lua源码剖析

作者: simohayha http://simohayha.javaeye.com

lua源码剖析

目 录

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free position。

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：

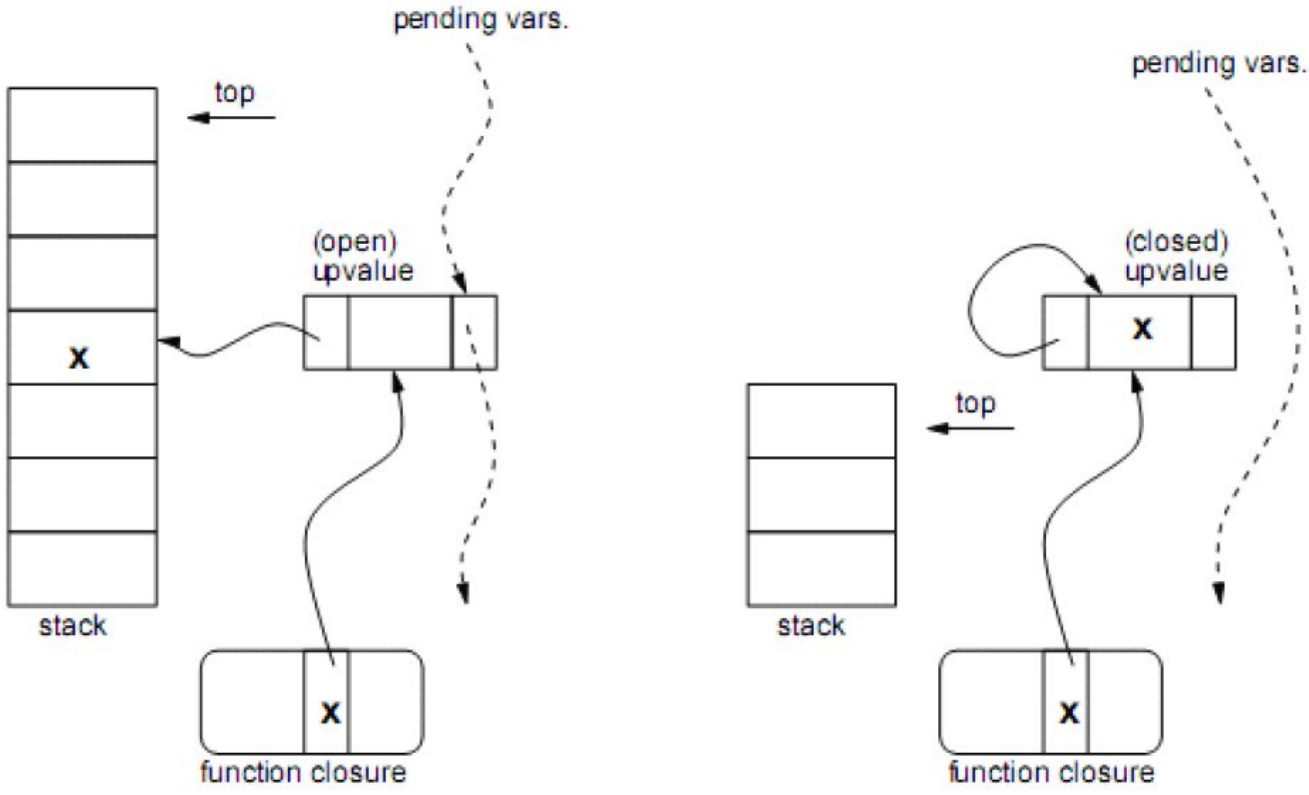

Figure 4: An upvalue before and after being "closed".

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
    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 */
}
```