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，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，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。

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。

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的操作平摊到每个inesrt、find和remove操作中，得到了一种近似的时间复杂度。

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，，这时只使用t1来进行实际的哈希表inesrt、find、remove操作；当数据量count过大（比如）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），从开始，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。

我们的源码示例中使用了MurmurHash2（https://sites.google.com/site/murmurhash/）作为哈希函数。