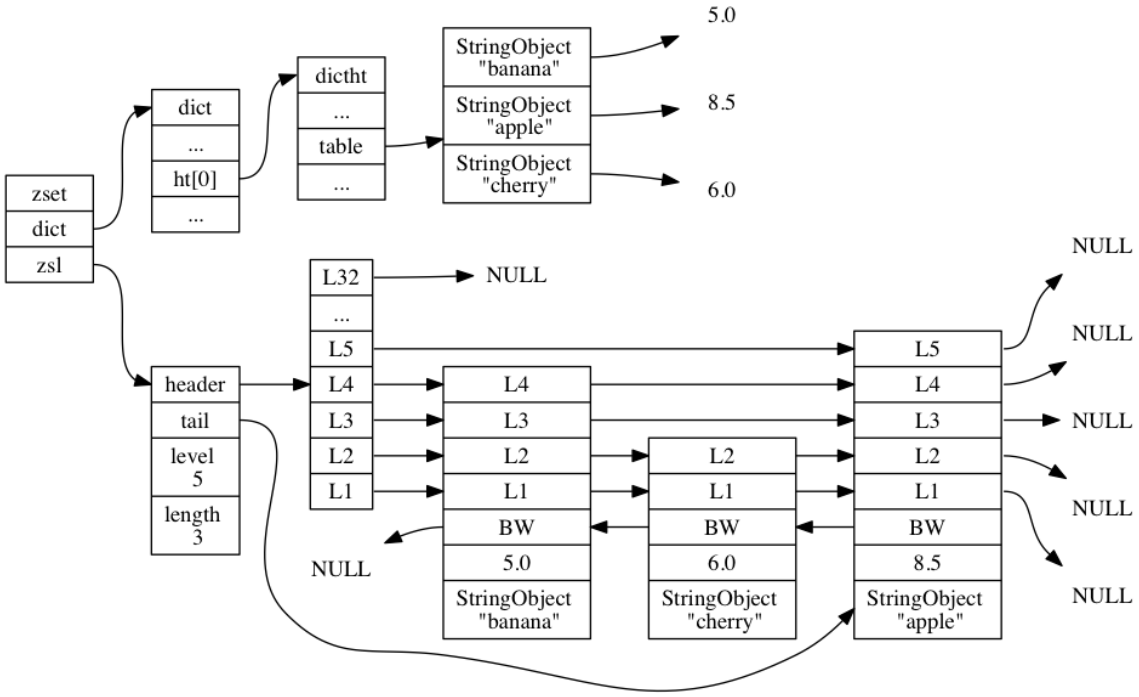


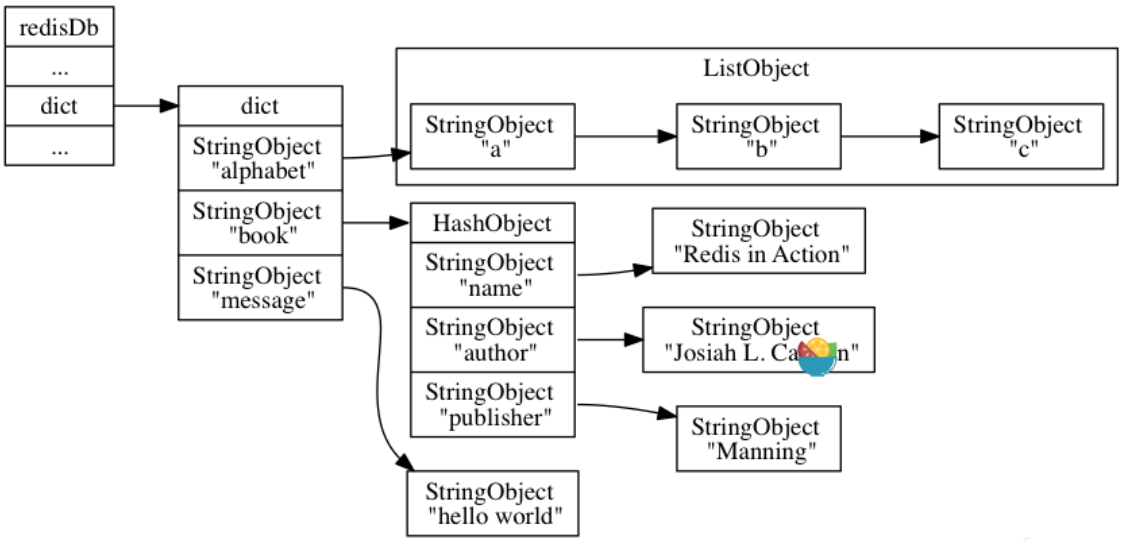
图 8-17　有序集合元素同时被保存在字典和跳跃表中

二、键值空间

整个redis数据库其实是一个巨大的dict，这个dict组织着我们所定义的redis对象，dict的key储存着redis对象的name，value储存着字符串对象、列表对象、哈希表对象、集合对象和有序集合对象在内的任意一种 Redis 对象。

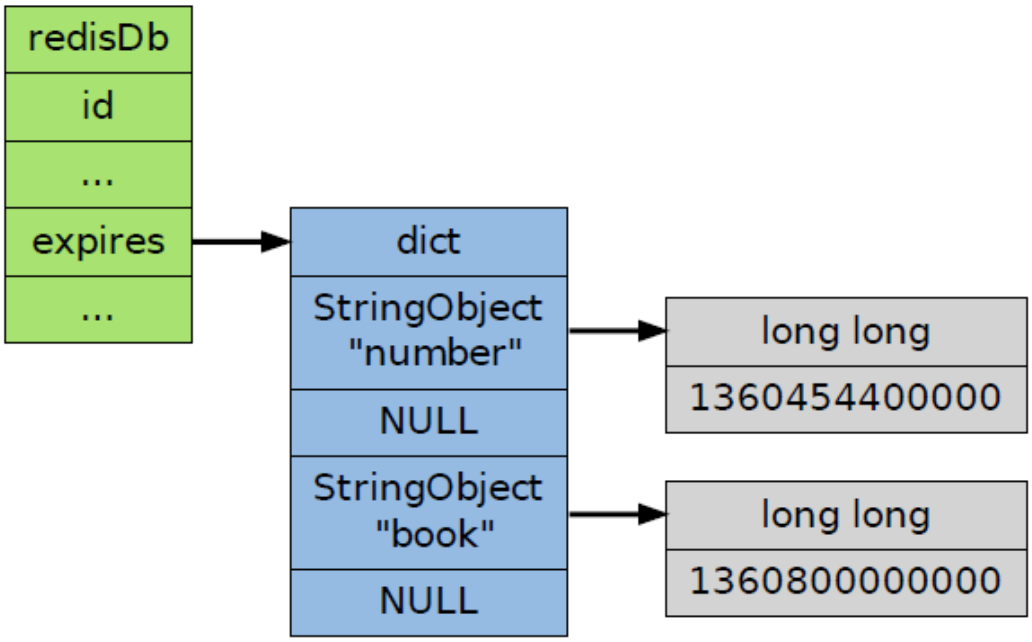
举个例子：

```
redis> SET message "hello world"
OK

redis> RPUSH alphabet "a" "b" "c"
(integer) 3

redis> HSET book name "Redis in Action"
(integer) 1

redis> HSET book author "Josiah L. Carlson"
(integer) 1

redis> HSET book publisher "Manning"
(integer) 1
```



三、过期时间

同样，设置了过期的键也是以一个dict维护：



3.1 过期的清除

- 定时清除

  到点清除，内存友好，cpu不友好

- 惰性清除

  使用该键时判断是否过期，如果过期就删除，内存不友好，cpu最友好

- 定期清除

  定期去检查是否有过期的键，有则清除，前两种的折衷

三、持久化

3.1 RDB持久化

写RDB文件时，先把内存中数据写到临时文件，然后替换原来的RDB文件，所以每时每刻的rdb文件都是完整的

3.1.2 RDB文件结构

REDIS	db_version	database 0	database 3	EOF	check_sum

图 IMAGE_RDB_WITH_TWO_DB　带有两个非空数据库的 RDB 文件示例

、

- `REDIS` 部分的长度为 `5` 字节， 保存着 `"REDIS"` 五个字符。
- `db_version` 长度为 `4` 字节， 它的值是一个字符串表示的整数， 这个整数记录了 RDB 文件的版本号。
- `databases` 部分包含着零个或任意多个非空数据库， 以及各个数据库中的键值对数据，如果数据库为空，那么长度为0字节，数据库数据格式如下：

SELECTDB	db_number	key_value_pairs

图 IMAGE_DATABASE_STRUCT_OF_RDB　RDB 文件中的数据库结构

  - `SELECTDB` 常量的长度为 `1` 字节， 当读入程序遇到这个值的时候， 它知道接下来要读入的将是一个数据库号码。
  - `db_number` 保存着一个数据库号码， 根据号码的大小不同， 这个部分的长度可以是 `1` 字节、 `2` 字节或者 `5` 字节。 当程序读入 `db_number` 部分之后， 服务器会调用 SELECT 命令， 根据读入的数据库号码进行数据库切换， 使得之后读入的键值对可以载入到正确的数据库中。
  - `key_value_pairs` 部分保存了数据库中的所有键值对数据， 如果键值对带有过期时间， 那么过期时间也会和键值对保存在一起。 根据键值对的数量、类型、内容、以及是否有过期时间等条件的不同， `key_value_pairs` 部分的长度也会有所不同。

OPTIONAL_EXPIRE_TIME	TYPE_OF_VALUE	KEY	VALUE

  - `check_sum` 所记录的校验和进行对比， 以此来检查 RDB 文件是否有出错或者损坏的情况出现。

3.1.3 SAVE & BGSAVE

- 可以手动执行：
  - save：会阻塞redis服务器进程，直到创建RDB文件完毕为止（在此期间进程不能处理任何请求）
  - bgsave：fork一个子进程来创建RDB文件，父进程可以继续处理命令请求
- 同时可以配置使服务器自动执行bgsave
- 考虑到竞争问题：在执行bgsave时，服务器会拒绝任何save和bgsave的命令
- 考虑到性能问题：bgsave和bgrewriteaof不会同时执行，而是推迟为顺序执行

3.2 AOF持久化

- RDB 将数据库的快照（snapshot）以二进制的方式保存到磁盘中。
- AOF 则以协议文本的方式，将所有对数据库进行过写入的命令（及其参数）记录到AOF文件，以此达到记录数据库状态的目的。
- AOF相较于RDB持久化的粒度更为细致，服务器故障停机后，RDB持久化会丢失所有上次SAVE or BGSAVE后所有更新的数据，而AOF只会丢失没来得及同步的AOF操作

3.2.1 命令追加

当redis服务器执行完一条命令后，会将命令按照RESP协议编码，然后储存到AOF缓冲区

```
struct redisServer {

    // ...

    // AOF 缓冲区
    sds aof_buf;

    // ...
};
```

3.2.2 AOF写入和同步

redis服务器主线程是一个事件循环，每当有客户端发来请求，就会进行一次循环去处理事件，每次循环的末尾，redis服务器会执行一次 flushAppendOnlyFile()来进行aof文件的写入和同步。

注意：操作系统写入文件，通常会将数据先保存到写缓冲区中，直到缓冲区满了后才会写入到文件中，redis服务器强制将文件数据从缓冲区写入文件称为同步。

同步的方式根据配置文件的配置有以下三种：

`appendfsync` 选项的值	`flushAppendOnlyFile` 函数的行为
`always`	将 `aof_buf` 缓冲区中的所有内容写入并同步到 AOF 文件。
`everysec`	将 `aof_buf` 缓冲区中的所有内容写入到 AOF 文件， 如果上次同步 AOF 文件的时间距离现在超过一秒钟， 那么再次对 AOF 文件进行同步， 并且这个同步操作是由一个线程专门负责执行的。
`no`	将 `aof_buf` 缓冲区中的所有内容写入到 AOF 文件， 但并不对 AOF 文件进行同步， 何时同步由操作系统来决定。

- always：效率最慢，因为每次写入文件都要执行一次同步，但是最安全，不会因为服务器突然停机导致写缓冲区的数据还没写入文件就丢失了。
- everysec：服务器每隔超过一秒，子线程就对 AOF 文件进行一次同步，从效率足够快， 出现故障停机时 数据库也只丢失一秒钟的命令数据。
- no：该模式的同步交由操作系统执行，所以每次时间循环返回都是最快的，但是只要出现故障停机，丢失的数据就会相当的多了。

3.2.3 AOF重写

随着运行时间的流逝，AOF 文件会变得越来越大，redis执行AOF重写创建一个状态与原AOF文件完全相同但是体积更小的新AOF文件。AOF文件重写并不需要对现有AOF文件进行任何读取、分析或者写入操作，而是从数据库中读取键现在的值，然后使用一条命令去记录键值对，代替之前记录这个键值对的多条命令。

例子：

```
redis执行了如下命令：

RPUSH list 1 2 3 4 // [1, 2, 3, 4]
RPOP list // [1, 2, 3]
LPOP list // [2, 3]
LPUSH list 1 // [1, 2, 3]
那么当前列表键list 在数据库中的值就为[1, 2, 3]

重写的时候完全可以使用
RPUSH list 1 2 3
这一条命令来替代原来的四条命令
```

由于bgrewriteaof执行的同时，服务器依旧会接受新命令，为了保证新的aof和数据库状态同步，redis会将这些命令保存在aof重写缓冲区，待新的aof文件重写完成后，再将此缓冲区的命令append到新aof文件中，然后将新的aof文件重命名，替代原先的aof文件。