字典是一种用于保存键值对(key value pair)的抽象数据结构。在字典中，一个键和一个值进行关联，就是所谓的键值对。字典中的每个键都是独一无二的，可以根据键查找、更新值，或者删除整个键值对等等。

字典在Redis中的应用相当广泛，如Redis的数据库就是使用字典来作为底层实现的，对数据库的增、删、查、改操作也是构建在对字典的操作之上的。比如下面的命令：

redis**>** SET msg "hello world"

OK

该命令会在Redis数据库中创建一个键为”msg”，值为”hello world”的键值对，该键值对保存在代表数据库的字典中。

除了表示数据库之外，字典还是Redis中哈希键的底层实现之一，当一个哈希键包含的键值对比较多，或者键值对中的元素都是比较长的字符串时，Redis就会使用字典作为哈希键的底层实现。

Redis中，字典使用哈希表作为底层实现。有关字典和哈希表的结构体都定义在dict.h中，实现在dict.c中。

**1：哈希表节点**

哈希表节点使用dictEntry结构表示，dictEntry结构用来保存键值对。该结构体定义如下：

**typedef** struct dictEntry **{**

void **\***key**;**

union **{**

void **\***val**;**

uint64\_t u64**;**

int64\_t s64**;**

double d**;**

**}** v**;**

struct dictEntry **\***next**;**

**}** dictEntry**;**

key保存键值对中的键，而v保存键值对中的值，v可以是一个指针，一个uint64\_t整数，又或者是一个int64\_ t整数。

next是指向另一个哈希表节点的指针，利用该指针，可将多个哈希值相同的dictEntry连接在一起，来解决哈希表中的键冲突的问题。比如下图：

digraph {

    label = "\n 图 4-2    连接在一起的键 k1 和键 k0";

    rankdir = LR;

    //

    node [shape = record];

    dictht [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 2"];

    table [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];

    dictEntry0 [label = " <head> dictEntry | { k0 | v0 }"];
    dictEntry1 [label = " <head> dictEntry | { k1 | v1 }"];

    //

    node [shape = plaintext, label = "NULL"];

    null0;
    null1;
    null2;
    null3;

    //

    dictht:table -> table:head;

    table:0 -> null0;
    table:1 -> null1;
    table:2 -> dictEntry1;
    dictEntry1 -> dictEntry0 -> null2 [label = "next"];
    table:3 -> null3;

}

**2：哈希表**

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节点，每个节点保存字典中的一个键值对。哈希表的结构体定义在dict.h中：

**typedef** struct dictht **{**

dictEntry **\*\***table**;**

unsigned long size**;**

unsigned long sizemask**;**

unsigned long used**;**

**}** dictht**;**

table成员是一个指针数组，数组中的每个元素都是一个指向dictEntry结构的指针，每个dictEntry结构保存着一个键值对。

size成员记录了哈希表的大小，也就是table数组的大小，used成员则记录了哈希表目前已有节点（键值对）的数量。sizemask成员的值总是等于size-1，该值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。下图展示了一个大小为4的空哈希表：

digraph {

    label = "\n 图 4-1    一个空的哈希表";

    rankdir = LR;

    //

    node [shape = record];

    dictht [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 0"];

    table [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];

    //

    node [shape = plaintext, label = "NULL"];

    null0;
    null1;
    null2;
    null3;

    //

    dictht:table -> table:head;

    table:0 -> null0;
    table:1 -> null1;
    table:2 -> null2;
    table:3 -> null3;

}

**3：字典**

字典结构体定义如下：

**typedef** struct dict **{**

dictType **\***type**;**

void **\***privdata**;**

dictht ht**[**2**];**

long rehashidx**;** */\* rehashing not in progress if rehashidx == -1 \*/*

int iterators**;** */\* number of iterators currently running \*/*

**}** dict**;**

ht是一个包含两个哈希表元素dictht的数组。一般情况下，字典只使用哈希表ht[0]，而ht[1]只会在进行rehash时使用。另一个和rehash有关的属性就是rehashidx，它表示rehash目前的进度，如果目前没有在进行rehash，那么它的值为-1。rehash的介绍见下文。

type是一个指向dictType结构的指针，每个dictType结构保存了一组函数指针，这些函数用于操作特定类型的键值对，Redis会为用途不同的字典设置不同的类型特定函数。而privdata则保存了需要传给这些类型特定函数的可选参数。dictType结构定义如下：

**typedef** struct dictType **{**

unsigned int **(\***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**;**

这些函数的用途可以轻易的从函数名看出来，不再赘述。

下图就是一个普通状态下（没有进行rehash）的字典：

digraph {

    label = "\n 图 4-3    普通状态下的字典";

    rankdir = LR;

    //

    node [shape = record];

    dict [label = " <head> dict | type | privdata | <ht> ht | rehashidx \n -1 "];

    dictht0 [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 2"];

    dictht1 [label = " <head> dictht | <table> table | <size> size \n 0 | <sizemask> sizemask \n 0 | <used> used \n 0"];

    table0 [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];
    table1 [label = "NULL", shape = plaintext];

    dictEntry0 [label = " <head> dictEntry | { k0 | v0 }"];
    dictEntry1 [label = " <head> dictEntry | { k1 | v1 }"];

    //

    node [shape = plaintext, label = "NULL"];

    null0;
    null1;
    null2;
    null3;

    //

    dict:ht -> dictht0:head [label = "ht[0]"];
    dict:ht -> dictht1:head [label = "ht[1]"];

    dictht0:table -> table0:head;
    dictht1:table -> table1;

    table0:0 -> null0;
    table0:1 -> dictEntry0:head -> null1;
    table0:2 -> null2;
    table0:3 -> dictEntry1:head -> null3;
}

**4：哈希算法**

将一个新的键值对添加到字典中时，首先根据键计算出哈希值，然后根据哈希值计算出索引值，最后根据索引值，将包含新键值对的哈希表节点存储到哈希表数组中的指定索引上。

*//使用哈希函数，计算key的哈希值*

hash **=** dict**->**type**->**hashFunction**(**key**);**

*//使用哈希表的sizemask属性，根据哈希值计算出索引值*

index **=** hash **&** dict**->**ht**[**x**].**sizemask**;**

比如，针对一个长度为4的哈希表来说，要将一个键值对k0和v0添加到字典中，先使用语句：hash = dict->type->hashFunction(k0); 计算出键k0的哈希值，假设得到的哈希值为8，则接着用：index = hash & dict->ht[0].sizemask; 得到索引值（8 & 3 = 0）。最终，将包含键值对k0和v0的节点放置到哈希表数组的索引0上，如下图：

digraph {

    label = "\n 图 4-5    添加键值对 k0 和 v0 之后的字典";

    rankdir = LR;

    //

    node [shape = record];

    dict [label = " <head> dict | type | privdata | <ht> ht | rehashidx \n -1 "];

    dictht0 [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 1"];

    dictht1 [label = "...", shape = plaintext];

    table0 [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];
    //table1 [label = "NULL", shape = plaintext];

    dictEntry [label = " <head> dictEntry | { k0 | v0 } "];

    //

    node [shape = plaintext, label = "NULL"];

    null0;
    null1;
    null2;
    null3;

    //

    dict:ht -> dictht0:head [label = "ht[0]"];
    dict:ht -> dictht1:head [label = "ht[1]"];

    dictht0:table -> table0:head;
    //dictht1:table -> table1;

    table0:0 -> dictEntry:head -> null0;
    table0:1 -> null1;
    table0:2 -> null2;
    table0:3 -> null3;

}

当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。这种算法的优点在于，即使输人的键是有规律的，算法仍能给出一个很好的随机分布性，并且算法的计算速度也非常快。关于MurmurHash算法的更多信息可以参考该算法的主页：http://code.google.com/p/smhasher/。

**5：解决键冲突**

当两个以上的键计算得到的哈希值一样时，称这些键发生了冲突。Redis的哈希表使用链接法来解决键冲突。通过哈希表节点dictEntry的next指针，将多个哈希表节点链接成一个单向链表。

因dictEntry节点组成的链表没有指向链表表尾的指针，为了速度考虑，总是将新节点添加到链表的表头位置。如下图，就是用链接法解决k1和k2的冲突：



**6：rehash**

给定一个具有m个槽位，存储了n个元素的哈希表T，定义T的负载因子为n/m，也就是一个链表中的平均元素数目。

Redis哈希表的负载因子计算方法是：

*//负载因子 = 哈希表已保存的节点数量 / 哈希表大小*

load\_factor **=** ht**[**0**].**used **/** ht**[**0**].**size

随着操作的不断进行，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太少时，需要对哈希表的大小进行相应的扩展或者收缩。这就是通过执行rehash操作来完成，rehash的步骤如下：

a：为字典的哈希表ht[1]分配空间，分配的空间大小取决于要执行的操作，以及ht[0].used 的值：

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2\*(ht[0].used)的2^n（2的n次幂）；

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（2的n次幂）；

收缩空间的函数是dictResize，分配空间的函数为dictExpand，它们的实现如下：

int dictResize**(**dict **\***d**)**

**{**

int minimal**;**

**if** **(!**dict\_can\_resize **||** dictIsRehashing**(**d**))** **return** DICT\_ERR**;**

minimal **=** d**->**ht**[**0**].**used**;**

**if** **(**minimal **<** DICT\_HT\_INITIAL\_SIZE**)**

minimal **=** DICT\_HT\_INITIAL\_SIZE**;**

**return** dictExpand**(**d**,** minimal**);**

**}**

int dictExpand**(**dict **\***d**,** unsigned long size**)**

**{**

dictht n**;** */\* the new hash table \*/*

unsigned long realsize **=** \_dictNextPower**(**size**);**

*/\* the size is invalid if it is smaller than the number of*

*\* elements already inside the hash table \*/*

**if** **(**dictIsRehashing**(**d**)** **||** d**->**ht**[**0**].**used **>** size**)**

**return** DICT\_ERR**;**

*/\* Rehashing to the same table size is not useful. \*/*

**if** **(**realsize **==** d**->**ht**[**0**].**size**)** **return** DICT\_ERR**;**

*/\* Allocate the new hash table and initialize all pointers to NULL \*/*

n**.**size **=** realsize**;**

n**.**sizemask **=** realsize**-**1**;**

n**.**table **=** zcalloc**(**realsize**\*sizeof(**dictEntry**\*));**

n**.**used **=** 0**;**

*/\* Is this the first initialization? If so it's not really a rehashing*

*\* we just set the first hash table so that it can accept keys. \*/*

**if** **(**d**->**ht**[**0**].**table **==** **NULL)** **{**

d**->**ht**[**0**]** **=** n**;**

**return** DICT\_OK**;**

**}**

*/\* Prepare a second hash table for incremental rehashing \*/*

d**->**ht**[**1**]** **=** n**;**

d**->**rehashidx **=** 0**;**

**return** DICT\_OK**;**

**}**

dictResize对字典d进行收缩，主要是通过dictExpand实现的。收缩后的空间是DICT\_HT\_INITIAL\_SIZE(4)和d->ht[0].used两值中的较大者。该函数对应着rehash前的收缩操作。

dictExpand函数要么用来在创建字典时为哈希表ht[0]分配空间，要么用来在rehash之前为ht[1]分配空间。

参数size为分配空间的基准值，实际要分配空间的大小realsize为大于等于size的2的n次幂，但是realsize最小为DICT\_HT\_INITIAL\_SIZE(4)。比如size为1，2，3或4，则realsize为4；size为17，则realsize为32等等。

如果字典d当前正在进行rehash，或者ht[0].used大于size，或者ht[0]的当前size值等于realsize，则直接报错退出！

然后，开始初始化一个哈希表n，并且为其table申请空间。如果字典d的ht[0]为NULL，则直接：ht[0] = n; 否则，表示将要进行rehash： d->ht[1] = n; d->rehashidx = 0;

b：重新计算ht[0]中每个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。

c：当ht[0]上所有键值对都迁移到ht[1]之后，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。

dictRehash的函数实现如下：

int dictRehash**(**dict **\***d**,** int n**)** **{**

int empty\_visits **=** n**\***10**;** */\* Max number of empty buckets to visit. \*/*

**if** **(!**dictIsRehashing**(**d**))** **return** 0**;**

**while(**n**--** **&&** d**->**ht**[**0**].**used **!=** 0**)** **{**

dictEntry **\***de**,** **\***nextde**;**

*/\* Note that rehashidx can't overflow as we are sure there are more*

*\* elements because ht[0].used != 0 \*/*

assert**(**d**->**ht**[**0**].**size **>** **(**unsigned long**)**d**->**rehashidx**);**

**while(**d**->**ht**[**0**].**table**[**d**->**rehashidx**]** **==** **NULL)** **{**

d**->**rehashidx**++;**

**if** **(--**empty\_visits **==** 0**)** **return** 1**;**

**}**

de **=** d**->**ht**[**0**].**table**[**d**->**rehashidx**];**

*/\* Move all the keys in this bucket from the old to the new hash HT \*/*

**while(**de**)** **{**

unsigned int h**;**

nextde **=** de**->**next**;**

*/\* Get the index in the new hash table \*/*

h **=** dictHashKey**(**d**,** de**->**key**)** **&** d**->**ht**[**1**].**sizemask**;**

de**->**next **=** d**->**ht**[**1**].**table**[**h**];**

d**->**ht**[**1**].**table**[**h**]** **=** de**;**

d**->**ht**[**0**].**used**--;**

d**->**ht**[**1**].**used**++;**

de **=** nextde**;**

**}**

d**->**ht**[**0**].**table**[**d**->**rehashidx**]** **=** **NULL;**

d**->**rehashidx**++;**

**}**

*/\* Check if we already rehashed the whole table... \*/*

**if** **(**d**->**ht**[**0**].**used **==** 0**)** **{**

zfree**(**d**->**ht**[**0**].**table**);**

d**->**ht**[**0**]** **=** d**->**ht**[**1**];**

\_dictReset**(&**d**->**ht**[**1**]);**

d**->**rehashidx **=** **-**1**;**

**return** 0**;**

**}**

*/\* More to rehash... \*/*

**return** 1**;**

**}**

参数n表示要进行rehash的步数（要进行rehash的buckets数量）。如果所有bucket都是有内容的（链表非空），则该函数会进行n个bucket的rehash操作。但可能有些bucket是空的（空链表），所以，该函数总共会跳过10\*n个空bucket。因此，在遇到一个真正有内容的bucket之前，如果存在10\*n个以上的空bucket，该函数只是跳过10\*n个空bucket，直接返回1，而不进行任何rehash操作。

d->rehashidx表示在d->ht[0]哈希表中要进行rehash操作的bucket的索引。在dictExpand中它被置为0，表示从d->ht[0].table[0]开始进行rehash操作。每次rehash操作之前，都要保证rehashidx的值小于d->ht[0].size。

每一步rehash操作，首先从rehashidx开始找到第一个有内容的bucket，如果在找到之前，遍历过的空bucket的数量超过了10\*n个，则直接返回1.

找到要进行rehash操作的ht[0]中的bucket之后，遍历该bucket中的链表，对其中的每个节点进行rehash，首先计算该节点在d->ht[1].table中所在bucket的索引，然后插入到ht[1]的该bucket中的链表中。

遍历完ht[0]中的该bucket的链表后，将该bucket置空，并且rehashidx++，开始进行下一步rehash。

遍历完n个bucket之后，会判断d->ht[0]中的节点是否都已经rehash完成，如果已全部完成，则释放d->ht[0].table，将ht[1]置为ht[0]，并初始化新的ht[1]，置rehashidx为-1，最后返回0，表示rehash已完成。如果ht[0]中尚有节点未进行rehash，则直接返回1。

比如，要对下面的字典进行rehash操作：

digraph {

    label = "\n 图 4-8    执行 rehash 之前的字典";

    rankdir = LR;

    node [shape = record];

    // 字典

    dict [label = " <head> dict | type | privdata | <ht> ht | rehashidx \n -1 "];

    // 哈希表

    dictht0 [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 4"];

    dictht1 [label = " <head> dictht | <table> table | <size> size \n 0 | <sizemask> sizemask \n 0 | <used> used \n 0"];

    table0 [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];

    table1 [label = "NULL", shape = plaintext];

    // 哈希表节点

    kv0 [label = " <head> dictEntry | { k0 | v0 } "];
    kv1 [label = " <head> dictEntry | { k1 | v1 } "];
    kv2 [label = " <head> dictEntry | { k2 | v2 } "];
    kv3 [label = " <head> dictEntry | { k3 | v3 } "];

    //

    node [shape = plaintext, label = "NULL"];

    null0;
    null1;
    null2;
    null3;

    //

    dict:ht -> dictht0:head [label = "ht[0]"];
    dict:ht -> dictht1:head [label = "ht[1]"];

    dictht0:table -> table0:head;
    dictht1:table -> table1;

    table0:0 -> kv2:head -> null0;
    table0:1 -> kv0:head -> null1;
    table0:2 -> kv3:head -> null2;
    table0:3 -> kv1:head -> null3;

}

ht[0].used当前的值为4，4\*2=8，而8恰好是2的3次方，所以将ht[1]哈希表的大小设置为8。如下图：

digraph {

    label = "\n 图 4-9    为字典的 ht[1] 哈希表分配空间";

    rankdir = LR;

    node [shape = record];

    // 字典

    dict [label = " <head> dict | type | privdata | <ht> ht | rehashidx \n -1 "];

    // 哈希表

    dictht0 [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 4"];

    dictht1 [label = " <head> dictht | <table> table | <size> size \n 8 | <sizemask> sizemask \n 7 | <used> used \n 0"];

    table0 [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];

    table1 [label = " <head> dictEntry*[8] | <0> 0 | <1> 1 | <2> 2 | ... | <7> 7 "];

    // 哈希表节点

    kv0 [label = " <head> dictEntry | { k0 | v0 } "];
    kv1 [label = " <head> dictEntry | { k1 | v1 } "];
    kv2 [label = " <head> dictEntry | { k2 | v2 } "];
    kv3 [label = " <head> dictEntry | { k3 | v3 } "];

    //

    node [shape = plaintext, label = "NULL"];

    //

    dict:ht -> dictht0:head [label = "ht[0]"];
    dict:ht -> dictht1:head [label = "ht[1]"];

    dictht0:table -> table0:head;
    dictht1:table -> table1:head;

    table0:0 -> kv2:head -> null0;
    table0:1 -> kv0:head -> null1;
    table0:2 -> kv3:head -> null2;
    table0:3 -> kv1:head -> null3;

    table1:0 -> null10;
    table1:1 -> null11;
    table1:2 -> null12;
    table1:7 -> null17;

}

然后将ht[0]包含的四个键值对都rehash到ht[1]上，如下图所示：

digraph {

    label = "\n 图 4-10    ht[0] 的所有键值对都已经被迁移到 ht[1]";

    rankdir = LR;

    node [shape = record];

    // 字典

    dict [label = " <head> dict | type | privdata | <ht> ht | rehashidx \n -1 "];

    // 哈希表

    dictht0 [label = " <head> dictht | <table> table | <size> size \n 4 | <sizemask> sizemask \n 3 | <used> used \n 0"];

    dictht1 [label = " <head> dictht | <table> table | <size> size \n 8 | <sizemask> sizemask \n 7 | <used> used \n 4"];

    table0 [label = " <head> dictEntry*[4] | <0> 0 | <1> 1 | <2> 2 | <3> 3 "];

    table1 [label = " <head> dictEntry*[8] | ... | <1> 1 | ... | <4> 4 | <5> 5 | ... | <7> 7 "];

    // 哈希表节点

    kv0 [label = " <head> dictEntry | { k0 | v0 } "];
    kv1 [label = " <head> dictEntry | { k1 | v1 } "];
    kv2 [label = " <head> dictEntry | { k2 | v2 } "];
    kv3 [label = " <head> dictEntry | { k3 | v3 } "];

    //

    node [shape = plaintext, label = "NULL"];

    //

    dict:ht -> dictht0:head [label = "ht[0]"];
    dict:ht -> dictht1:head [label = "ht[1]"];

    dictht0:table -> table0:head;
    dictht1:table -> table1:head;

    table0:0 -> null0;
    table0:1 -> null1;
    table0:2 -> null2;
    table0:3 -> null3;

    table1:1 -> kv3:head -> null11;
    table1:4 -> kv2:head -> null14;
    table1:5 -> kv0:head -> null15;
    table1:7 -> kv1:head -> null17;

}

最后，释放ht[0]，将ht[1]设置为ht[0]，然后为ht[1]分配一个空白哈希表。至此，对哈希表的扩展操作执行完毕，将哈希表的大小从原来的4改为了现在的8。如下图所示。

digraph {

    label = "\n 图 4-11    完成 rehash 之后的字典";

    rankdir = LR;

    node [shape = record];

    // 字典

    dict [label = " <head> dict | type | privdata | <ht> ht | rehashidx \n -1 "];

    // 哈希表

    dictht0 [label = " <head> dictht | <table> table | <size> size \n 8 | <sizemask> sizemask \n 7 | <used> used \n 4"];

    dictht1 [label = " <head> dictht | <table> table | <size> size \n 0 | <sizemask> sizemask \n 0 | <used> used \n 0"];

    table0 [label = " <head> dictEntry*[8] | ... | <1> 1 | ... | <4> 4 | <5> 5 | ... | <7> 7 "];

    table1 [label = "NULL", shape = plaintext];

    // 哈希表节点

    kv0 [label = " <head> dictEntry | { k0 | v0 } "];
    kv1 [label = " <head> dictEntry | { k1 | v1 } "];
    kv2 [label = " <head> dictEntry | { k2 | v2 } "];
    kv3 [label = " <head> dictEntry | { k3 | v3 } "];

    //

    node [shape = plaintext, label = "NULL"];

    //

    dict:ht -> dictht0:head [label = "ht[0]"];
    dict:ht -> dictht1:head [label = "ht[1]"];

    dictht0:table -> table0:head;
    dictht1:table -> table1;

    table0:1 -> kv3:head -> null11;
    table0:4 -> kv2:head -> null14;
    table0:5 -> kv0:head -> null15;
    table0:7 -> kv1:head -> null17;

}

**7：rehash的时机**

当哈希表的负载因子小于0.1时，程序自动开始对哈希表执行收缩操作。

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：

a：若服务器目前没有在执行BGSAVE命令或者BGREWRITEAOF命令，且哈希表的负载因子大于等于1。

b：若服务器目前正在执行BGSAVE命令或者BGREWRITEAOF命令，且哈希表的负载因子大于等于5。

根据BGSAVE或BGREWRITEAOF命令是否正在执行，服务器执行扩展操作所需的负载因子并不相同。

这是因为在执行这些命令的过程中，Redis要用fork创建当前服务器进程的子进程，大多数操作系统在fork时都采用写时复制技术来优化内存的使用效率。因此，在子进程存在期间，通过提高扩展操作所需的负载因子，减少进行哈希表扩展操作的可能，避免不必要的内存写人操作，从而最大限度地节约内存。

**8：渐进式rehash**

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]。

原因在于，如果哈希表里保存的键值对数量非常多时，要一次性的将所有键值对全部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。

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添加、删除、查找和更新操作上，从而避免了集中式rehash而带来的庞大计算量。以下是哈希表渐进式rehash的详细步骤：

a：为ht[1]分配空间，让字典同时持有ht [0]和ht[1]两个哈希表。

b：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正式开始。

c：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，除了执行指定的操作以外，还会顺带将ht[0]在rehashidx索引上的所有键值对rehash到ht[1]上，当rehash工作完成之后，rehashidx++。

d：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。

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，字典的查找、删除、更新等操作会在两个哈希表上进行。例如，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。下面就是查找过程的实现：

dictEntry **\***dictFind**(**dict **\***d**,** const void **\***key**)**

**{**

dictEntry **\***he**;**

unsigned int h**,** idx**,** table**;**

**if** **(**d**->**ht**[**0**].**size **==** 0**)** **return** **NULL;** */\* We don't have a table at all \*/*

**if** **(**dictIsRehashing**(**d**))** \_dictRehashStep**(**d**);**

h **=** dictHashKey**(**d**,** key**);**

**for** **(**table **=** 0**;** table **<=** 1**;** table**++)** **{**

idx **=** h **&** d**->**ht**[**table**].**sizemask**;**

he **=** d**->**ht**[**table**].**table**[**idx**];**

**while(**he**)** **{**

**if** **(**dictCompareKeys**(**d**,** key**,** he**->**key**))**

**return** he**;**

he **=** he**->**next**;**

**}**

**if** **(!**dictIsRehashing**(**d**))** **return** **NULL;**

**}**

**return** **NULL;**

**}**

该函数中，如果字典当前正在rehash，则首先调用\_dictRehashStep进行1步rehash操作。

然后调用dictHashKey得到该key的哈希值；先得到该哈希值在ht[0]中对应的索引值，得到索引值之后，就在哈希表ht[0]相应的bucket中，对比链表中的每个节点，寻找该key，如果找到，则直接返回对应的dictEntry。

如果没找到，且字典当前正在rehash，则接着在ht[1]中继续寻找过程，否则，直接返回NULL。

如果处于rehash中，则字典的其他操作，如增加、更新和删除都会进行\_dictRehashStep操作，需要注意的是增加操作，新的键值对一律被保存到ht[1]上，而ht[0]不再进行任何添加操作，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。

其他关于redis的dict代码，可以参考：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/dict.c

另外，字典的迭代，因为rehash的缘故而变得复杂。下一篇文章介绍Redis中字典迭代的实现。