Redis中dict.c文件的源码解析

姓名:江英翔 学号:1120212709 班级:2107

1.前言

Redis是一个使用ANSI C编写的开源、支持网络、基于内存、分布式、可选持久性的键值对存储数据库。Redis的数据结构采用多种多样的数据结构，其中最常用的就是哈希表，Redis中的哈希表是通过字典来实现的。 本文将对Redis中的dict.c进行源码解析。

2.字典的定义

Redis中的字典是一个哈希表，它的定义如下：

```
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;
```

代码位于dict.h第79行
字典的定义中包含了三个哈希表，一个是ht[0]，一个是ht[1]，还有一个是rehashidx。 ht[0]和ht[1]是两个哈希表，rehashidx是一个索引，用于标识当前正在进行rehash操作的哈希表。 字典中的type和privdata是一个指向dictType结构的指针，用于指定字典的类型，privdata是一个指向任意类型的指针，用于保存字典的私有数据。 iterators是一个整数，用于记录当前正在运行的迭代器的数量。

dict.c源码解析

字典的初始化操作

字典的创建

```
dict *dictCreate(dictType *type)
{
    dict *d = zmalloc(sizeof(*d));
```

```
    _dictInit(d,type);
    return d;
}
```

代码位于dict.c第105行
dictCreate函数用于创建一个字典， 它首先使用zmalloc函数分配一个字典的大小的内存， 然后调用_dictInit函数对字典进行初始化。

初始化字典

```
int _dictInit(dict *d, dictType *type)
{
    _dictReset(d, 0);
    _dictReset(d, 1);
    d->type = type;
    d->rehashidx = -1;
    d->pauserehash = 0;
    return DICT_OK;
}
```

代码位于dict.c第114行
_dictInit函数首先调用_dictReset函数对字典进行初始化， 然后将type赋值给字典的type， 将rehashidx赋值为-1， 表示当前没有进行rehash操作， 最后将pauserehash赋值为0，表示当前没有暂停rehash操作。

重置字典

```
static void _dictReset(dict *d, int htidx)
{
    d->ht_table[htidx] = NULL;
    d->ht_size_exp[htidx] = -1;
    d->ht_used[htidx] = 0;
}
```

代码位于dict.c第97行
_dictReset函数用于对字典进行初始化， 它将字典的ht_table赋值为空，代表当前哈希表为空， 将字典的ht_size_exp赋值为-1，代表当前哈希表的大小为0， 将字典的ht_used赋值为0，代表当前哈希表的已使用节点数为0。

字典的大小

```
static signed char _dictNextExp(unsigned long size)
{
    unsigned char e = DICT_HT_INITIAL_EXP;

    if (size >= LONG_MAX) return (8*sizeof(long)-1);
```

```
    while(1) {
        if (((unsigned long)1<<e) >= size)
            return e;
        e++;
    }
}
```

代码位于dict.c文件第1030行， 该函数用于计算字典的大小， size是字典的大小， 该函数返回值是字典的大小。 首先将e赋值为DICT_HT_INITIAL_EXP， 然后判断size是否大于等于LONG_MAX， 如果大于等于， 则返回8*sizeof(long)-1， 否则进入循环， 然后判断(unsigned long)1<<e是否大于等于size， 如果大于等于， 则返回e， 否则将e加1， 然后进入下一次循环。

字典的增删改查

字典的增加

```
int dictAdd(dict *d, void *key, void *val)
{
    dictEntry *entry = dictAddRaw(d,key);

    if (!entry) return DICT_ERR;
    dictSetVal(d, entry, val);
    return DICT_OK;
}
```

代码位于dict.c第295行
dictAdd函数用于向字典中添加一个键值对， 它首先调用dictAddRaw函数向字典中添加一个键值对， 如果添加失败，那么返回的是DICT_ERR， 如果添加成功，那么调用dictSetVal函数将值赋值给新添加的节点， 最后返回DICT_OK。

```
dictEntry *dictAddRaw(dict *d, void *key, dictEntry **existing)
{
    long index;
    dictEntry *entry;
    int htidx;

    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;
    htidx = dictIsRehashing(d) ? 1 : 0;
    size_t metasize = dictMetadataSize(d);
    entry = zmalloc(sizeof(*entry) + metasize);
    if (metasize > 0) {
        memset(dictMetadata(entry), 0, metasize);
```

```
    }
    entry->next = d->ht_table[htidx][index];
    d->ht_table[htidx][index] = entry;
    d->ht_used[htidx]++;

    /* Set the hash entry fields. */
    dictSetKey(d, entry, key);
    return entry;
}
```

代码位于dict.c第322行
dictAddRaw函数用于向字典中添加一个键值对， 它首先判断字典是否正在进行rehash操作， 如果正在进行rehash操作，那么调用_dictRehashStep函数进行一步rehash操作， 然后调用_dictKeyIndex函数获取键值对的索引， 如果索引为-1，那么表示键值对已经存在， 如果索引不为-1，那么表示键值对不存在， 那么调用zmalloc函数分配一个节点的内存， 然后将节点插入到哈希表中， 最后返回新添加的节点。

字典的删除

```
int dictDelete(dict *ht, const void *key) {
    return dictGenericDelete(ht,key,0) ? DICT_OK : DICT_ERR;
}
```

代码位于dict.c第355行
dictDelete函数用于从字典中删除一个键值对， 它首先调用dictGenericDelete函数从字典中删除一个键值对， 如果删除成功，那么返回DICT_OK， 如果删除失败，那么返回DICT_ERR。

```
static dictEntry *dictGenericDelete(dict *d, const void *key, int nofree) {
    uint64_t h, idx;
    dictEntry *he, *prevHe;
    int table;

    if (dictSize(d) == 0) return NULL;

    if (dictIsRehashing(d)) _dictRehashStep(d);
    h = dictHashKey(d, key);

    for (table = 0; table <= 1; table++) {
        idx = h & DICTHT_SIZE_MASK(d->ht_size_exp[table]);
        he = d->ht_table[table][idx];
        prevHe = NULL;
        while(he) {
            if (key==he->key || dictCompareKeys(d, key, he->key)) {
                if (prevHe)
                    prevHe->next = he->next;
                else
                    d->ht_table[table][idx] = he->next;
                if (!nofree) {
                    dictFreeUnlinkedEntry(d, he);
```

```
                }
                d->ht_used[table]--;
                return he;
            }
            prevHe = he;
            he = he->next;
        }
        if (!dictIsRehashing(d)) break;
    }
    return NULL; /* not found */
}
```

代码位于dict.c第398行
dictGenericDelete函数用于从字典中删除一个键值对， 他首先判断字典是否为空， 如果字典为空，那么返回NULL， 如果字典不为空，那么判断字典是否正在进行rehash操作， 如果正在进行rehash操作，那么调用_dictRehashStep函数进行一步rehash操作， 然后调用dictHashKey函数计算键的哈希值， 然后遍历字典中的两个哈希表， 如果键值对存在，那么调用dictFreeUnlinkedEntry函数释放节点， 最后返回被删除的节点。

字典的修改

```
int dictReplace(dict *d, void *key, void *val)
{
    dictEntry *entry, *existing, auxentry;

    entry = dictAddRaw(d,key,&existing);
    if (entry) {
        dictSetVal(d, entry, val);
        return 1;
    }
    auxentry = *existing;
    dictSetVal(d, existing, val);
    dictFreeVal(d, &auxentry);
    return 0;
}
```

代码位于dict.c第359行
dictReplace函数用于向字典中添加一个键值对， 如果键值对已经存在，那么修改键值对的值， 如果键值对不存在，那么添加键值对。

字典的查找

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

    if (dictSize(d) == 0) return NULL; /* dict is empty */
```

```
    if (dictIsRehashing(d)) _dictRehashStep(d);
    h = dictHashKey(d, key);
    for (table = 0; table <= 1; table++) {
        idx = h & DICTHT_SIZE_MASK(d->ht_size_exp[table]);
        he = d->ht_table[table][idx];
        while(he) {
            if (key==he->key || dictCompareKeys(d, key, he->key))
                return he;
            he = he->next;
        }
        if (!dictIsRehashing(d)) return NULL;
    }
    return NULL;
}
```

代码位于dict.c第509行
dictFind函数用于从字典中查找一个键值对， 如果找到，那么返回该键值对， 如果没有找到，那么返回NULL。

```
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 & DICTHT_SIZE_MASK(d->ht_size_exp[table]);
        /* Search if this slot does not already contain the given key */
        he = d->ht_table[table][idx];
        while(he) {
            if (key==he->key || dictCompareKeys(d, key, he->key)) {
                if (existing) *existing = he;
                return -1;
            }
            he = he->next;
        }
        if (!dictIsRehashing(d)) break;
    }
    return idx;
}
```

代码位于dict.c文件第1049行， 该函数用于获取键的索引， 首先判断existing是否为空， 如果不为空， 则将*existing赋值为NULL， 然后调用_dictExpandIfNeeded函数， 如果返回值为DICT_ERR， 则返回-1， 然后将table赋值为0， 然后判断table是否小于等于1， 如果小于等于1， 则将idx赋值为hash & DICTHT_SIZE_MASK(d->ht_size_exp[table])， 然后将he赋值为d->ht_table[table][idx]， 然后判断

he是否为空， 如果不为空， 则判断key是否等于he->key或者调用dictCompareKeys函数返回值为真， 如果是， 则判断existing是否为空， 如果不为空， 则将*existing赋值为he， 然后返回-1， 否则将he赋值为he->next， 然后判断!dictIsRehashing(d)是否为真， 如果为真， 则跳出循环， 否则将table加1， 最后返回idx。

```
void *dictFetchValue(dict *d, const void *key)
{
    dictEntry *he;

    he = dictFind(d,key);
    return he ? dictGetVal(he) : NULL;
}
```

代码位于dict.c第525行
dictFetchValue函数用于从字典中查找一个键值对， 如果找到，那么返回该键值对的值， 如果没有找到，那么返回NULL。

字典的扩容

```
static int dictTypeExpandAllowed(dict *d) {
    if (d->type->expandAllowed == NULL) return 1;
    return d->type->expandAllowed(
                    DICTHT_SIZE(_dictNextExp(d->ht_used[0] + 1)) *
sizeof(dictEntry*),
                    (double)d->ht_used[0] / DICTHT_SIZE(d->ht_size_exp[0]));
}
```

代码位于dict.c第998行
dictTypeExpandAllowed函数用于判断字典是否允许扩容， 如果字典的类型没有实现expandAllowed函数，那么返回1， 如果字典的类型实现了expandAllowed函数，那么调用该函数判断是否允许扩容。

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

    if (DICTHT_SIZE(d->ht_size_exp[0]) == 0) return dictExpand(d,
DICT_HT_INITIAL_SIZE);

    if (d->ht_used[0] >= DICTHT_SIZE(d->ht_size_exp[0]) &&
        (dict_can_resize ||
         d->ht_used[0]/ DICTHT_SIZE(d->ht_size_exp[0]) > dict_force_resize_ratio)
&&
        dictTypeExpandAllowed(d))
    {
        return dictExpand(d, d->ht_used[0] + 1);
    }
```

```
    return DICT_OK;
}
```

代码位于dict.c第1006行

_dictExpandIfNeeded函数用于判断字典是否需要扩容， 它首先判断字典是否正在进行rehash操作， 如果正在进行rehash操作，那么返回DICT_OK， 如果没有进行rehash操作，那么判断字典的哈希表是否为空， 如果哈希表为空，那么调用dictExpand函数扩容字典， 如果哈希表不为空，那么判断字典的使用节点数是否大于等于哈希表的大小， 如果大于等于，那么判断是否允许扩容， 如果允许扩容，那么调用dictExpand函数扩容字典， 如果不允许扩容，那么返回DICT_OK。

```
int dictExpand(dict *d, unsigned long size) {
    return _dictExpand(d, size, NULL);
}
int _dictExpand(dict *d, unsigned long size, int* malloc_failed)
{
    if (malloc_failed) *malloc_failed = 0;

    if (dictIsRehashing(d) || d->ht_used[0] > size)
        return DICT_ERR;

    dictEntry **new_ht_table;
    unsigned long new_ht_used;
    signed char new_ht_size_exp = _dictNextExp(size);

    size_t newsize = 1ul<<new_ht_size_exp;
    if (newsize < size || newsize * sizeof(dictEntry*) < newsize)
        return DICT_ERR;

    if (new_ht_size_exp == d->ht_size_exp[0]) return DICT_ERR;

    if (malloc_failed) {
        new_ht_table = ztrycalloc(newsize*sizeof(dictEntry*));
        *malloc_failed = new_ht_table == NULL;
        if (*malloc_failed)
            return DICT_ERR;
    } else
        new_ht_table = zcalloc(newsize*sizeof(dictEntry*));

    new_ht_used = 0;

    if (d->ht_table[0] == NULL) {
        d->ht_size_exp[0] = new_ht_size_exp;
        d->ht_used[0] = new_ht_used;
        d->ht_table[0] = new_ht_table;
        return DICT_OK;
    }

    /* Prepare a second hash table for incremental rehashing */
    d->ht_size_exp[1] = new_ht_size_exp;
    d->ht_used[1] = new_ht_used;
    d->ht_table[1] = new_ht_table;
```

```
    d->rehashidx = 0;
    return DICT_OK;
}
```

代码位于dict.c第140行与191行
dictExpand与_dictExpand函数用于扩容字典， 它们首先判断字典是否正在进行rehash操作， 如果正在进行rehash操作，那么返回DICT_ERR， 如果没有进行rehash操作，那么判断字典的使用节点数是否大于等于size，如果大于等于，那么返回DICT_ERR， 如果小于，那么判断字典的哈希表是否为空， 如果哈希表为空，那么调用_dictExpand函数扩容字典， 如果哈希表不为空，那么判断字典的哈希表的大小是否等于size， 如果等于，那么返回DICT_ERR， 如果不等于，那么调用_dictExpand函数扩容字典。

字典的迭代器

初始化字典迭代器

```
void dictInitIterator(dictIterator *iter, dict *d)
{
    iter->d = d;
    iter->table = 0;
    iter->index = -1;
    iter->safe = 0;
    iter->entry = NULL;
    iter->nextEntry = NULL;
}
```

代码位于dict.c第575行
dictInitIterator函数用于初始化字典的迭代器， 它首先将字典的地址赋值给迭代器的字典指针，代表迭代器正在迭代的字典， 然后将迭代器的哈希表索引赋值为-1，代表迭代器正在迭代的哈希表索引， 之后将迭代器的安全标志赋值为0，代表迭代器不是安全的， 最后将迭代器的当前节点指针与下一个节点指针赋值为NULL，代表迭代器当前没有节点。

安全地初始化字典迭代器

```
void dictSafeIterator(dictIterator *iter, dict *d)
{
    dictInitIterator(iter, d);
    iter->safe = 1;
}
```

代码位于dict.c第585行
dictSafeIterator函数用于安全地初始化字典的迭代器， 它首先调用dictInitIterator函数初始化字典的迭代器， 然后将迭代器的安全标志赋值为1，代表迭代器是安全的。

释放字典迭代器

```
void  (dictIterator *iter)
{
    dictResetIterator(iter);
    zfree(iter);
}
```

代码位于dict.c第648行
dictReleaseIterator函数用于释放字典的迭代器， 它首先调用dictResetIterator函数重置字典的迭代器， 然后释放迭代器的内存。

重置字典迭代器

```
void dictResetIterator(dictIterator *iter)
{
    if (!(iter->index == -1 && iter->table == 0)) {
        if (iter->safe)
            dictResumeRehashing(iter->d);
        else
            assert(iter->fingerprint == dictFingerprint(iter->d));
    }
}
```

代码位于dict.c第596行
dictResetIterator函数用于重置字典的迭代器， 它首先判断迭代器的哈希表索引是否等于-1，且迭代器的哈希表索引是否等于0， 如果不等于，那么判断迭代器是否是安全的， 如果是安全的，那么调用dictResumeRehashing函数恢复字典的rehash操作， 如果不是安全的，那么判断迭代器的指纹是否等于字典的指纹， 如果不等于，那么程序出错。

返回字典迭代器的当前节点

```
dictEntry *dictNext(dictIterator *iter)
{
    while (1) {
        if (iter->entry == NULL) {
            if (iter->index == -1 && iter->table == 0) {
                if (iter->safe)
                    dictPauseRehashing(iter->d);
                else
                    iter->fingerprint = dictFingerprint(iter->d);
            }
            iter->index++;
            if (iter->index >= (long) DICTHT_SIZE(iter->d->ht_size_exp[iter-
>table])) {
                if (dictIsRehashing(iter->d) && iter->table == 0) {
                    iter->table++;
                    iter->index = 0;
                } else {
```

```
                    break;
                }
            }
            iter->entry = iter->d->ht_table[iter->table][iter->index];
        } else {
            iter->entry = iter->nextEntry;
        }
        if (iter->entry) {
            /* We need to save the 'next' here, the iterator user
             * may delete the entry we are returning. */
            iter->nextEntry = iter->entry->next;
            return iter->entry;
        }
    }
    return NULL;
}
```

代码位于dict.c第615行
dictNext函数用于返回字典迭代器的当前节点， 它首先进入一个死循环， 然后判断迭代器的当前节点是否为空， 如果为空，那么判断迭代器的哈希表索引是否等于-1，且迭代器的哈希表索引是否等于0， 如果等于，那么判断迭代器是否是安全的， 如果是安全的，那么调用dictPauseRehashing函数暂停字典的rehash操作， 如果不是安全的，那么将迭代器的指纹赋值为字典的指纹， 然后将迭代器的哈希表索引加1， 如果迭代器的哈希表索引大于等于字典的哈希表大小， 那么判断字典是否正在rehash，且迭代器的哈希表索引是否等于0， 如果是，那么将迭代器的哈希表索引赋值为1，代表正在迭代第二个哈希表， 然后将迭代器的哈希表索引赋值为0，如果不是，那么跳出循环， 然后将迭代器的当前节点赋值为字典的哈希表的当前节点， 如果迭代器的当前节点不为空， 那么将迭代器的下一个节点赋值为当前节点的下一个节点， 然后返回当前节点。

获取安全的字典迭代器

```
dictIterator *dictGetSafeIterator(dict *d) {
    dictIterator *i = dictGetIterator(d);

    i->safe = 1;
    return i;
}
```

代码位于dict.c第608行
dictGetSafeIterator函数用于获取安全的字典迭代器， 它首先调用dictGetIterator函数获取一个字典迭代器， 然后将迭代器的安全标识赋值为1， 最后返回迭代器。

释放字典迭代器

```
void dictReleaseIterator(dictIterator *iter)
{
    dictResetIterator(iter);
    zfree(iter);
}
```

代码位于dict.c第648行
dictReleaseIterator函数用于释放字典迭代器， 它首先调用dictResetIterator函数重置字典迭代器， 然后释放迭代器。

获得迭代器

```
dictIterator *dictGetIterator(dict *d)
{
    dictIterator *iter = zmalloc(sizeof(*iter));
    dictInitIterator(iter, d);
    return iter;
}
```

代码位于dict.c第601行
dictGetIterator函数用于获取字典迭代器， 它首先为字典迭代器分配内存， 然后调用dictInitIterator函数初始化字典迭代器， 最后返回字典迭代器。

哈希相关

哈希函数

```
uint64_t dictGenHashFunction(const void *key, size_t len) {
    return siphash(key,len,dict_hash_function_seed);
}
```

代码位于dict.c第86行
dictGenHashFunction函数用于生成哈希值， 它首先调用siphash函数生成哈希值， 然后返回哈希值。

rehash函数

```
int dictRehash(dict *d, int n) {
    int empty_visits = n*10;
    if (!dictIsRehashing(d)) return 0;

    while(n-- && d->ht_used[0] != 0) {
        dictEntry *de, *nextde;

        assert(DICTHT_SIZE(d->ht_size_exp[0]) > (unsigned long)d->rehashidx);
        while(d->ht_table[0][d->rehashidx] == NULL) {
            d->rehashidx++;
            if (--empty_visits == 0) return 1;
        }
        de = d->ht_table[0][d->rehashidx];
        while(de) {
```

```
            uint64_t h;

            nextde = de->next;
            h = dictHashKey(d, de->key) & DICTHT_SIZE_MASK(d->ht_size_exp[1]);
            de->next = d->ht_table[1][h];
            d->ht_table[1][h] = de;
            d->ht_used[0]--;
            d->ht_used[1]++;
            de = nextde;
        }
        d->ht_table[0][d->rehashidx] = NULL;
        d->rehashidx++;
    }

    /* Check if we already rehashed the whole table... */
    if (d->ht_used[0] == 0) {
        zfree(d->ht_table[0]);
        /* Copy the new ht onto the old one */
        d->ht_table[0] = d->ht_table[1];
        d->ht_used[0] = d->ht_used[1];
        d->ht_size_exp[0] = d->ht_size_exp[1];
        _dictReset(d, 1);
        d->rehashidx = -1;
        return 0;
    }

    /* More to rehash... */
    return 1;
}
```

代码位于dict.c第211行
dictRehash函数用于rehash操作， 它首先判断字典是否正在rehash， 如果不是，那么返回0， 如果是，那么首先初始化空的哈希表索引的访问次数， 然后判断字典的哈希表索引是否小于哈希表的大小， 如果不小于，那么判断空的哈希表索引的访问次数是否等于0， 如果等于，那么返回1， 如果不等于，那么判断哈希表的索引是否为空， 如果为空，那么将哈希表的索引加1， 如果不为空，那么将哈希表的索引的元素赋值给de 然后判断de是否为空， 如果为空，那么将哈希表的索引加1， 如果不为空，那么将哈希表的索引的元素赋值给nextde，然后调用dictHashKey函数生成哈希值， 然后将哈希值与哈希表的大小取与， 然后将哈希表的索引的元素的下一个元素赋值给哈希表的索引的元素， 然后将哈希表的索引的元素赋值给哈希表的索引的元素的下一个元素， 然后将哈希表的索引的元素赋值给de， 然后将哈希表的索引的元素赋值为NULL， 然后将哈希表的索引加1， 然后判断哈希表的使用量是否等于0， 如果等于，那么释放哈希表的索引， 然后将哈希表的索引赋值给哈希表的索引的下一个元素， 然后将哈希表的使用量赋值给哈希表的使用量的下一个元素， 然后将哈希表的大小赋值给哈希表的大小的下一个元素， 然后调用_dictReset函数重置字典， 然后将哈希表的索引赋值为-1， 如果不等于，那么返回1。

哈希函数种子设置与获取

```
static uint8_t dict_hash_function_seed[16];

void dictSetHashFunctionSeed(uint8_t *seed) {
```

```
    memcpy(dict_hash_function_seed,seed,sizeof(dict_hash_function_seed));
}

uint8_t *dictGetHashFunctionSeed(void) {
    return dict_hash_function_seed;
}
```

代码位于dict.c第70行
dictSetHashFunctionSeed函数用于设置哈希函数种子， 它首先调用memcpy函数将哈希函数种子的值赋值给dict_hash_function_seed， 然后返回dict_hash_function_seed， dictGetHashFunctionSeed函数用于获取哈希函数种子， 它直接返回dict_hash_function_seed。

哈希函数

```
uint64_t siphash(const uint8_t *in, const size_t inlen, const uint8_t *k);
uint64_t siphash_nocase(const uint8_t *in, const size_t inlen, const uint8_t *k);

uint64_t dictGenCaseHashFunction(const unsigned char *buf, size_t len) {
    return siphash_nocase(buf,len,dict_hash_function_seed);
}
```

代码位于dict.c第83行
dictGenCaseHashFunction函数用于生成哈希值， 它首先调用siphash_nocase函数生成哈希值， 然后返回哈希值。

字典的其他操作

字典的重构

```
int dictResize(dict *d)
{
    unsigned long minimal;

    if (!dict_can_resize || dictIsRehashing(d)) return DICT_ERR;
    minimal = d->ht_used[0];
    if (minimal < DICT_HT_INITIAL_SIZE)
        minimal = DICT_HT_INITIAL_SIZE;
    return dictExpand(d, minimal);
}
```

代码位于dict.c第126行
dictResize函数用于字典的重构， 它首先判断字典是否可以重构或者字典是否正在重构， 如果可以，那么将哈希表的使用量赋值给minimal， 如果哈希表的使用量小于哈希表的初始大小， 那么将哈希表的初始大小赋值给minimal， 然后调用dictExpand函数扩展字典， 否则返回DICT_ERR。

字典试图扩展

```
int dictTryExpand(dict *d, unsigned long size) {
    int malloc_failed;
    _dictExpand(d, size, &malloc_failed);
    return malloc_failed? DICT_ERR : DICT_OK;
}
```

代码位于dict.c第196行
dictTryExpand函数用于字典试图扩展，也就是在内存分配失败时不会报错， 它首先调用_dictExpand函数扩展字典， 然后判断内存分配是否失败， 如果失败，那么返回DICT_ERR， 否则返回DICT_OK。

rehash移动指定毫秒时间

```
int dictRehashMilliseconds(dict *d, int ms) {
    if (d->pauserehash > 0) return 0;

    long long start = timeInMilliseconds();
    int rehashes = 0;

    while(dictRehash(d,100)) {
        rehashes += 100;
        if (timeInMilliseconds()-start > ms) break;
    }
    return rehashes;
}
```

代码位于dict.c第269行
dictRehashMilliseconds函数用于rehash移动指定毫秒时间， 它首先获取当前时间， 然后初始化rehashes为0， 接着循环调用dictRehash函数， 每次循环rehashes加100， 如果当前时间减去开始时间大于指定毫秒时间， 那么跳出循环， 最后返回rehashes。

rehash移动指定数量

```
static void _dictRehashStep(dict *d) {
    if (d->pauserehash == 0) dictRehash(d,1);
}
```

代码位于dict.c第290行
_dictRehashStep函数用于rehash移动指定数量， 它首先判断字典是否暂停rehash， 如果没有，那么调用dictRehash函数， 否则不做任何操作。

字典的释放及内存回收

清除字典

```
int _dictClear(dict *d, int htidx, void(callback)(dict*)) {
    unsigned long i;

    /* Free all the elements */
    for (i = 0; i < DICTHT_SIZE(d->ht_size_exp[htidx]) && d->ht_used[htidx] > 0;
i++) {
        dictEntry *he, *nextHe;

        if (callback && (i & 65535) == 0) callback(d);

        if ((he = d->ht_table[htidx][i]) == NULL) continue;
        while(he) {
            nextHe = he->next;
            dictFreeKey(d, he);
            dictFreeVal(d, he);
            zfree(he);
            d->ht_used[htidx]--;
            he = nextHe;
        }
    }
    /* Free the table and the allocated cache structure */
    zfree(d->ht_table[htidx]);
    /* Re-initialize the table */
    _dictReset(d, htidx);
    return DICT_OK; /* never fails */
}
```

代码位于dict.c第475行
_dictClear函数用于清除字典， 它首先初始化i为0， 然后循环遍历哈希表， 如果哈希表的使用量大于0， 那么调用callback函数， 然后判断哈希表的第i个桶是否为空， 如果不为空，那么循环遍历哈希表的第i个桶，首先获取下一个哈希表节点， 然后释放哈希表节点的键和值， 最后释放哈希表节点， 最后将哈希表的使用量减1， 最后释放哈希表， 然后重置哈希表， 最后返回DICT_OK。

释放字典

```
void dictRelease(dict *d)
{
    _dictClear(d,0,NULL);
    _dictClear(d,1,NULL);
    zfree(d);
}
```

代码位于dict.c第502行
dictRelease函数用于释放字典， 它首先调用_dictClear函数清除哈希表0， 然后调用_dictClear函数清除哈希表1， 最后释放字典。

字典的指纹

字典的指纹定义

> - A fingerprint is a 64 bit number that represents the state of the dictionary at a given time, it's just a few dict properties xored together. When an unsafe iterator is initialized, we get the dict fingerprint, and check the fingerprint again when the iterator is released.If the two fingerprints are different it means that the user of the iterator performed forbidden operations against the dictionary while iterating.

字典的指纹计算

```
unsigned long long dictFingerprint(dict *d) {
    unsigned long long integers[6], hash = 0;
    int j;

    integers[0] = (long) d->ht_table[0];
    integers[1] = d->ht_size_exp[0];
    integers[2] = d->ht_used[0];
    integers[3] = (long) d->ht_table[1];
    integers[4] = d->ht_size_exp[1];
    integers[5] = d->ht_used[1];

    /* We hash N integers by summing every successive integer with the integer
     * hashing of the previous sum. Basically:
     *
     * Result = hash(hash(hash(int1)+int2)+int3) ...
     *
     * This way the same set of integers in a different order will (likely) hash
     * to a different number. */
    for (j = 0; j < 6; j++) {
        hash += integers[j];
        /* For the hashing step we use Tomas Wang's 64 bit integer hash. */
        hash = (~hash) + (hash << 21); // hash = (hash << 21) - hash - 1;
        hash = hash ^ (hash >> 24);
        hash = (hash + (hash << 3)) + (hash << 8); // hash * 265
        hash = hash ^ (hash >> 14);
        hash = (hash + (hash << 2)) + (hash << 4); // hash * 21
        hash = hash ^ (hash >> 28);
        hash = hash + (hash << 31);
    }
    return hash;
}
```

代码位于dict.c第543行
dictFingerprint函数用于计算字典的指纹， 它首先初始化integers数组， 然后将哈希表0的表指针、哈希表0的大小指数、哈希表0的使用量、哈希表1的表指针、哈希表1的大小指数、哈希表1的使用量分别赋值给integers数组， 然后初始化hash为0， 然后循环遍历integers数组， 首先将hash加上integers数组的第j个元素， 然后将hash进行哈希运算， 最后返回hash。

字典的随机条目

```
dictEntry *dictGetRandomKey(dict *d)
{
    dictEntry *he, *orighe;
    unsigned long h;
    int listlen, listele;

    if (dictSize(d) == 0) return NULL;
    if (dictIsRehashing(d)) _dictRehashStep(d);
    if (dictIsRehashing(d)) {
        unsigned long s0 = DICTHT_SIZE(d->ht_size_exp[0]);
        do {
            /* We are sure there are no elements in indexes from 0
             * to rehashidx-1 */
            h = d->rehashidx + (randomULong() % (dictSlots(d) - d->rehashidx));
            he = (h >= s0) ? d->ht_table[1][h - s0] : d->ht_table[0][h];
        } while(he == NULL);
    } else {
        unsigned long m = DICTHT_SIZE_MASK(d->ht_size_exp[0]);
        do {
            h = randomULong() & m;
            he = d->ht_table[0][h];
        } while(he == NULL);
    }

    /* Now we found a non empty bucket, but it is a linked
     * list and we need to get a random element from the list.
     * The only sane way to do so is counting the elements and
     * select a random index. */
    listlen = 0;
    orighe = he;
    while(he) {
        he = he->next;
        listlen++;
    }
    listele = random() % listlen;
    he = orighe;
    while(listele--) he = he->next;
    return he;
}
```

代码位于dict.c第656行
dictGetRandomKey函数用于获取字典的随机条目， 它首先判断字典的大小是否为0， 如果为0，则返回NULL， 否则判断字典是否正在进行rehash， 如果正在进行rehash，则调用_dictRehashStep函数进行rehash， 然后判断字典是否正在进行rehash， 如果正在进行rehash， 则首先初始化s0为哈希表0的大小， 然后循环遍历哈希表0和哈希表1的所有槽， 首先初始化h为rehash索引加上随机数对dictSlots减去rehash索引的余数， 然后判断h是否大于等于s0， 如果大于等于s0， 则将哈希表1的第h减去s0个槽赋值给he， 否则将哈希表0的第h个槽赋值给he， 然后判断he是否为NULL， 如果为NULL， 则继续循环遍历哈希表0和哈希表1的所有槽， 否则跳出循环， 如果字典不是正在进行rehash， 则首先初始化m为哈希表0的大小掩码， 然后循环遍历哈

希表0的所有槽， 首先初始化h为随机数与m的与运算结果， 然后将哈希表0的第h个槽赋值给he， 然后判断he是否为NULL， 如果为NULL， 则继续循环遍历哈希表0的所有槽， 否则跳出循环， 然后初始化listlen为0，初始化orighe为he， 然后循环遍历he， 首先将he的下一个元素赋值给he， 然后将listlen加1， 然后初始化listele为随机数对listlen的余数， 然后将orighe赋值给he， 然后循环遍历listele， 首先将he的下一个元素赋值给he， 然后返回he。

释放未连接的条目

```
void dictFreeUnlinkedEntry(dict *d, dictEntry *he) {
    if (he == NULL) return;
    dictFreeKey(d, he);
    dictFreeVal(d, he);
    zfree(he);
}
```

代码位于dict.c第467行
dictFreeUnlinkedEntry函数用于释放未连接的条目，

随机返回几个字典的键

```
unsigned int dictGetSomeKeys(dict *d, dictEntry **des, unsigned int count) {
    unsigned long j; /* internal hash table id, 0 or 1. */
    unsigned long tables; /* 1 or 2 tables? */
    unsigned long stored = 0, maxsizemask;
    unsigned long maxsteps;

    if (dictSize(d) < count) count = dictSize(d);
    maxsteps = count*10;

    /* Try to do a rehashing work proportional to 'count'. */
    for (j = 0; j < count; j++) {
        if (dictIsRehashing(d))
            _dictRehashStep(d);
        else
            break;
    }

    tables = dictIsRehashing(d) ? 2 : 1;
    maxsizemask = DICTHT_SIZE_MASK(d->ht_size_exp[0]);
    if (tables > 1 && maxsizemask < DICTHT_SIZE_MASK(d->ht_size_exp[1]))
        maxsizemask = DICTHT_SIZE_MASK(d->ht_size_exp[1]);

    /* Pick a random point inside the larger table. */
    unsigned long i = randomULong() & maxsizemask;
    unsigned long emptylen = 0; /* Continuous empty entries so far. */
    while(stored < count && maxsteps--) {
        for (j = 0; j < tables; j++) {
            /* Invariant of the dict.c rehashing: up to the indexes already
```

```
             * visited in ht[0] during the rehashing, there are no populated
             * buckets, so we can skip ht[0] for indexes between 0 and idx-1. */
            if (tables == 2 && j == 0 && i < (unsigned long) d->rehashidx) {
                /* Moreover, if we are currently out of range in the second
                 * table, there will be no elements in both tables up to
                 * the current rehashing index, so we jump if possible.
                 * (this happens when going from big to small table). */
                if (i >= DICTHT_SIZE(d->ht_size_exp[1]))
                    i = d->rehashidx;
                else
                    continue;
            }
            if (i >= DICTHT_SIZE(d->ht_size_exp[j])) continue; /* Out of range for
this table. */
            dictEntry *he = d->ht_table[j][i];

            /* Count contiguous empty buckets, and jump to other
             * locations if they reach 'count' (with a minimum of 5). */
            if (he == NULL) {
                emptylen++;
                if (emptylen >= 5 && emptylen > count) {
                    i = randomULong() & maxsizemask;
                    emptylen = 0;
                }
            } else {
                emptylen = 0;
                while (he) {
                    /* Collect all the elements of the buckets found non
                     * empty while iterating. */
                    *des = he;
                    des++;
                    he = he->next;
                    stored++;
                    if (stored == count) return stored;
                }
            }
        }
        i = (i+1) & maxsizemask;
    }
    return stored;
}
```

代码位于dict.c文件第718行， 该函数用于随机返回几个字典的键， 该函数的参数d为字典， 参数des为字典的键数组， 参数count为随机返回的键的个数， 首先判断字典的大小是否小于count， 如果小于count， 则将字典的大小赋值给count， 然后初始化maxsteps为count*10， 然后循环遍历count， 首先判断字典是否正在进行rehash， 如果正在进行rehash， 则调用_dictRehashStep函数， 否则跳出循环， 然后初始化tables为1， 如果字典正在进行rehash， 则将tables赋值为2， 然后初始化maxsizemask为哈希表0的大小掩码， 然后判断tables是否大于1， 如果大于1， 并且maxsizemask小于哈希表1的大小掩码， 则将maxsizemask赋值为哈希表1的大小掩码， 然后随机生成一个数， 并将该数与maxsizemask进行与运算， 然后将结果赋值给i， 然后初始化emptylen为0， 然后循环遍历count和maxsteps， 首先循环遍历tables， 然后判断tables是否等于2， 如果等于2， 并且j等于0， 并且i小于rehashidx， 则判断i是否大于等于哈希表1的大小， 如果大于等于

哈希表1的大小， 则将rehashidx赋值给i， 否则跳过该次循环， 然后判断i是否大于等于哈希表的大小， 如果大于等于哈希表的大小， 则跳过该次循环， 然后将哈希表j的第i个元素赋值给he， 然后判断he是否为空， 如果为空， 则将emptylen加1， 然后判断emptylen是否大于等于5， 并且emptylen大于count， 如果满足条件， 则随机生成一个数， 并将该数与maxsizemask进行与运算， 然后将结果赋值给i， 然后将emptylen赋值为0， 否则将emptylen赋值为0， 然后判断he是否为空， 如果不为空， 则循环遍历he， 首先将he的值赋值给des， 然后将des指向下一个元素， 然后将he指向下一个元素， 然后将stored加1， 然后判断stored是否等于count， 如果等于count， 则返回stored， 否则跳过该次循环， 然后将i加1与maxsizemask进行与运算， 返回stored。

字典的迭代

```
unsigned long dictScan(dict *d,
                       unsigned long v,
                       dictScanFunction *fn,
                       dictScanBucketFunction* bucketfn,
                       void *privdata)
{
    int htidx0, htidx1;
    const dictEntry *de, *next;
    unsigned long m0, m1;

    if (dictSize(d) == 0) return 0;

    /* This is needed in case the scan callback tries to do dictFind or alike. */
    dictPauseRehashing(d);

    if (!dictIsRehashing(d)) {
        htidx0 = 0;
        m0 = DICTHT_SIZE_MASK(d->ht_size_exp[htidx0]);

        /* Emit entries at cursor */
        if (bucketfn) bucketfn(d, &d->ht_table[htidx0][v & m0]);
        de = d->ht_table[htidx0][v & m0];
        while (de) {
            next = de->next;
            fn(privdata, de);
            de = next;
        }

        /* Set unmasked bits so incrementing the reversed cursor
         * operates on the masked bits */
        v |= ~m0;

        /* Increment the reverse cursor */
        v = rev(v);
        v++;
        v = rev(v);

    } else {
        htidx0 = 0;
```

```
        htidx1 = 1;

        /* Make sure t0 is the smaller and t1 is the bigger table */
        if (DICTHT_SIZE(d->ht_size_exp[htidx0]) > DICTHT_SIZE(d-
>ht_size_exp[htidx1])) {
            htidx0 = 1;
            htidx1 = 0;
        }

        m0 = DICTHT_SIZE_MASK(d->ht_size_exp[htidx0]);
        m1 = DICTHT_SIZE_MASK(d->ht_size_exp[htidx1]);

        /* Emit entries at cursor */
        if (bucketfn) bucketfn(d, &d->ht_table[htidx0][v & m0]);
        de = d->ht_table[htidx0][v & m0];
        while (de) {
            next = de->next;
            fn(privdata, de);
            de = next;
        }

        /* Iterate over indices in larger table that are the expansion
         * of the index pointed to by the cursor in the smaller table */
        do {
            /* Emit entries at cursor */
            if (bucketfn) bucketfn(d, &d->ht_table[htidx1][v & m1]);
            de = d->ht_table[htidx1][v & m1];
            while (de) {
                next = de->next;
                fn(privdata, de);
                de = next;
            }

            /* Increment the reverse cursor not covered by the smaller mask.*/
            v |= ~m1;
            v = rev(v);
            v++;
            v = rev(v);

            /* Continue while bits covered by mask difference is non-zero */
        } while (v & (m0 ^ m1));
    }

    dictResumeRehashing(d);

    return v;
}
```

代码位于dict.c文件第907行， 该函数用于迭代字典， 该函数接收5个参数， d是字典， v是游标， fn是回调函数， bucketfn是桶回调函数， privdata是私有数据， 该函数返回值是v。 首先判断字典的大小是否为0，如果为0， 则返回0， 否则跳过该次循环， 然后调用dictPauseRehashing函数暂停字典的rehash， 然后判断字典是否正在rehash， 如果不是， 则将htidx0赋值为0，代表ht_table[0]， 然后将m0赋值为

DICTHT_SIZE_MASK(d->ht_size_exp[htidx0])， 然后判断bucketfn是否为空， 如果不为空， 则调用bucketfn函数， 然后将de赋值为d->ht_table[htidx0][v & m0]， 然后判断de是否为空， 如果不为空， 则将next赋值为de->next， 然后调用fn函数， 然后将de赋值为next， 然后将v与~m0进行或运算， 然后将v赋值为rev(v)， 然后将v加1， 然后将v赋值为rev(v)， 如果字典正在rehash， 则将htidx0赋值为0，代表ht_table[0]， 然后将htidx1赋值为1，代表ht_table[1]， 然后判断DICTHT_SIZE(d->ht_size_exp[htidx0])是否大于DICTHT_SIZE(d->ht_size_exp[htidx1])， 如果大于， 则将htidx0赋值为1，代表t0是较小的表， 然后将htidx1赋值为0，代表t1是较大的表， 然后将m0赋值为DICTHT_SIZE_MASK(d->ht_size_exp[htidx0])， 然后将m1赋值为DICTHT_SIZE_MASK(d->ht_size_exp[htidx1])， 然后判断bucketfn是否为空， 如果不为空， 则调用bucketfn函数， 然后将de赋值为d->ht_table[htidx0][v & m0]， 然后判断de是否为空， 如果不为空， 则将next赋值为de->next， 然后调用fn函数， 然后将de赋值为next， 然后将v与~m1进行或运算， 然后将v赋值为rev(v)， 然后将v加1， 然后将v赋值为rev(v)， 然后判断v与(m0 ^ m1)进行与运算的结果是否为0， 如果不为0， 则跳过该次循环， 否则跳出循环， 然后调用dictResumeRehashing函数恢复字典的rehash， 最后返回v。

字典使用指针查找dictEntry引用

```
dictEntry **dictFindEntryRefByPtrAndHash(dict *d, const void *oldptr, uint64_t
hash) {
    dictEntry *he, **heref;
    unsigned long idx, table;

    if (dictSize(d) == 0) return NULL; /* dict is empty */
    for (table = 0; table <= 1; table++) {
        idx = hash & DICTHT_SIZE_MASK(d->ht_size_exp[table]);
        heref = &d->ht_table[table][idx];
        he = *heref;
        while(he) {
            if (oldptr==he->key)
                return heref;
            heref = &he->next;
            he = *heref;
        }
        if (!dictIsRehashing(d)) return NULL;
    }
    return NULL;
}
```

代码位于dict.c文件第1098行， 该函数用于查找指针oldptr对应的dictEntry的引用， 首先判断字典的大小是否为0， 如果为0， 则返回NULL， 否则进入循环， 首先将idx赋值为hash与DICTHT_SIZE_MASK(d->ht_size_exp[table])进行与运算的结果， 然后将heref赋值为d->ht_table[table][idx]， 然后将he赋值为*heref， 然后判断he是否为空， 如果不为空， 则判断oldptr是否等于he->key， 如果等于， 则返回heref， 否则将heref赋值为he->next， 然后将he赋值为*heref， 然后判断he是否为空， 如果不为空， 则跳过该次循环， 否则判断字典是否正在rehash， 如果不是， 则返回NULL， 否则将table加1， 然后判断table是否大于1， 如果大于1， 则返回NULL， 否则跳过该次循环， 最后返回NULL。

调试

字典信息获取

```
size_t _dictGetStatsHt(char *buf, size_t bufsize, dict *d, int htidx) {
    unsigned long i, slots = 0, chainlen, maxchainlen = 0;
    unsigned long totchainlen = 0;
    unsigned long clvector[DICT_STATS_VECTLEN];
    size_t l = 0;

    if (d->ht_used[htidx] == 0) {
        return snprintf(buf,bufsize,
            "No stats available for empty dictionaries\n");
    }

    /* Compute stats. */
    for (i = 0; i < DICT_STATS_VECTLEN; i++) clvector[i] = 0;
    for (i = 0; i < DICTHT_SIZE(d->ht_size_exp[htidx]); i++) {
        dictEntry *he;

        if (d->ht_table[htidx][i] == NULL) {
            clvector[0]++;
            continue;
        }
        slots++;
        /* For each hash entry on this slot... */
        chainlen = 0;
        he = d->ht_table[htidx][i];
        while(he) {
            chainlen++;
            he = he->next;
        }
        clvector[(chainlen < DICT_STATS_VECTLEN) ? chainlen : (DICT_STATS_VECTLEN-
1)]++;
        if (chainlen > maxchainlen) maxchainlen = chainlen;
        totchainlen += chainlen;
    }

    /* Generate human readable stats. */
    l += snprintf(buf+l,bufsize-l,
        "Hash table %d stats (%s):\n"
        " table size: %lu\n"
        " number of elements: %lu\n"
        " different slots: %lu\n"
        " max chain length: %lu\n"
        " avg chain length (counted): %.02f\n"
        " avg chain length (computed): %.02f\n"
        " Chain length distribution:\n",
        htidx, (htidx == 0) ? "main hash table" : "rehashing target",
        DICTHT_SIZE(d->ht_size_exp[htidx]), d->ht_used[htidx], slots, maxchainlen,
        (float)totchainlen/slots, (float)d->ht_used[htidx]/slots);

    for (i = 0; i < DICT_STATS_VECTLEN-1; i++) {
        if (clvector[i] == 0) continue;
```

```
        if (l >= bufsize) break;
        l += snprintf(buf+l,bufsize-l,
            "   %ld: %ld (%.02f%%)\n",
            i, clvector[i], ((float)clvector[i]/DICTHT_SIZE(d-
>ht_size_exp[htidx]))*100);
    }

    /* Unlike snprintf(), return the number of characters actually written. */
    if (bufsize) buf[bufsize-1] = '\0';
    return strlen(buf);
}
```

代码位于dict.c文件第1121行， 该函数用于获取字典的统计信息， 输出到buf中， 统计信息含有：

- table size：字典的大小
- number of elements：字典中的元素数量
- different slots：字典中不同的槽的数量
- max chain length：字典中链表的最大长度
- avg chain length (counted)：字典中链表的平均长度（计算得到）
- avg chain length (computed)：字典中链表的平均长度（统计得到）
- Chain length distribution：字典中链表的长度分布
  同时存在

```
void dictGetStats(char *buf, size_t bufsize, dict *d) {
    size_t l;
    char *orig_buf = buf;
    size_t orig_bufsize = bufsize;

    l = _dictGetStatsHt(buf,bufsize,d,0);
    buf += l;
    bufsize -= l;
    if (dictIsRehashing(d) && bufsize > 0) {
        _dictGetStatsHt(buf,bufsize,d,1);
    }
    /* Make sure there is a NULL term at the end. */
    if (orig_bufsize) orig_buf[orig_bufsize-1] = '\0';
}
```

代码位于dict.c文件第1181行， 也是为了获取字典的统计信息， 但是该函数会判断字典是否正在rehash， 如果正在rehash， 则会获取rehash的字典的统计信息。

字典的基准

基准测试

```
int dictTest(int argc, char **argv, int flags) {
    long j;
```

```
    long long start, elapsed;
    dict *dict = dictCreate(&BenchmarkDictType);
    long count = 0;
    int accurate = (flags & REDIS_TEST_ACCURATE);

    if (argc == 4) {
        if (accurate) {
            count = 5000000;
        } else {
            count = strtol(argv[3],NULL,10);
        }
    } else {
        count = 5000;
    }

    start_benchmark();
    for (j = 0; j < count; j++) {
        int retval = dictAdd(dict,stringFromLongLong(j),(void*)j);
        assert(retval == DICT_OK);
    }
    end_benchmark("Inserting");
    assert((long)dictSize(dict) == count);

    /* Wait for rehashing. */
    while (dictIsRehashing(dict)) {
        dictRehashMilliseconds(dict,100);
    }

    start_benchmark();
    for (j = 0; j < count; j++) {
        char *key = stringFromLongLong(j);
        dictEntry *de = dictFind(dict,key);
        assert(de != NULL);
        zfree(key);
    }
    end_benchmark("Linear access of existing elements");

    start_benchmark();
    for (j = 0; j < count; j++) {
        char *key = stringFromLongLong(j);
        dictEntry *de = dictFind(dict,key);
        assert(de != NULL);
        zfree(key);
    }
    end_benchmark("Linear access of existing elements (2nd round)");

    start_benchmark();
    for (j = 0; j < count; j++) {
        char *key = stringFromLongLong(rand() % count);
        dictEntry *de = dictFind(dict,key);
        assert(de != NULL);
        zfree(key);
    }
    end_benchmark("Random access of existing elements");
```

```
    start_benchmark();
    for (j = 0; j < count; j++) {
        dictEntry *de = dictGetRandomKey(dict);
        assert(de != NULL);
    }
    end_benchmark("Accessing random keys");

    start_benchmark();
    for (j = 0; j < count; j++) {
        char *key = stringFromLongLong(rand() % count);
        key[0] = 'X';
        dictEntry *de = dictFind(dict,key);
        assert(de == NULL);
        zfree(key);
    }
    end_benchmark("Accessing missing");

    start_benchmark();
    for (j = 0; j < count; j++) {
        char *key = stringFromLongLong(j);
        int retval = dictDelete(dict,key);
        assert(retval == DICT_OK);
        key[0] += 17; /* Change first number to letter. */
        retval = dictAdd(dict,key,(void*)j);
        assert(retval == DICT_OK);
    }
    end_benchmark("Removing and adding");
    dictRelease(dict);
    return 0;
}
```

代码位于dict.c文件第1252行， 该函数用于测试字典的性能， 测试的内容有：

- 现有元素的线性访问
- 随机访问现有元素
- 访问随机密钥
- 删除和添加

基准测试基础

```
void start_benchmark(void) {
    start = ustime();
}
void end_benchmark(char *title) {
    elapsed = ustime()-start;
    printf("%s: %lld microseconds\n", title, elapsed);
}
uint64_t hashCallback(const void *key) {
    return dictGenHashFunction((unsigned char*)key, strlen((char*)key));
}
```

```
int compareCallback(dict *d, const void *key1, const void *key2) {
    int l1,l2;
    UNUSED(d);

    l1 = strlen((char*)key1);
    l2 = strlen((char*)key2);
    if (l1 != l2) return 0;
    return memcmp(key1, key2, l1) == 0;
}

void freeCallback(dict *d, void *val) {
    UNUSED(d);

    zfree(val);
}

char *stringFromLongLong(long long value) {
    char buf[32];
    int len;
    char *s;

    len = snprintf(buf,sizeof(buf),"%lld",value);
    s = zmalloc(len+1);
    memcpy(s, buf, len);
    s[len] = '\0';
    return s;
}

dictType BenchmarkDictType = {
    hashCallback,
    NULL,
    NULL,
    compareCallback,
    freeCallback,
    NULL,
    NULL
}
```

代码位于`dict.c`文件第1203行， 该函数为基准测试提供了一些基础函数， 包括：

- `start_benchmark`函数用于记录开始时间
- `end_benchmark`函数用于记录结束时间并打印时间差
- `hashCallback`函数用于计算哈希值
- `compareCallback`函数用于比较两个键是否相等
- `freeCallback`函数用于释放键值对
- `stringFromLongLong`函数用于将整数转换为字符串
- `BenchmarkDictType`结构体用于初始化字典

结语

本文介绍了Redis的字典实现，其中很多操作都是基于哈希表的。想要读懂Redis中dict.c的源码，必须要对哈希表有一定的了解。dict.c让c语言的指针变得更加灵活，让c语言的结构体变得更加强大。实现了其它语言中内置的字典功能，让c语言也可以实现类似的功能。
本文中的代码都是基于Redis 7.0.5版本的，如果有错误，欢迎指正。

你可以在通过如下方式联系我：

- 邮箱:apexleapsean@gmail.com
- QQ:592841725
