

一、底层数据结构

二、必知必会的铺垫

三、String数据结构

1、SDS

1.1、sds概念

1.2、sds干啥的？

1.3、为什么要有sds？

1.3.1、优化获取字符串长度

1.3.2、减少内存分配

1.3.3、惰性释放空间

1.3.4、防止缓冲区溢出

1.3.5、二进制安全

1.3.6、与C总结

2、int

3、embstr/raw

4、总结

四、list数据结构

1、ziplist

1.1、什么是ziplist？

1.2、ziplist结构

1.3、总结

2、linkedlist

3、ziplist和linkedlist如何选择？

4、quicklist

5、总结

五、Hash

1、ziplist

2、hashtable

2.1、基础原理

2.2、总结

3、rehash

3.1、什么是rehash？

3.2、什么时候需要rehash？

3.3、怎么rehash？

3.4、总结

4、ziplist和hashtable如何选择？

5、总结

六、Set

1、intset

2、hashtable

3、intset和hashtable如何选择？

4、总结

七、Zset

1、什么是zskiplist？

2、为什么Redis选择使用跳表而不是红黑树来实现有序集合？

3、总结

author：编程界的小学生

date：2021/03/18

一、底层数据结构

数据类型	底层数据结构
String	int/embstr/raw
List	3.0之前: ziplist/linkedlist; 3.0之后: quicklist
Hash	ziplist/hashtable
Set	intset/hashtable
Zset	ziplist/zskiplist

二、必知必会的铺垫

Redis对象，干嘛的？想象成对象头，不管你什么类型，都必须要带的，里面包含数据类型等等信息。

```
1  /*
2   * Redis 对象
3   */
4  typedef struct redisObject {
5      // 类型 4bits, 即上面[String,List,Hash,Set,Zset]中的一个
6      unsigned type:4;
7      // 编码方式 4bits, encoding表示对象底层所使用的编码。
8      unsigned encoding:4;
9      // LRU 时间（相对于 server.lruclock） 24bits
10     unsigned lru:22;
11     // 引用计数 Redis里面的数据可以通过引用计数进行共享 32bits
12     int refcount;
13     // 指向对象的值 64-bit
14     void *ptr;
15 } robj; // 16bytes
```

一个RedisObject占用的字节数： $4+4+24+32+64=128$ 位/8=16字节。

三、String数据结构

sds/int/embstr/raw

1、SDS

1.1、sds概念

sds (simple dynamic string)：简单动态字符串。SDS只是字符串类型中存储字符串内容的结构，Redis中的字符串分为两种存储方式，分别是embstr和raw。

sds中包含了free(当前可用空间大小)，len(当前存储字符串长度)，buf[] (存储的字符串内容)，来看下SDS的源码：

```

1 struct sdshdr{
2     //记录buf数组中已使用字节的数量
3     //等于 SDS 保存字符串的长度 4byte
4     int len;
5     //记录 buf 数组中未使用字节的数量 4byte
6     int free;
7     //字节数组，用于保存字符串 字节\0结尾的字符串占用了1byte
8     char buf[];
9 }

```

包含了len、free、buf[]三个属性。那么他占用字节最少是：4+4+1=9字节。（仅限redis3.2版本之前。Redis3.2版本之后的sds结构发生了变化。）

先剧透一个知识点：字符串长度如果小于39的话，则采取embstr存储，否则采取raw类型存储。

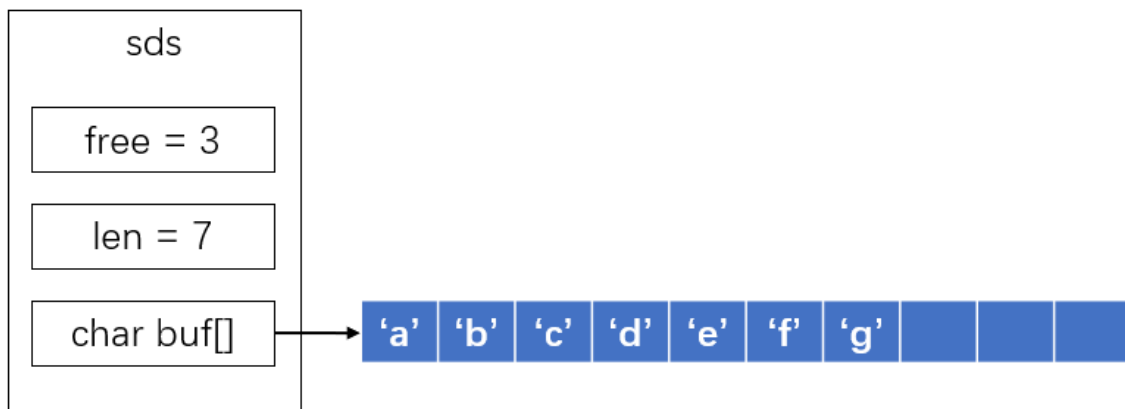
为啥是39？原因：对象头占16字节，空的sdshdr占用9字节，也就是一个数据至少占用16+9=25字节

其次操作系统使用jmalloc和tmalloc进行内存的分配，而内存分配的单位都是2的N次方，所以是2,4,8,16,32,64等字节，但是redis如果采取32的话，那么32-25=7，也太他妈少了，所以Redis采取的是64字节，所以：64-25=39。

1.2、sds干啥的？

比如你 set abc abcdefg，简单的一个set会创建出两个sds，一个存key: abc，一个存value: abcdefg。

比如如下



1.3、为什么要有sds？

带着问题看答案：C语言中也有字符串类型，为啥她不用C的，反正他都是C语言写的，为啥要造个轮子sds？

1.3.1、优化获取字符串长度

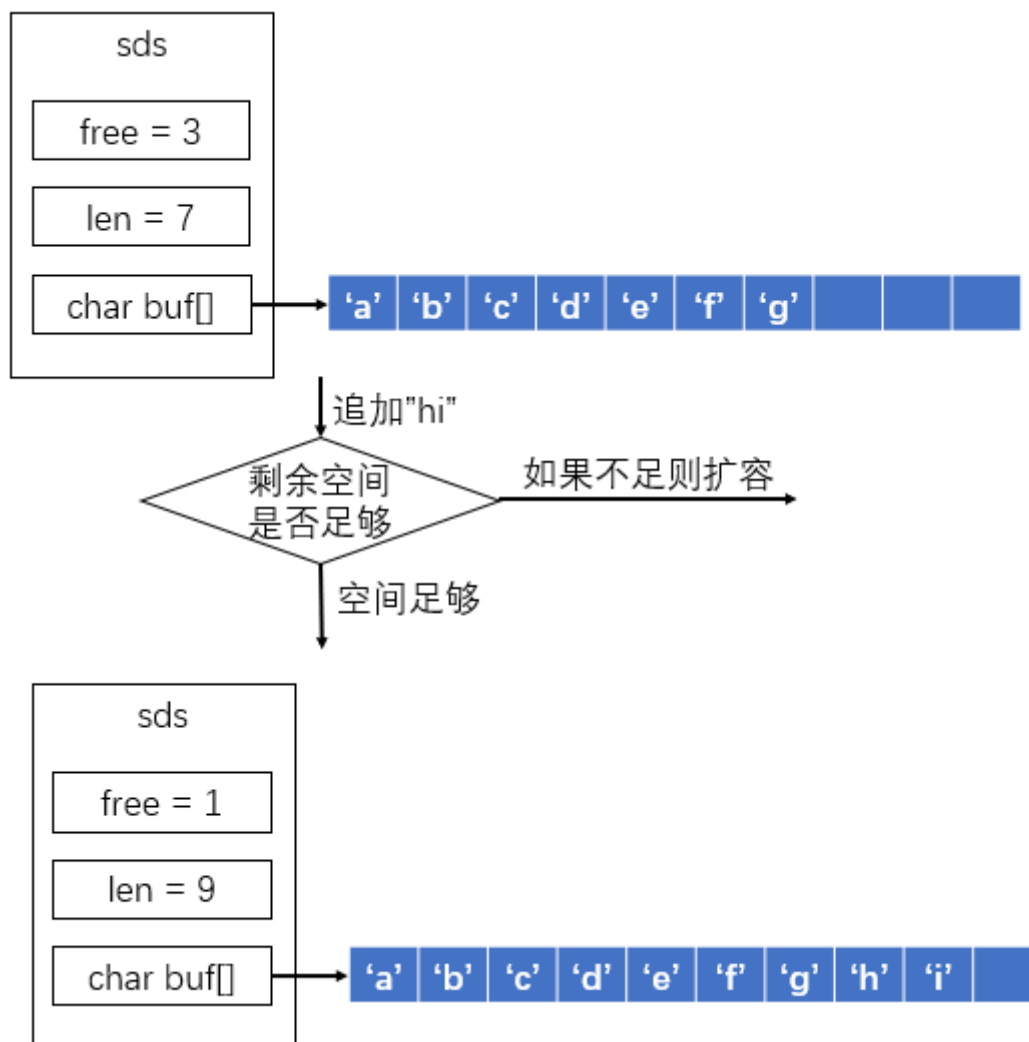
C语言要想获取字符串长度必须遍历整个字符串的每一个字符，然后自增做累加，时间复杂度为O(n)；sds直接维护了一个len变量，时间复杂度为O(1)。

1.3.2、减少内存分配

当我们对一个字符串类型进行追加的时候，可能会发生两种情况：

- 当前剩余空间(free)足够容纳追加内容时，我们就不需要再去分配内存空间，这样可以减少内存分配次数。
- 当前剩余空间不足以容纳追加内容，我们需要重新为其申请内存空间。

比如下面的sds的方式，free还有三个空余空间呢，你插入的是hi两个字符，所以足够，不需要调用函数重新分配，提升效率。



而C语言字符串在进行字符串的扩充和收缩的时候，都会面临着内存空间的重新分配问题。如果忘记分配或者分配大小不合理还会造成数据污染问题。

那么sds的free值哪来的呢？也就是字符串扩容策略

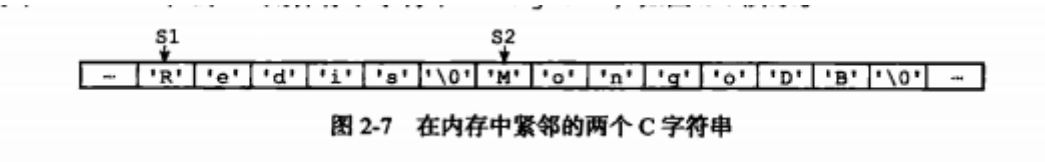
- 当给sds的值追加一个字符串，而当前的剩余空间不够时，就会触发sds的扩容机制。扩容采用了空间预分配的优化策略，即分配空间的时候：如果sds 值大小< 1M ,则增加一倍； 反之如果>1M，则当前空间加1M作为新的空间。
- 当sds的字符串缩短了，sds的buf内会多出来一些空间，这个空间并不会马上被回收，而是暂时留着以防再用的时候进行多余的内存分配。这个是惰性空间释放的策略

1.3.3、惰性释放空间

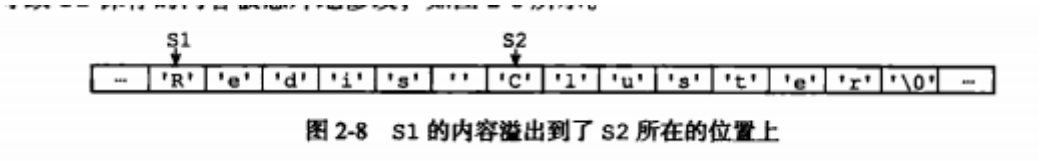
当我们截断字符串时，Redis会把截断部分置空，只保留剩余部分，且立即释放截断部分的内存空间，这样做的好处就是下次再对这个字符串追加内容的时候，如果当前剩余空间足以容纳追加内容时，就不需要再去重新申请空间，避免了频繁的内存申请。暂时用不上的空间可以被Redis定时删除或者惰性删除。

1.3.4、防止缓冲区溢出

其实和减少内存分配是成套的，都是因为sds预先检查内存自动分配来做到防止缓冲区溢出的。比如：程序中有两个在内存中紧邻着的字符串 s1 和 s2，其中s1 保存了字符串“redis”，s2 则保存了字符串“MongoDb”：



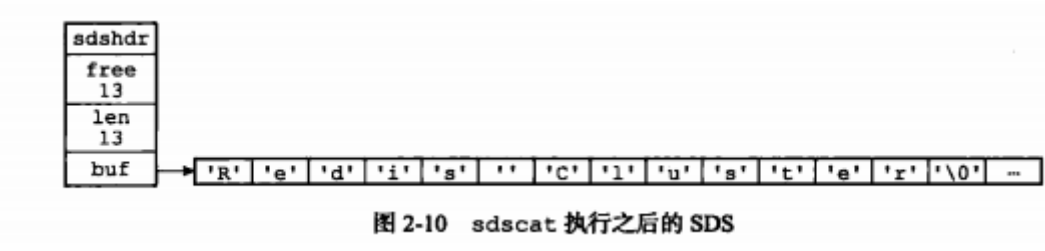
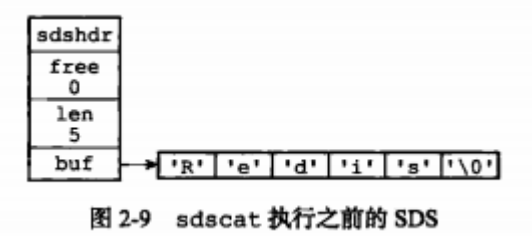
如果我们现在将s1 的内容修改为redis cluster，但是又忘了重新为s1 分配足够的空间，这时候就会出现以下问题：



我们可以看到，原本s2 中的内容已经被s1的内容给占领了，s2 现在为 cluster，而不是“Mongodb”。造成了缓冲区溢出，也是数据污染。

Redis中SDS的空间分配策略完全杜绝了发生缓冲区溢出的可能性：

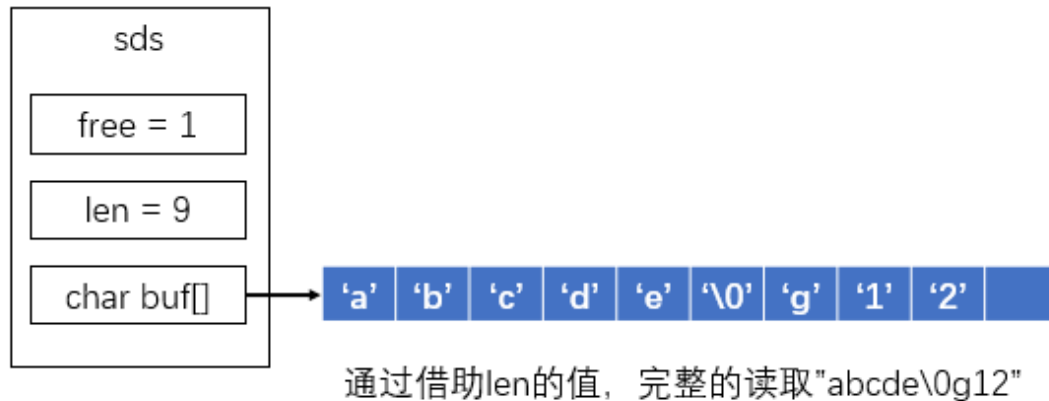
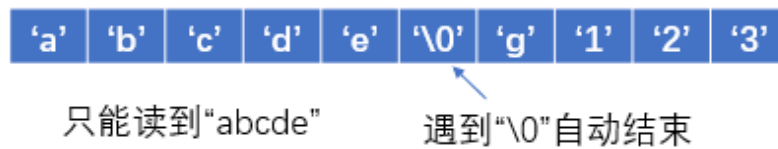
当我们需要对一个SDS 进行修改的时候，redis 会在执行拼接操作之前，预先检查给定SDS 空间是否足够，如果不够，会先拓展SDS 的空间，然后再执行拼接操作



1.3.5、二进制安全

在C语言中通过判断当前字符是否为'\0'来确定字符串是否结束，而在sds结构中，只要遍历长度没有达到len，即使遇到'\0'，也不会认为字符串结束。比如下面内存，C语言的字符串类型会丢失g123这四个字符，因为他遇到'\0'就结束了，而sds不会存在此问题。

C语言字符串



1.3.6、与C总结

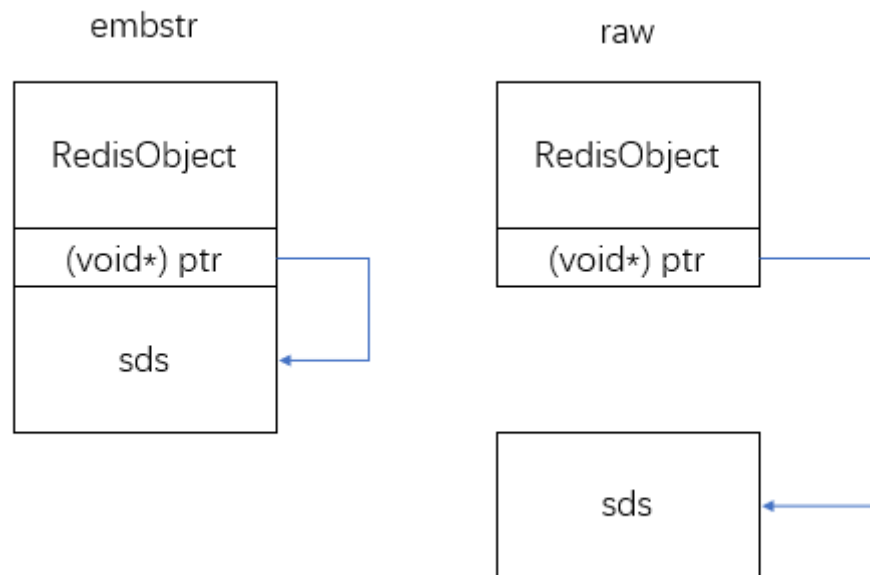
C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度N次 必然需要 执行N次内存重分配	修改字符串长度N次 最多执行 N次内存重分配
只能保存文本数据，二进制不安全	可以保存二进制数据和文本文数据，二进制安全

2、int

如果一个字符串内容可转为 long，那么该字符串会被转化为 long 类型，redisObject的对象 ptr 指向该 long，并将 encoding 设置为 int，这样就不需要重新开辟空间，算是长整形的一个优化。

3、embstr/raw

上面的SDS只是字符串类型中存储字符串内容的结构，Redis中的字符串分为两种存储方式，分别是 embstr和raw，当字符串长度特别短（redis3.2之前是39字节，redis3.2之后是44字节）的时候，Redis使用embstr来存储字符串，而当字符串长度超过39（redis3.2之前）的时候，就需要用raw来存储，下面是他们的字符串完整结构的示意图：



embstr的存储方式是将RedisObject对象头和SDS结构放在内存中连续的空间位置，也就是使用malloc方法一次分配，而raw需要两次malloc，分别分配对象头和SDS的空间。释放空间也一样，embstr释放一次，raw释放两次，所以embstr是一种优化，但是为什么是39字节才采取embstr呢？39哪来的？

这个问题在上面sds里已经说过了。

原因：对象头占16字节，空的sdshdr占用9字节，也就是一个数据至少占用16+9=25字节。

其次操作系统使用jmalloc和tmalloc进行内存的分配，而内存分配的单位都是2的N次方，所以是2,4,8,16,32,64等字节，但是redis如果采取32的话，那么32-25=7，也太他妈少了，所以Redis采取的是64字节，所以：64-25=39。

(仅限redis3.2版本之前。Redis3.2版本之后的sds结构发生了变化【最小的是sdshdr5，空的话占用3字节+1个空白=4字节，16+4=20；64-20=44】。)

4、总结

1. redis的string底层数据结构使用的是sds，但是sds有两种存储方式，一种是embstr，一种是raw。
2. embstr的优势在于和对象头一起分配到连续空间，只需要调用函数malloc一次就行。raw需要两次，一次是对象头，一次是sds。释放也一样，embstr释放一次，raw释放两次。
3. 字符串内容可转为 long，采用 int 类型，否则长度<39（3.2版本前是39，3.2版本后分界线是44）用 embstr，其他用 raw。
4. SDS 是Redis自己构建的一种简单动态字符串的抽象类型，并将 SDS 作为 Redis 的默认字符串表示。
5. SDS 与 C 语言字符串结构相比，具有四大优势。

四、list数据结构

Redis3.0之前：ziplist/linkedlist

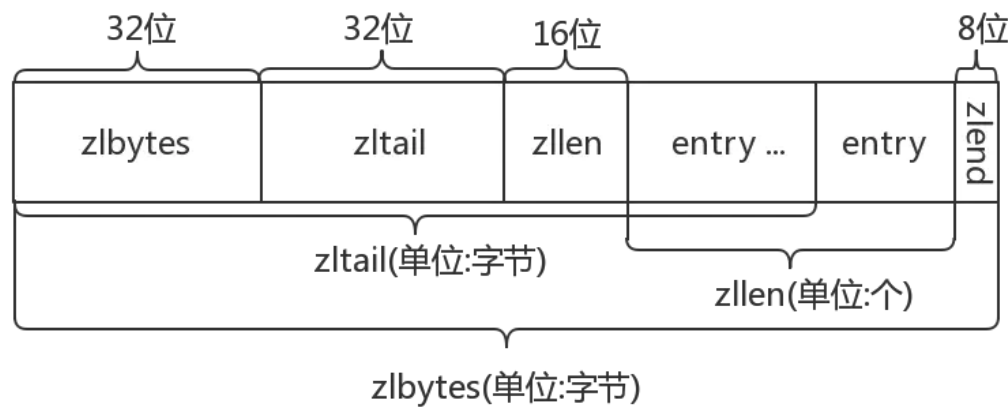
Redis3.0之后：quicklist

1、ziplist

1.1、什么是ziplist?

ziplist是一个经过特殊编码的双向链表，它的设计目标就是为了提高存储效率。

1.2、ziplist结构



- **zlbytes**: ziplist的长度，32位无符号整数。
- **zltail**: ziplist最后一个节点的偏移量，反向遍历ziplist或者pop尾部节点的时候用来提升性能。
- **zllen**: ziplist的entry（节点）个数。
- **entry**: 节点，并不是一个数组，然后里面存的值。而是一个数据结构。下面说。
- **zlend**: 值为255，用于标记ziplist的结尾。

entry的布局

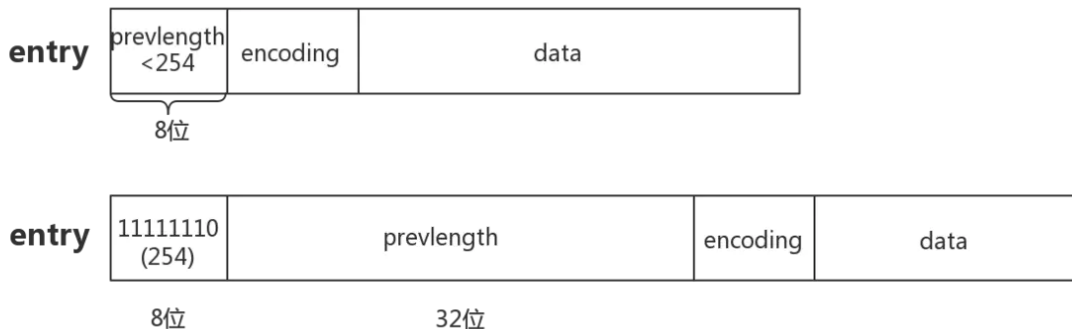
由三部分组成：

- **prevlengh**: 记录上一个节点的长度，为了方便反向遍历ziplist。
- **encoding**: 当前的编码规则，记录了节点的content属性所保存数据类型以及长度。
- **data**: 保存节点的值。可以是字符串或者数字，值的类型和长度由encoding决定。

如果前一节点的长度小于254字节，那么 `previous_entry_ength` 属性的长度为1字节，前一节点的长度就保存在这一个字节里面。

如果前一个节点的长度大于等于254，那么 `previous_entry_ength` 属性的长度为5字节，其中属性的第一字节会被设置为0xFE（十进制254），而之后的四个字节则用于保存前一节点的长度。**用254 不用255(11111111)作为分界是因为255是zlend的值，它用于判断ziplist是否到达尾部。**

利用此原理即当前节点位置减去上一个节点的长度即得到上一个节点的起始位置，压缩列表可以从尾部向头部遍历，这么做很有效地减少了内存的浪费。



1.3、总结

- ziplist是为节省内存空间而生的。
- ziplist是一个为Redis专门提供的底层数据结构之一，本身可以有序也可以无序。当作为list和hash的底层实现时，节点之间没有顺序；当作为zset的底层实现时，节点之间会按照大小顺序排列。
- ziplist的弊端也很明显了，对于较多的entry或者entry长度较大时，需要大量的连续内存，并且节省的空间比例相对不在占优势，就可以考虑使用其他结构了。

2、linkedlist

就是双向链表，对首尾节点的定位很快，O(1)复杂度。在首位前后插入节点也很是O(1)。

3、ziplist和linkedlist如何选择？

Redis的list类型什么时候会使用ziplist编码，什么时候又会使用linkedlist编码呢？

当列表对象可以同时满足下列两个条件时，列表对象采用ziplist编码，否则采用linkedlist编码。

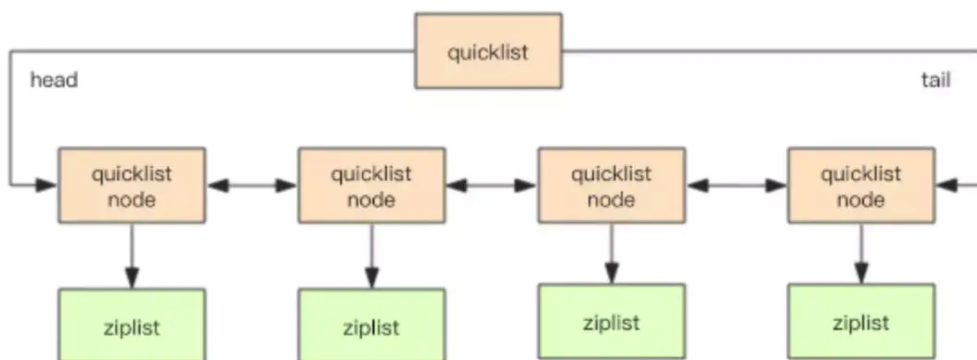
- (1) 列表对象保存的所有字符串元素的长度都小于64字节；
- (2) 列表元素保存的元素数量小于512个；

上述两个参数可以更改配置进行自定义。

4、quicklist

Redis3.0版本开始对list数据结构采取quicklist了，抛弃了之前的ziplist和linkedlist。

quicklist 是一个双向链表，并且是一个ziplist的双向链表，也就是说一个quicklist由多个quicklistNode组成，每个quicklistNode指向一个ziplist，一个ziplist包含多个entry元素，每个entry元素就是我们push的list的元素。ziplist本身也是一个能维持数据项先后顺序的列表，而且数据项保存在一个连续的内存块中。意味着quicklist结合了ziplist和linkedlist的特点！更为优化了，还省去了ziplist和linkedlist之间转换的步骤了。



5、总结

linkedlist:

- 1.双端链表便于在表的两端进行push和pop操作，但是它的内存开销比较大；
- 2.双端链表每个节点上除了要保存数据之外，还要额外保存两个指针(pre/next)；
- 3.双端链表的各个节点是单独的内存块，地址不连续，节点多了容易产生内存碎片；

ziplist:

- 1.ziplist由于是一整块连续内存，所以存储效率很高；
- 2.ziplist不利于修改操作，每次数据变动都会引发一次内存的realloc；
- 3.当ziplist长度很长的时候，一次realloc可能会导致大批量的数据拷贝，进一步降低性能；

quicklist:

- 1.空间换时间，之前linkedlist需要两个指针，浪费空间，我现在不用linkedlist，我都采取ziplist，然后上面封装一层quicklistnode，底层存储还是ziplist，只是空间上多了一层指针用于检索。
- 2.结合了双端链表和压缩列表的优点。

五、Hash

ziplist/hashtable

1、ziplist

ziplist在上面list结构里介绍了。这里只说下hash是怎么用ziplist的。

ziplist 编码的哈希对象使用压缩列表作为底层实现，每当有新的键值对要加入到哈希对象时，程序会先将保存了键的压缩列表节点推入到压缩列表表尾，然后再将保存了值的压缩列表节点推入到压缩列表表尾，因此保存了同一键值对的两个节点总是紧挨在一起，保存键的节点在前，保存值的节点在后；先添加到哈希对象中的键值对会被放在压缩列表的表头方向，而后来添加到哈希对象中的键值对会被放在压缩列表的表尾方向。

例如，我们执行以下 HSET 命令，那么服务器将创建一个列表对象作为 profile 键的值：

```
1 redis> HSET profile name "Tom"(integer) 1
2 redis> HSET profile age 25(integer) 1
3 redis> HSET profile career "Programmer"(integer) 1
```

profile 键的值对象使用的是 ziplist 编码，其中对象所使用的压缩列表结构如下图所示：

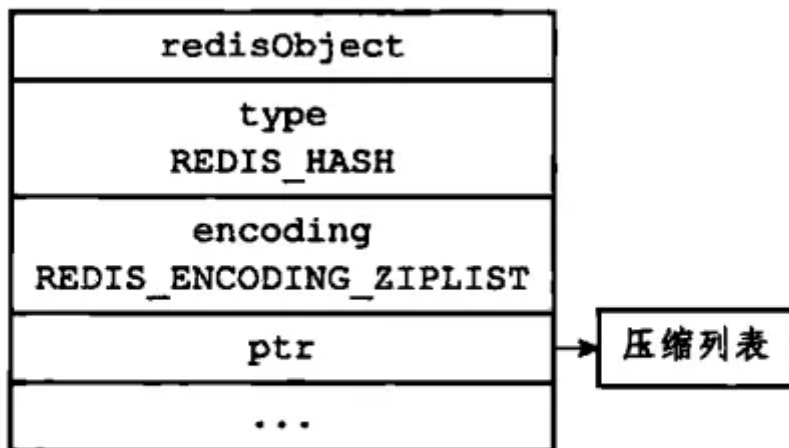


图 8-9 ziplist 编码的 profile 哈希对象

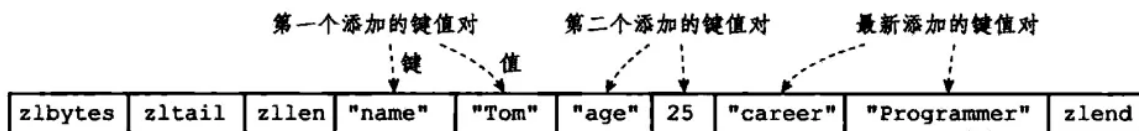


图 8-10 profile 哈希对象的压缩列表底层实现

2、hashtable

2.1、基础原理

类比hashmap，哈希对象中的每个键值对都使用一个字典键值对来保存：

- 字典的每个键都是一个字符串对象，对象中保存了键值对的键。
- 字典的每个值都是一个字符串对象，对象中保存了键值对的值。

比如上面ziplist的hset案例如果用hashtable来存储的话长下面这个样子：

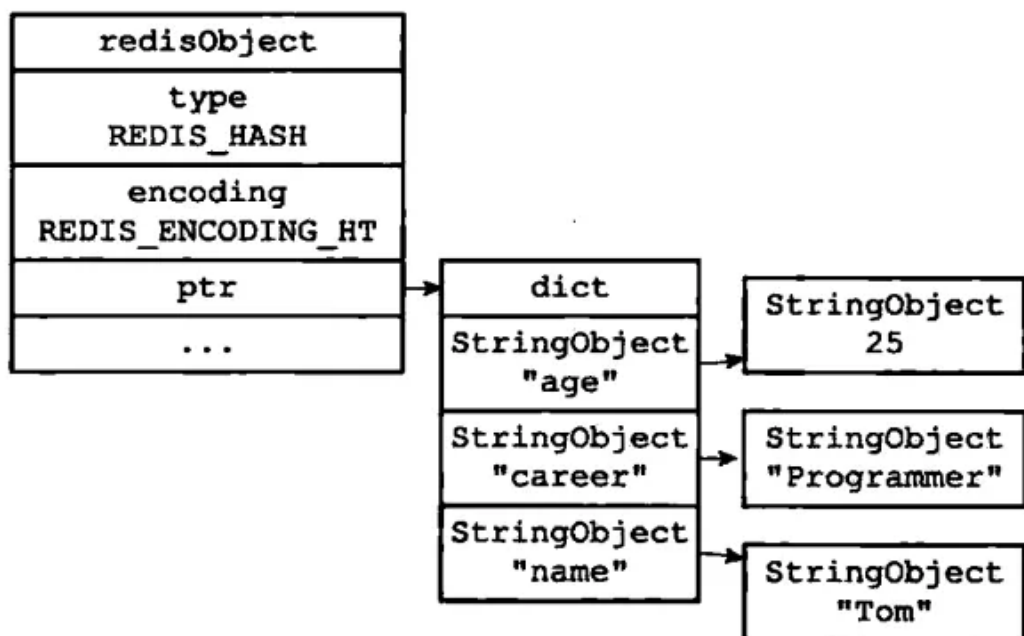



图 8-11 hashtable 编码的 profile 哈希对象  大道七哥

更为详细点的如下：

hashtable的结构是：dict指向dictht，dictht包含多个dictEntry，dictEntry包含next指针，指向下一个entry，形成一个链表，key冲突的话就会形成链表（理解成hashmap里的hash碰撞）。如下：

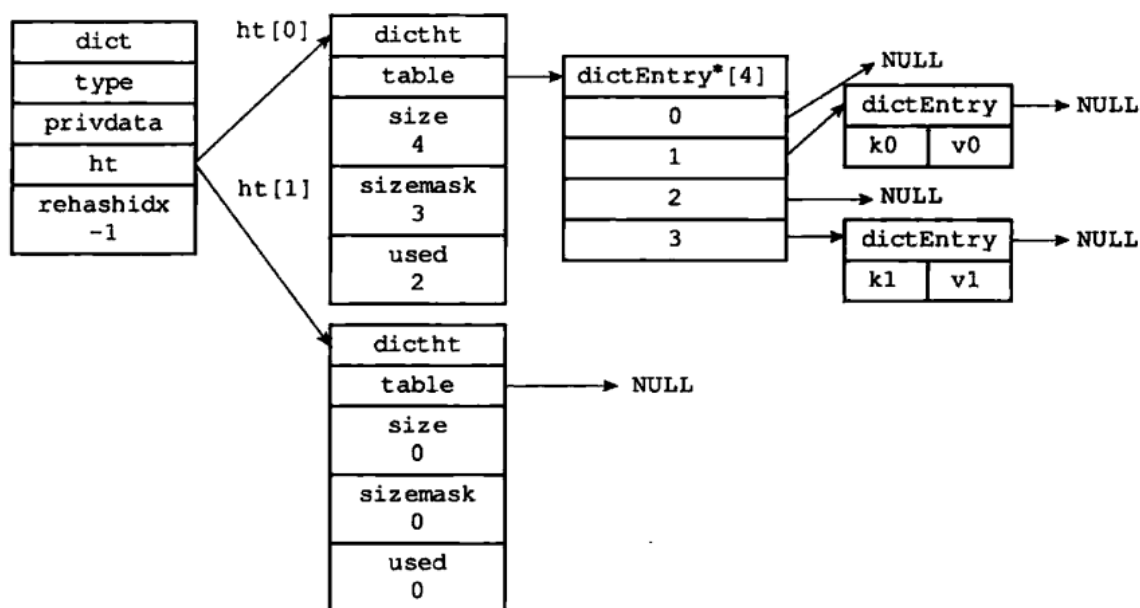



图 4-3 普通状态下的字典  大道七哥

可以看到dict包含ht，ht是个数组，包含ht[0]和ht[1]两部分。ht[0]是我们数据真实存储的地方，ht[1]是为了伸缩容量时候进行rehash用的。目前没有rehash，所以指向null。

key冲突的图示如下(在dictEntry下标为2的位置发生了key冲突，采取链表的方式解决)：

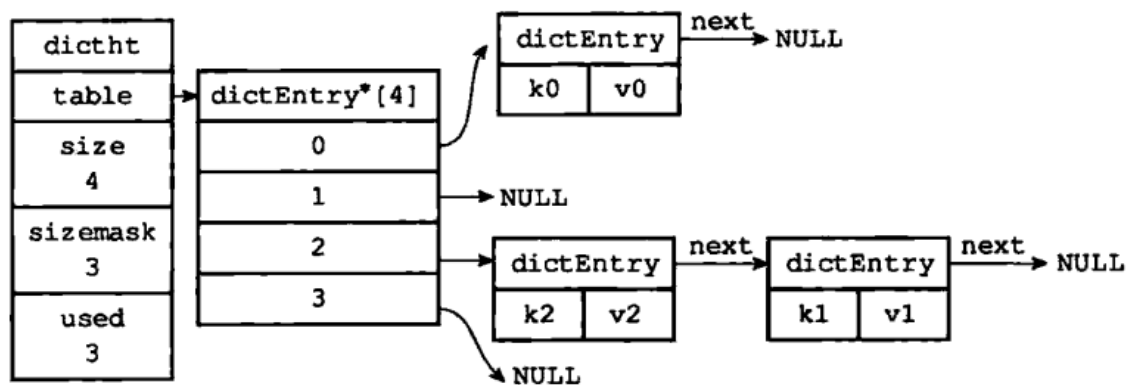


图 4-7 使用链表解决 k2 和 k1 的冲突

大道七哥

因为 dictEntry 节点组成的链表没有指向链表表尾的指针，所以为了速度考虑，程序总是将新节点添加到链表的表头位置（复杂度为 $O(1)$ ），排在其他已有节点的前面。

2.2、总结

- 字典 ht 属性是包含两个哈希表项的数组，一般情况下，字典只使用 ht[0]，ht[1] 哈希表只会在对 ht[0] 哈希表进行 rehash 时使用
- 哈希表使用链表形式来解决键冲突
- 键值对添加到字典的过程，先根据键值对的键计算出哈希值和索引值，然后再根据索引值，将包含新键值对的哈希表节点放到哈希表数组的指定索引上面

3、rehash

3.1、什么是rehash？

就是 hashtable 容量需要进行伸缩容。

3.2、什么时候需要rehash？

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：

- (1) 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；
- (2) 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；

其中哈希表的负载因子可以通过公式计算：

$$1 \quad \text{负载因子} = \text{哈希表已保存节点数量} / \text{哈希表大小} \quad \text{load_factor} = \text{ht}[0].\text{used} / \text{ht}[0].\text{size}$$

3.3、怎么rehash？

1. 为字典 (dict) 的 ht[1] 哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及 ht[0] 当前包含的键值对数量（也即是 ht[0].used 属性的值）：

- 1) 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 $\text{ht}[0].\text{used} * 2$ 的 2^n （ 2 的 n 次方幂）；
- 2) 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 $\text{ht}[0].\text{used}$ 的 2^n ；

- 2. 将保存在 ht[0] 中的所有键值对 rehash 到 ht[1] 上面：rehash 指的是重新计算键的哈希值和索引值，然后将键值对放置到 ht[1] 哈希表的指定位置上。
- 3. 当 ht[0] 包含的所有键值对都迁移到了 ht[1] 之后（ht[0] 变为空表），释放 ht[0]，将 ht[1] 设置为 ht[0]，并在 ht[1] 新创建一个空白哈希表，为下一次 rehash 做准备。

但是整个rehash的过程是渐进式的，因为如果几十万条记录一次性rehash的话，扛不住，浪费性能，所以演进除了渐进式rehash。

以下是哈希表渐进式 rehash 的详细步骤：

- 1. 为 ht[1] 分配空间，让字典同时持有 ht[0] 和 ht[1] 两个哈希表。
- 2. 在字典中维持一个索引计数器变量 rehashidx，并将它的值设置为 0，表示 rehash 工作正式开始。
- 3. 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 属性的值增一。
- 4. 随着字典操作的不断执行，最终在某个时间点上，ht[0] 的所有键值对都会被 rehash 至 ht[1]，这时程序将 rehashidx 属性的值设为 -1，表示 rehash 操作已完成。

渐进式 rehash 的好处在于它采取分而治之的方式，将 rehash 键值对所需的计算工作均摊到对字典的每个增删改查操作上，从而避免了集中式 rehash 而带来的庞大计算量。

图示：

- 1. 准备开始rehash

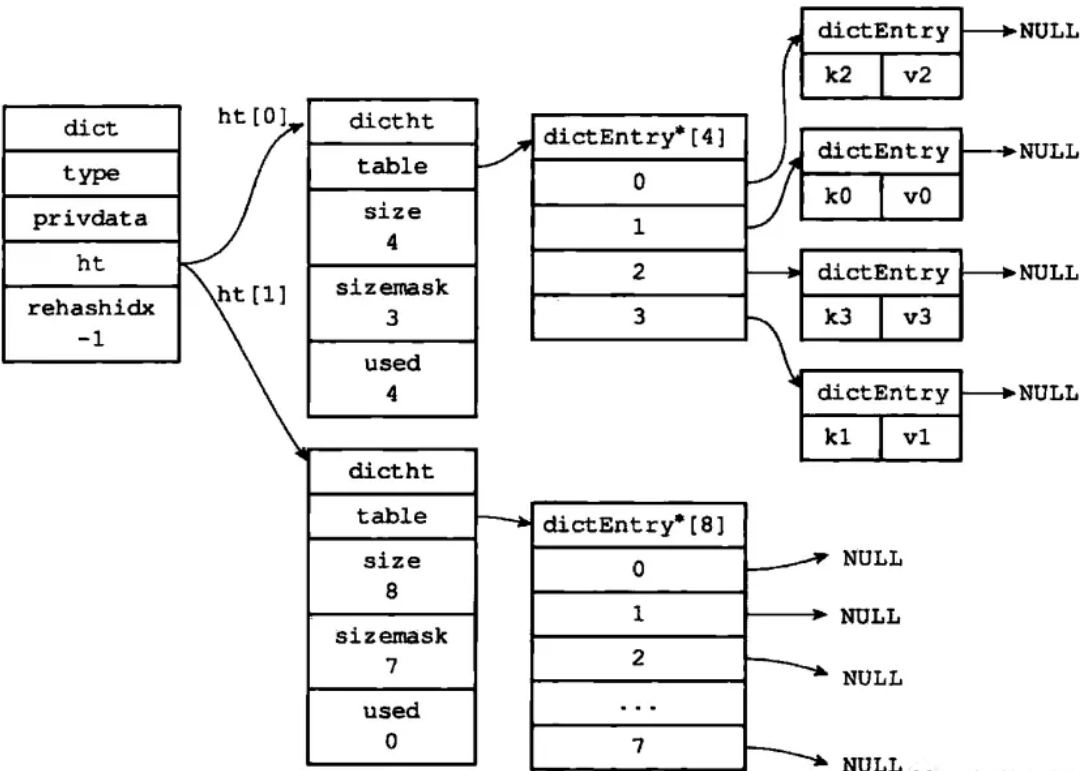


图 4-12 准备开始 rehash

- 2. rehash索引为0的数据

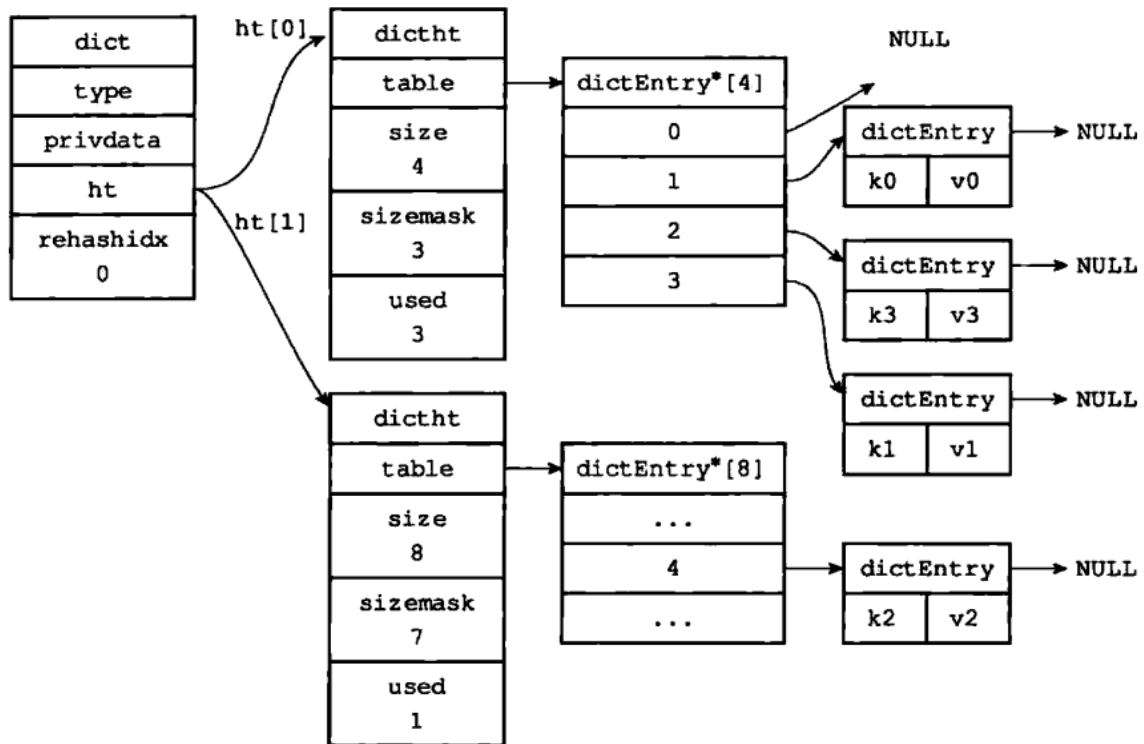


图 4-13 rehash 索引 0 上的键值对

大道七哥

3. rehash索引为1的数据

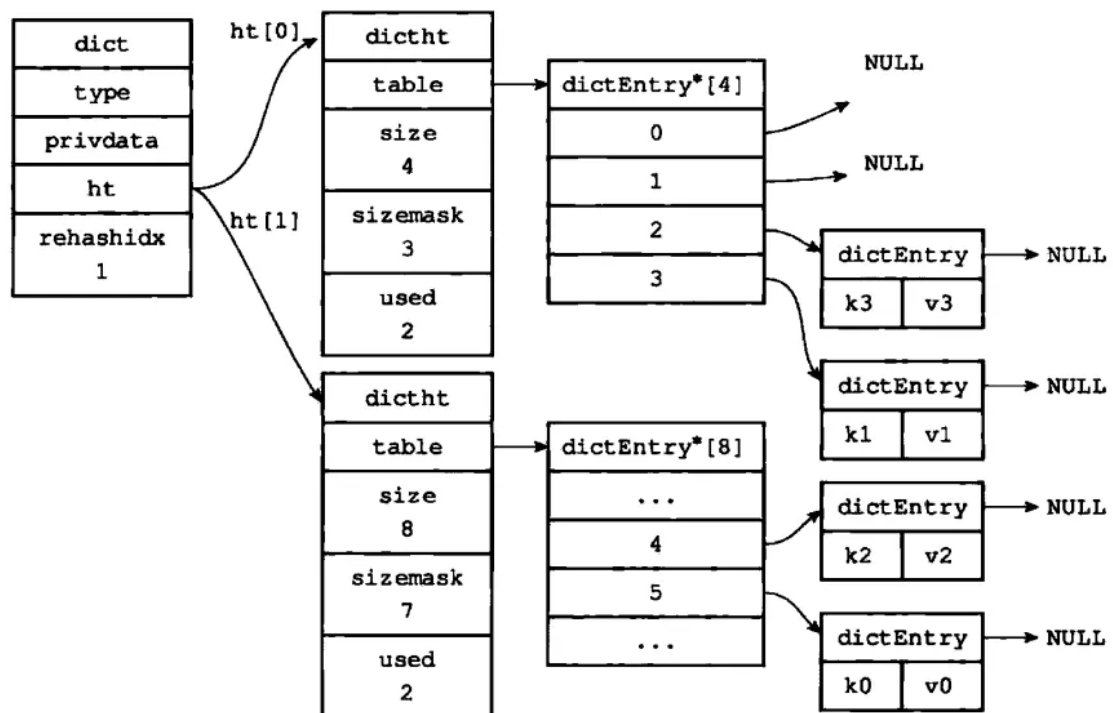


图 4-14 rehash 索引 1 上的键值对

大道七哥

4. rehash完毕 (跳过了dictEntry下标为2和3上的数据)

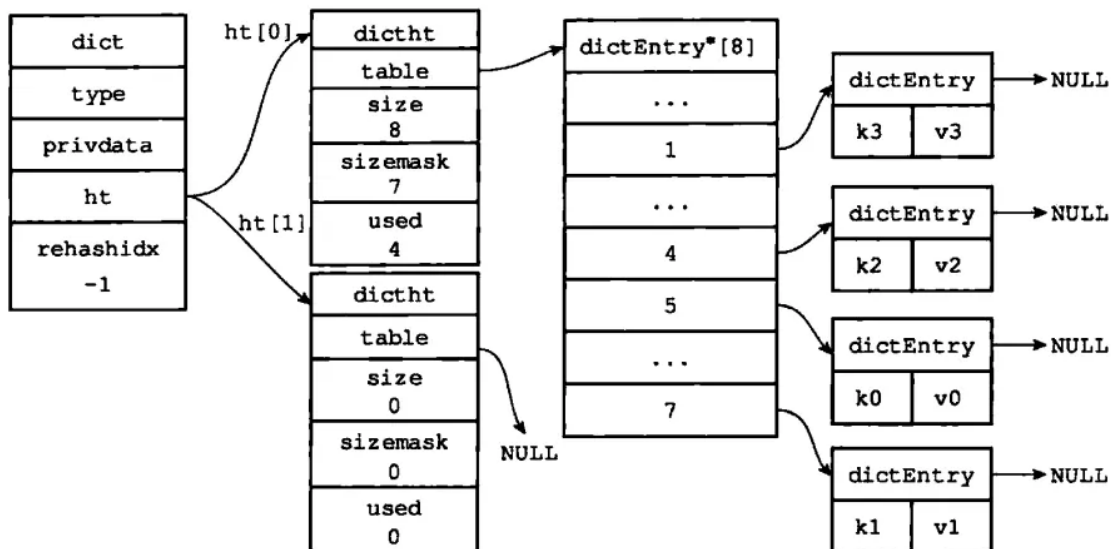


图 4-17 rehash 执行完毕

大道七哥

因为在进行渐进式 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，字典的删改查（没有增）等操作会在两个哈希表上进行：比如说，要在字典里面查找一个键的话，程序会先在 ht[0] 里面进行查找，如果没找到的话，就会继续到 ht[1] 里面进行查找，诸如此类。

另外，在渐进式 rehash 执行期间，新添加到字典的键值对一律会被保存到 ht[1] 里面，而 ht[0] 则不再进行任何添加操作：这一措施保证了 ht[0] 包含的键值对数量会只减不增，并随着 rehash 操作的执行而最终变成空表。

3.4、总结

1. 字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个用于平时使用，另一个仅在进行 rehash 时使用
2. 当哈希表保存的键值对数量太多或者太少时，程序需要对哈希表的大小进行相应的扩展或者收缩 (rehash)
3. rehash 动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的
4. 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

4、ziplist和hashtable怎么选？

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，不能满足这两个条件的哈希对象需要使用 hashtable 编码。

- 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。
- 哈希对象保存的键值对数量小于 512 个。

上述两个参数可以更改配置进行自定义。

5、总结

- hash采取ziplist压缩列表和hashtable字典来实现。
- hashtable伸缩容会发生rehash
- rehash过程是渐进式的

六、Set

1、intset

intset编码的集合对象使用**整数集**作为底层实现，集合对象包含的所有元素都被保存在**整数集**里面。

比如：

```
1 redis> SADD numbers 1 3 5
2 (integer) 3
```

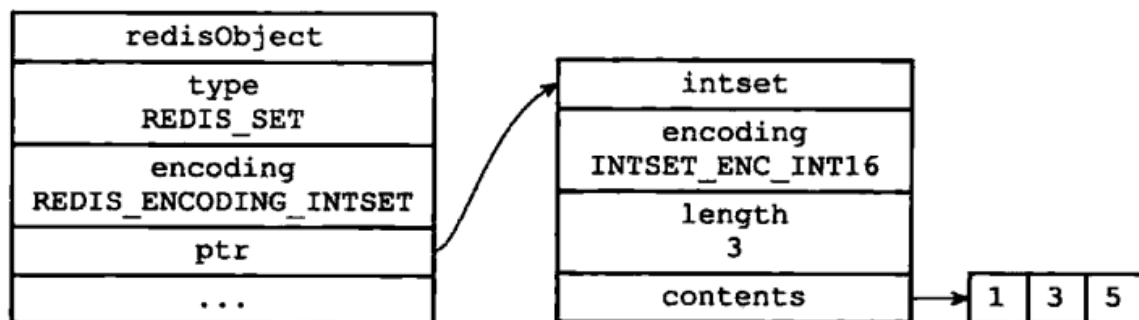


图 8-12 intset 编码的 numbers 集合对象

程序员油七

2、hashtable

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。

```
1 redis> SADD fruits "apple" "banana" "cherry"
2 (integer) 3
```

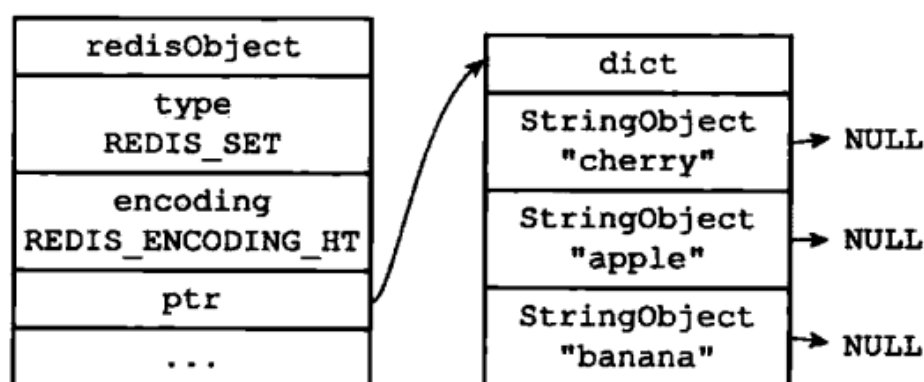


图 8-13 hashtable 编码的 fruits 集合对象

程序员油七

3、intset和hashtable怎么选择？

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，不能满足这两个条件的集合对象需要使用 hashtable 编码。

- 集合对象保存的所有元素都是整数值。
- 集合对象保存的元素数量不超过 512 个。

第二个条件是可以配置自定义的。

4、总结

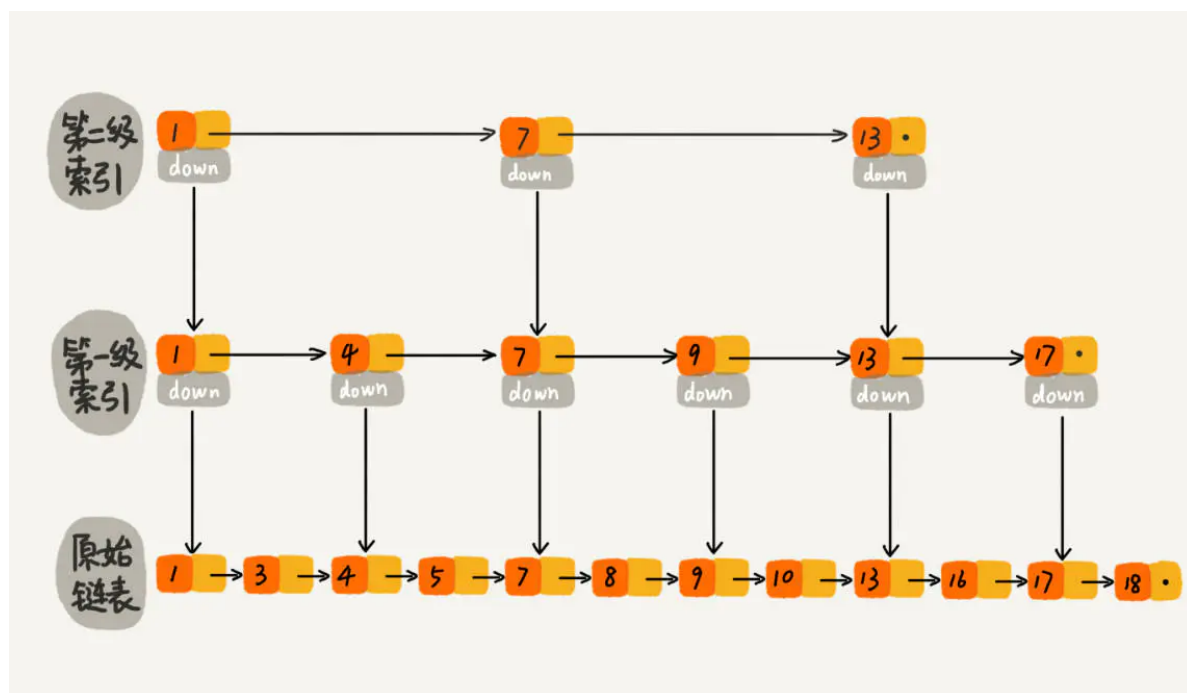
- (1) 集合对象的编码可以是 intset 或者 hashtable 。
- (2) intset 编码的集合对象使用**整数集合**作为底层实现。
- (3) hashtable 编码的集合对象使用**字典**作为底层实现。
- (4) intset 与 hashtable 编码之间，符合条件的情况下，可以转换。

七、Zset

ziplist/zskiplist

1、什么是zskiplist?

跳跃表，一种**可以实现二分查找的有序链表**。使普通链表的查找时间复杂度由 $O(N)$ 变为平均 $O(\log N)$ ，最坏 $O(N)$ 。原理是通过多级辅助索引进行分层，如下图：



2、为什么Redis选择使用跳表而不是红黑树来实现有序集合?

Redis 中的有序集合(zset) 支持的操作：

1. 插入一个元素
2. 删除一个元素
3. 查找一个元素
4. 有序输出所有元素
5. 按照范围区间查找元素（比如查找值在 [100, 356] 之间的数据）

其中，前四个操作红黑树也可以完成，且时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。按照区间查找数据时，跳表可以做到 $O(\log n)$ 的时间复杂度定位区间的起点，然后在原始链表中顺序往后遍历就可以了，非常高效。

3、总结

- (1) 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。

- (2) 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。
- (3) 每个跳跃表节点的层高都是 1 至 32 之间的随机数。
- (4) 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。
- (5) **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。**