算法中的数据结构

七月算法　邹博

2015年6月28日

下面这段代码，会输出什么？

- 5 or 6

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int c = 5;
    int* p = (int*)&c;
    *p = 6;
    cout << c;
    int x = c;

    int i = 8;
    int j = i;
    int* p2 = &i;
    return 0;
}
```

深层理解
const & pointer

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int c = 5;
    int* p = (int*)&c;
    *p = 6;
    cout << c;
    int x = c;

    int i = 8;
    int j = i;
    int* p2 = &i;
    return 0;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
0041BD60  push        ebp
0041BD61  mov         ebp,esp
0041BD63  sub         esp,108h
0041BD69  push        ebx
0041BD6A  push        esi
0041BD6B  push        edi
0041BD6C  lea         edi,[ebp-108h]
0041BD72  mov         ecx,42h
0041BD77  mov         eax,0CCCCCCCCh
0041BD7C  rep stos    dword ptr [edi]
    const int c = 5;
0041BD7E  mov         dword ptr [c],5
    int* p = (int*)&c;
0041BD85  lea         eax,[c]
0041BD88  mov         dword ptr [p],eax
    *p = 6;
0041BD8B  mov         eax,dword ptr [p]
0041BD8E  mov         dword ptr [eax],6
    cout << c;
0041BD94  push        5
0041BD96  mov         ecx,offset std::cout (457668h)
0041BD9B  call        ostream<char,char_traits<char> >::operator<<(4195D2h)
    int x = c;
0041BDA0  mov         dword ptr [x],5

    int i = 8;
0041BDA7  mov         dword ptr [i],8
    int j = i;
0041BDAE  mov         eax,dword ptr [i]
0041BDB1  mov         dword ptr [j],eax
    int* p2 = &i;
0041BDB4  lea         eax,[i]
0041BDB7  mov         dword ptr [p2],eax
    return 0;
0041BDBA  xor         eax,eax
}
```

从上述汇编分析得出结论

- 指针是汇编级直接支持的结构：
  - i=func();
    - 伪代码：mov dword ptr [i], func
  - p是指针：即p的值表示了某内存地址。
    - *p = i;
      - lea eax,[i]
      - mov dword ptr [p], exa
- const是高级语言在编译期间实现的内容：
  - const int c = 5;
    - move dword ptr [c], 5
  - int x = c;
    - mov dword ptr [x], 5

Code

- 这段代码有问题吗？

Kitty	Puppy
1	2

```
class animal
{
protected:
    int age;
public:
    virtual void MyAge(void) = 0;
};

class dog : public animal
{
public:
    dog()
    {
        age = 2;
    }
    ~dog() {}
    virtual void MyAge(void)
    {
        cout<< "Wang, my age = " << age <<endl;
    }
};

class cat: public animal
{
public:
    cat()
    {
        age = 1;
    }
    ~cat() {}
    virtual void MyAge(void)
    {
        cout<< "Miao, my age = " << age << endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    cat kitty;
    dog puppy;
    kitty = puppy;
    return 0;
}
```

使用指针的方法传递

```
int _tmain(int argc, _TCHAR* argv[])
{
    cat kitty;
    dog puppy;
    animal* pKitty = &kitty;
    animal* pPuppy = &puppy;
    *pKitty = *pPuppy;
    kitty.MyAge();
    return 0;
}
```

Kitty	Puppy
1	2

如果换成其他指针呢？

```
int _tmain(int argc, _TCHAR* argv[])
{
    cat kitty;
    dog puppy;
    int* pKitty = (int*)&kitty;
    int* pPuppy = (int*)&puppy;
    *pKitty = *pPuppy;
    kitty.MyAge();
    return 0;
}
```

Kitty	Puppy
1	2

Kitty	Puppy
virtual	virtual
1	2

思考

- 可以看出，所谓指针的类型，仅为高级语言加入的特性，在汇编层，都是变量的地址而已。
- 这段代码输出什么？

```
class animal
{
private:
    int age;
public:
    animal(int _age)
    {
        age = _age;
    }
    void PrintAge()
    {
        cout << age << endl;
    }

};

int _tmain(int argc, _TCHAR* argv[])
{
    animal a(1);
    int* p = (int*)&a;
    *p = 2;
    a.PrintAge();
    return 0;
}
```

一种NoSQL数据库 - Riak

从事过建筑的人都知道，有种用来加强混凝土的钢条，称作钢筋。正如 Riak（读作”Ree-ahck”）一样，钢筋不会单独使用，它往往用于互相作用的各个部分，以使整个系统持久耐用。于是，系统中的每个组件都廉价又不起眼，但只要使用得当，就可构建起足够简单且牢固的基础结构。

Riak 是一种分布式的键-值（key-value）数据库。其中，值可以是任何类型的数据，如普通文本、JSON、XML、图片，甚至视频片段；而所有这些都可以通过普通的 HTTP 接口访问。你有什么，Riak 就能存什么。

容错是 Riak 的另一特性。服务器可在任何时刻启动或者停止，而不会引起任何单点故障。不管是增加或者移除服务器，甚至有节点崩溃（谁都不想这样），集群依然可以持续忙碌地运行。Riak 让你不再整夜无眠担心集群，某个节点失效不再是紧急事件，完全可以等到第二天早晨处理。Riak 的核心开发者 Justin Sheehy 曾提到：“（Riak 团队）非常注重可写入性……为的是可以回家睡觉。”

然而万事都有利弊取舍，Riak 的灵活性自有其代价。对于自由定义的（ad hoc）查询，Riak 缺乏有力支持；而键-值存储的设计，使得数据值无法相互连接（即，Riak 没有外键）。Riak 试图将这些问题各个击破，我们会在后面几天读到相关内容。

字典

- 字典，又称符号表(symbol table)、关联数组(associative array)或者映射(map)，是一种用于保存键值对(key-value pair)的抽象数据结构。
- 在字典中，一个键(key)可以和一个值(value)进行关联(或者说将键映射为值)，这些关联的键和值就被称为键值对。
- 字典中的每个键都是独一无二的，程序可以在字典中根据键查找与之关联的值，或者通过键来更新值，又或者根据键来删除整个键值对，等等。
- 字典经常作为一种数据结构内置在很多高级编程语言里面，但Redis所使用的C语言并没有内置这种数据结构，因此Redis构建了自己的字典实现。

字典

- 字典在Redis中的应用相当广泛，比如Redis的数据库就是使用字典来作为底层实现的，对数据库的增、删、查、改操作也是构建在对字典的操作之上的。
- 除了用来表示数据库之外，字典还是哈希键的底层实现之一：当一个哈希键包含的键值对比较多，又或者键值对中的元素都是比较长的字符串时，Redis就会使用字典作为哈希键的底层实现。

字典的实现

- Redis的字典使用哈希表作为底层实现，一个哈希表里面可以有多个哈希表节点，而每个哈希表节点就保存了字典中的一个键值对。

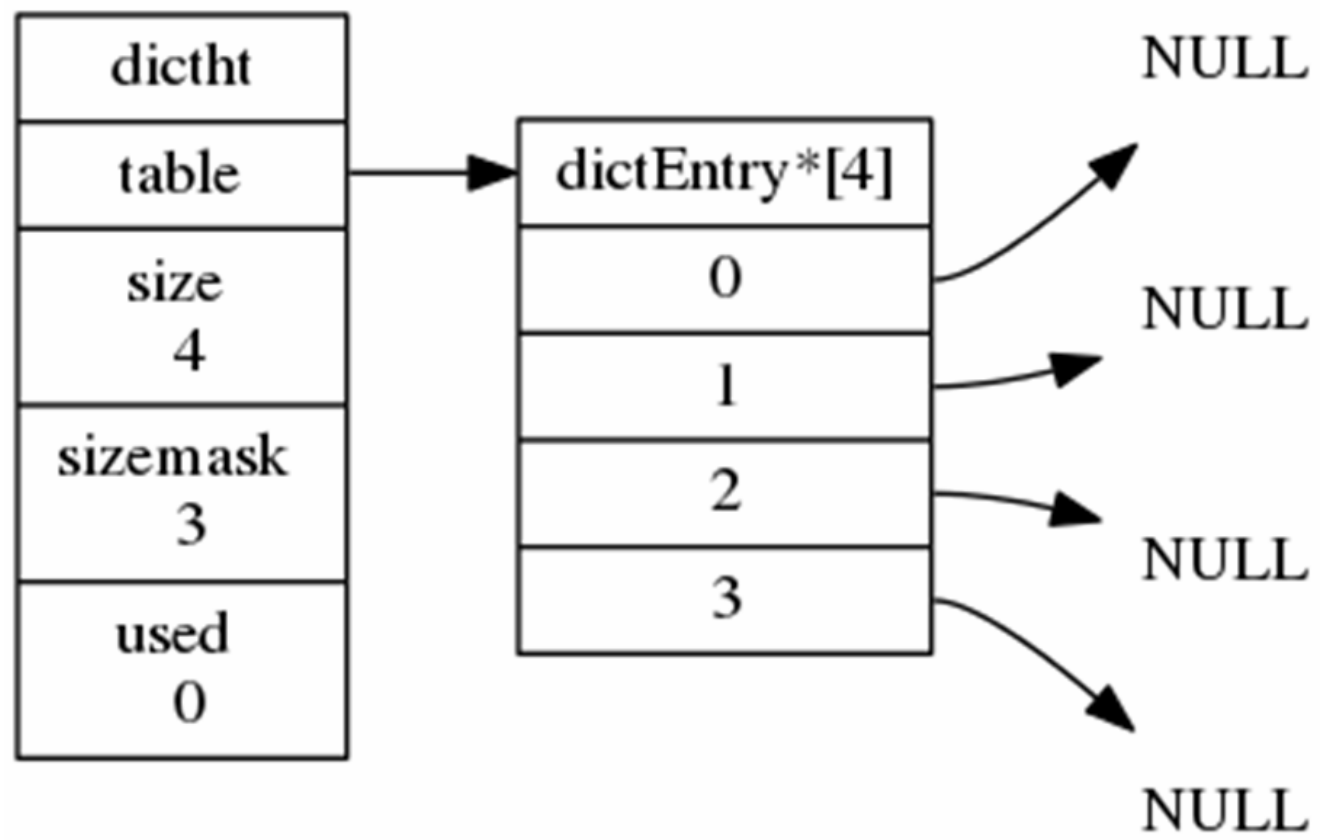
哈希表

```
typedef struct dictht {

    // 哈希表数组
    dictEntry **table;

    // 哈希表大小
    unsigned long size;

    // 哈希表大小掩码，用于计算索引值
    // 总是等于 size - 1
    unsigned long sizemask;

    // 该哈希表已有节点的数量
    unsigned long used;

} dictht;
```

哈希表

- table属性是一个数组，数组中的每个元素都是一个指向dictEntry结构的指针，每个dictEntry结构保存着一个键值对。
- size属性记录了哈希表的大小，也即是table数组的大小，而used属性则记录了哈希表目前已有节点(键值对)的数量。
- sizemask属性的值总是等于size-1，这个属性和哈希值一起决定一个键应该被放到table数组的哪个索引上面。

一个大小为 4 的空哈希表



哈希表节点

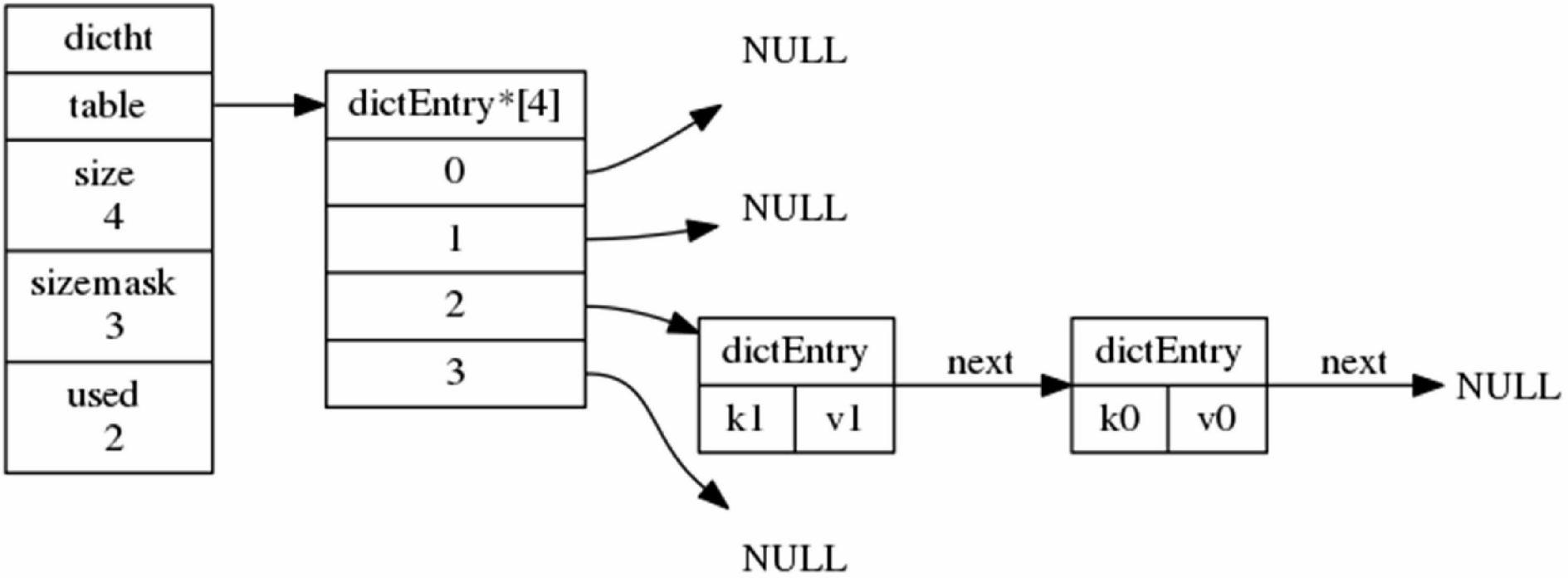
- 哈希表节点使用dictEntry结构表示，每个dictEntry结构都保存着一个键值对。

```
typedef struct dictEntry {

    // 键
    void *key;

    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;

    // 指向下个哈希表节点，形成链表
    struct dictEntry *next;

} dictEntry;
```

哈希表节点

- key属性保存着键值对中的键，而val属性则保存着键值对中的值，其中键值对的值可以是一个指针，或者是一个uint64_t整数，又或者是一个int64_t整数。
- next属性是指向另一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对连接在一次，以此来解决键冲突(collision)的问题。

两个索引值相同的键k1和k0



字典

```
typedef struct dict {

    // 类型特定函数
    dictType *type;

    // 私有数据
    void *privdata;

    // 哈希表
    dictht ht[2];

    // rehash 索引
    // 当 rehash 不在进行时，值为 -1
    int rehashidx;

} dict;
```

字典的多态

- type属性和privdata属性是针对不同类型的键值对，为创建多态字典而设置的：
- type属性是一个指向dictType结构的指针，每个dictType结构保存了一簇用于操作特定类型键值对的函数，Redis会为用途不同的字典设置不同的类型特定函数。
- privdata属性则保存了需要传给那些类型特定函数的可选参数。

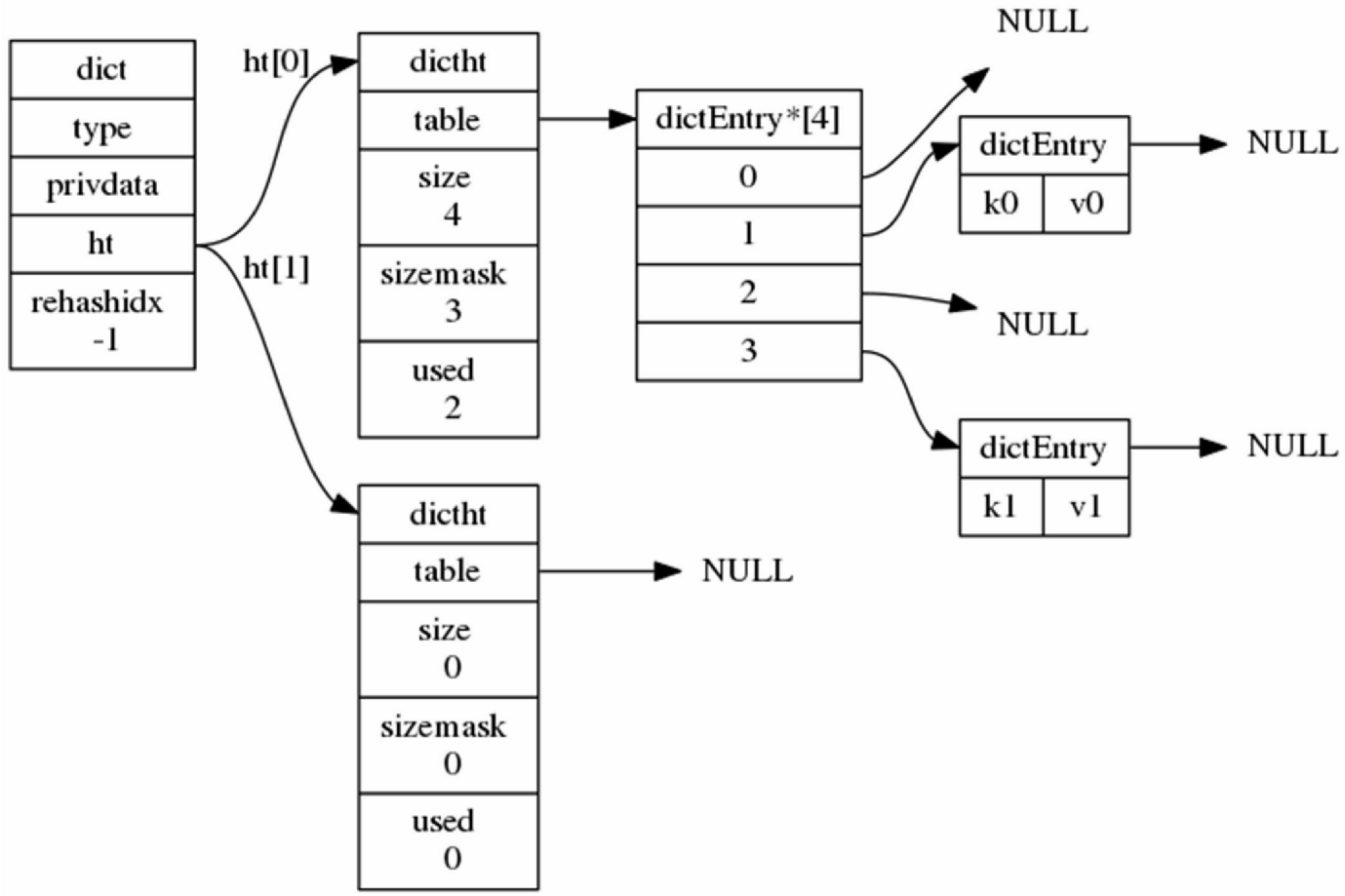
字典的多态

```
typedef struct dictType {

    // 计算哈希值的函数
    unsigned int (*hashFunction)(const void *key);

    // 复制键的函数
    void *(*keyDup)(void *privdata, const void *key);

    // 复制值的函数
    void *(*valDup)(void *privdata, const void *obj);

    // 对比键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);

    // 销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);

    // 销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);

} dictType;
```

Rehash

- ht属性是一个包含两个项的数组，数组中的每个项都是一个dictht哈希表，一般情况下，字典只使用ht[0]哈希表，ht[1]哈希表只会在对ht[0]哈希表进行rehash时使用。
- 除了ht[1]之外，另一个和rehash有关的属性就是rehashidx：它记录了rehash目前的进度，如果目前没有在进行rehash，那么它的值为-1。

普通状态下的字典


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
2
dictEntry*[4]
0
1
2
3
NULL
dictEntry
k0
v0
NULL
NULL
dictEntry
k1
v1
NULL
dictht
table
size
0
sizemask
0
used
0
NULL

哈希算法

- 当要将一个新的键值对添加到字典里面时，程序需要先根据键值对的键计算出哈希值和索引值，然后再根据索引值，将包含新键值对的哈希表节点放到哈希表数组的指定索引上面。
- 当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。

Redis计算哈希值和索引值的方法

```
# 使用字典设置的哈希函数，计算键 key 的哈希值
hash = dict->type->hashFunction(key);

# 使用哈希表的 sizemask 属性和哈希值，计算出索引值
# 根据情况不同， ht[x] 可以是 ht[0] 或者 ht[1]
index = hash & dict->ht[x].sizemask;
```

djb2

- this algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c. another version of this algorithm (now favored by bernstein) uses xor: hash(i) = hash(i - 1) * 33 ^ str[i]; the magic of number 33 (why it works better than many other constants, prime or not) has never been adequately explained.
- 这个算法(k=33)是多年前首先由dan bernstein 在comp.lang.c中提出的，这个算法的另外一个版本(bernstein的贡献)是使用异或方式：hash(i) = hash(i - 1) * 33 ^ str[i];魔数33尚未得到足够的解释(为何它能够比其他很多素数的效果更好)

djb2 Code

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

sdbm Code(65599)

```
static unsigned long
sdbm(str)
unsigned char *str;
{
    unsigned long hash = 0;
    int c;

    while (c = *str++)
        hash = c + (hash << 6) + (hash << 16) - hash;

    return hash;
}
```

sdbm的说明

- This algorithm was created for sdbm (a public-domain reimplementation of ndbm) database library. it was found to do well in scrambling bits, causing better distribution of the keys and fewer splits.
- It also happens to be a good general hashing function with good distribution. the actual function is hash(i) = hash(i - 1) * 65599 + str[i]; which is the faster version used in gawk. The magic constant 65599 was picked out of thin air while experimenting with different constants, and turns out to be a prime. this is one of the algorithms used in berkeley db and elsewhere.

MurmurHash

- MurmurHash是一种非加密型哈希函数，适用于一般的哈希检索操作。由AustinAppleby在2008年发明，并出现了多个变种，都已经发布到了公有领域(public domain)。与其它流行的哈希函数相比，对于规律性较强的key，Murmur Hash的随机分布特征表现更良好。
- 当前的版本是MurmurHash3，能够产生出32-bit或128-bit哈希值。
- 较早的MurmurHash2能产生32-bit或64-bit哈希值。对于大端存储和强制对齐的硬件环境有一个较慢的MurmurHash2可以用。MurmurHash2A变种增加了Merkle–Damgård构造，所以能够以增量方式调用。有两个变种产生64-bit哈希值：MurmurHash64A，为64位处理器做了优化；MurmurHash64B，为32位处理器做了优化。MurmurHash2-160用于产生160-bit哈希值，而MurmurHash1已经不再使用。

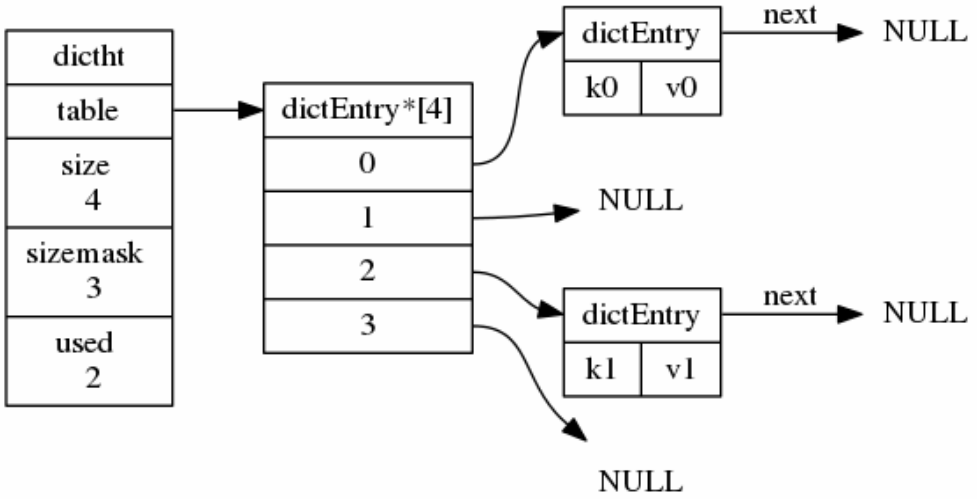
MurmurHash

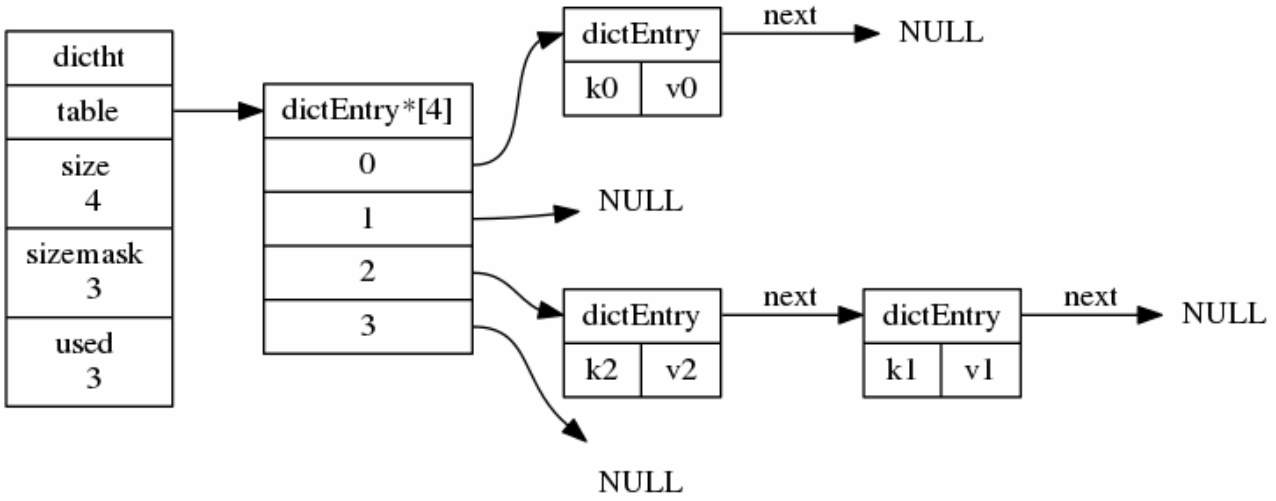
```
Murmur3_32(key, len, seed)
    c1 ← 0xcc9e2d51
    c2 ← 0x1b873593
    r1 ← 15
    r2 ← 13
    m ← 5
    n ← 0xe6546b64

    hash ← seed

    for each fourByteChunk of key
        k ← fourByteChunk

        k ← k * c1
        k ← (k << r1) OR (k >> (32-r1))
        k ← k * c2

        hash ← hash XOR k
        hash ← (hash << r2) OR (hash >> (32-r2))
        hash ← hash * m + n

    with any remainingBytesInKey
        remainingBytes ← SwapEndianOrderOf(remainingBytesInKey)
        remainingBytes ← remainingBytes * c1
        remainingBytes ← (remainingBytes << r1) OR (remainingBytes >> (32 - r1))
        remainingBytes ← remainingBytes * c2

        hash ← hash XOR remainingBytes

    hash ← hash XOR len

    hash ← hash XOR (hash >> 16)
    hash ← hash * 0x85ebca6b
    hash ← hash XOR (hash >> 13)
    hash ← hash * 0xc2b2ae35
    hash ← hash XOR (hash >> 16)
```

哈希冲突

- 当有两个或以上数量的键被分配到了哈希表数组的同一个索引上面时，我们称这些键发生了冲突(collision)。
- Redis的哈希表使用链地址法(separate chaining)来解决键冲突：每个哈希表节点都有一个next指针，多个哈希表节点可以用next指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表连接起来，这就解决了键冲突的问题。

冲突


dictht
table
size
4
sizemask
3
used
2
dictEntry*[4]
0
1
2
3
dictEntry
k0
v0
next
NULL
NULL
dictEntry
k1
v1
next
NULL
NULL


dictht
table
size
4
sizemask
3
used
3
dictEntry*[4]
0
1
2
3
dictEntry
k0
v0
next
NULL
NULL
dictEntry
k2
v2
next
dictEntry
k1
v1
next
NULL
NULL

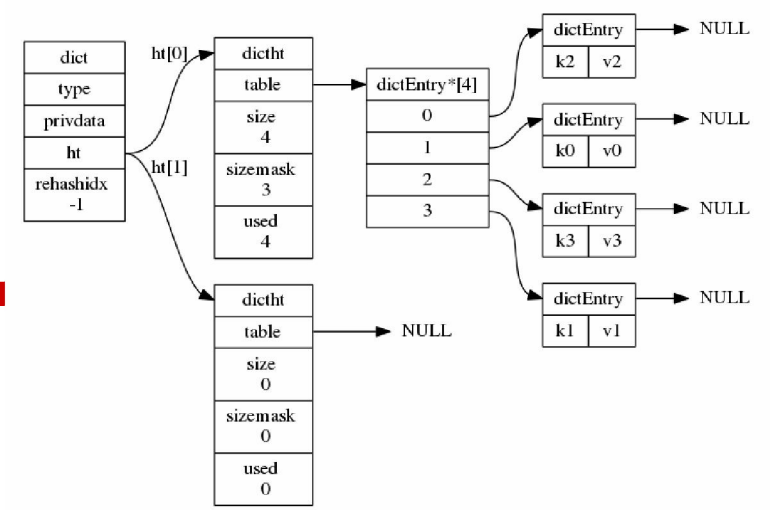
Rehash

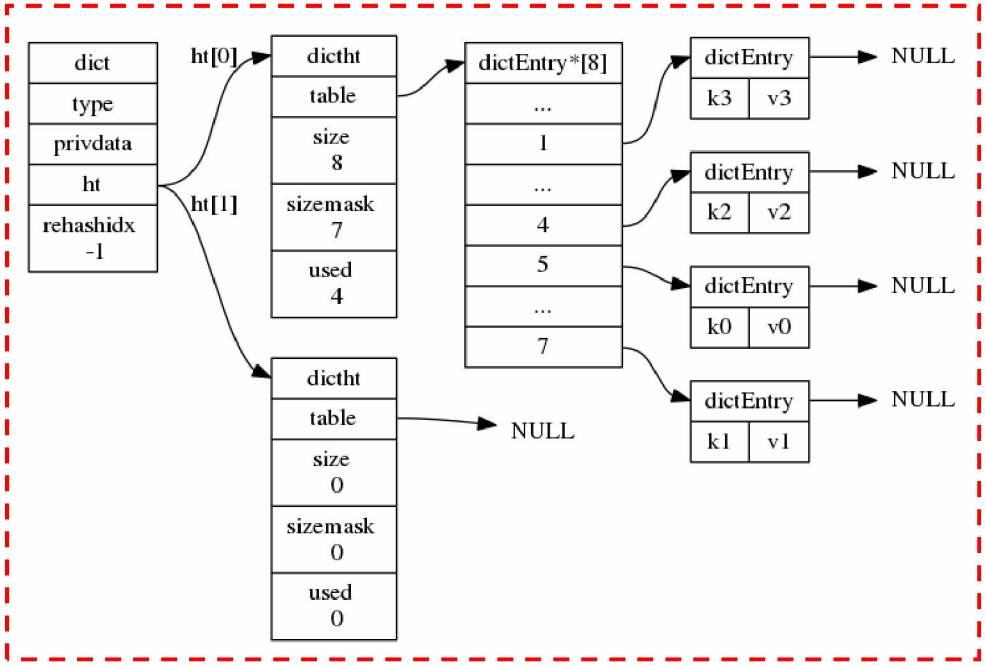
- rehash是在hash table的大小不能满足需求，造成过多hash碰撞后需要进行的扩容hash table的操作，其实通常的做法确实是建立一个额外的hash table，将原来的hash table中的数据在新的数据中进行重新输入，从而生成新的hash表。
- 优先使用0号hash table，当空间不足时会调用dictExpand来扩展hash table，此时准备1号hash table用于增量的rehash使用。rehash完成后把0号释放，1号保存到0号。
- rehashidx是下一个需要rehash的项在ht[0]中的索引，不需要rehash时置为-1。也就是说-1时，表示不进行rehash。

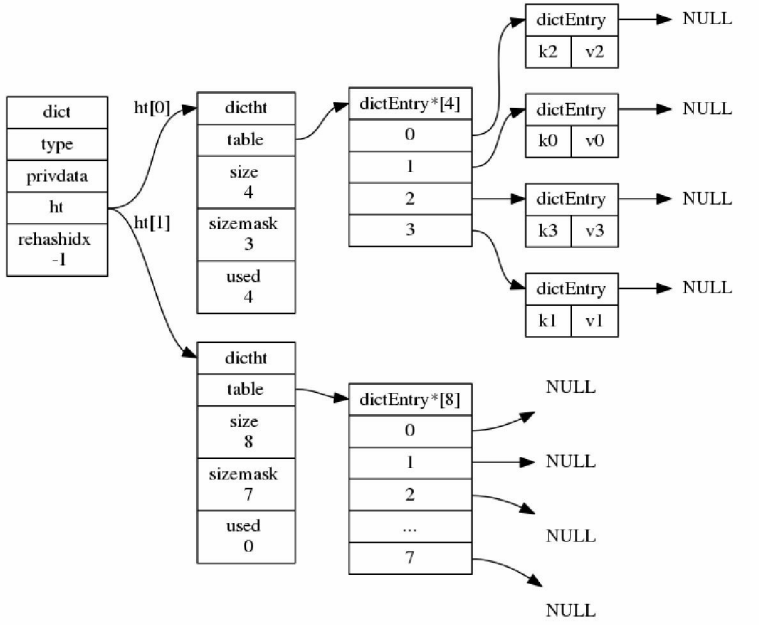
Rehash

- 随着操作的不断执行，哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表的负载因子(load factor)维持在一个合理的范围之内，当哈希表保存的键值对数量太多或者太少时，程序需要对哈希表的大小进行相应的扩展或者收缩。
- 扩展和收缩哈希表的工作可以通过执行rehash(重新散列)操作来完成，Redis对字典的哈希表执行rehash的步骤如下：
  - 为字典的ht[1]哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及ht[0]当前包含的键值对数量(也即是ht[0].used属性的值)：
    - 如果执行的是扩展操作，那么ht[1]的大小为大于等于ht[0].used*2的2的n次方幂的最小值；
    - 如果执行的是收缩操作，那么ht[1]的大小为大于等于ht[0].used的2的n次方幂的最小值。
  - 将保存在ht[0]中的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]哈希表的指定位置上。
  - 当ht[0]包含的所有键值对都迁移到了ht[1]之后(ht[0]变为空表)，释放ht[0]，将ht[1]设置为ht[0]，并在ht[1]新创建一个空白哈希表，为下一次rehash做准备。

Rehash


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
4
dictEntry*[4]
0
1
2
3
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
NULL
size
0
sizemask
0
used
0

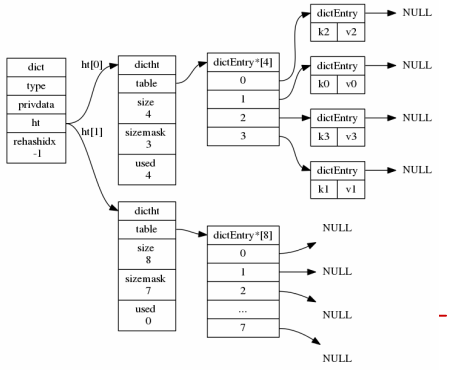

dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
8
sizemask
7
used
4
dictEntry*[8]
...
1
...
4
5
...
7
dictEntry
k3
v3
NULL
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k1
v1
NULL
dictht
table
NULL
size
0
sizemask
0
used
0


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
4
dictEntry*[4]
0
1
2
3
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
0
dictEntry*[8]
0
1
2
...
7
NULL
NULL
NULL
NULL

渐进式rehash

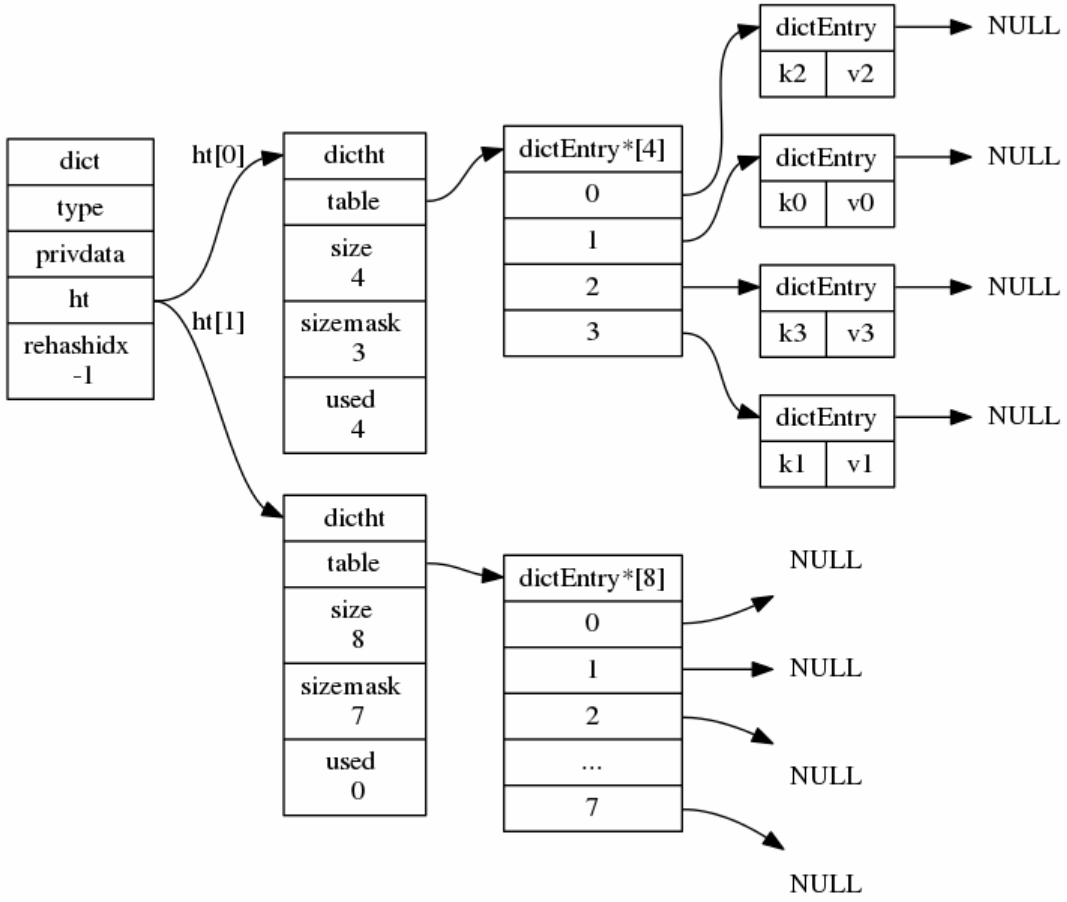
- 扩展或收缩哈希表需要将ht[0]里面的所有键值对rehash到ht[1]里面，但是，这个rehash动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。
- 这样做的原因在于，如果ht[0]里只保存着四个键值对，那么服务器可以在瞬间就将这些键值对全部rehash到ht[1]；但是，如果哈希表里保存的键值对数量不是四个，而是四百万、四千万甚至四亿个键值对，那么要一次性将这些键值对全部rehash到ht[1]的话，庞大的计算量可能会导致服务器在一段时间内停止服务。
- 因此，为了避免rehash对服务器性能造成影响，服务器不是一次性将ht[0]里面的所有键值对全部rehash到ht[1]，而是分多次、渐进式地将ht[0]里面的键值对慢慢地rehash到ht[1]。

哈希表渐进rehash详细步骤

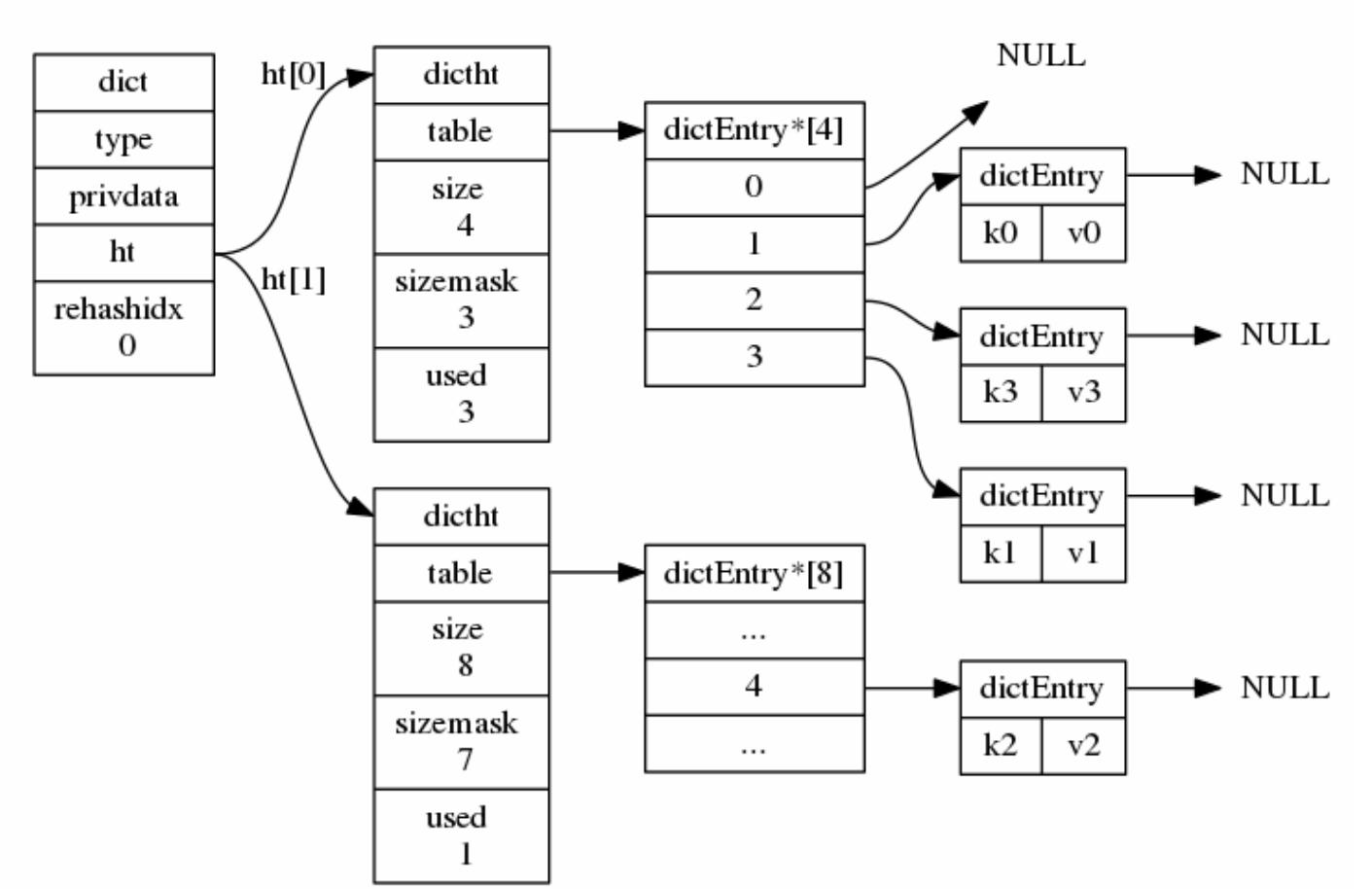


- 为ht[1]分配空间，字典持有ht[0]和ht[1]两个哈希表。
- 在字典中维持索引计数器变量rehashidx，并将它的值设置为0，表示rehash工作正式开始。
- 在rehash进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将ht[0]哈希表在rehashidx索引上的所有键值对rehash到ht[1]，当rehash工作完成之后，程序将rehashidx的值加一。
- 随着字典操作的不断执行，最终在某个时间点上，ht[0]的所有键值对都会被rehash至ht[1]，这时程序将rehashidx属性的值设为-1，表示rehash操作已完成。

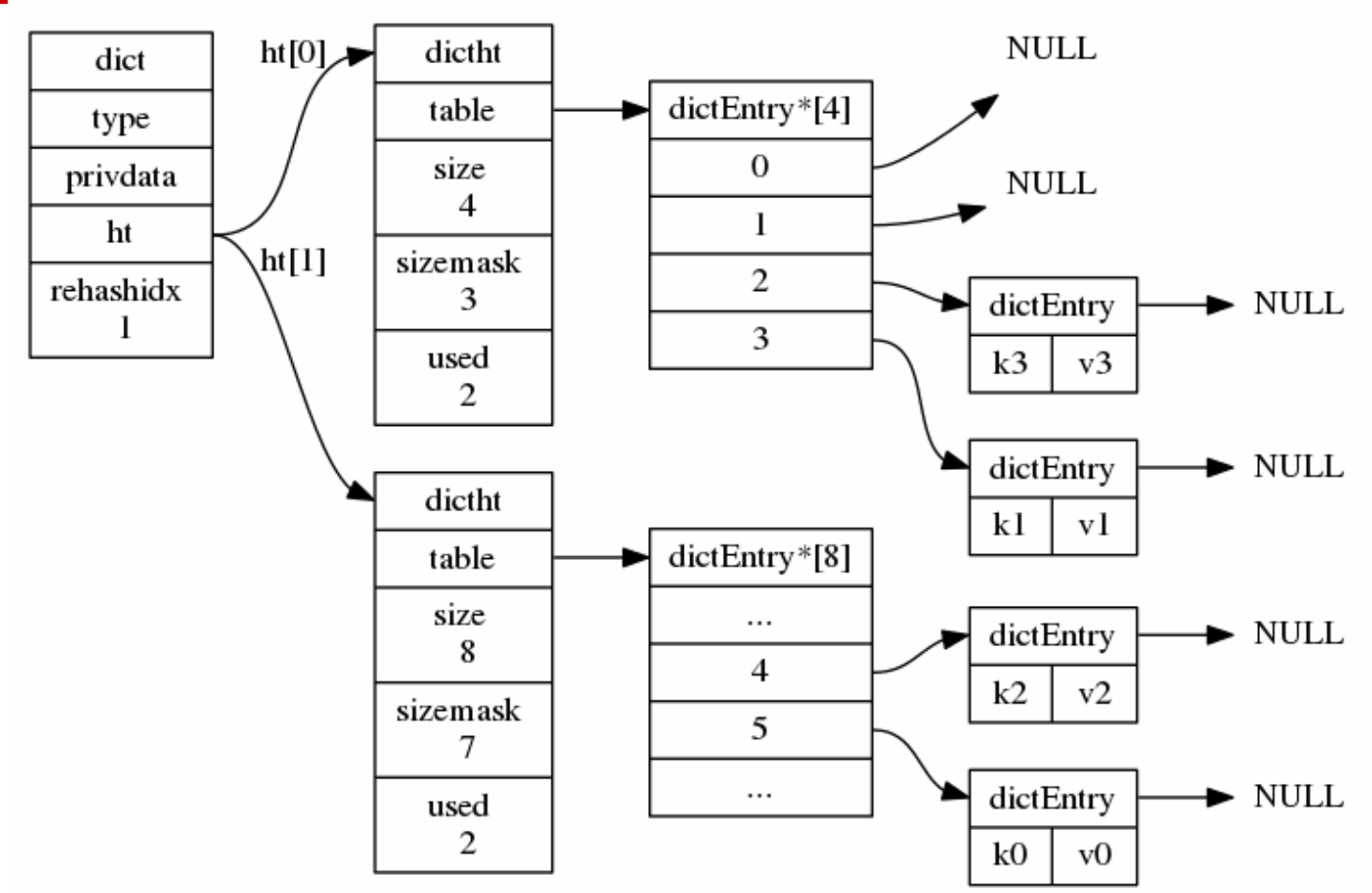
准备开始Rehash


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
4
dictEntry*[4]
0
1
2
3
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
0
dictEntry*[8]
0
1
2
...
7
NULL
NULL
NULL
NULL

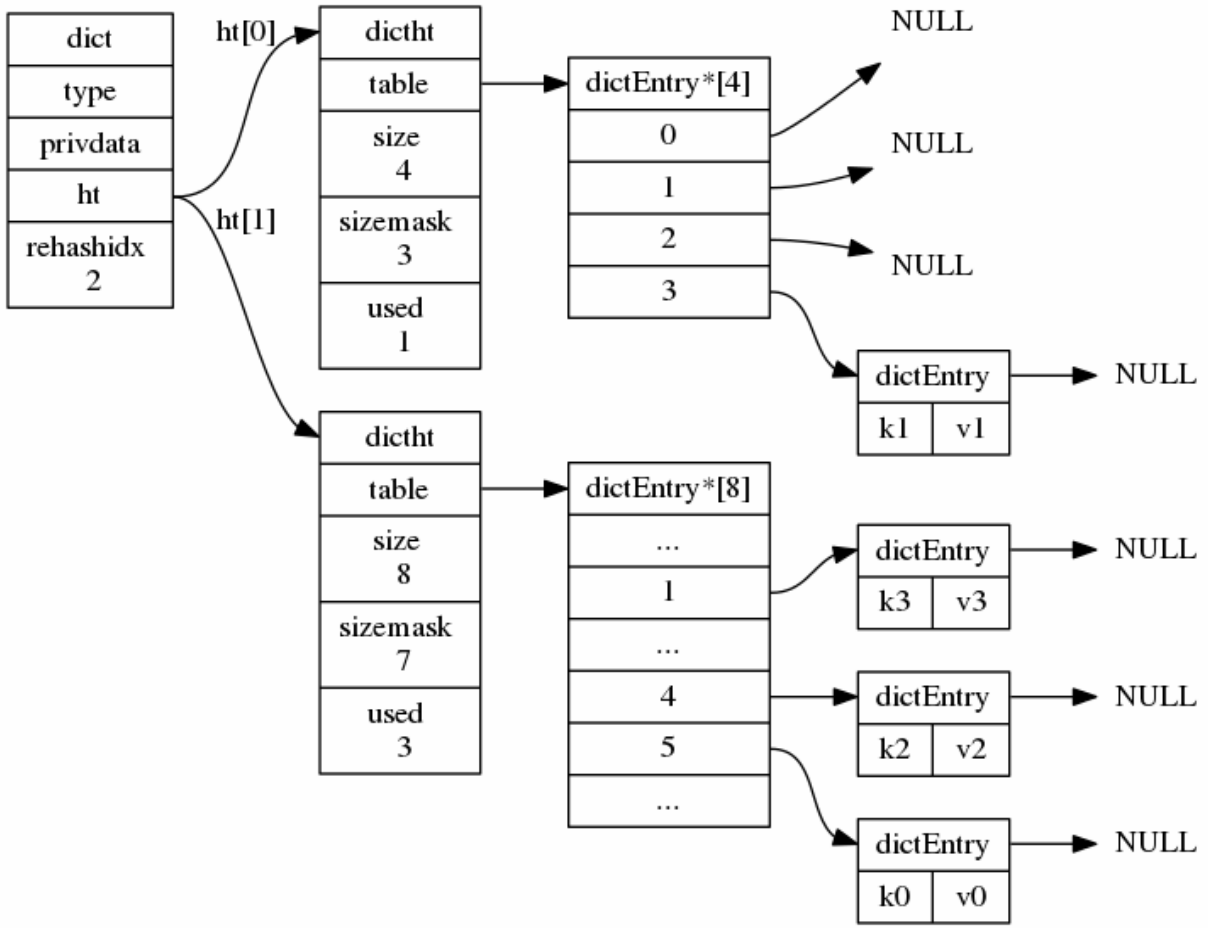
索引0上的键值对


dict
type
privdata
ht
rehashidx
0
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
3
dictEntry*[4]
0
1
2
3
NULL
dictEntry
k0
v0
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
1
dictEntry*[8]
...
4
...
dictEntry
k2
v2
NULL

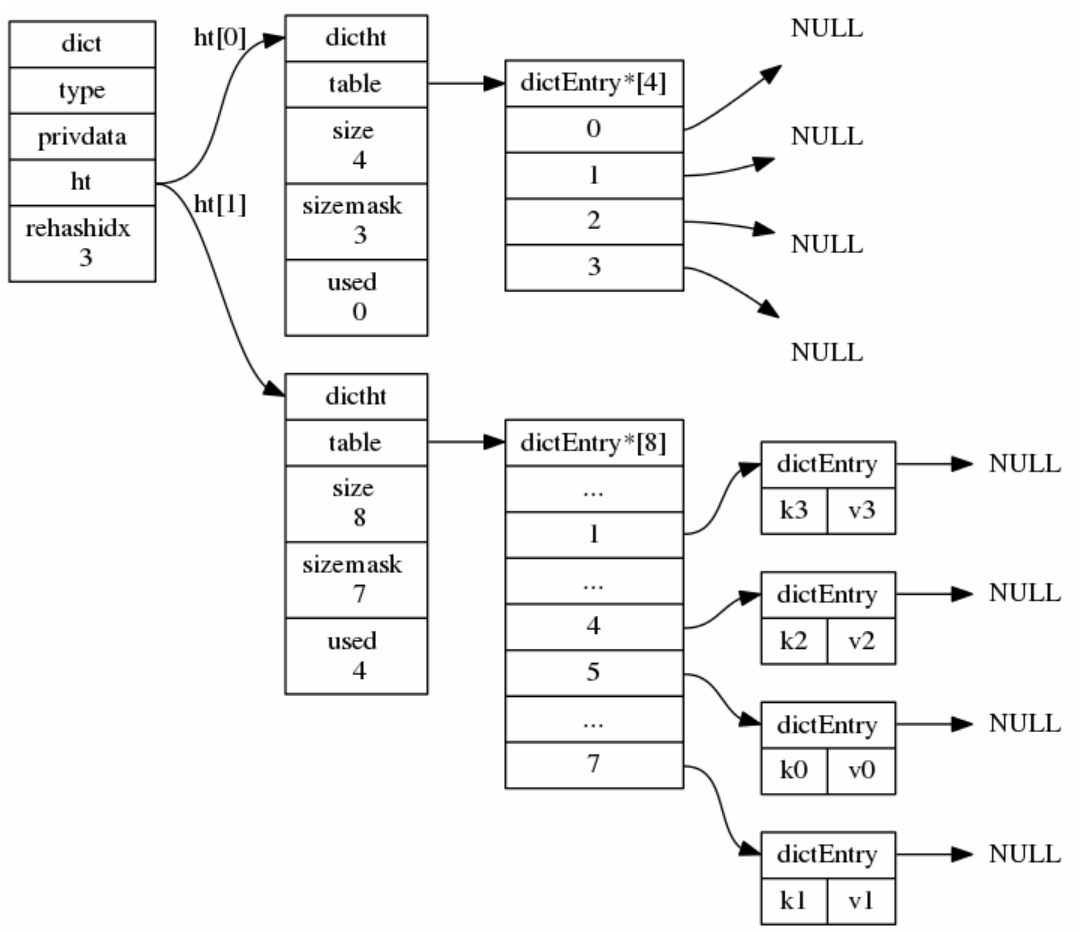
索引1上的键值对


dict
type
privdata
ht
rehashidx
1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
2
dictEntry*[4]
0
1
2
3
NULL
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
2
dictEntry*[8]
...
4
5
...
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL

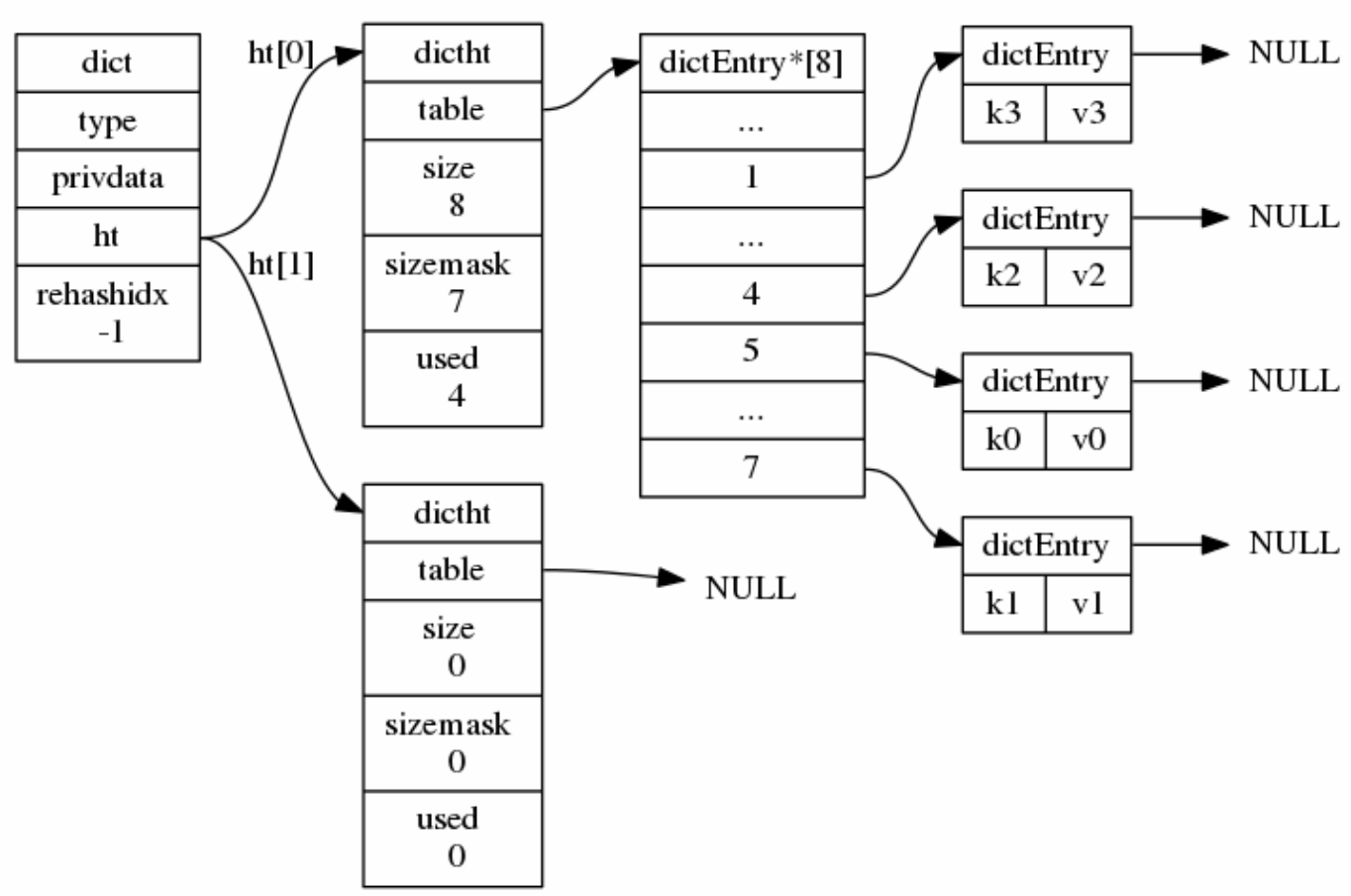
索引2上的键值对


dict
type
privdata
ht
rehashidx
2
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
1
dictEntry*[4]
0
1
2
3
NULL
NULL
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
3
dictEntry*[8]
...
1
...
4
5
...
dictEntry
k3
v3
NULL
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL

索引3上的键值对


dict
type
privdata
ht
rehashidx
3
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
0
dictEntry*[4]
0
1
2
3
NULL
NULL
NULL
NULL
dictht
table
size
8
sizemask
7
used
4
dictEntry*[8]
...
1
...
4
5
...
7
dictEntry
k3
v3
NULL
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k1
v1
NULL

Rehash执行完毕


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
8
sizemask
7
used
4
dictEntry*[8]
...
1
...
4
5
...
7
dictEntry
k3
v3
NULL
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k1
v1
NULL
dictht
table
size
0
sizemask
0
used
0
NULL

渐进式Rehash的说明

- 因为在进行渐进式rehash的过程中，字典会同时使用ht[0]和ht[1]两个哈希表，所以在渐进式rehash进行期间，字典的删除(delete)、查找(find)、更新(update)等操作会在两个哈希表上进行：比如说，要在字典里面查找一个键的话，程序会先在ht[0]里面进行查找，如果没找到的话，就会继续到ht[1]里面进行查找，诸如此类。
- 另外，在渐进式rehash执行期间，新添加到字典的键值对一律会被保存到ht[1]里面，而ht[0]则不再进行任何添加操作：这一措施保证了ht[0]包含的键值对数量会只减不增，并随着rehash操作的执行而最终变成空表。

Redis数据字典总结

- 字典被广泛用于实现Redis的各种功能，其中包括数据库和哈希键。
- Redis中的字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个用于平时使用，另一个仅在进行rehash时使用。
- 当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。
- 哈希表使用链地址法来解决键冲突，被分配到同一个索引上的多个键值对会连接成一个单向链表。
- 在对哈希表进行扩展或者收缩操作时，程序需要将现有哈希表包含的所有键值对rehash到新哈希表里面，并且这个rehash过程并不是一次性地完成的，而是渐进式地完成的。

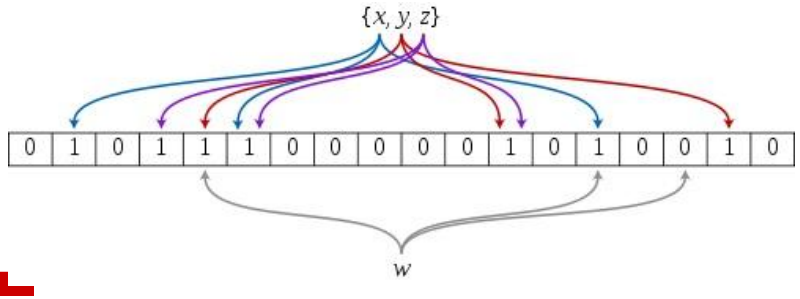
Bloom Filter

- 布隆过滤器(Bloom Filter)是由Burton Howard Bloom于1970年提出的，它是一种空间高效(space efficient)的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单、爬虫(Crawler)的网址判重等问题中经常被用到。
- 哈希表也能用于判断元素是否在集合中，但是BloomFilter只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。BloomFilter可以插入元素，但不可以删除已有元素。集合中的元素越多，误报率(false positive rate)越大，但是不会漏报(false negative)。

Bloom Filter

- 如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比对来判定是否在集合内：链表、树等数据结构都是这种思路。但是随着集合中元素数目的增加，我们需要的存储空间越来越大，检索速度也越来越慢(O(n),O(logn))。
- 可以利用Bitmap：只要检查相应点是不是1就知道可以集合中有没有某个数。这就是Bloom Filter的基本思想。

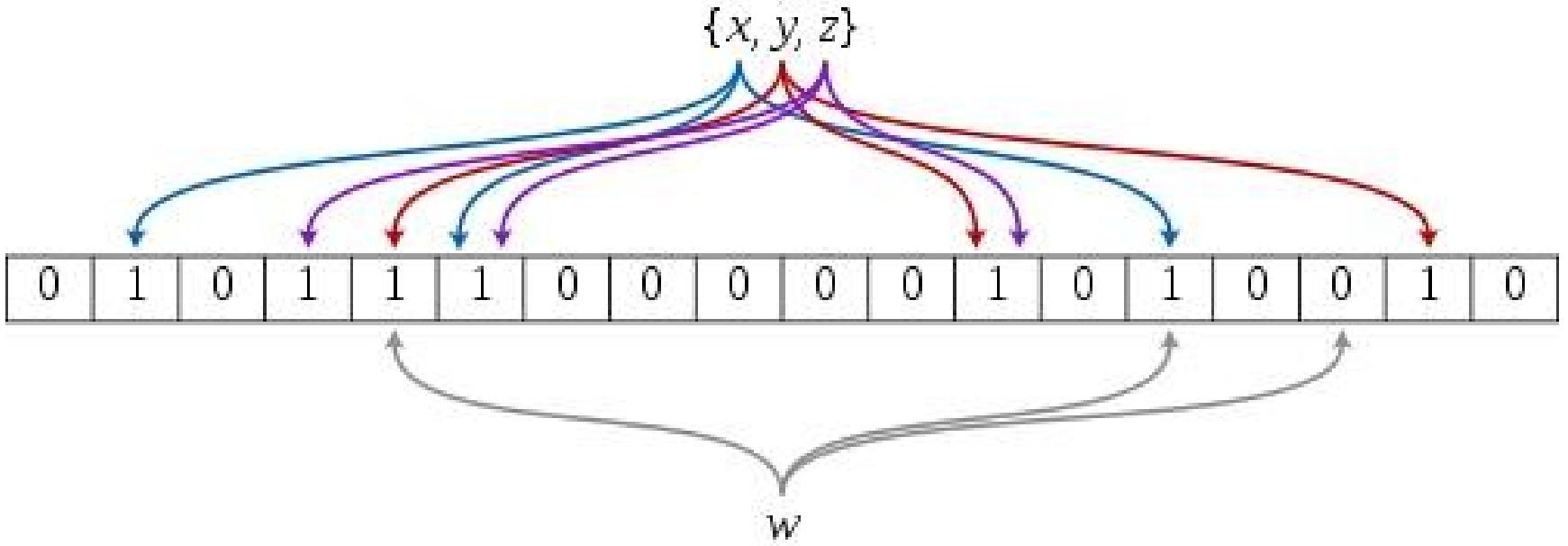
Bloom Filter算法描述



- 一个空的Bloom Filter是一个有m位的位向量B，每一个bit位都初始化为0。同时，定义k个不同的Hash函数，每个Hash函数都将元素映射到m个不同位置中的一个。
  - 记：n为元素数，m为位向量B的长度(位：槽slot)，k为Hash函数的个数。
- 增加元素x
  - 计算k个Hash(x)的值$(h_1,h_2...h_k)$，将位向量B的相应槽$B[h_1,h_2...h_k]$都设置为1；
- 查询元素x
  - 即判断x是否在集合中，计算k个Hash(x)的值$(h_1,h_2...h_k)$。若$B[h_1,h_2...h_k]$全为1，则x在集合中；若其中任一位不为1，则x不在集合中；
- 删除元素x
  - 不允许删除！因为删除会把相应的k个槽置为0，而其中很有可能有其他元素对应的位。

Bloom Filter 插入查找数据

- 插入x,y,z
- 判断w是否在该数据集中



BloomFilter的特点

- 不存在漏报：某个元素在某个集合中，肯定能报出来；
- 可能存在误报：某个元素不在某个集合中，可能也被认为存在：false positive；
- 确定某个元素是否在某个集合中的代价和总的元素数目无关
  - 查询时间复杂度：O(1)

Bloom Filter参数的确定

- 单个元素某次没有被置位为1的概率为：$1-\frac{1}{m}$
- k个Hash函数中没有一个对其置位的概率为：$\left(1-\frac{1}{m}\right)^{k}$
- 如果插入n个元素，仍未将其置位的概率为：$\left(1-\frac{1}{m}\right)^{kn}$
- 因此，此位被置位的概率为：$1-\left(1-\frac{1}{m}\right)^{kn}$

Bloom Filter参数的确定

- 查询中，若某个待查元素对应的k位都被置位，则算法会判定该元素在集合中。因此，该元素被误判的概率(上限)为：

$$q(k)=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}$$

- 考虑到：

$$\left(1-\frac{1}{m}\right)^{kn}=\left(1+\frac{1}{-m}\right)^{-m\cdot\frac{-kn}{m}}=\left(\left(1+\frac{1}{-m}\right)^{-m}\right)^{-\frac{kn}{m}}\approx e^{-\frac{kn}{m}}$$

- 从而：

$$P(k)\approx\left(1-e^{-\frac{kn}{m}}\right)^{k}$$

Bloom Filter参数的确定

- P(k)为幂指函数，取对数后求导：

$$P(k)=\left(1-e^{-\frac{kn}{m}}\right)^k \xrightarrow{令b=e^{\frac{n}{m}}} \left(1-b^{-k}\right)^k$$

$$\Rightarrow \ln P(k)=k\ln\left(1-b^{-k}\right)$$

$$\xrightarrow{取关于k的导数} \frac{1}{P(k)}P'(k)=\ln\left(1-b^{-k}\right)+k\frac{b^{-k}\ln b}{1-b^{-k}}$$

$$\ln\left(1-b^{-k}\right)+k\frac{b^{-k}\ln b}{\left(1-b^{-k}\right)}=0$$

$$\Rightarrow \left(1-b^{-k}\right)\ln\left(1-b^{-k}\right)=b^{-k}\ln b^{-k}$$

$$\Rightarrow 1-b^{-k}=b^{-k} \Rightarrow b^{-k}=\frac{1}{2} \Rightarrow e^{-\frac{kn}{m}}=\frac{1}{2}$$

$$\Rightarrow k=\ln 2\cdot\frac{m}{n}\approx 0.693\cdot\frac{m}{n}$$

$$P(k)=\left(1-\frac{1}{2}\right)^k=2^{-k}=2^{-\ln 2\frac{m}{n}}\approx 0.6185^{\frac{m}{n}}$$

参数m、k的确定

- m的计算公式：
  - 由 $$P(k)=\left(1-\frac{1}{2}\right)^{k}=2^{-k}=2^{-\ln 2\frac{m}{n}}\approx 0.6185^{\frac{m}{n}}$$
  - 得 $$P=2^{-\ln 2\frac{m}{n}}\Rightarrow \ln P=\left(\ln 2\cdot\frac{m}{n}\right)\ln 2\Rightarrow m=\frac{\ln P^{-1}}{(\ln 2)^{2}}\cdot n$$
- 此外，k的计算公式：$$k=\ln 2\cdot\frac{m}{n}=\frac{\ln P^{-1}}{\ln 2}$$
- 至此，任意先验给定可接受的错误率，即可确定参数空间m和Hash函数个数k。

Bloom Filter参数的讨论

- 1.442695041
- 若接收误差率为10^{-6}时，需要位的数目为29*n。

$$\begin{cases} m = \dfrac{\ln P^{-1}}{(\ln 2)^2} \cdot n \\ k = \ln 2 \cdot \dfrac{m}{n} = \dfrac{\ln P^{-1}}{\ln 2} \end{cases}$$

p	m/n	k
0.5	1.442695041	1
2^-2	2.885390082	2
2^-3	4.328085123	3
2^-4	5.770780164	4
2^-5	7.213475204	5
2^-6	8.656170245	6
2^-7	10.09886529	7
2^-8	11.54156033	8
2^-9	12.98425537	9
2^-10	14.42695041	10
2^-20	28.85390082	11
2^-30	43.28085123	12
2^-40	57.70780164	13
2^-50	72.13475204	14

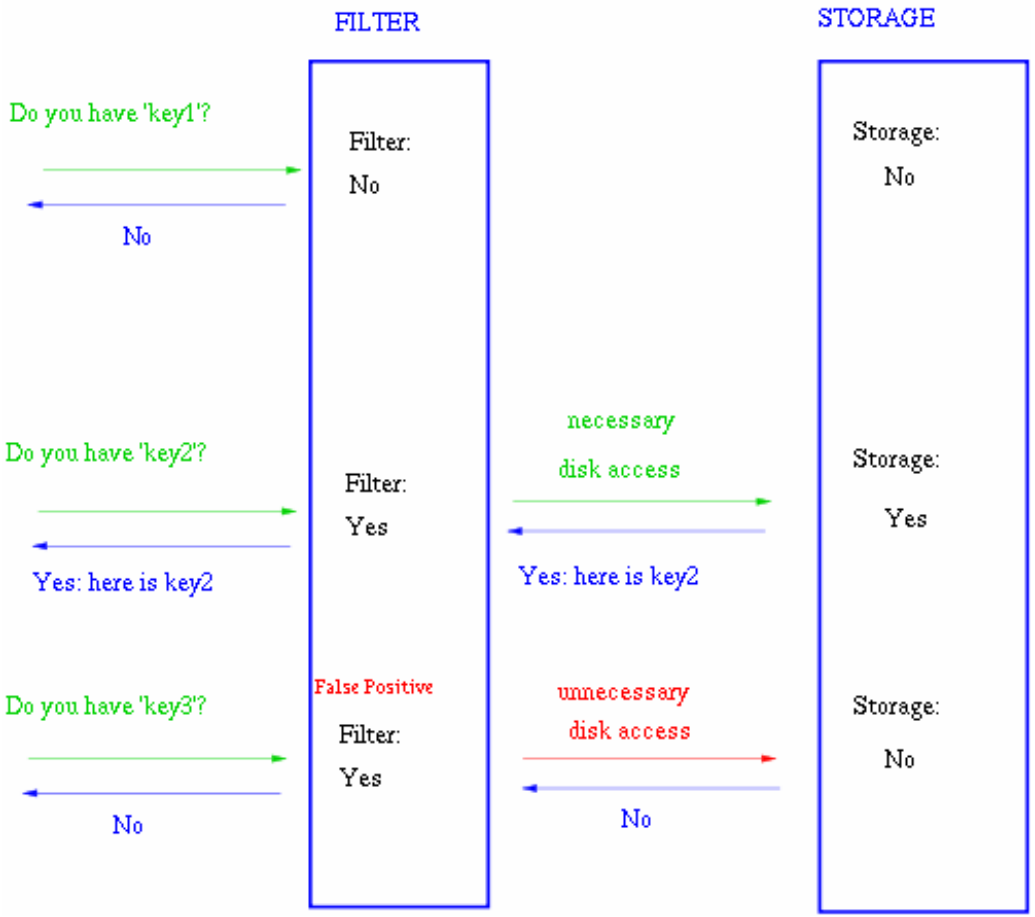
Bloom Filter的特点

- 优点：相比于其它的数据结构，Bloom Filter在空间和时间方面都有巨大的优势。Bloom Filter存储空间是线性的，插入/查询时间都是常数。另外，Hash函数相互之间没有关系，方便由硬件并行实现。Bloom Filter不存储元素本身，在某些对保密要求非常严格的场合有优势。
- 很容易想到把位向量变成整数数组，每插入一个元素相应的计数器加1，这样删除元素时将计数器减掉就可以了。然而要保证安全的删除元素并非如此简单。首先我们必须保证删除的元素的确在BloomFilter里面。这一点单凭这个过滤器是无法保证的。另外计数器下溢出也会造成问题(槽的值已经是0了，仍然执行删除操作)。

BloomFilter用例

- Google著名的分布式数据库Bigtable使用了布隆过滤器来查找不存在的行或列，以减少磁盘查找的IO次数；
- Squid网页代理缓存服务器在cachedigests中使用了BloomFilter；
- Venti文档存储系统采用BloomFilter来检测先前存储的数据；
- SPIN模型检测器使用BloomFilter在大规模验证问题时跟踪可达状态空间；
- Google Chrome浏览器使用BloomFilter加速安全浏览服务；
- 在很多Key-Value系统中也使用BloomFilter来加快查询过程，如Hbase，Accumulo，Leveldb。
  - 一般而言，Value保存在磁盘中，访问磁盘需要花费大量时间，然而使用BloomFilter可以快速判断某个Key是否存在，因此可以避免很多不必要的磁盘IO操作；另外，引入布隆过滤器会带来一定的内存消耗。

Bloom Filter + Storage结构



排序的目的

- 排序本身：得到有序的序列
- 方便查找
  - 长度为N的有序数组，查找某元素的时间复杂度是多少？
  - 长度为N的有序链表，查找某元素的时间复杂度是多少？
    - 单链表、双向链表
    - 如何解决该问题？

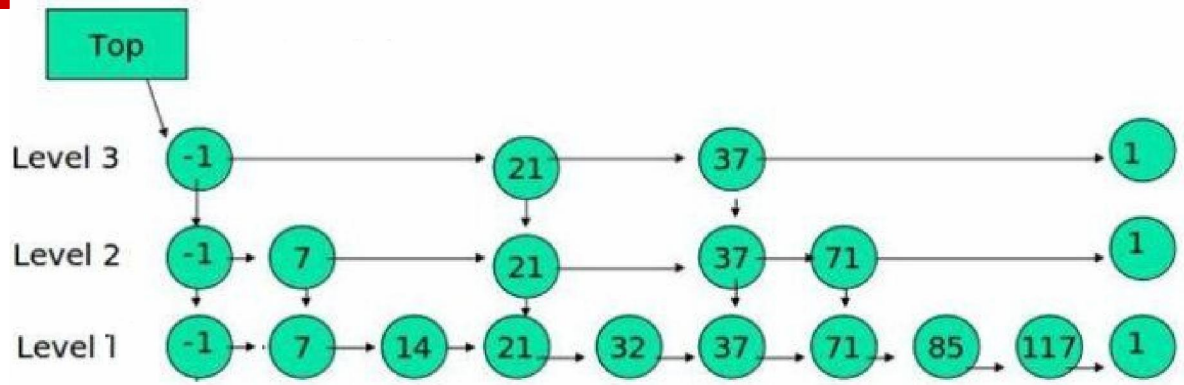
跳跃链表(Skip List)

- Treaps/RB-Tree/BTree
- 跳跃链表是一种随机化数据结构，基于并联的链表，其效率可比拟于二叉查找树(对于大多数操作需要O(logn)平均时间)。具有简单、高效、动态(Simple、Effective、Dynamic)的特点。
- 基本上，跳跃列表是对有序的链表附加辅助链表，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分结点(因此得名)。查找结点、增加结点、删除结点操作的期望时间都是logN的(with high probability≈1-1/(n^ α), W.H.P.)。

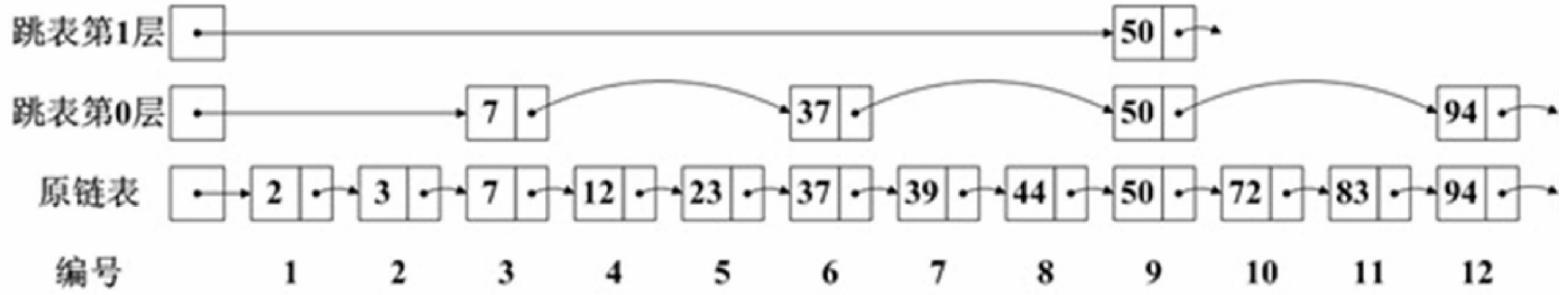
跳跃链表(Skip List)

- 跳跃列表在并行计算中也很有用，这里的插入可以在跳跃列表不同的部分并行进行，而不用全局的数据结构重新平衡。
- 跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面列表的“快速跑道”，这里在层i中的元素按某个固定的概率p出现在层i+1中。平均起来，每个元素都在1/(1-p)个列表中出现。
  - 思考：为什么是1/(1-p)?

跳跃表示例

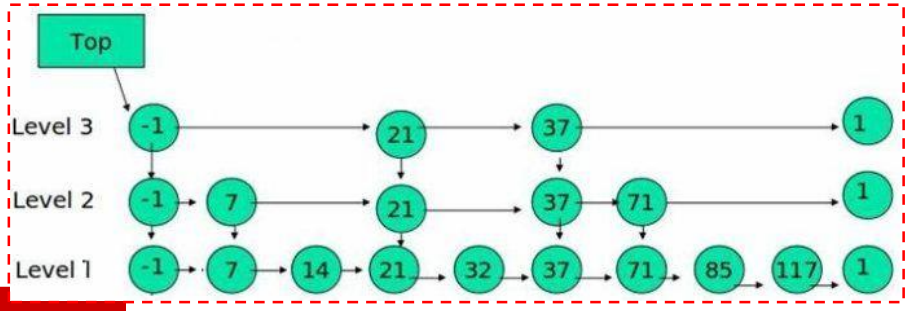


跳跃表：跳跃间隔(Skip Interval) 为 3，层次(Level)共2层



注：各个文献中对于“层”、“间隔”的定义略有差别

双层跳表时间计算

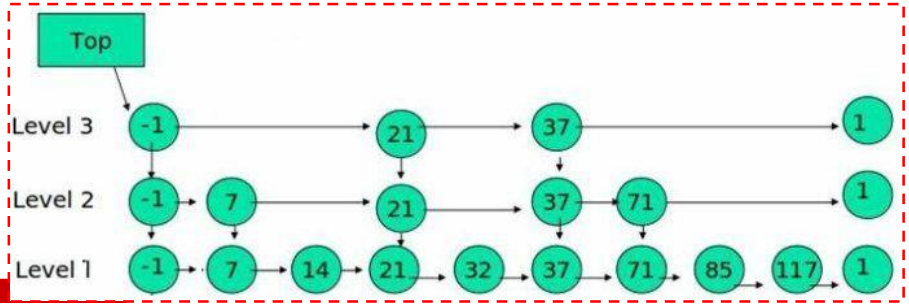


- 粗略估算查找时间：T=L1 + L2/L1(L1是稀疏层，L2是稠密层)
  - 在L2上均匀取值，构成L1；则L2/L1是L1上相邻两个元素在L2上的平均长度
  - L1：在稀疏层的最差查找次数
  - L1/L2：在稀疏层没有找到元素，跳转到稠密层需要找的次数
- 若基本链表的长度为n，即|L2|=n，|L1|为多少，T最小呢？
  - T(x)=x+n/x，对x求导，得到x=sqrt(n)
  - min(T(x))=2sqrt(n)

时间复杂度分析

- 粗略估算查找时间：T=L1 + L2/L1 + L3/L2(L1是稀疏层，L2是稠密层，L3是基本层)
  - 在L3上均匀取值，构成L2；在L2上均匀取值，构成L1；则L2/L1是L1上相邻两个元素在L2上的平均长度， L3/L2是L2上相邻两个元素在L3上的平均长度
- 若基本链表的长度为n，即|L3|=n，|L1|、|L2|为多少，T最小呢？
  - T(x,y)=x+y/x+n/y，对x,y求偏导，得到x= $\sqrt[3]{n}$
  - min(T(x,y))= $3*\sqrt[3]{n}$

跳表最优时间分析



- 建立k层的辅助链表，可以得到最小时间 $T(n)=k*\sqrt[k]{n}$
- 问题：在n已知的前提下，k取多大最好呢？
- 显然， 当k=logN时， T(n)=logN * n^(-logN)
- 问： n^(-logN)等于几？
- 一个很容易在实践中使用的结论是：
  - 当基本链表的数目为N时，共建立k=logN个辅助链表，每个上层链表的数码取下层链表的一半，则能够达到logN的时间复杂度！
- 理想跳表：ideal skip list

插入元素

- 随着底层链表的插入，某一段上的数据将不满足理想跳表的要求，需要做些调整。
  - 将底层链表这一段上元素的中位数在拷贝到上层链表中；
  - 重新计算上层链表，使得上层链表仍然是底层链表的1/2；
  - 如果上述操作过程中，上层链表不满足要求，继续上上层链表的操作。
- 新的数据应该在上层甚至上上层链表中吗？因为要找一半的数据放在上层链表(为什么是一半？)，因此：抛硬币！

插入元素后的跳表维护

- 考察待需要提升的某段结点。
- 若抛硬币得到的随机数p>0.5，则提升到上层，继续抛硬币，直到p>0.5；
  - 或者到了顶层仍然p>0.5，建立一个新的顶层

删除元素

- 在某层链表上找到了该元素，则删除；如果该层链表不是底层链表，跳转到下一层，继续本操作。

进一步说明的问题

- 若多次查找，并且相邻查找的元素有相关性(相差不大)，可使用记忆化查找进一步加快查找速度。
- 对关于k的函数$T(n)=k*\sqrt[k]{n}$求导，可计算得到k=lnN处函数取最小值，最小值是elnN。
  - 对于N=10000，k取lnN和logN，两个最小值分布是：25.0363和26.5754。
- 强调：编程方便，尤其方便将增删改查操作扩展成并行算法。
- 跳表用单链表可以实现吗？用双向链表呢？

进行10^6次随机操作后的统计结果

P	平均操作时间	平均列高	总结点数	每次查找跳跃次数（平均值）	每次插入跳跃次数（平均值）	每次删除跳跃次数（平均值）
2/3	0.0024690 ms	3.004	91233	39.878	41.604	41.566
1/2	0.0020180 ms	1.995	60683	27.807	29.947	29.072
1/e	0.0019870 ms	1.584	47570	27.332	28.238	28.452
1/4	0.0021720 ms	1.330	40478	28.726	29.472	29.664
1/8	0.0026880 ms	1.144	34420	35.147	35.821	36.007

进行10^6次随机操作后的统计结果

其他数据结构

- 并查集
- 红黑树
- 舞蹈链(Dancing Links)
- ……

我们在这里

- 更多算法面试题在 7 七月算法
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @研究者July
  - @七月算法问答
  - @邹博_机器学习

感谢大家！

恳请大家批评指正！