1.1 lua 源码剖析(一)

发表时间: 2009-11-15 关键字: 源码

先来看 lua 中值的表示方式。

```
#define TValuefields Value value; int tt


typedef struct lua_TValue {
TValuefields;
} TValue;
```

其中 tt 表示类型，value 也就是 lua 中对象的表示。

```
typedef union {
    GCObject *gc; void *p;
    lua_Number n;
    int b;
} Value;
```

gc 用于表示需要垃圾回收的一些值，比如 string，table 等等。

p 用于表示 light userdata 它是不会被 gc 的。

n 表示 double

b 表示 boolean

tvalue 这样表示会有空间的浪费.可是由于要完全符合 c99,因此只能这么做.否则我们为了效率可以这么做.由于在大多数机器上,指针都是严格对齐(4 或者 8 字节对齐).因此后面的 2,3 位就是 0,因此我们可以将类型存储在这几位,从而极大地压缩了 Value 的大小.

更新:这里经的老朱同学的提醒,其实 tvalue 之所以不使用指针的后几位来存储类型,更重要的时候由于和 c 的交互.

因为那样的话,我们就必须强制和 lua 交互的 c 模块也必须保持和我们一样的内存模型了.

lua_state 表示一个 lua 虚拟机，它是 per-thread 的，也就是一个协程（多个和 lua 交互的 c 程序，那自然也会有多

个 lua-state)一个 lua_state,然后来看它的几个比较重要的域。

StkId top 这个域表示在这个栈上的第一个空闲的 slot。
StkId base 这个域表示当前所在函数的 base。这个 base 可以说就是栈底。只不过是当前函数的。
StkId stack_last 在栈上的最后一个空闲的 slot
StkId stack 栈的 base，这个是整个栈的栈底。
StkId 是一个 Tvalue 类型的指针。

在 lstrlib 中，基本上所有的 str 函数都是首先调用 luaL_checklstring 来得到所需要处理的字符串然后再进行处理。如果是需要改变字符串的话，那么都会首先生成一个 luaL_Buffer 对象(主要原因是在 lua 中，都会做一个传递进来的字符串的副本的)，然后最终将处理的结果通过调用 luaL_pushXXX 放到栈中。

luaL_checklstring 函数，这个函数只是简单的对 lua_tolstring 进行了一层简单的封装。而 luaL_tolstring 也是对 index2adr 函数做了一层简单封装，然后判断所得到的值是否为字符串，是的话返回字符串，并修改 len 为字符串长度。

```
LUALIB_API const char *luaL_checklstring (lua_State *L, int narg, size_t *len) {
///通过 luaL_tolstring 得到字符串 s
   const char *s = lua_tolstring(L, narg, len); if (!s) tag_error(L, narg,
   LUA_TSTRING); return s;
}
```

因此我们详细来看 index2adr 这个函数，这个函数目的很简单，就是通过索引得到对应的值的指针。第一个参数 lua_state,第二个参数为索引值。

我们首先要知道在 lua 中，索引值可以为负数也可以为正数，当为负数的话，top 为-1,当为正数第一个压入栈的元素为 1,依此类推.

而且有些类型的对象当转换时还需要一些特殊处理，比如闭包中的变量。

除去特殊的，一般的存取很简单，当 index>0 则我们只需要用 base+i -1 来取得这个指针，为什么要用 base 而不
是 top 呢，我们上面已经说过了，当 index 为正数，所取得的是第一个值，因此也就是栈的最下面那个值，而 base 表示当前函数在栈里面的位置，因此我们加上 i -1 就可以了。当 index<0 则更简单，我们用 top+index 就可以了。

```
static TValue *index2adr (lua_State *L, int idx) { if (idx > 0) {
///索引为正值时，通过 base 取得 value
      TValue *o = L->base + (idx - 1); api_check(L, idx <=
      L->ci->top - L->base);
///如果超过 top，则返回 nil，否则返回 o。
      if (o >= L->top) return cast(TValue *, luaO_nilobject); else return o;
   } else if (idx > LUA_REGISTRYINDEX) {
///正常的小于 0 的索引。则直接通过 top+idx 取得对象。
      api_check(L, idx != 0 && -idx <= L->top - L->base); return L->top + idx; }
///下面省略的部分是取得闭包以及其他一些类型的值,等我们后面分析完所有类型后，会再次回到这个函数
.............................
}
```

而 lmathlib.c 中处理数字更简单，因为数字不需要转换，因此基本都是直接调用 lua_pushnumber 来压入栈。

接下来就来看 lua_pushXXX 这些函数。这些函数都是用来从 C->stack 的。

我们先来看不需要 gc 的类型，不需要 gc 的类型的话都是比较简单。比如 lua_pushinteger.内部实现都是调用 setnvalue 来将值 set 进栈顶。

```
#define setnvalue(obj,x) \
   { TValue *i_o=(obj); i_o->value.n=(x); i_o->tt=LUA_TNUMBER; }
```

可以看到很简单的实现，就是给 value 赋值，然后给类型也赋值。

而这里我们要知道基本上每个类型都会有一个 setXXvalue 的宏来设置相应的值。这里还要注意一个就是 nil 值，在 lua 中，nil 有一个专门的类型就是 LUA_TNIL,下面就是 lua 中的值的类型。

```
#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
```

接下来我们先来看 lua 中 gc 的结构，在 lua 中包括 table，string，function 等等都是需要 gc 的。因此 gc 的 union

也就包含了这几个类型：

```
union GCObject {
   GCheader gch; union TString ts; /*string*/
   union Udata u;   /*user data*/ union Closure cl;
   /* 闭包 */ struct Table h; /*表*/ struct Proto p;
   /*函数*/ struct UpVal uv; struct lua_State th;
   /* thread */
};
```

而这里 gc 的头主要就是用来实现 gc 算法，它包括了 next(指向下一个 gc 对象),tt 表示类型，marked 用来标记这个对象的使用。

```
#define CommonHeader    GCObject *next; lu_byte tt; lu_byte marked
```

接下来我们就来详细分析下这几种需要 gc 的类型的结构。

首先来看 TSring:

```
typedef union TString {
    L_Umaxalign dummy;  /* ensures maximum alignment for strings */ struct
        { CommonHeader; lu_byte reserved; unsigned int hash; size_t len;
    } tsv;
} TString;
```

我们知道在 lua 中会将字符串通过一定的算法计算出散列值，并保存这个散列值到 hash 域中，然后以后的操作，都是通过这个散列值来进行操作。

而 TSring 其实只是字符串的一个头，而字符串的值会紧跟在头的后面，详细可以看 newlstr 函数。

在 lus_state 中的 global_State *l_G 也就是全局状态中有一个 stringtable strt 的域，所有的字符串都是保存在这个散列表中。

```
typedef struct stringtable {
    GCObject **hash; lu_int32 nuse;  /* number of
    elements */ int size;
} stringtable;
```

可以看到 hash 也就是保存了所有的字符串。这里 size 表示为 hash 桶的大小。

在 luaS_newlstr 中会先计算字符串的 hash 值，然后遍历 stringtable 这个全局 hash 表，如果查找到对应的字符串就返回 ts，否则调用 newlstr 重新生成一个。

而 newlstr 则就是新建一个 tsring 然后给相应位赋值，然后计算 hash 值插入到全局的 global_State 的 stringtable 中。然后每次都会比较 nuse 和 size 的大小，如果大于 size 则说明碰撞太严重，因此增加桶的大小。这里增加每次都是 2 的倍数增加。

```
static TString *newlstr (lua_State *L, const char *str, size_t l, unsigned int h)
    { TString *ts; stringtable *tb; if (l+1 > (MAX_SIZET -
    sizeof(TString))/sizeof(char)) luaM_toobig(L);
///初始化字符串。
    ts = cast(TString *, luaM_malloc(L, (l+1)*sizeof(char)+sizeof(TString))); ts->tsv.len = l; ts->tsv.hash = h;
    ts->tsv.marked = luaC_white(G(L)); ts->tsv.tt = LUA_TSTRING; ts->tsv.reserved = 0;
///开始拷贝字符串数据到 ts 的末尾
    memcpy(ts+1, str, l*sizeof(char)); ((char *)(ts+1))[l] = '\0';  /*
    ending 0 */
///取得全局的 strtable tb =
    &G(L)->strt;
///计算位置
    h = lmod(h, tb->size);
///链接到相应的位置，并更新 nuse。
    ts->tsv.next = tb->hash[h];  /* chain new entry */ tb->hash[h] = obj2gco(ts);
    tb->nuse++;
///判断是否需要增加桶的大小
    if (tb->nuse > cast(lu_int32, tb->size) && tb->size <= MAX_INT/2) luaS_resize(L, tb->size*2);  /* too crowded
        */
    return ts;
}
```

还有一个就是 TSring 和插入到 stringtable 中时所要计算的 hash 值是不一样的。接下来就来看这两个 hash 值如何生成的。先来看 tsring 中的 hash 的生成：

```
size_t step = (l>>5)+1; for (l1=l; l1>=step; l1-=step)  /* compute hash */ h = h ^
((h<<5)+(h>>2)+cast(unsigned char, str[l1-1]));
```

step 表示要计算的次数，l 为字符串的长度，这里主要是为了防止太长的字符串。因此右移 5 位并加一。

这个 hash 算法叫做 JS Hash Function ,计算完后对桶的大小 size 取模然后插入到 hash 表。下面来看 luaS_newlstr.

```
TString *luaS_newlstr (lua_State *L, const char *str, size_t l) { GCObject *o; unsigned int h = cast(unsigned
   int, l);  /* seed */ size_t step = (l>>5)+1;  /* if string is too long, don't hash all its chars */ size_t l1;
///计算字符串  hash，
   for (l1=l; l1>=step; l1-=step)  /* compute hash */ h = h ^
      ((h<<5)+(h>>2)+cast(unsigned char, str[l1-1]));
///遍历全局的字符串表
   for (o = G(L)->strt.hash[lmod(h, G(L)->strt.size)]; o != NULL; o =
         o->gch.next) {
      TString *ts = rawgco2ts(o); if (ts->tsv.len == l && (memcmp(str, getstr(ts), l) == 0)) {
         /* string may be dead */ if (isdead(G(L), o)) changewhite(o);
         return ts;
      } } return newlstr(L, str, l, h);  /* not found */
}
```

这里要注意 lua 每次都会 memcpy 传递进来的字符串的。而且在 lua 内部字符串也都是以 0 结尾的。

接下来来看 lua 中最重要的一个结构 Table.

```
typedef union TKey { struct {
        TValuefields; struct Node *next;  /* for chaining */
    } nk;
    TValue tvk; } TKey;



typedef struct Node {
    TValue i_val;
    TKey i_key; } Node;


typedef struct Table {
    CommonHeader; lu_byte flags;  /* 1<<p means tagmethod(p) is not present */
    lu_byte lsizenode;  /* log2 of size of `node' array */ struct Table *metatable; TValue
    *array;  /* array part */
    Node *node;
    Node *lastfree;  /* any free position is before this position */ GCObject *gclist; int
    sizearray;  /* size of `array' array */
} Table;
```

这里它的头和 TSring 是一样的，其实所有 gc 的类型的头都是相同的。在 lua5.0 中 table 表示为一种混合的

数据结构，包含一个数组部分和一个散列表部分，当键为整数时，他不会保存这个键而是直接保存这个值到数组中。也就是数组保存在上面的 array 中，而散列表保存在 node 中。其中 tkey 保存了当前 slot 的下一个

node 的指针。

我们可以通过 lapi.c 来详细分析 table 的实现。

比较核心的函数就是 luaH_get。

const TValue *luaH_get (Table *t, const TValue *key)

这个函数就是用来从表 t 中查找 key 对应的值，从而返回。因此这里我们可以看到它会通过 key 的类型不同，从而进行不同的处理。

1 如果是 NIl 则直接返回 luaO_nilobject。

2 如果是 string，则调用 luaH_getstr 进行处理(下面会介绍)

3 如果是 number，则调用 luaH_getnum 来处理。这里要注意如果是非 int 类型的话，它会跳过这里，进入 default 处理。

4 然后就是 default 了。它会计算 key 的 hash 值，然后在 hash 表中查找到 slot，然后遍历这个链表查找到对应的 key，然后返回 value。如果没有找到则返回 nil。

此时由于 lua 的 lua_Number 默认是 double 型的，而数组的下标是 int 的，因此这里有一个转换 double 到 int 的一个过程。在 lua 中是通过 lua_number2int 这个函数来实现的，它用了一个小技巧。

```
union luai_Cast { double l_d; long l_l; }; #define lua_number2int(i,d)
\
  { volatile union luai_Cast u; u.l_d = (d) + 6755399441055744.0; (i) = u.l_l; }
```

可以看到 lua 是定义了一个联合，然后将要转换的 d 加上 6755399441055744.0。然后将 l_l 赋值给最终的值 i。

6755399441055744.0 是一个 magic number ，它也就是 1.5*2^52 ，而在 ia-32 的架构中，fraction 是 52 位。而在浮点数加法中，首先要做的就是小数点对齐，而对齐标准就是和幂大的对齐。并且小数点前的 1 是忽略的。
因此当相加时，就会将小数点后的四舍五入掉了。而为什么是 1.5 呢，主要是为了处理负数。

我这里只是简单的分析了下，详细的，自己动笔算一下就清楚了。

ok，现在我们来看 luaH_getstr 的实现。这个函数的实现其实很简单，就是计算 hash 然后得到链表，并遍

历，得到对应 key 的值。这里我们要知道当 key 为字符串时，在 table 中的 hash 不等于 string 本身的 hash(也就是全局字符串 hash 的那个 hash).

```
const TValue *luaH_getstr (Table *t, TString *key) {
///得到对应的节点。
   Node *n = hashstr(t, key);
///然后开始遍历链表。
   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;
}
```

然后是 luaH_getnum 的实现。这个函数首先判断这个 key，也就是数组下标是否在范围内。如果在则直接返回相应的值。否则将这个 key 计算 hash 然后在 hash 链表中查找相应的值。

```
const TValue *luaH_getnum (Table *t, int key) {
   /* (1 <= key && key <= t->sizearray) */
///判断  key 的范围。
   if (cast(unsigned int, key-1) < cast(unsigned int, t->sizearray)) return &t->array[key-1]; else
   {
///如果不在，则说明在  hash 部分，因此开始遍历对应的  node。
      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;
   }
}
```

看完 get 我们来看 set 方法。

TValue *luaH_set (lua_State *L, Table *t, const TValue *key)

这个函数会判断是否 key 已经存在，如果已经存在则直接返回对应的值。否则会调用 newkey 来新建一个 key，并返回对应的 value。(这里主要并不是所有的数字的 key 都会加到数组里面，有一部分会加入到 hash 表中).可以说这个 hash 表中包含两个链表，一个是空的槽的链表，一个是已经填充了的槽的链表。

```
TValue *luaH_set (lua_State *L, Table *t, const TValue *key) {
///调用 get 得到对应的值（也就是在表中查找是否存在这个 key）
   const TValue *p = luaH_get(t, key); t->flags = 0;
///不为空，则直接返回这个值
   if (p != luaO_nilobject) 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");
///调用 newkey，返回一个新的值。 return
      newkey(L, t, key);
   }
}
```

然后来看 newkey。

static TValue *newkey (lua_State *L, Table *t, const TValue *key)

这里 lua 使用的是 open-address hash。不过做了一些改良。这里它会有专门的一个 free position 的链表（也就是所有空闲槽的一个链表)，来保存所有冲突的 node，换句话说就是如果有冲突，则从 free position 中取得位置，然后将冲突元素放进去，并从 free position 中删除。

这个函数的具体流程是这样的：

1　首先调用 mainposition 返回一个 node，然后判断 node 的 value 是否为空，如果为空，则给 value 赋值,然后返回这个 node 的 value。

2　如果 node 的 value 非空，或者说这个 node 就是空的，则先通过 getfreepo 从空的槽的链表得到一个空的槽，如果没有空着的槽，则说明 hash 表已满，此时扩容 hash 表，然后继续调用 luaH-set.

3　如果此时有空着的槽，再次计算 mainposition,通过 key 的 value.(这是因为我们是开地址散列，每次冲突的元素都会放到 free position 中)。如果得到的 node 和第一步计算的 node 相同，则将空着的槽(也就是链表）n 链接到第一步得到的 node 后面，这个也就是将当前要插入的 key 的 node 到 free position，然后移动 node 指针到 n 的位置，然后赋值并返回。

4　如果和第一步计算的 node 不同，则将新的 node 插入到这个 node。然后将本身这个 node 移动到 freeposition。

接下来来看源码。

```
static TValue *newkey (lua_State *L, Table *t, const TValue *key) {
///得到主位置的值。
   Node *mp = mainposition(t, key);
   if (!ttisnil(gval(mp)) || mp == dummynode) {
      Node *othern;
///得到 freee position 的 node。
      Node *n = getfreepos(t);  /* get a free place */ if (n == NULL) {  /*
      cannot find a free place? */
///如果为空，则说明 table 需要增长，因此 rehash
         rehash(L, t, key);  /* grow table */ return luaH_set(L, t, key);  /* re-insert key into grown table
         */
      } lua_assert(n != dummynode);
///得到 mp 的主位置。
      othern = mainposition(t, key2tval(mp));
///如果不等，则说明 mp 本身就是一个冲突元素。
      if (othern != mp) {  /* is colliding node out of its main position? */
///链接冲突元素到 free position
         while (gnext(othern) != mp) othern = gnext(othern);  /* find previous */ 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 */ setnilvalue(gval(mp)); }
      else {  /* colliding node is in its own main position */
         /* new node will go into free position */
///这个说明我们当前的 key 是冲突元素。
         gnext(n) = gnext(mp);  /* chain new position */ gnext(mp) = n; mp
         = n;
      }
   }
///赋值。
   gkey(mp)->value = key->value; gkey(mp)->tt = key->tt; luaC_barriert(L, t, key);
   lua_assert(ttisnil(gval(mp)));
///返回 value
```

```
    return gval(mp);
}
```

接下来来看 rehash 的实现。每次表满了之后，都会重新计算散列值。

具体的函数是
static void rehash (lua_State *L, Table *t, const TValue *ek)

再散列的流程很简单。第一步是确定新数组部分和新散列部分的尺寸。所以，Lua 遍历所有条目，计数并分类它们，每次满的时候，都会是最接近数组当前大小的值的次幂(0->1,3->4,9->16 等等)，它使得数组部分超过半数的元素被填充。然后散列尺寸是能容纳所有剩余条目的 2 的最小乘幂。 lua 为了提高效率，尽量不去做 rehash，因为 rehash 非常非常耗时，因此看下面的代码：

```
local a={} print("----------\n") a.x=1
a.y=2
a.z=3
a.u=4
a.o=5 for i = 1, 1 do print(i) print("==============\n") a[i]=1 print("==============\n") end
```

当 a.o 之后表的散列部分大小为 8,因此下面的 a[i]=1,尽管属于数组部分，可是不会进行 rehash，而是暂时放到 hash 部分中。而当必须要 rehash 表的时候，计算数组大小时，会将放到 hash 部分中的数组重新插入到数组部分。

来看代码，这里注释很详细，我就简单的介绍下。

我们知道在 lua 中，数组部分有个最大值（为 2^26)，而这里它准备了一个数组，大小为 26+1,然后数组每一个的值都表示了在某一个段的范围内的值得多少：

nums[i] = number 出表示了在 2^(i-1) 和 2^i 之间的数组部分的有多少值。这样做的目的主要是为了防止数组部分过于稀疏，太过于稀疏的话，会将一些值放到 hash 部分中，我们下面分析 computesizes 时，会详细介绍这个。

```
///这里表示数组部分的最大容量为 2^26
#define MAXBITS        26 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;
///首先初始化每部分都为 0
    for (i=0; i<=MAXBITS; i++) nums[i] = 0;  /* reset counts */
///计算 array 部分的元素个数
    nasize = numusearray(t, nums);  /* count keys in array part */ totaluse = nasize;  /* all those keys
    are integer keys */
///计算 hash 部分的元素个数
    totaluse += numusehash(t, nums, &nasize);  /* count keys in hash part */ /* count extra key */
    nasize += countint(ek, nums); totaluse++;
///计算新的数组部分的大小
    na = computesizes(nums, &nasize); /* resize the table to new
computed sizes */ ///调用 resize 调整 table 的大小。
    resize(L, t, nasize, totaluse - na);
}
```

这里比较关键就是上面几个计算函数，我们一个个来分析：

numusearray 计算当前的数组部分的元素个数，并且给 num 赋值。

```
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;
..... /* count elements in range (2^(lg-1), 2^lg] */ for (;
       i <= lim; i++) { if (!ttisnil(&t->array[i-1])) lc++; }
///得到相应的段的个数
       nums[lg] += lc;
///计算总的元素个数。
       ause += lc;
   } return ause;
}
```

然后是 numusehash ，这个函数计算 hash 部分的元素个数。

```
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);
///遍历  node。(由于是开地址散列，因此遍历很简单) while (i-
   -) {
       Node *n = &t->node[i];
```

```
///判断是否为 nil
      if (!ttisnil(gval(n))) {
///countint 就是判断 n 是否可以进入数组部分，是的话返回 1,否则为 0
         ause += countint(key2tval(n), nums);
///总得大小加一
         totaluse++;
      }
   }
///更新数组部分的大小
   *pnasize += ause; return
   totaluse;
}
```

接下来是 computesizes,它用来计算新的数组部分的大小。这里扩展的大小也就是最接近数组当前的大小的 2 的次幂。

这里遍历也就是每次一个段的遍历。

如果数组的利用率小于 50%的话，大的元素就不会计算到数组部分，也就是会放到 hash 部分。

```
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) {
///如果大于 0,说明这个段中有数据
      if (nums[i] > 0) { a +=
         nums[i]; if (a > twotoi/2) {
///如果多于一半，则设置数组当前的大小为 twotoi(2^i)
            n = twotoi;  /* optimal size (till now) */ na = a;  /* all elements smaller than n will go to array
            part */
         }
```

```
///如果少于一半，则这个值将不会计算到数组部分，也就是 n 值不会更新
        } if (a == *narray) break;  /* all elements already counted */ }
    *narray = n; lua_assert(*narray/2 <= na && na <= *narray);
    return na;
}
```

resize 就不介绍了，这个函数比较简单，就是重新分配数组部分和 hash 部分的大小，这里用 realloc 来调整大小，然后重新插入值。

1.2 lua 源码剖析(二)

发表时间: 2009-12-04

这次紧接着上次的，将 gc 类型的数据分析完毕。

谢谢老朱同学的指正,这里 CClosure 和 LClosure 理解有误.

先来看闭包:

可以看到闭包也是会有两种类型，这是因为在 lua 中，函数不过是一种特殊的闭包而已。

更新:这里 CClosure 表示是 c 函数,也就是和 lua 外部交互传递进来的 c 函数以及内部所使用的 c 函数.

LClosure 表示 lua 的函数,这些函数是由 lua 虚拟机进行管理的..

```
typedef union Closure {
    CClosure c;
    LClosure l;
} Closure;
```

接下来来看这个两个结构。

在看着两个结构之前，先来看宏 ClosureHeader，这个也就是每个闭包(函数的头).它包括了一些全局的东西:

更新 :
isC:如果是 c 函数这个值为 1,为 lua 的函数则为 0.
nupvalues:表示 upvalue 或者 upvals 的大小(闭包和函数里面的)。
gclist:链接到全局的 gc 链表。
env:环境，可以看到它是一个 table 类型的，他里面保存了一些全局变量等。

```
#define ClosureHeader \ CommonHeader; lu_byte isC; lu_byte nupvalues; GCObject *gclist; \ struct
                Table *env
```

ok 接下来先来看 CClosure 的实现.他很简单,就是保存了一个函数原型,以及一个参数列表

更新: lua_CFunction f: 这个表示所要执行的 c 函数的原
型.
TValue upvalue[1]:这个表示函数运行所需要的一些参数(比如 string 的 match 函数,它所需要的几个参数都会保存在 upvalue 里面

```
typedef struct CClosure { ClosureHeader; lua_CFunction f; TValue upvalue[1];
} CClosure;
```

更新:
这里我们只简要的介绍 CClosure ,主要精力我们还是放在 LClosure 上.我来简要介绍下 CClosure 的操作.一般当我们将 CClosure 压栈,然后还有一些对应的调用函数 f 所需要的一些参数,此时我们会将参数都放到 upvalue 中,然后栈中只保存 cclosure 本身,这样当我们调用函数的时候(有一个全局的指针指向当前的调用函数),能够直接得到所需参数,然后调用函数.

```
LUA_API void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n) { Closure *cl; lua_lock(L); luaC_checkGC(L);
    api_checknelems(L, n);
///new 一个  cclosure cl = luaF_newCclosure(L, n,
    getcurrenv(L));









    cl->c.f = fn; L->top -= n;
///开始将参数值放到  upvalue 中. while (n--) setobj2n(L,
    &cl->c.upvalue[n], L->top+n); setclvalue(L, L->top, cl);
    lua_assert(iswhite(obj2gco(cl))); api_incr_top(L); lua_unlock(L);
}
```

然后来看 LClosure 的实现。

在 lua 中闭包和函数是原型是一样的,只不过函数的 upvalue 为空罢了,而闭包 upvalue 包含了它所需要的局部变量
值.

这里我们要知道在 lua 中闭包的实现。Lua 用一种称为 upvalue 的结构来实现闭包。对任何外层局部变量的存取间接地通过 upvalue 来进行，也就是说当函数创建的时候会有一个局部变量表 upvals（下面会介绍到).

然后当闭包创建完毕，它就会复制 upvals 的值到 upvalue。详细的描述可以看 the implementation of lua 5.0(云风的 blog 上有提供下载).

struct Proto *p：这个指针包含了很多的属性，比如变量，比如嵌套函数等等。
UpVal *upvals[1]：这个数组保存了指向外部的变量也就是我们闭包所需要的局部变量。

下面会详细分析这个东西。

```
typedef struct LClosure {
    ClosureHeader; struct
    Proto *p; UpVal
    *upvals[1];
} LClosure;
```

这里我摘录一段 the implementation of lua 5.0 里面的描述：

引用

> 通过为每个变量至少创建一个 upvalue 并按所需情况进行重复利用，保证了未决状态（是否超过生存期）的局部变量（pending vars）能够在闭包间正确地
> 共享。为了保证这种唯一性，Lua 为整个运行栈保存了一个链接着所有正打开着的 upvalue（那些当前正指向栈内局部变量的 upvalue）的链表（图 4 中未决状态的局部变量的链表）。当 Lua 创建一个新的闭包时，它开始遍历所有的外层局部变量，对于其中的每一个，若在上述 upvalue 链表中找到它，就重用此 upvalue，
> 否则，Lua 将创建一个新的 upvalue 并加入链表中。注意，一般情况下这种遍历过程在探查了少数几个节点后就结束了，因为对于每个被内层函数用到的外层局部变量来说，
> 该链表至少包含一个与其对应的入口（upvalue）。一旦某个关闭的 upvalue 不再被任何闭包所引用，那么它的存储空间就立刻被回收。

下面是示意图：

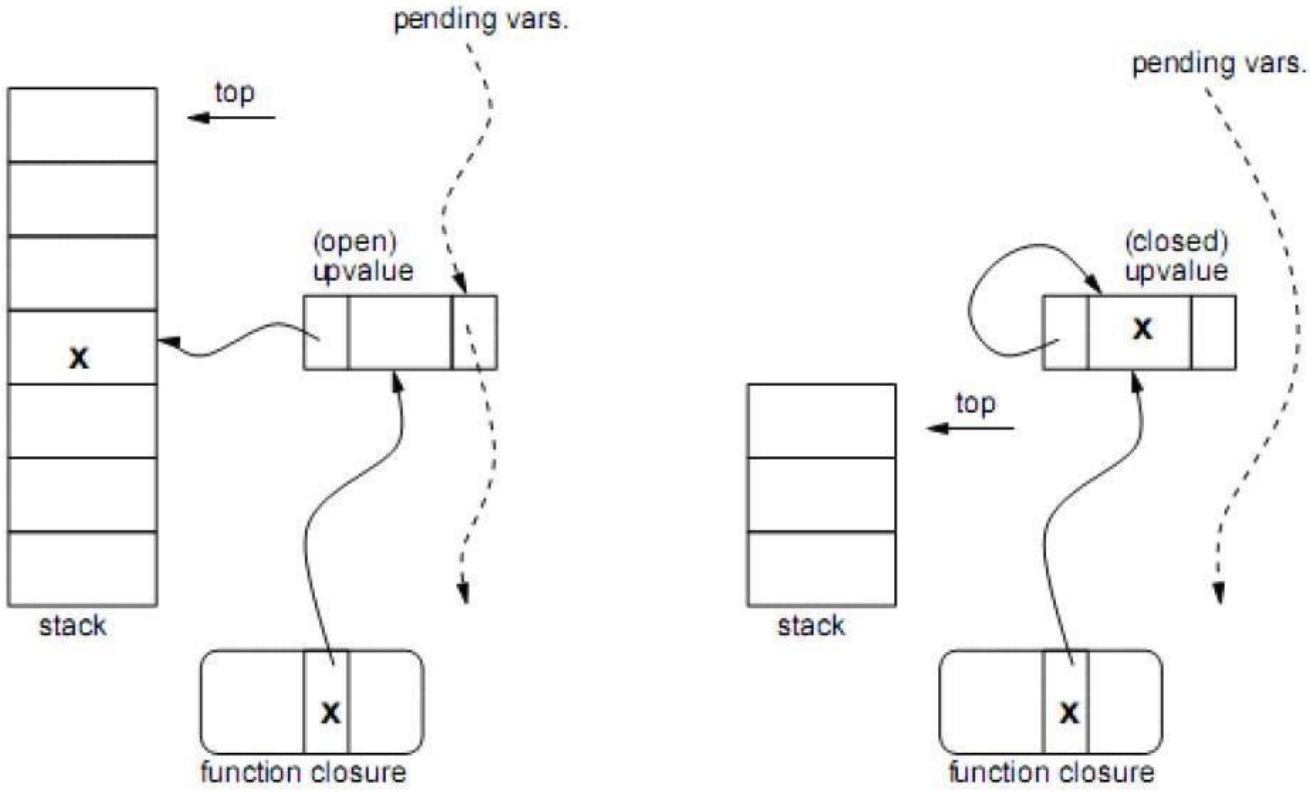

Figure 4: An upvalue before and after being "closed".

这里的未决状态（是否超过生存期）的局部变量指的就是我们下面的 UpVal，其中：

TValue *v:指向栈内的自己的位置或者自己(这里根据是否这个 uvalue 被关闭）。

union u:这里可以看到如果是被关闭则直接保存 value。如果打开则为一个链表。

```
typedef struct UpVal {
   CommonHeader; TValue *v;  /* points to stack or to its own value */ union
   {
      TValue value;  /* the value (when closed) */ struct {  /* double
      linked list (when open) */ struct UpVal *prev; struct UpVal
      *next;
      } l;
   } u;
} UpVal;
```

然后来看 luaF_newLclosure 的实现，它与 cclosure 类似。

```
Closure *luaF_newLclosure (lua_State *L, int nelems, Table *e) { Closure *c = cast(Closure *,
luaM_malloc(L, sizeLclosure(nelems))); luaC_link(L, obj2gco(c), LUA_TFUNCTION); c->l.isC = 0;
c->l.env = e; ///更新  upvals。
    c->l.nupvalues = cast_byte(nelems); while (nelems--) c->l.upvals[nelems]
    = NULL; return c;
}
```

ok，接下来我们就通过一些函数来更详细的理解闭包的实现。先分析 CClosure。我们来看 luaF_newCclosure 的实现，这个函数创建一个 CClosure,也就是创建一个所需要执行的 c 函数. 这个函数实现比较简单，就是 malloc 一个 Closure，然后链接到全局 gc，最后初始化 Closure 。

```
Closure *luaF_newCclosure (lua_State *L, int nelems, Table *e) {
///分配内存
    Closure *c = cast(Closure *, luaM_malloc(L, sizeCclosure(nelems)));
///链接到全局的  gc 链表
    luaC_link(L, obj2gco(c), LUA_TFUNCTION);
///开始初始化。
    c->c.isC = 1; c->c.env = e; c->c.nupvalues = cast_byte(nelems);
    return c;
}
```

在 lua_State 中它里面包含有 GCObject 类型的域叫 openupval，这个域也就是当前的栈上的所有 open 的 uvalue。可以看到这里是 gcobject 类型的，这里我们就知道为什么 gcobvject 中为什么还要包含 struct UpVal uv 了。而在 global_State 中的 UpVal uvhead 则是整个 lua 虚拟机里面所有栈的 upvalue 链表的头。

然后我们来看 lua 中如何 new 一个 upval。它很简单就是 malloc 一个 UpVal 然后链接到 gc 链表里面。这边要注意，每次 new 的 upval 都是 close 的。

```
UpVal *luaF_newupval (lua_State *L) { ///new 一个 upval
  UpVal *uv = luaM_new(L, UpVal);
///链接到全局的 gc 中
  luaC_link(L, obj2gco(uv), LUA_TUPVAL);
///可以看到这里的 upval 是 close 的。 uv->v =
  &uv->u.value; setnilvalue(uv->v);


  return uv;
}
```

接下来我们来看闭包如何来查找到对应的 upval，所有的实现就在函数 luaF_findupval 中。我们接下来来看这个函数的实现。

这个函数的流程是这样的。

1 首先遍历 lua_state 的 openupval，也就是当前栈的 upval，然后如果能找到对应的值，则直接返回这个 upval。

2 否则新建一个 upval（这里注意 new 的是 open 的)，然后链接到 openupval 以及 uvhead 中。而且每次新的

upval 的插入都是插入到链表头的。而且这里插入了两次。这里为什么要有两个链表，那是因为有可能会有多个栈，而 uvhead 就是用来管理多个栈的 upvalue 的（也就是多个 openupval)。

```
UpVal *luaF_findupval (lua_State *L, StkId level) { global_State *g = G(L);
///得到 openupval 链表
  GCObject **pp = &L->openupval;
  UpVal *p;
  UpVal *uv;
///开始遍历 open upvalue。
  while (*pp != NULL && (p = ngcotouv(*pp))->v >= level) { lua_assert(p->v != &p->u.value);
///发现已存在。
    if (p->v == level) { if (isdead(g, obj2gco(p)))  /* is it dead? */ changewhite(obj2gco(p));
      /* ressurect it */
///直接返回
      return p;
    } pp = &p->next;
  }
///否则 new 一个新的 upvalue
  uv = luaM_new(L, UpVal);  /* not found: create a new one */
```

```
  uv->tt = LUA_TUPVAL; uv->marked = luaC_white(g);
///设置值
  uv->v = level;  /* current value lives in the stack */
///首先插入到 lua_state 的 openupval 域 uv->next = *pp;  /* chain it in
  the proper position */ *pp = obj2gco(uv);
///然后插入到 global_State 的 uvhead（这个也就是双向链表的头)
  uv->u.l.prev = &g->uvhead;  /* double link it in `uvhead' list */ uv->u.l.next = g->uvhead.u.l.next;
  uv->u.l.next->u.l.prev = uv; g->uvhead.u.l.next = uv; lua_assert(uv->u.l.next->u.l.prev == uv && uv->u.l.prev->u.l.next
  == uv); return uv;
}
```

更新:

上面可以看到我们 new 的 upvalue 是 open 的,那么什么时候我们关闭这个 upvalue 呢,当函数关闭的时候,我们就会 unlink 掉 upvalue,从全局的 open upvalue 表中:

```
void luaF_close (lua_State *L, StkId level) {
    UpVal *uv; global_State *g =
    G(L);
///开始遍历  open upvalue while (L->openupval != NULL && (uv = ngcotouv(L->openupval))->v >= level)
    {
        GCObject *o = obj2gco(uv);
        lua_assert(!isblack(o) && uv->v != &uv->u.value); L->openupval =
        uv->next;  /* remove from `open' list */ if (isdead(g, o))
        luaF_freeupval(L, uv);  /* free upvalue */ else {
///unlink 掉当前的  uv.
            unlinkupval(uv);



            setobj(L, &uv->u.value, uv->v); uv->v = &uv->u.value;  /* now current value lives
        here */ luaC_linkupval(L, uv);  /* link upvalue into `gcroot' list */ }
    } }

static void unlinkupval (UpVal *uv) { lua_assert(uv->u.l.next->u.l.prev == uv && uv->u.l.prev->u.l.next
    == uv); uv->u.l.next->u.l.prev = uv->u.l.prev;  /* remove from `uvhead' list */ uv->u.l.prev->u.l.next
    = uv->u.l.next;
}
```

接下来来看 user data。这里首先我们要知道，在 lua 中，创建一个 userdata，其实也就是分配一块内存紧跟在

Udata 的后面。后面我们分析代码的时候就会看到。也就是说 Udata 相当于一个头。

```
typedef union Udata {
    L_Umaxalign dummy; struct {
///gc 类型的都会包含这个头，前面已经描述过了。
        CommonHeader;
///元标
        struct Table *metatable;
///环境
        struct Table *env;
///当前  user data 的大小。
        size_t len;
    } uv;
} Udata;
```

ok，接下来我们来看代码，我们知道调用 lua_newuserdata 能够根据指定大小分配一块内存，并将对应的 userdata 压入栈。

这里跳过了一些代码，跳过的代码以后会分析到。

```
LUA_API void *lua_newuserdata (lua_State *L, size_t size) { Udata *u;
   lua_lock(L); luaC_checkGC(L);
///new 一个新的 user data，然后返回地址 u
   = luaS_newudata(L, size, getcurrenv(L));
///将 u 压入压到栈中。
   setuvalue(L, L->top, u);
///更新栈顶指针
   api_incr_top(L); lua_unlock(L);
///返回 u+1,也就是去掉头(Udata)然后返回。 return u +
   1;
}
```

我们可以看到具体的实现都包含在 luaS_newudata 中，这个函数也满简单的，malloc 一个 size+sizeof(Udata) 的内存，然后初始化 udata。

我们还要知道在全局状态，也就是 global_State 中包含一个 struct lua_State *mainthread，这个主要是用来管理 userdata 的。它也就是表示当前的栈，因此下面我们会将新建的 udata 链接到它上面。

```
Udata *luaS_newudata (lua_State *L, size_t s, Table *e)
   { Udata *u;

///首先检测 size，userdata 是由大小限制的。 if (s >
   MAX_SIZET - sizeof(Udata)) luaM_toobig(L);
///然后 malloc 一块内存。
   u = cast(Udata *, luaM_malloc(L, s + sizeof(Udata)));
///这里 gc 相关的东西，以后分析 gc 时再说。
```

```
    u->uv.marked = luaC_white(G(L));  /* is not finalized */
///设置类型
    u->uv.tt = LUA_TUSERDATA;

///设置当前  udata 大小 u->uv.len = s; u->uv.metatable = NULL;
    u->uv.env = e; /* chain it on udata list (after main thread) */
///然后链接到  mainthread 中
    u->uv.next = G(L)->mainthread->next; G(L)->mainthread->next = obj2gco(u);

///然后返回。 return u;
}
```

还剩下两个 gc 类型，一个是 proto(函数包含的一些东西)一个是 lua_State（也就是协程).

我们来简单看一下 lua_state,顾名思义，它就代表了状态，一个 lua 栈(或者叫做线程也可以)，每次 c 与 lua 交互都会新建一个 lua_state,然后才能互相通过交互。可以看到在 new state 的时候它的 tt 就是 LUA_TTHREAD。

并且每个协程也都有自己独立的栈。

我们就来看下我们前面已经触及到的一些 lua-state 的域：

```
struct lua_State {
    CommonHeader;

///栈相关的
    StkId top;  /* first free slot in the stack */
    StkId base;  /* base of current function */
    StkId stack_last;  /* last free slot in the stack */
    StkId stack;  /* stack base */
```

```
///指向全局的状态。
    global_State *l_G;

///函数相关的
    CallInfo *ci;  /* call info for current function */
    const Instruction *savedpc;  /* `savedpc' of current function */
    CallInfo *end_ci;  /* points after end of ci array*/ CallInfo *base_ci;  /* array
    of CallInfo's */ lu_byte status;
///一些要用到的 len，栈大小，c 嵌套的数量，等。 int stacksize; int size_ci;  /* size of array
    `base_ci' */ unsigned short nCcalls;  /* number of nested C calls */ unsigned short baseCcalls;  /*
    nested C calls when resuming coroutine */ lu_byte hookmask; lu_byte allowhook; int
    basehookcount; int hookcount; lua_Hook hook;

///一些全局(这个状态)用到的东西，比如 env 等。
    TValue l_gt;  /* table of globals */
    TValue env;  /* temporary place for environments */

///gc 相关的东西。
    GCObject *openupval;  /* list of open upvalues in this stack */ GCObject
    *gclist;

///错误处理相关。
    struct lua_longjmp *errorJmp;  /* current error recover point */ ptrdiff_t errfunc;  /* current error handling
    function (stack index) */
};
```

而 global_State 主要就是包含了 gc 相关的东西。现在基本类型的分析就告一段落了，等到后面分析 parse 以及 gc 的时候会再回到这些类型。

1.3 lua 源码剖析(三)

发表时间: 2009-12-20

这次

简单的补充一下前面类型部分剩下的东西。

首先我们要知道当我们想为 lua 来编写扩展的时候，有时候可能需要一些全局变量。可是这样会有问题，这是因为这样的话，我们就无法用于多个 lua 状态(也就是 new 多个 state).

于是 lua 提供了三种可以代替全局变量的方法。分别是注册表，环境变量和 upvalue。

其中注册表和环境变量都是 table。而 upvalue 也就是我们前面介绍的用来和指定函数关联的一些值。

由于 lua 统一了从虚拟的栈上存取数据的接口，而这三个值其实并不是在栈上保存，而 lua 为了统一接口，通过伪索引来存取他们。接下来我们就会通过函数 index2adr 的代码片断来分析这三个类型。

其实还有一种也是伪索引来存取的，那就是全局状态。也就是 state 的 l_gt 域。

ok，我们来看这几种伪索引的表示，每次传递给 index2adr 的索引就是下面这几个：

```
#define LUA_REGISTRYINDEX      (-10000)
#define LUA_ENVIRONINDEX       (-10001)
#define LUA_GLOBALSINDEX       (-10002)

///这个就是来存取  upvalue。
#define lua_upvalueindex(i)    (LUA_GLOBALSINDEX-(i))
```

来看代码,这个函数我们前面有分析过，只不过跳过了伪索引这部分，现在我们来看剩下的部分。

其实很简单，就是通过传递进来的 index 来确定该到哪部分处理。这里他们几个处理有些不同，这是因为注册表是全局的（不同模块也能共享)，环境变量可以是整个 lua_state 共享，也可以只是这个函数所拥有。而 upvalue 只能属于某个函数。

看下它们所在的位置，他们的作用域就很一目了然了。

其中注册表包含在 global_State 中，环境变量 closure 和 state 都有，upvalue 只在 closure 中包含。

```
static TValue *index2adr (lua_State *L, int idx) { .....................
   else switch (idx) {  /* pseudo-indices */
///注册表读取
       case LUA_REGISTRYINDEX: return registry(L);
///环境变量的存取
       case LUA_ENVIRONINDEX: {
///先得到当前函数
          Closure *func = curr_func(L);
///将当前函数的  env 设置为整个  state 的  env。这样整个模块都可以共享。
          sethvalue(L, &L->env, func->c.env); return &L->env;
       }
///用来取  global_State。
       case LUA_GLOBALSINDEX: return gt(L);


///取  upvalue default: {
///取得当前函数
          Closure *func = curr_func(L);
///转换索引
          idx = LUA_GLOBALSINDEX - idx;
///从  upvalue 数组中取得对应的值。
          return (idx <= func->c.nupvalues)
                        ? &func->c.upvalue[idx-1]
                        : cast(TValue *, luaO_nilobject);
       }
   }
}
```

下面就是取得环境变量和注册表的对应的宏。

```
#define registry(L)     (&G(L)->l_registry)
#define gt(L)   (&L->l_gt)
```

我们一个个的来看，首先是注册表。由于注册表是全局的，所以我们需要很好的选择 key，尽量避免冲突，而在选择 key 中，不能使用数字类型的 key，这是因为在 lua 中，数字类型的 key 是被引用系统所保留的。

来看引用系统，我们编写 lua 模块时可以看到所有的值，函数，table，都是在栈上保存着，也就是说它们都是由
lua 来管理，我们要存取只能通过栈来存取。可是 lua 为了我们能够在 c 这边保存一个 lua 的值的指针，提供了 luaL_ref 这个函数。

引用也就是在 c 这边保存 lua 的值对象。

来看引用的实现，可以看到它是传递 LUA_REGISTRYINDEX 给 luaL_ref 函数，也就是说引用也是全局的，保存在注册表中的。

```
#define lua_ref(L,lock) ((lock) ? luaL_ref(L, LUA_REGISTRYINDEX) : \
        (lua_pushstring(L, "unlocked references are obsolete"), lua_error(L), 0))
```

然后来看它的 key 的计算。

可以看到当要引用的值是 nil 时，直接返回 LUA_REFNIL 这个常量，并不会创建新的引用。

还有一个要注意的就是这里注册表有个 FREELIST_REF 的 key，这个 key 所保存的值就是我们最后一次 unref 掉的那个 key。我们接下来看 luaL_unref 的时候会看到。

这里为什么要这么做呢，这是因为在注册表中 key 是不能重复的，因此这里的 key 的选择是通过注册表这个 table 的大小来做 key 的，而这里每次 unref 之后我们通过设置 t[FREELIST_REF]的值为上一次被 unref 掉的引用的 key。
这样当我们再次需要引用的时候，我们就不需要增长 table 的大小并且也不需要再次计算 key，而是直接将上一次被 unref 掉得 key 返回就可以了。而这里上上一次被 unref 掉得 ref 的 key 是被保存在 t[ref]中的。我们先来看 luaL_unref 的实现。

```
LUALIB_API void luaL_unref (lua_State *L, int t, int ref) { if (ref >= 0) {
///取出注册表的 table
      t = abs_index(L, t);
///得到 t[FREELIST_REF]; lua_rawgeti(L, t,
      FREELIST_REF);
///这里可以看到如果再次 unref 的话 t[ref]就保存就的是上上一次的 key 的值。
      lua_rawseti(L, t, ref);  /* t[ref] = t[FREELIST_REF] */

///将 ref 压入栈
      lua_pushinteger(L, ref);
///设置 t[FREELIST_REF］为 ref。
      lua_rawseti(L, t, FREELIST_REF);  /* t[FREELIST_REF] = ref */
   }
}
```

通过上面可以看到 lua 这里实现得很巧妙，通过表的 t[FREELIST_REF]来保存最新的被 unref 掉得 key，t[ref]来保存上一次被 unref 掉得 key.然后我们就可以通过这个递归来得到所有已经被 unref 掉得 key。接下来的 luaL_ref 就可以清晰的看到这个操作。也就是说 t[FREELIST_REF]相当于一个表头。

来看 luaL_ref,这个流程很简单，就是先取出注册表的那个 table，然后将得到 t[FREELIST_REF]来看是否有已经 unref 掉得 key，如果有则进行一系列的操作(也就是上面所说的，将这个 ref 从 freelist 中 remove，然后设置 t[FREELIST_REF]为上上一次 unref 掉得值(t[ref])),最后设置 t[ref]的值。这样我们就不需要遍历链表什么的。

这里要注意就是调用这个函数之前栈的最顶端保存的就是我们要引用的值。

```
LUALIB_API int luaL_ref (lua_State *L, int t) { int ref;
///取得索引
   t = abs_index(L, t); if
   (lua_isnil(L, -1)) {
```

```
    lua_pop(L, 1);  /* remove from stack */ ///如果为 nil，则直接返回 LUA_REFNIL.
    return LUA_REFNIL;
  }
///得到 t[FREELIST_REF]. lua_rawgeti(L, t,
  FREELIST_REF);
///设置 ref = t[FREELIST_REF] ref =
  (int)lua_tointeger(L, -1);
///弹出 t[FREELIST_REF] lua_pop(L, 1);  /* remove it from stack */

///如果 ref 不等于 0,则说明有已经被 unref 掉得 key。 if
  (ref != 0) {  /* any free element? */
///得到 t[ref]，这里 t[ref]保存就是上上一次被 unref 掉得那个 key。
    lua_rawgeti(L, t, ref);  /* remove it from list */
///设置 t[FREELIST_REF] = t[ref],这样当下次再进来，我们依然可以通过 freelist 来直接返回 key。
    lua_rawseti(L, t, FREELIST_REF);
  } else {  /* no free elements */
///这里是通过注册表的大小来得到对应的 key
    ref = (int)lua_objlen(L, t); ref++;  /* create new reference */
  }

//设置 t[ref]=value; lua_rawseti(L, t, ref); return ref;
}
```

所以我们可以看到我们如果要使用注册表的话，尽量不要使用数字类型的 key，不然的话就很容易和引用系统冲突。

不过在 PIL 中介绍了一个很好的 key 的选择，那就是使用代码中静态变量的地址（也就是用 light userdata)，因为 c 链接器可以保证 key 的唯一性。详细的东西可以去看 PIL. 然后我们来看 LUA_ENVIRONINDEX,环境是可以被整个模块共享的。可以先看 PIL 中的例子代码：

```
int luaopen_foo(lua_State *L) { lua_newtable(L);
```

```
lua_replace(L,LUA_ENVIRONIDEX); luaL_register(L,<lib name>,<func list>);
..... }
```

可以看到我们一般都是为当前模块创建一个新的 table，然后当 register 注册的所有函数就都能共享这个 env 了。来看代码片断，register 最终会调用 luaI_openlib：

```
LUALIB_API void luaI_openlib (lua_State *L, const char *libname,const luaL_Reg *l, int nup) { .....
///遍历模块内的所有函数。
   for (; l->name; l++) { int i; for (i=0; i<nup; i++)  /* copy upvalues to the top
       */ lua_pushvalue(L, -nup);
///这里将函数压入栈，这个函数我们前面分析过，他最终会把当前 state 的 env 赋值给新建的 closure，也就是说这里最
       lua_pushcclosure(L, l->func, nup); lua_setfield(L, -(nup+2), l->name);
   } lua_pop(L, nup);  /* remove upvalues */
}
```

通过我们一开始分析的代码，我们知道当我们要存取环境的时候每次都是将当前调用的函数的 env 指针赋值给 state 的 env，然后返回 state 的 env(&L->env)。这是因为 state 是被整个模块共享的，每个函数修改后必须与 state 的那个同步。最后我们来看 upvalue。这里指的是 c 函数的 upvalue，我们知道在 lua 中 closure 分为两个类型，一个是 c 函数，一个是 lua 函数，我们现在主要就是来看 c 函数。 c 函数的 upvalue 和 lua 的类似，也就是将我们以后函数调用所需要得一些值保存在 upvalue 中。

这里一般都是通过 lua_pushcclosure 这个函数来做的。下面先来看个例子代码：

```
static int counter(lua_state *L);

int newCounter(lua_State *L)
{ lua_pushinteger(L,0); lua_pushcclosure(L,&counter,1); return 1;
}
```

上面的代码很简单，就是先 push 进去一个整数 0,然后再 push 一个 closure，这里 closure 的第三个参数就是 upvalue 的个数(这里要注意在 lua 中的 upvalue 的个数只有一个字节，因此你太多 upvalue 会被截断)。

lua_pushcclosure 的代码前面已经分析过了，我们这里简单的再介绍一下。

这个函数每次都会新建一个 closure，然后将栈上的对应的 value 拷贝到 closure 的 upvalue 中，这里个数就是它的第三个参数来确定的。而取得 upvalue 也很简单，就是通过 index2adr 来计算对应的 upvalue 中的索引值，最终返回对应的值。

然后我们来看 light userdata，这种 userdata 和前面讲得 userdata 的区别就是这种 userdata 的管理是交给 c 函数这边来管理的。这个实现很简单，由于它只是一个指针，因此只需要将这个值压入栈就可以了。

```
LUA_API void lua_pushlightuserdata (lua_State *L, void *p) { lua_lock(L);
///设置对应的值。
    setpvalue(L->top, p); api_incr_top(L); lua_unlock(L);
}
```

最后我们来看元表。我们知道在 lua 中每个值都有一个元表，而 table 和 userdata 可以有自己独立的元表，其他类型的值共享所属类型的元表。在 lua 中可以使用 setmetatable.而在 c 中我们是通过 luaL_newmetatable 来创建一个元表。
元表其实也就是保存了一种类型所能进行的操作。

这里要知道在 lua 中元表是保存在注册表中的。

因此我们来看 luaL_newmetatable 的实现。
这里第二个函数就是当前所要注册的元表的名字。这里一般都是类型名字。这个是个 key，因此我们一般要小心选择类型名。

```
LUALIB_API int luaL_newmetatable (lua_State *L, const char *tname) {
///首先从注册表中取得 key 为 tname 的元表 lua_getfield(L,
   LUA_REGISTRYINDEX, tname);
///如果存在则失败，返回 0
   if (!lua_isnil(L, -1))  /* name already in use? */ return 0; lua_pop(L,
   1);
///创建一个元表
   lua_newtable(L);  /* create metatable */
///压入栈
   lua_pushvalue(L, -1);
///设置注册表中的对应的元表。
   lua_setfield(L, LUA_REGISTRYINDEX, tname);

   return 1;
}
```

当我们设置完元表之后我们就可以通过调用 luaL_checkudata 来检测栈上的 userdata 的元表是否和指定的元表匹配。

这里第二个参数是 userdata 的位置，tname 是要匹配的元表的名字。

这里我们要知道在 lua 中，Table 和 userdata 中都包含一个 metatable 域，这个也就是他们对应的元表，而基本类型的元表是保存在 global_State 的 mt 中的。这里 mt 是一个数组。

这里我们先来看 lua_getmetatable,这个函数返回当前值的元表。这里代码很简单，就是取值，然后判断类型。最终返回设置元表。

```
LUA_API int lua_getmetatable (lua_State *L, int objindex) { const TValue *obj;
   Table *mt = NULL; int res; lua_lock(L);
///取得对应索引的值
   obj = index2adr(L, objindex);
///开始判断类型。
   switch (ttype(obj)) {
///table 类型 case
       LUA_TTABLE:
          mt = hvalue(obj)->metatable; break;
///userdata 类型 case
       LUA_TUSERDATA: mt =
          uvalue(obj)->metatable; break;
       default:
///这里是基础类型
          mt = G(L)->mt[ttype(obj)]; break;

   } if (mt == NULL) res
   = 0;
   else {
///设置元表到栈的 top
       sethvalue(L, L->top, mt); api_incr_top(L);
       res = 1;
   } lua_unlock(L); return
   res;
}
```

接下来来看 checkudata 的实现。他就是取得当前值的元表，然后取得 tname 对应的元表，最后比较一下。

```
LUALIB_API void *luaL_checkudata (lua_State *L, int ud, const char *tname) { void *p = lua_touserdata(L,
   ud); if (p != NULL) {  /* value is a userdata? */
///首先取得当前值的元表。
      if (lua_getmetatable(L, ud)) {
///然后取得  taname 对应的元表。
         lua_getfield(L, LUA_REGISTRYINDEX, tname);
//比较。
         if (lua_rawequal(L, -1, -2)) { lua_pop(L, 2);  /* remove both metatables
            */ return p;
         }
      } } luaL_typerror(L, ud, tname);  /* else error */ return

   NULL;  /* to avoid warnings */
}
```

Lua 源码剖析（四）

这篇主要来分析 lua 的虚拟机的实现，我看的代码依旧是 5.1

因此首先从 luaL_loadfile 开始，这个函数我们知道是在当前的 lua state 加载一个 lua 文件，其中第二个参数就是 filename。其中 LoadF 结构很简单，它用来表示一个 load file：

```
typedef struct LoadF {
  int extraline;
FILE *f;
  char buff[LUAL_BUFFERSIZE]; }
LoadF;
```

其中会使用 fopen 来打开对应的文件名,然后根据第一个字符来判断是否是注释(#)，如果是则跳过

```
    lua_pushfstring(L, "@%s", filename);
    lf.f = fopen(filename, "r");
    if (lf.f == NULL) return errfile(L, "open", fnameindex);
    }
    c = getc(lf.f);
    if (c == '#') {  /* Unix exec. file? */
    lf.extraline = 1;
    while ((c = getc(lf.f)) != EOF && c != '\n') ;  /* skip first li
    if (c == '\n') c = getc(lf.f);
}
```

然后会判断是否是 lua 字节码文件，通过文件头的 magic number(Lua),如果是，则用二进制打开。

```
   if (c == LUA_SIGNATURE[0] && filename) {  /* binary file? */
   lf.f = freopen(filename, "rb", lf.f);  /* reopen in binary mode */
   if (lf.f == NULL) return errfile(L, "reopen", fnameindex);
   /* skip eventual `#!...' */
   while ((c = getc(lf.f)) != EOF && c != LUA_SIGNATURE[0]) ;
   lf.extraline = 0;
   }
```

最后就是调用 lua_load 来 load 文件。这里有一个结构要注意，那就是 zio，这个结构就是一个 parse 的结构。这里的 reader 就是一个回调函数。

```
struct Zio {
```

```
  size_t n;                /* bytes still unread */   const
char *p;           /* current position in buffer */
lua_Reader reader;
  void* data;               /* additional data */
  lua_State *L;               /* Lua state (for reader) */ };
```

可以看看这个结构如何被初始化。

```
void luaZ_init (lua_State *L, ZIO *z, lua_Reader reader, void *data
  z->L = L;
  z->reader = reader;
  z->data = data;
  z->n = 0;
  z->p = NULL;
}
```

先看看 lua 的 BNF

chunk ::= {stat [`;´]} [laststat [`;´]] block ::= chunk stat ::= varlist `=´ explist | functioncall | do block end | while exp do block end | repeat block until exp | if exp then block {elseif exp then block} [else block] end | for Name `=´ exp `,´ exp [`,´ exp] do block end | for namelist in explist do block end | function funcname funcbody | local function Name funcbody | local namelist [`=´ explist] laststat ::= return [explist] | break funcname ::= Name {`.´ Name} [`:´ Name] varlist ::= var {`,´ var} var ::= Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name namelist ::= Name {`,´ Name} explist ::= {exp `,´} exp exp ::= nil | false | true | Number | String | `...´ | function | prefixexp | tableconstructor | exp binop exp | unop exp prefixexp ::= var | functioncall | `(´ exp `)´ functioncall ::= prefixexp args | prefixexp `:´ Name args args ::= `(´ [explist] `)´ | tableconstructor | String function ::= function funcbody funcbody ::= `(´ [parlist] `)´ block end parlist ::= namelist [`,´ `...´] | `...´ tableconstructor ::= `{´ [fieldlist] `}´ fieldlist ::= field {fieldsep field} [fieldsep] field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp fieldsep ::= `,´ | `;´ binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ | `<´ | `<=´ | `]]>´ | `>=´ | `==´ | `~=´ | and | or unop ::= `-´ | not | `#´

可以看到 reader 就是被初始化为 getF，而 data 是被初始化位上面的 LoadF 结构，然后就是调用 luaD_protectedparser 来 parse 源代码。

在初始化的时候调用 luaX_init 将保留关键字作为字符串放入全局的 string hash 中(ts->reserved)，因此当解析到字符串时，能够很容易的到当前字符串的类型。

解析函数是 luaY_parser, 而核心的 parse 方法是 llex，这个函数会返回对应的 token，然后根据解析出来的 token，来决定接下来理应是什么符号(核心在 chunk 函数里面,这函数有一个循环), 然后每次都会调用 luaX_next 函数，来继续解析(它会调用 llex).

```
25        Proto *luaY_parser (lua_State *L, ZIO *z, Mbuffer *buff, const cha
26        struct LexState lexstate;   struct FuncState funcstate;
27        lexstate.buff = buff;
28          luaX_setinput(L, &lexstate, z, luaS_new(L, name));
29        open_func(&lexstate, &funcstate);
30          funcstate.f->is_vararg = VARARG_ISVARARG; /* main func. is alway
31        luaX_next(&lexstate); /* read first token */   chunk(&lexstate);
32        check(&lexstate, TK_EOS);   close_func(&lexstate);
33          lua_assert(funcstate.prev == NULL);
34  lua_assert(funcstate.f->nups == 0);   lua_assert(lexstate.fs ==
35  NULL);   return funcstate.f; } 可以看到最终 luaY_parser 返回的是一个 proto，也
36  就是说每一个 lua 文件也就相当于一个 proto(函数). 下面湿核心的 chunk 方法.
```

```
static void chunk (LexState *ls) {
  /* chunk -> { stat [`;'] } */
int islast = 0;   enterlevel(ls);
  while (!islast && !block_follow(ls->t.token))
{     islast = statement(ls);     testnext(ls,
';');
    lua_assert(ls->fs->f->maxstacksize >= ls->fs->freereg &&
ls->fs->freereg >= ls->fs->nactvar);
    ls->fs->freereg = ls->fs->nactvar;  /* free registers */
  }
  leavelevel(ls);
}


```

lua 里面 token 定义为：

```
  enum RESERVED {
  /* terminal symbols denoted by reserved words */
  TK_AND = FIRST_RESERVED, TK_BREAK,
  TK_DO, TK_ELSE, TK_ELSEIF, TK_END, TK_FALSE, TK_FOR, TK_FUNCTION
  TK_IF, TK_IN, TK_LOCAL, TK_NIL, TK_NOT, TK_OR, TK_REPEAT,
  TK_RETURN, TK_THEN, TK_TRUE, TK_UNTIL, TK_WHILE,
  /* other terminal symbols */
  TK_CONCAT, TK_DOTS, TK_EQ, TK_GE, TK_LE, TK_NE, TK_NUMBER,
  TK_NAME, TK_STRING, TK_EOS
}
```

可以看到上方是关键字，而下方是其他的一些终结符。核心的 parse 结构 LexState，它主要是用于 parse 期间保存状态。

```
  typedef struct LexState {
```

```
typedef union {
  lua_Number r;
  TString *ts;
} SemInfo;  /* semantics information */
```

```
5
6
7  typedef struct Token
8  {   int token;
9  SemInfo seminfo;
9  } Token;
10
2    int current;  /* current character (charint) */
3    int linenumber;  /* input line counter */
4    int lastline;  /* line of last token `consumed' */
5    Token t;  /* current token */
6    //表示前一个 token
7    Token lookahead;  /* look ahead token */
8    struct FuncState *fs;  /* `FuncState' is private to the parser */ 9
     struct lua_State *L;
10   ZIO *z;  /* input stream */
11   Mbuffer *buff;  /* buffer for tokens */
12   TString *source;  /* current source name */
13   char decpoint;  /* locale decimal point */ 14 } LexState;
```

上面这个结构中最需要注意的是 Token 结构，这个结构保存了对应的解析出来的 token。seminfo 保存了经过词法解析后的词(不包括运算符以及[]等),只包括字母以及数字 number and string。在 lua 中所有的 tiken 分为 13 类，也就是 BNF 中的 binop(不包括 and or)+各种关键字。然后就是每个解析出来的函数的数据结构，这个结构保存了解析到的函数的状态。

```
1      typedef struct FuncState {
2      Proto *f;  /* current function header */
3      Table *h;  /* table to find (and reuse) elements in `k' */
4      struct FuncState *prev;  /* enclosing function */
5      struct LexState *ls;  /* lexical state */
6      struct lua_State *L;  /* copy of the Lua state */
7      struct BlockCnt *bl;  /* chain of current blocks */
8      int pc;  /* next position to code (equivalent to `ncode') */
9      int lasttarget;   /* `pc' of last `jump target' */
10     int jpc;  /* list of pending jumps to `pc' */
11     int freereg;  /* first free register */
12     int nk;  /* number of elements in `k' */
13     int np;  /* number of elements in `p' */
14     short nlocvars;  /* number of elements in `locvars' */
15     lu_byte nactvar;  /* number of active local variables */
16     upvaldesc upvalues[LUAI_MAXUPVALUES];  /* upvalues */
17     unsigned short actvar[LUAI_MAXVARS];  /* declared-variable stack
       18 } FuncState;
```

上面这几个结构的初始化都是放在 luaY_parser 中的，llex 方法其实只是一个词法扫描器，它只负责扫描符号以及单词，然后它会将扫描到的符号交给后续的语法分析器去判断。其中 luaX_setinput 用于初始化设置 lexState 的一些输入属性。

```
1       void luaX_setinput (lua_State *L, LexState *ls, ZIO *z, TString *s
2       ls->decpoint = '.';
3       ls->L = L;
4       //前一个 token
5       ls->lookahead.token = TK_EOS;  /* no look-ahead token */
6       ls->z = z;
7       ls->fs = NULL;
8       ls->linenumber = 1;
9       ls->lastline = 1;
10      ls->source = source;
11      luaZ_resizebuffer(ls->L, ls->buff, LUA_MINBUFFER);  /* initializ
12      //第一次读取 token
13      next(ls);  /* read first char */
14  }
```

而 open_func 函数则是新建一个 func 对象.

```
1   static void open_func (LexState *ls, FuncState *fs) {
2     lua_State *L = ls->L;   Proto
3   *f = luaF_newproto(L);   fs->f
4   = f;
5     fs->prev = ls->fs;  /* linked list of funcstates */
6   fs->ls = ls;   fs->L = L;
7
```

```
  ls->fs = fs;   fs->pc =
0;   fs->lasttarget = -1;
fs->jpc = NO_JUMP;
fs->freereg = 0;   fs->nk
= 0;   fs->np = 0;
fs->nlocvars = 0;
fs->nactvar = 0;   fs->bl
= NULL;   f->source =
ls->source;
  f->maxstacksize = 2;  /* registers 0/1 are always valid */
fs->h = luaH_new(L, 0, 0);
  /* anchor table of constants and prototype (to avoid being colle
sethvalue2s(L, L->top, fs->h);   incr_top(L);
  setptvalue2s(L, L->top, f);
incr_top(L); }
```

然后会在 statement 函数中对语句进行语法解析，然后不同的函数解析不同的语句。

assignment 函数用于解析赋值语法，而它会调用 exploits1，explist1 对应的解析 explist1 -> expr { `,' expr }

```
typedef struct expdesc {
```

```
  expkind k;
union {
    struct { int info, aux; } s;
lua_Number nval;
  } u;
  int t; /* patch list of `exit when true' */
int f; /* patch list of `exit when false' */
```

，因此在它里面就会调用 expr 函数，而在 expr 中是直接调用 subexpr 函数，它主要是解析这类语法 subexpr
-> (simpleexp | unop subexpr) { binop subexpr } . 因此在 subexpr 中就会调用三个函数，分别是 simpleexp
(simpleexp -> NUMBER | STRING | NIL | true | false | ... | constructor | FUNCTION body | primaryexp)，
getunopr 来解析一元操作符，getbinopr 来解析二元操作符。 在 simpleexp 中，则会调用 primaryexp 来解析 primaryexp -> prefixexp { `.' NAME | `[' exp `]' | `:' NAME funcargs | funcargs } 这类语法。每次使用 checknext 方法来检测下一个 token 是否是期望的。

```
static void check (LexState *ls, int c) {
  if (ls->t.token != c)
error_expected(ls, c);
}
static void checknext (LexState *ls, int c)
{   check(ls, c);   luaX_next(ls);
}
```

lua 的自顶向下的解析最核心的就是 lcode.c 和 lparse.c.其中 lua 对待一个文件就是把这个文件当做一个函数，因此在解析完毕之后，返回的就是一个 proto 指针。对应的 lparse 是解析 lua 源代码，而 lcode.c 则将对应的源码翻译为虚拟机指令(保存在 proto 的 code 数组中).

在 lua 中每一个表达式解析完毕后就表示为一个这样的结构体：

```
} expdesc;
```

这里最重要的就是联合体 u，其中 info 表示语句对应的索引(在指令数组 code(proto 结构体)中).或者说表示为对应 constant 的索引(proto 的 TValue *k 这个数组的索引).这就体现了程序是由数据+指令组成。

lua 的虚拟机指令的格式：

```
      /*===========================================================================
        We assume that instructions are unsigned numbers.
        All instructions have an opcode in the first 6 bits.
```

```
76          Instructions can have the following fields:
77        `A' : 8 bits
78        `B' : 9 bits
79        `C' : 9 bits
80        `Bx' : 18 bits (`B' and `C' together)
81        `sBx' : signed Bx
82
83          A signed argument is represented in excess K; that is, the numbe
84        value is the unsigned value minus K. K is exactly the maximum va
85        for that argument (so that -max is represented by 0, and +max is
86        represented by 2*max), which is half the maximum for the corresp
87        unsigned argument.
88        ===========================================================*/
```

```
89  1 enum OpMode {iABC, iABx, iAsBx}; /* basic instruction format */
```

90 可以看到也就是 3 种指令，而下面的宏定义了指令的参数以及 opcode 的大小以及偏移，这里可以看到

91 在 lua 里面所有的指令都是定长的。

```
1  #define SIZE_C9
2  #define SIZE_B9
3  #define SIZE_Bx(SIZE_C + SIZE_B)
4  #define SIZE_A8
5
6  #define SIZE_OP6
7
8  #define POS_OP0
9  #define POS_A(POS_OP + SIZE_OP)
10 #define POS_C(POS_A + SIZE_A)
11 #define POS_B(POS_C + SIZE_C)
12 #define POS_BxPOS_C
```

92 再接下来就是如何取得对应的 opcode 以及各个参数：

```
1  /* creates a mask with `n' 1 bits at position `p' */
2  #define MASK1(n,p)((~((~(Instruction)0)<<n))<<p)
3
4  /* creates a mask with `n' 0 bits at position `p' */
5  #define MASK0(n,p)(~MASK1(n,p))
6
7  /*
8  ** the following macros help to manipulate instructions
9  */
10
11 #define GET_OPCODE(i)(cast(OpCode, ((i)>>POS_OP) & MASK1(SIZE_OP,0
```

93

```
12 #define SET_OPCODE(i,o)((i) = (((i)&MASK0(SIZE_OP,POS_OP)) | \ 13
((cast(Instruction, o)<<POS_OP)&MASK1(SIZE_OP,POS_OP))))
```

对应的上面就是如何取得对应指令 opcode 的宏。下面就是所有的 opcode 类型。

```
1  typedef enum {
```

```
/*------------------------------------------------------------------
nameargsdescription
------------------------------------------------------------------*/
OP_MOVE,/*A BR(A) := R(B)*/
OP_LOADK,/*A BxR(A) := Kst(Bx)*/
OP_LOADBOOL,/*A B CR(A) := (Bool)B; if (C) pc++*/
OP_LOADNIL,/*A BR(A) := ... := R(B) := nil*/
OP_GETUPVAL,/*A BR(A) := UpValue[B]*/

OP_GETGLOBAL,/*A BxR(A) := Gbl[Kst(Bx)]*/
OP_GETTABLE,/*A B CR(A) := R(B)[RK(C)]*/
OP_SETGLOBAL,/*A BxGbl[Kst(Bx)] := R(A)*/
OP_SETUPVAL,/*A BUpValue[B] := R(A)*/
OP_SETTABLE,/*A B CR(A)[RK(B)] := RK©*/

OP_NEWTABLE,/*A B CR(A) := {} (size = B,C)*/

OP_SELF,/*A B CR(A+1) := R(B); R(A) := R(B)[RK(C)]*/

OP_ADD,/*A B CR(A) := RK(B) + RK©*/
OP_SUB,/*A B CR(A) := RK(B) - RK©*/
OP_MUL,/*A B CR(A) := RK(B) * RK©*/
OP_DIV,/*A B CR(A) := RK(B) / RK©*/
OP_MOD,/*A B CR(A) := RK(B) % RK©*/
OP_POW,/*A B CR(A) := RK(B) ^ RK©*/
OP_UNM,/*A BR(A) := -R(B)*/
OP_NOT,/*A BR(A) := not R(B)*/
OP_LEN,/*A BR(A) := length of R(B)*/

OP_CONCAT,/*A B CR(A) := R(B).. ... ..R©*/

OP_JMP,/*sBxpc+=sBx*/

OP_EQ,/*A B Cif ((RK(B) == RK(C)) ~= A) then pc++*/
OP_LT,/*A B Cif ((RK(B) < RK(C)) ~= A) then pc++ */
OP_LE,/*A B Cif ((RK(B) <= RK(C)) ~= A) then pc++ */

OP_TEST,/*A Cif not (R(A) <=> C) then pc++*/
OP_TESTSET,/*A B Cif (R(B) <=> C) then R(A) := R(B) else pc++*/

OP_CALL,/*A B CR(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
OP_TAILCALL,/*A B Creturn R(A)(R(A+1), ... ,R(A+B-1))*/
OP_RETURN,/*A Breturn R(A), ... ,R(A+B-2)(see note)*/

OP_FORLOOP,/*A sBxR(A)+=R(A+2);
   if R(A) <?= R(A+1) then { pc+=sBx; R(A+3)=R(A) }*/
OP_FORPREP,/*A sBxR(A)-=R(A+2); pc+=sBx*/
```

```
48
49  OP_TFORLOOP,/*A CR(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
50
51
```

```
                                  if R(A+3) ~= nil then R(A+2)=R(A+3) else p
    OP_SETLIST,/*A B CR(A)[(C-1)*FPF+i] := R(A+i), 1 <= i <= B*/

    OP_CLOSE,/*A close all variables in the stack up to (>=) R(A)*/
    OP_CLOSURE,/*A BxR(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))*/

    OP_VARARG/*A BR(A), R(A+1), ..., R(A+B-1) = vararg*/ }
    OpCode;
```

然后我们来看一个最简单的语句是如何解析进 lua 的虚拟机的，假设有一条 local t = 3 的语句。这个会通

过 parse 进入 localstat，然后调用 new_localvar 来新建一个 local 变量，所有的 local 变量都放在 Proto 结构的 f->locvars 中，locvars 也就是一个数组

```
1 struct LocVar *locvars; /* information about local variables */
```

数组索引放到 FuncState 的 actvar 数组中。因为在 lua 中可以这么写 local t, t1 = 3。解析完毕名字之后，就开始解析=，解析=会在 explist1 函数中进行。然后会讲解析掉的数字 3 放入到表达式(expdesc)的 nval 域。

```
typedef struct expdesc {
  expkind k;
union {
    struct { int info, aux; } s;
lua_Number nval;
  } u;
  int t; /* patch list of `exit when true' */
int f; /* patch list of `exit when false' */
} expdesc;


    case TK_NUMBER:
{       init_exp(v, VKNUM, 0);
v->u.nval = ls->t.seminfo.r;
break;
    }

```

这里可以看到直接赋值给 nval。

然后就是 adjust_assign 以及 adjustlocalvars 两个函数，其中 adjust_assign 将会调用 luaK_exp2nextreg 函数，而

在这个函数中则会调用 discharge2reg 函数来设置对应的指令， 这里比较关键的，就是 lua 如何来选择寄存器。

```
void luaK_reserveregs (FuncState *fs, int n) {
  luaK_checkstack(fs, n);
fs->freereg += n;
}
void luaK_exp2nextreg (FuncState *fs, expdesc *e)
{   luaK_dischargevars(fs, e);   freeexp(fs, e);
luaK_reserveregs(fs, 1);
  exp2reg(fs, e, fs->freereg - 1);
}
static void exp2reg (FuncState *fs, expdesc *e, int reg) {
```

```
14  discharge2reg(fs, e, reg);
15 ..............
16 }
17
18 static void discharge2reg (FuncState *fs, expdesc *e, int reg) { 19
luaK_dischargevars(fs, e); 20 ........................
21         case VKNUM: {
22         luaK_codeABx(fs, OP_LOADK, reg, luaK_numberK(fs, e->u.nval))
23         break; 24     }
25     ........................
26     e->u.s.info = reg;
27     e->k = VNONRELOC;
28 }
```

通过上面我们可以看到寄存器的选择是通过 funcstate 的 freereg 域来进行选择的。最终我们可以看到当字节码生成之后，e->u.s.info 中放的就是寄存器了。而在一开始进入 luaK_exp2nextreg 之后，立即对 freereg 进行+1,这里寄存器我们可以看做是一个索引. 接下来我们来看 luaK_numberK 的实现，也就是 lua 会把 constant 放到哪里去。

```
1     int luaK_numberK (FuncState *fs, lua_Number r) {
2     TValue o;
3     setnvalue(&o, r);
4     return addk(fs, &o, &o);
5     }
6
7
8       static int addk (FuncState *fs, TValue *k, TValue *v) {
9       lua_State *L = fs->L;
10      TValue *idx = luaH_set(L, fs->h, k);
11      Proto *f = fs->f;
12      int oldsize = f->sizek;
13      if (ttisnumber(idx)) {
14      lua_assert(luaO_rawequalObj(&fs->f->k[cast_int(nvalue(idx))], v
15      return cast_int(nvalue(idx));
16 }
17                      else {  /* constant not found; create a new entry
                        */
18                      setnvalue(idx, cast_num(fs->nk));
19                      luaM_growvector(L, f->k, fs->nk, f->sizek,
                        TValue,
20                      MAXARG_Bx, "constant table overflow");
21                      while (oldsize < f->sizek)
                        setnilvalue(&f->k[oldsize++]);
22                      //存储值
```

```
23                          setobj(L, &f->k[fs->nk], v);
24                          luaC_barrier(L, f, v);
25                          return fs->nk++;
26  } 27 }
```

可以看到最终值会放到 f->k 这个数组中。并且会返回对应的索引，然后讲索引保存到字节码中。

这个时候可以看到这条语句对应的字节码是 LOADK, 而 loadk 对应的指令类型是 ABx，我们来看对应的域都填充的是什么。

```
1     int luaK_codeABx (FuncState *fs, OpCode o, int a, unsigned int bc)
2     lua_assert(getOpMode(o) == iABx || getOpMode(o) == iAsBx);
3     lua_assert(getCMode(o) == OpArgN);
4     return luaK_code(fs, CREATE_ABx(o, a, bc), fs->ls->lastline);
5     }
```

可以看到对应的 a 填充到 ABx 类型的 A，而 bc 则填充到 bx 域，而 a 是什么呢？通过上面的代码可以看到 a 就是寄存器，bc 就是对应的值.也就是说讲 3 这个值放到 bx，而把 t 对应的寄存器 放到 a 域.字节码生成了。然后看来 luaK_code 这个函数,他也就是讲指令放入到 proto 结构体的 code 数组中

```
1                           static int luaK_code (FuncState *fs, Instruction
                            i, int line) {
2                           Proto *f = fs->f;
3                           dischargejpc(fs);  /* `pc' will change */
4                           /* put new instruction in code array */
5                           luaM_growvector(fs->L, f->code, fs->pc,
                            f->sizecode, Instruction
6                           MAX_INT, "code size overflow");
7                           f->code[fs->pc] = i;
8                           /* save corresponding line information */
9                           luaM_growvector(fs->L, f->lineinfo, fs->pc,
                            f->sizelineinfo, int
10                          MAX_INT, "code size overflow");
11                          f->lineinfo[fs->pc] = line;
12                          return fs->pc++;
13  }
```

接下来我们来看 lua 虚拟机会如何来解析字节码。

所有的解析都在 luaV_execute 中。来看对应的代码, luaV_execute 也就是会通过遍历 L->code(索引在 savedpc

中)来执行所有的字节码。

```
void luaV_execute (lua_State *L) {
  CallInfo *ci = L->ci; .................

  pc = L->savedpc;
  cl = &clvalue(L->ci->func)->l;
base = L->base; //取出 constants 数
组   k = cl->p->k;
  /* main loop of interpreter */
for (;;) {
    const Instruction i = *pc++;
StkId ra;
    if ((L->hookmask & (LUA_MASKLINE | LUA_MASKCOUNT)) &&
(--L->hookcount == 0 || L->hookmask & LUA_MASKLINE))
{       traceexec(L, pc);
      if (L->status == LUA_YIELD) {  /* did hook yield? */
        L->savedpc = pc - 1;
return;
      }
      base = L->base;
    }
.....................
ra = RA(i);
```

```
25  ..............
26        case OP_LOADK:
27  {           setobj2s(L, ra, KBx(i));
28  continue;
29        }
30      }
```

这里 ra 就是对应的栈的位置(通过寄存器来确定栈的位置), 然后 KBx 主要是用于得到对应的值(也就是我们例子中的 5).

```
1 #define KBx(i)  check_exp(getBMode(GET_OPCODE(i)) == OpArgK, k+GETA
```

可以看到他是以 k = cl->p->k;为基准值，然后加上对应的偏移.

最后我们来看 setobj2s 这个函数,这个函数其实很简单，就是把从寄存器取出来的值放入到对应的 value 结构中。

```
1  #define setobj(L,obj1,obj2) \
2    { const TValue *o2=(obj2); TValue *o1=(obj1); \
3  o1->value = o2->value; o1->tt=o2->tt; \
4  checkliveness(G(L),o1); }
```

可以看到最终 t 的值被放入到 L->base 为基础的一段内存中。

Lua 源码剖析(五)

这次主要来分析 lua 的 gc。

首先 lua 中的数据类型包括下面 9 种，ni， Boolean， number， string， table，user data，thread , functions 以及 lightusedata.其中 string， table，thread , function 是会被垃圾回收管理的，其他的都是值存在。因此我们来看对应的 GC 数据结构.

```
#define CommonHeader     GCObject *next; lu_byte tt; lu_byte marked

typedef struct GCheader {
  CommonHeader; }
GCheader;

union GCObject
{   GCheader gch;
union TString ts;
union Udata u;
union Closure cl;
struct Table h;
struct Proto p;
struct UpVal uv;
  struct lua_State th;  /* thread */
};
```

我们可以看到在 lua 中字符串，userdata, thread, table ,string, thread(以及 Upval, proto) 都会被垃圾回收管理。这里比较关键的就是 GCheader 这个结构体，我们可以看到这个结构体其实就是一个链表，也就是说所有的 gc 对象都会被链到一个链表中，其中 tt 表示当前对象的类型，在 lua 中包括下面这些类型：

```
#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
```

而 marked 表示当前对象的状态(涉及到 gc 算法，后续会详细分析),状态位包括下面这些：

```
#define WHITE0BIT     0
#define WHITE1BIT     1
#define BLACKBIT     2
#define FINALIZEDBIT     3
#define KEYWEAKBIT     3
#define VALUEWEAKBIT     4
#define FIXEDBIT     5
#define SFIXEDBIT     6
#define WHITEBITS     bit2mask(WHITE0BIT, WHITE1BIT)
```

然后我们来看 lua_state 这个数据结构，这个结构也就是一个 lua 意义上的 thread。

```
struct lua_State
{   CommonHeader;
lu_byte status;
StkId top;  /* first
free slot in the stack
*/   StkId base;  /*
base of current
function */
global_State *l_G;
  CallInfo *ci;  /* call info for current function */
  const Instruction *savedpc;  /* `savedpc' of current function */
  StkId stack_last;  /* last free slot in the stack */
  StkId stack;  /* stack base */
  CallInfo *end_ci;  /* points after end of ci array*/
CallInfo *base_ci;  /* array of CallInfo's */   int
stacksize;
  int size_ci;  /* size of array `base_ci' */   unsigned
short nCcalls;  /* number of nested C calls */
  unsigned short baseCcalls;  /* nested C calls when resuming coroutine */
lu_byte hookmask;   lu_byte allowhook;   int basehookcount;   int
hookcount;   lua_Hook hook;
  TValue l_gt;  /* table of globals */
  TValue env;  /* temporary place for environments */
  GCObject *openupval;  /* list of open upvalues in this stack */
GCObject *gclist;
  struct lua_longjmp *errorJmp;  /* current error recover point */
ptrdiff_t errfunc;  /* current error handling function (stack index) */
};
```

每一个 lua 虚拟机可能会包含很多个 lua_state 结构。而所有的 lua_State 所共享的数据(比如 string，比如 gc 数据等), 都将会放到 global_State 中。

这里要注意所有的 GC 数据中，string 和其他的是不同的，他和其他的 GC 对象分开管理，我们先来看非 string 类的对象如何管理。先来看 global_State 这个结构：

```
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;
```

着重来看 GC 相关的几个数据结构。当前虚拟机的所有的 GC 对象都会保存在一个链表中，这个链表的根

就是 global_State 的 rootgc 中。我们来看 roottgc 的初始化，代码在 lua_newstate 中：

```
  g->rootgc = obj2gco(L);
```

然后每一个被创建的 gc 对象都会被挂载到这个链表中。挂载函数就是 luaC_link。这个函数主要用来将需要 gc 的对象 link 到全局的 global_state 中。

```
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;
}
```

通过上面的函数，我们可以看到每次新的 gc 对象插入的时候，总是放到链表的最前端(rootgc 为当前对象).

不过这里的 upvalue 和 userdata 都是使用另外的方法挂载到全局的涟中的，先来看 upvalue(upvalue 是什么，我这里就不介绍了，前面的 lua 源码分析有介绍过的).

```
void luaC_linkupval (lua_State *L, UpVal *uv) {
  global_State *g = G(L);
GCObject *o = obj2gco(uv);
  o->gch.next = g->rootgc;  /* link upvalue into `rootgc' list */
g->rootgc = o;   if (isgray(o)) {
    if (g->gcstate == GCSpropagate) {
      gray2black(o);  /* closed upvalues need barrier */
luaC_barrier(L, uv, uv->v);
    }
    else {  /* sweep phase: sweep it (turning it into white) */
makewhite(g, o);
      lua_assert(g->gcstate != GCSfinalize && g->gcstate != GCSpause);
    }
  }
}
```

可以看到和 luaC_link 不同的是，进行了 gc 算法的一些操作，这里我们先搁置，后续介绍 gc 算法的时候，会再来看这里。

然后就是 userdata 的特殊处理：

```
Udata *luaS_newudata (lua_State *L, size_t s, Table *e)
{   Udata *u;
  if (s > MAX_SIZET - sizeof(Udata))
luaM_toobig(L);
  u = cast(Udata *, luaM_malloc(L, s + sizeof(Udata)));
u->uv.marked = luaC_white(G(L));  /* is not finalized */
u->uv.tt = LUA_TUSERDATA;   u->uv.len = s;
u->uv.metatable = NULL;   u->uv.env = e;
  /* chain it on udata list (after main thread) */
  u->uv.next = G(L)->mainthread->next;
G(L)->mainthread->next = obj2gco(u);
return u;
}
```

可以看到它是和上面的两种方式完全不同,这是因为 userdata 一般来说都有自己的 gc 方法，因此最好能够放在一起处理，因此这里会将 udata 放到最末尾. 在 lua 中的 gc 算法，是 mark-sweep 算法，这个算法简单来说就分为两步，第一步是遍历所有的 GCObject 的
对象，然后做标记 。 第二步是遍历所有可回收对象，然后清除没有做过标记的对象。 在 lua 中通过两个
参数来控制 gc 的频率和周期，分别是 garbage-collector pause 和 garbage-collector step multiplier， 这两个值都是使用百分比(100 表示 100%). 其中 garbage-collector pause 控制回收器等待多久开始一次新的垃圾回收，比如默认值是 200，那么就说明只有当等待内存使用为上一次 gc 时的 2 倍才会进行下一次 gc。而
garbage-collector step multiplier 控制垃圾回收的相对速度(相对于分配的速度), 默认也是 200，说明垃圾回收的速度为内存分配的两倍，这两个值都可以通过 lua_gc 来修改(LUA_GCSETPAUSE 与 LUA_GCSETSTEPMUL).

而在 lua 5.1 中实现的是 Tri-color marking 算法，算法描述见 wiki(http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29#Tri-color_marking)，这个算法将每一个对象分为三种颜色，分别是白色(初始状态), 灰色(和 root 有连接，可是它所连接的对象还没有被扫描,因此这个状态不能被 gc),以及黑色(可以被释放的对象集合), 所有的对象都会经历从白色到灰色再到黑色的过程.

算法的具体步骤如下:

1. Create initial white, grey, and black sets; these sets will be used to maintain progress during the cycle.
2.

* Initially the white set or condemned set is the set of objects that are candidates for having theirmemory recycled.

* The black set is the set of objects that can cheaply be proven to have no references to objects in thewhite set, but are also not chosen to be candidates for recycling; in many implementations, the black set starts off empty.
* The grey set is all the objects that are reachable from root references but the objects referenced bygrey objects haven’ t been scanned yet. Grey objects are known to be reachable from the root, so cannot be garbage collected: grey objects will eventually end up in the black set. The grey state means we still need to check any objects that the object references.
* The grey set is initialised to objects which are referenced directly at root level; typically all otherobjects are initially placed in the white set.
* Objects can move from white to grey to black, never in the other direction.

3. Pick an object from the grey set. Blacken this object (move it to the black set), by greying all thewhite objects it references directly. This confirms that this object cannot be garbage collected, and also that any objects it references cannot be garbage collected.
4. Repeat the previous step until the grey set is empty.
5. When there are no more objects in the grey set, then all the objects remaining in the white set havebeen demonstrated not to be reachable, and the storage occupied by them can be reclaimed.

然后我们来看具体实现，我们就从最常见的 table 来分析。首先来看 lua gc 的启动。这里核心方法就是 luaC_checkGC 这个宏, 因为 gc 的启动一般来说就是通过这个宏开始的。

```
#define luaC_checkGC(L) { \
  condhardstacktests(luaD_reallocstack(L, L->stacksize - EXTRA_STACK - 1)); \   if
(G(L)->totalbytes >= G(L)->GCthreshold) \      luaC_step(L); }
```

这里我们可以看到它会比较当前分配的字节数与 GCthreshold 进行比较，如果大于这个值才会进行 step。而 GCthreshold 就是一个阀值.. 然后来看 luaC_step 这个函数:

```
void luaC_step (lua_State *L)
{   global_State *g = G(L);
//首先计算需要回收的内存大小
  l_mem lim = (GCSTEPSIZE/100) * g->gcstepmul;
if (lim == 0)
    lim = (MAX_LUMEM-1)/2;  /* no limit */
g->gcdept += g->totalbytes - g->GCthreshold;
do {
//开始 gc 处理
    lim -= singlestep(L);
if (g->gcstate == GCSpause)
```

```
break;   } while (lim > 0);
if (g->gcstate != GCSpause)
{     if (g->gcdept <
GCSTEPSIZE)
      g->GCthreshold = g->totalbytes + GCSTEPSIZE;  /* - lim/g->gcstepmul;*/
else {
      g->gcdept -= GCSTEPSIZE;
g->GCthreshold = g->totalbytes;
    }
}   else
{
    setthreshold(g);
  }
}
```

这里主要就是 singlestep 函数:

```
static l_mem singlestep (lua_State *L)
{   global_State *g = G(L);
/*lua_checkmemory(L);*/   switch
(g->gcstate) {     case GCSpause: {
      markroot(L);  /* start a new collection */
return 0;     }
    case GCSpropagate: {       if (g->gray)
return propagatemark(g);       else {  /*
no more `gray' objects */
atomic(L);  /* finish mark phase */
return 0;
      }
}
    case GCSsweepstring:
{       lu_mem old =
g->totalbytes;
      sweepwholelist(L, &g->strt.hash[g->sweepstrgc++]);       if
(g->sweepstrgc >= g->strt.size)  /* nothing more to sweep? */
g->gcstate = GCSsweep;  /* end sweep-string phase */
lua_assert(old >= g->totalbytes);       g->estimate -= old -
g->totalbytes;       return GCSWEEPCOST;
    }
    case GCSsweep: {       lu_mem
old = g->totalbytes;
      g->sweepgc = sweeplist(L, g->sweepgc, GCSWEEPMAX);
if (*g->sweepgc == NULL) {  /* nothing more to sweep? */
checkSizes(L);
        g->gcstate = GCSfinalize;  /* end sweep phase */
      }
      lua_assert(old >= g->totalbytes);
g->estimate -= old - g->totalbytes;
return GCSWEEPMAX*GCSWEEPCOST;
    }
    case GCSfinalize:
{       if (g->tmudata)
{         GCTM(L);
        if (g->estimate > GCFINALIZECOST)
g->estimate -= GCFINALIZECOST;         return
GCFINALIZECOST;
      }
else {
        g->gcstate = GCSpause;  /* end collection */
g->gcdept = 0;         return 0;
```

```
      }
}
    default: lua_assert(0); return 0;
  }
}
```

可以看到这里是一个状态机. 这里 lua 的 gc 执行顺序也是按照上面的状态的从大到小开始。

其中 GCSpause 是初始化状态。在这个状态主要就是标记主线程对象(也就是从白色染成灰色)。我们就从这个状态开始,我们可以看到这个状态的处理很简单，就是调用 markroot 函数来标记对象：

```
static void markroot (lua_State *L)
{   global_State *g = G(L);   g->gray
= NULL;   g->grayagain = NULL;
g->weak = NULL;
  markobject(g, g->mainthread);
  /* make global table be traversed before main stack */
markvalue(g, gt(g->mainthread));   markvalue(g,
registry(L));   markmt(g);
  g->gcstate = GCSpropagate;
}
```

上面的 markXXX 的几个函数最终都会调用 reallymarkobject 函数，因此我们就从这个函数开始：

```
static void reallymarkobject (global_State *g, GCObject *o)
{   lua_assert(iswhite(o) && !isdead(g, o));   white2gray(o);
switch (o->gch.tt) {     case LUA_TSTRING:
{       return;     }
    case LUA_TUSERDATA: {
      Table *mt = gco2u(o)->metatable;
gray2black(o);  /* udata are never gray */
if (mt) markobject(g, mt);       markobject(g,
gco2u(o)->env);       return;     }
    case LUA_TUPVAL:
{       UpVal *uv =
gco2uv(o);
markvalue(g, uv->v);
      if (uv->v == &uv->u.value)  /* closed? */
gray2black(o);  /* open upvalues are never black */
return;     }
    case LUA_TFUNCTION:
{       gco2cl(o)->c.gclist =
g->gray;       g->gray = o;
break;
    }
    case LUA_TTABLE:
{       gco2h(o)->gclist =
g->gray;       g->gray = o;
break;     }
    case LUA_TTHREAD:
{       gco2th(o)->gclist =
g->gray;       g->gray = o;
break;     }
    case LUA_TPROTO:
{       gco2p(o)->gclist =
g->gray;       g->gray = o;
break;     }
```

```
    default: lua_assert(0);
  }
}
```

这个函数我们可以看到首先它会将白色染成灰色，然后会根据对象的类型来做不同的操作，这里特殊操作就 3 种类型，分别是 string(不通过 gc 管理),userdata, 以及 upval，其他的类型都是将灰色的对象连接到 global state 的链表中.

当 GCSpause 状态之后，会进入 GCSpropagate 状态(上面的 markroot 函数最后一个语句).这个状态也是一个标记过程，并且这个状态会被进入多次，也就是分布迭代。如果 gray 对象一直存在的话，会反复调用
propagatemark 函数，等所有的 gray 对象都被标记了，那么就将会进入 atomic 函数处理。这个函数，顾名思义，也就是原子操作，最终在这个状态之后，gc 进入清理字符串的阶段.

```
static l_mem propagatemark (global_State *g)
{   GCObject *o = g->gray;
lua_assert(isgray(o));   gray2black(o);
switch (o->gch.tt) {     case LUA_TTABLE:
{       Table *h = gco2h(o);       g->gray =
h->gclist;
      if (traversetable(g, h))  /* table is weak? */
black2gray(o);  /* keep it gray */
      return sizeof(Table) + sizeof(TValue) * h->sizearray +
sizeof(Node) * sizenode(h);
    }
    case LUA_TFUNCTION: {       Closure *cl = gco2cl(o);
g->gray = cl->c.gclist;       traverseclosure(g, cl);
return (cl->c.isC) ? sizeCclosure(cl->c.nupvalues) :
sizeLclosure(cl->l.nupvalues);
    }
    case LUA_TTHREAD:
{       lua_State *th =
gco2th(o);       g->gray =
th->gclist;       th->gclist =
g->grayagain;       g->grayagain
= o;       black2gray(o);
traversestack(g, th);
      return sizeof(lua_State) + sizeof(TValue) * th->stacksize +
sizeof(CallInfo) * th->size_ci;
    }
    case LUA_TPROTO:
{       Proto *p =
gco2p(o);       g->gray =
p->gclist;
traverseproto(g, p);
return sizeof(Proto) +
sizeof(Instruction) *
p->sizecode +
sizeof(Proto *) * p->sizep
+
sizeof(TValue) * p->sizek +
sizeof(int) *
p->sizelineinfo +
sizeof(LocVar) *
```

```
p->sizelocvars +
sizeof(TString *) *
p->sizeupvalues;
    }
    default: lua_assert(0); return 0;
  }
}
```

可以看到在 propagate 状态，也会根据对象类型来进行标记，这里我们可以看到它首先会把当前的对象节点标记为黑色，然后再进行后续处理,主要来看 table 类型。它会将对象挂载到 gray 链表，然后开始遍历标记 table，这里注意如果 table 是 weak 的，那么则会将 black 节点重新染成 gray 的, 最后返回这次标记的内存大小，而核心方法就在 traversetable.

```
static int traversetable (global_State *g, Table *h)
{   int i;   int weakkey = 0;   int weakvalue = 0;
const TValue *mode;   if (h->metatable)
    markobject(g, h->metatable);   mode =
gfasttm(g, h->metatable, TM_MODE);
  if (mode && ttisstring(mode)) {  /* is there a weak mode? */
weakkey = (strchr(svalue(mode), 'k') != NULL);     weakvalue =
(strchr(svalue(mode), 'v') != NULL);     if (weakkey || weakvalue)
{  /* is really weak? */       h->marked &= ~(KEYWEAK | VALUEWEAK);
/* clear bits */       h->marked |= cast_byte((weakkey << KEYWEAKBIT)
|                              (weakvalue << VALUEWEAKBIT));
h->gclist = g->weak;  /* must be cleared after GC, ... */
g->weak = obj2gco(h);  /* ... so put in the appropriate list */
    }
}
  if (weakkey && weakvalue) return 1;
if (!weakvalue) {     i =
h->sizearray;     while (i--)
      markvalue(g,
&h->array[i]);   }
```

//对表的数组部分进行处理.

```
  i = sizenode(h);
while (i--) {
    Node *n = gnode(h, i);
    lua_assert(ttype(gkey(n)) != LUA_TDEADKEY || ttisnil(gval(n)));
if (ttisnil(gval(n)))
      removeentry(n);  /* remove empty entries */
else {
      lua_assert(!ttisnil(gkey(n)));
if (!weakkey) markvalue(g, gkey(n));
if (!weakvalue) markvalue(g, gval(n));
    }
}
  return weakkey || weakvalue;
}
```

traversetable 方法首先会标记元表，然后主要是对 weak table 进行特殊处理,由于 weak table 是弱引用，因此这里将会在 gc 之后单独处理弱表(g->weak).如果不是 weak 表，那么将会对这个对象进行

mark。最后返回值是表示当前的表是否处于 weak 模式. 如果 traversetable 返回 1，则表示表是 weak 模式，此时重新将对象的颜色染回灰色，因为 weak table，后续会统一处理，也就是脱离 lua 的 gc.

最后如果已经将所有的 gray 对象染色完毕(weak 表的话，gray 对象会被移到 g->weak),那么 GCSpropagate 状态最后将会进入 atomic 这个函数。这个函数之所以叫 atomic，是因为在这个状态下 lua 的标记是不会被打
断的，它最终会做一次清理，也就是对于在标记期间有改变的对象再次进行 mark。这里就涉及到一个 barrier 的概念，之所以要有 barrier，是因为由于 lua 的 gc 是分步的，因此在进入最终的清理状态之前，有可能被标记的对象的颜色已经改变(比如本来是白色，可是我们第一次扫描之后，它又被使用了，此时自
然就变成灰色了，或者是已经被染色为黑色了，可是对象后续又没有对应的引用了),在这些情况下，都会将颜色染回灰色，要么是 barrier fwd(white->gray),要么是 barrier back(black->gray).后续我们会详细介绍 barrier，这里先跳过.

```
static void atomic (lua_State *L)
{   global_State *g = G(L);
  size_t udsize;  /* total size of userdata to be finalized */
/* remark occasional upvalues of (maybe) dead threads */
remarkupvals(g);
  /* traverse objects cautch by write barrier and by 'remarkupvals' */
propagateall(g);   /* remark weak tables */   g->gray = g->weak;
g->weak = NULL;
  lua_assert(!iswhite(obj2gco(g->mainthread)));
markobject(g, L);  /* mark running thread */
markmt(g);  /* mark basic metatables (again) */
propagateall(g);   /* remark gray again */
g->gray = g->grayagain;   g->grayagain = NULL;
propagateall(g);
  udsize = luaC_separateudata(L, 0);  /* separate userdata to be finalized */
marktmu(g);  /* mark `preserved' userdata */
  udsize += propagateall(g);  /* remark, to propagate `preserveness' */
cleartable(g->weak);  /* remove collected objects from weak tables */
  /* flip current white */
  g->currentwhite = cast_byte(otherwhite(g));
//这里注意，这个值后面清理 string 的时候会用到.
  g->sweepstrgc = 0;
//清理其他对象的时候，会用到.到达这里说明在 rootgc 上挂在的都是不可达对象，因此我们需要将他们后续
清理.   g->sweepgc = &g->rootgc;   g->gcstate = GCSsweepstring;
  g->estimate = g->totalbytes - udsize;  /* first estimate */
}
```

这里还有一个要注意的，那就是处理 useadata，由于 userdata 是会有自己的 gc 方法，因此 userdata 最终会单独处理(前面我们看到链接到 gcroot 的时候，也是放在最末尾).来看 luaC_separateudata:

```
size_t luaC_separateudata (lua_State *L, int all)
{   global_State *g = G(L);   size_t deadmem = 0;
```

```
//取出 userdata
  GCObject **p = &g->mainthread->next;
  GCObject *curr;
//开始遍历
  while ((curr = *p) != NULL) {
    if (!(iswhite(curr) || all) || isfinalized(gco2u(curr)))
p = &curr->gch.next;  /* don't bother with them */     else if
(fasttm(L, gco2u(curr)->metatable, TM_GC) == NULL)
{       markfinalized(gco2u(curr));  /* don't need finalization
*/       p = &curr->gch.next;
    }
    else {  /* must call its gc method */
//到达这里说明有 gc 方法
      deadmem += sizeudata(gco2u(curr));
markfinalized(gco2u(curr));
      *p = curr->gch.next;
      /* link `curr' at the end of `tmudata' list */
if (g->tmudata == NULL)  /* list is empty? */
        g->tmudata = curr->gch.next = curr;  /* creates a circular list */
else {
        curr->gch.next = g->tmudata->gch.next;
g->tmudata->gch.next = curr;
g->tmudata = curr;
      }
    }
}
  return deadmem;
}
```

通过上面我们可以看到这里并没有真正的释放 userdata，只是将有 gc 方法的 userdata 链接到 g->tmudata 上。我们要谨记，在 lua gc 中，只有清理阶段才会真正释放内存。然后我们来看 GCSsweepstring 状态，也就是清理 string。

```
    case GCSsweepstring:
{       lu_mem old =
g->totalbytes;
//这里可以看到每次进来都会对当前的 hash 进行释放，这里 sweepstrgc 相当于一个索引
      sweepwholelist(L, &g->strt.hash[g->sweepstrgc++]);
//判断是否释放完毕
      if (g->sweepstrgc >= g->strt.size)  /* nothing more to sweep? */
g->gcstate = GCSsweep;  /* end sweep-string phase */
lua_assert(old >= g->totalbytes);       g->estimate -= old -
g->totalbytes;       return GCSWEEPCOST;     }

#define sweepwholelist(L,p)     sweeplist(L,p,MAX_LUMEM)
```

可以看到核心的方法就是 sweeplist,这个方法是清理阶段所有的对象都会调用这个方法.这里注意有一个 other white，这个也就是当我们在标记之后，清理之前新添加的对象，我们都会认为他们是 other white，等待下次处理.

```
static GCObject **sweeplist (lua_State *L, GCObject **p, lu_mem count)
{   GCObject *curr;   global_State *g = G(L);   int deadmask =
otherwhite(g);
```

//遍历 gc 对象

```
  while ((curr = *p) != NULL && count-- > 0) {
    if (curr->gch.tt == LUA_TTHREAD)  /* sweep open upvalues of each thread */
sweepwholelist(L, &gco2th(curr)->openupval);
```

//如果还没有 dead，则会重新染成白色，等待下次处理

```
    if ((curr->gch.marked ^ WHITEBITS) & deadmask) {  /* not dead? */
lua_assert(!isdead(g, curr) || testbit(curr->gch.marked, FIXEDBIT));
makewhite(g, curr);  /* make it white (for next cycle) */       p =
&curr->gch.next;
    }
    else {  /* must erase `curr' */
      lua_assert(isdead(g, curr) || deadmask == bitmask(SFIXEDBIT));
      *p = curr->gch.next;
      if (curr == g->rootgc)  /* is the first element of the list? */
        g->rootgc = curr->gch.next;  /* adjust first */
```

//释放对象

```
      freeobj(L, curr);
    }   }
return p;
}
```

我们来看最后一个状态 GCSfinalize，这个状态最终会处理 useadata，主要是调用 GCTM 来调用 userdata 的 gc 方法.

```
static void GCTM (lua_State *L)
{   global_State *g = G(L);
  GCObject *o = g->tmudata->gch.next;  /* get first element */
  Udata *udata = rawgco2u(o);
const TValue *tm;
  /* remove udata from `tmudata' */   if (o
== g->tmudata)  /* last element? */
g->tmudata = NULL;   else
    g->tmudata->gch.next = udata->uv.next;
  udata->uv.next = g->mainthread->next;  /* return it to `root' list */
g->mainthread->next = o;   makewhite(g, o); //取的 gc 元方法.
  tm = fasttm(L, udata->uv.metatable, TM_GC);
if (tm != NULL) {
    lu_byte oldah = L->allowhook;
lu_mem oldt = g->GCthreshold;
    L->allowhook = 0;  /* stop debug hooks during GC tag method */
g->GCthreshold = 2*g->totalbytes;  /* avoid GC steps */
setobj2s(L, L->top, tm);     setuvalue(L, L->top+1, udata);
L->top += 2;
```

//调用对应的 gc 方法.

```
    luaD_call(L, L->top - 2, 0);
    L->allowhook = oldah;  /* restore hooks */
g->GCthreshold = oldt;  /* restore threshold */
  }
}
```

最后我们来看 write barrier，它主要用来解决在扫描过程中，一些已经染色的对象，或者说新添加的对象能够被正确的染色。来看对应的 4 个 api：

```
#define luaC_barrier(L,p,v) { if (valiswhite(v) && isblack(obj2gco(p)))  \
luaC_barrierf(L,obj2gco(p),gcvalue(v)); }

#define luaC_barriert(L,t,v) { if (valiswhite(v) && isblack(obj2gco(t)))  \
luaC_barrierback(L,t); }

#define luaC_objbarrier(L,p,o)  \
     { if (iswhite(obj2gco(o)) && isblack(obj2gco(p))) \
luaC_barrierf(L,obj2gco(p),obj2gco(o)); }

#define luaC_objbarriert(L,t,o)  \
   { if (iswhite(obj2gco(o)) && isblack(obj2gco(t))) luaC_barrierback(L,t); }
```

其中带 obj 的表示是针对 gc 类型的，而不带 obj 的 api 表示是针对 Tvalue 类型的。而对应的 luaC_barrierf(forwardl 表示 从白色到灰色，而 luaC_barrierback(back)表示从黑色到灰色. 并且可以看到 table 被拿出来特殊处理，这里之所以特殊处理，是因为 table 的修改是很频繁的，而其他的对象之间联系会比较少.因此前面的代码我们也可以看到(atomic)，也是针对 table 对象特殊处理. 这四个函数都是用于将对象 v(o)关联到 p(t)时，需要做的操作.先来看 luaC_barrierf

```
void luaC_barrierf (lua_State *L, GCObject *o, GCObject *v)
{   global_State *g = G(L);
  lua_assert(isblack(o) && iswhite(v) && !isdead(g, v) && !isdead(g, o));
lua_assert(g->gcstate != GCSfinalize && g->gcstate != GCSpause);
lua_assert(ttype(&o->gch) != LUA_TTABLE);
  /* must keep invariant? */   if
(g->gcstate == GCSpropagate)
    reallymarkobject(g, v);  /* restore invariant */
else  /* don't mind */
    makewhite(g, o);  /* mark as white just to avoid other barriers */
}
```

这个函数很简单，我们可以看到只有处于 mark 状态时，我们才需要重新 mark 将要挂在的对象，否则就直接把对象染成白色(和需要清理的白色不同)。

然后是 luaC_barrierback

```
void luaC_barrierback (lua_State *L, Table *t) {
  global_State *g = G(L);
GCObject *o = obj2gco(t);
  lua_assert(isblack(o) && !isdead(g, o));
  lua_assert(g->gcstate != GCSfinalize && g->gcstate != GCSpause);
black2gray(o);  /* make table gray (again) */   t->gclist =
g->grayagain;   g->grayagain = o;
}
```

可以看到是直接变为灰色，然后再把对象加载到 grayagain 这个链表上.而最终会在 atomic 函数中特殊处理.