数据结构与对象

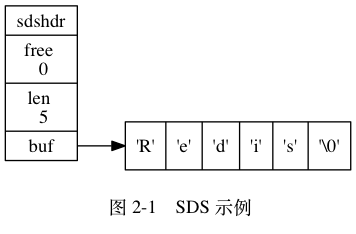
Redis的底层数据结构，了解Redis的底层数据结构有助于我们更好的运用Redis。

# redis数据结构

## SDS（字符串）

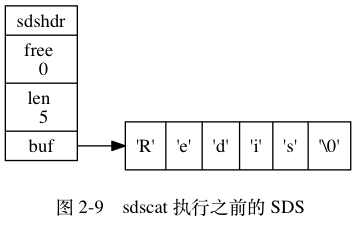
Redis在实现上使用了，自定义的SDS(simple dynamic string)，来代替C语言传统的字符串表示方式。

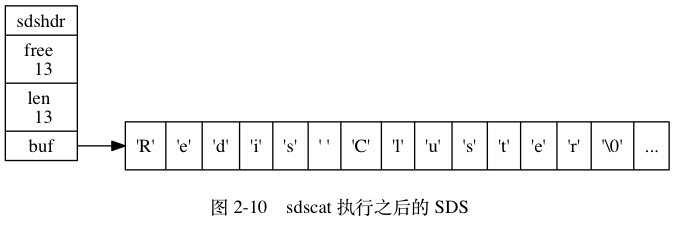
* free 属性的值为 0 ， 表示这个 SDS 没有分配任何未使用空间。
* len 属性的值为 5 ， 表示这个 SDS 保存了一个五字节长的字符串。
* buf 属性是一个 char 类型的数组， 数组的前五个字节分别保存了 'R' 、 'e' 、 'd' 、 'i' 、 's' 五个字符， 而最后一个字节则保存了空字符 '\0' 。



SDS对比C语言传统的字符串有以下优点：

* 在常数时间复杂度获取字符串长度：因为在SDS结构中已经保存了字符串的长度信息，并且在修改字符串时，也会对其进行相应的修改，所以获取字符串长度，无需遍历整个字符串。
* 杜绝缓存区溢出：在C字符串中如果你要对一个字符串进行扩展的话，就必须预先计算空间，如果空间不足久，就需要再申请一些内存空间。但要是忘记了预先分配空间的话，就会参数缓存区溢出。而使用SDS的API会自动的帮你计算并分配空间，从而杜绝了缓存区溢出的可能

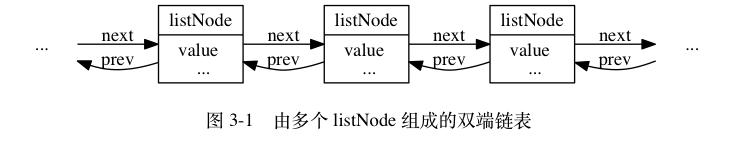




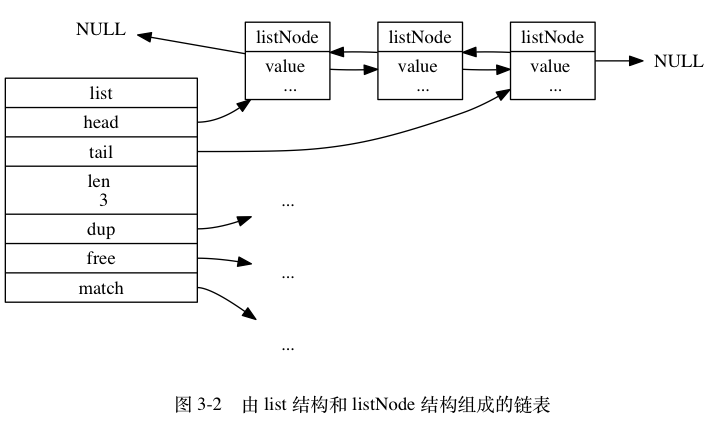
* 减少内存分配次数：相比于C字符串不记录长度信息，导致每次修改都需要进行内存重新分配(分配内存是一个相对耗时的系统调用) SDS就要灵活的多了，SDS的API通过 空间预先分配 和 惰性空间释放 这两个方式去减少空间的重新分配，其中空间预先分配 是指当第一次对SDS字符串进行修改时，SDS不止分配足够的空间，而且会根据分配策略，多分配一些空间以备下次使用。惰性空间释放就是在缩短SDS字符串时仅仅删除字符内容，并不回收剩余的内存空间。
* 二进制安全：SDS 的 API 都是二进制安全的（binary-safe）： 所有 SDS API 都会以处理二进制的方式来处理 SDS 存放在 buf 数组里的数据， 程序不会对其中的数据做任何限制、过滤、或者假设 —— 数据在写入时是什么样的， 它被读取时就是什么样。

## 链表

链表是Redis的列表键的底层实现之一。



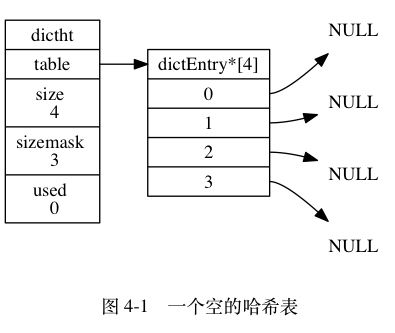
可以由上图结构看出，redis的链表底层是双端链表，并且由一个list结构表示，list结构为链表提供了表头指针 head 、表尾指针 tail ， 以及链表长度计数器 len ， 而 dup 、 free 和 match 成员则是用于实现多态链表所需的类型特定函数。



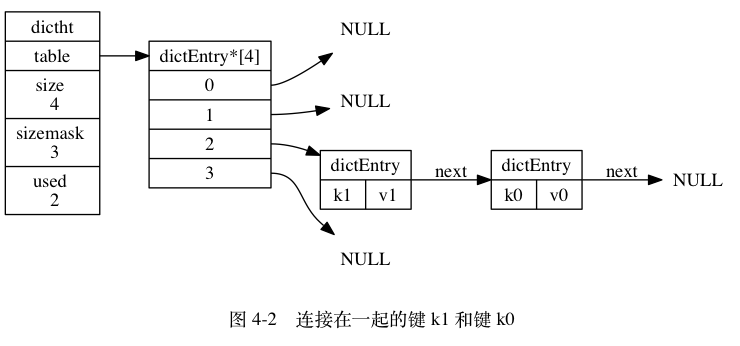
总结：redis 链表是一个无环的双端链表，并且通过list结构的表头和表尾指针，达到O(1)复杂度的首尾节点获取，利用len属性进行O(1)复杂度的链表长度获取。

## 字典

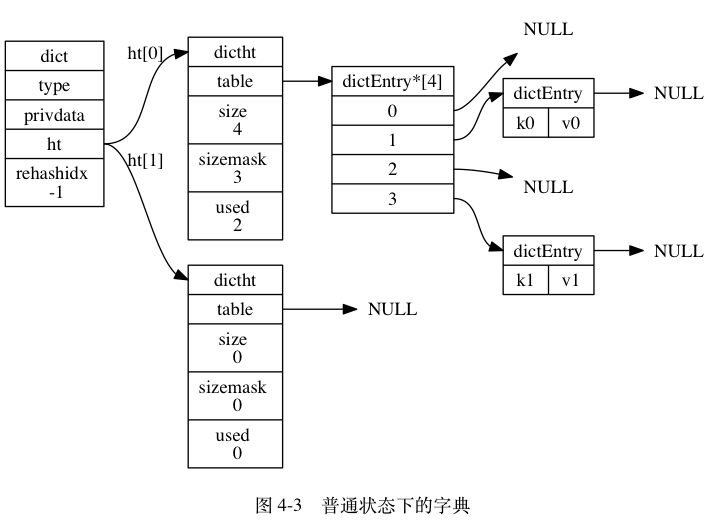
字典在Redis中被用于实现数据库本身和哈希键，当我们使用HSET,HGET的时候底层就是使用redis的字典实现。Redis的底层使用字典作为哈希表的实现，一个哈希表有多个哈希节点，每个节点保存一个键值对.**（单个hash结构）**



哈希表中，table属性是一个数组，size属性记录了哈希表的大小，也就是table数组的大小，而used属性则记录了哈希表目前已有节点（键值对）的数量。sizemask 属性的值总是等于 size - 1 ， 这个属性和哈希值一起决定一个键应该被放到 table 数组的哪个索引上面。



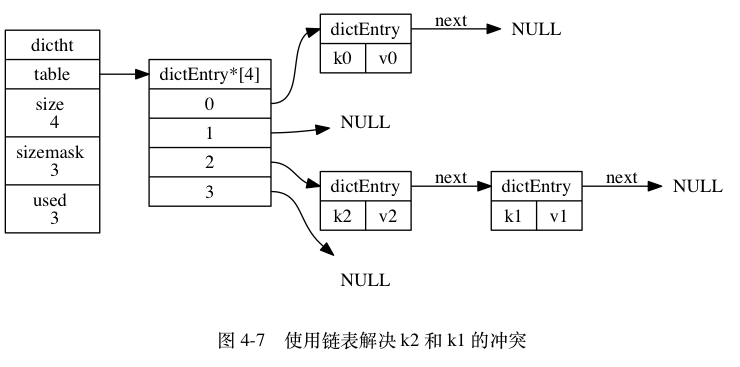
|  |
| --- |
| typedef struct dictEntry {  // 键  void \*key;  // 值  union {  void \*val;  uint64\_t u64;  int64\_t s64;  } v;  // 指向下个哈希表节点，形成链表  struct dictEntry \*next;  } dictEntry;  哈希节点结构，是由键指针(key)，联合体类型的值(v)，和指向下一个哈希节点的指针next组成，其中值可以64位有，无符号数，或是任意指针类型。next是为了实现链地址解决冲突而存在的。 |



* type 属性是一个指向 dictType 结构的指针， 每个 dictType 结构保存了一簇用于操作特定类型键值对的函数， Redis 会为用途不同的字典设置不同的类型特定函数。
* privdata 属性则保存了需要传给那些类型特定函数的可选参数。
* ht属性是一个只有两个元素的哈希表类型的数组，h[0]用于正在的数据，h[-1]用于Rehash的中间过渡。
* rehashidx属性，用于表示哈希表rehash的状态

### 哈希算法

Redis使用MurmurHash2算法来计算哈希值，每次有新的键值对要被添加到字典中，redis会先使用 MurmurHash2哈希函数来计算出哈希值，然后在用哈希值与哈希表结构中的sizemask进行与运算，最后就会得出键值对要存储的下标位置了。如上面的流程，当两个键值对最后计算出来的下标是一样的，那么Redis就会使用链地址法来解决冲突。



这里就使用到哈希节点中预留的next属性，指向下一个节点，后一个添加的节点，会被添加到头部，这样添加节点的复杂度就是O(1)了。

### rehash

rehash(重新散列），被用于保证字典的负载因子在一个平衡的范围内(1-5之间)，也就是需要根据哈希表的大小进行相应的扩展和收缩。rehash的过程：

1. 给字典ht[1]分配空间，分配空间大小的公式是$2^n$，其中在扩展时: n=ht[0].used \* 2, 在收缩时: n=ht[0].used。
2. 设置字典的rehashidx为0，开始rehash。
3. 每当对字典执行添加、删除、查找或者更新操作时，将rehashidx作为下标，对应在h[0]的节点取出，然后重新计算哈希值和下标，保存的对应的h[1]的位置上，最后自增字典的rehasidx属性。
4. 当h[0]中的所有节点都被rehash到h[1]后，释放h[0]的空间，将h[1]设置成h[0]，再为h[1]设置一个空白哈希表，并且将字典的rehashidx设置为-1。

在rehash期间，如果添加键值对到字典的操作，都会被直接存储在h[1]中，而查找的时候会先从h[0]找，然后再到h[1]，这样就保证了h[0]的元素只减无增。

当以下条件中的任意一个被满足时， 程序会自动开始对哈希表执行扩展操作：

* 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
* 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

## 跳跃表

跳跃表是有序集合的底层实现之一， 除此之外它在 Redis 中没有其他应用。

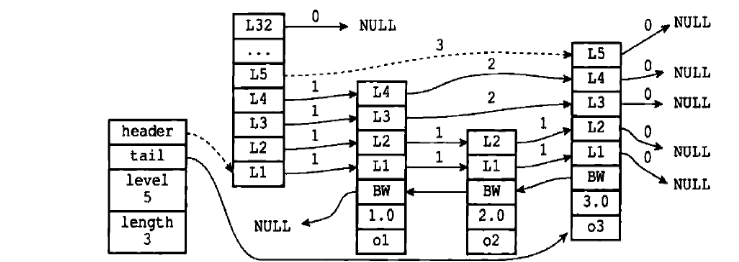
|  |
| --- |
| typedef struct zskiplist {  struct zskiplistNode \*header, \*tail;  unsigned long length;  int level;  } zskiplist;  typedef struct zskiplistNode {  robj \*obj;  double score;  struct zskiplistNode \*backward;  struct zskiplistLevel {  struct zskiplistNode \*forward;  unsigned int span;  } level[];  } zskiplistNode; |

Redis 的跳跃表实现由 zskiplist 和 zskiplistNode 两个结构组成， 其中 zskiplist 用于保存跳跃表信息（比如表头节点、表尾节点、长度）， 而 zskiplistNode 则用于表示跳跃表节点。

每个跳跃表节点的层高都是 1 至 32 之间的随机数。

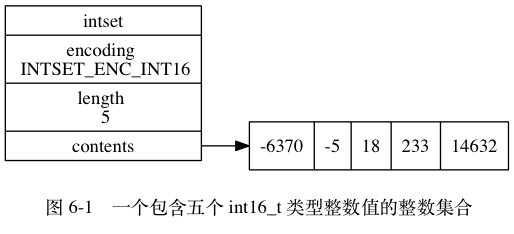
在同一个跳跃表中， 多个节点可以包含相同的分值， 但每个节点的成员对象必须是唯一的。

跳跃表中的节点按照分值大小进行排序， 当分值相同时， 节点按照成员对象的大小进行排序。



## 整数集合

整数集合是redis Set数据结构的实现之一，等Set中的数据只有整数的时候，就会使用它，整数集合可以保存，从16位到64位的整数，



* encoding 属性决定了，集合使用那种长度的int类型存储数据。
* length 记录总数
* cotents 数组按从小到大的顺序保存着整数数据，它的类型是由encoding属性决定的。

升级：如果一个新添加的元素的长度超过了 encoding类型的长度，那么整数集合就会自动执行升级操作，就是指，按照新元素的大小，重新分配一个数组空间，然后将原来contents中的内容转换成新的类型，在顺序存在新的数组中，最后将新元素存放在length-1处。

## 压缩列表

Redis会在当列表键的，元素是小整数，或是短字符串时，使用压缩列表作为底层实现，可见压缩列表是Redis为了节省内存而开发的。

# redis对象

Redis并没有直接使用如SDS，字典，整数集合等等的数据结构来实现键值对数据库，而是基于这些数据结构构建的对象系统，分别是

* 字符串对象
* 列表对象
* 哈希对象
* 集合对象
* 有序集合对象

Redis 的对象系统还实现了引用计数的垃圾回收器，并且Redis 还通过引用计数技术实现了对象共享机制， 这一机制可以在适当的条件下， 通过让多个数据库键共享同一个对象来节约内存。Redis 的对象带有访问时间记录信息， 该信息可以用于计算数据库键的空转时长， 在服务器启用了 maxmemory 功能的情况下， 空转时长较大的那些键可能会优先被服务器删除。

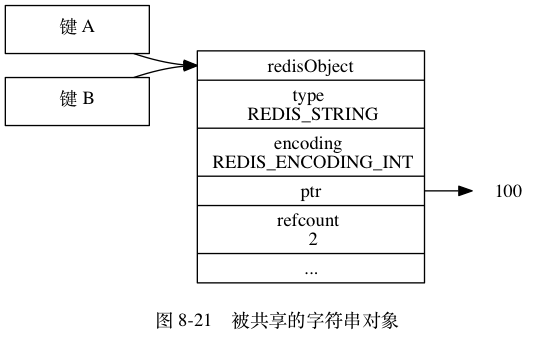
## 内存回收

redis 使用 redisObject结构中的refcount属性记录引用计数，当对象初始化的时候计数为1，每有一个新的引用就自增1，相反就自减1，到最后计数为0就会被回收。

|  |
| --- |
| typedef struct redisObject {  // ...  // 引用计数  int refcount;  // ...  } robj; |

## 对象共享

redis利用引用计数的功能实现了整数字符串对象的共享功能，（两个键指向同一个对象，引用计数加1）



Redis 会在初始化服务器时， 创建一万个字符串对象， 这些对象包含了从 0 到 9999 的所有整数值， 当服务器需要用到值为 0 到 9999 的字符串对象时， 服务器就会使用这些共享对象， 而不是新创建对象。

因为Redis共享对象需要先确认两个对象是否相同，字符串对象和哈希对象验证操作的复杂度为O(N)和O O(N^2)，而整数字符串是O(1)所以redis只共享整数字符串对象。

## 类型检查与命令多态

Redis中操作对象的命令分为两种分别是，多态命令和特定类型命令，Redis在执行特定类型命令是会先对操作值进行类型检查，如果配型不匹配的话就会直接返回错误。

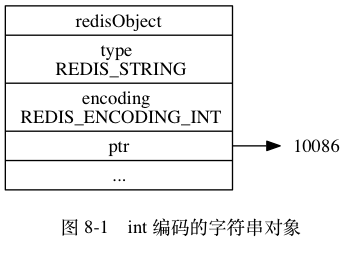
## 对象的编码与类型

Redis中的一个键值对有两部分组成，键对象和值对象，对象在底层都由redisObject结构存储。

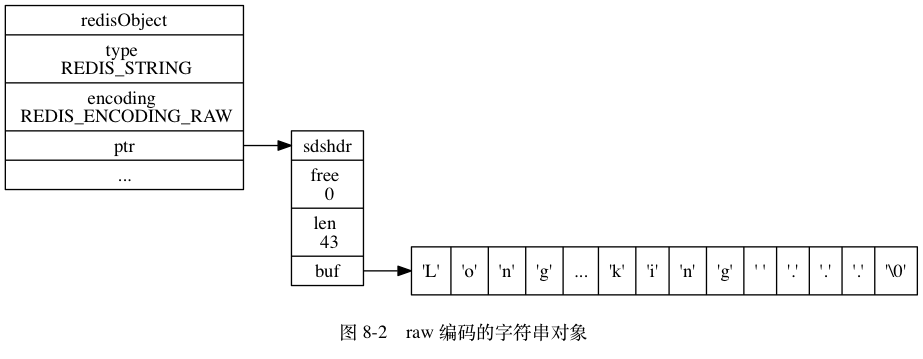
|  |
| --- |
| typedef struct redisObject {  // 类型  unsigned type:4;  // 编码  unsigned encoding:4;  // 指向底层实现数据结构的指针  void \*ptr;  // ...  } robj;  其中编码代表着ptr指针指向的结构的类型，就比如：同样是列表键对象，但是可以由压缩表，双端链表实现，encoding 就是决定到底使用那种数据结构去实现对象结构。 |

### 字符串对象

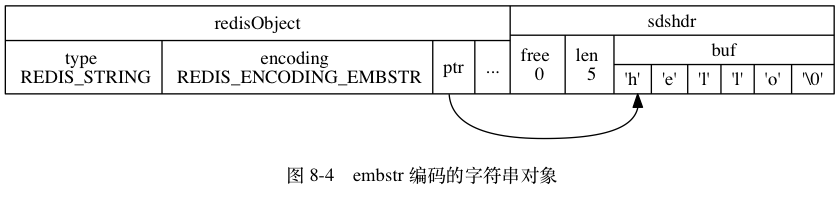
-- 编码方式：int, 转换条件：整数字符串 结构



-- 编码方式：raw, 转换条件：大于39字符的字符串值 结构



-- 编码方式：embstr, 转换条件：小于等于39个字符 结构

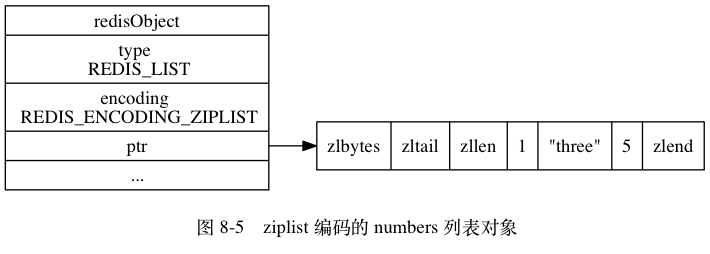


### 列表对象

-- 编码方式：ziplist

-- 转换条件：列表对象保存的所有字符串元素的长度都小于 64 字节，元素数量小于 512 个

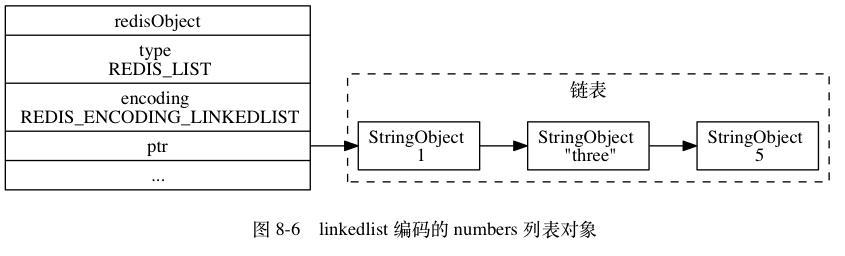
-- 结构



-- 编码方式：linkedlist

-- 转换条件： 相反

-- 结构

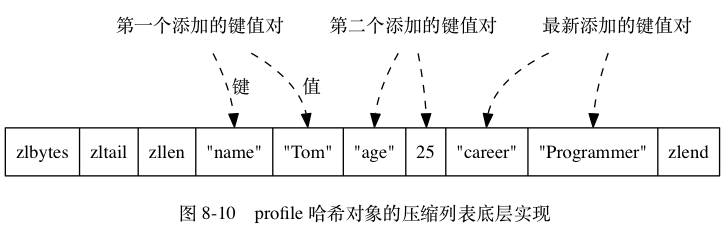


### 哈希对象

-- 编码方式：ziplist

-- 转换条件：列表对象保存的所有字符串元素的长度都小于 64 字节，元素数量小于 512 个

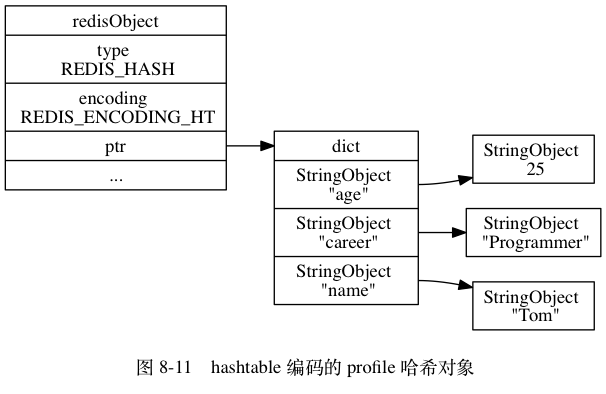
-- 结构



-- 编码方式：hashtable

-- 转换条件：相反

-- 结构

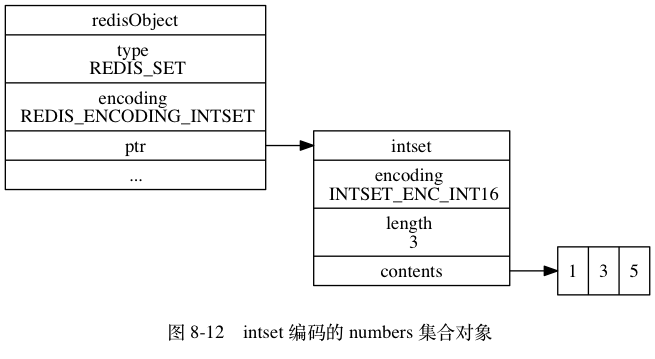


### 集合对象

-- 编码方式：intset

-- 转换条件：所有元素都是整数值，元素数量不超过 512 个

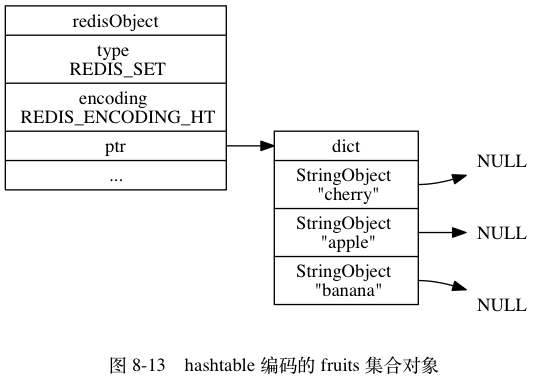
-- 结构



-- 编码方式：hashtable

-- 转换条件：相反

-- 结构

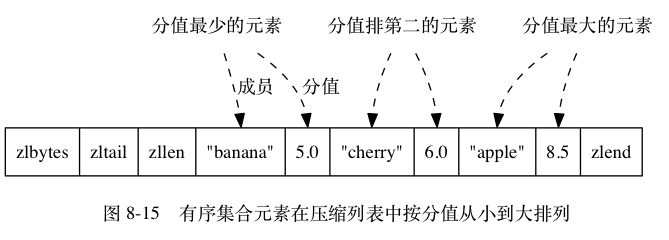


### 有序集合对象

-- 编码方式：ziplist

-- 转换条件：元素数量小于 128 个，所有元素成员的长度都小于 64 字节

-- 结构



-- 编码方式：hash

-- 转换条件：底层同时使用跳跃表和字典两种结构

-- 结构

