

Hash Table

哈希表

描述:

哈希表是一种高效的 **key-value** 存储结构，时间复杂度为 $O(1)$ ，通过散列函数对 **key** 值计算，得到 **value** 在哈希表数组中应该存储的位置。散列函数/哈希函数是哈希表的核心，一般形式为 $index = hash(key)$ ，具有速度快的优势，设计好的散列函数的碰撞率低、数据分布平均的优点，可以将任意情况下的数据都分散在哈希表内部的数组中。

将散列函数 **hash** 计算出 **key** 的哈希值作为这个 **key** 在哈希表中的存储位置，即数组下标 **index**。很多哈希函数的结果都是一个 32 位或 64 位的整数，远远大于哈希表中数组的长度，实际应用时需要再对数组长度 **n** 取模，即 $index = hash(key) \% n$ 。

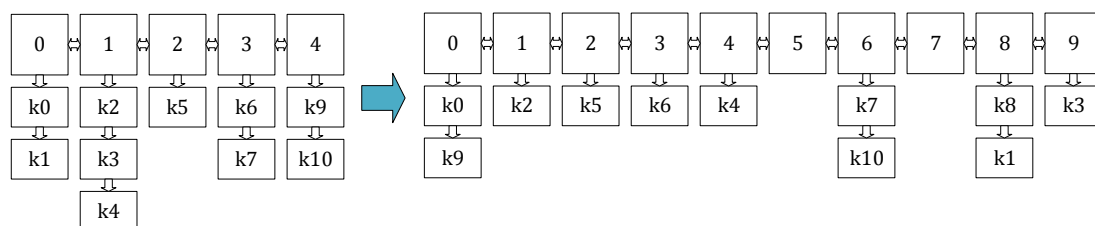
当哈希表中存储的数据量 **count** 超过 **n** 时，显然某些 **key** 会存储在相同的下标中，即碰撞情况，设计良好的散列函数可以让 **key** 均匀的分布在数组中，保证存取每个 **key** 的时间复杂度近似为 $O(1)$ 。

当数据量 **count** 过多时，可以将数组 **n** 的长度扩充，然后重新哈希所有 **key**，找出在新数组中的存储位置，同一个 **key** 的哈希值不变，但对于不同长度的数组其存储位置会改变。Redis (<http://redis.io/>) 中哈希表的实现使用了一种分步哈希的优化来将这个时间复杂度为 $O(count)$ 的操作平摊到每个 **insert**、**find** 和 **remove** 操作中，得到了一种近似 $O(1)$ 的时间复杂度。

具体的设计如下:

```
struct HashTable {  
    struct RealHashTable t1;  
    struct RealHashTable t2;  
    int rehash_index;  
};
```

HashTable 实际内部有两个哈希表，**t1** 是主要哈希表，**t2** 作为哈希表扩充时的暂时存放处，设 **n1**、**n2** 分别为 **t1**、**t2** 的数组长度。HashTable 在非扩充/一般状态下，**rehash_index** = -1，这时只使用 **t1** 来进行实际的哈希表 **insert**、**find**、**remove** 操作；当数据量 **count** 过大（比如 $count > 5 \times n1$ ）时 HashTable 进入扩充/重新哈希状态，比如下图所示的情况：



整个重新哈希会先分配一个更大的数组 **t2**（比如 $n2 = 2 \times n1$ ），从**rehash_index** = 0开始，**rehash_index** 代表当前正在哈希的数组下标。每一次 **insert**、**find** 或 **remove** 操作时，若**rehash_index** $\in [0, n)$ 则表明处于重新哈希状态，则（1）**insert** 操作中新加入的 **key** 应该插入 **t2** 中；（2）**find** 操作应该对 **t1** 和 **t2** 都进行查找；（3）**remove** 同样对 **t1** 和 **t2** 都进行查找，至多找到一个 **key** 并删除该 **key**。并且在以上 3 种操作之前，都将 **t1[rehash_index]** 列表中的所有 **key** 重新哈希到 **t2** 中，从而将重新哈希的操作平摊为 $O(1)$ ，使得数据结构的性能更加平稳。