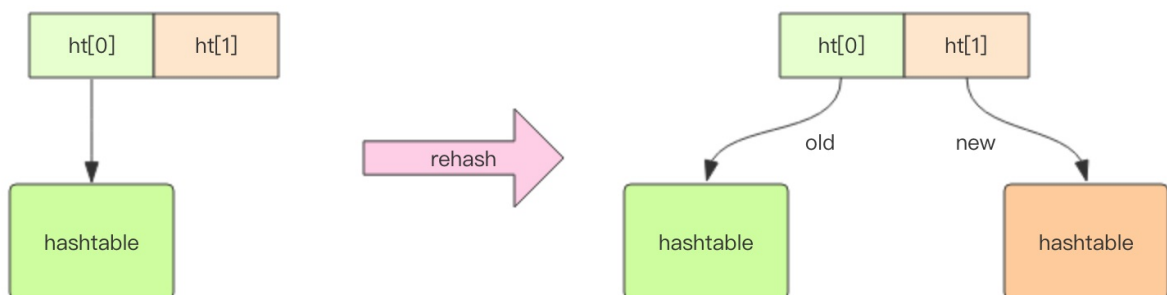


源码 2：循序渐进 —— 探索「字典」内部

dict 是 Redis 服务器中出现最为频繁的复合型数据结构，除了 hash 结构的数据会用到字典外，整个 Redis 数据库的所有 key 和 value 也组成了一个全局字典，还有带过期时间的 key 集合也是一个字典。zset 集合中存储 value 和 score 值的映射关系也是通过 dict 结构实现的。

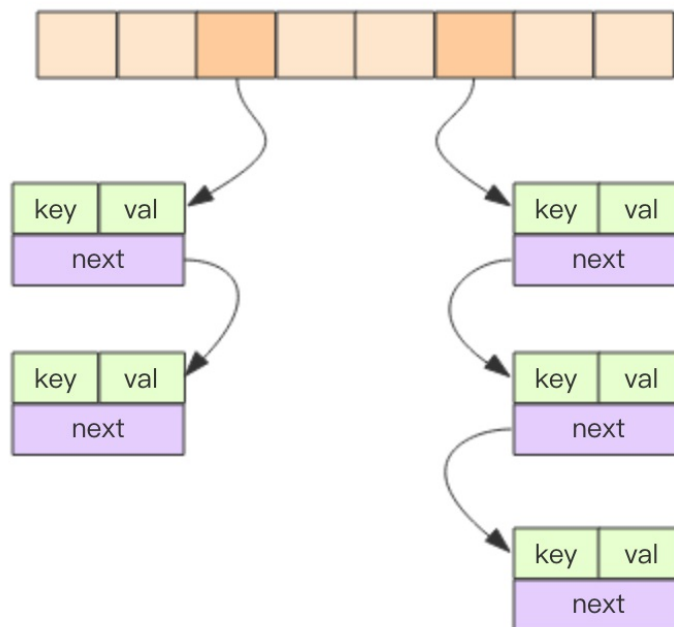
```
struct RedisDb {  
    dict* dict; // all keys  key=>value  
    dict* expires; // all expired keys  
    key=>long(timestamp)  
    ...  
}  
  
struct zset {  
    dict *dict; // all values  value=>score  
    zskiplist *zsl;  
}
```

dict 内部结构



dict 结构内部包含两个 hashtable，通常情况下只有一个 hashtable 是有值的。但是在 dict 扩容缩容时，需要分配新的 hashtable，然后进行渐进式搬迁，这时候两个 hashtable 存储的分别是旧的 hashtable 和新的 hashtable。待搬迁结束后，旧的 hashtable 被删除，新的 hashtable 取而代之。

```
struct dict {  
    ...  
    dictht ht[2];  
}
```



所以，字典数据结构的精华就落在了 hashtable 结构上了。hashtable 的结构和 Java 的 HashMap 几乎是一样的，都是通过分桶的方式解决 hash 冲突。第一维是数组，第二维是链表。数组中存储的是第二维链表的第一个元素的指针。

```
struct dictEntry {
    void* key;
    void* val;
    dictEntry* next; // 链接下一个 entry
}
struct dictht {
    dictEntry** table; // 二维
    long size; // 第一维数组的长度
    long used; // hash 表中的元素个数
    ...
}
```

渐进式rehash

大字典的扩容是比较耗时间的，需要重新申请新的数组，然后将旧字典所有链表中的元素重新挂接到新的数组下面，这是一个O(n)级别的操作，作为单线程的Redis表示很难承受这样耗时的过程。步子迈大了会扯着蛋，所以Redis使用渐进式rehash小步搬迁。虽然慢一点，但是肯定可以搬完。

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

    // 这里进行小步搬迁
    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. */
    // 如果字典处于搬迁过程中，要将新的元素挂接到新的数组
下面
    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;
}

```

搬迁操作埋伏在当前字典的后续指令中(来自客户端的hset/hdel指令等)，但是有可能客户端闲下来了，没有了后续指令来触发这个搬迁，那么Redis就置之不理了么？当然不会，优雅的Redis怎么可能设计的这样潦草。Redis还会在定时任务中对字典进行主动搬迁。

```
// 服务器定时任务
void databaseCron() {
    ...
    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 the next one. */
                rehash_db++;
                rehash_db %= server.dbnum;
            }
        }
    }
}
}
```

查找过程

插入和删除操作都依赖于查找，先必须把元素找到，才可以进行数据结构的修改操作。hashtable 的元素是在第二维的链表上，所以首先我们得想办法定位出元素在哪个链表上。

```
func get(key) {  
    let index = hash_func(key) % size;  
    let entry = table[index];  
    while(entry != NULL) {  
        if entry.key == target {  
            return entry.value;  
        }  
        entry = entry.next;  
    }  
}
```

值得注意的是代码中的`hash_func`，它会将 `key` 映射为一个整数，不同的 `key` 会被映射成分布比较均匀散乱的整数。只有 `hash` 值均匀了，整个 `hashtable` 才是平衡的，所有的二维链表的长度就不会差距很远，查找算法的性能也就比较稳定。

hash 函数

`hashtable` 的性能好不好完全取决于 `hash` 函数的质量。`hash` 函数如果可以将 `key` 打散的比较均匀，那么这个 `hash` 函数就是个好函数。Redis 的字典默认的 `hash` 函数是 `siphash`。`siphash` 算法即使在输入 `key` 很小的情况下，也可以产生随机性特别好的输出，而且它的性能也非常突出。对于 Redis 这样的单线程来说，字典数据结构如此普遍，字典操作也会非常频繁，`hash` 函数自然也是越快越好。

hash 攻击

如果 `hash` 函数存在偏向性，黑客就可能利用这种偏向性对服务器进行攻击。存在偏向性的 `hash` 函数在特定模式下的输入会导致 `hash` 第二维链表长度极为不均匀，甚至所有的元素都集中到个别链表中，

直接导致查找效率急剧下降，从 $O(1)$ 退化到 $O(n)$ 。有限的服务器计算能力将会被 hashtable 的查找效率彻底拖垮。这就是所谓 hash 攻击。

扩容条件

```

/* 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 doubling
    * 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;
}

```

正常情况下，当 hash 表中元素的个数等于第一维数组的长度时，就会开始扩容，扩容的新数组是原数组大小的 2 倍。不过如果 Redis 正在做 bgsave，为了减少内存页的过多分离 (Copy On Write)，Redis 尽量不去扩容 (dict_can_resize)，但是如果 hash 表已经

非常满了，元素的个数已经达到了第一维数组长度的 5 倍 (dict_force_resize_ratio)，说明 hash 表已经过于拥挤了，这个时候就会强制扩容。

缩容条件

```
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));
}
```

当 hash 表因为元素的逐渐删除变得越来越稀疏时，Redis 会对 hash 表进行缩容来减少 hash 表的第一维数组空间占用。缩容的条件是元素个数低于数组长度的 10%。缩容不会考虑 Redis 是否正在做 bgsave。

set 的结构

Redis 里面 set 的结构底层实现也是字典，只不过所有的 value 都是 NULL，其它的特性和字典一模一样。

思考

1. 为什么缩容不用考虑 bgsave?
2. Java 语言和 Python 语言内置的 set 容器是如何实现的?