**Lua53对象模型与GC**

2019.4 孟雷，张兵

目录

[**Lua53对象模型与GC** 1](#_Toc41930920)

[*2019.4 孟雷，张兵* 1](#_Toc41930921)

[**Charpter01** 3](#_Toc41930922)

[**lua对象模型的基础介绍** 3](#_Toc41930923)

[**参考文章：** 10](#_Toc41930924)

[**Chapter02** 11](#_Toc41930925)

[**string** 11](#_Toc41930926)

[**参考文章：** 27](#_Toc41930927)

[**Chapter03** 28](#_Toc41930928)

[**table** 29](#_Toc41930929)

[**最后就会调用到下面的函数luaH\_newkey** 32](#_Toc41930930)

[**更多table的用法库** 56](#_Toc41930931)

[**参考文章：** 58](#_Toc41930932)

[**Chapter04** 60](#_Toc41930933)

[**lua的gc算法以及碎片整理** 60](#_Toc41930934)

[**那些值会放到allgc链表中？** 60](#_Toc41930935)

[**字符串和userdata** 64](#_Toc41930936)

[**GC Garbage Collect** 67](#_Toc41930937)

[**参考文章：** 71](#_Toc41930938)

[**Chapter05 GC详述** 73](#_Toc41930939)

[**参考文章：** 98](#_Toc41930940)

[**Chapter06 闭包** 99](#_Toc41930941)

[参考文章： 107](#_Toc41930942)

**Charpter01**

源码以lua5.3.3进行分析。更多版本源码下载请访问官网 <https://www.lua.org/ftp/>

目录：

1. 基本的对象模型
2. 介绍字符串的存储
3. 介绍table的存储（比较多）
4. 介绍一点内存释放（gc）相关
5. 函数 TODO
6. 闭包 TODO
7. 编译 TODO
8. 其它 TODO

* [Charpter01](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#charpter01)
  + [lua对象模型的基础介绍](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#lua%E5%AF%B9%E8%B1%A1%E6%A8%A1%E5%9E%8B%E7%9A%84%E5%9F%BA%E7%A1%80%E4%BB%8B%E7%BB%8D)
    - [GCObject Value TValue](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#gcobject-value-tvalue)
    - [nil值](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#nil%E5%80%BC)
    - [lua\_State](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#lua_state)
    - [内存分配函数](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#%E5%86%85%E5%AD%98%E5%88%86%E9%85%8D%E5%87%BD%E6%95%B0)
    - [字节码](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#%E5%AD%97%E8%8A%82%E7%A0%81)
  + [参考文章：](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter01.md#%E5%8F%82%E8%80%83%E6%96%87%E7%AB%A0)

**lua对象模型的基础介绍**

Lua 8种数据类型

nil （以下5种是以值的形式存在）

boolean

number

userdata （full userdata）

lightuserdata (它不算lua的基本类型)

string (下面4中在vm中以引用方式共享)

table

function

thread

下面介绍些lua的C源码中，常见的数据结构

**GCObject Value TValue**

lobject.h 72行，迎来了首个比较重要的数据结构GCObject，另外CommonHeader中的tt字段，可以理解为是用来标记lua 8种数据类型，故：

/\*

\*\* Common Header for all collectable objects (in macro form, to be

\*\* included in other objects)

\*/

// 所有需要GC操作的数据都会加一个CommonHeader类型的宏定义

// next指向下一个GC链表的数据

// tt代表数据的类型以及扩展类型以及GC位的标志

// marked是执行GC的标记为，用于具体的GC算法

#define CommonHeader GCObject \*next; lu\_byte tt; lu\_byte marked

struct GCObject {

CommonHeader;

};

// 展开之后

struct GCObject {

GCObject \* next;

lu\_byte tt;

lu\_byte marked;

}

另外一个非常重要的数据结构，Value

typedef union Value {

GCObject \*gc; /\* collectable objects \*/

void \*p; /\* light userdata \*/

int b; /\* booleans \*/

lua\_CFunction f; /\* light C functions \*/

lua\_Integer i; /\* integer numbers \*/

lua\_Number n; /\* float numbers \*/

} Value;

Value类型是一个联合体，其中gc指针，把所有的GCObject串起来，这个在后面垃圾回收（gc）的时候会用到  
其中p字段表示的是light userdata，下面介绍下light userdata 和 userdata的区别：

Userdata represent C values in Lua. A light userdata represents a pointer . It is a value (like a number)

上面这句话是 lua 官网对二者的描述，从这里能看出来 light userdata实际上就是一个C指针，它的生命周期有C/C++控制，而userdata可以理解为，C/C++对象，创建是在lua里创建，内存的释放也是lua虚拟机来进行管理的。

接着上面的Value联合体说，b字段用来标示bool值，f字段是跟lua51版本比，新增加出来的一个字段

i字段，用来标示整型，n字段表示浮点数。（lua51中，只有一个lua\_Number字段，53版本进行了一个细化拆分）

再往下面看，TValue类型

typedef struct lua\_TValue

{

Value value\_;

int tt\_;

}TValue;

这里在Value基础上，增加了tt\_字段，来标示数据是什么类型，基础类型定义在lua.h头文件中

后面定义了一些常用的宏  
ttisXXX表示判断这个o对象，是否是XXX类型

略过一段，在往下看  
setXXXvalue宏，设置一个obj对象为XXX类型，并且赋值x  
从这些宏就可以看出来，是如何设置一个TValue对象的类型，和对它进行赋值操作的是TValue的那个字段。

其它类型，string table Closure(闭包) Proto 暂时先不介绍，等到后面具体到某个类型在介绍。先了解这么多就可以了，因为后面忘了还是会回来重新看这些宏定义的。

**nil值**

参考chapter03，查找key

**lua\_State**

另外还需要了解一个概念，就是lua\_State，这是lua里的一个线程。定义在lstate.h  
这里面有个global\_State对象，理解为全局量都保存在这里。  
后面介绍的 字符串 全部都保存在global\_State的strt这个hash表中

**内存分配函数**

在global\_State中，有个字段

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

这是一个函数指针。  
在strt的resize以及table的resize函数里，都会有看到类似的代码，去申请一块内存空间。  
最终都会调用到如下部分：

/\*

\*\* generic allocation routine.

\*/

void \*luaM\_realloc\_ (lua\_State \*L, void \*block, size\_t osize, size\_t nsize) {

void \*newblock;

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

size\_t realosize = (block) ? osize : 0;

lua\_assert((realosize == 0) == (block == NULL));

#if defined(HARDMEMTESTS)

if (nsize > realosize && g->gcrunning)

luaC\_fullgc(L, 1); /\* force a GC whenever possible \*/

#endif

newblock = (\*g->frealloc)(g->ud, block, osize, nsize);

if (newblock == NULL && nsize > 0) {

lua\_assert(nsize > realosize); /\* cannot fail when shrinking a block \*/

if (g->version) { /\* is state fully built? \*/

luaC\_fullgc(L, 1); /\* try to free some memory... \*/

newblock = (\*g->frealloc)(g->ud, block, osize, nsize); /\* try again \*/

}

if (newblock == NULL)

luaD\_throw(L, LUA\_ERRMEM);

}

lua\_assert((nsize == 0) == (newblock == NULL));

g->GCdebt = (g->GCdebt + nsize) - realosize;

return newblock;

}

搜来搜去，最后找到了这个。这个函数是lua提供的默认内存管理函数。

static void \*l\_alloc (void \*ud, void \*ptr, size\_t osize, size\_t nsize) {

(void)ud; (void)osize; /\* not used \*/

if (nsize == 0) {

free(ptr);

return NULL;

}

else

return realloc(ptr, nsize);

}

下面这段摘取自[博客](https://www.cnblogs.com/heartchord/p/4527494.html)

ud　 ：Lua默认内存管理器并未使用该参数。不过在用户自定义内存管理器中，可以让内存管理在不同的堆上进行。

ptr　：非NULL表示指向一个已分配的内存块指针，NULL表示将分配一块nsize大小的新内存块。

osize：原始内存块大小，默认内存管理器并未使用该参数。Lua的设计强制在调用内存管理器函数时候需要给出原始内存块的大小信息，如果用户需要自定义一个高效的内存管理器，那么这个参数信息将十分重要。这是因为大多数的内存管理算法都需要为所管理的内存块加上一个cookie，里面存储了内存块尺寸的信息，以便在释放内存的时候能够获取到尺寸信息(譬如多级内存池回收内存操作)。而Lua内存管理器刻意在调用内存管理器时提供了这个信息，这样就不必额外存储这些cookie信息，这样在大量使用小内存块的环境中将可以节省不少的内存。另外在ptr传入NULL时，osize表示Lua对象类型（LUA\_TNIL、LUA\_TBOOLEAN、LUA\_TTHREAD等等），这样内存管理器就可以知道当前在分配的对象的类型，从而可以针对它做一些统计或优化的工作。

nsize：新的内存块大小，特别地，在nsize为0时需要提供内存释放的功能。

返回的就是realloc这个指针，最后都会调用到这个函数。

void \*realloc (void \*ptr, size\_t new\_size );

realloc函数用于修改一个原先已经分配的内存块的大小，可以使一块内存的扩大或缩小。当起始空间的地址为空，即\*ptr = NULL,则同malloc。当\*ptr非空：若nuw\_size < size,即缩小\*ptr所指向的内存空间，该内存块尾部的部分内存被拿掉，剩余部分内存的原先内容依然保留；若nuw\_size > size,即扩大\*ptr所指向的内存空间，如果原先的内存尾部有足够的扩大空间，则直接在原先的内存块尾部新增内存，如果原先的内存尾部空间不足，或原先的内存块无法改变大小，realloc将重新分配另一块new\_size大小的内存，并把原先那块内存的内容复制到新的内存块上。因此，使用realloc后就应该改用realloc返回的新指针。

* lua中的gc管理，是采用了改进的三色标记法，具体三色标记法是如何设计，和如何gc的，请参考Chapter04，有很多动图讲解的很清楚，所以后面代码中会看到很多的白灰黑三色的一些变量以及宏定义。

**字节码**

luac.exe -l -p xxx.lua

来查看xxx.lua 编译的字节码是什么格式的，其中lua的源码如下：

local a = { key = 1}

a.key = nil

menglei@menglei-PC MINGW64 /d/software/lua-5.3.3\_Win32\_bin

$ ./luac.exe -l -p a.lua

main <a.lua:0,0> (4 instructions at 0045e840)

0+ params, 2 slots, 1 upvalue, 1 local, 3 constants, 0 functions

1 [2] NEWTABLE 0 0 1

2 [2] SETTABLE 0 -1 -2 ; "key" 1

3 [3] SETTABLE 0 -1 -3 ; "key" nil

4 [3] RETURN 0 1

**参考文章：**

<https://www.cnblogs.com/heartchord/p/4527494.html> Lua内存管理器规则

**Chapter02**

* [Chapter02](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#chapter02)
  + [string](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#string)
    - [区分长字符串和短字符串](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%8C%BA%E5%88%86%E9%95%BF%E5%AD%97%E7%AC%A6%E4%B8%B2%E5%92%8C%E7%9F%AD%E5%AD%97%E7%AC%A6%E4%B8%B2)
    - [结构定义](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E7%BB%93%E6%9E%84%E5%AE%9A%E4%B9%89)
    - [构造一个lua字符串的一般方法](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E6%9E%84%E9%80%A0%E4%B8%80%E4%B8%AAlua%E5%AD%97%E7%AC%A6%E4%B8%B2%E7%9A%84%E4%B8%80%E8%88%AC%E6%96%B9%E6%B3%95)
    - [构造一个长字符串](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E6%9E%84%E9%80%A0%E4%B8%80%E4%B8%AA%E9%95%BF%E5%AD%97%E7%AC%A6%E4%B8%B2)
    - [字符串的hash算法](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%AD%97%E7%AC%A6%E4%B8%B2%E7%9A%84hash%E7%AE%97%E6%B3%95)
    - [构造一个短字符串](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E6%9E%84%E9%80%A0%E4%B8%80%E4%B8%AA%E7%9F%AD%E5%AD%97%E7%AC%A6%E4%B8%B2)
    - [字符串缓存](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%AD%97%E7%AC%A6%E4%B8%B2%E7%BC%93%E5%AD%98)
    - [resize strt数组](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#resize-strt%E6%95%B0%E7%BB%84)
    - [字符串的比较](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%AD%97%E7%AC%A6%E4%B8%B2%E7%9A%84%E6%AF%94%E8%BE%83)
    - [字符串拼接](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%AD%97%E7%AC%A6%E4%B8%B2%E6%8B%BC%E6%8E%A5)
    - [删除字符串](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%88%A0%E9%99%A4%E5%AD%97%E7%AC%A6%E4%B8%B2)
  + [参考文章：](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter02.md#%E5%8F%82%E8%80%83%E6%96%87%E7%AB%A0)

**string**

这里介绍下字符串在lua中是如何存储的

lua中的字符串全部是引用，相同的字符串只有一份存储（global\_State的strt字段）。  
string这部分相对比较独立，可以先拿出来讲一下。主要在lstring.h lstring.c 这两个文件中。

lua中的字符串分为长字符串和短字符串，

**区分长字符串和短字符串**

luaS\_newlstr 方法是构造一个字符串，在这里有一些判断条件， 用来划分什么样的字符串是长字符串，什么样的字符串是短字符串。

长度不大于40（#define LUAI\_MAXSHORTLEN 40）即为短字符串，否则为长字符串。

**结构定义**

lua中字符串数据结构的定义 在文件lobject中，

typedef struct TString {

CommonHeader;

lu\_byte extra; /\* reserved words for short strings; "has hash" for longs \*/

lu\_byte shrlen; /\* length for short strings \*/

unsigned int hash;

union {

size\_t lnglen; /\* length for long strings \*/

struct TString \*hnext; /\* linked list for hash table \*/

} u;

} TString;

从这个数据结构中，我们可以看出来lua是如何来存储字符串的：

* CommonHeader 标示这是一个需要GC的对象
* extra 用于记录辅助信息。对于短字符串，该字段用来标记字符串是否为保留字，用于词法分析器中对保留字的快速判断；对于长字符串，该字段将用于惰性求哈希值的策略（第一次用到才进行哈希）。
* strlen 字符串长度，lua里的字符串并不是以\0结尾的
* hash 哈希值
* hnext 哈希表中将所有相同hash值的字符串串成一个链表，该字段为下一个节点的指针
* lnglen 长字符串长度

紧接着TString,定义了UTString类型

/\*

\*\* Ensures that address after this type is always fully aligned.

\*/

typedef union UTString {

L\_Umaxalign dummy; /\* ensures maximum alignment for strings \*/

TString tsv;

} UTString;

UserData在lua中和string类似，可以看成是拥有独立元表，不被内部化，也不需要追加\0的字符串  
故定义了一个这样的联合体，L\_Umaxalign字段在UData中也有类似的声明。

lua 字符串在内存中的表示如上图（[图片来自博客](https://www.cnblogs.com/heartchord/p/4561308.html)）

lua中的字符串的存储结构 如下图：

**构造一个lua字符串的一般方法**

lstring.c createstrobj 方法会构造出来一个新的字符串，其函数原型如下：

/\*

\*\* creates a new string object

\*/

// l字符串的长度，tag是字符串的类型，h是默认的hash种子

// sizelstring就是求出UTString的size

// luaC\_newobj创建一个可以被GC的对象

// 然后再把o转换成TString类型，继续设置ts的hash字段，和extra字段(extra用于标记是否是虚拟机保留的字符串，如果这个值为1，那么不会GC)

// 然后把字符串的最后以'\0'结尾

static TString \*createstrobj (lua\_State \*L, size\_t l, int tag, unsigned int h){

TString \*ts;

GCObject \*o;

size\_t totalsize; /\* total size of TString object \*/

totalsize = sizelstring(l);

o = luaC\_newobj(L, tag, totalsize);

ts = gco2ts(o); // gc object to TString

ts->hash = h;

ts->extra = 0;

getstr(ts)[l] = '\0'; /\* ending 0 \*/

return ts;

}

其中 sizelstring 宏定义如下：

#define sizelstring(l) (sizeof(union UTString) + ((l) + 1) \* sizeof(char))

求一个UTString的大小+ （l+1）个char类型的内存空间大小。TString对象之后的内存空间，存储了真正的字符串的内容。

luaC\_newobj这个函数在后面也会有很多地方用到，创建一个可以回收的对象，并且把这个对象添加到g->allgc表头

/\*

\*\* create a new collectable object (with given type and size) and link

\*\* it to 'allgc' list.

\*/

GCObject \*luaC\_newobj (lua\_State \*L, int tt, size\_t sz) {

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

GCObject \*o = cast(GCObject \*, luaM\_newobject(L, novariant(tt), sz));

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

o->tt = tt;

o->next = g->allgc;

g->allgc = o;

return o;

}

gco2ts这个宏，设置GCObject转换为TString类型，并且设置tt字段

// gc object to TString

#define gco2ts(o) \

check\_exp(novariant((o)->tt) == LUA\_TSTRING, &((cast\_u(o))->ts))

getstr这个宏也蛮重要的，后面也会多次遇到，其中check\_exp宏，有两个参数，第一个参数只是执行下，并没有任何返回值，一般只是用来校验是否有该字段，如果没有的话就会崩溃，返回cast的那部分值。这里返回一个指针，将ts+UTString类型大小偏移的一块内存（注意这里是ts的地址加上UTString size，获得的一个偏移地址，用来存储真正的字符串的首地址，这里也印证了上图中字符串的存储结构），转换为char\* 返回回去。用于memcmp

/\*

\*\* Get the actual string (array of bytes) from a 'TString'.

\*\* (Access to 'extra' ensures that value is really a 'TString'.)

\*/

#define getstr(ts) \

check\_exp(sizeof((ts)->extra), cast(char \*, (ts)) + sizeof(UTString))

**构造一个长字符串**

搜索createstrobj会发现，只有两处调用，分别是构造一个长字符串，一个短字符串。

构造一个长字符串很简单，调用createstrobj，将tag参数设置为长字符串类型即可。

// 创建一个长字符串

TString \*luaS\_createlngstrobj (lua\_State \*L, size\_t l) {

TString \*ts = createstrobj(L, l, LUA\_TLNGSTR, G(L)->seed);

ts->u.lnglen = l;

return ts;

}

这里hash种子直接用的seed字段，设置u.lnglen 为字符串长度

注意，hash 长字符串和短字符串的哈希方法不同

**字符串的hash算法**

unsigned int luaS\_hash (const char \*str, size\_t l, unsigned int seed) {

unsigned int h = seed ^ cast(unsigned int, l);

size\_t step = (l >> LUAI\_HASHLIMIT) + 1;

for (; l >= step; l -= step)

h ^= ((h<<5) + (h>>2) + cast\_byte(str[l - 1]));

return h;

}

对于比较长的字符串（32字节以上），为了加快哈希过程，计算字符串哈希值是跳跃进行的。跳跃的步长（step）是由LUAI\_HASHLIMIT宏控制的。

/\*

\*\* Lua will use at most ~(2^LUAI\_HASHLIMIT) bytes from a string to

\*\* compute its hash

\*/

#if !defined(LUAI\_HASHLIMIT)

#define LUAI\_HASHLIMIT 5

#endif

Hash DoS攻击：攻击者构造出上千万个拥有相同哈希值的不同字符串，用来数十倍地降低Lua从外部压入字符串到内部字符串表的效率。当Lua用于大量依赖字符串处理的服务（例如HTTP）的处理时，输入的字符串将不可控制， 很容易被人恶意利用 。

为了防止Hash DoS攻击的发生，Lua一方面将长字符串独立出来，大文本的输入字符串将不再通过哈希内部化进入全局字符串表中；另一方面使用一个随机种子用于字符串哈希值的计算，使得攻击者无法轻易构造出拥有相同哈希值的不同字符串。

随机种子是在创建虚拟机的global\_State（全局状态机）时构造并存储在global\_State中的。随机种子也是使用luaS\_hash函数生成，它利用内存地址随机性以及一个用户可配置的一个随机量（luai\_makeseed宏）同时来决定。

用户可以在luaconf.h中配置luai\_makeseed来定义自己的随机方法，Lua默认是利用time函数获取系统当前时间来构造随机种子。luai\_makeseed的默认行为有可能给调试带来一些困扰： 由于字符串hash值的不同，程序每次运行过程中的内部布局将有一些细微变化，不过字符串池使用的是开散列算法， 这个影响将非常小。如果用户希望让嵌入Lua的程序每次运行都严格一致，那么可以自定义luai\_makeseed函数来实现。

**构造一个短字符串**

直接看代码吧

/\*

\*\* checks whether short string exists and reuses it or creates a new one

\*/

// 判断这个（短）字符串是否存在，存在的话就重用不然就创建一个新的

static TString \*internshrstr (lua\_State \*L, const char \*str, size\_t l) {

TString \*ts;

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

unsigned int h = luaS\_hash(str, l, g->seed);

TString \*\*list = &g->strt.hash[lmod(h, g->strt.size)];

lua\_assert(str != NULL); /\* otherwise 'memcmp'/'memcpy' are undefined \*/

for (ts = \*list; ts != NULL; ts = ts->u.hnext) {

if (l == ts->shrlen &&

(memcmp(str, getstr(ts), l \* sizeof(char)) == 0)) {

/\* found! \*/

if (isdead(g, ts)) /\* dead (but not collected yet)? \*/

changewhite(ts); /\* resurrect it \*/

return ts;

}

}

if (g->strt.nuse >= g->strt.size && g->strt.size <= MAX\_INT/2) {

luaS\_resize(L, g->strt.size \* 2);

list = &g->strt.hash[lmod(h, g->strt.size)]; /\* recompute with new size \*/

}

ts = createstrobj(L, l, LUA\_TSHRSTR, h);

memcpy(getstr(ts), str, l \* sizeof(char));

ts->shrlen = cast\_byte(l);

ts->u.hnext = \*list;

\*list = ts;

g->strt.nuse++;

return ts;

}

获取字符串hash值的方法使用的是luaS\_hash

后面会遇到一个很重要的，根据hash值，来获取在数组中index的方法lmod

/\*

\*\* 'module' operation for hashing (size is always a power of 2)

\*/

#define lmod(s,size) \

(check\_exp((size&(size-1))==0, (cast(int, (s) & ((size)-1)))))

其中size一定是2的n次幂  
当size是2的幂次时，(s) & ((size)-1)) = s % size 只进行了一次与运算。  
为什么采取这种hash方式可以参考[博文](https://manistein.github.io/blog/post/program/build-a-lua-interpreter/%E6%9E%84%E5%BB%BAlua%E8%A7%A3%E9%87%8A%E5%99%A8part4/)

摘取例子：

* 计算”table”这个字符串的hash值，假设得到01101011 00100100 10001101 001011002
* table的hash表的lsizenode值为3，也就是size为8，于是有(2^lsizenode)-1 = 7 = 0111
* 计算”table”在hash表中的下标，于是有01101011 00100100 10001101 001011002 & 0111，由于右边的值高位全是0，因此只需要截取”table”字符串hash值的低4位即可，于是有index = 1100 & 0111 = 0100 = 4
* 于是key为”table”的node，将会被定位到hash[4]的位置上

接着看这个函数internshrstr。  
找到了list这个值，这是一个指向指针的指针，它表示hash数组里面，冲突的链表的头结点。下面开始在这个链表里去遍历，

这里memcmp,只比较l长度的内容（实际字符串的内容），而前面TString部分的并没有比较。  
如果找到了，也就是之前在内存里就有存储，那么返回这个地址。  
如果没有找到，那么就先判断strt还能不能放下，是否需要resize操作，进行扩容。  
调用createstrobj申请一块内存空间，存放字符串，并且把串到链表的表头。

**字符串缓存**

这个函数是构造一个字符串的入口函数。  
每当构造一个字符串类型对象时，先去strcache中去查找，若没有找到，那么就会调用luaS\_newlstr来创建新的字符串。  
luaS\_newlstr函数中根据字符串的长度，来区分长字符串还是短字符串，然后调用不同的构造方法，来构造字符串。  
其中在构造短字符串，会先去hash链表中去查找有没有相同的字符串，而对于长字符串则没有做重复检查，也就是长字符串是有可能重复的。  
最后都会调用到createstrobj函数，创建一个TString对象，连接到allgc的表头。

TString \*luaS\_new (lua\_State \*L, const char \*str) {

unsigned int i = point2uint(str) % STRCACHE\_N; /\* hash \*/

int j;

TString \*\*p = G(L)->strcache[i];

for (j = 0; j < STRCACHE\_M; j++) {

if (strcmp(str, getstr(p[j])) == 0) /\* hit? \*/

return p[j]; /\* that is it \*/

}

/\* normal route \*/

for (j = STRCACHE\_M - 1; j > 0; j--)

p[j] = p[j - 1]; /\* move out last element \*/

/\* new element is first in the list \*/

p[0] = luaS\_newlstr(L, str, strlen(str));

return p[0];

}

网上关于strcache字段讲的不是很多，这是一个二维数组  
TString \*strcache[53][2];  
每次new一个字符串的时候，会先去缓存里找，如果没找到，那么就创建一个新的字符串，p[1] = p[0],将原来p[0]位置的字符串放到p[1],并且把新字符串的地址放在p[0]的位置。一种LRU置换算法。

**resize strt数组**

到这里，lua 字符串相关基本上都已经顺利完成了。（还差删除字符串）

我们知道，lua 中的字符串全部保存在global\_State的strt字段，另外strcache字段里，保存了一份53\*2个字符串的缓存，下面就分析下strt字段，是如何进行resize的。

先看下strt的结构体是如何定义的：

typedef struct stringtable {

TString \*\*hash;

int nuse; /\* number of elements \*/

int size;

} stringtable;

保存了一个数组指针，一个当前保存了多少个元素字段nuse，size表示当前hash表的容量。

void luaS\_resize (lua\_State \*L, int newsize) {

int i;

stringtable \*tb = &G(L)->strt;

if (newsize > tb->size) { /\* grow table if needed \*/

luaM\_reallocvector(L, tb->hash, tb->size, newsize, TString \*);

for (i = tb->size; i < newsize; i++)

tb->hash[i] = NULL;

}

for (i = 0; i < tb->size; i++) { /\* rehash \*/

TString \*p = tb->hash[i];

tb->hash[i] = NULL;

while (p) { /\* for each node in the list \*/

TString \*hnext = p->u.hnext; /\* save next \*/

unsigned int h = lmod(p->hash, newsize); /\* new position \*/

p->u.hnext = tb->hash[h]; /\* chain it \*/

tb->hash[h] = p;

p = hnext;

}

}

if (newsize < tb->size) { /\* shrink table if needed \*/

/\* vanishing slice should be empty \*/

lua\_assert(tb->hash[newsize] == NULL && tb->hash[tb->size - 1] == NULL);

luaM\_reallocvector(L, tb->hash, tb->size, newsize, TString \*);

}

tb->size = newsize;

}

我们先全局搜索下luaS\_resize函数在哪里调用的，一共有3处调用

* gc的时候，如果当前使用量小于总容量的1/4，那么就把容量缩小为原来的一半
* 初始化，初始容量为128（MINSTRTABSIZE）
* 插入新的短字符串，会检查当前nuse字段是否不小于size字段，并且不大于MAX\_INT/2，则容量翻倍

（思考，为什么在插入长字符串的时候没有检查扩容？）  
答：长字符串是链接到allgc上的，而短字符串是放到hash部分的。

for (i = 0; i < tb->size; i++) { /\* rehash \*/

TString \*p = tb->hash[i];

tb->hash[i] = NULL;

while (p) { /\* for each node in the list \*/

TString \*hnext = p->u.hnext; /\* save next \*/

unsigned int h = lmod(p->hash, newsize); /\* new position \*/

p->u.hnext = tb->hash[h]; /\* chain it \*/

tb->hash[h] = p;

p = hnext;

}

}

中间这部分rehash函数很有趣，巧妙的安排在了扩容之后，或者是在做缩小容量之前。  
这里稍加思考下，这样循环遍历一遍，从0 ~ tb->size，当前的hash[i]这个链表的内容，在重新hash下，可能会插在hash[i+n]的位置上，也就是hash[i]的后面，同时i+n < tb->size。  
也就是这里面的内容可能还要在遍历一次，因为是链接到链表上了。  
这里并不会造成死循环，或者其他问题，只可能会多遍历一遍或者 tb->size - 1次（最坏情况下）

申请内存空间，请参考realloc。

最后，修改 tb->size 为新的大小。

**字符串的比较**

先去分长短字符串，然后在根据不同的策略去比较

在函数luaV\_equalobj中

case LUA\_TSHRSTR: return eqshrstr(tsvalue(t1), tsvalue(t2));

case LUA\_TLNGSTR: return luaS\_eqlngstr(tsvalue(t1), tsvalue(t2));

由于短字符串已经内化的一种数据，所以直接比较其地址即可

/\*

\*\* equality for short strings, which are always internalized

\*/

#define eqshrstr(a,b) check\_exp((a)->tt == LUA\_TSHRSTR, (a) == (b))

对于长字符串，先比较是否是同一个实例，在比较字符串长度，在逐字节比较

int luaS\_eqlngstr (TString \*a, TString \*b) {

size\_t len = a->u.lnglen;

lua\_assert(a->tt == LUA\_TLNGSTR && b->tt == LUA\_TLNGSTR);

return (a == b) || /\* same instance or... \*/

((len == b->u.lnglen) && /\* equal length and ... \*/

(memcmp(getstr(a), getstr(b), len) == 0)); /\* equal contents \*/

}

**字符串拼接**

luaV\_concat 这个函数，拼接字符串都会生成一个新的字符串  
如果是少量的字符串拼接性能还可以接受，但是如果是大量的字符串拼接，使用..来拼接，那么性能就会非常差

table.concat函数就提供了一个相对较好的性能，实测（xlnt库 + lua），使用..来拼接导出道具表，耗时60.119s，而使用table.concat来拼接所有的字符串时，耗时10.119s。将表load到内存占据了主要时间，可见table.concat方法拼接大量字符串还是很快的。

源码中使用了一个luaL\_Buffer缓存

static int tconcat (lua\_State \*L) {

luaL\_Buffer b;

lua\_Integer last = aux\_getn(L, 1, TAB\_R);

size\_t lsep;

const char \*sep = luaL\_optlstring(L, 2, "", &lsep);

lua\_Integer i = luaL\_optinteger(L, 3, 1);

last = luaL\_optinteger(L, 4, last);

luaL\_buffinit(L, &b);

for (; i < last; i++) {

addfield(L, &b, i);

luaL\_addlstring(&b, sep, lsep);

}

if (i == last) /\* add last value (if interval was not empty) \*/

addfield(L, &b, i);

luaL\_pushresult(&b);

return 1;

}

**删除字符串**

local str = "hello"

str = nil

对应的机器码如下：

4 [5] LOADK 1 -4 ; "hello"

5 [6] LOADNIL 1 0

会调用setnilvalue 将tt\_字段设置为空。而内存回收阶段才会对这个字符串占用的内存空间进行清理。

luaS\_remove 这个函数，是在gc阶段删除掉字符串的时候，会调用的

void luaS\_remove (lua\_State \*L, TString \*ts) {

stringtable \*tb = &G(L)->strt;

TString \*\*p = &tb->hash[lmod(ts->hash, tb->size)];

while (\*p != ts) /\* find previous element \*/

p = &(\*p)->u.hnext;

\*p = (\*p)->u.hnext; /\* remove element from its list \*/

tb->nuse--;

}

目前网上传了一份关于lstring.c这个文件的中文注释，其中对这个函数的注释如下：

// 从全局变量就是global\_State的strt成员里面移除特定字符串

// 首先得到tb，指向strt数组，然后再通过tb的hash数组通过提供tb的长度和字符串的hash，来找到字符串属于哪个链表

// 然后一直循环，直到找到等于ts的，然后就把这个字符串的地址给抹去了(不会内存泄漏？？？）

确实，单看这个函数，会造成内存泄露，原来持有这个对象的指针变成了一个悬空指针。  
但是，在lgc.c中freeobj这个函数的调用处可以看到，在remove之后，紧接着释放掉了该对象的内存。这也正符合一个函数只做一件事情的原则。

luaS\_remove(L, gco2ts(o)); /\* remove it from hash table \*/

luaM\_freemem(L, o, sizelstring(gco2ts(o)->shrlen));

**参考文章：**

<https://www.cnblogs.com/heartchord/p/4561308.html>

[https://manistein.github.io/blog/post/program/build-a-lua-interpreter/构建lua解释器part4/](https://manistein.github.io/blog/post/program/build-a-lua-interpreter/%E6%9E%84%E5%BB%BAlua%E8%A7%A3%E9%87%8A%E5%99%A8part4/)

<http://lua-users.org/lists/lua-l/2012-01/msg00497.html>

<https://blog.csdn.net/u013517637/article/details/79002243>

**Chapter03**

* [Chapter03](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#chapter03)
  + [table](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#table)
    - [构造一个空table](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E6%9E%84%E9%80%A0%E4%B8%80%E4%B8%AA%E7%A9%BAtable)
    - [插入一个key](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E6%8F%92%E5%85%A5%E4%B8%80%E4%B8%AAkey)
    - [向table中插入一个元素](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E5%90%91table%E4%B8%AD%E6%8F%92%E5%85%A5%E4%B8%80%E4%B8%AA%E5%85%83%E7%B4%A0)
    - [table的rehash](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#table%E7%9A%84rehash)
    - [那些key存在数组部分那些存在hash部分？](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E9%82%A3%E4%BA%9Bkey%E5%AD%98%E5%9C%A8%E6%95%B0%E7%BB%84%E9%83%A8%E5%88%86%E9%82%A3%E4%BA%9B%E5%AD%98%E5%9C%A8hash%E9%83%A8%E5%88%86)
    - [查找key](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E6%9F%A5%E6%89%BEkey)
    - [#求table大小](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E6%B1%82table%E5%A4%A7%E5%B0%8F)
    - [遍历，pairs ipairs](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E9%81%8D%E5%8E%86pairs-ipairs)
    - [删除key](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E5%88%A0%E9%99%A4key)
  + [更多table的用法库](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E6%9B%B4%E5%A4%9Atable%E7%9A%84%E7%94%A8%E6%B3%95%E5%BA%93)
  + [参考文章：](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter03.md#%E5%8F%82%E8%80%83%E6%96%87%E7%AB%A0)

**table**

在第一章介绍了TValue类型数据结构，在介绍table之前，需要介绍两个重要的类型：

typedef union TKey {

struct {

TValuefields;

int next; /\* for chaining (offset for next node) \*/

} nk; // node key

TValue tvk;

} TKey;

typedef struct Node {

TValue i\_val;

TKey i\_key;

} Node;

对于TKey，任何时候只有两种类型，要么是整数，要么不是整数(not nil)  
next字段在之前版本是指针，5.3版本换成了偏移，指向下一个偏移的节点

Node是table的hash部分节点值。

然后是真正的table类型的定义：

typedef struct Table {

CommonHeader;

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

lu\_byte lsizenode; /\* log2 of size of 'node' array \*/

unsigned int sizearray; /\* size of 'array' array \*/

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

Node \*node; // hash部分

Node \*lastfree; /\* any free position is before this position \*/

struct Table \*metatable;

GCObject \*gclist;

} Table;

对于Table，先挑重点的说。  
我们知道table内部实际上分为数组部分和hash部分，其中数组部分存在array数组里，hash部分存放在node数组里；数组部分的容量为sizearray，hash部分容量为2的lsizenode次幂（hash部分容量总是2的N次幂，这个规则后面还会提到）。lastfree是一个指针，初始指向一个dummyNode，之后会随着插入新节点产生冲突时，由node数组的尾部向前移动。

根据lua代码中使用table的情况，会从构造一个table，索引，插入，删除等来分析table内部是如何存储值的。主要在ltable.c中

**构造一个空table**

Table \*luaH\_new (lua\_State \*L) {

GCObject \*o = luaC\_newobj(L, LUA\_TTABLE, sizeof(Table));

Table \*t = gco2t(o);

t->metatable = NULL;

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

t->array = NULL;

t->sizearray = 0;

setnodevector(L, t, 0);

return t;

}

luaH\_new函数是创建一个空的table，luaC\_newobj函数定义在lgc.c中，在上一章，创建一个字符串对象createstrobj方法时候也有用到。

gco2t这个宏，将o的对象类型转换为Table，这里还要介绍下GCUnion

/\*

\*\* Union of all collectable objects (only for conversions)

\*/

union GCUnion {

GCObject gc; /\* common header \*/

struct TString ts;

struct Udata u;

union Closure cl;

struct Table h;

struct Proto p;

struct lua\_State th; /\* thread \*/

};

#define gco2t(o) check\_exp((o)->tt == LUA\_TTABLE, &((cast\_u(o))->h))

gco2t最后会调用到cast\_u这个宏

#define cast\_u(o) cast(union GCUnion \*, (o))

还有很多类似 gco2XXX 的宏，这里就不一一介绍了。

将new出来的table对象，metatable元表字段设置为空，数组部分初始为空，大小为0

hash部分，则通过setnodevector函数来调整。（这个函数在resize时候还会提到）  
初始化时候size为0，node指向了一个dummynode，hash部分的size也是0，lastfree是个空指针。

**插入一个key**

lua\_settable 这个函数是从C调过来的。会调用到luaV\_settable这个宏。  
它会优先调用luaV\_fastset，如果luaV\_fastset返回false，那么会调用luaV\_finishset。

插入一个key，先会去这个table里查这个key是否存在，如果存在，就重新设置新的值。  
否则会先去找这个table里面有没有元表，没有元表并且上步查找key对应的slot是一个luaO\_nilobject，那么就设置一个新的值。如果有元表，那么就去执行元表的方法。

**最后就会调用到下面的函数luaH\_newkey**

**向table中插入一个元素**

这个函数比较长，下面慢慢说。其中一些简单的宏定义就不说明了，基本上lua源码里的宏都还算很好理解。  
luaH\_newkey方法返回一个节点的指针，这个节点的指针即为key对应的value节点。

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

Node \*mp;

TValue aux;

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

else if (ttisfloat(key)) {

lua\_Integer k;

if (luaV\_tointeger(key, &k, 0)) { /\* does index fit in an integer? \*/

setivalue(&aux, k);

key = &aux; /\* insert it as an integer \*/

}

else if (luai\_numisnan(fltvalue(key)))

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

}

mp = mainposition(t, key);

if (!ttisnil(gval(mp)) || isdummy(t)) { /\* main position is taken? \*/

Node \*othern;

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

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

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

/\* whatever called 'newkey' takes care of TM cache \*/

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

}

lua\_assert(!isdummy(t));

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

if (othern != mp) { /\* is colliding node out of its main position? \*/

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

while (othern + gnext(othern) != mp) /\* find previous \*/

othern += gnext(othern);

gnext(othern) = cast\_int(f - othern); /\* rechain to point to 'f' \*/

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

if (gnext(mp) != 0) {

gnext(f) += cast\_int(mp - f); /\* correct 'next' \*/

gnext(mp) = 0; /\* now 'mp' is free \*/

}

setnilvalue(gval(mp));

}

else { /\* colliding node is in its own main position \*/

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

if (gnext(mp) != 0)

gnext(f) = cast\_int((mp + gnext(mp)) - f); /\* chain new position \*/

else lua\_assert(gnext(f) == 0);

gnext(mp) = cast\_int(f - mp);

mp = f;

}

}

setnodekey(L, &mp->i\_key, key);

luaC\_barrierback(L, t, key);

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

return gval(mp);

}

有几个关键的函数，其中一个就是mainposition，可以理解为根据传进来的参数和它对应的类型，计算出来一个在hash数组里的Node地址，也就是在hash表中的位置。当然不同的key可能会有相同的Node\*地址，这个时候就发生了冲突。

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

switch (ttype(key)) {

case LUA\_TNUMINT:

return hashint(t, ivalue(key));

case LUA\_TNUMFLT:

return hashmod(t, l\_hashfloat(fltvalue(key)));

case LUA\_TSHRSTR:

return hashstr(t, tsvalue(key));

case LUA\_TLNGSTR:

return hashpow2(t, luaS\_hashlongstr(tsvalue(key)));

case LUA\_TBOOLEAN:

return hashboolean(t, bvalue(key));

case LUA\_TLIGHTUSERDATA:

return hashpointer(t, pvalue(key));

case LUA\_TLCF:

return hashpointer(t, fvalue(key));

default:

lua\_assert(!ttisdeadkey(key));

return hashpointer(t, gcvalue(key));

}

}

这里将元素放入指定的hash[]位置时候，有一些原则。初始化的时候lastfree指向hash数组的最后一个指针；如果计算出来的mainposition没有元素，那么就把元素放在这个位置；若这个mainposition有元素，那么就向前移动lastfree，直到找到一个空的位置，将元素放在这里，并且设置一个next指针指向这里；还有一种情况，就是mainposition有元素，但是mainposition位置的元素计算出来的mainposition并不是这个位置，也就是说他是用链表连接起来的，那么就把这个元素向前找lastfree，然后把真正mainposition的元素插在这里。

这里描述的挺乱的，下面就引用一篇[博文](https://blog.csdn.net/fwb330198372/article/details/88579361)中的图片进行详细解释：

* 初始化时，向Table插入一个元素
* Table["k0"] = "v0"

假设k0落在node[3]的位置，此时hash部分如下图：

* 向Table插入第二个元素
* Table["k1"] = "v1"

假设此时mainposition计算的节点与k0的节点相同，那么此时发生了冲突，向左移动lastfree指针，找到node[2]位置是个空的，所以讲k1节点放在node[2]，并且node[3]的next指向node[2]

* 向Table插入第三个元素
* Table["k2"] = "v2"

假设此时mainposition计算的节点与k1的节点相同，又冲突了，但是此时与上面的冲突稍有不同，此时k1节点并不是在它真正的mainposition位置，而k2的真正的mainposition是这个节点，那么就要做出优先级让步，移动k1这个节点，把k2放在k1的位置，同时修改next指向

**table的rehash**

触发table 做rehash 操作的地方只有在向table中插入一个新key的时候，会调用getfreepos来寻找一个可用的位置，当这个函数返回空的时候，才进行rehash操作，也就是hash部分全部填满

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

if (!isdummy(t)) {

while (t->lastfree > t->node) { // 从后面向前找位置

t->lastfree--;

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

return t->lastfree;

}

}

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

}

rehash函数：

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

unsigned int asize; /\* optimal size for array part \*/

unsigned int na; /\* number of keys in the array part \*/

unsigned int nums[MAXABITS + 1];

int i;

int totaluse;

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

na = numusearray(t, nums); /\* count keys in array part \*/

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

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

/\* count extra key \*/

na += countint(ek, nums);

totaluse++;

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

asize = computesizes(nums, &na);

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

luaH\_resize(L, t, asize, totaluse - na);

}

rehash 函数里做了一个统计工作，将统计好的数据存储在nums数组里。

nums[i] = number of keys 'k' where 2^(i - 1) < k <= 2^i

nums[i] 保存了key值在 2^(i-1) 到 2^i 之间（左开右闭）区间内，key值的数量。

nums[1] (1, 2]

nums[2] (2, 4]

nums[3] (4, 8]

nums[4] (8, 16]

...

nums[i] (2^(i-1), 2^i]

在统计完之后，调用computesizes来计算数组部分的大小。

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

int i;

unsigned int twotoi; /\* 2^i (candidate for optimal size) \*/

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

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

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

/\* loop while keys can fill more than half of total size \*/

for (i = 0, twotoi = 1; \*pna > twotoi / 2; i++, twotoi \*= 2) {

if (nums[i] > 0) {

a += nums[i];

if (a > twotoi/2) { /\* more than half elements present? \*/

optimal = twotoi; /\* optimal size (till now) \*/

na = a; /\* all elements up to 'optimal' will go to array part \*/

}

}

}

lua\_assert((optimal == 0 || optimal / 2 < na) && na <= optimal);

\*pna = na;

return optimal;

}

遍历这个nums数组，获得其范围区间内所包含的整数数量大于50%的最大索引，作为重新哈希之后的数组大小，超过这个范围的正整数，就分配到哈希部分了

如果数值key的元素个数大于对应个数幂大小的一半，则生成对应幂长度的数组链表。

举个例子：

local tbl = {}

tbl[2] = 0

tbl[3] = 0

tbl[4] = 0

tbl[5] = 0

查找过程如下表格（对应上述代码段，\*pna = 4）

其中a的值为，key的个数每次累加的结果。  
twotoi在每次循环都\*2

| **i** | **区间** | **a的值(key的个数)** | **条件(a > twotoi/2)** | **optimal数组长度** | **key** |
| --- | --- | --- | --- | --- | --- |
| n | (2^(n-1), 2^n] |  |  |  |  |
| 0 | (0, 1] | 0 | 0 > 1/2 不成立 | 0 |  |
| 1 | (1, 2] | 1 | 1 > 2/2 不成立 | 0 | 2 |
| 2 | (2, 4] | 3 | 3 > 4/2 成立 | 4 | 2,3,4 |

当i = 3时，不满足 4 > 8 / 2，跳出循环，此时optimal值为4，即数组部分的大小为4

其中 key为2,3,4的value存放在数组部分，key值为5的存放在hash部分。  
此时若加入一行 tbl[1] = 0;放在第二行，那么数组的部分大小为8，1~5全部存放在数组中。并且空余出3个位置。

luaH\_resize 函数是根据之前计算的结果，来对数组部分或者hash部分，进行扩容或者收缩

void luaH\_resize (lua\_State \*L, Table \*t, unsigned int nasize,

unsigned int nhsize) {

unsigned int i;

int j;

unsigned int oldasize = t->sizearray;

int oldhsize = allocsizenode(t);

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]))

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

}

/\* shrink array \*/

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

}

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

for (j = oldhsize - 1; j >= 0; j--) {

Node \*old = nold + j;

if (!ttisnil(gval(old))) {

/\* doesn't need barrier/invalidate cache, as entry was

already present in the table \*/

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

}

}

if (oldhsize > 0) /\* not the dummy node? \*/

luaM\_freearray(L, nold, cast(size\_t, oldhsize)); /\* free old hash \*/

}

其中allocsizenode返回以2为底的散列表大小的对数值。  
setarrayvector函数 对表的数组部分进行大小调整，在chapter01中，介绍了申请内存空间的函数是realloc，这里扩容的话，对超出原有容量的数组部分，初始化其tt\_字段为LUA\_TNIL，标记为空。

如果数组部分的比原来小，那么就要收缩数组部分的大小；将 nasize 到 oldasize 之间的非空元素重新插入到数组部分。这里数组收缩部分代码，调用了一个叫luaH\_setint的函数，下面分析下这个函数：

void luaH\_setint (lua\_State \*L, Table \*t, lua\_Integer key, TValue \*value) {

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

TValue \*cell;

if (p != luaO\_nilobject)

cell = cast(TValue \*, p);

else {

TValue k;

setivalue(&k, key);

cell = luaH\_newkey(L, t, &k);

}

setobj2t(L, cell, value);

}

/\*

\*\* search function for integers

\*/

const TValue \*luaH\_getint (Table \*t, lua\_Integer key) {

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

if (l\_castS2U(key) - 1 < t->sizearray)

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

else {

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

for (;;) { /\* check whether 'key' is somewhere in the chain \*/

if (ttisinteger(gkey(n)) && ivalue(gkey(n)) == key)

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

else {

int nx = gnext(n);

if (nx == 0) break;

n += nx;

}

}

return luaO\_nilobject;

}

}

luaH\_getint这个函数可以看出来，在查找一个key值为integer类型的value时，先会比较这个integer和数组大小，如果是小于数组大小，那么就去数组中查找，否则会在hash部分查找。  
通过这个查找过程，我们也能理解部分关于key为整型时，table中数据的存储方法。正好与前文说的computesizes函数相对应。

其中luaH\_getint查找key值为integer类型对应的value值。  
luaH\_setint这个函数将需要收缩的数组部分，重新插入收缩后的数组中去。  
在这之后，收缩数组部分大小。

这里逻辑顺序比较乱，重新捋一次应该就清晰一点，总是插播

setnodevector在table初始化的时候提过，那个时候的size为0，所以只是简单的初始化即可。  
数组部分和hash部分虽然都是数组，但是申请内存空间的宏定义却是有稍许的不同。

其中数组部分申请内存的宏如下：

// 数组部分申请内存的宏

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

//...

#define luaM\_reallocvector(L, v, oldn, n, t) \

((v)=cast(t \*, luaM\_reallocv(L, v, oldn, n, sizeof(t))))

hash部分的申请内存空间的宏：

t->node = luaM\_newvector(L, size, Node);

//...

#define luaM\_newvector(L, n, t) \

cast(t \*, luaM\_reallocv(L, NULL, 0, n, sizeof(t)))

最后都会调用到luaM\_reallocv这个宏：

#define luaM\_reallocv(L, b, on, n, e) \

(((sizeof(n) >= sizeof(size\_t) && cast(size\_t, (n)) + 1 > MAX\_SIZET/(e)) \

? luaM\_toobig(L) : cast\_void(0)) , \

luaM\_realloc\_(L, (b), (on)\*(e), (n)\*(e)))

luaM\_realloc\_函数的代码在chapter01最后有贴过。  
从调用到luaM\_realloc\_这个函数，传递的参数来看，区别就是，第二个参数是否为NULL，和第三个参数是否为0。

根据l\_alloc函数的定义，第二个参数第三个参数都没用到，所以在使用默认的l\_alloc这个函数作为内存管理函数的话，二者是没有区别的。

回头接着说luaH\_resize。

由于在申请新的hash部分数组之前，已经把原来的hash部分数组的指针保存了起来，所以新申请的hash数组直接遍历一遍，初始为空类型。  
并且保存下新hash部分数组大小，lastfree指向数组最后一个指针。上述是setnodevector这个函数干的事情。

数组部分收缩在上文已经说过了。之后就是重新插入hash部分的元素（hash部分容量变了）

最后，释放掉旧的hash部分数组。

* 总结：

通过resize函数可以看出来，table中的数组部分和hash部分是如何动态变化的。其中数组部分和hash部分可能会收缩，也可能会增大其数组的容量。

只有hash部分满的时候，才会触发rehash

key为整型的值，部分存在数组里，部分存在hash里，50%的最大索引

**那些key存在数组部分那些存在hash部分？**

根据 50%的最大索引 这一规则，决定一个整型值存放在数组部分还是hash部分，其它类型的key存放在hash部分。

**查找key**

查找一个key的主方法为luaH\_get

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

switch (ttype(key)) {

case LUA\_TSHRSTR: return luaH\_getshortstr(t, tsvalue(key));

case LUA\_TNUMINT: return luaH\_getint(t, ivalue(key));

case LUA\_TNIL: return luaO\_nilobject;

case LUA\_TNUMFLT: {

lua\_Integer k;

if (luaV\_tointeger(key, &k, 0)) /\* index is int? \*/

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

/\* else... \*/

} /\* FALLTHROUGH \*/

default:

return getgeneric(t, key);

}

}

根据key的类型，去调用不同的查找方法来查找，对于key值可以转换为int类型的，那么就优先到数组里查找，大于数组大小了则去hash部分查找，如果没有找到，这里返回了一个luaO\_nilobject。

其中,luaO\_nilobject定义是一个TValue类型常量对象luaO\_nilobject\_的地址。

/\*

\*\* (address of) a fixed nil value

\*/

#define luaO\_nilobject (&luaO\_nilobject\_)

// LUAI\_DDEC extern

LUAI\_DDEF const TValue luaO\_nilobject\_ = {NILCONSTANT};

/\* macro defining a nil value \*/

#define NILCONSTANT {NULL}, LUA\_TNIL

所以luaO\_nilobject\_展开为：

luaO\_nilobject\_ =

{

value\_ = NULL;

tt\_ = LUA\_TNIL;

}

可以看到lua内部，是用这样一个常量对象的地址，来表示唯一一个nil值。

**#求table大小**

luaV\_objlen函数是主入口，Main operation 'ra' = #rb'.  
当rb的类型为table时，会走luaH\_getn函数

/\*

\*\* Try to find a boundary in table 't'. A 'boundary' is an integer index

\*\* such that t[i] is non-nil and t[i+1] is nil (and 0 if t[1] is nil).

\*/

int luaH\_getn (Table \*t) {

unsigned int j = t->sizearray;

if (j > 0 && ttisnil(&t->array[j - 1])) {

/\* there is a boundary in the array part: (binary) search for it \*/

unsigned int i = 0;

while (j - i > 1) {

unsigned int m = (i+j)/2;

if (ttisnil(&t->array[m - 1])) j = m;

else i = m;

}

return i;

}

/\* else must find a boundary in hash part \*/

else if (isdummy(t)) /\* hash part is empty? \*/

return j; /\* that is easy... \*/

else return unbound\_search(t, j);

}

先看看lua的代码运行的结果

local test1 = { 1, 3 , 5 , 2 , 4 }

print(#test1)-- 5

local test1 = {[1] = 1 , [2] = 2 , [3] = 3 , [4] = 4 ,[5] = 5}

print(#test1)-- 5

local test1 = {[1] = 1 ,[2] = 1, [3] = 1 , [4] = 1 , [6] = 1 }

print(#test1) -- 6 中间[5]没有，但是返回的是6

local test1 = {[4] = 4 , [6] = 6 ,[2] = 2}

print(#test1) -- 0

local test1 = {[1] = 1 , [2] = 2 ,[4] = 4 ,[6] = 6}

print(#test1) -- 6

local test1 = {[1] = 1, [2] = 2 ,[5] = 5 ,[6] = 6}

print(#test1) -- 2

local test1 = { ['a'] = 1, ['b'] = 2 ,['c'] = 3}

print(#test1) -- 0

根据#运算求得的值，与上面函数源码，不难发现其求值的方法。  
当数组部分不连续的时候，用#来求数组的大小是不准确的。

**遍历，pairs ipairs**

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

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

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

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

setivalue(key, i + 1);

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

return 1;

}

}

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

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

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

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

return 1;

}

}

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

}

下面是官网对ipairs 和 pairs的说明文档

ipairs (t)

Returns three values (an iterator function, the table t, and 0) so that the construction

for i,v in ipairs(t) do body end

will iterate over the key–value pairs (1,t[1]), (2,t[2]), ..., up to the first nil value.

------------------

pairs (t)

If t has a metamethod \_\_pairs, calls it with t as argument and returns the first three results from the call.

Otherwise, returns three values: the next function, the table t, and nil, so that the construction

for k,v in pairs(t) do body end

will iterate over all key–value pairs of table t.

See function next for the caveats of modifying the table during its traversal.

可以看到  
ipairs返回三个值：迭代方法、table、0  
pairs也返回三个值：next方法，table，nil

for k,v in pairs(t) do

print(k,v)

end

展开：

for k, v in iter, tab, nil do

body

end

《Programming in Lua》给出的代码是：

do

local \_f,\_s,\_var = iter,tab,var

while true do

local \_var,value = \_f(\_s, \_var) -- 通过上一个key找下一个key

if not \_var then break end

body

end

end

LUA\_API int lua\_next (lua\_State \*L, int idx) {

StkId t;

int more;

lua\_lock(L);

t = index2addr(L, idx);

api\_check(L, ttistable(t), "table expected");

more = luaH\_next(L, hvalue(t), L->top - 1);

if (more) {

api\_incr\_top(L);

}

else /\* no more elements \*/

L->top -= 1; /\* remove key \*/

lua\_unlock(L);

return more;

}

其中上述代码中\_f即为lua\_next, 函数的内部会调用luaH\_next

luaH\_next 每次 传入一个table，一个key值，在迭代方法里，每次通过上一个key值，来找下一个key，直到找到的key值为空时，跳出循环。

初始迭代的key值，分别为0和nil

**删除key**

例如如下的代码，

local a = { key = 1}

a.key = nil

解析成字节码如下：

$ ./luac.exe -l -p a.lua

main <a.lua:0,0> (4 instructions at 0045e840)

0+ params, 2 slots, 1 upvalue, 1 local, 3 constants, 0 functions

1 [2] NEWTABLE 0 0 1

2 [2] SETTABLE 0 -1 -2 ; "key" 1

3 [3] SETTABLE 0 -1 -3 ; "key" nil

4 [3] RETURN 0 1

可以看到执行了SETTABLE这个OP\_CODE，在代码中搜索，最后在lvm.c中找到。

vmcase(OP\_SETTABLE) {

TValue \*rb = RKB(i);

TValue \*rc = RKC(i);

settableProtected(L, ra, rb, rc);

vmbreak;

}

其中宏定义如下:

/\* same for 'luaV\_settable' \*/

#define settableProtected(L,t,k,v) { const TValue \*slot; \

if (!luaV\_fastset(L,t,k,slot,luaH\_get,v)) \

Protect(luaV\_finishset(L,t,k,v,slot)); }

/\*

\*\* Fast track for set table. If 't' is a table and 't[k]' is not nil,

\*\* call GC barrier, do a raw 't[k]=v', and return true; otherwise,

\*\* return false with 'slot' equal to NULL (if 't' is not a table) or

\*\* 'nil'. (This is needed by 'luaV\_finishget'.) Note that, if the macro

\*\* returns true, there is no need to 'invalidateTMcache', because the

\*\* call is not creating a new entry.

\*/

#define luaV\_fastset(L,t,k,slot,f,v) \

(!ttistable(t) \

? (slot = NULL, 0) \

: (slot = f(hvalue(t), k), \

ttisnil(slot) ? 0 \

: (luaC\_barrierback(L, hvalue(t), v), \

setobj2t(L, cast(TValue \*,slot), v), \

1)))

可以看到，最后是将rc这个TValue类型的对象设置进去了，所以a.key = nil这样一行代码并不是简单的将tt\_字段修改，而是修改了整个key值对应的TValue对象。

下面是一些库函数中，设置某个键为nil值的方法。例如table.remove(t, index)

lua\_pushnil 在代码中有很多处引用，基本上都在 xxxlib.c文件中，比如将一个对象转换成数字，会调用到luaB\_tonumber接口，当转换失败之后，会调用lua\_pushnil，操作L （lua\_State对象）的栈顶对象。而会调用到setnilvalue，只是将这个对象的tt\_字段标记为LUA\_TNIL。

#define setnilvalue(obj) settt\_(obj, LUA\_TNIL)

LUA\_API void lua\_pushnil (lua\_State \*L) {

lua\_lock(L);

setnilvalue(L->top);

api\_incr\_top(L);

lua\_unlock(L);

}

**更多table的用法库**

前面介绍的一些源码，大多在文件ltable.c中，下面就介绍一些table使用的其它库函数。  
ltablib.c

static const luaL\_Reg tab\_funcs[] = {

{"concat", tconcat},

#if defined(LUA\_COMPAT\_MAXN)

{"maxn", maxn},

#endif

{"insert", tinsert},

{"pack", pack},

{"unpack", unpack},

{"remove", tremove},

{"move", tmove},

{"sort", sort},

{NULL, NULL}

};

LUAMOD\_API int luaopen\_table (lua\_State \*L) {

luaL\_newlib(L, tab\_funcs);

#if defined(LUA\_COMPAT\_UNPACK)

/\* \_G.unpack = table.unpack \*/

lua\_getfield(L, -1, "unpack");

lua\_setglobal(L, "unpack");

#endif

return 1;

}

看到上面的代码就有种很熟悉的感觉，这样写与 将C/C++函数的接口暴露给lua去调用非常相似：

lua\_State\* L = luaL\_newstate();

luaL\_openlibs(L);

/\* 注册函数 \*/

lua\_register(L, "dosomething", readImpl);

lua\_register(L, "ReadExcel", readExcel);

lua\_register(L, "func\_return\_table", func\_return\_table);

可以猜测，上述代码是实现table.remove之类的功能的。实际的实现方式以注册函数的方式来实现这些方法的。  
当然自己也可以修改这些函数，以实现自己的需求。定制lua源码。

下面简单介绍下lua的栈。更多请参考[博文](https://blog.csdn.net/zhuzhuyule/article/details/41086745)

lua的虚拟机是一个栈，在与C++交互的时候，用的就是一个栈。  
栈底的元素索引到栈顶，是123456... ,而从栈顶索引到栈底，是-1,-2,-3...

栈顶

5 "e" -1

4 "d" -2

3 "c" -3

2 "b" -4

1 "a" -5

栈底

table.remove(t, key) 要求key必须为number类型

static int tremove (lua\_State \*L) {

lua\_Integer size = aux\_getn(L, 1, TAB\_RW);

lua\_Integer pos = luaL\_optinteger(L, 2, size);

if (pos != size) /\* validate 'pos' if given \*/

luaL\_argcheck(L, 1 <= pos && pos <= size + 1, 1, "position out of bounds");

lua\_geti(L, 1, pos); /\* result = t[pos] \*/

for ( ; pos < size; pos++) {

lua\_geti(L, 1, pos + 1);

lua\_seti(L, 1, pos); /\* t[pos] = t[pos + 1] \*/

}

lua\_pushnil(L);

lua\_seti(L, 1, pos); /\* t[pos] = nil \*/

return 1;

}

上述函数即为table.remove的实现方法，下面先分析下L这个栈上都有那些数据。  
栈索引1处为table，2处为table.remove的第二个参数，是个integer类型，函数的前两行是获取table的长度和第二个参数的值。  
取得t[pos]的值，放到栈顶，在循环一次，将t[pos] = t[pos + 1]，最后再将t[pos] = nil，完成删除

**参考文章：**

<https://blog.csdn.net/fwb330198372/article/details/88579361>

<http://geekluo.com/contents/2014/04/11/3-lua-table-structure.html>

lua中关于取长度问题  
<https://www.2cto.com/kf/201501/370498.html>

**Chapter04**

* [Chapter04](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#chapter04)
  + [lua的gc算法以及碎片整理](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#lua%E7%9A%84gc%E7%AE%97%E6%B3%95%E4%BB%A5%E5%8F%8A%E7%A2%8E%E7%89%87%E6%95%B4%E7%90%86)
  + [那些值会放到allgc链表中？](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#%E9%82%A3%E4%BA%9B%E5%80%BC%E4%BC%9A%E6%94%BE%E5%88%B0allgc%E9%93%BE%E8%A1%A8%E4%B8%AD)
  + [字符串和userdata](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#%E5%AD%97%E7%AC%A6%E4%B8%B2%E5%92%8Cuserdata)
  + [GC Garbage Collect](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#gc-garbage-collect)
    - [gc初始化阶段](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#gc%E5%88%9D%E5%A7%8B%E5%8C%96%E9%98%B6%E6%AE%B5)
    - [扫描标记阶段](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#%E6%89%AB%E6%8F%8F%E6%A0%87%E8%AE%B0%E9%98%B6%E6%AE%B5)
    - [回收阶段](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#%E5%9B%9E%E6%94%B6%E9%98%B6%E6%AE%B5)
    - [思考？](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#%E6%80%9D%E8%80%83)
  + [参考文章：](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter04.md#%E5%8F%82%E8%80%83%E6%96%87%E7%AB%A0)

**lua的gc算法以及碎片整理**

* lua 5.3 使用mark-sweep（后文会介绍 三色增量标记法）
* lua 的gc算法并不做内存整理

Cloud：  
lua 的 GC 算法并不做内存整理，它不会在内存中迁移数据。实际上，如果你能肯定一个 string 不会被清除，那么它的内存地址也是不变的，这样就带来的优化空间。ltm.c 中就是这样做的。  
评论：lua 中的内存碎片问题可以通过定制内存分配器解决。对于数据类型很少的 lua ，大多数内存块尺寸都是非常规则的。

关于这个问题，可以参考 [云风的博客](https://blog.codingnow.com/2011/03/lua_gc_2.html)  
同时关于Mark-Sweep的优化方法(mark-compact等)，参考[博客](https://liujiacai.net/blog/2018/07/08/mark-sweep/)

**那些值会放到allgc链表中？**

luaC\_newobj根据这个函数，全局搜索其调用的地方，可以发现，当new一个以下类型的对象时，会被连接到链表表头：

- string

- table

- userdata UserData在lua中和string类似，可以看成是拥有独立元表，不被内部化，也不需要追加\0的字符串

- proto 函数原型数据结构

- CClosure c函数闭包

- LClosure lua 闭包

// creates a new string object lstring.c

static TString \*createstrobj (lua\_State \*L, size\_t l, int tag, unsigned int h) {

TString \*ts;

GCObject \*o;

size\_t totalsize; /\* total size of TString object \*/

totalsize = sizelstring(l);

o = luaC\_newobj(L, tag, totalsize);

ts = gco2ts(o); // gc object to TString

ts->hash = h;

ts->extra = 0;

getstr(ts)[l] = '\0'; /\* ending 0 \*/

return ts;

}

// userdata lstring.c

Udata \*luaS\_newudata (lua\_State \*L, size\_t s) {

Udata \*u;

GCObject \*o;

if (s > MAX\_SIZE - sizeof(Udata))

luaM\_toobig(L);

o = luaC\_newobj(L, LUA\_TUSERDATA, sizeludata(s));

u = gco2u(o);

u->len = s;

u->metatable = NULL;

setuservalue(L, u, luaO\_nilobject);

return u;

}

// 创建一个空table ltable.c

Table \*luaH\_new (lua\_State \*L) {

GCObject \*o = luaC\_newobj(L, LUA\_TTABLE, sizeof(Table));

Table \*t = gco2t(o);

t->metatable = NULL;

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

t->array = NULL;

t->sizearray = 0;

setnodevector(L, t, 0);

return t;

}

// c函数闭包 lfunc.c

CClosure \*luaF\_newCclosure (lua\_State \*L, int n) {

GCObject \*o = luaC\_newobj(L, LUA\_TCCL, sizeCclosure(n));

CClosure \*c = gco2ccl(o);

c->nupvalues = cast\_byte(n);

return c;

}

// lua 闭包 lfunc.c

LClosure \*luaF\_newLclosure (lua\_State \*L, int n) {

GCObject \*o = luaC\_newobj(L, LUA\_TLCL, sizeLclosure(n));

LClosure \*c = gco2lcl(o);

c->p = NULL;

c->nupvalues = cast\_byte(n);

while (n--) c->upvals[n] = NULL;

return c;

}

// 函数原型数据结构 lfunc.c

Proto \*luaF\_newproto (lua\_State \*L) {

GCObject \*o = luaC\_newobj(L, LUA\_TPROTO, sizeof(Proto));

Proto \*f = gco2p(o);

f->k = NULL;

f->sizek = 0;

f->p = NULL;

f->sizep = 0;

f->code = NULL;

f->cache = NULL;

f->sizecode = 0;

f->lineinfo = NULL;

f->sizelineinfo = 0;

f->upvalues = NULL;

f->sizeupvalues = 0;

f->numparams = 0;

f->is\_vararg = 0;

f->maxstacksize = 0;

f->locvars = NULL;

f->sizelocvars = 0;

f->linedefined = 0;

f->lastlinedefined = 0;

f->source = NULL;

return f;

}

**字符串和userdata**

TString下面，又封装了一层

typedef struct TString {

CommonHeader;

lu\_byte extra; /\* reserved words for short strings; "has hash" for longs \*/

lu\_byte shrlen; /\* length for short strings \*/

unsigned int hash;

union {

size\_t lnglen; /\* length for long strings \*/

struct TString \*hnext; /\* linked list for hash table \*/

} u;

} TString;

/\*

\*\* Ensures that address after this type is always fully aligned.

\*/

typedef union UTString {

L\_Umaxalign dummy; /\* ensures maximum alignment for strings \*/

TString tsv;

} UTString;

类似的，对于userdata类型，也有一个对应的结构，

typedef struct Udata {

CommonHeader;

lu\_byte ttuv\_; /\* user value's tag \*/

struct Table \*metatable;

size\_t len; /\* number of bytes \*/

union Value user\_; /\* user value \*/

} Udata;

typedef union UUdata {

L\_Umaxalign dummy; /\* ensures maximum alignment for 'local' udata \*/

Udata uv;

} UUdata;

UserData在lua中和string类似，可以看成是拥有独立元表，不被内部化，也不需要追加\0的字符串

再看获取字符串和userdata那块内存的宏。

/\*

\*\* Get the actual string (array of bytes) from a 'TString'.

\*\* (Access to 'extra' ensures that value is really a 'TString'.)

\*/

#define getstr(ts) \

check\_exp(sizeof((ts)->extra), cast(char \*, (ts)) + sizeof(UTString))

/\*

\*\* Get the address of memory block inside 'Udata'.

\*\* (Access to 'ttuv\_' ensures that value is really a 'Udata'.)

\*/

#define getudatamem(u) \

check\_exp(sizeof((u)->ttuv\_), (cast(char\*, (u)) + sizeof(UUdata)))

两者的类型很相似。其中dummy字段，是用来保证最大程度的内存对齐

**GC Garbage Collect**

iscollectable这个宏，用来检查一个TValue对象是否被标记为可以回收

/\* raw type tag of a TValue \*/

#define rttype(o) ((o)->tt\_)

// 这个是看tag的第六位是不是1，是1的话就属于垃圾回收，否则就不需要关心它的生命周期

/\* Bit mark for collectable types \*/

#define BIT\_ISCOLLECTABLE (1 << 6)

#define iscollectable(o) (rttype(o) & BIT\_ISCOLLECTABLE)

// 检查obj的生存期

// iscollectable(obj)检查obj是否为GC对象

// righttt(obj)返回obj的tt\_是否等于gc里面的tt

// isdead(obj)返回obj是否已经被清理

// 总而言之，返回true代表未被GC的和不需要GC的，返回false代表已经被GC了的

#define checkliveness(L,obj) \

lua\_longassert(!iscollectable(obj) || \

(righttt(obj) && (L == NULL || !isdead(G(L),gcvalue(obj)))))

从lua5.1开始，使用了三色增量标记清除算法。  
它不必在要求GC一次性扫描完所有的对象，这个GC过程可以是增量的，可以被终止再恢复并继续进行

伪代码：

每个新创建的对象标记为白色

// 初始化阶段

遍历root节点中引用的对象，从白色置为灰色，并放入灰色节点列表中

// 标记阶段

当灰色链表中海油未扫描的元素：

取出一个对象标记为黑色

遍历这个对象关联的其它所有对象：

如果是白色：

标记为灰色，加入灰色链表中

// 回收阶段

遍历所有对象：

如果是白色：

这些对象都是没有引用的对象，回收

否则：

重新加入对象链表中等待下一轮GC

那么这样会有一个问题，没有被引用的对象在扫描过程之中颜色不变，如果一个对象在gc过程标记阶段之后创建，它应该是白色，这样在回收阶段，这个对象就会被认为没有引用而被回收掉。

所以lua又细分出来一个“双白色”的概念。当前白色（currentwhite）和 非当前白色（otherwhite）。这两种白色交替使用。

在回收阶段，会判断某个对象的白色是不是这次gc的标记白色，否则会不回收这个对象。

**gc初始化阶段**

lua的gc过程是增量的，中间可以被打断的，所以每次单独进入gc的是，都会根据当前gc的所处的阶段来进行不同的出来，函数的入口是singlestep在lgc.c中  
lua53中，初始化阶段的入口函数为restartcollection, 会调用到reallymarkobject函数来标记节点为灰色。

* 对于字符串类型，由于字符串没有引用其他结构，所以略过标记为灰色，直接标记为黑色。
* 对于udata类型，这种类型也不会引用其他类型，所以标记为黑色，对于这种类型还要标记对应的元表

注意，这里没有对对象所引用的对象进行递归调用reallymarkobject函数进行标记，比如table类型递归遍历key和value，原因是希望这个标记过程尽量快。

**扫描标记阶段**

该阶段就是遍历灰色对象链表，来分析对象的引用情况，这个过程最长。 函数propagatemark，这步将gray链表中的对象以及其引用到的对象标记为黑色。

上一步是一次到位的，而这一步却可以多次进行，每次扫描之后会返回本次扫描标记的对象大小之和

**回收阶段**

entersweep函数，如果是当前白色，那么就回收，否则就改变所有对象的标记为白色，准备下一次回收过程。

freeobj 函数，释放掉o对象的内存空间，根据o的不同类型，执行不同的释放内存的方法

static void freeobj (lua\_State \*L, GCObject \*o) {

switch (o->tt) {

case LUA\_TPROTO: luaF\_freeproto(L, gco2p(o)); break;

case LUA\_TLCL: {

freeLclosure(L, gco2lcl(o));

break;

}

case LUA\_TCCL: {

luaM\_freemem(L, o, sizeCclosure(gco2ccl(o)->nupvalues));

break;

}

case LUA\_TTABLE: luaH\_free(L, gco2t(o)); break;

case LUA\_TTHREAD: luaE\_freethread(L, gco2th(o)); break;

case LUA\_TUSERDATA: luaM\_freemem(L, o, sizeudata(gco2u(o))); break;

case LUA\_TSHRSTR:

luaS\_remove(L, gco2ts(o)); /\* remove it from hash table \*/

luaM\_freemem(L, o, sizelstring(gco2ts(o)->shrlen));

break;

case LUA\_TLNGSTR: {

luaM\_freemem(L, o, sizelstring(gco2ts(o)->u.lnglen));

break;

}

default: lua\_assert(0);

}

}

**思考？**

1. 在删除一个key的时候，为什么不能增加一个新的key？数组部分或者hash部分大小要变化？
2. 当一个字符串key，在table里对应的value置空，那么key有释放吗？

local t { "ddd" = 111}

t["ddd"] = nil

当执行xxx = nil的时候，table的hash部分，这个key值依旧存在，当插入一个新节点，如果计算出来的mainposition值相同，那么会覆盖掉，另外在触发resize的时候，也会释放掉这个空的value对应的key值  
两个值指向同一个地址

a = t.ddd

**参考文章：**

<https://github.com/lichuang/Lua-Source-Internal>

<https://wenku.baidu.com/view/c96a0e1055270722192ef772.html>

[《Lua设计与实现》](https://book.douban.com/subject/27108476/)

<https://www.lua.org/wshop18/Ierusalimschy.pdf>

<https://blog.codingnow.com/2011/03/lua_gc_2.html>

<http://www.zenyuhao.com/2017/10/13/lua-gc.html>

<https://www.e-learn.cn/content/qita/909901>

<https://liujiacai.net/blog/2018/08/04/incremental-gc/> 深入浅出垃圾回收（三）增量式 GC

<https://liujiacai.net/blog/2018/07/08/mark-sweep/> 深入浅出垃圾回收（二）Mark-Sweep 详析及其优化

<https://blog.codingnow.com/2011/04/lua_gc_6.html> Lua GC 的源码剖析 (6) 完结(string的gc细节)

[https://chenanbao.github.io/2018/07/27/Lua虚拟机创建分析/](https://chenanbao.github.io/2018/07/27/Lua%E8%99%9A%E6%8B%9F%E6%9C%BA%E5%88%9B%E5%BB%BA%E5%88%86%E6%9E%90/)

**Chapter05 GC详述**

* [三色标记的概念](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#%E4%B8%89%E8%89%B2%E6%A0%87%E8%AE%B0%E7%9A%84%E6%A6%82%E5%BF%B5)
* [barrier](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#barrier)
* [全局状态机](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#%E5%85%A8%E5%B1%80%E7%8A%B6%E6%80%81%E6%9C%BA)
* [GC流程](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#%E8%AF%A6%E7%BB%86%E7%AE%97%E6%B3%95)
* [阶段](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#%E9%98%B6%E6%AE%B5)
* [详细算法解析](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#%E8%AF%A6%E7%BB%86%E7%AE%97%E6%B3%95%E8%A7%A3%E6%9E%90)
* [手动GC](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#collectgarbage)
* [思考](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#%E6%80%9D%E8%80%83)
* [内存工具](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter05.md#lua%E5%86%85%E5%AD%98%E5%88%86%E6%9E%90%E5%B7%A5%E5%85%B7)

[弱表的概念](https://www.runoob.com/manual/lua53doc/manual.html#2.5.2)

**Tri-Color Incremental Mark & Sweep**

* 三色增量标记法状态变化图

**三色标记的概念**

* White:表示当前对象为待访问状态,用于表示对象还没有被GC的标记过,这也是任何一个Lua对象在创建之后的初始状态,换言之,如果一个对象,在一个GC扫描过程完毕之后,仍然是白色的,那么说明该对象没有被系统中任何一个对象所引用,可以回收其空间了.
* Gray:表示当前对象为待扫描状态,用于表示对象已经被GC访问过,但是该对象引用的其他对象还没有被访问到.
* Black:表示当前对象为已扫描状态,用于表示对象已经被GC访问过,并且该对象引用的其他对象也已经被访问过了.

那么这样会有一个问题，没有被引用的对象在扫描过程之中颜色不变，如果一个对象在gc过程标记阶段之后创建，它应该是白色，这样在回收阶段，这个对象就会被认为没有引用而被回收掉。

所以lua又细分出来一个“双白色”的概念。当前白色currentwhite和 非当前白色otherwhite，这两种白色交替使用。  
我们来看代码中是如何区分的:

可以看到初始化lua\_newstate的时候currentwhite为，也就是二进制值01

//lstate.c

g->currentwhite = bitmask(WHITE0BIT);

#define bitmask(b) (1<<(b))

// lgc.h

/\* Layout for bit use in 'marked' field: \*/

#define WHITE0BIT 0 /\* object is white (type 0) \*/

#define WHITE1BIT 1 /\* object is white (type 1) \*/

#define BLACKBIT 2 /\* object is black \*/

#define FINALIZEDBIT 3 /\* object has been marked for finalization \*/

我们可以看到，白色有两个宏定义，黑色只有一个宏定义，而灰色则一个宏定义都没有，这是因为在lua官方的实现版本中，白色有两种，在不同的gc轮之间执行乒乓交换，而既不是白色也不是黑色则被视为是灰色，因此没有对灰色进行直接定义。最下面的WHITEBITS则是非常关键的存在，因为它是我们切换白色、判断对象是否dead以及标记对象为白色的重要参数，根据bit2mask的定义，WHITEBITS实际相当于被这样定义：

// lgc.h

#define WHITEBITS ((1<<0) | (1<<1)) // 01 | 11 --> 11

也就是说，WHITEBITS最后相当于二进制值11，现在我们可以来看一下标记一个对象为白色的接口是怎样的：

// lgc.h

#define luaC\_white(g) (g->currentwhite & WHITEBITS) // 01 & 11 --> 01

因为上面说了g->currentwhite在初始化阶段，被赋值为01值，由于WHITEBITS是11，01 & 11还是01能够获得当前白色的值。而切换不同白的宏定义，则是这样的：

// lgc.h

#define otherwhite(g) (g->currentwhite ^ WHITEBITS) // 01 ^ 11 --> 10

这里是一个异或操作，也就是说如果g->currentwhite的值是01的话，通过otherwhite计算，则是01 ^ 11 = 10，如果g->currentwhite的值是10的话，结果正好相反。  
乒乓切换后当前白色为10时，结果如下：

// luaC\_white(g) (g->currentwhite & WHITEBITS) // 10 & 11 --> 10

// otherwhite(g) (g->currentwhite ^ WHITEBITS) // 10 ^ 11 --> 01

从这里的逻辑我们可以看出，white的值只有两种，要么是01，要么是10

**barrier**

在每个步骤之间，由于程序可以正常执行，所以会破坏当前对象之间的引用关系。 Black对象表示已经被扫描的对象，所以他应该不可能引用到一个white对象。当程序的改变使得一个black对象引用到一个white对象时，就会造成错误。

增量gc在mark阶段，为了保证"所有的black对象不会引用white对象"这个不变性，需要使用barrier

* barrier在程序正常运行过程中，监控所有的引用改变。如果一个black对象需要引用一个white对象，存在两种处理办法：
  + barrier forward 将white对象设置成gray，并添加到gray列表中等待扫描。这样等于帮助整个GC的标识过程向前推进了一步。luaC\_barrier\_
  + barrier back 将black对象该回成gray,并添加到gray列表中等待扫描.这样等于使整个GC的标识过程后退了一步。 luaC\_barrierback\_

//lgc.c

void luaC\_barrier\_ (lua\_State \*L, GCObject \*o, GCObject \*v) {

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

lua\_assert(isblack(o) && iswhite(v) && !isdead(g, v) && !isdead(g, o));

if (keepinvariant(g)) /\* must keep invariant? \*/

reallymarkobject(g, v); /\* restore invariant \*/

else { /\* sweep phase \*/

lua\_assert(issweepphase(g));

makewhite(g, o); /\* mark main obj. as white to avoid other barriers \*/

}

}

// lgc.h

#define keepinvariant(g) ((g)->gcstate <= GCSatomic) //在GCSatomic原子阶段前才会有效

可以看到luaC\_barrier\_是在原子阶段前才会把被黑色对象引用到的白色对象标记为灰色

// lgc.h

#define luaC\_barrierback(L,p,v) ( \

(iscollectable(v) && isblack(p) && iswhite(gcvalue(v))) ? \

luaC\_barrierback\_(L,p) : cast\_void(0))

通过搜索luaC\_barrierback的引用可以看到，当设置table的时候会进行barrier back的检查

从而保证增量式GC在GC流程中暂停时，对象引用状态的改变不会引起GC流程产生错误的结果。

这样增量GC所检测出来的垃圾对象集合比实际的集合要小，也就是说,有些在GC过程中变成垃圾的对象，有可能在本轮GC中检测不到。不过，这些残余的垃圾对象一定会在下一轮GC被检测出来，不会造成泄露。

**全局状态机**

* global\_state中gc相关的字段  
  totalbytes：实际内存分配器所分配的内存与GCdebt的差值。 真实的大小是totalbytes+GCdebt gettotalbytes  
  GCdebt:需要回收的内存数量；可以为负数的变量，主要用于控制gc触发时机，大于0时才能触发gc luaC\_condGC luaM\_realloc\_ 精确统计内存大小  
  GCmemtrav：内存实际使用量的估计值；每次进行gc操作时，所遍历的对象字节大小之和，单位是byte，当其值大于单步执行的内存上限时，gc终止  
  GCestimate：在sweep阶段结束时，会被重新计算，本质是totalbytes+GCdebt，它的作用是，在本轮gc结束时，将自身扩充两倍大小，然后让真实大小减去扩充后的自己得到差debt，然后totalbytes会等于扩充后的自己，而GCdebt则会被负数debt赋值，就是是说下一次执行gc流程，要在有|debt|个bytes内存被开辟后，才会开始。目的是避免gc太过频繁。  
  currentwhite:上面详细解释过，当前gc的白色状态 10和01中的一种，在atomic阶段最后切换状态  
  gcstate:gc的状态，定义在lua.h中  
  allgc:单项链表，新建gc对象都要放到这个链表中，放入的方式是链到表的头部  
  sweepgc:当前sweep的进度  
  gray:初次转换为gary的对象都会加入到gary链表中  
  grayagain:前面已经介绍过，当被标记为black的对象重新指向white对象时，进行barrier会放入到grayagain链表中  
  gcpause: gc间隔  
  gcstepmul:gc的速率，下面详细讲

void \*luaM\_realloc\_ (lua\_State \*L, void \*block, size\_t osize, size\_t nsize) {

void \*newblock;

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

size\_t realosize = (block) ? osize : 0;

lua\_assert((realosize == 0) == (block == NULL));

#if defined(HARDMEMTESTS)

if (nsize > realosize && g->gcrunning)

luaC\_fullgc(L, 1); /\* force a GC whenever possible \*/

#endif

newblock = (\*g->frealloc)(g->ud, block, osize, nsize);

if (newblock == NULL && nsize > 0) {

lua\_assert(nsize > realosize); /\* cannot fail when shrinking a block \*/

if (g->version) { /\* is state fully built? \*/

luaC\_fullgc(L, 1); /\* try to free some memory... \*/

newblock = (\*g->frealloc)(g->ud, block, osize, nsize); /\* try again \*/

}

if (newblock == NULL)

luaD\_throw(L, LUA\_ERRMEM);

}

lua\_assert((nsize == 0) == (newblock == NULL));

g->GCdebt = (g->GCdebt + nsize) - realosize;

return newblock;

}

由luaM\_realloc\_最后的 g->GCdebt可知，GCdebt就是在不断的统计释放与分配的内存。

* 当新增分配内存时，GCdebt值将会增加，即GC需要释放的内存增加；
* 当释放内存时，GCdebt将会减少，即GC需要释放的内存减少。

那些值会放到allgc链表中？

这里要说明下，thread永远是灰色的

/\* lgc.c propagatemark

\*\* traverse one gray object, turning it to black (except for threads,

\*\* which are always gray).

\*/

lua\_state本质就是一个LUA\_TTHREAD本质上没有什么特殊性

1. 和nil, string, table一样, lua\_State也是lua中的一种基本类型, lua中的表示是TValue {value = lua\_State, tt = LUA\_TTHREAD}
2. lua\_State的成员和功能

* 栈的管理, 包括管理整个栈和当前函数使用的栈的情况.
* CallInfo的管理, 包括管理整个CallInfo数组和当前函数的CallInfo.
* hook相关的, 包括hookmask, hookcount, hook函数等.
* 全局表l\_gt, 注意这个变量的命名, 很好的表现了它其实只是在本lua\_State范围内是全唯一的的, 和注册表不同, 注册表是lua虚拟机范围内是全局唯一的.
* gc的一些管理和当前栈中upvalue的管理.
* 错误处理的支持.

**阶段**

/\*lgc.h

\*\* Possible states of the Garbage Collector

\*/

#define GCSpropagate 0

#define GCSatomic 1

#define GCSswpallgc 2

#define GCSswpfinobj 3

#define GCSswptobefnz 4

#define GCSswpend 5

#define GCScallfin 6

#define GCSpause 7

GCSpause: GC cycle的初始化过程；一步完成。

GCSpropagate: 可以分多次执行，直到 gray 链表处理完，进入 GCSatomic

GCSatomic: 一次性的处理所有需要回顾一遍的地方, 保证一致性, 然后进入清理阶段,注意这个过程不可以再被打断(原子阶段)

GCSswpallgc: 清理 allgc 链表

GCSswpfinobj: 清理 finobj 链表

GCSswptobefnz: 清理 tobefnz 链表

GCSswpend: sweep main thread

GCScallfin: 执行一些 finalizer (\_\_gc) 完成循环

注意propagate 和各个sweep阶段都是可以每次执行一点，多次执行直到完成的，所以是增量式 gc, 增量式过程中依靠 barrier 来保证一致性，上面对barrier已经详细介绍过了这里不再赘述。

**详细算法**

伪代码：

每个新创建的对象标记为白色

// 初始化阶段

遍历root节点中引用的对象，从白色置为灰色，并放入灰色节点列表中

// 标记阶段

当灰色链表中还有未扫描的元素：

取出一个对象标记为黑色

遍历这个对象关联的其它所有对象：

如果是白色：

标记为灰色，加入灰色链表中

// 回收阶段

遍历所有对象：

如果是白色：

这些对象都是没有引用的对象，回收

否则：

重新加入对象链表中等待下一轮GC

* 详细算法流程图

**详细算法解析**

1. 新建可回收对象，将其置为白色

/\*lgc.c

\*\* create a new collectable object (with given type and size) and link

\*\* it to 'allgc' list.

\*/

GCObject \*luaC\_newobj (lua\_State \*L, int tt, size\_t sz) {

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

GCObject \*o = cast(GCObject \*, luaM\_newobject(L, novariant(tt), sz));

o->marked = luaC\_white(g); // 初始化GC对象都为white

o->tt = tt;

o->next = g->allgc; // 把gc对象放到globa\_State allgc 链表中

g->allgc = o;

return o;

}

1. 何时触发GC

// lgc.h

/\*

\*\* Does one step of collection when debt becomes positive. 'pre'/'pos'

\*\* allows some adjustments to be done only when needed. macro

\*\* 'condchangemem' is used only for heavy tests (forcing a full

\*\* GC cycle on every opportunity)

\*/

// 可以看到当 GCdebt 大于0是才会尝试GC操作，而GC的工作流都是在luaC\_step中执行的

#define luaC\_condGC(L,pre,pos) \

{ if (G(L)->GCdebt > 0) { pre; luaC\_step(L); pos;}; \

condchangemem(L,pre,pos); }

/\* more often than not, 'pre'/'pos' are empty \*/

#define luaC\_checkGC(L) luaC\_condGC(L,(void)0,(void)0)

我们来看下luaC\_step这个函数

/\*

\*\* performs a basic GC step when collector is running

\*/

void luaC\_step (lua\_State \*L) {

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

l\_mem debt = getdebt(g); /\* GC deficit (be paid now) \*/

if (!g->gcrunning) { /\* not running? \*/

luaE\_setdebt(g, -GCSTEPSIZE \* 10); /\* avoid being called too often \*/

return;

}

do { /\* repeat until pause or enough "credit" (negative debt) \*/

lu\_mem work = singlestep(L); /\* perform one single step \*/

debt -= work;

} while (debt > -GCSTEPSIZE && g->gcstate != GCSpause);

// 从这里可以看出当debt小于 GCSTEPSIZE 时那么GC将是一步到位执行完毕的

if (g->gcstate == GCSpause)

setpause(g); /\* pause until next cycle \*/

else {

debt = (debt / g->gcstepmul) \* STEPMULADJ; /\* convert 'work units' to Kb \*/

luaE\_setdebt(g, debt);

runafewfinalizers(L);

}

}

从这里可以看出luaC\_step会根据debt的值（受设置的stepmul的影响）执行多步singlestep

singlestep状态机

1. 从根节点开始标记，将白色对象置为灰色，并加入到灰色链表中

//lgc.c

switch (g->gcstate) {

case GCSpause: {

g->GCmemtrav = g->strt.size \* sizeof(GCObject\*);

restartcollection(g); // 注意这个函数，重启一次gc，重置所有的灰色链表

g->gcstate = GCSpropagate;

return g->GCmemtrav;

}

其实就是restartcollection来完成的

/\*lgc.c

\*\* mark root set and reset all gray lists, to start a new collection

\*/

static void restartcollection (global\_State \*g) {

g->gray = g->grayagain = NULL;

g->weak = g->allweak = g->ephemeron = NULL;

markobject(g, g->mainthread);

markvalue(g, &g->l\_registry);

markmt(g);

markbeingfnz(g); /\* mark any finalizing object left from previous cycle \*/

}

1. 持续遍历对象的关联对象把灰色对象置为黑色对象

// lgc.c

case GCSpropagate: {

g->GCmemtrav = 0;

lua\_assert(g->gray);

propagatemark(g); // 持续遍历对象的关联对象 gray to black

if (g->gray == NULL) /\* no more gray objects? \*/

g->gcstate = GCSatomic; /\* finish propagate phase \*/

return g->GCmemtrav; /\* memory traversed in this step \*/

}

由函数propagatemark来完成:

/\*

\*\* traverse one gray object, turning it to black (except for threads,

\*\* which are always gray).

\*/

static void propagatemark (global\_State \*g) {

lu\_mem size;

GCObject \*o = g->gray;

lua\_assert(isgray(o));

gray2black(o);

switch (o->tt) {

case LUA\_TTABLE: {

Table \*h = gco2t(o);

g->gray = h->gclist; /\* remove from 'gray' list \*/

size = traversetable(g, h);

break;

}

case LUA\_TLCL: {

LClosure \*cl = gco2lcl(o);

g->gray = cl->gclist; /\* remove from 'gray' list \*/

size = traverseLclosure(g, cl);

break;

}

case LUA\_TCCL: {

CClosure \*cl = gco2ccl(o);

g->gray = cl->gclist; /\* remove from 'gray' list \*/

size = traverseCclosure(g, cl);

break;

}

case LUA\_TTHREAD: {

lua\_State \*th = gco2th(o);

g->gray = th->gclist; /\* remove from 'gray' list \*/

linkgclist(th, g->grayagain); /\* insert into 'grayagain' list \*/

black2gray(o);

size = traversethread(g, th);

break;

}

case LUA\_TPROTO: {

Proto \*p = gco2p(o);

g->gray = p->gclist; /\* remove from 'gray' list \*/

size = traverseproto(g, p);

break;

}

default: lua\_assert(0); return;

}

g->GCmemtrav += size;

}

可以看出propagatemark每次只会从链表中取出一个灰色节点对象，并遍历此节点相关的引用节点置为灰色，这样就完成了一次GCSpropagate，这是因为遍历完成一个对象的引用节点开销会很大，lua希望每次GCSpropagate时都只处理一个这样的节点。从而可以减少每次阻塞的时间。

不难看出propagatemark对各个类型的处理，最终都会调用到reallymarkobject

/\*lgc.c

\*\* mark an object. Userdata, strings, and closed upvalues are visited

\*\* and turned black here. Other objects are marked gray and added

\*\* to appropriate list to be visited (and turned black) later. (Open

\*\* upvalues are already linked in 'headuv' list.)

\*/

// 时间复杂度是O(1) 不会递归标记相关对象

// O(1)使得标记过程可以均匀分摊在逐个短小的时间片中，不至于停留太长时间

static void reallymarkobject (global\_State \*g, GCObject \*o) {

reentry:

white2gray(o); //首先通过宏来标记为灰色

// 下面再根据具体的对象类型，当一个对象的所有关联的对象都被标记后，再从灰色转化为黑色

switch (o->tt) {

// 对于

case LUA\_TSHRSTR: {

gray2black(o);

g->GCmemtrav += sizelstring(gco2ts(o)->shrlen);

break;

}

case LUA\_TLNGSTR: {

gray2black(o);

g->GCmemtrav += sizelstring(gco2ts(o)->u.lnglen);

break;

}

//标记LUA\_TUSERDATA的原表和

case LUA\_TUSERDATA: {

TValue uvalue;

markobjectN(g, gco2u(o)->metatable); /\* mark its metatable \*/

gray2black(o);

g->GCmemtrav += sizeudata(gco2u(o));

getuservalue(g->mainthread, gco2u(o), &uvalue); // 把o的值给uvalue

if (valiswhite(&uvalue)) { /\* markvalue(g, &uvalue); \*/

o = gcvalue(&uvalue); // 获取uvalue并赋值给o

goto reentry;

}

break;

}

case LUA\_TLCL: {

linkgclist(gco2lcl(o), g->gray);

break;

...

}

}

1. 对灰色链表进行一次清除，且保证是原子操作

// lgc.c

case GCSatomic: {

lu\_mem work;

propagateall(g); /\* make sure gray list is empty \*/

work = atomic(L); /\* work is what was traversed by 'atomic' \*/

entersweep(L);

g->GCestimate = gettotalbytes(g); /\* first estimate \*/;

return work;

}

//lgc.c

static void propagateall (global\_State \*g) {

while (g->gray) propagatemark(g);

}

// lgc.c

// 一次性的将 grayagain 链表中的所有对象扫描和标记

static l\_mem atomic (lua\_State \*L) {

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

l\_mem work;

GCObject \*origweak, \*origall;

GCObject \*grayagain = g->grayagain; /\* save original list \*/

...

work += g->GCmemtrav; /\* stop counting (objects being finalized) \*/

...

g->currentwhite = cast\_byte(otherwhite(g)); /\* flip current white \*/

work += g->GCmemtrav; /\* complete counting \*/

return work; /\* estimate of memory marked by 'atomic' \*/

}

atomic函数主要做的事情：

* 重新遍历根对象
* 遍历grayagain列表
  + black objects got in a write barrier;
  + all kinds of weak tables during propagation phase;
  + all threads.
* 调用separatetobefnz函数不可达的(白色)对象放到tobefnz中,留待以后清理
* 将当前白色值切换到新一轮的白色值(前面说的乒乓切换就只在这里做的)

1. 清除阶段

* 对前面不同的链表进行清除操作
* 释放对象所占的内存
* 将对象颜色置为白
* GCSswpallgc将通过sweepstep将allgc上的所有对象释放并将活对象重新标记为当前白色值
* GCSswpfinobj和GCSswptobefnz两个状态也调用了sweepstep函数。但是finobj和tobefnz链表上是不可能有死对象的，作用仅仅是将这些对象重新设置为新一轮的白色
* GCSswpend用来释放mainthread上的一些空间，调整字符串hash桶大小

// lgc.c

static lu\_mem sweepstep (lua\_State \*L, global\_State \*g,

int nextstate, GCObject \*\*nextlist) {

if (g->sweepgc) {

l\_mem olddebt = g->GCdebt;

g->sweepgc = sweeplist(L, g->sweepgc, GCSWEEPMAX); //清除操作

g->GCestimate += g->GCdebt - olddebt; /\* update estimate \*/

if (g->sweepgc) /\* is there still something to sweep? \*/

return (GCSWEEPMAX \* GCSWEEPCOST);

}

/\* else enter next state \*/

g->gcstate = nextstate;

g->sweepgc = nextlist;

return 0;

}

1. GCScallfin状态

// lgc.c

case GCScallfin: { /\* call remaining finalizers \*/

if (g->tobefnz && g->gckind != KGC\_EMERGENCY) {

int n = runafewfinalizers(L);

return (n \* GCFINALIZECOST);

}

else { /\* emergency mode or no more finalizers \*/

g->gcstate = GCSpause; /\* finish collection \*/

return 0;

}

}

runafewfinalizers逐个取出tobefnz链表上的对象，然后调用其\_\_gc函数，并将其放入allgc链表中，准备在下个GC回收

\_\_gc也很有意思，在lua中叫做finalizer终结器，我们可以看到lua只有在设置原表的时候才会设置\_\_gc方法。luaC\_checkfinalizer函数检查表中是否有\_\_gc方法，如果有，则将对象从allgc链中移到了finobj链中。  
还有一点需要注意的是当设置原表的时候\_\_gc方法就必须存在了，否则在后面再加也是不生效的。  
大致的流程是：

lua\_setmetatable separatetobefnz GCTM

allgc -------------------> finobj -------------------> tobefnz -------------------> allgc

// lgc.c

static void GCTM (lua\_State \*L, int propagateerrors) {

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

const TValue \*tm;

TValue v;

setgcovalue(L, &v, udata2finalize(g));

tm = luaT\_gettmbyobj(L, &v, TM\_GC);

if (tm != NULL && ttisfunction(tm)) { /\* is there a finalizer? \*/

int status;

lu\_byte oldah = L->allowhook;

int running = g->gcrunning;

L->allowhook = 0; /\* stop debug hooks during GC metamethod \*/

g->gcrunning = 0; /\* avoid GC steps \*/

setobj2s(L, L->top, tm); /\* push finalizer... \*/

setobj2s(L, L->top + 1, &v); /\* ... and its argument \*/

L->top += 2; /\* and (next line) call the finalizer \*/

L->ci->callstatus |= CIST\_FIN; /\* will run a finalizer \*/

status = luaD\_pcall(L, dothecall, NULL, savestack(L, L->top - 2), 0);

L->ci->callstatus &= ~CIST\_FIN; /\* not running a finalizer anymore \*/

L->allowhook = oldah; /\* restore hooks \*/

g->gcrunning = running; /\* restore state \*/

if (status != LUA\_OK && propagateerrors) { /\* error while running \_\_gc? \*/

if (status == LUA\_ERRRUN) { /\* is there an error object? \*/

const char \*msg = (ttisstring(L->top - 1))

? svalue(L->top - 1)

: "no message";

luaO\_pushfstring(L, "error in \_\_gc metamethod (%s)", msg);

status = LUA\_ERRGCMM; /\* error in \_\_gc metamethod \*/

}

luaD\_throw(L, status); /\* re-throw error \*/

}

}

}

**collectgarbage**

lua53对collectgarbage函数的说明  
collectgarbage([opt [, arg]])

"collect": 做一次完整的垃圾收集循环。 这是默认选项。

"stop": 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。

"restart": 重启垃圾收集器的自动运行。

"count": 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。

"step": 单步运行垃圾收集器。 步长“大小”由 arg 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。 如果收集器结束一个循环将返回 true 。

"setpause": 将 arg 设为收集器的 间歇率 返回 间歇率 的前一个值。

"setstepmul": 将 arg 设为收集器的 步进倍率 返回 步进倍率 的前一个值。

"isrunning": 返回表示收集器是否在工作的布尔值 （即未被停止）

对应到c中的代码其实就是luaB\_collectgarbage函数

//lbaselib.c

static int luaB\_collectgarbage (lua\_State \*L) {

static const char \*const opts[] = {"stop", "restart", "collect",

"count", "step", "setpause", "setstepmul",

"isrunning", NULL};

static const int optsnum[] = {LUA\_GCSTOP, LUA\_GCRESTART, LUA\_GCCOLLECT,

LUA\_GCCOUNT, LUA\_GCSTEP, LUA\_GCSETPAUSE, LUA\_GCSETSTEPMUL,

LUA\_GCISRUNNING};

int o = optsnum[luaL\_checkoption(L, 1, "collect", opts)];

int ex = (int)luaL\_optinteger(L, 2, 0);

int res = lua\_gc(L, o, ex);

switch (o) {

case LUA\_GCCOUNT: {

int b = lua\_gc(L, LUA\_GCCOUNTB, 0);

lua\_pushnumber(L, (lua\_Number)res + ((lua\_Number)b/1024));

return 1;

}

case LUA\_GCSTEP: case LUA\_GCISRUNNING: {

lua\_pushboolean(L, res);

return 1;

}

default: {

lua\_pushinteger(L, res);

return 1;

}

}

}

选项setpause的使用方法：collectgarbage("setpause", 200)，表示当收集器在总使用内存数量达到上次垃圾收集时的两倍时再开启新的收集周期。

//lstate.c

void luaE\_setdebt (global\_State \*g, l\_mem debt) {

l\_mem tb = gettotalbytes(g);

lua\_assert(tb > 0);

if (debt < tb - MAX\_LMEM)

debt = tb - MAX\_LMEM; /\* will make 'totalbytes == MAX\_LMEM' \*/

g->totalbytes = tb - debt; //负值 负债

g->GCdebt = debt;

}

选项setstepmul的使用方法： collectgarbage("setstepmul", 200)，表示垃圾收集器的运行速度是内存分配的2倍，如果此值小于100可能会导致垃圾回收不能形成完整的周期。

垃圾回收器有两个参数用于控制它的节奏：

第一个参数，称为暂停时间，控制回收器在完成一次回收之后和开始下次回收之前要等待多久；

第二个参数，称为步进系数，控制回收器每个步进回收多少内容。粗略地来说，暂停时间越小、步进系数越大，垃圾回收越快。这些参数对于程序的总体性能的影响难以预测，更快的垃圾回收器显然会浪费更多的CPU周期，但是它会降低程序的内存消耗总量，并可能因此减少分页。只有谨慎地测试才能给你最佳的参数值。反复垃圾回收会降低lua的性能

**思考**

通过上面的介绍我们知道luaGC是增量式的分步执行的，那么GC的分步过程是如何控制进度的？

singlestep的返回值决定了GC的进度，GCSpause GCSpropagate GCSatomic返回内存的估值，在luaC\_step函数中多次调用singlestep

如何知道做完整个GC流程的时间，以及目前的进度？

大致上GC的时间和GCObject的数量成正比。 但是每个类型的GCObject的处理时间复杂度各不相同；仔细衡量每种类型的处理时间差别不太现实，这可能跟具体机器也有关系。但我们大体可以认为，占用内存较多的对象，需要的时间也更长些，当然string和userdata类型除外，因为这两种类型都没有增加mark的时间。  
所以在propagatemark函数中，每mark一个灰色节点都返回该节点的内存占用。

**lua内存分析工具**

* [云风c实现的lua内存分析工具](https://blog.codingnow.com/2012/12/lua_snapshot.html)
* [lua实现](https://www.cnblogs.com/yaukey/p/unity_lua_memory_leak_trace.html)

**参考文章：**

<https://github.com/lichuang/Lua-Source-Internal>

<https://www.lua.org/wshop18/Ierusalimschy.pdf>

<https://blog.codingnow.com/2011/03/lua_gc_1.html>

<http://www.zenyuhao.com/2017/10/13/lua-gc.html>

<https://www.e-learn.cn/content/qita/909901>

<https://liujiacai.net/blog/2018/08/04/incremental-gc/> 深入浅出垃圾回收（三）增量式 GC

<https://liujiacai.net/blog/2018/07/08/mark-sweep/> 深入浅出垃圾回收（二）Mark-Sweep 详析及其优化

<https://blog.codingnow.com/2011/04/lua_gc_6.html> Lua GC 的源码剖析 (6) 完结(string的gc细节)

[https://chenanbao.github.io/2018/07/27/Lua虚拟机创建分析/](https://chenanbao.github.io/2018/07/27/Lua%E8%99%9A%E6%8B%9F%E6%9C%BA%E5%88%9B%E5%BB%BA%E5%88%86%E6%9E%90/)

<https://zhuanlan.zhihu.com/p/22403251>

<https://www.jianshu.com/p/1359ac5c741a>

<https://blog.csdn.net/BigBrick/article/details/85317491#42_101>

[https://manistein.github.io/blog/post/program/build-a-lua-interpreter/构建lua解释器part2/a](https://manistein.github.io/blog/post/program/build-a-lua-interpreter/%E6%9E%84%E5%BB%BAlua%E8%A7%A3%E9%87%8A%E5%99%A8part2/a)

<https://blog.codingnow.com/2012/12/lua_snapshot.html> 云风c实现的lua内存分析工具

<https://www.cnblogs.com/yaukey/p/unity_lua_memory_leak_trace.html> lua内存分析

<https://www.cnblogs.com/lijiajia/p/8468054.html> lua内存分析c

<https://blog.csdn.net/yuanlin2008/article/details/8684869>

<https://www.cnblogs.com/JesseFang/archive/2012/12/27/2836160.html> lua GC之finalizer

**Chapter06 闭包**

闭包是函数原型和upvalue的结合体，在lua中，闭包被统一称为函数，而在lua中函数其实就是这里所说的函数原型。函数原型是不可以被调用的，只有和upvalue结合在一起才变成了Lua中提供的函数对象。所以公开的API定义中，不存在函数原型这个类型。只有函数原型和upvalue绑定到一起时，形成闭包才能被lua所识别，才是Lua VM所识别的对象。

可以简单理解，闭包是指函数加上函数的upvalue

下面我看一下闭包在lua中的数据结构：

#define ClosureHeader \

CommonHeader; lu\_byte nupvalues; GCObject \*gclist

typedef struct CClosure {

ClosureHeader;

lua\_CFunction f;

TValue upvalue[1]; /\* list of upvalues \*/

} CClosure;

typedef struct LClosure {

ClosureHeader;

struct Proto \*p;

UpVal \*upvals[1]; /\* list of upvalues \*/

} LClosure;

typedef union Closure {

CClosure c;

LClosure l;

} Closure;

Lua支持的两种闭包，分别是C闭包CClosure和lua闭包LClosure，他们都同属于lua定义的数据类型LUA\_TFUNCTION  
由Closure是一个联合体可以知道创建一个闭包，要么是c闭包要么是lua闭包

我们来看在lua虚拟机中如何形成闭包的：

// lvm.c

vmcase(OP\_CLOSURE) {

Proto \*p = cl->p->p[GETARG\_Bx(i)];

LClosure \*ncl = getcached(p, cl->upvals, base); /\* cached closure \*/

if (ncl == NULL) /\* no match? \*/

pushclosure(L, p, cl->upvals, base, ra); /\* create a new one \*/

else

setclLvalue(L, ra, ncl); /\* push cashed closure \*/

checkGC(L, ra + 1);

vmbreak;

}

在生成闭包的过程中，首先调用getcached函数，从缓存中取上次生成的闭包，如果存在，就重复利用。这对函数式编程特别有效，因为当你返回一个没有任何 upvalue 的纯函数，或是只绑定有全局变量的函数时，不会生成新的闭包 实例。

lua的闭包结构如图：

GC：垃圾回收相关。  
Prototype：指向原形的指针。原形中包括函数代码，变量，调试信息等。[函数原型](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter06.md#%E5%87%BD%E6%95%B0%E5%8E%9F%E5%9E%8B)  
upvalue：非局部变量，是一个比较特殊的类型，在lua编程，已经写C或者和lua交互的代码时，都看不到这个类型。它是为了解决多个闭包共享一个upvalue的情况。实际上是对一个upvalue的引用。[Upval](file:///I:\git\lua53-codedump\src-dump-NO_GIT_URL\chapter06.md#upvalue)

##### 函数原型

/\*

\*\* Function Prototypes

\*/

// lobject.h

typedef struct Proto {

CommonHeader;

lu\_byte numparams; /\* number of fixed parameters \*/

lu\_byte is\_vararg;

lu\_byte maxstacksize; /\* number of registers needed by this function \*/

int sizeupvalues; /\* size of 'upvalues' \*/

int sizek; /\* size of 'k' \*/

int sizecode;

int sizelineinfo;

int sizep; /\* size of 'p' \*/

int sizelocvars;

int linedefined; /\* debug information \*/

int lastlinedefined; /\* debug information \*/

TValue \*k; /\* constants used by the function \*/

Instruction \*code; /\* opcodes \*/

struct Proto \*\*p; /\* functions defined inside the function \*/

int \*lineinfo; /\* map from opcodes to source lines (debug information) \*/

LocVar \*locvars; /\* information about local variables (debug information) \*/

Upvaldesc \*upvalues; /\* upvalue information \*/

struct LClosure \*cache; /\* last-created closure with this prototype \*/

TString \*source; /\* used for debug information \*/

GCObject \*gclist;

} Proto;

* 从数据结构体中可以看出，里面包含了很多debug所需要的信息，包含了函数引用的常量表、调试信息。以及有多少个参数，调用这个函数需要多大的数据空间。
* lua 将原型和变量绑定的过程，都尽量避免重复生成不必要的闭包。当生成一次闭包后，闭包将被cache 引用，下次再通过这个原型生成闭包时，比较 upvalue 是否一致来决定复用。cache 是一个弱引用，一旦在 gc 流程发现引用的闭包已不存在，cache 将被置空。

#### upvalue

UpVal是一个比较特殊的类型，在lua编程，已经写C或者和lua交互的代码时，都看不到这个类型。它是为了解决多个闭包共享一个upvalue的情况。实际上是对一个upvalue的引用。

/\*

\*\* Upvalues for Lua closures

\*/

struct UpVal {

TValue \*v; /\* points to stack or to its own value \*/

lu\_mem refcount; /\* reference counter \*/

union {

struct { /\* (when open) \*/

UpVal \*next; /\* linked list \*/

int touched; /\* mark to avoid cycles with dead threads \*/

} open;

TValue value; /\* the value (when closed) \*/

} u;

};

无须用特别的标记区分一个UpVal在开放还是关闭的状态。当upvalue关闭时，UpVal中的指针v一定指向结构体内部的value.

为什么TUPVAL会有open和closed两种状态？

* open状态  
  调用luaF\_newLclosure生成完一个 Lua Closure 后，会去填那张 upvalue表。当upvalue尚在堆栈上时，其实是调用luaF\_findupval去生成一个对堆栈上的特定值之引用的TUPVAL对象的。luaF\_findupval 的实现不再列在这里，它的主要作用就是保证对堆栈相同位置的引用之生成一次。生成的这个对象就是 open 状态的。所有 open 的 TUPVAL 用一个链表串起来，挂在 global state 的 openupval 中。
* close状态  
  一旦函数返回，某些堆栈上的变量就会消失，这时，还被某些 upvalue 引用的变量就必须找个地方妥善安置。这个安全的地方就是 TUPVAL 结构之中。修改引用指针的结果，就被认为是 close 了这个 TUPVAL 。相关代码可以去看 lfunc.c 中 luaF\_close 的实现。

open和close状态入下图所示：

#### lua闭包

* 一个简单的闭包如下：
* function makecounter()
* local t = 0
* return function()
* t = t + 1
* return t
* end
* end
* local n1 = makecounter()
* local n2 = makecounter()
* print(n1()) -- 1
* print(n1()) -- 2
* print(n2()) -- 1
* print(n1()) -- 3
* 当调用makecounter后，会得到一个函数。这个函数每调用一次，返回值就会递增一。顾名思义，我们  
  可以把这个返回的函数看作一个计数器。makecounter可以产生多个计数器，每个都独立计数。也就是说，  
  每个计数器函数都独享一个变量 t ，相互不干扰。这个 t 被称作计数器函数的 upvalue ，被绑定到计数器函  
  数中。拥有了upvalue的函数就是闭包。

当函数n1执行时，函数makecounter已经返回，makecounter的局部变量t已经在栈中退出，但是n1却能访问x。这是因为x是函数f的upvalue。

而n2函数执行的结果表明n1和n2并没有共享upvalue，而是单独有一份自己的upvalue。

#### 共享和关闭upvalue

前述的模式并未提供复用。如果两个闭包需要一个共同的外部变量，每个闭包都会有一个独立的upvalue。当这些upvalue关闭后，每个闭包都包含该共同变量的一个独立的拷贝。当一个闭包修改该变量时，另一个闭包将看不到此修改。

为避免这个问题，解释器必须确保每个变量最多只有一个upvalue指向它。解释器维护了一个保存栈中所有open upvalue的链表。该链表中upvalue顺序与栈中对应变量的顺序相同。当解释器需要一个变量的upvalue时，它首先遍历这个链表：如果找到变量对应的upvalue，则复用它，因此确保了共享；否则创建一个新的upvalue并将其链入链表中正确的位置。

由于upvalue链表是有序的，且每个变量最多有一个对应的upvalue，因此当在链表中查找变量的upvalue时，遍历元素的最大数量是静态确定的。最大数量是逃往（escape to）内层闭包的变量个数和在闭包和外部变量之间声明的变量个数之和。例如，以下的代码段：

function foo ()

local a, b, c, d

local f1 = function () return d + b end

local f2 = function () return f1() + a end

...

当解释器初始化f2时，解释器在确定a没有对应的upvalue之前会遍历3个upvalue，按顺序分别是f1、d和b。

当一个变量退出作用域时，它所对应的upvalue（如果有）必须被关闭。open upvalue链表也被用于关闭upvalue。当Lua编译一个包含逃离的变量（被作为upvalue）的块时，它在块的末尾生成一个CLOSE指令，该指令“关闭到某一层级（level）为止的upvalue”。执行该指令时，解释器遍历open upvalue链表直到到达给定层级为止，将栈中变量值复制到upvalue中，并将upvalue从链表中移除。

为描述open upvalue链表如何确保upvalue共享，考虑如下的代码段：

local a = {} -- an empty array

local x = 10

for i = 1, 2 do

local j = i

a[i] = function () return x + j end

end

x = 20

在代码段开头，open upvalue链表是空的。因此，当解释器在循环中创建第一个闭包时，它会为x和j创建upvalue，并将其插入upvalue链表中。在循环体的末尾有一条CLOSE指令标识j退出了作用域，当解释器执行这条指令时，它关闭j的upvalue并将其从链表移除。解释器在第二次迭代中创建闭包时，它找到x的upvalue并复用，但找不到j的upvalue，因此创建一个新的upvalue。在循环体末尾，解释器再一次关闭j的upvalue。

在循环结束之后，程序中包含两个闭包，这两个闭包共享一个x的upvalue，但每个闭包有一个独立的j的拷贝。x的upvalue是开启的，即x的值仍在栈中。因此最后一行的赋值（x=20）改变了两个闭包使用的x值。

### 参考文章：

<https://www.cs.tufts.edu/~nr/cs257/archive/roberto-ierusalimschy/closures-draft.pdf> Closures in Lua

<https://blog.csdn.net/liutianshx2012/article/details/77367920>

<https://www.cnblogs.com/plodsoft/p/5900270.html?utm_source=tuicool&utm_medium=referral>Closures in Lua 翻译