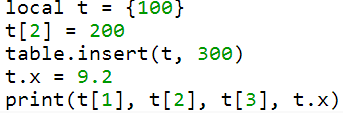
<http://blog.csdn.net/zr339361504/article/details/52432163>

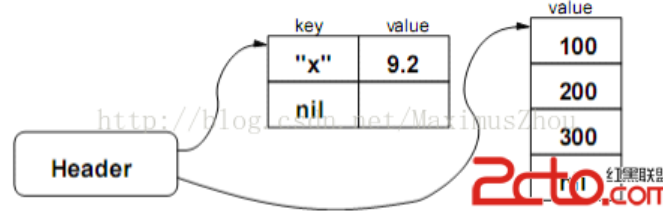
<http://www.2cto.com/kf/201504/390645.html>

lua中的rehash函数<http://blog.csdn.net/ball32109/article/details/44906403>

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，表会根据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。

typedef struct Table {

CommonHeader;

lu\_byte **flags**; /\* 1<<p means tagmethod(p) is not present \*/

lu\_byte lsizenode; /\* log2 of size of `node' array \*/ // 以2的lsizenode次方作为哈希表长度

struct Table \*metatable; // 元表

TValue \*array; /\* array part \*/ // 数组

Node \*node; // 哈希表

Node \*lastfree; /\* any free position is before this position \*/ // 指向最后一个为空置的链表空间

GCObject \*gclist;

int sizearray; /\* size of `array' array \*/ // 数组的大小

} Table;

注意这里的flags

typedef struct global\_State {

stringtable strt; /\* hash table for strings \*/

lua\_Alloc frealloc; /\* function to reallocate memory \*/

void \*ud; /\* auxiliary data to `frealloc' \*/

lu\_byte currentwhite;

lu\_byte gcstate; /\* state of garbage collector \*/

int sweepstrgc; /\* position of sweep in `strt' \*/

GCObject \*rootgc; /\* list of all collectable objects \*/

GCObject \*\*sweepgc; /\* position of sweep in `rootgc' \*/

GCObject \*gray; /\* list of gray objects \*/

GCObject \*grayagain; /\* list of objects to be traversed atomically \*/

GCObject \*weak; /\* list of weak tables (to be cleared) \*/

GCObject \*tmudata; /\* last element of list of userdata to be GC \*/

Mbuffer buff; /\* temporary buffer for string concatentation \*/

lu\_mem GCthreshold;

lu\_mem totalbytes; /\* number of bytes currently allocated \*/

lu\_mem estimate; /\* an estimate of number of bytes actually in use \*/

lu\_mem gcdept; /\* how much GC is `behind schedule' \*/

int gcpause; /\* size of pause between successive GCs \*/

int gcstepmul; /\* GC `granularity' \*/

lua\_CFunction panic; /\* to be called in unprotected errors \*/

TValue l\_registry;

struct lua\_State \*mainthread;

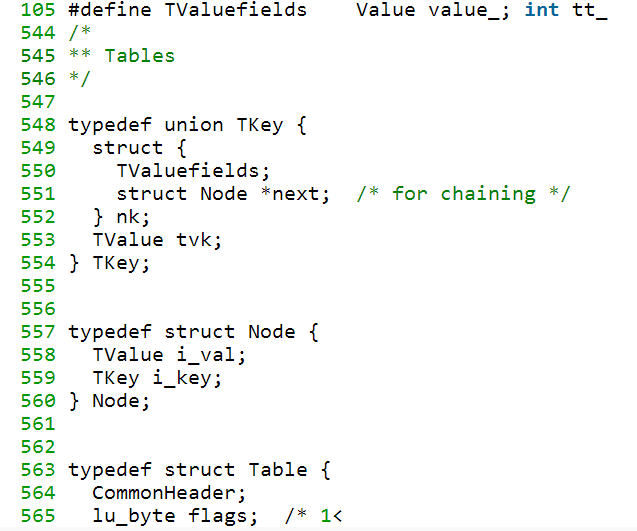
UpVal uvhead; /\* head of double-linked list of all open upvalues \*/

struct Table \*mt[NUM\_TAGS]; /\* metatables for basic types \*/

TString \*tmname[TM\_N]; /\* array with tag-method names \*/

} global\_State;

**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);

}

**（lgc.c）实际上每次新产生一个Table就前向插入到了GC链表**

void luaC\_link (lua\_State \*L, GCObject \*o, lu\_byte tt) {

global\_State \*g = G(L);

o->gch.next = **g->rootgc;**

g->rootgc = o;

o->gch.marked = luaC\_white(g);

o->gch.tt = tt;

}

#define dummynode (&dummynode\_)

static const Node dummynode\_ = {

{{NULL}, LUA\_TNIL}, /\* value \*/

{{{NULL}, LUA\_TNIL, NULL}} /\* key \*/

};

static void setarrayvector (lua\_State \*L, Table \*t, int size) {

int i;

luaM\_reallocvector(L, t->array, t->sizearray, size, TValue);

for (i=t->sizearray; i<size; i++)

setnilvalue(&t->array[i]); **// 循环将数组元素设置为nil类型**

t->sizearray = size;

}

static void setnodevector (lua\_State \*L, Table \*t, int size) {

int lsize;

if (size == 0) { /\* no elements to hash part? \*/

**t->node = cast(Node \*, dummynode); /\* use common `dummynode' \*/ // hash为空的话就指向dummynode**

lsize = 0;

}

else {

int i;

lsize = ceillog2(size);

if (lsize > MAXBITS)

luaG\_runerror(L, "table overflow");

size = twoto(lsize);

**t->node = luaM\_newvector(L, size, Node); // hash不为空则指向该hash数组**

for (i=0; i<size; i++) {

Node \*n = gnode(t, i);

gnext(n) = NULL;

setnilvalue(gkey(n));

setnilvalue(gval(n));

}

}

t->lsizenode = cast\_byte(lsize);

t->lastfree = gnode(t, size); /\* all positions are free \*/

}

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**

const TValue \*luaH\_get (Table \*t, const TValue \*key) {

switch (ttype(key)) {

case LUA\_TNIL: return luaO\_nilobject;

case LUA\_TSTRING: return luaH\_getstr(t, rawtsvalue(key));

case LUA\_TNUMBER: {

int k;

lua\_Number n = nvalue(key);

lua\_number2int(k, n); **// 浮点数转化为整数**

if (luai\_numeq(cast\_num(k), nvalue(key))) /\* index is int? \*/ **// 判断当前key是否为整数**

return luaH\_getnum(t, k); /\* use specialized version \*/

/\* else go through \*/ **// 注意：这里如果不是LUA\_TNUMBER类型，会继续到下面的default，因为这里没有break**

}

default: {

Node \*n = mainposition(t, key);

do { /\* check whether `key' is somewhere in the chain \*/

if (**luaO\_rawequalObj**(key2tval(n), key))

return gval(n); /\* that's it \*/

else n = gnext(n);

} while (n);

return luaO\_nilobject;

}

}

}

int luaO\_rawequalObj (const TValue \*t1, const TValue \*t2) {

if (ttype(t1) != ttype(t2)) return 0;

else switch (ttype(t1)) {

case LUA\_TNIL:

return 1;

case LUA\_TNUMBER:

return luai\_numeq(nvalue(t1), nvalue(t2));

case LUA\_TBOOLEAN:

return bvalue(t1) == bvalue(t2); /\* boolean true must be 1 !! \*/

case LUA\_TLIGHTUSERDATA:

return pvalue(t1) == pvalue(t2);

default:

lua\_assert(iscollectable(t1));

return gcvalue(t1) == gcvalue(t2);

}

}

#define iscollectable(o) (ttype(o) >= LUA\_TSTRING)

#define LUA\_TNONE (-1)

#define LUA\_TNIL 0

#define LUA\_TBOOLEAN 1

#define LUA\_TLIGHTUSERDATA 2

#define LUA\_TNUMBER 3

#define LUA\_TSTRING 4

#define LUA\_TTABLE 5

#define LUA\_TFUNCTION 6

#define LUA\_TUSERDATA 7

#define LUA\_TTHREAD 8

#define gcvalue(o) check\_exp(iscollectable(o), (o)->value.gc)

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是一个字符串类型LUA\_TSTRING，则调用lua\_getstr来查找，查找代码如下：

const TValue \*luaH\_getstr (Table \*t, TString \*key) {

Node \*n = **hashstr**(t, key);

do { /\* check whether `key' is somewhere in the chain \*/

if (ttisstring(gkey(n)) && rawtsvalue(gkey(n)) == key)

return gval(n); /\* that's it \*/

else n = gnext(n);

} while (n);

return luaO\_nilobject;

}

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

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

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hash值对哈希表长取余来确定在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，只需要比较地址，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是一个数字类型LUA\_NUMBER并且是一个int类型，则调用luaH\_getnum函数去查找

const TValue \*luaH\_getnum (Table \*t, int key) {

/\* (1 <= key && key <= t->sizearray) \*/

if (cast(unsigned int, key-1) < cast(unsigned int, t->sizearray)) **// 数组部分直接获取**

return &t->array[key-1];

else {

lua\_Number nk = cast\_num(key);

Node \*n = hashnum(t, nk);

do { /\* check whether `key' is somewhere in the chain \*/

if (ttisnumber(gkey(n)) && luai\_numeq(nvalue(gkey(n)), nk))

return gval(n); /\* that's it \*/

else n = gnext(n);

} while (n);

return luaO\_nilobject;

}

}

static Node \*hashnum (const Table \*t, lua\_Number n) {

unsigned int a[numints];

int i;

if (luai\_numeq(n, 0)) /\* avoid problems with -0 \*/

return gnode(t, 0);

memcpy(a, &n, sizeof(a));

for (i = 1; i < numints; i++) a[0] += a[i];

return hashmod(t, a[0]);

}

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

#define sizenode(t) (twoto((t)->lsizenode))

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（因为拥有相同哈希值的冲突键值对，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：

static Node \*mainposition (const Table \*t, const TValue \*key) {

switch (ttype(key)) {

case LUA\_TNUMBER: **// 整数在之前已经处理了，这里应该是浮点型**

return hashnum(t, nvalue(key));

case LUA\_TSTRING:

return hashstr(t, rawtsvalue(key));

case LUA\_TBOOLEAN:

return hashboolean(t, bvalue(key));

case LUA\_TLIGHTUSERDATA:

return hashpointer(t, pvalue(key));

default:

return hashpointer(t, gcvalue(key));

}

}

#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)

#define IntPoint(p) ((unsigned int)(lu\_mem)(p)) // lu\_mem实际上是size\_t

Table \*table = luaH\_new(state, 3, 3);

cout << table << endl;

cout << IntPoint(table) << endl;

 实际上0358EFE8的十进制表示就是56160232

**（2）赋值**



当给table执行赋值操作时，比如t[“key”]=1，会调用函数luaV\_settable（lvm.c），键值对赋值的过程，就是通过获取栈顶的前两个位置作为key和value，如果key在table里是不存在的则创建新的key，并返回key对应的TValue，再对指针进行赋值

LUA\_API void lua\_settable (lua\_State \*L, int idx) { （lapi.c）

StkId t;

lua\_lock(L);

api\_checknelems(L, 2);

t = index2adr(L, idx);

api\_checkvalidindex(L, t);

**luaV\_settable(L, t, L->top - 2, L->top - 1);**

L->top -= 2; /\* pop index and value \*/

lua\_unlock(L);

}

LUA\_API void lua\_setfield (lua\_State \*L, int idx, const char \*k) { （lapi.c）

StkId t;

TValue key;

lua\_lock(L);

api\_checknelems(L, 1);

t = index2adr(L, idx);

api\_checkvalidindex(L, t);

setsvalue(L, &key, luaS\_new(L, k));

**luaV\_settable(L, t, &key, L->top - 1);**

L->top--; /\* pop value \*/

lua\_unlock(L);

}

void luaV\_settable (lua\_State \*L, const TValue \*t, TValue \*key, StkId val) {

int loop;

for (loop = 0; loop < MAXTAGLOOP; loop++) { **// 这里循环100次，是因为要遍历所有的元表有无对应的key**

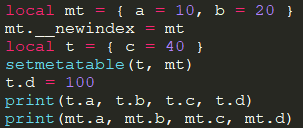
const TValue \*tm;

if (ttistable(t)) { /\* `t' is a table? \*/

Table \*h = hvalue(t);

TValue \*oldval = **luaH\_set(L, h, key);** /\* do a primitive set \*/ **// 判断这个key是否存在，不存在则创建一个**

**// 找到了已有的节点（也就是说节点不为空，因为，如果找到的是空节点，那么它有可能存在于元表中，继续查找），或者发现它的元表为空，则可以直接退出，实际上如果有元表，并且元表有\_\_newindex，最后如果没有找到的话，会把新的节点放到最底层的元表中，下面用一个例子证明：**



if (!ttisnil(oldval) || /\* result is no nil? \*/

(tm = **fasttm**(L, h->metatable, TM\_NEWINDEX)) == NULL) { /\* or no TM? \*/

setobj2t(L, oldval, val);

luaC\_barriert(L, h, val);

return;

}

/\* else will try the tag method \*/

}

else if (ttisnil(tm = luaT\_gettmbyobj(L, t, TM\_NEWINDEX)))

luaG\_typeerror(L, t, "index");

if (ttisfunction(tm)) { **// 如果元表是函数，则直接调用函数**

callTM(L, tm, t, key, val);

return;

}

t = tm; /\* else repeat with `tm' \*/

}

luaG\_runerror(L, "loop in settable");

}

#define fasttm(l,et,e) gfasttm(G(l), et, e)

这里会发现用到了Table的flags

// 是否有元表，如果没有元表则返回NULL，如果有则查看元表中的元方法((et)->flags & (1u<<(e)))，如果没有元方法的对应的flags（默认为~0），对应位置为1表示没有出现，对应位置为0则表示出现了，则返回NULL，否则执行luaT\_gettm

#define gfasttm(g,et,e) ((et) == NULL ? NULL : \

((et)->flags & (1u<<(e))) ? NULL : luaT\_gettm(et, e, (g)->tmname[e]))

**注意：tmname在global\_State中**

typedef enum {

TM\_INDEX,

TM\_NEWINDEX,

TM\_GC,

TM\_MODE,

TM\_EQ, /\* last tag method with `fast' access \*/

TM\_ADD,

TM\_SUB,

TM\_MUL,

TM\_DIV,

TM\_MOD,

TM\_POW,

TM\_UNM,

TM\_LEN,

TM\_LT,

TM\_LE,

TM\_CONCAT,

TM\_CALL,

TM\_N /\* number of elements in the enum \*/

} TMS;

\*\* function to be used with macro "fasttm": optimized for absence of

\*\* tag methods

Events表示元表，TMS表示元表类型（Index、newIndex等），ename元表名称

const TValue \*luaT\_gettm (Table \*events, TMS event, TString \*ename) {

const TValue \*tm = luaH\_getstr(events, ename);**// 通过名字（\_\_index、\_\_newindex等等）找到元表中的TValue**

lua\_assert(event <= TM\_EQ);

if (ttisnil(tm)) { /\* no tag method? \*/ **//对应元方式的值为nil，也就是说设置了类似的结构\_\_newindex = nil**

events->flags |= cast\_byte(1u<<event); /\* cache this fact \*/ **// 当前位置标志位1，表示没有该元方法**

return NULL;

}

else return tm; **// 返回的实际上是元表中的元方法对应的TValue，可能为function，也可能为table**

}

**lua\_setmetatable（luaapi.c）**

LUA\_API int lua\_setmetatable (lua\_State \*L, int objindex) {

TValue \*obj;

Table \*mt;

lua\_lock(L);

api\_checknelems(L, 1);

obj = index2adr(L, objindex);

api\_checkvalidindex(L, obj);

if (ttisnil(L->top - 1))

mt = NULL;

else {

api\_check(L, ttistable(L->top - 1));

mt = hvalue(L->top - 1);

}

switch (ttype(obj)) {

case LUA\_TTABLE: {

hvalue(obj)->metatable = mt;

if (mt)

luaC\_objbarriert(L, hvalue(obj), mt);

break;

}

case LUA\_TUSERDATA: {

uvalue(obj)->metatable = mt;

if (mt)

luaC\_objbarrier(L, rawuvalue(obj), mt);

break;

}

default: {

G(L)->mt[ttype(obj)] = mt;

break;

}

}

L->top--;

lua\_unlock(L);

return 1;

}

TValue \*luaH\_set (lua\_State \*L, Table \*t, const TValue \*key) {

const TValue \*p = luaH\_get(t, key);

t->flags = 0;

if (p != luaO\_nilobject) **// 当前table中存在key值，返回value值，实际上如果是整数且在数组部分，肯定是可以找到的，因为分配了空间，所以p!=luaO\_nilobject肯定成立**

return cast(TValue \*, p);

else {

if (ttisnil(key)) luaG\_runerror(L, "table index is nil");

else if (ttisnumber(key) && luai\_numisnan(nvalue(key)))

luaG\_runerror(L, "table index is NaN");

return newkey(L, t, key); **// 不存在则直接新建一个key值**

}

}

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。

/\*

\*\* inserts a new key into a hash table; first, check whether key's main

\*\* position is free. If not, check whether colliding node is in its main

\*\* position or not: if it is not, move colliding node to an empty place and

\*\* put new key in its main position; otherwise (colliding node is in its main

\*\* position), new key goes to an empty position.

\*/

static TValue \*newkey (lua\_State \*L, Table \*t, const TValue \*key) {

Node \*mp = mainposition(t, key); **// 找到当前key值在table中对应的hash节点**

if (!ttisnil(gval(mp)) || mp == dummynode) {

Node \*othern;

Node \*n = getfreepos(t); /\* get a free place \*/

if (n == NULL) { /\* cannot find a free place? \*/

rehash(L, t, key); /\* grow table \*/

return luaH\_set(L, t, key); /\* re-insert key into grown table \*/

}

lua\_assert(n != dummynode);

othern = mainposition(t, key2tval(mp));

if (othern != mp) { /\* is colliding node out of its main position? \*/ // **这里说明之前真实的位置othern肯定被占用了，而且othern链表中肯定有一个节点是指向mp的**

/\* yes; move colliding node into free position \*/

while (gnext(othern) != mp) othern = gnext(othern); /\* find previous \*/ **// 在mp的前驱节点之后插入**

gnext(othern) = n; /\* redo the chain with `n' in place of `mp' \*/

\*n = \*mp; /\* copy colliding node into free pos. (mp->next also goes) \*/

gnext(mp) = NULL; /\* now `mp' is free \*/ **// 返回一个空节点**

setnilvalue(gval(mp));

}

else { /\* colliding node is in its own main position \*/ **// 如果冲突了，则将n插入到mp的后面**

/\* new node will go into free position \*/

gnext(n) = gnext(mp); /\* chain new position \*/

gnext(mp) = n;

mp = n;

}

}

gkey(mp)->value = key->value; gkey(mp)->tt = key->tt;

luaC\_barriert(L, t, key);

lua\_assert(ttisnil(gval(mp)));

return gval(mp);

}

对于上面mp!=othern的情况，举个例子，十个元素的组数，依次插入5，15，25，35，9

在插入5，15，25，35之后的情况：



然后插入9，发现被占用了，且25的的位置不在该位置，此时mp指向9，othern指向5

通过while (gnext(othern) != mp) othern = gnext(othern);使得othern指向8

通过gnext(othern) = n，此时n（最后一个空闲节点）指向7，使得8指向7

通过\*n = \*mp，此时将9的数据移动到7的位置

通过gnext(mp) = NULL，将9的位置空出来，返回一个空节点

static Node \*mainposition (const Table \*t, const TValue \*key) {

switch (ttype(key)) {

case LUA\_TNUMBER:

return hashnum(t, nvalue(key));

case LUA\_TSTRING:

return hashstr(t, rawtsvalue(key));

case LUA\_TBOOLEAN:

return hashboolean(t, bvalue(key));

case LUA\_TLIGHTUSERDATA:

return hashpointer(t, pvalue(key));

default:

return hashpointer(t, gcvalue(key));

}

}

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->lastfree-- > t->node) {

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

return t->lastfree;

}

return NULL; /\* could not find a free place \*/

}

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。

static void rehash (lua\_State \*L, Table \*t, const TValue \*ek) {

int nasize, na;

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

int i;

int totaluse; **// 整个table中的非空值数量**

for (i=0; i<=MAXBITS; i++) nums[i] = 0; /\* reset counts \*/

nasize = **numusearray**(t, nums); /\* count keys in array part \*/ **// nasize实际上保存了所有的key为整数的个数**

totaluse = nasize; /\* all those keys are integer keys \*/

totaluse += **numusehash**(t, nums, &nasize); /\* count keys in hash part \*/

/\* count extra key \*/

nasize += countint(ek, nums); **// 处理新建的key值**

totaluse++;

/\* compute new size for array part \*/

na = **computesizes**(nums, &nasize); **// nasize会在computesizes函数中改变为新的数组容量，2的幂次方，返回为新数组容量中所有已存在的key为整数的个数**

/\* resize the table to new computed sizes \*/

**resize**(L, t, nasize, totaluse - na);

}

static int numusearray (const Table \*t, int \*nums) {

int lg;

int ttlg; /\* 2^lg \*/

int ause = 0; /\* summation of `nums' \*/

int i = 1; /\* count to traverse all array keys \*/

for (lg=0, ttlg=1; lg<=MAXBITS; lg++, ttlg\*=2) { /\* for each slice \*/

int lc = 0; /\* counter \*/

int lim = ttlg;

if (lim > t->sizearray) {

lim = t->sizearray; /\* adjust upper limit \*/

if (i > lim)

break; /\* no more elements to count \*/

}

/\* count elements in range (2^(lg-1), 2^lg] \*/

for (; i <= lim; i++) {

if (!ttisnil(&t->array[i-1]))

lc++;

}

nums[lg] += lc;

ause += lc;

}

return ause;

}

统计2^(n-1) -> 2^n 数组中非空的元素

Num[0] 0

Num[1] 1

Num[2] 2, 3

Num[3] 4, 5, 6, 7

static int numusehash (const Table \*t, int \*nums, int \*pnasize) {

int totaluse = 0; /\* total number of elements \*/

int ause = 0; /\* summation of `nums' \*/

int i = sizenode(t);

while (i--) {

Node \*n = &t->node[i];

if (!ttisnil(gval(n))) {

ause += **countint**(key2tval(n), nums);

totaluse++;

}

}

\*pnasize += ause;

return totaluse;

}

**#define sizenode(t) (twoto((t)->lsizenode))**

**实际上lsizenode表示hash长度的log2**

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 \*/

static int countint (const TValue \*key, int \*nums) {

int k = **arrayindex**(key); **// 获取key的整数索引**

if (0 < k && k <= MAXASIZE) { /\* is `key' an appropriate array index? \*/

nums[ceillog2(k)]++; /\* count as such \*/ **// 将合适的整数key计数都num中**

return 1;

}

else

return 0;

}

static int arrayindex (const TValue \*key) {

if (ttisnumber(key)) {

lua\_Number n = nvalue(key);

int k;

lua\_number2int(k, n);

if (luai\_numeq(cast\_num(k), n))

return k;

}

return -1; /\* `key' did not match some condition \*/

}

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：

static int computesizes (int nums[], int \*narray) {

int i;

int twotoi; /\* 2^i \*/

int a = 0; /\* number of elements smaller than 2^i \*/

int na = 0; /\* number of elements to go to array part \*/

int n = 0; /\* optimal size for array part \*/

for (i = 0, twotoi = 1; twotoi/2 < \*narray; i++, twotoi \*= 2) { **// narray是当前实际上key为整数的个数，要保证在数组中大于50%的占有率，遍历的上限也就是2\*narray，因此只需要twotoi/2 < \*narray即可**

if (nums[i] > 0) {

a += nums[i];

if (a > twotoi/2) { /\* more than half elements present? \*/ **// 只要发现满足超过一半占有率就更新**

n = twotoi; /\* optimal size (till now) \*/ **// n表示当前数组应该设置的容量**

na = a; /\* all elements smaller than n will go to array part \*/ **// na表示到<当前容量数组的整数个数**

}

}

if (a == \*narray) break; /\* all elements already counted \*/

}

\*narray = n;

lua\_assert(\*narray/2 <= na && na <= \*narray);

return na;

}

**// resize函数中nasize为新数组的容量，nhsize为放在hash散列表中实际元素的个数**

static void resize (lua\_State \*L, Table \*t, int nasize, int nhsize) {

int i;

int oldasize = t->sizearray;

int oldhsize = t->lsizenode;

Node \*nold = t->node; /\* save old hash ... \*/

if (nasize > oldasize) /\* array part must grow? \*/

setarrayvector(L, t, nasize);

/\* create new hash part with appropriate size \*/

setnodevector(L, t, nhsize);

if (nasize < oldasize) { /\* array part must shrink? \*/

t->sizearray = nasize;

/\* re-insert elements from vanishing slice \*/

for (i=nasize; i<oldasize; i++) {

if (!ttisnil(&t->array[i]))

setobjt2t(L, luaH\_setnum(L, t, i+1), &t->array[i]);

}

/\* shrink array \*/

luaM\_reallocvector(L, t->array, oldasize, nasize, TValue);

}

/\* re-insert elements from hash part \*/

for (i = twoto(oldhsize) - 1; i >= 0; i--) {

Node \*old = nold+i;

if (!ttisnil(gval(old)))

setobjt2t(L, luaH\_set(L, t, key2tval(old)), gval(old));

}

if (nold != dummynode)

luaM\_freearray(L, nold, twoto(oldhsize), Node); /\* free old array \*/

}

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空间利用率必须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，一次就把元素都重新插入到新的hash表和数组里面去了。

此函数是根据之前nums区域的数据去求知rehash之后的数组大小。从前往后把nums的区域数据相加，看看每个2^n之中的有多少个key为number形的。

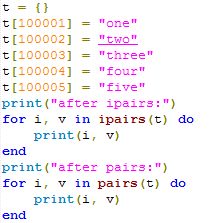
如果数量大于2^n /2= 2^n-1的话，则把rehash后的数组长度确定起来，就是n,然后数

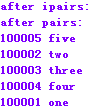
组长度为2^n,然后能进入到数组部分的数量为na.如果a == \*narry表示已经统计完。上面

lua伪代码有说,把所有nums的数量加起来就是narray,也就是之前所说的nasize.，

对于hash部分，也就是看剩余的元素（不能放在数组中的）取一个2的最小次幂

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必定不能满足数组部分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

int luaH\_next (lua\_State \*L, Table \*t, StkId key) {

int i = **findindex**(L, t, key); /\* find original element \*/

for (i++; i < t->sizearray; i++) { /\* try first array part \*/

if (!ttisnil(&t->array[i])) { /\* a non-nil value? \*/

setnvalue(key, cast\_num(i+1));

setobj2s(L, key+1, &t->array[i]);

return 1;

}

}

for (i -= t->sizearray; i < sizenode(t); i++) { /\* then hash part \*/

if (!ttisnil(gval(gnode(t, i)))) { /\* a non-nil value? \*/

setobj2s(L, key, key2tval(gnode(t, i)));

setobj2s(L, key+1, gval(gnode(t, i)));

return 1;

}

}

return 0; /\* no more elements \*/

}

static int findindex (lua\_State \*L, Table \*t, StkId key) {

int i;

if (ttisnil(key)) return -1; /\* first iteration \*/

i = arrayindex(key);

if (0 < i && i <= t->sizearray) /\* is `key' inside array part? \*/

return i-1; /\* yes; that's the index (corrected to C) \*/

else {

Node \*n = mainposition(t, key);

do { /\* check whether `key' is somewhere in the chain \*/

/\* key may be dead already, but it is ok to use it in `next' \*/

if (luaO\_rawequalObj(key2tval(n), key) ||

(ttype(gkey(n)) == LUA\_TDEADKEY && iscollectable(key) &&

gcvalue(gkey(n)) == gcvalue(key))) {

i = cast\_int(n - gnode(t, 0)); /\* key index in hash table \*/

/\* hash elements are numbered after array ones \*/

return i + t->sizearray;

}

else n = gnext(n);

} while (n);

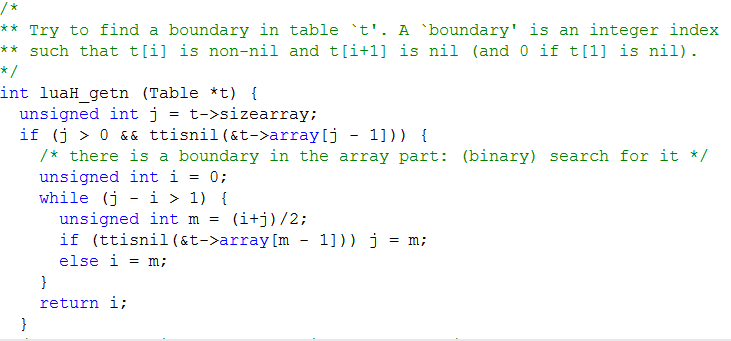
luaG\_runerror(L, "invalid key to " LUA\_QL("next")); /\* key not found \*/

return 0; /\* to avoid warnings \*/

}

}

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，在脚本中可以用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