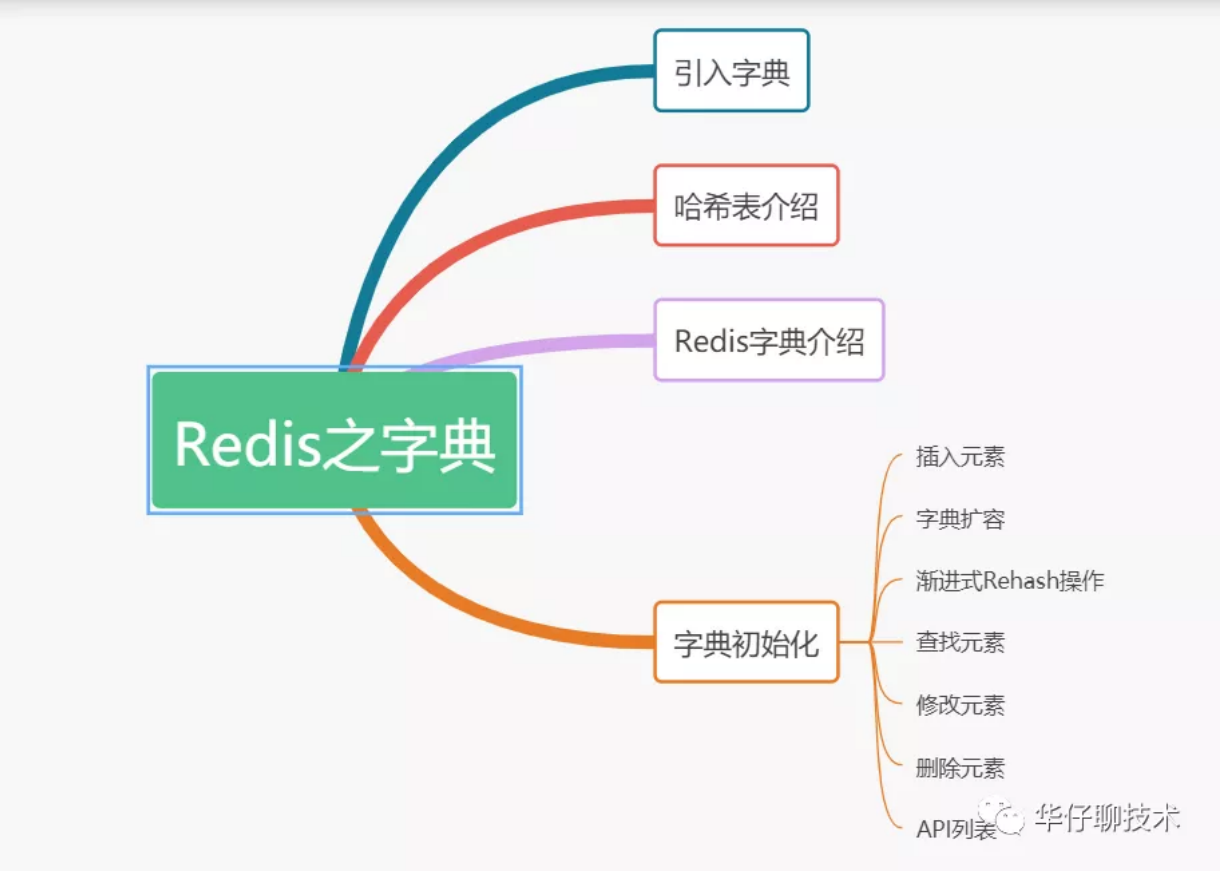
【Redis5.X源码分析】系列之字典

原创 王江华 华仔聊技术 2021-07-04 16:43

收录于话题
#面试 14 #redis 2



1 引入字典

从本文开始慢慢揭开Redis字典的神秘面纱，字典又称为散列表，用来存储键值对的一种数据结构，在很多高级语言中都有实现，比如PHP的数组，Redis的整个数据库都是用字典来进行存储的，对Redis数据库的CURD操作，实际就是对字典中的数据进行CURD。由此可以得出字典的特征

1). 可以存储海量数据，KV对是映射关系，可以根据键以O(1)的时间复杂度读取或者插入KV对。
2). KV对的键的类型可以是字符串，整型等，且唯一。
3). KV对中值的类型可以是String，Hash，List，Set，SortedSet。

2 哈希表介绍

Redis 本质上就是在字典上面挂载着各种数据结构，我们先来看看 字典 这种数据结构。Redis中的 字典 其实是就是 哈希表（HashTable），我们来看看 哈希表 的定义：

哈希表（HashTable）是根据键值(Key)直接进行访问的数据结构。也就是说，它通过把键值映射到哈希表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

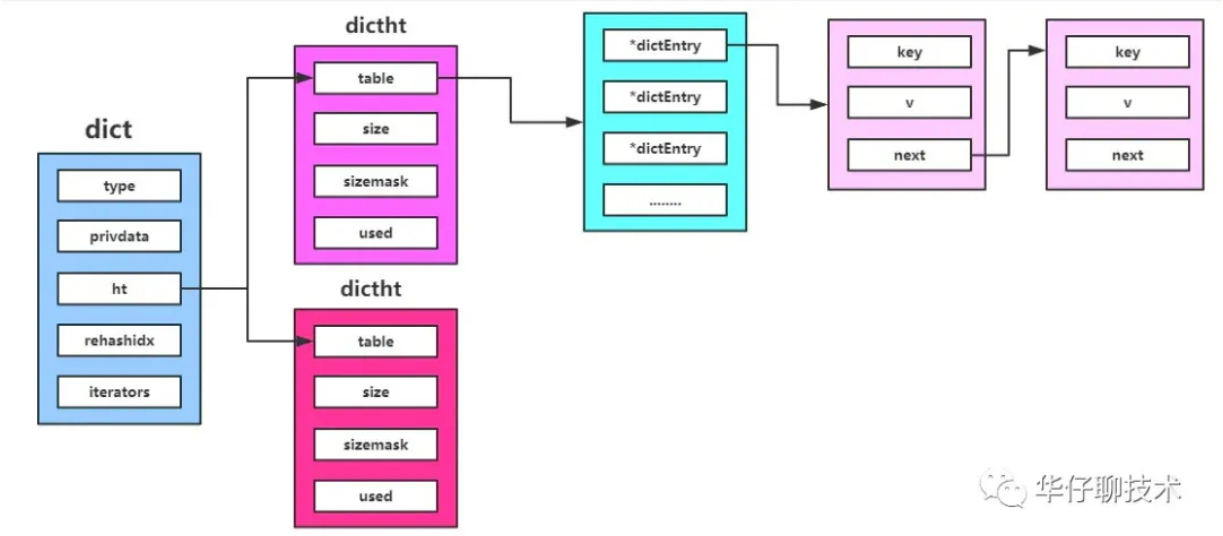
3 Redis字典介绍

Redis的字典也是通过哈希表设计来实现的。当然在通用性和性能上面考虑并优化了一些。接下来让我们来看下Redis字典的具体实现。

首先让我们来看看在Redis中字典数据结构的定义：

```
typedef struct dictEntry {
    void *key;  /*存储键*/
    union {
        void *val;       /*db.dict中的val*/
        uint64_t u64;
        int64_t s64;    /*db.expires中存储过期时间*/
        double d;
    } v;      /*值是个联合体*/
    struct dictEntry *next;  /*当hash冲突时，指向冲突的元素，形成单链表*/
} dictEntry;

/* This is our hash table structure. Every dictionary has two of this as we
 * implement incremental rehashing, for the old to the new table. */
typedef struct dictht {
    dictEntry **table;      /*指针数组，用于存储键值对*/
    unsigned long size;  /*table数组的大小*/
    unsigned long sizemask;  /*掩码 = size - 1*/
    unsigned long used;  /*table数组已存元素个数，包含next单链表的数据*/
} dictht;

typedef struct dict {
    dictType *type;  /*该字典对应的特定操作函数*/
    void *privdata;    /*该字典依赖的数据*/
    dictht ht[2];    /*Hash表，键值对存储在这里*/
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    unsigned long iterators; /* 当前运行的迭代函数 */
} dict;
```

关系结构图如下:



Redis的字典实现主要依赖的数据结构包括三部分：dict，dictht，dictEntry节点。dict中嵌入了2个dictht表，dictht表中的table字典存放着dictEntry

下面介绍一下这些结构体的各个字段作用：

dict结构
type：是用户自定义的函数列表，主要用于插入数据到字典时进行的一些操作，比如计算key哈希值的hashFunction 函数句柄。
privdata：创建字典时指定的私有数据，一般提供给 type 字段中的函数句柄使用。
ht[2]：类型为 dictht 结构的数组，这个数组有两个元素，而 dictht 结构主要用于保存数据，一般情况下只用ht[0],只有当字典扩容，缩容需要进行rehash时才会用到ht[1].
rehashidx：rehash操作过程中最后一次处理的桶索引。
iterators：用于迭代字典中的数据。

dictht结构
table：类型为 dictEntry 结构指针的数组，用于保存数据，每个 key-value 的数据对通过类型为 dictEntry 的结构来存储。
size：table数组的大小。
sizemark：用于取模，得到一个 0 ～ size 的索引值。恒等于size-1
used：表示字典中有多少个元素。包含next单链表数据

dictEntry结构
key：数据的键，主要用于查找数据。
v：数据的值，数据主要通过这个字段来存储。
next：用于解决Hash冲突，把冲突的key连接起来（拉链法）。

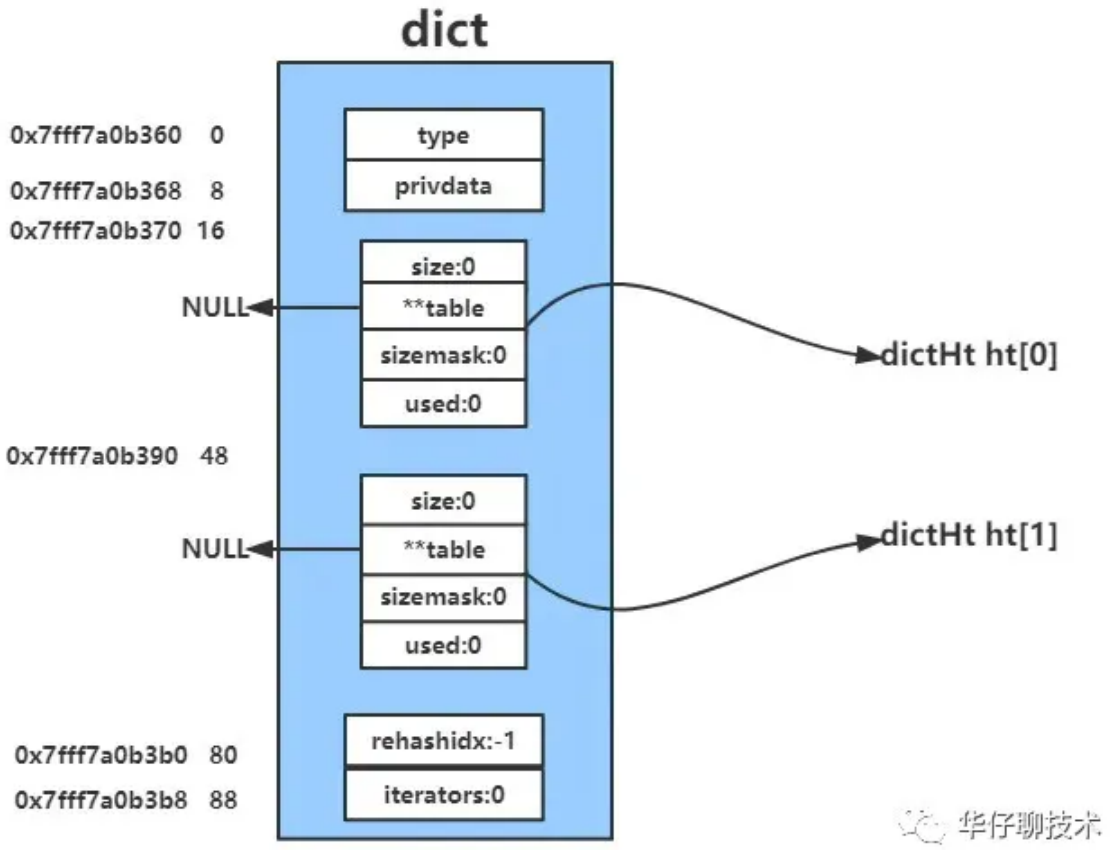
4 字典初始化

在redis-server启动中，整个数据库会先初始化一个空的字典用于存储整个数据库的键值对，初始化一个空字典，主要调用的是dict.h文件中的dictCreate函数，对应的源码为:

```
/* Reset a hash table already initialized with ht_init().
 * NOTE: This function should only be called by ht_destroy(). */
static void _dictReset(dictht *ht)
{
    ht->table = NULL;
    ht->size = 0;
    ht->sizemask = 0;
    ht->used = 0;
}

/* 创建一个新的hash表 */
dict *dictCreate(dictType *type,
        void *privDataPtr)
{
    dict *d = zmalloc(sizeof(*d));  //96字节
    _dictInit(d,type,privDataPtr);  //结构体初始化值
    return d;
}

/* 初始化hash表 */
```

```
int _dictInit(dict *d, dictType *type,
        void *privDataPtr)
{
    _dictReset(&d->ht[0]);
    _dictReset(&d->ht[1]);
    d->type = type;
    d->privdata = privDataPtr;
    d->rehashidx = -1;
    d->iterators = 0;
    return DICT_OK;
}
```

dictCreate函数初始化一个空字典的主要步骤为：申请空间，调用_dictInit函数，给字典的各个字段赋予初始值。初始化后，一个字典内存占用情况如下图所示：



下面来看看在Redis中怎么创建一个字典的：

```
------server.c------
/* Command table. sds string -> command struct pointer. */
dictType commandTableDictType = {
    dictSdsCaseHash,            /* hash function */
    NULL,                       /* key dup */
    NULL,                       /* val dup */
    dictSdsKeyCaseCompare,      /* key compare */
    dictSdsDestructor,          /* key destructor */
    NULL                        /* val destructor */
};

/*初始化服务端配置*/
void initServerConfig(void) {
    ........
    server.commands = dictCreate(&commandTableDictType,NULL);
    server.orig_commands = dictCreate(&commandTableDictType,NULL);
```

```
    ........
}
```

创建字典时，需要提供 dictType 参数，而这个参数主要定义了插入数据到字典时进行的一些操作，比如插入数据时key是否要进行复制的keyDup函数，那么我们来看看 dictType 的定义：

```
typedef struct dictType {
    uint64_t (*hashFunction)(const void *key);  /*用于计算键的哈希值*/
    void *(*keyDup)(void *privdata, const void *key);  /*用于复制数据的键*/
    void *(*valDup)(void *privdata, const void *obj);  /*用于复制数据的值*/
    /*用于比较键是否相等*/
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);
    void (*keyDestructor)(void *privdata, void *key);  /*用于释放复制出来的键的内存*/
    void (*valDestructor)(void *privdata, void *obj);  /*用于释放复制出来的值的内存*/
} dictType;
```

插入元素

redis-server启动后，再启动redis-client连上server，执行命令 “set hello world”：

127.0.0.1:6379> set hello world

Server端收到命令后，会执行void setKey(redisDb *db, robj *key, robj *val);根据之前介绍字典的特性，每个键必须是唯一的，主要流程如下:

1). 调用dictFind函数，查找键是否存在，则调用dbOverwrite函数修改键值对，否则调用dictAdd函数添加元素

2). dbAdd会调用dict.h中的dictAdd函数插入键值对.

dictAdd函数源码如下：

```
/* 调用之前会查找key是否存在，不存在则调用dictAdd函数 */
int dictAdd(dict *d, void *key, void *val)
{
    /*添加键，字典中键已存在则返回NULL,否则添加键到新节点中，返回新节点*/
    dictEntry *entry = dictAddRaw(d,key,NULL);
    if (!entry) return DICT_ERR;    /*键存在则返回错误*/
    dictSetVal(d, entry, val);  /*设置新值*/
    return DICT_OK;
}
```

而dictAdd() 函数主要还是通过调用 dictAddRaw() 函数来完成插入操作，dictAddRaw() 函数的代码如下：

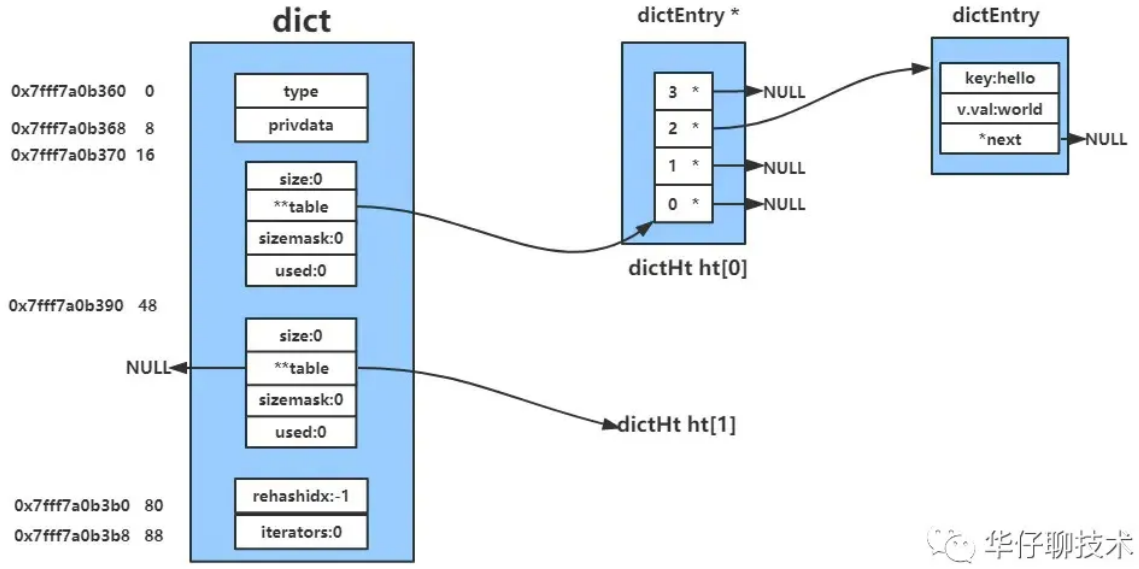
```
/*入参字典，键，Hash表节点地址*/
dictEntry *dictAddRaw(dict *d, void *key, dictEntry **existing)
{
    long index;
    dictEntry *entry;
    dictht *ht;
    /*该字典是否在进行rehash操作，如果是则执行一次rehash*/
```

```
    if (dictIsRehashing(d)) _dictRehashStep(d);
    /*查找键，找到则直接返回-1，并把老节点存入existing字段，否则把新节点的索引值返回，
    如果遇到Hash表容量不足，则进行扩容*/
    if ((index = _dictKeyIndex(d, key, dictHashKey(d,key), existing)) == -1)
        return NULL;
    /*是否进行rehash操作中，如果是则插入到散列表ht[1]中，否则插入到散列表ht[0]*/
    ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
    /*申请新节点内存，插入散列表中，给新节点存入键信息*/
    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;
}
```

dictAddRaw() 函数主要完成以下几个工作：

1). 判断是否正在进行rehash操作（dictIsRehashing() 判断为真），如果是就调用 _dictRehashStep() 函数进行rehash。
2). 通过调用 _dictKeyIndex() 函数计算key对应所在哈希表的位置（索引）index。
3).如果正在进行rehash，那么就使用ht数组的第二个哈希表，否则就用第一个
4).创建一个 dictEntry 结构用于保存数据的键和值。

dictAddRaw() 函数会返回一个类型为 dictEntry 结构的值，然后 dictAdd() 函数通过调用 dictSetVal() 函数设置其值。
插入元素，字典对应的内存占用结构如下图:



字典扩容

当哈希表中的数据个数超过一定数量时，哈希冲突的链表过长，从而导致查询效率降低，这个时候就需要Rehash操作。Rehash操作是将哈希表的数组扩大，从而减少哈希冲突的比率。当然扩大哈希表的数组

会导致之前的映射关系无效，所以需要把旧数据重新迁移到新的哈希表数组中。

Redis在插入数据到字典时，会通过 _dictExpandIfNeeded() 函数来判断是否需要进行Rehash操作，_dictExpandIfNeeded() 函数代码如下：

```
static int _dictExpandIfNeeded(dict *d)
{
    if (dictIsRehashing(d)) return DICT_OK;

    if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_SIZE);

    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;
}
```

_dictExpandIfNeeded() 函数主要完成3个工作：

1). 通过 dictIsRehashing() 来判断字典是否正在Rehash操作，如果是就直接返回OK，不需要再进行Rehash。
2). 如果字典的第一个哈希表的大小为0，表示需要对第一个哈希表进行初始化。
3). 如果第一个哈希表的元素个数大于等于哈希表的大小，那么就对第一个哈希表进行Rehash操作（把哈希表的大小扩大为原来的2倍）。

进行Rehash操作通过调用 dictExpand() 函数来完成，扩容对应的源码是dict.h文件中的dictExpand函数，源码如下:

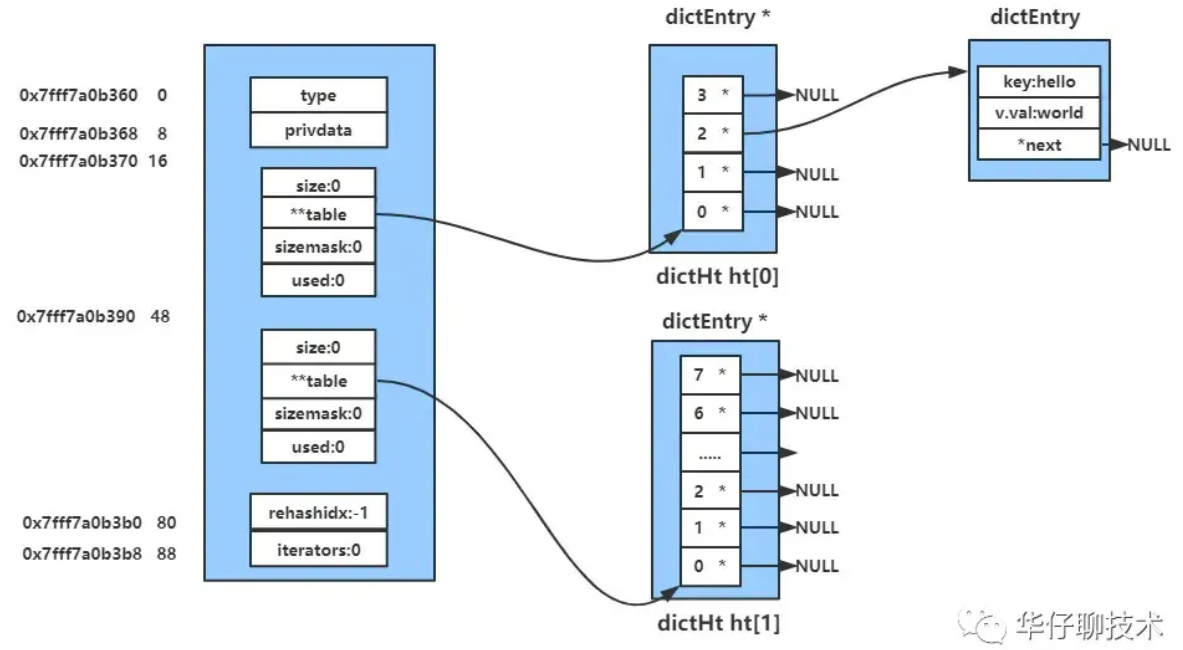
```
/* Expand or create the hash table */
int dictExpand(dict *d, unsigned long size)   /*传入size = d->ht[0].used * 2 */
{
    /* the size is invalid if it is smaller than the number of
     * elements already inside the hash table */
    if (dictIsRehashing(d) || d->ht[0].used > size)
        return DICT_ERR;

    dictht n; /* the new hash table */
    /*重新计算扩容后的值，必须为2的N次方幂*/
    unsigned long realsize = _dictNextPower(size);

    /* 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;  /*扩容后的新内存放入ht[1]中*/
    d->rehashidx = 0; /*非-1，表示需要进行rehash*/
    return DICT_OK;
}
```

dictExpand() 函数比较简单，就是申请一个更大的哈希数组，如果第一个哈希表的哈希数组为空，那么就把第一个哈希表的哈希数组设置为新的哈希数组。否则将第二个哈希表的哈希数组设置为新的哈希数组。

扩容的主要流程如下:

1)．申请一块新内存后，初次申请是默认大小为4个dictEntry；非初次申请时，申请内存的大小为当前Hash表容量的一倍。
2）把新申请的内存地址赋值给ht[1],并把字典的rehashidx标识从-1改为0，表示只有需要进行rehash操作，此时字典的内存结构如下



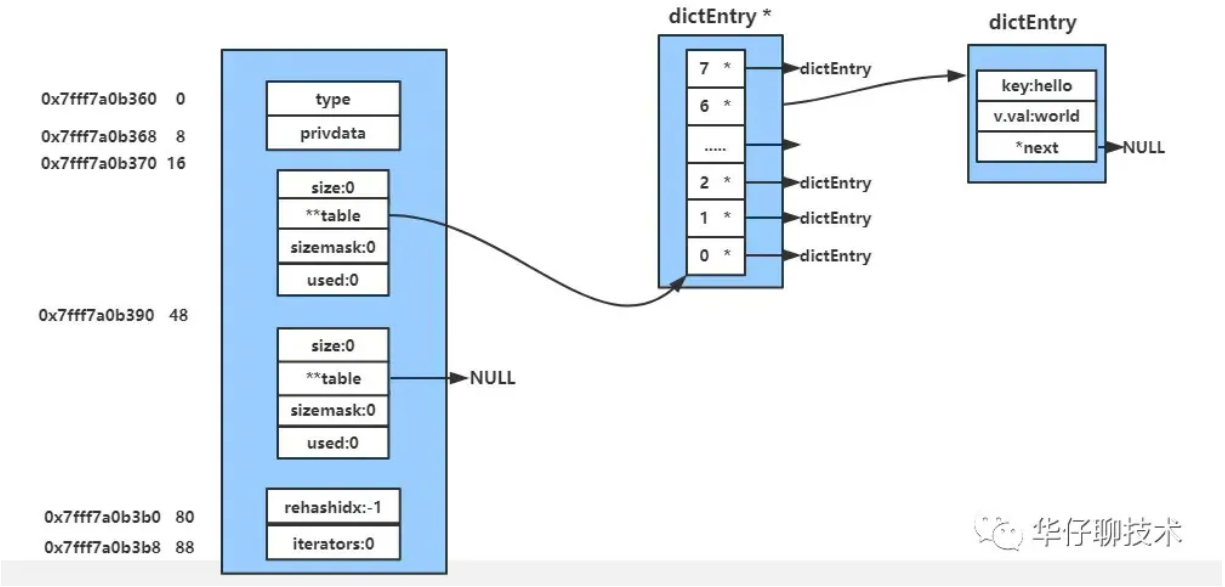
渐进式Rehash操作

从 dictExpand() 函数的实现来看，并没有在这里对数据进行Rehash操作，只是把哈希数组扩大2倍而已，那么Rehash操作在什么时候进行呢？对数据进行Rehash操作的触发点有很多个，如插入、删除和查找，当然最后都会调用 dictRehash() 函数来完成，我们来看看 dictRehash() 函数的实现：rehash 扩缩容操作都会进行触发。Redis的整个rehash实现，源码如下：

```
/* Performs N steps of incremental rehashing. Returns 1 if there are still
 * keys to move from the old to the new hash table, otherwise 0 is returned.
 *
 * Note that a rehashing step consists in moving a bucket (that may have more
 * than one key as we use chaining) from the old to the new hash table, however
```

```
 * since part of the hash table may be composed of empty spaces, it is not
 * guaranteed that this function will rehash even a single bucket, since it
 * will visit at max N*10 empty buckets in total, otherwise the amount of
 * work it does would be unbound and the function may block for a long time. */
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) {
            uint64_t 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;
}
```

dictRehash() 函数的第二个参数是指定了每次要对多少个槽进行Rehash（也就是冲突链表），Rehash操作就是遍历第一个哈希表的所有数据，然后重新计算key的哈希值，保存到第二个哈希表中，并且从第一个哈希表中删除。当第一个哈希表的元素个数为0时，就将第一个哈希表替换成第二个哈希表，并且完成Rehash操作。



查找元素

我们看下查找元素，从客户端读取hello的值：

127.0.0.1:6379> get hello “world”

Server端收到get命令后，最终要在字典中查找某个key键值对会执行dict.h中的dictFind()函数，源码如下:

```
dictEntry *dictFind(dict *d, const void *key)
{
    dictEntry *he;
    uint64_t h, idx, table;

    if (d->ht[0].used + d->ht[1].used == 0) return NULL; /* dict is empty */
    if (dictIsRehashing(d)) _dictRehashStep(d);
    h = dictHashKey(d, key);    /*得到键的Hash值*/
    for (table = 0; table <= 1; table++) {  /*遍历查找Hash表  ht[0]与ht[1]*/
        idx = h & d->ht[table].sizemask;  /*根据Hash值获取到对应的索引值*/
        he = d->ht[table].table[idx];  /*获取值*/
        while(he) {   /*如果存在值则遍历该值中的单链表*/
            if (key==he->key || dictCompareKeys(d, key, he->key))
                return he;     /*找到与键相等的值，返回该元素*/
            he = he->next;
        }
        if (!dictIsRehashing(d)) return NULL;  /*如果未进行rehash操作，则只读取ht[0]*/
    }
    return NULL;
}
```

通过上面的介绍说明，dictFind() 函数的实现也比较容易理解，主要进行了如下操作:

1). 如果字典中第一个和第二个哈希表都为空，那么就返回NULL。
2). 如果判断正在进行Rehash操作，调用 _dictRehashStep() 对数据进行分步Rehash。
3). 根据键调用Hash函数取得其Hash值
4). 遍历字典的2个Hash表，读取索引对应的元素
5). 根据Hash值取到索引值
6). 根据索引在hash表中找到元素值，并先在第一个哈希表中查找是否存在，如果存在就返回key对应的

值。如果key不在第一个哈希表中，那么就要判断当前是否正在Rehash操作，如果是就在第二个哈希表中查找key是否存在。因为在Rehash的过程中，key有可能被移动到第二个哈希表中。

7)． 找不到则返回NULL

修改元素

说完查找元素后，继续跟进修改字典中的键值对元素，客户端执行命令:

127.0.0.1:6379> set hello world2

OK

Server端收到set命令后，会查询键是否已经在数据库中存在，存在则执行db.c文件中的dbOverwrite函数，源码如下:

```
/* Overwrite an existing key with a new value. Incrementing the reference
 * count of the new value is up to the caller.
 * This function does not modify the expire time of the existing key.
 * The program is aborted if the key was not already present. */
void dbOverwrite(redisDb *db, robj *key, robj *val) {
    dictEntry *de = dictFind(db->dict,key->ptr);  /*查找键是否存在，返回存在的节点*/
    serverAssertWithInfo(NULL,key,de != NULL); /*不存在则中断执行*/
    dictEntry auxentry = *de;
    robj *old = dictGetVal(de);   /*获取老节点val字段值*/
    if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
        val->lru = old->lru;
    }
    dictSetVal(db->dict, de, val);   /*给节点设置新的值*/
    if (server.lazyfree_lazy_server_del) {
        freeObjAsync(old);
        dictSetVal(db->dict, &auxentry, NULL);
    }
    dictFreeVal(db->dict, &auxentry);   /*释放节点中旧val内存*/
}
```

删除元素

继续跟进删除字典中键值对，客户端执行命令：

127.0.0.1:6379> del hello

(integer) 1

Server收到del命令后，删除键值对会执行dict.h文件中的dictDelete函数,源码如下：

```
/* 查找并删除元素 */
static int dictDelete(dict *ht, const void *key) {
    unsigned int h;
    dictEntry *de, *prevde;

    if (ht->size == 0)
        return DICT_ERR;
    h = dictHashKey(ht, key) & ht->sizemask;
    de = ht->table[h];

    prevde = NULL;
    while(de) {
        if (dictCompareHashKeys(ht,key,de->key)) {  /*比对hash值*/
```

```
            /* Unlink the element from the list */
            if (prevde)
                prevde->next = de->next;
            else
                ht->table[h] = de->next;

            dictFreeEntryKey(ht,de);  /*释放该节点对应的键占用的内存*/
            dictFreeEntryVal(ht,de);  /*释放该节点对应的值占用的内存*/
            free(de);   /*释放本身占用内存*/
            ht->used--;  /*used -1*/
            return DICT_OK;
        }
        prevde = de;
        de = de->next;
    }
    return DICT_ERR; /* not found */
}
```

删除函数主要进行以下操作：

1). 查找该键释放存在该字典中。
2). 存在则把该节点从单链表中删除。
3). 释放该节点对应键占用的内存，值占用的内存，以及本身占用的内存。
4). 给对应的Hash表的used字典减1操作

API列表

前面讲解了字典概念以及难点，在此开始介绍字典相关的API函数列表，主要声明在dict.h中：

```
/* API */
dict *dictCreate(dictType *type, void *privDataPtr); /*初始化字典*/
int dictExpand(dict *d, unsigned long size);  /*字典扩容*/
int dictAdd(dict *d, void *key, void *val); /*添加键值对，已存在则不加*/
 /*添加key，并返回新添加的key对应的节点。若已存在，则存入existing字段中，并返回-1*/
dictEntry *dictAddRaw(dict *d, void *key, dictEntry **existing);
dictEntry *dictAddOrFind(dict *d, void *key); /*添加或者查找key*/
int dictReplace(dict *d, void *key, void *val);  /*添加键值对，若存在则修改，否则添加*/
int dictDelete(dict *d, const void *key);  /*删除元素*/
dictEntry *dictUnlink(dict *ht, const void *key);  /*删除key，但不释放内存*/
void dictFreeUnlinkedEntry(dict *d, dictEntry *he); /*释放dictUnlink函数删除key的内存*/
void dictRelease(dict *d);   /*释放字典*/
dictEntry * dictFind(dict *d, const void *key);  /*根据键查找元素*/
void *dictFetchValue(dict *d, const void *key);  /*根据键查找出值*/
int dictResize(dict *d);   /*扩缩容字典*/
dictIterator *dictGetIterator(dict *d); /*初始化普通迭代器*/
dictIterator *dictGetSafeIterator(dict *d);   /*初始化安全迭代器*/
dictEntry *dictNext(dictIterator *iter);  /*通过迭代器获取下一个节点*/
void dictReleaseIterator(dictIterator *iter);  /*释放迭代器*/
dictEntry *dictGetRandomKey(dict *d);  /*随机得到一个键*/
dictEntry *dictGetFairRandomKey(dict *d);
/*随机得到一些键*/
unsigned int dictGetSomeKeys(dict *d, dictEntry **des, unsigned int count);
void dictGetStats(char *buf, size_t bufsize, dict *d);  /*读取字典的状态，使用情况等*/
uint64_t dictGenHashFunction(const void *key, int len); /*hash函数 字母大小写敏感*/
/*hash函数 字母大小写不敏感*/
uint64_t dictGenCaseHashFunction(const unsigned char *buf, int len);
void dictEmpty(dict *d, void(callback)(void*));  /*清空一个字典*/
void dictEnableResize(void); /*开启Resize*/
```

```
void dictDisableResize(void); /*关闭Resize*/
int dictRehash(dict *d, int n);  /*渐进式rehash, n为进行几步*/
int dictRehashMilliseconds(dict *d, int ms);   /*持续性rehash， ms为持续多久*/
void dictSetHashFunctionSeed(uint8_t *seed);  /*设置新的散列种子*/
uint8_t *dictGetHashFunctionSeed(void);  /*获取当前散列种子值*/
unsigned long dictScan(dict *d, unsigned long v, dictScanFunction *fn,
dictScanBucketFunction *bucketfn, void *privdata);  /*间断性的迭代字段数据*/
uint64_t dictGetHash(dict *d, const void *key);  /*得到键的hash值*/
 /*使用指针+hash值去查找元素*/
dictEntry **dictFindEntryRefByPtrAndHash(dict *d, const void *oldptr, uint64_t hash);
```

5 总结

字典在Redis数据库中起到了很重要的作用，本文主要介绍了哈希表和Redis字典的设计与实现。其中Redis字典对普通哈希表进行优化和改进，以减少Rehash操作对服务造成的阻塞。后面会继续介绍Hash的一些命令讲解。

收录于话题 #面试 14

上一篇
八大步骤带你深度剖析Kafka生产级容量评估方案

下一篇
kafka三高架构设计剖析

喜欢此内容的人还喜欢

聊聊sql优化的15个小技巧
华仔聊技术

婴幼儿水育风靡全球，这家机构凭什么靠着“防溺水”出圈了？
旧叔笔谈

孟邦一寺庙起火，疑因太阳能过热引起
今日仰光