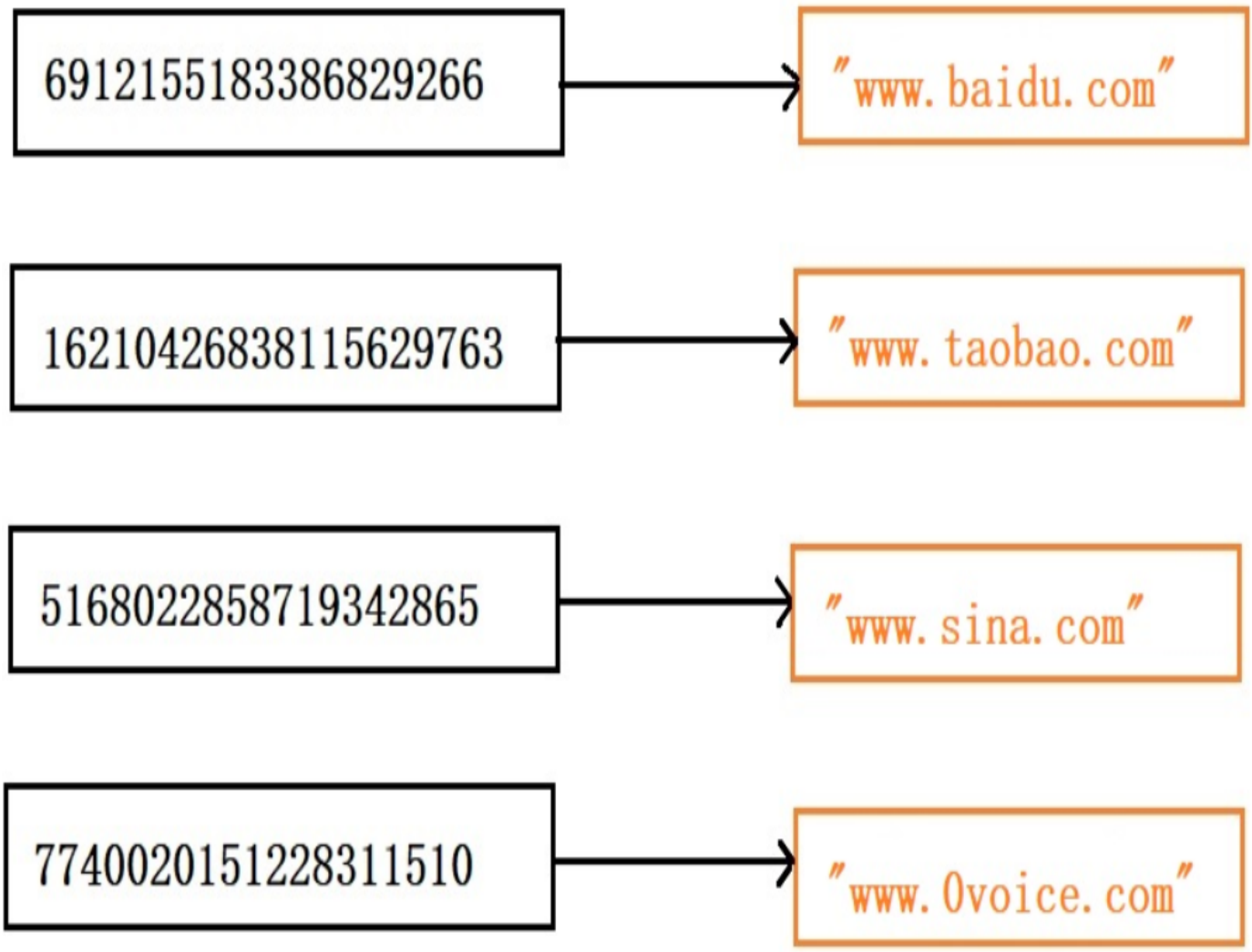
数据库的存储结构

零声学院：https://0voice.ke.qq.com/?tuin=137bb271

主流的key-value存储系统，都是在系统内部维护一个hash表，因为对hash表的操作时间复杂度为O(1)。如果数据增加以后，导致冲突严重，时间复杂度增加，则可以对hash表进行rehash，以此来保证操作的常量时间复杂度。
那么，对于这样一个基于hash表的key-value存储系统，是如何提供这么丰富的数据结构的呢？这些数据结构在内存中如何存储呢？这篇文章将用大量的图片演示redis的内存布局和数据存储。

1、Hash算法

先来看一个思考题，比如我们有一个k-v系统，如果我们把k-v系统设计成链表，比如
{"baidu":www.baidu.com, "taobao":www.taobao.com, "xinlang":www.sina.com, "lingsheng":www.0voice.com}
那如果我们要去这个链表中查找“lingsheng”的url的话，那么我们就必须遍历整个链表，然后与每一个链表的key去做strcmp才能找到或者找不到这个key，这样的做法是极其低效的，是一个O(n)的复杂度，如果我们有10W个key是不可能使用链表去存储的，那怎么办呢？
在这里我们先来介绍下hash，所谓hash就是给定一个字符串或者其它的任意值x，通过hash函数得到一个散列值hash(x)，比如说我们给定“lingsheng”，使用MurmurHash64B得到的hash值就是：7740020151228311510。
好了，介绍到这里，我们可以尝试来解决这个k-v系统了，在这里我们就是使用hash表，hash表的意思就是就是建立一个数组：



此时引入了一个矛盾：

1、如果通过索引（hash值）去读取hash表，这样设计的hash表会非常大，占用的内存是非常庞大的；

2、如果是通过hash值遍历hash表，如果k-v数量很多，则查找性能会是O(N)，则时间性能也很低。

那redis是如何解决这个矛盾的？ 另外，hash会有碰撞，即使再好的算法只是碰撞率低而已，那么redis是如何解决碰撞的?

2、redis的字典

2.1 、redis的hash表

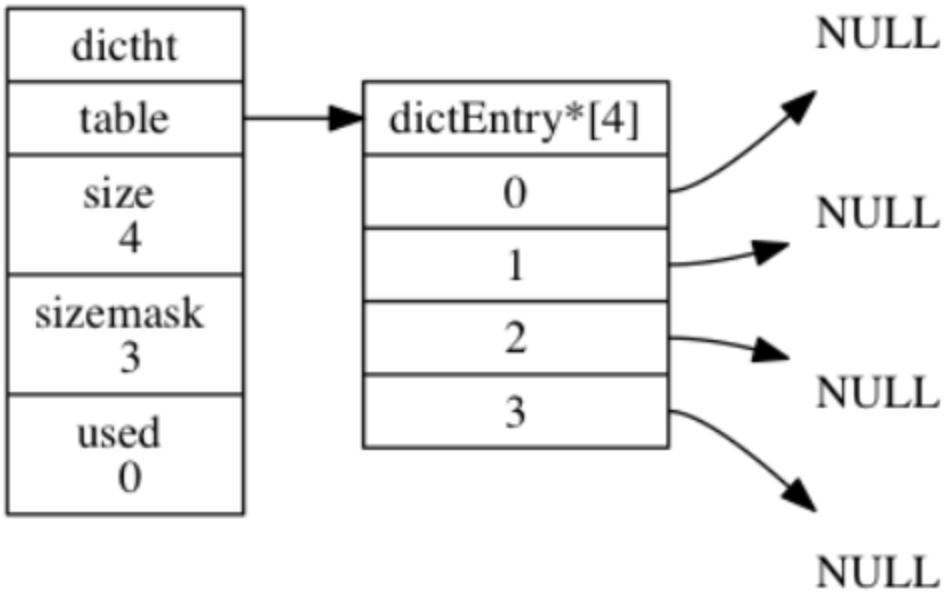
在介绍hash字典的时候，先来看看redis的hash表：

```
1 /* This is our hash table structure. Every dictionary has two of t
  his as we
2  * implement incremental rehashing, for the old to the new table.
   */
3 typedef struct dictht {
4     dictEntry **table;
```

```
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;
```

- 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；
- used属性，表示hash表里已有的数量。

如下图展示了一个空的hash表：



2.2 、hash表的节点

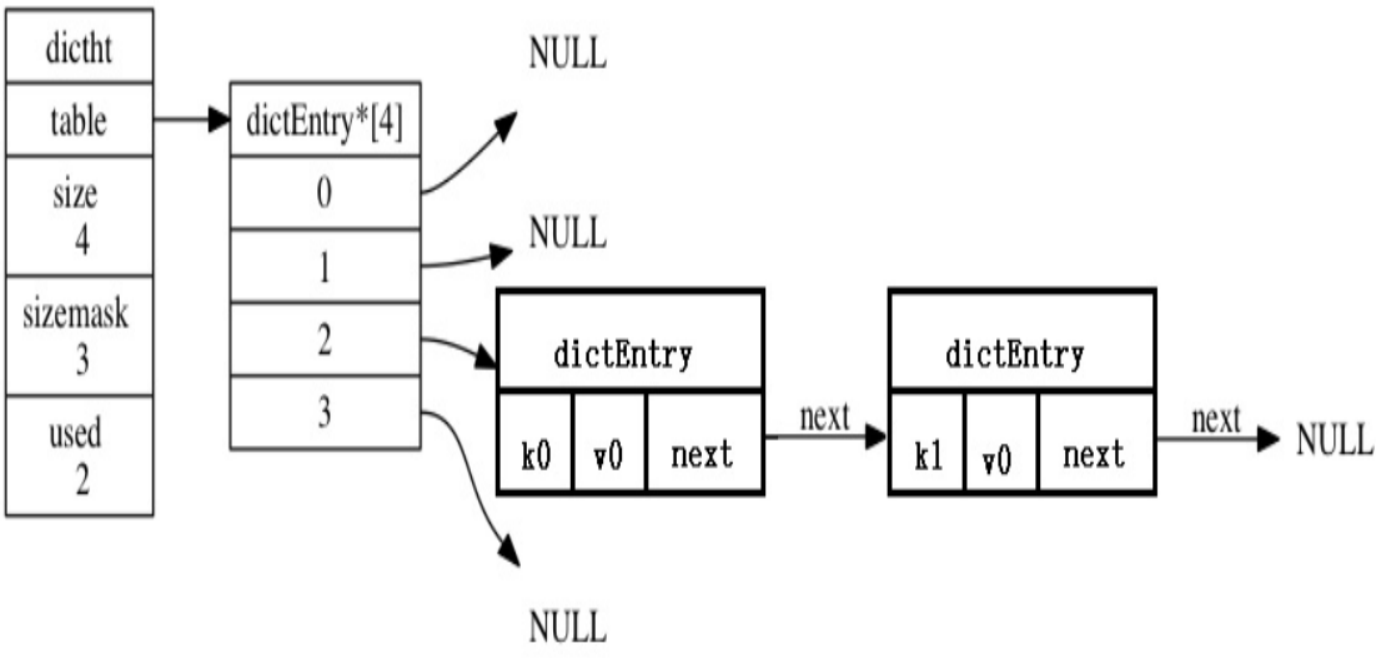
哈希表节点使用 dictEntry 结构表示， 每个 dictEntry 结构都保存着一个键值对：

```
typedef struct dictEntry {
    // 键
    void *key;

    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
```

```
10          double d;
11      } v;
12
13      // 指向下个哈希表节点，形成链表
14      struct dictEntry *next;
15  } 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，或者是double类型。
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次， 以此来解决键冲突（collision）的问题。
举个例子， 下图图就展示了如何通过 next 指针， 将两个索引值相同的键 k1 和 k0 连接在一起。



2.3 、redis字典

先来看字典的定义：

```
1  typedef struct dict {
2      // 字典类型
3      dictType *type;
4
5      // 字典的私有数据
6      void *privdata;
7
8      // 哈希表，二维的，默认使用ht[0],当需要进行rehash的时候，会利用ht[1]进
   行
```

```
    dictht ht[2];

    // rehash的索引，当没有进行rehash时其值为-1
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */

    // hash表的迭代器，一般用于rehash和主从复制等等
    unsigned long 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：

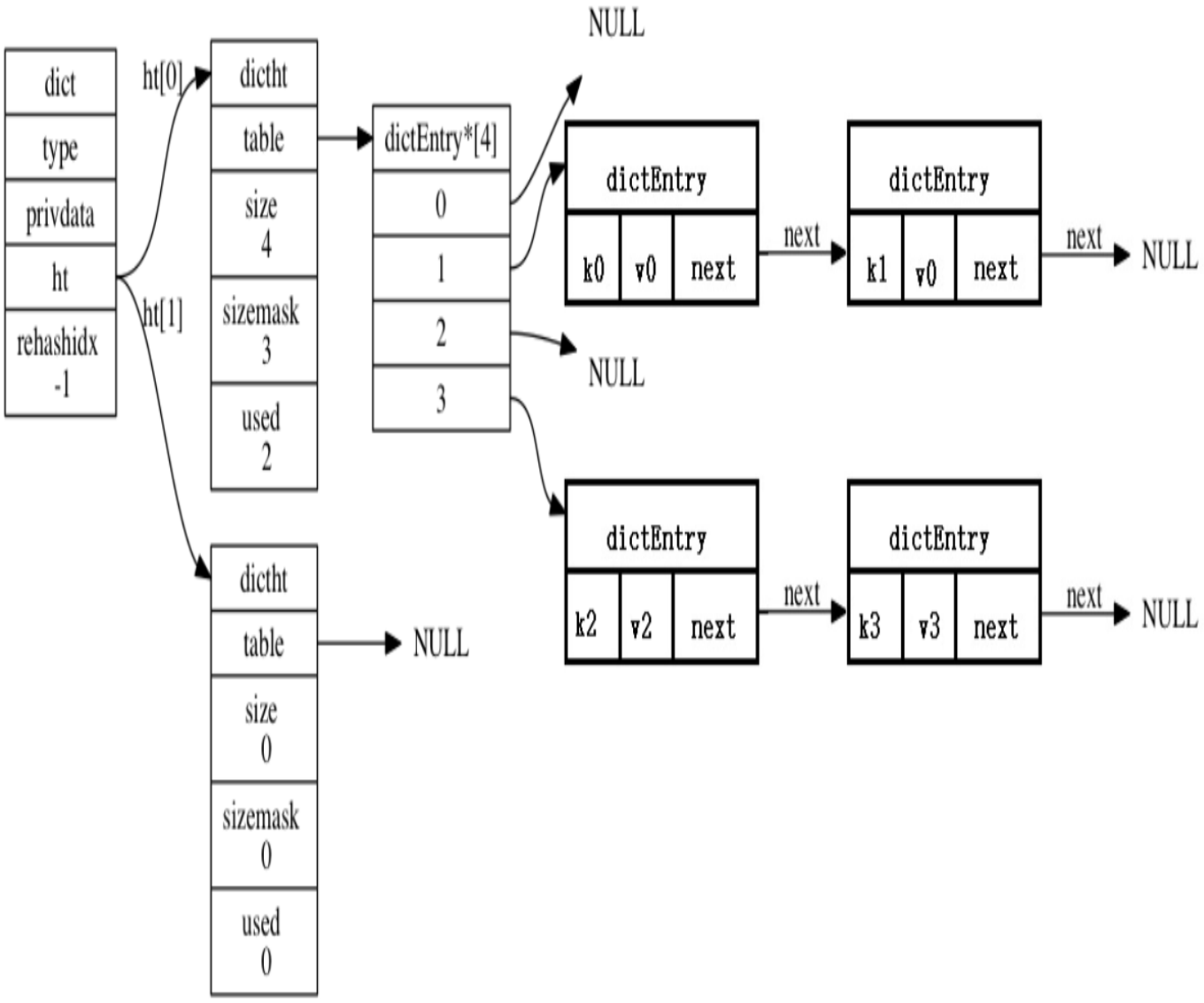
- type 属性是一个指向 dictType 结构的指针， 每个 dictType 结构保存了一簇用于操作特定类型键值对的函数， Redis 会为用途不同的字典设置不同的类型特定函数。
- 而 privdata 属性则保存了需要传给那些类型特定函数的可选参数。

```
typedef struct dictType {
    uint64_t (*hashFunction)(const void *key);
    void *(*keyDup)(void *privdata, const void *key);
    void *(*valDup)(void *privdata, const void *obj);
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);
    void (*keyDestructor)(void *privdata, void *key);
    void (*valDestructor)(void *privdata, void *obj);
} dictType;
```

ht 属性是一个包含两个项的数组， 数组中的每个项都是一个 dictht 哈希表， 一般情况下， 字典只使用 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 ： 它记录了 rehash 目前的进度， 如果目前没有在进行 rehash ， 那么它的值为 -1 。

如下图展示了一个普通的字典：



那么现在，我们先来回答下上面的第一个问题：

> redis是如何解决时间效率和空间效率的?

先来回答下： 1、初始时，字典的hash表的大小只有4（sizemask为3），那么通过hash函数计算出的hash值可能会很大，此时hash值会与上（&）sizemask，得到存储在hash表里的table[index]，序号的，见如下代码：

```
# 使用字典设置的哈希函数，计算键 key 的哈希值
hash = dict->type->hashFunction(key);
# 使用哈希表的 sizemask 属性和哈希值，计算出索引值
# 根据情况不同， ht[x] 可以是 ht[0] 或者 ht[1]
index = hash & dict->ht[x].sizemask;
```

比如我们dictFind的实现，通过key获取hash表的节点（即通过key获取value）

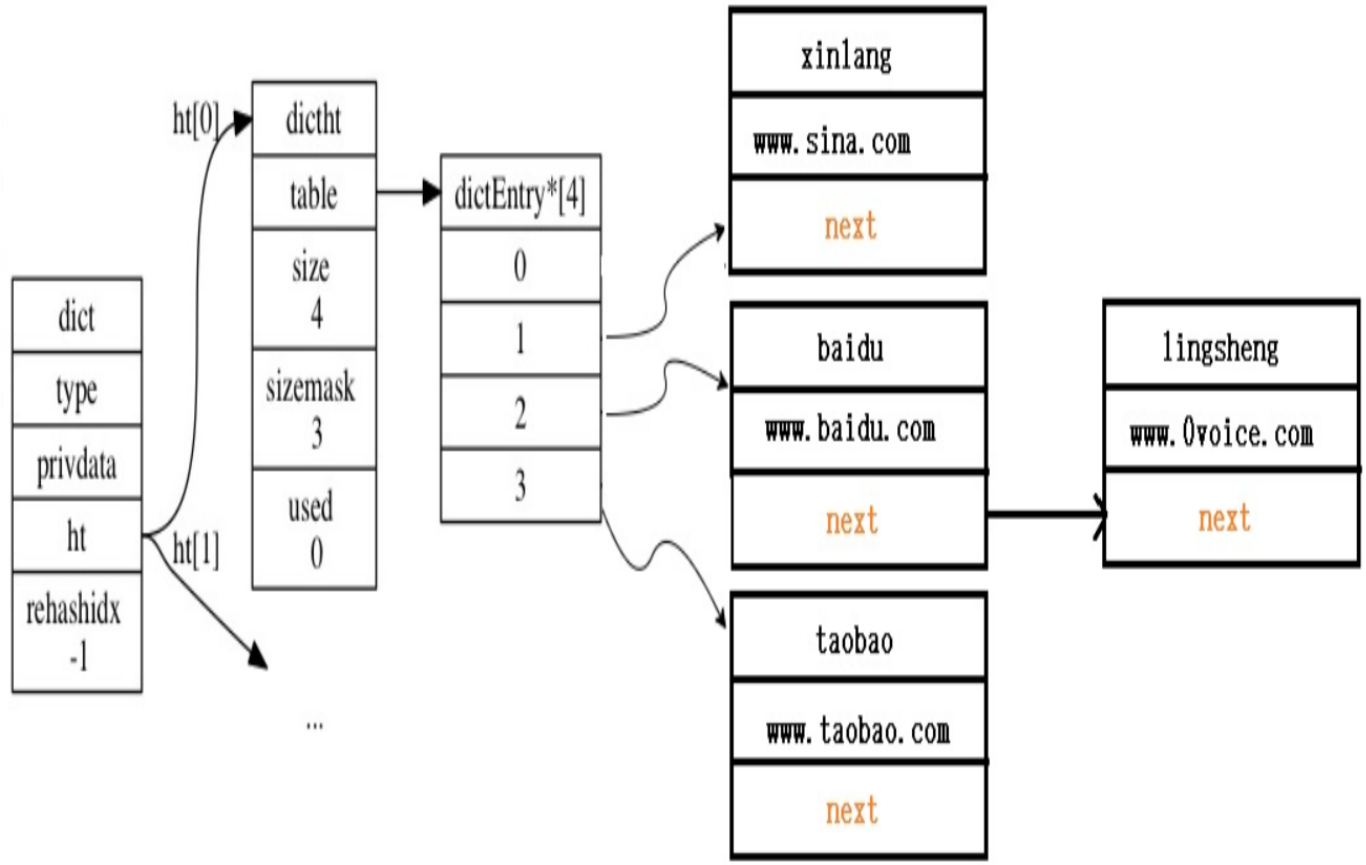
```
dictEntry *dictAddRaw(dict *d, void *key, dictEntry **existing)
{
    long index;
    dictEntry *entry;
    dictht *ht;
    if (dictIsRehashing(d)) _dictRehashStep(d);
    /* Get the index of the new element, or -1 if
     * the element already exists. */
    if ((index = _dictKeyIndex(d, key, dictHashKey(d,key), existing)) == -1)
        return NULL;
    /* Allocate the memory and store the new entry.
     * Insert the element in top, with the assumption that in a database
     * system it is more likely that recently added entries are accessed
     * more frequently. */
    ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
    entry = zmalloc(sizeof(*entry));
    entry->next = ht->table[index];
    ht->table[index] = entry;
    ht->used++;
    /* Set the hash entry fields. */
    dictSetKey(d, entry, key);
    return entry;
}
static long _dictKeyIndex(dict *d, const void *key, uint64_t hash, dictEntry **existing)
{
    unsigned long idx, table;
    dictEntry *he;
    if (existing) *existing = NULL;
    /* Expand the hash table if needed */
    if (_dictExpandIfNeeded(d) == DICT_ERR)
        return -1;
    for (table = 0; table <= 1; table++) {
        idx = hash & d->ht[table].sizemask;
        /* Search if this slot does not already contain the given
        key */
        he = d->ht[table].table[idx];
```

```
36          while(he) {
37              if (key==he->key || dictCompareKeys(d, key, he->key))
    {
38                  if (existing) *existing = he;
39                  return -1;
40              }
41              he = he->next;
42          }
43          if (!dictIsRehashing(d)) break;
44      }
45      return idx;
46  }
```

举个例子，我们上文说到了4个url，那是如何存储到一个空的hash表里的呢?

key	value	hash	hash & sizemask (index)
baidu	www.baidu.com	6912155183386829266	2
taobao	www.taobao.com	1621042683811562976 3	3
xinlang	www.sina.com	5168022858719342865	1
lingsheng	www.0voice.com	7740020151228311510	2

插入到字典后，字典结构如下：



总结下：hash表是随着K-V数量的增大而逐步增大的，并不直接以key的hash值为下标去取值得，而是以hash & sizemask去获取hash表的对应节点的；hash表的节点实际上是一个链表，如果hash & sizemask有冲突，则也把冲突key放在hash表的链表上，取值得时候还得遍历hash表里的链表。

3、redis的数据库的结构

在redis的内部，有一个redisServer结构体的全局变量server，server保存了redis服务端所有的信息，包括当前进程的PID、服务器的端口号、数据库个数、统计信息等等。当然，它也包含了数据库信息，包括数据库的个数、以及一个redisDb数组。

```
struct redisServer {
    ...
    redisDb *db;
    int dbnum;    /* Total number of configured DBs */
    ...
```

显然，dbnum就是redisDb数组的长度，每一个数据库，都对应于一个redisDb，在redis的客户端中，可以通过select N来选择使用哪一个数据库，各个数据库之间互相独立。例如：可以在不同的数据库中同时存在名为”0voice”的key。

从上面的分析中可以看到，server是一个全局变量，它包含了若干个redisDb，每一个redisDb是一个keyspace，各个keyspace互相独立，互不干扰。

下面来看一下redisDb的定义：

```
/* Redis database representation. There are multiple databases id
entified
 * by integers from 0 (the default database) up to the max config
ured
 * database. The database number is the 'id' field in the structu
re. */
typedef struct redisDb {
    dict *dict;                 /* The keyspace for this DB */
    dict *expires;              /* Timeout of keys with a timeout
set */
    dict *blocking_keys;        /* Keys with clients waiting for
 data (BLPOP)*/
    dict *ready_keys;           /* Blocked keys that received a P
USH */
    dict *watched_keys;         /* WATCHED keys for MULTI/EXEC CA
S */
    int id;                     /* Database ID */
    long long avg_ttl;          /* Average TTL, just for stats */
    list *defrag_later;         /* List of key names to attempt t
o defrag one by one, gradually. */
} redisDb;
```

redis的每一个数据库是一个独立的keyspace，因此，我们理所当然的认为，redis的数据库是一个hash表。但是，从redisDb的定义来看，它并不是一个hash表，而是一个包含了很多hash表的结构。之所以这样做，是因为redis还需要提供除了set、get以外更加丰富的功能(例如：键的超时机制)。

4、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（ratio）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。

> ratio = ht[0].used / ht[0].size

比如，hash表的size为4，如果已经插入了4个k-v的话，则ratio 为 1

> ratio = 4 / 4 = 1

扩展和收缩哈希表的工作可以通过执行 rehash （重新散列）操作来完成， Redis 对字典的哈希表执行 rehash 的策略如下：

- 1、 如果ratio小于0.1，则会对hash表进行收缩操作

```
/* If the percentage of used slots in the HT reaches HASHTABLE_MI
```

```
N_FILL
 * we resize the hash table to save memory */
void tryResizeHashTables(int dbid) {
    if (htNeedsResize(server.db[dbid].dict))
        dictResize(server.db[dbid].dict);
    if (htNeedsResize(server.db[dbid].expires))
        dictResize(server.db[dbid].expires);
}
int htNeedsResize(dict *dict) {
    long long size, used;
    size = dictSlots(dict);
    used = dictSize(dict);
    return (size > DICT_HT_INITIAL_SIZE &&
            (used*100/size < HASHTABLE_MIN_FILL));
}
```

- 2、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 ，则扩容hash表，扩容大小为当前ht[0].used*2
- 3、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，则扩容hash表，并且扩容大小为当前ht[0].used*2

```
/* Expand the hash table if needed */
static int _dictExpandIfNeeded(dict *d)
{
    /* Incremental rehashing already in progress. Return. */
    if (dictIsRehashing(d)) return DICT_OK;
    /* If the hash table is empty expand it to the initial size.
 */
    if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_
SIZE);
    /* If we reached the 1:1 ratio, and we are allowed to resize
 the hash
     * table (global setting) or we should avoid it but the ratio
between
     * elements/buckets is over the "safe" threshold, we resize d
oubling
     * the number of buckets. */
    if (d->ht[0].used >= d->ht[0].size &&
        (dict_can_resize ||
         d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
```

```
    {
        return dictExpand(d, d->ht[0].used*2);
    }
    return DICT_OK;
}
```

其实上文说的扩容为ht[0].uesd*2 是不严谨的，实际上是一个刚好大于该书的2的N次方

```
unsigned long realsize = _dictNextPower(size);
```

```
/* Our hash table capability is a power of two */
static unsigned long _dictNextPower(unsigned long size)
{
    unsigned long i = DICT_HT_INITIAL_SIZE;
    if (size >= LONG_MAX) return LONG_MAX + 1LU;
    while(1) {
        if (i >= size)
            return i;
        i *= 2;
    }
}
```

扩容的步骤如下： 1、为字典ht[1]哈希表分配合适的空间；

2、将ht[0]中所有的键值对rehash到ht[1]：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。

4、渐进式rehash

上一节说过， 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 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。

这样做的原因在于， 如果 ht[0] 里只保存着四个键值对， 那么服务器可以在瞬间就将这些键值对全部 rehash 到 ht[1]； 但是， 如果哈希表里保存的键值对数量不是四个， 而是四百万、四千万甚至四亿个键值对， 那么要一次性将这些键值对全部 rehash 到 ht[1] 的话， 庞大的计算量可能会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， 服务器不是一次性将 ht[0] 里面的所有键值对全部 rehash 到 ht[1]， 而是分多次、渐进式地将 ht[0] 里面的键值对慢慢地 rehash 到 ht[1]。

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滩到对字典的每个添加、删除、查找和更新操作上，甚至是后台启动一个定时器，每次时间循环时只工作一毫秒， 从而避免了集中式 rehash 而带来的庞大计算量。

```
/* This function handles 'background' operations we are required to do
 * incrementally in Redis databases, such as active key expiring, resizing,
 * rehashing. */
void databasesCron(void) {
    ...

    /* Perform hash tables rehashing if needed, but only if there are no
     * other processes saving the DB on disk. Otherwise rehashing is bad
     * as will cause a lot of copy-on-write of memory pages. */
    if (server.rdb_child_pid == -1 && server.aof_child_pid == -1) {
        /* Resize */
        for (j = 0; j < dbs_per_call; j++) {
            tryResizeHashTables(resize_db % server.dbnum);
            resize_db++;
        }
        /* Rehash */
        if (server.activerehashing) {
            for (j = 0; j < dbs_per_call; j++) {
                int work_done = incrementallyRehash(rehash_db);
                if (work_done) {
                    /* If the function did some work, stop here, we'll do
                     * more at the next cron loop. */
                    break;
                } else {
                    /* If this db didn't need rehash, we'll try t
```

```
he next one. */
                    rehash_db++;
                    rehash_db %= server.dbnum;
                }
            }
        }
    }
}
/* Our hash table implementation performs rehashing incrementally while
 * we write/read from the hash table. Still if the server is idle, the hash
 * table will use two tables for a long time. So we try to use 1 millisecond
 * of CPU time at every call of this function to perform some rehahsing.
 *
 * The function returns 1 if some rehashing was performed, otherwise 0
 * is returned. */
int incrementallyRehash(int dbid) {
    /* Keys dictionary */
    if (dictIsRehashing(server.db[dbid].dict)) {
        dictRehashMilliseconds(server.db[dbid].dict,1);
        return 1; /* already used our millisecond for this loop... */
    }
    /* Expires */
    if (dictIsRehashing(server.db[dbid].expires)) {
        dictRehashMilliseconds(server.db[dbid].expires,1);
        return 1; /* already used our millisecond for this loop... */
    }
    return 0;
}
```

5、redisObject

上文讲了那么多，可是我们从来没有讲过redis里的对象，那么hash表里的一个个对象都是什么呢?

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                            * LFU data (least significant 8 bits frequency
                            * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
} robj;
```

redis里的对象有11种，之多，他们全都继承于redisObject。

```
/* Objects encoding. Some kind of objects like Strings and Hashes can be
 * internally represented in multiple ways. The 'encoding' field of the object
 * is set to one of this fields for this object. */
#define OBJ_ENCODING_RAW 0     /* Raw representation */
#define OBJ_ENCODING_INT 1     /* Encoded as integer */
#define OBJ_ENCODING_HT 2      /* Encoded as hash table */
#define OBJ_ENCODING_ZIPMAP 3  /* Encoded as zipmap */
#define OBJ_ENCODING_LINKEDLIST 4 /* No longer used: old list encoding. */
#define OBJ_ENCODING_ZIPLIST 5 /* Encoded as ziplist */
#define OBJ_ENCODING_INTSET 6  /* Encoded as intset */
#define OBJ_ENCODING_SKIPLIST 7  /* Encoded as skiplist */
#define OBJ_ENCODING_EMBSTR 8  /* Embedded sds string encoding */
#define OBJ_ENCODING_QUICKLIST 9 /* Encoded as linked list of ziplists */
#define OBJ_ENCODING_STREAM 10 /* Encoded as a radix tree of listpacks */
```