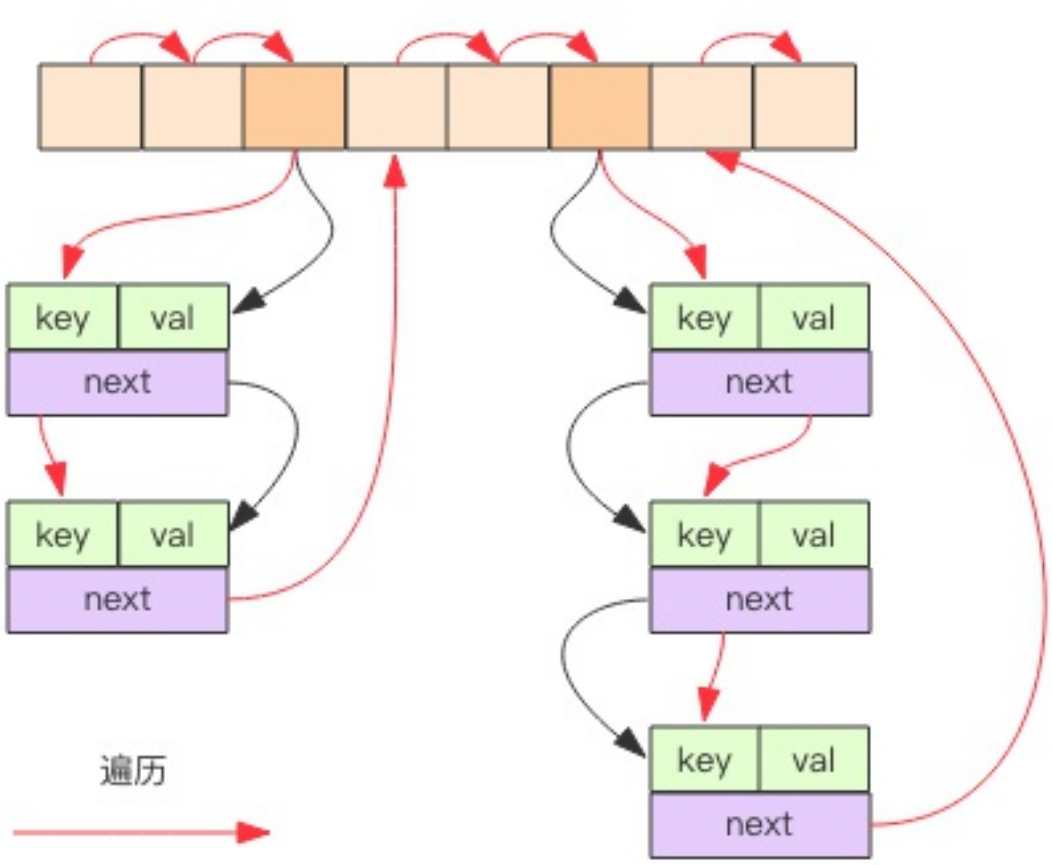
源码 10：跋山涉水 —— 深入字典遍历

Redis 字典的遍历过程逻辑比较复杂，互联网上对这一块的分析讲解非常少。我也花了不少时间对源码的细节进行了整理，将我个人对字典遍历逻辑的理解呈现给各位读者。也许读者们对字典的遍历过程有比我更好的理解，还请不吝指教。



一边遍历一边修改

我们知道 Redis 对象树的主干是一个字典，如果对象很多，这个主干字典也会很大。当我们使用 keys 命令搜寻指定模式的 key 时，它会遍历整个主干字典。值得注意的是，在遍历的过程中，如果满足模式匹配条件的 key 被找到了，还需要判断 key 指向的对象是否已经过期。如果过期了就需要从主干字典中将该 key 删除。

```
void keysCommand(client *c) {
    dictIterator *di; // 迭代器
    dictEntry *de; // 迭代器当前的entry
    sds pattern = c->argv[1]->ptr; // keys的匹配模式参数
    int plen = sdslen(pattern);
    int allkeys; // 是否要获取所有key，用于keys *这样的指令
    unsigned long numkeys = 0;
    void *replylen = addDeferredMultiBulkLength(c);

    // why safe?
    di = dictGetSafeIterator(c->db->dict);
    allkeys = (pattern[0] == '*' && pattern[1] == '\0');
    while((de = dictNext(di)) != NULL) {
        sds key = dictGetKey(de);
        robj *keyobj;

        if (allkeys || stringmatchlen(pattern,plen,key,sdslen(key),0)) {
            keyobj = createStringObject(key,sdslen(key));
            // 判断是否过期，过期了要删除元素
            if (expireIfNeeded(c->db,keyobj) == 0) {
```

```
                addReplyBulk(c,keyobj);
                numkeys++;
            }
            decrRefCount(keyobj);
        }
    }
    dictReleaseIterator(di);

setDeferredMultiBulkLength(c,replylen,numkeys);
}
```

那么，你是否想到了其中的困难之处，在遍历字典的时候还需要修改字典，会不会出现指针安全问题?

重复遍历

字典在扩容的时候要进行渐进式迁移，会存在新旧两个 hashtable。遍历需要对这两个 hashtable 依次进行，先遍历完旧的 hashtable，再继续遍历新的 hashtable。如果在遍历的过程中进行了 rehashStep，将已经遍历过的旧的 hashtable 的元素迁移到了新的 hashtable 中，那么遍历会不会出现元素的重复？这也是遍历需要考虑的疑难之处，下面我们来看看 Redis 是如何解决这个问题的。

迭代器的结构

Redis 为字典的遍历提供了 2 种迭代器，一种是安全迭代器，另一种是不安全迭代器。

```c
typedef struct dictIterator {
    dict *d; // 目标字典对象
    long index; // 当前遍历的槽位置，初始化为-1
    int table; // ht[0] or ht[1]
    int safe; // 这个属性非常关键，它表示迭代器是否安全
    dictEntry *entry; // 迭代器当前指向的对象
    dictEntry *nextEntry; // 迭代器下一个指向的对象
    long long fingerprint; // 迭代器指纹，放置迭代过程中字典被修改
} dictIterator;

// 获取非安全迭代器，只读迭代器，允许rehashStep
dictIterator *dictGetIterator(dict *d)
{
    dictIterator *iter = zmalloc(sizeof(*iter));

    iter->d = d;
    iter->table = 0;
    iter->index = -1;
    iter->safe = 0;
    iter->entry = NULL;
    iter->nextEntry = NULL;
    return iter;
}

// 获取安全迭代器，允许触发过期处理，禁止rehashStep
dictIterator *dictGetSafeIterator(dict *d) {
    dictIterator *i = dictGetIterator(d);

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

迭代器的「安全」指的是在遍历过程中可以对字典进行查找和修改，不用感到担心，因为查找和修改会触发过期判断，会删除内部元素。「安全」的另一层意思是迭代过程中不会出现元素重复，为了保证不重复，就会禁止 rehashStep。

而「不安全」的迭代器是指遍历过程中字典是只读的，你不可以修改，你只能调用 dictNext 对字典进行持续遍历，不得调用任何可能触发过期判断的函数。不过好处是不影响 rehash，代价就是遍历的元素可能会出现重复。

安全迭代器在刚开始遍历时，会给字典打上一个标记，有了这个标记，rehashStep 就不会执行，遍历时元素就不会出现重复。

```
typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    long rehashidx;
    // 这个就是标记，它表示当前加在字典上的安全迭代器的数量
    unsigned long iterators;
} dict;

// 如果存在安全的迭代器，就禁止rehash
static void _dictRehashStep(dict *d) {
    if (d->iterators == 0) dictRehash(d,1);
}
```

迭代过程

安全的迭代器在遍历过程中允许删除元素，意味着字典第一维数组下面挂接的链表中的元素可能会被摘走，元素的 next 指针就会发生变动，这是否会影响迭代过程呢？下面我们仔细研究一下迭代函数的代

码逻辑。

```
dictEntry *dictNext(dictIterator *iter)
{
    while (1) {
        if (iter->entry == NULL) {
            // 遍历一个新槽位下面的链表，数组的index往前移动了
            dictht *ht = &iter->d->ht[iter->table];
            if (iter->index == -1 && iter->table == 0) {
                // 第一次遍历，刚刚进入遍历过程
                // 也就是ht[0]数组的第一个元素下面的链表
                if (iter->safe) {
                  // 给字典打安全标记，禁止字典进行rehash
                  iter->d->iterators++;
                } else {
                  // 记录迭代器指纹，就好比字典的md5值
                  // 如果遍历过程中字典有任何变动，指纹就会改变
                  iter->fingerprint = dictFingerprint(iter->d);
                }
            }
            iter->index++; // index=0，正式进入第一个槽位
            if (iter->index >= (long) ht->size) {
                // 最后一个槽位都遍历完了
                if (dictIsRehashing(iter->d) && iter->table == 0) {
```

```
                    // 如果处于rehash中，那就继续遍历第二个 hashtable
                    iter->table++;
                    iter->index = 0;
                    ht = &iter->d->ht[1];
                } else {
                    // 结束遍历
                    break;
                }
            }
            // 将当前遍历的元素记录到迭代器中
            iter->entry = ht->table[iter->index];
        } else {
            // 直接将下一个元素记录为本次迭代的元素
            iter->entry = iter->nextEntry;
        }
        if (iter->entry) {
            // 将下一个元素也记录到迭代器中，这点非常关键
            // 防止安全迭代过程中当前元素被过期删除后，找不到下一个需要遍历的元素

            // 试想如果后面发生了rehash，当前遍历的链表被打散了，会发生什么
            // 这里要使劲发挥自己的想象力来理解
            // 旧的链表将一分为二，打散后重新挂接到新数组的两个槽位下
            // 结果就是会导致当前链表上的元素会重复遍历

            // 如果rehash的链表是index前面的链表，那么这部分链表也会被重复遍历
            iter->nextEntry = iter->entry->next;
            return iter->entry;
```

```
        }
    }
    return NULL;
}

// 遍历完成后要释放迭代器，安全迭代器需要去掉字典的禁止
rehash的标记
// 非安全迭代器还需要检查指纹，如果有变动，服务器就会奔溃
(failfast)
void dictReleaseIterator(dictIterator *iter)
{
    if (!(iter->index == -1 && iter->table == 0))
{
        if (iter->safe)
            iter->d->iterators--; // 去掉禁止rehash
的标记
        else
            assert(iter->fingerprint ==
dictFingerprint(iter->d));
    }
    zfree(iter);
}

// 计算字典的指纹，就是将字典的关键字段进行按位糅合到一起
// 这样只要有任意的结构变动，指纹都会发生变化
// 如果只是某个元素的value被修改了，指纹不会发生变动
long long dictFingerprint(dict *d) {
    long long integers[6], hash = 0;
    int j;

    integers[0] = (long) d->ht[0].table;
    integers[1] = d->ht[0].size;
    integers[2] = d->ht[0].used;
```

```
    integers[3] = (long) d->ht[1].table;
    integers[4] = d->ht[1].size;
    integers[5] = d->ht[1].used;

    for (j = 0; j < 6; j++) {
        hash += integers[j];
        hash = (~hash) + (hash << 21);
        hash = hash ^ (hash >> 24);
        hash = (hash + (hash << 3)) + (hash <<
8);
        hash = hash ^ (hash >> 14);
        hash = (hash + (hash << 2)) + (hash <<
4);
        hash = hash ^ (hash >> 28);
        hash = hash + (hash << 31);
    }
    return hash;
}
```

值得注意的是在字典扩容时进行 rehash，将旧数组中的链表迁移到新的数组中。某个具体槽位下的链表只可能会迁移到新数组的两个槽位中。

```
hash mod 2^n = k
hash mod 2^(n+1) = k or k+2^n
```

迭代器的选择

除了 keys 指令使用了安全迭代器，因为结果不允许重复。那还有其它的地方使用了安全迭代器么，什么情况下遍历适合使用非安全迭代器呢?

简单一点说，那就是如果遍历过程中不允许出现重复，那就使用 SafeIterator，比如下面的两种情况

1. bgaofrewrite 需要遍历所有对象转换称操作指令进行持久化，绝对不允许出现重复
2. bgsave 也需要遍历所有对象来持久化，同样不允许出现重复

如果遍历过程中需要处理元素过期，需要对字典进行修改，那也必须使用 SafeIterator，因为非安全的迭代器是只读的。

其它情况下，也就是允许遍历过程中出现个别元素重复，不需要对字典进行结构性修改的情况下一律使用非安全迭代器。

思考

请继续思考 rehash 对非安全遍历过程的影响，会重复哪些元素，重复的元素会非常多么还是只是少量重复?