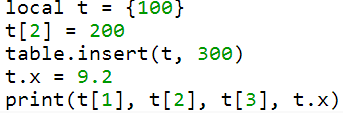
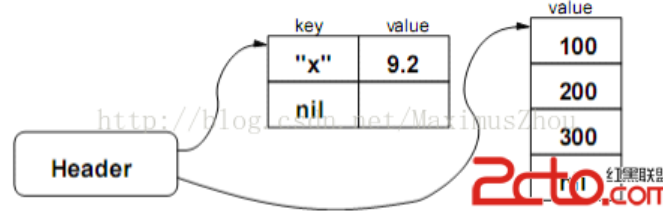
<http://www.2cto.com/kf/201504/390645.html>

在5.0以后，table是一种混合型数据结构来实现的，它包含一个哈希表部分和一个数组部分，比如下面的操作：



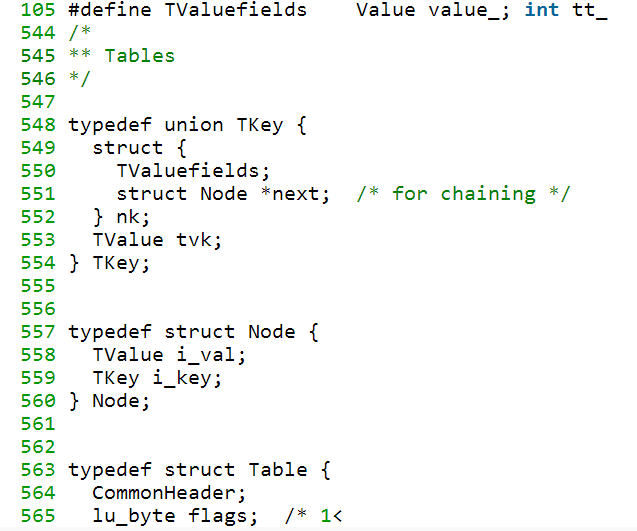
其可能的实现方式如下：



在往表中插入数值时，标会根据key-value的之和表当前的数据内容自动动态使用这两个部分：数组部分试图保存所有key值介于1和某个上限n（也就是array的最大长度，最大只能是2^26）之间的值，非整数key和超过数据范围n的整数key对应的值将存入哈希表部分。对于数组部分，要求数组的大小同时满足：1到n之间至少一半的空间被利用（避免像稀疏数组一样浪费空间）；并且n/2+1到n之间的空间至少有一个空间被另（避免n/2个空间就能容纳所有数据时申请了n个空间浪费）

对于哈希部分，解决冲突的方式是用开放寻址法，即所有的元素都存在哈希表中，使用这种方法往往可以让hash表内部数据更紧凑，有更高效的空间利用率，并且在用这个方法时还做了改进。

**源码实现：**



Table结构的头CommonHeader与TString中是一样的，用于GC，实质上所有GC类型的头是相同的，都包含这个宏。

TKey中tvk是这个key的值，nk中的next则指向key冲突的下一个节点

成员TValue \*array就是Table的数组部分，TValue表示lua数据类型的通用实现，成员int sizearray指明了这个数组的大小

成员Node \*node就是Table的哈希表部分，其大小保存在成员lu\_byte lsizenode中，注意保存的是哈希表大小的次幂，而不是实质大小。比如哈希表的大小是2^n，则lsizenode中保存的值是n，同时也说明哈希表的程度只能是2的幂次，注意2^lsizenode不等于哈希表存储变量的数目，因为hash表时有可能有冲突的，所以一个哈希表节点是一个链表的表头，它可能对应多个存储变量。Lastfree指向node里面最后一个未用的节点

结构体Node中包含两个成员i\_key和i\_val，很显然分别表示key、value，其中value的数据类型就是通用的lua数据类型TValue；key的数据类型是一个联合体，除了通常存储数据外，key还有一个作用是保存Node中的next指针，也就是说key除了保存TValue的数据结构外，还多了一个next指针，这个next指针就是用作同一个hash值下冲突时的链表指针。成员Node \*lastfree就是链表的最后一个空元素。成员struct Table \*metatable是元表的指针，每个table的元表也是一个table。Lu\_byte flags用于元表元方法的一些优化手段，一共有8位用于标记是否没有某个元方法，初始值都是有的，成员GCObject \*gclist用于GC，下面是创建一个表的接口：

Table \*luaH\_new(lua\_State \*L, int narray, int nhash){

Table\* t = luaM\_new(L, Table);

luaC\_link(L, obj2gco(t), LUA\_TTABLE);

t->metatable = NULL;

t->flags = cast\_byte(~0);

a->array = NULL;

t->sizearray = 0;

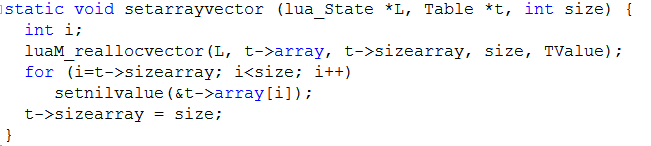
t->lsizenode = 0;

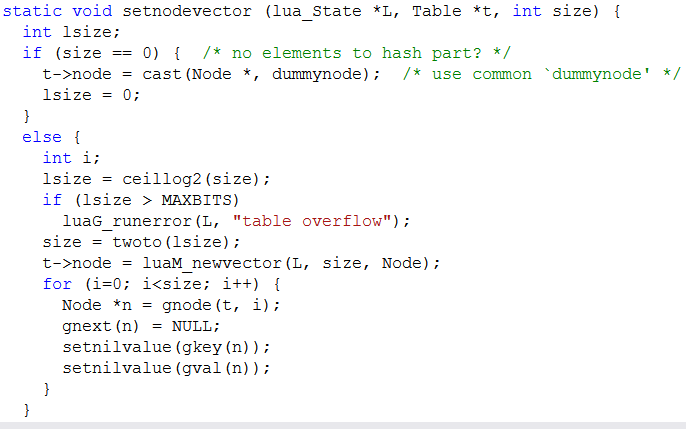
t->node = cast(Node\*, dummynode);

setarrayvector(L, t, narray);

setnodevector(L, t, nhash);

}

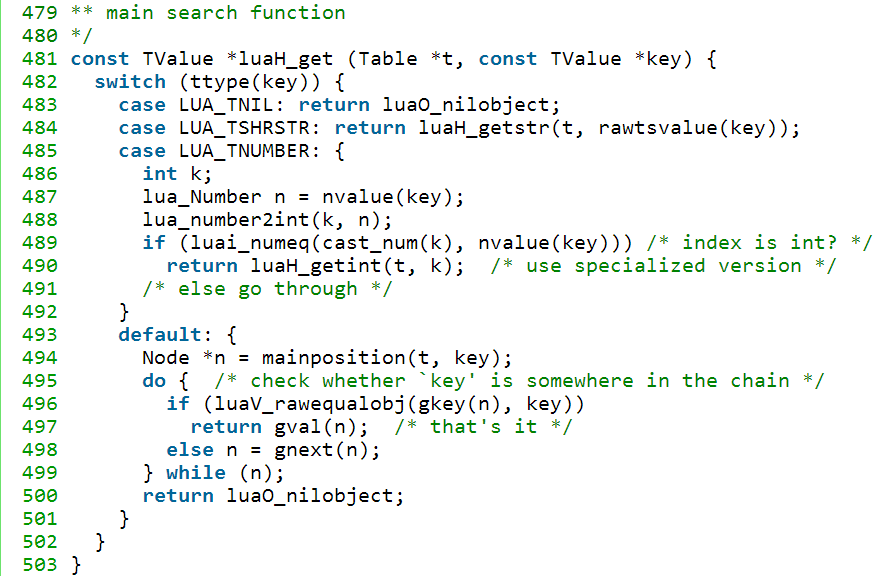




其中函数luaC\_newobj(lgc.c中定义)用来创建一个新的可回收对象，并把创建的对象放到GC链表中。Lua中所有可回收对象都是调用这个接口来创建的，方便后面GC回收，其中setnodevector用来初始化table的哈希表部分，初始值哈希表大小为1，并且node指向一个静态全局变量dummynode\_而不是NULL，这样做的目的是减少操作表时的判断操作。

一目了然，array和hash部分默认值都是0，然后用narray和nhash来初始化array和hash部分，array分布就是初始化为narray长度的数组，hash部分就是初始化为2^ceil(log(nhash))长度的哈希表，table的哈希表的长度必须是2的幂。

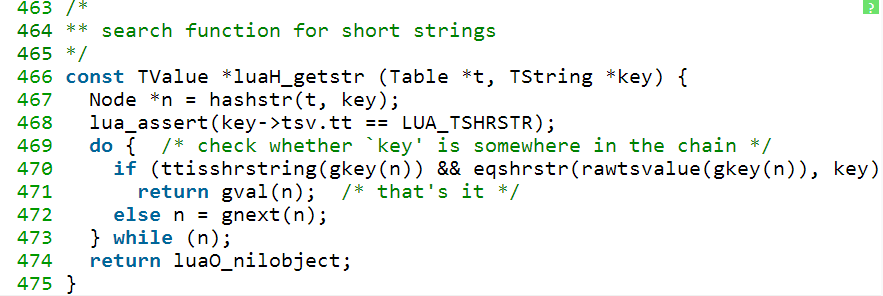
**（1）查找**



从代码中可以看出：

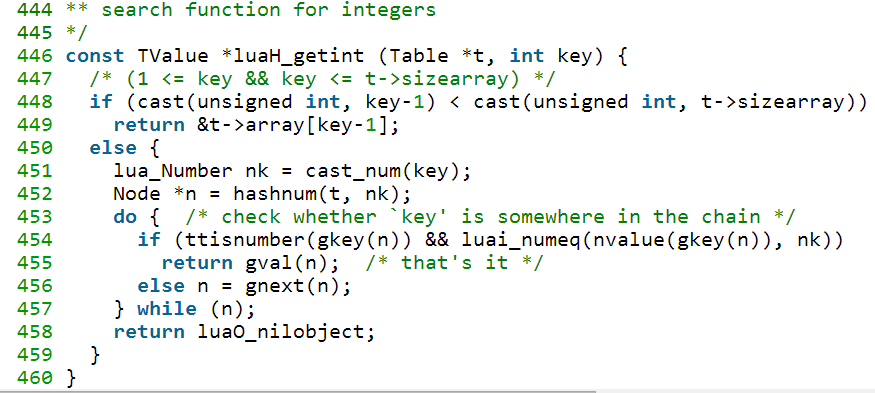
1）若key是一个nil类型，则返回nil值

2）若key是一个短字符串类型LUA\_TSTRING，则调用lua\_getstr来查找，查找代码如下：



该函数首先获得相应字符串在哈希表中的链表（使用字符串的值对哈希表取余来确定在node数组中的位置），遍历这个链表，查找字符串。主要对短字符串比较，不用逐个字符去比较，只需要比较自制，因为对整个lua虚拟机来说，短字符串只有一份，若找到了，则返回相应的值，否则返回nil

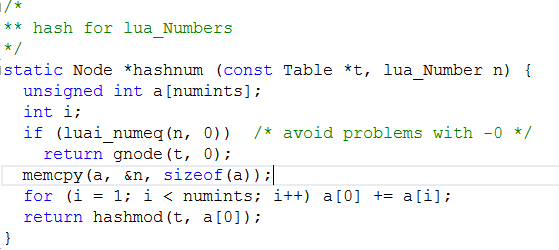
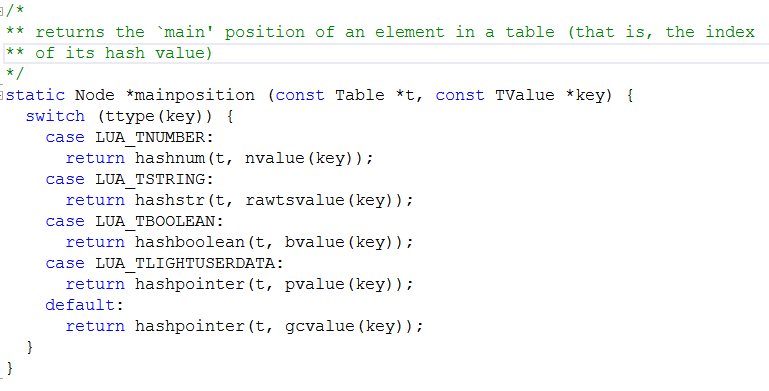
3）若key是一个柱子类型LUA\_NUMBER并且是一个int类型，则调用luaH\_getint函数去查找



如果key的值小于等于数组大小，则直接返回相应的值，否则去哈希表中查找

（4）对于其他类型，也就是不是nil、整型和短字符串类型，都是计算hash值，然后的链表中去查找（因为拥有相同哈细致的冲突键值对，在哈希表中由Node的next成员连接起来了），因此，对于长字符串来说，会逐个字符去比较，这个可以在lvm.c的luaV\_equalobj\_函数中可以看到各种数据类型比较的方法。

获取各种类型的hash值：



#define hashmod(t,n) (gnode(t, ((n) % ((sizenode(t)-1)|1))))

#define hashpointer(t,p) hashmod(t, IntPoint(p))

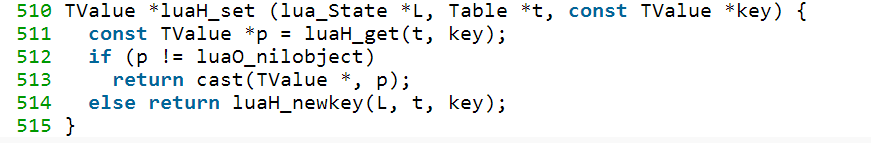
#define hashpow2(t,n) (gnode(t, lmod((n), sizenode(t))))

#define hashstr(t,str) hashpow2(t, (str)->tsv.hash)

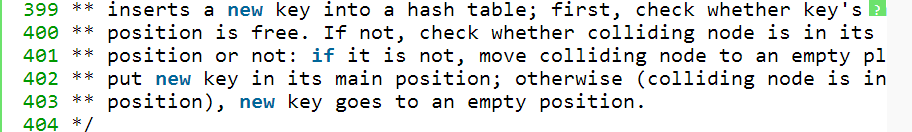
#define hashboolean(t,p) hashpow2(t, p)

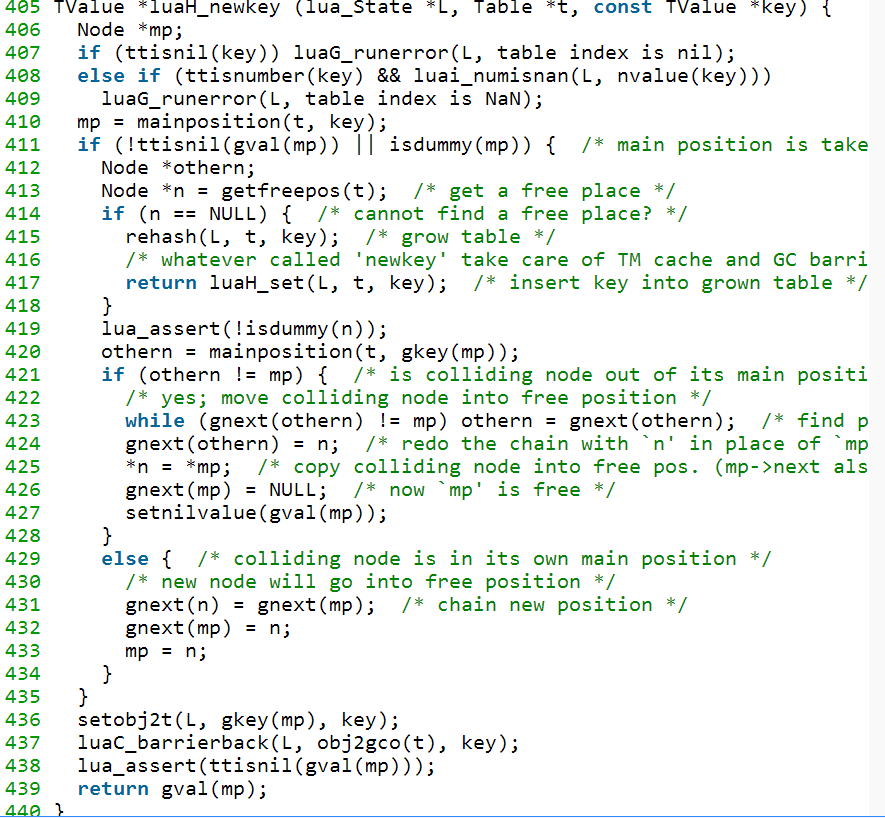
**（2）赋值**

当给table执行赋值操作时，比如t[“key”]=1，会调用函数luaH\_set：



它首先查找key是否在table中，若在，则直接替换原来的值，否则调用luaH\_newkey，插入新的(key,value)。往table中插入新的值，其基本思路是检测key的主位置是否为空，这里主位置就是key的哈希值node数组中（哈希表）的位置，若主位置为空，则直接把相应的（key，value）入到这个node中，若主位置被占了，检查占领该位置（key，value）的主位置是不是在这个地方，若不在这个地方，则移动相应该位置的（lkey，value）到一个新的空node中，并且把要插入的（key，value）插入到相应的主位置，若在这个地方（即占领该位置的（key，value）的主位置就是要插入的位置），则把要插入的（key，value）插入到一个新的空node中。





Lua中的hash表的hash算法比较特别，一般的hash表都是根据key算出hash(key),然后把这个key放在hash表的hash(key)位置上，如果有冲突的话就放在hash(key)位置的链表上。

但是在lua的hash表中，如果有冲突的话，lua会找hash表中一个空的位置（从后向前找，假设为x），然后把新的key放在这个空的位置x上，并且让hash表中hash(key)处的节点的nv.next指向x，这个意思就是，加入有冲突的话，不用重新分配内存来存储冲突的key，而是利用hash表上未用过的空格来存储，这样就会引入一个新的问题，本来key是不应该放在x上的，加入有另外一个key2，hash(key2)算出来的位置也在x的话，那就表示本来x这个位置应该是给key2的，但是由于x被key占用了，导致key2没地方放了。这时候，lua的处理方式吧key放到另外一个空格，然后让key2占用x，当hash表没有空格的时候，lua就会resize这个hash表。这样做的好处是不用动态申请内存空间，hash表初始化的时候有多少内存就用多少，不够就resize这个hash表

里面需要注意的一个函数是getfreepos

static Node\* getfreepos(Table \*t){

while(t->lastNode > t->Node){

if(ttisnil(getkey(t->lastfree)))

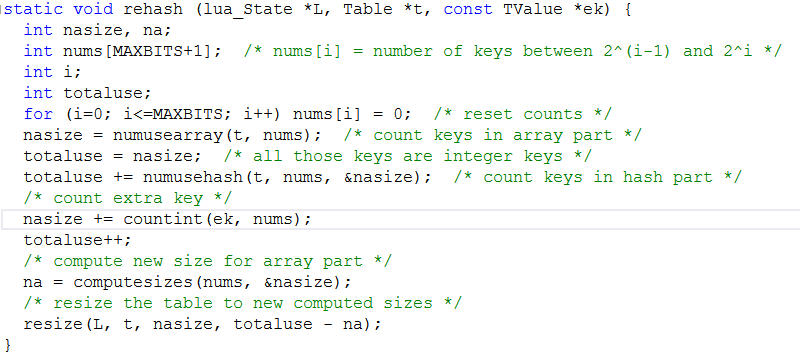
return t->lastfree;

}

return NULL;

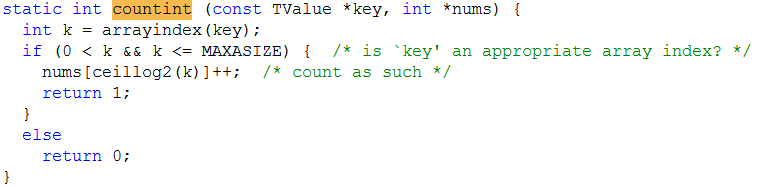
}

这个函数从后往前搜索hash表的空位置，找到的话就返回指向这个空位置的指针，lastfree在创建hash表的时候指向hash表最后一个元素，通过getfreepos可以知道hash表究竟有没有空位置，如果没有的话就要调用rehash来重新调整哈希表的大小。



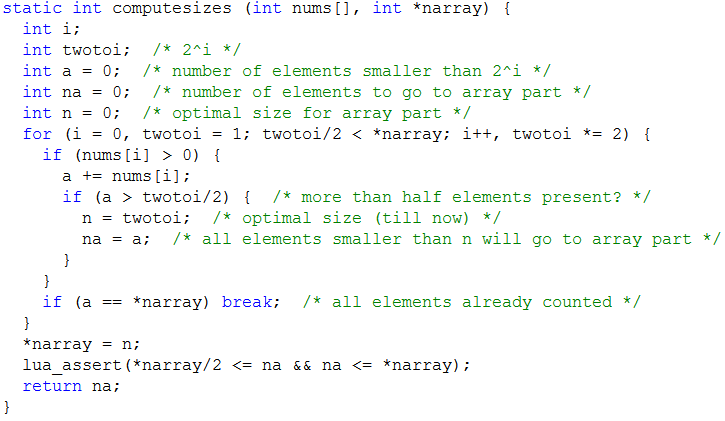
Rehash中，nums是用来统计key的数量分布的，它的定义说的非常清楚：

int nums[MAXBITS+1]; /\* nums[i] = number of keys between 2^(i-1) and 2^i \*/



计算key的值是否在1-MAXSIZE的范围之内，如果在就统计到nums中

numusearray和numuysehash的目的是统计数组和hash表部分key的使用情况，把它更新到nums里面去。然后根据nums计算要申请多大的数组部分以及多大的hash表，算法在computesizes处：



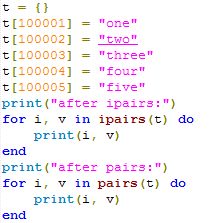
Rehash首先统计当前table中到底有value值不是nil的键值对的个数，然后根据这个数值确定table中数组部分的大小（其大小保证数组部分的控件利用率必须50%），最后调用luaH\_resize函数来重建table

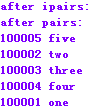
具体过程是首先调用函数numusearray计算table中数组部分非nil的数值的个数，然后调用numusehash函数计算table中哈希部分的非nil的键值对的个数。调用countint函数来确定将要插入的（key，value）是否可以放在数组中，接着调用computesizes来计算新的table数组部分的大小，最后调用luaH\_resize函数根据原来table中数据结构建新的table

Nums就是key的分布情况，narry是所有key里面，可以放在array里面的key的个数（array大小最大能到2^26，并且如果key不是整数不能放在array里），computesizes的算法简单来说就是找出一个最小的i，使其满足小于2^i的key的个数大于2^(i-1)，这个2^i就是数组部分的大小，如果找不到满足条件的i，数组部分长度为0。这样做的话就保证了数组部分包含尽可能多的元素，同时使用率在一半以上。

Computesizes算完之后就调用resize把hash表和array都重新分配一次，lua的resize和redis相比简单很多，一次过就吧元素都重新插入到新的hash表和数组里面去了。

注意：由于插入元素会导致rehash，rehash会重新调整元素是放在数组还是放在hash表中，所以一个元素无论原来是在数组还是在hash表中，rehash后都不能确定它是在数组还是hash表，这就说明了为啥对一个当作数组使用的table指定key赋值之后，ipairs遍历这个table的结果通常不靠谱。



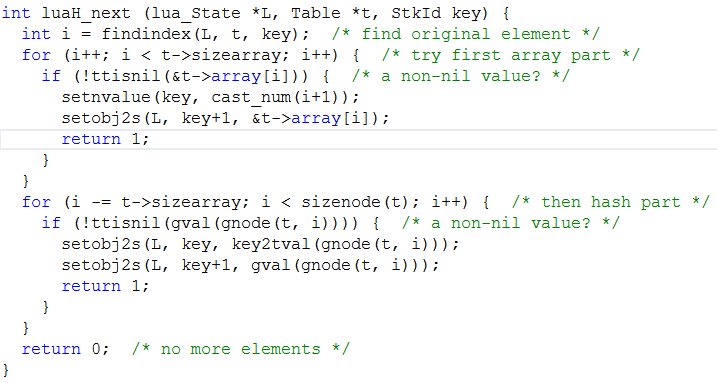


上面在插入到表中后，100001,100002，100003,100004,10000必定不能满足素组部分50%使用率的要求，所以都是放在hash表中，容纳这5个元素的hash表的大小是8

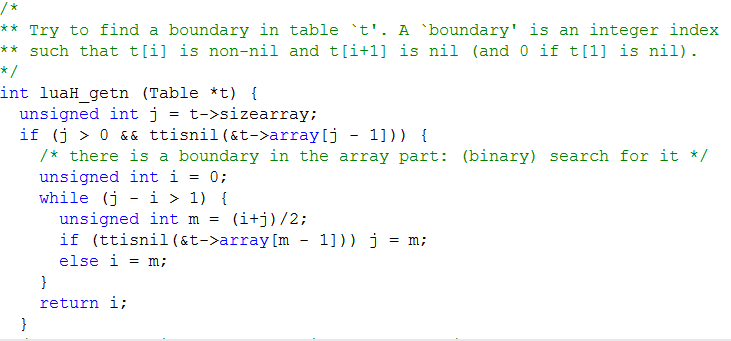
其中hashmod是计算a[0]/(哈希表大小-1)然后取t中这个位置的元素。所以100005的位置比其他元素的位置靠前，所以它会放在靠前的位置，这样在pairs的时候，会首先遍历到100005

**（3）迭代**

在lua中提供了函数next来迭代lua中键值对，即用next(t)或next(t,key)返回下一个键值对，这是在luaH\_next中来实现的



首先调用findindex获得开始检索的位置（比如，通常从等于key的位置开始查找），然后因此查找table中的数组部分和哈希部分的第一个非nil的位置



**总结：**

1. 在对table操作时，尽量不要触发rehash操作，因为这个开始时非常大的，在对table插入键值对时（也就是说key原来不在table中），可能会触发rehash操作，而直接修改已存在key对应的值，不会触发rehash操作的，包括赋值为nil
2. 在遍历一个table时，不允许向table插入一个新值，否则无法预测后续的遍历行为，但lua允许在遍历过程中修改table中已存在的键对应的值，包括修改后饿值为nil，也是允许的
3. Table中想要删除一个元素等同于向对应key赋值为key，等待垃圾回收。但是删除table一个元素的时候，并不会触发表的重构行为，即不会触发rehash操作
4. 为了减少rehash操作，当构造一个数组时，如果预先知道其大小，可以预分配数组大小，在脚本曾可以用local t = {nil, nil, nil}来预分配数组大小。在C语言层，可以使用接口void lua\_createtable(lua\_State \*L, int narr,int nrec)来预分配数组的大小