Redis设计与实现（一）

概述

Redis是一款内存高速缓存数据库。Redis全称为：**Remote Dictionary Server**（远程数据服务），使用C语言编写，Redis是一个key-value存储系统（键值存储系统），支持丰富的数据类型，如：String、list、set、zset、hash。

可用于缓存，事件发布或订阅，高速队列等场景。支持网络，提供字符串，哈希，列表，队列，集合结构直接存取，基于内存，可持久化。

核心特点：

1. **内存存储**：数据主要存放在内存中，访问速度非常快（通常在微秒级别）。
2. **多数据结构支持**：不仅支持简单的 key-value，还支持 Hash、List、Set、ZSet 等多种结构。
3. **持久化机制**：可以将内存中的数据持久化到磁盘，防止数据丢失。
4. **单线程 + I/O 多路复用**：通过事件驱动高效处理大量并发请求。
5. **丰富功能**：事务（MULTI/EXEC）、Lua 脚本、发布/订阅、位图、HyperLogLog、地理位置查询等。

> 小结：Redis 本质上是一个**内存数据库**，它通过精心设计的数据结构和单线程模型实现了极高的性能。

Redis 底层数据结构

1️⃣ SDS—— 简单动态字符串

- Redis 的 **String 类型**并不是直接用 C 语言的 `char*`，而是 SDS。
- **结构**：
  - `len`：字符串长度
  - `alloc`：已分配空间
  - `buf[]`：真正存储数据的数组（末尾有 `\0`）

✨ 好处：

1. **O(1) 获取长度**（不像 C 字符串要遍历）。
2. **空间预分配 + 惰性释放**，避免频繁扩容和缩容。
3. **二进制安全**，可以存图片、压缩数据。

👉 应用场景：`SET key "value"` 就是存 SDS。

2️⃣ Linkedlist —— 双端链表

- Redis 的 **List 类型**可能用到它。
- **结构**：
  - 每个节点有 `prev`、`next` 和 `value`。
  - 头尾指针支持快速 `LPUSH` / `RPUSH`。

👉 应用场景：消息队列、任务队列。

⚠️ 注意：当 List 很小的时候，Redis 不会用 linkedlist，而是用 **ziplist（压缩列表）** 来节省内存。

3️⃣ Ziplist —— 压缩列表

- 一种紧凑的连续内存结构，类似"数组 + 变长编码"。
- **结构**：
  - `zlbytes`：整个列表占用字节数
  - `zltail`：最后一个元素的偏移量
  - `zllen`：元素个数
  - `entry[]`：实际元素，一个接一个存放

✨ 特点：内存连续，节省空间，但插入删除需要移动数据。

👉 应用场景：

- 小 Hash（少量字段）
- 小 List（少量元素）
- 小 ZSet（少量元素）

4️⃣ Dict（哈希表）

- Redis 的 **Hash 类型**就是基于 Dict 实现的。
- **结构**：
  - `table[]`：数组（哈希桶）
  - `entry`：链表解决哈希冲突
  - 支持 **渐进式 rehash**（避免一次性扩容太耗时）

5️⃣ Intset —— 整数集合

- 一种专门为存储整数的紧凑数组。
- **结构**：
  - 元素有序排列，二分查找
  - 按照元素类型自动升级（16 位 → 32 位 → 64 位）

👉 应用场景：小规模的 Set（只含整数，比如用户 ID 集合）。

6️⃣ Skiplist —— 跳跃表

- Redis 的 ZSet（有序集合）由 skiplist + dict共同实现：
  - dict：快速查找成员是否存在
  - skiplist：根据 score 有序存储，支持范围查询

✨ 特点：

- 查找/插入/删除平均 O(log n)
- 实现比红黑树简单，且性能接近

👉 应用场景：排行榜、区间查询、按权重排序的数据。

SDS - 简单字符串

Redis没有直接使用C语言传统的字符串表示（以空字符结尾的字符数组），而是自己构建了一种简单动态字符串（simple dynamic string，SDS）的抽象类型，并将SDS用作Redis的默认字符串表示。

```
/*
 * 保存字符串对象的结构
 */
struct sdshdr {

    // buf 中已占用空间的长度
    int len;

    // buf 中剩余可用空间的长度
    int free;

    // 数据空间
    char buf[];
};
```

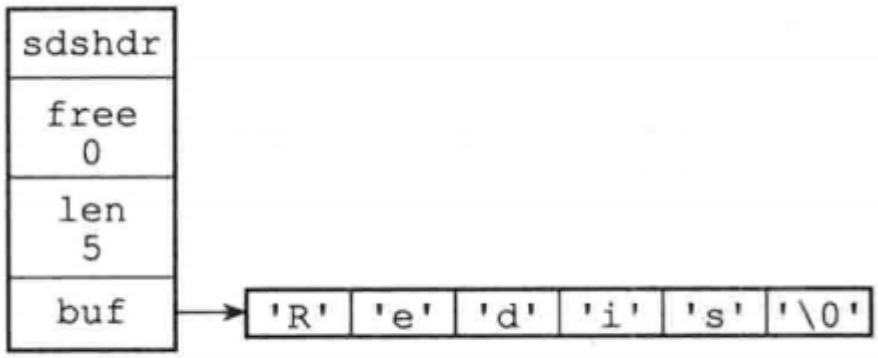

图 2-1　SDS 示例

- 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5字节长的字符串。
- buf属性是一个char类型的数组，最后一个字节保存了空字符'\0'。

SDS遵循C字符串以空字符结尾的惯例，保存空字符的1字节空间不计算在SDS的len属性里面，并且为空字符分配额外的1字节空间。添加空字符串到字符串末尾等操作，都是SDS函数自动完成的，所以这个空字符对于SDS的使用者来说是完全透明的。

好处：可以直接使用C的printf函数，无须为SDS编写专门的打印函数。

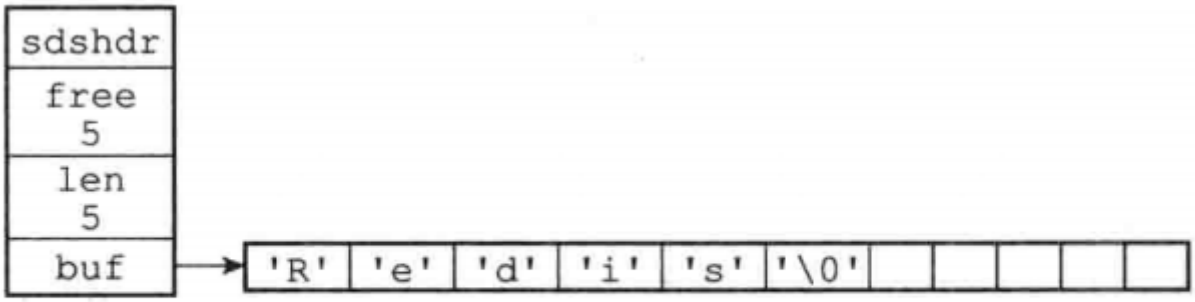

图 2-2　带有未使用空间的 SDS 示例

SDS与C字符串的区别

- O(1)获取字符串长度

因为C字符串不记录自身的长度信息，所以为了获取一个C字符串的长度，需要遍历整个字符串，对遇到的每个字符进行计数，直到遇到代表字符串结尾的空字符为止，复杂度为O(N)。

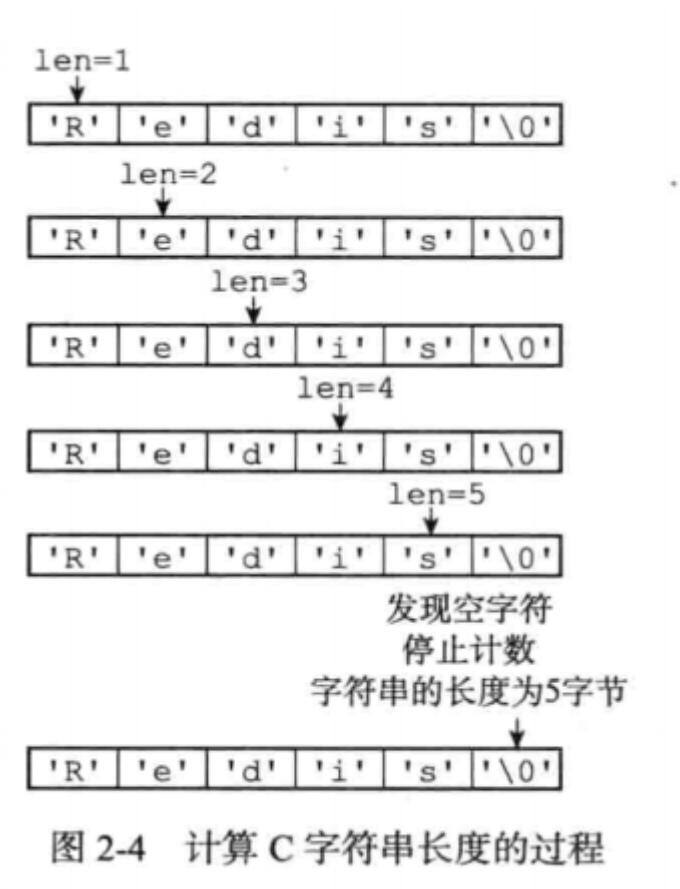

图 2-4　计算 C 字符串长度的过程

与C字符串不同，因为SDS在len属性中记录了SDS本身的长度，所以获取一个SDS长度的复杂度为O(1)。

图 2-5　5 字节长的 SDS

设置和更新SDS长度的工作是由SDS的API在执行时自动完成的，使用SDS无须进行任何修改长度的工作。

- 杜绝缓冲区溢出

除了获取字符串长度的复杂度高之外，C字符串不记录自身长度带来的另一个问题是容易造成缓冲区溢出。

当程序**向缓冲区写入的数据超过了它的容量**时，多余的数据会覆盖相邻的内存区域。

缓冲区溢出

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
    char original[] = "Hello,World";

    // 创建指向原始字符串不同部分的指针
    char *first = original;
    char *second = original + 6; // 指向"World"

    // 在逗号处添加结束符来分割字符串
    original[5] = '\0';

    strcat(first, "Kity");

    printf("原始字符串地址: %p\n", (void*)original);
    printf("第一个字符串地址: %p\n", (void*)first);
    printf("第二个字符串地址: %p\n", (void*)second);

    printf("第一部分: %s\n", first);
    printf("第二部分: %s\n", second);

    // 检查是否连续
    if (second == first + 6) {
        printf("内存是连续的\n");
    } else {
        printf("内存是不连续的\n");
        printf("地址差: %td\n", second - first);
    }

    return 0;
}
```

打印输出

```
原始字符串地址: 0x7fffe7525594
第一个字符串地址: 0x7fffe7525594
第二个字符串地址: 0x7fffe752559a
第一部分: HelloKity
第二部分: ity
内存是连续的
```

与C字符串不同，SDS的空间分配策略完全杜绝了发送缓冲区溢出的可能性：当SDS的API需要对SDS进行修改时，API会先检查空间是否满足，如果不满足API会自动将SDS的空间扩展至执行修改所需的大小，然后才执行实际的修改操作。

减少修改字符串时带来的内存重分配次数

因为C字符串并不记录自身的长度，所以对于一个包含了N个字符的C字符串来说，这个C字符串的底层实现总是一个N + 1个字符长的数组。每次增长或者缩短一个C字符串，程序都总要对保存这个C字符串的数组进行一次内存重分配操作：

- 增长字符串，拼接（append），程序需通过内存重分配来扩展底层数组的空间大小。如果忘了就会产生缓冲区溢出。
- 缩短字符串，截断（trim），程序需要先通过内存重分配来释放字符串不再使用的那部分空间，如果忘了就会产生内存泄漏。

为了避免C字符串的这种缺陷，SDS通过未使用空间解除了字符串长度和底层数组长度之间的关联：在SDS中，buf数组的长度不一定就是字符数量+ 1，数组里面可以包含未使用的字节，而这些字节的数量就由SDS的free属性记录。

通过未使用空间，SDS有以下两种优化策略：

- 空间预分配

  当SDS需要进行空间扩展的时候时，程序不仅分配所需空间，还会分配额外的未使用空间。

  - 对SDS进行修改后，SDS的长度(len)将小于1MB，那么也会分配同样大小的未使用空间，这时SDS的len = free
    - 如果修改之后，SDS的len将变成13字节，那么程序也会分配13字节的未使用空间，SDS的buf数组实际长度=13+13+1=27。
  - SDS的长度大于等于1MB，程序分配1MB的未使用空间。
    - SDS的len将变成30MB，那么程序会分配1MB的未使用空间，SDS的buf数组的实际长度=30MB+1MB+1byte

通过这种预分配策略，SDS将连续增长N次字符串所需的内存重分配次数从必定N次降低为最多N次。

- 惰性空间释放

惰性空间释放用于优化SDS的字符串缩短操作：当SDS的API需要缩短SDS保存的字符串时，程序并不立即使用内存重分配来回收缩短后多出来的字节，而是使用free属性将这些字节的数量记录起来，并等待将来使用。

举个例子，sdstrim 函数接受一个 SDS 和一个 C 字符串作为参数，移除 SDS 中所有在 C 字符串中出现过的字符。

比如对于图 2-14 所示的 SDS 值 s 来说，执行：

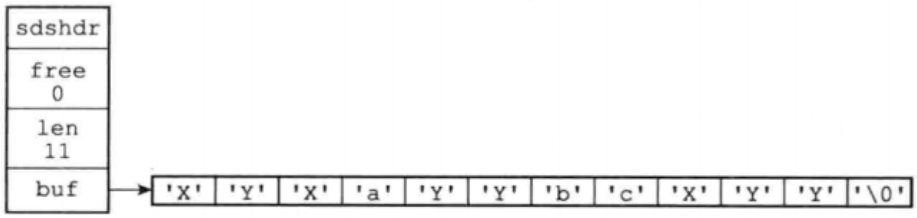

图 2-14　执行 sdstrim 之前的 SDS

会将 SDS 修改成图 2-15 所示的样子。

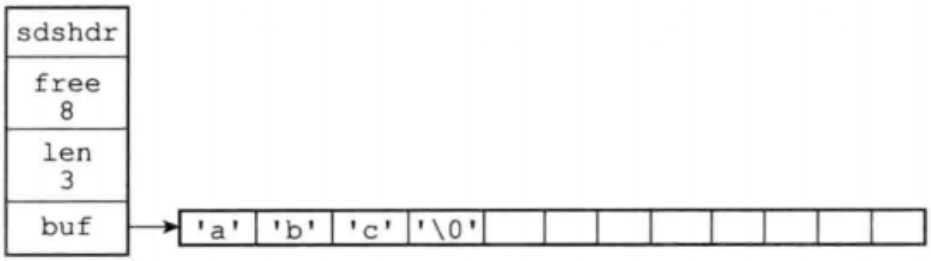

图 2-15　执行 sdstrim 之后的 SDS

通过惰性空间释放策略，SDS避免了缩短字符串时所需的内存重分配操作，并为将来可能有的增长操作提供了优化。

🔎 SDS 空闲内存回收时机

在 Redis 的 **SDS（Simple Dynamic String）** 实现中，惰性释放只是暂时保留空闲空间（`free` 字段记录），并不是永远不释放。真正释放内存的场景主要有以下几种：

1. 字符串缩小且显式调用 `sdsRemoveFreeSpace()`
2. 字符串需要扩容但现有空间不够
3. SDS 被整体释放
4. 后台内存优化或持久化过程

二进制安全

C字符串中的字符必须符合某种编码（比如ASCII），并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读入的空字符被误认为是字符串结尾，这些限制使得C字符串只能保存文本数据，而不能保存像图片、音频、视频、压缩文件这样的二进制数据。

为了确保Redis可以适用于不同的使用场景，SDS的API都是二进制安全的，所有SDS API都会以处理二进制的方式来处理SDS存放在buff数组里面的数据。

Redis不是用这个数组来保存字符，而是保存一系列二进制数据。

例如，使用 SDS 来保存之前提到的特殊数据格式就没有任何问题，因为 SDS 使用 len 属性的值而不是空字符来判断字符串是否结束，如图 2-18 所示。

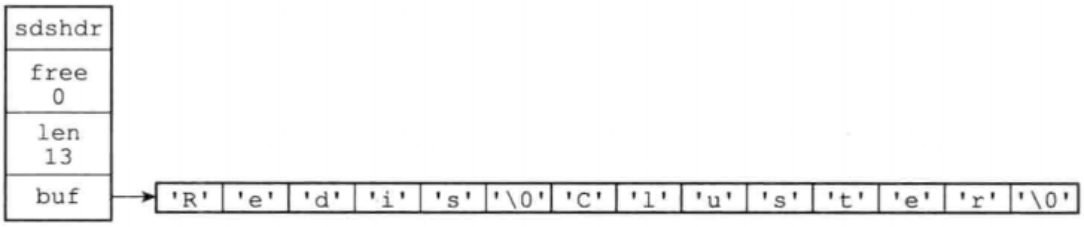

图 2-18　保存了特殊数据格式的 SDS

Q: 为什么 SDS 要保留 `'\0'` 结尾?

A: **SDS 是二进制安全的** → `len` **才是内容的真正边界**，SDS 仍然以 `'\0'` 结尾 → 兼容 C 库，方便调试，几乎无成本

所以：SDS 的本质是"二进制安全的动态字符串"，但保持 `'\0'` 结尾是为了兼容传统 C 生态。

链表

因为Redis使用的C语言没有内置这种数据结构，所以Redis构建了自己的链表实现。

```
/*
 * 双端链表节点
 */
typedef struct listNode {

    // 前置节点
    struct listNode *prev;

    // 后置节点
    struct listNode *next;

    // 节点的值
    void *value;

} listNode;
```

多个 listNode 可以通过 prev 和 next 指针组成双端链表，如图 3-1 所示。

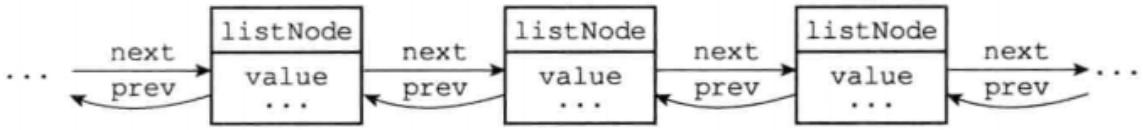

图 3-1　由多个 listNode 组成的双端链表

```
/*
 * 双端链表结构
 */
typedef struct list {

    // 表头节点
    listNode *head;

    // 表尾节点
    listNode *tail;

    // 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);

    // 链表所包含的节点数量
    unsigned long len;

} list;
```

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：

- dup函数用于复制链表节点所保存的值；
- free函数用于释放链表节点所保存的值；
- match函数则用于对比链表节点所保存的值和另一个输入值是否相等。

图 3-2 是由一个 list 结构和三个 listNode 结构组成的链表。

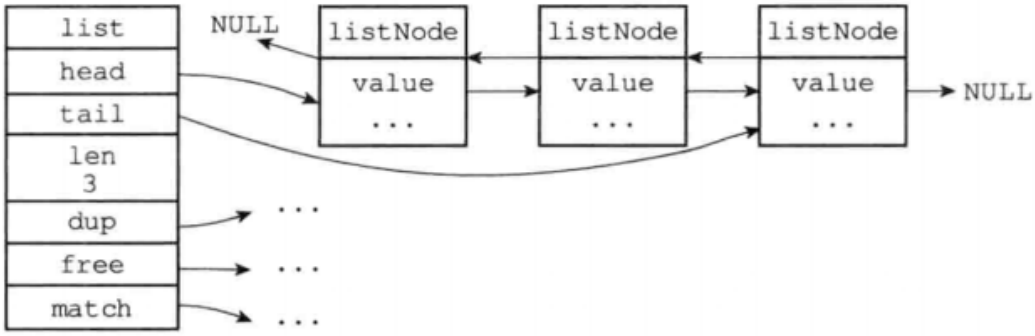

图 3-2　由 list 结构和 listNode 结构组成的链表

链表实现总结：

- 双端：链表节点带有prev和next指针，获取某个节点的前置节点和后置节点的复杂度都是O(1)。
- 无环：表头节点的prev指针和表尾节点的next指针都执行NULL，对链表的访问以NULL为终点。
- 带表头指针和表尾指针：通过list结构的head指针和tail指针，程序获取链表的表头节点和表尾节点的复杂度为O(1)。
- 带链表长度计数器：程序使用list结构的len属性对list持有的链表节点进行计数，程序获取链表中节点数量的复杂度为O(1)。
- 多态：链表节点使用void*指针来保存节点值，并且可以通过list结构的dup、free、match三个属性为节点值设置类型特定函数，所以链表可以用于保存各种不同类型的值。

字典

字典，又称为符号表、关联数组或者映射，是一种用于保存键值对的抽象数据结构。

字典的实现

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

```
/*
 * 哈希表
 *
 * 每个字典都使用两个哈希表，从而实现渐进式 rehash 。
 */
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 （因为索引计算非常频繁，所以空间换时间冗余了这个字段减少计算量）
    unsigned long sizemask;

    // 该哈希表已有节点的数量
    unsigned long used;

} dictht;
```

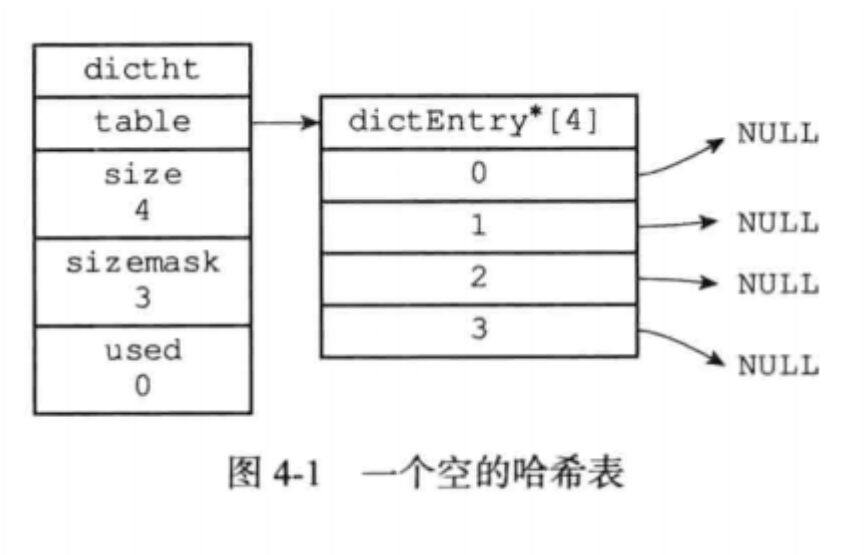

图 4-1　一个空的哈希表

table属性是一个数组，数组中的每个元素都是一个指向dict.h/dictEntry结构的指针，每个dictEntry结构保存着一个键值对。size属性记录了哈希表的大小，也即是table数组的大小，而used属性则记录哈希表目前已有节点的数量。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。

哈希表节点

哈希表节点使用dictEntry结构表示，每个dictEntry结构都保存着一个键值对：

```
/*
 * 哈希表节点
 */
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;
```

key属性保存着键值对中的键，而v属性则保存着键值对中的值，其中键值对的值可以是一个指针，或者是一个uint64_t整数，又或者是一个int64_t整数。

next属性是指向另一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对连接在一起，以此来解决键冲突的问题。

字典

```
/*
 * 字典
 */
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 */

    // 目前正在运行的安全迭代器的数量
    int iterators; /* number of iterators currently running */

} dict;
```

type属性和privdata属性是针对不同类型的键值对，为创建多态字典而设置的：

- 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。

```
/*
 * 字典类型特定函数
 */
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;
```

- ht属性是一个包含了两个项的数组，数组中的每个项都是一个dictht哈希表，一般情况下，字典只是用ht[0]哈希表，ht[1]哈希表只会在对ht[0]哈希表进行rehash时使用。

  除了ht[1]之外，另一个和rehash有关的属性就是rehashidx，他记录rehash目前的进度，如果目前没有在进行rehash，那么他的值为-1。

图 4-3 展示了一个普通状态下（没有进行 rehash）的字典。

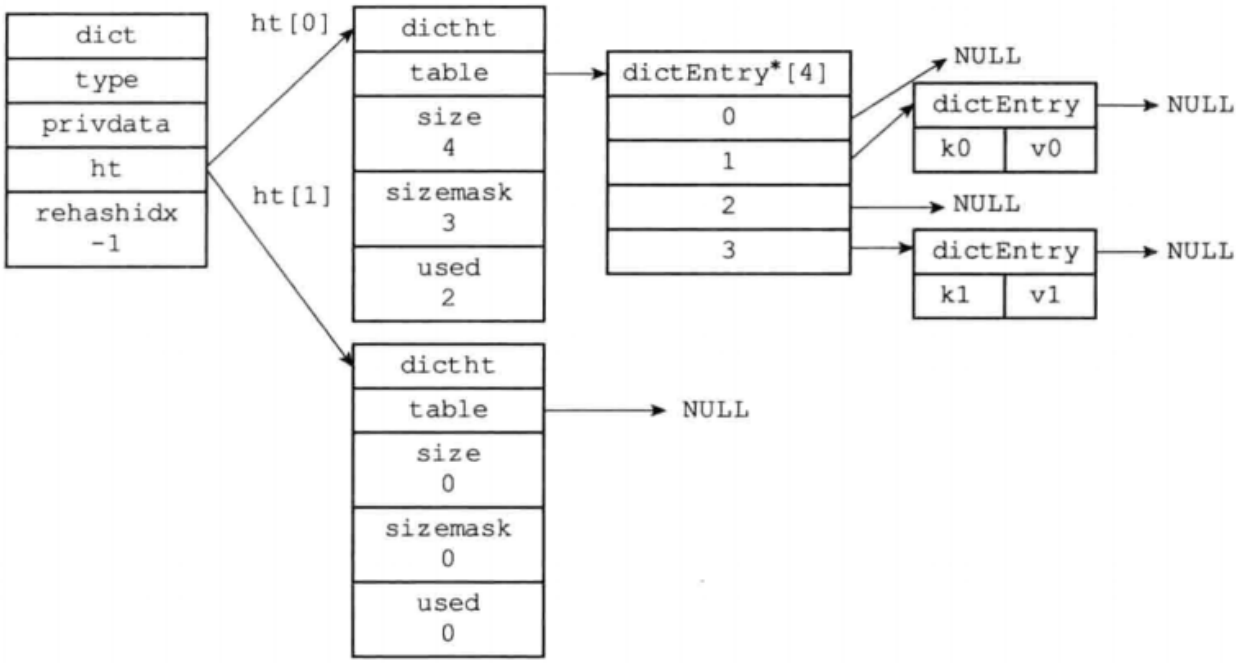

图 4-3　普通状态下的字典

rehash

随着操作的不断执行，哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表的负载因子(load factor)维持在一个合理的范围之内，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。

触发扩容的条件

Redis 判断是否扩容主要看 **负载因子** (load factor)：

```
负载因子 = 已用节点数 / 哈希表大小 = used / size
```

规则如下：

1. **正常情况下**
   - 当负载因子 ≥ 1 时触发扩容。
   - 即元素数 ≥ 槽数时，就要扩容。
2. **在 Redis 正在执行 bgsave（RDB 快照）或 AOF rewrite 时**
   - 扩容门槛会提高：负载因子 ≥ 5 才扩容。
   - 避免在持久化时频繁扩容影响性能。

扩容时表大小的选择

Redis 总是把新表的大小设为 **大于等于 2×used 的最小 2 的幂。**
比如：

- 当前 `used=10`，则新表大小选择 `32` （大于等于 20 的最小 2^n）。
- 如果 `used=1000`，新表大小选择 `2048`。

触发收缩的条件

收缩的规则更保守：

- 当负载因子 < **0.1** 时，触发收缩。

比如：

- 当前 `size=1024`， `used=50`，负载因子=0.0488 < 0.1 → 触发收缩。

收缩后的大小同样取 **大于等于 used 的最小 2 的幂**，并且不能小于初始值 `DICT_HT_INITIAL_SIZE=4`。

渐进式rehash

扩展或收缩哈希表需要将ht[0]里面的所有键值对rehash到ht[1]里面，但是这个动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成。

> 如果保存的键值对数量达到一定量级，一次性将全部键值对rehash到ht[1]这个过程会非常耗时并且导致服务器在一段时间内停止服务。

因此，为了避免rehash对服务器性能造成影响，选择分多次、渐进式地将ht[0]里面的键值对慢慢地rehash到ht[1]。

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，并将他的值设置为0，表示开始rehash。
3. 在rehash进行期间，每次CURD操作时，程序除了执行指定的操作之外，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操作完成。

渐进式rehash的好处在于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添加、删除、查找和更新操作上，从而避免了集中式rehash带来的庞大计算量。

图 4-12 至图 4-17 展示了一次完整的渐进式 rehash 过程，注意观察在整个 rehash 过程中，字典的 rehashidx 属性是如何变化的。

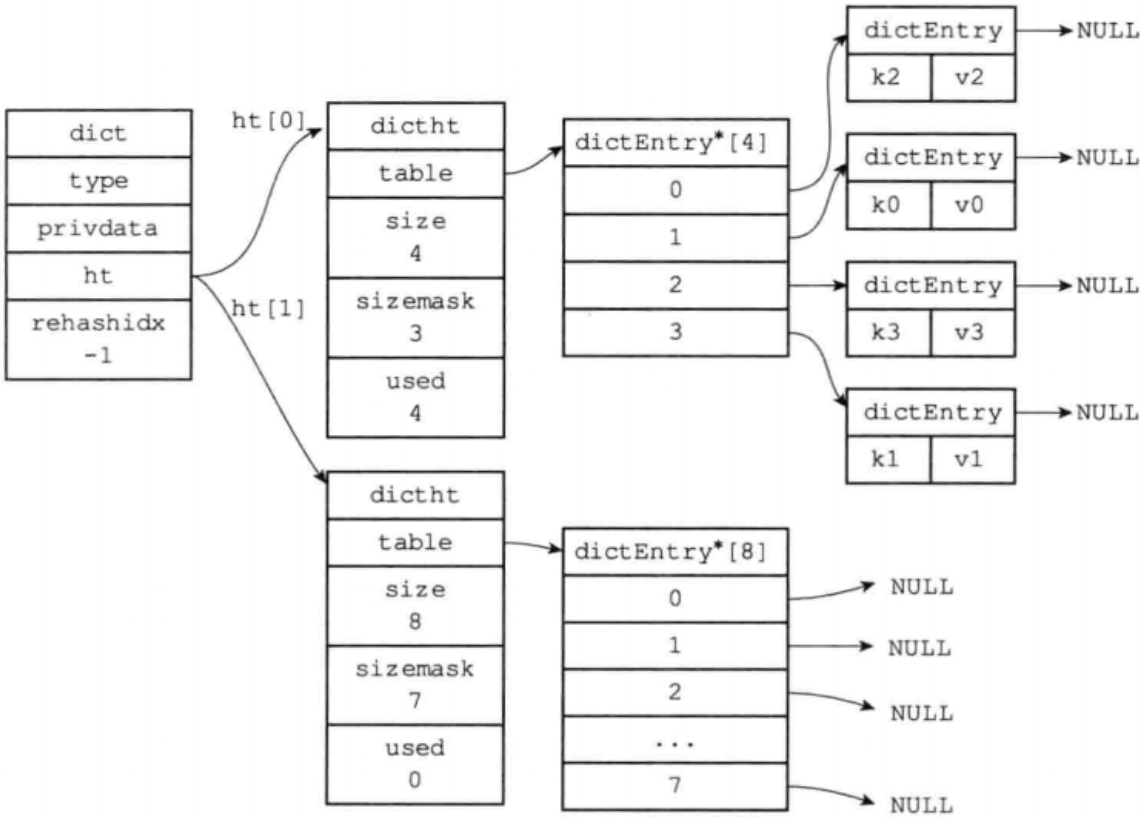

图 4-12　准备开始 rehash

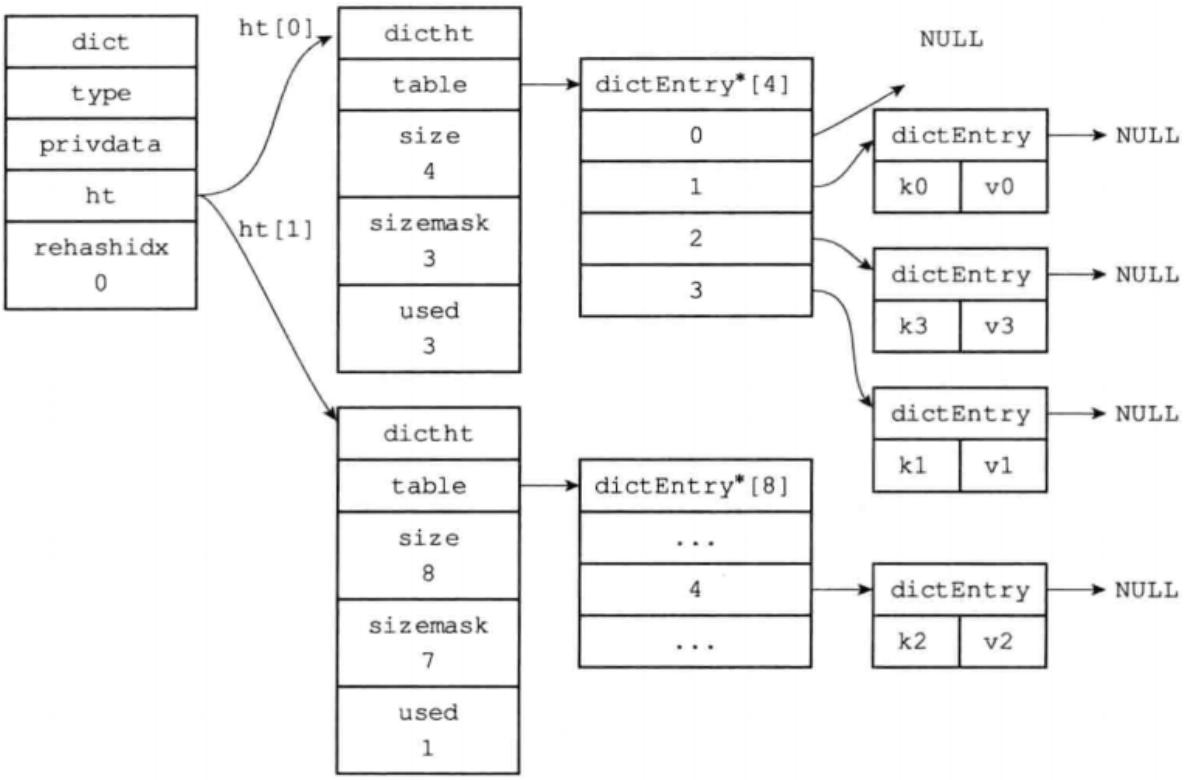

图 4-13　rehash 索引 0 上的键值对

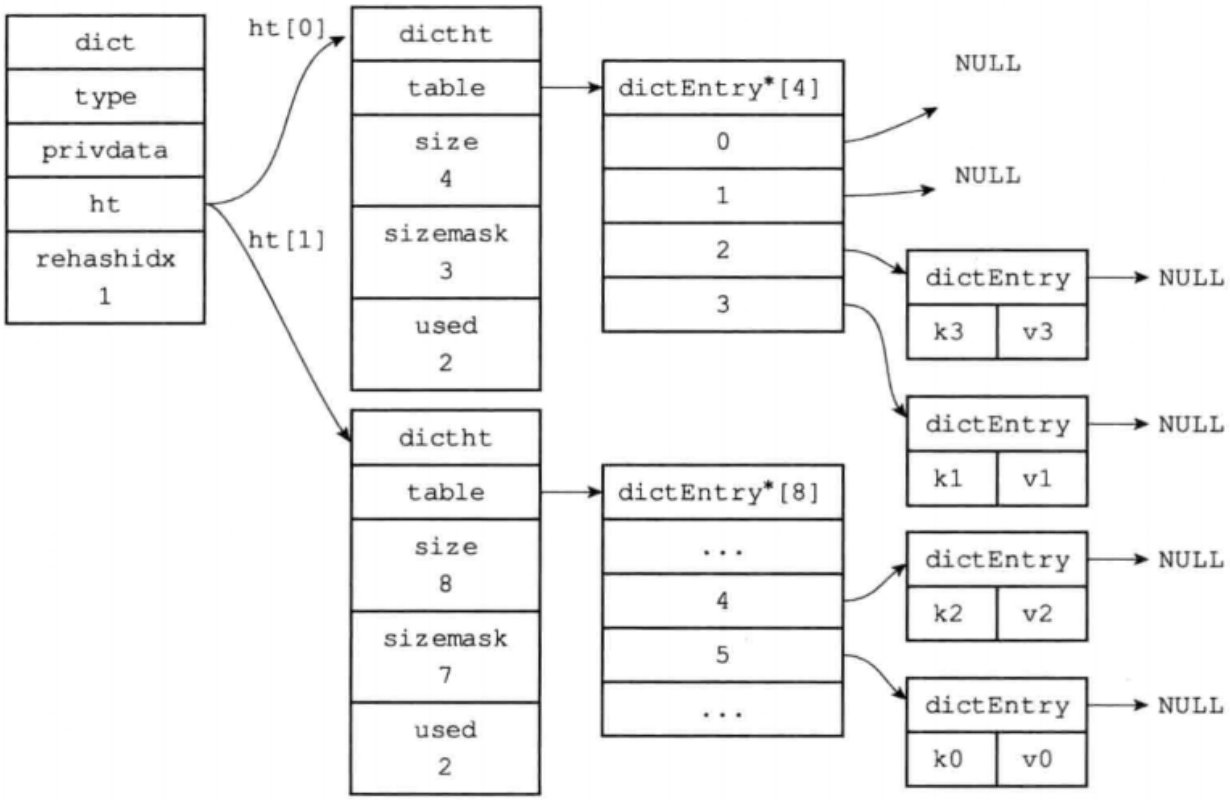

图 4-14　rehash 索引 1 上的键值对

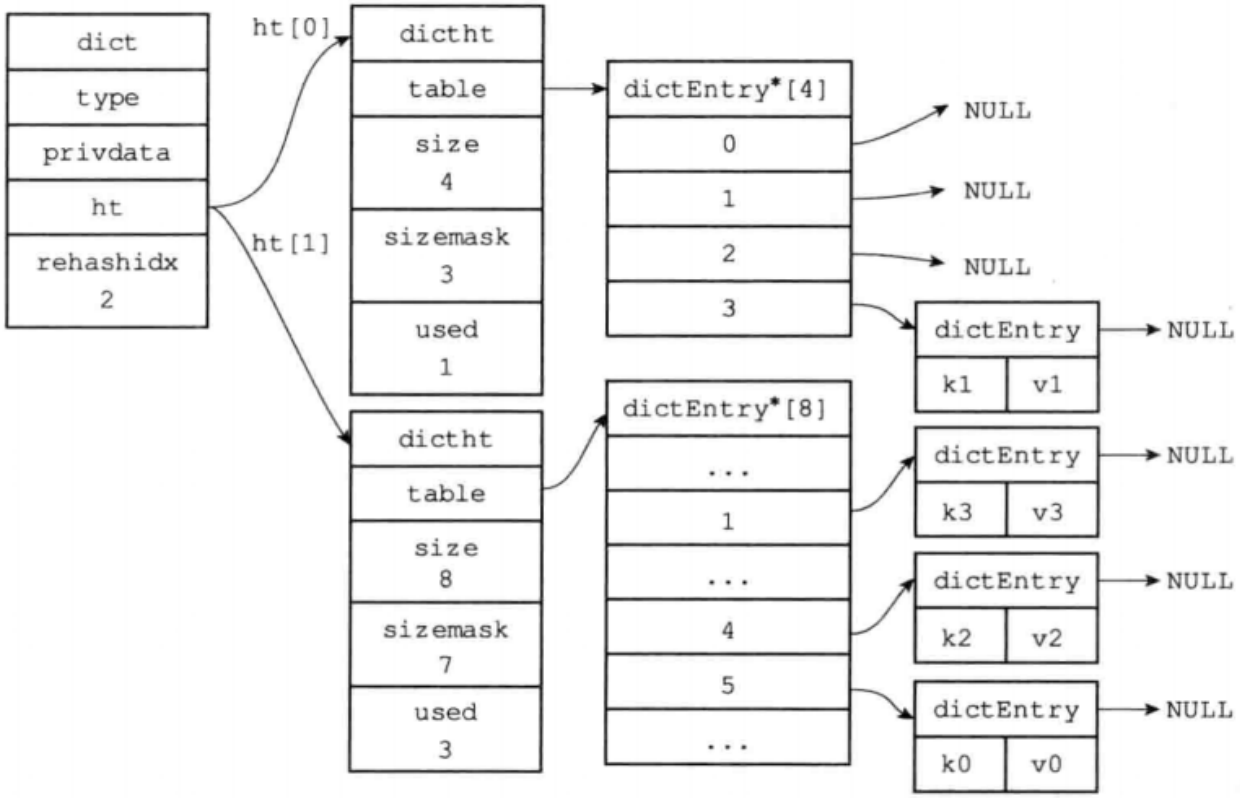

图 4-15　rehash 索引 2 上的键值对

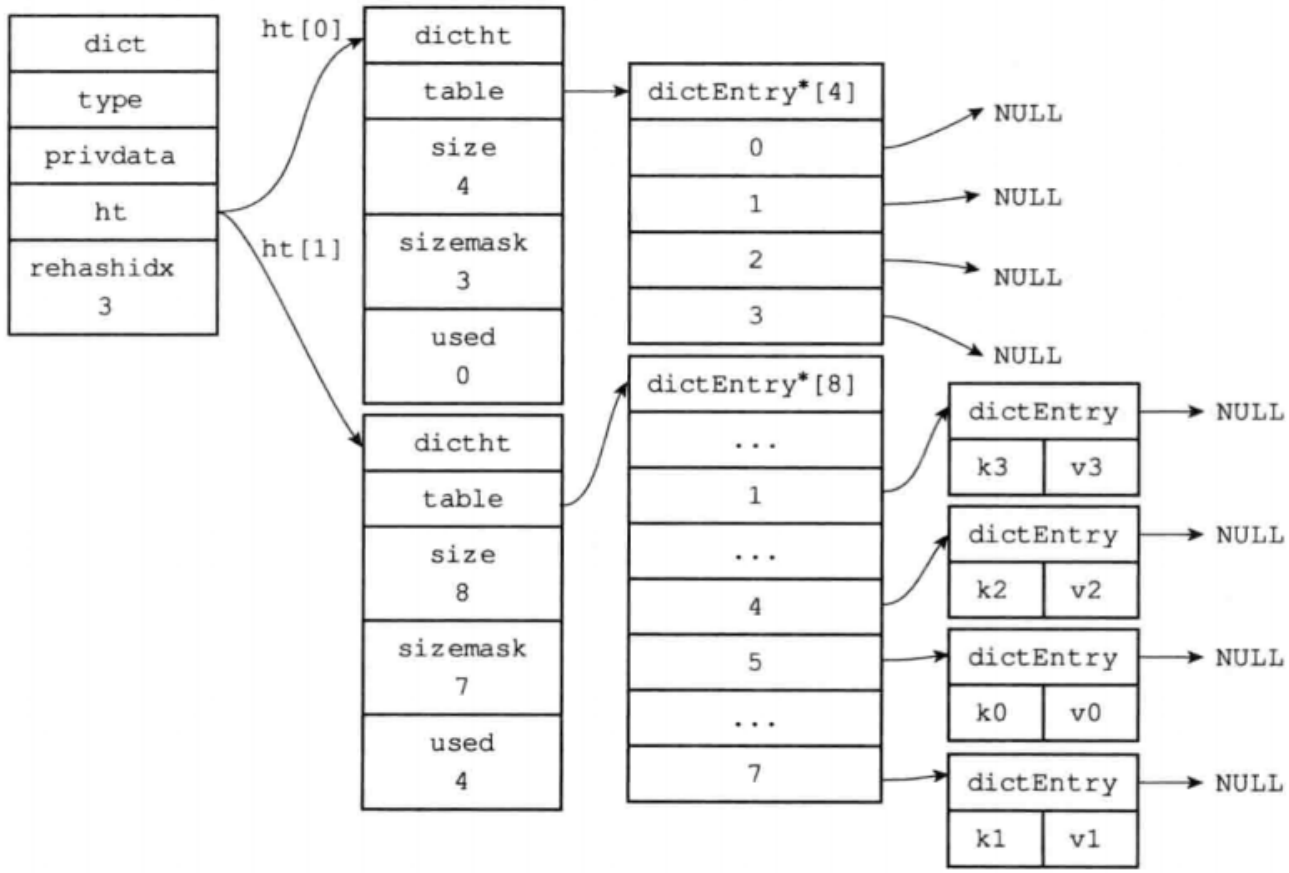

图 4-16　rehash 索引 3 上的键值对

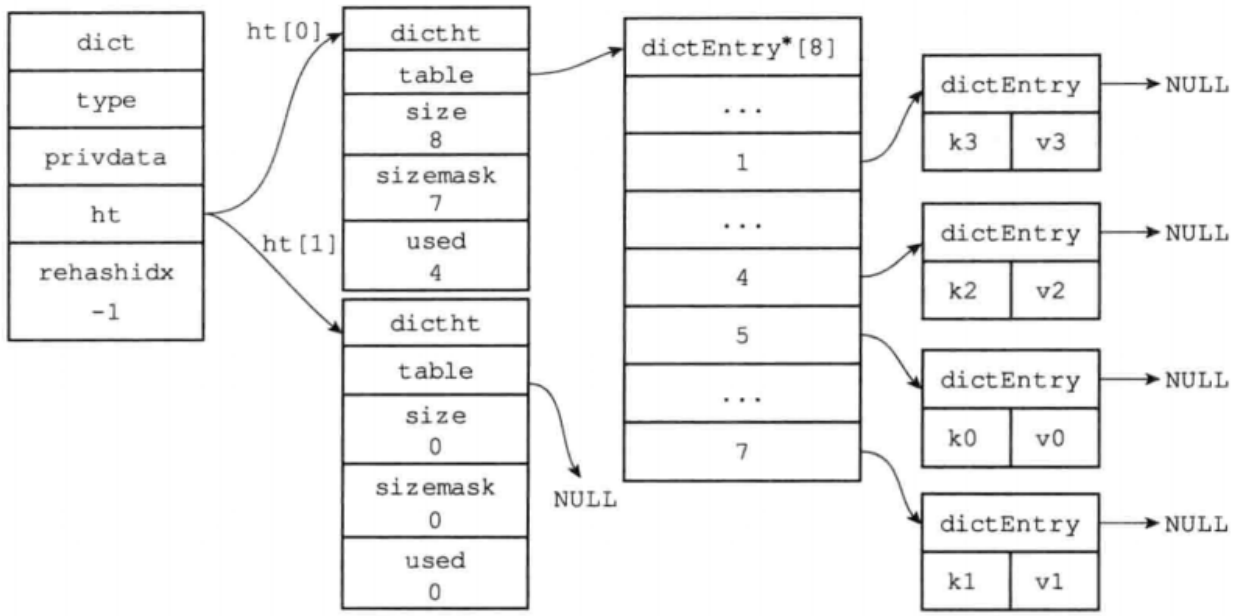

图 4-17　rehash 执行完毕

渐进式rehash执行期间的哈希表操作

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，字典的删除、查找、更新等操作会在两个哈希表上进行。

例如：查找一个键，会在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。