Lua设计与实现

PB15030776 郭振江

一、摘要

此次project阅读了《Lua设计与实现》的第一至五章和七八两章。项目开始之前，我有下面这些疑问：

1. 虚拟机到底是长什么样子的？
2. 虚拟机的ISA和CPU的ISA有何不同？
3. 如何写一个DSL而不是Library？
4. 如何实现GC?
5. 如何让一门语言有热更新的能力？

带着这些问题，和学习语言实现上的细节的目的，我开始阅读这本书。

Lua语言的历史就不再赘述了，这里主要讲一下选择Lua而不是Java的一些原因：

1. 代码少、质量高，用clean C编写。
2. 打折同时还买了《Lua游戏设计》、热更新能力，
3. 有GC有虚拟机

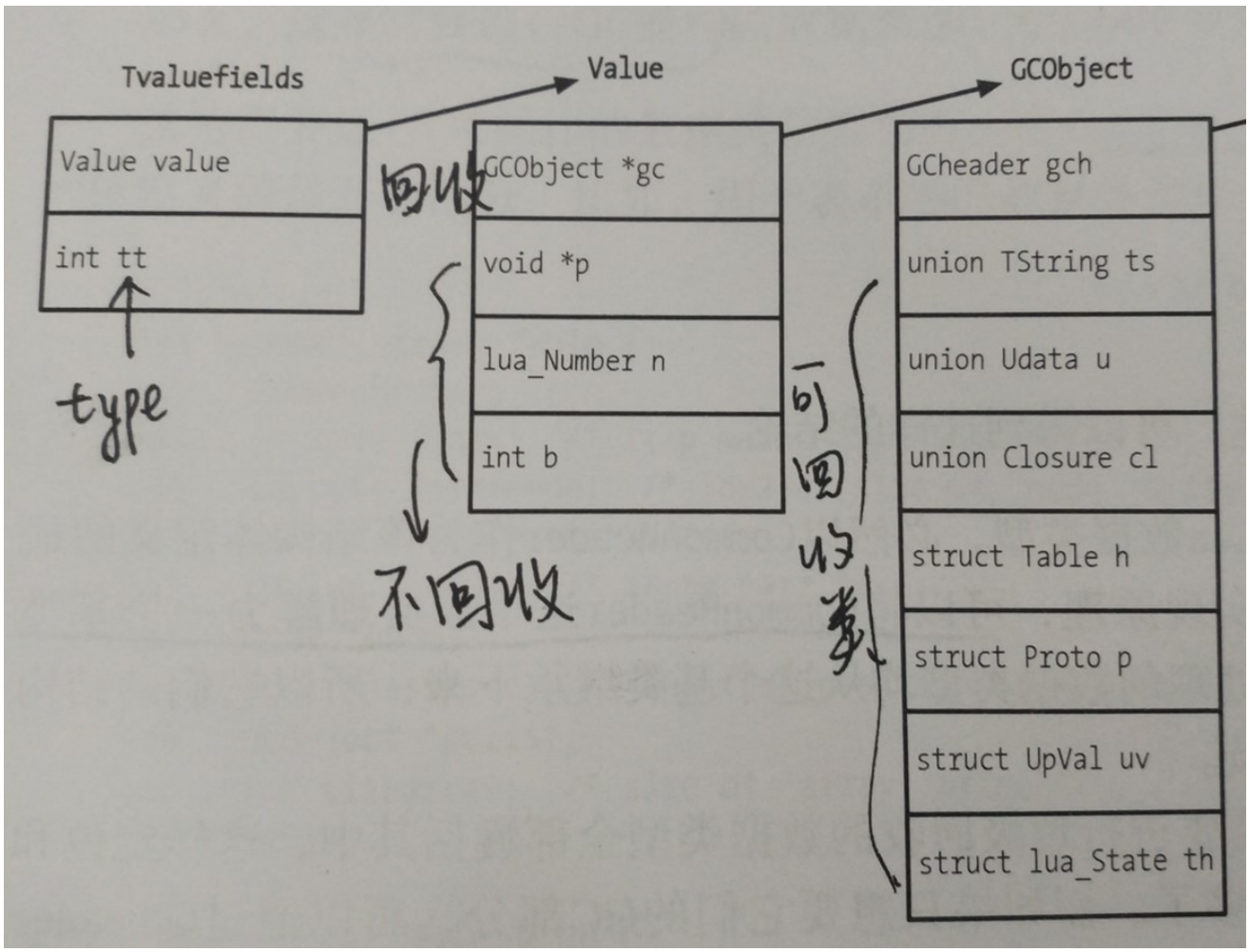
二、数据结构

1.1 数据类型

表示所有数据类型的数据结构包括数据的类型和数据的值。Lua中定义的基本数据类型如下图所示

```
#define LUA_TNONE        (-1)
#define LUA_TNIL         0
#define LUA_TBOOLEAN     1
#define LUA_TLIGHTUSERDATA  2
#define LUA_TNUMBER      3
#define LUA_TSTRING      4
#define LUA_TTABLE       5
#define LUA_TFUNCTION    6
#define LUA_TUSERDATA    7
#define LUA_TTHREAD      8
#define LUA_NUMTAGS      9
```

大于等于String类型的类型是需要GC的。



需要GC的数据都用一个公共的CommonHeader成员，相当于从一个同一个基类继承下来。比如Proto

```
typedef struct Proto {
  CommonHeader;          // <- GC Header
  lu_byte numparams;  /* number of fixed parameters */
  lu_byte is_vararg;
  lu_byte maxstacksize;  /* number of registers needed by this function */
  int sizeupvalues;  /* size of 'upvalues' */
  int sizek;  /* size of 'k' */
  int sizecode;
  int sizelineinfo;
```

```
  int sizep;  /* size of 'p' */
  int sizelocvars;
  int linedefined;
  int lastlinedefined;
  TValue *k;  /* constants used by the function */
  Instruction *code;  /* opcodes */
  struct Proto **p;  /* functions defined inside the function */
  int *lineinfo;  /* map from opcodes to source lines (debug information) */
  LocVar *locvars;  /* information about local variables (debug information) */
  Upvaldesc *upvalues;  /* upvalue information */
  struct LClosure *cache;  /* last-created closure with this prototype */
  TString  *source;  /* used for debug information */
  GCObject *gclist;
} Proto;
```

1.2 字符串类型

字符串类型最重要的两个方面就是长度和指向字符串数据的指针。

Lua中的字符串是常量，且只有一个副本。虚拟机中有一个全局散列桶专门存放所有的字符串常量，创建新的字符串时先查重，如果已经存在则不再创建。如果没有指针指向字符串，则会在GC截断被回收。这样的字符串实现由一下几个特点：

1. 比较和查找操作速度较快。
2. 空间占用少。
3. 可能会被rehash。
4. 频繁的使用字符串连接会显著影响性能。

例如下面这段代码

```
a = os.clock()
local s = ""
for i = 1,30000 do
    s = s .. 'a'
end
b = os.clock()
print(b-a)

-- better
a = os.clock()
local s = ""
local t = {}
for i = 1,30000 do
    t[#t + 1] = 'a'
end
s = table.concat(t, "")
b = os.clock()
print(b-a)
```

输出为：

```
0.041118
0.003197
```

有十几倍的性能差距。

1.3 表

表分为数组部分和散列表部分。例如下面这段代码：

```
local t = {}

t[1] = 0
t[100] = 0

-- print array
for k,v in ipairs(t) do
    print(k, v)
end

-- print all
for k,v in pairs(t) do
    print(k, v)
end
```

输出为：

```
1   0
1   0
100 0
```

这使得在表操作时需要兼顾这两者

1. 查找。比较键值是否小于数组部分长度。然后分别尝试查找。
2. 增加元素。需要判断是在数组部分还是在哈希桶部分。例如：

   ```
   t = {}
   t[1] = 1
   t[100] = 1
   ```

   则t[2]在数组部分，t[1]在哈希桶部分。

   如果没有足够的空间存新的数据，则需要rehash，rehash触发条件如下所示：

   ```
   Node *f = getfreepos(t);  /* get a free place */
   if (f == NULL) {  /* cannot find a free place? */
       rehash(L, t, key);  /* grow table */              // <- rehash
       /* whatever called 'newkey' takes care of TM cache and GC barrier */
       return luaH_set(L, t, key);  /* insert key into grown table */
   }
   ```

这时数据可能从散列桶中转移到数组中或反之。Rehash的开销挺大，应该尽力避免创建大量的短表。例如下面这段代码：

```
a = os.clock()
local s = ""
for i = 1,300000 do
    local a = {}
    a[1] = 1; a[2] = 2; a[3] = 3
end
b = os.clock()
print(b-a)

-- better
a = os.clock()
local s = ""
local t = {}
for i = 1,300000 do
    local a = {0, 0, 0}
    a[1] = 1; a[2] = 2; a[3] = 3
end
s = table.concat(t, "")
b = os.clock()
print(b-a)
```

运行结果为：

```
0.091696
0.041844
```

性能大约差了一倍。

3. rehash过程。首先计算nums数组，其第i个元素存放key在2^(i-1)和2^i之间的元素数量。然后遍历nums获得其范围区间所包含的整数数量大于50%的最大索引，作为重新散列之后的数组大小，其余的部分分配到散列桶中。例如对

```
t = {[1] = 1, [2] = 2, [3] = 3, [20] = 20}
```

有

```
         range  value  e.g.  ratio
nums[0]  (0,1]    1     1    100%
nums[1]  (1,2]    1     2    100%
nums[2]  (2,4]    1     3    50%    // <- array
nums[3]  (4,8]    0    none  0%     // <- hash
nums[4]  (8,16]   0    none  0%
nums[5]  (16,32]  1     20   1/16
```

4. 取长度操作。先在数组部分二分查找nil的位置，再在哈希桶部分二分查找nil的位置。这样可能会造成一些问题。例如：

```
print(#{10, 20, nil, 40}) -- 4

print(#{2, 10, 40, nil}) -- 3

print(#{10, 20, nil, 40, nil}) -- 2

print(#{[1]=1, [2]=2, 3, 4,5}) -- 3
```

因此建议尽量不要混用数组和哈希桶。

三、虚拟机

2.1 概览

Lua是已知的第一个使用基于寄存器虚拟机并被广泛使用的编程语言。实际执行的时候，则是把栈的某一位置称为寄存器。虚拟机主要模拟CPU和Memory。

虚拟机最外层的语句执行的是：

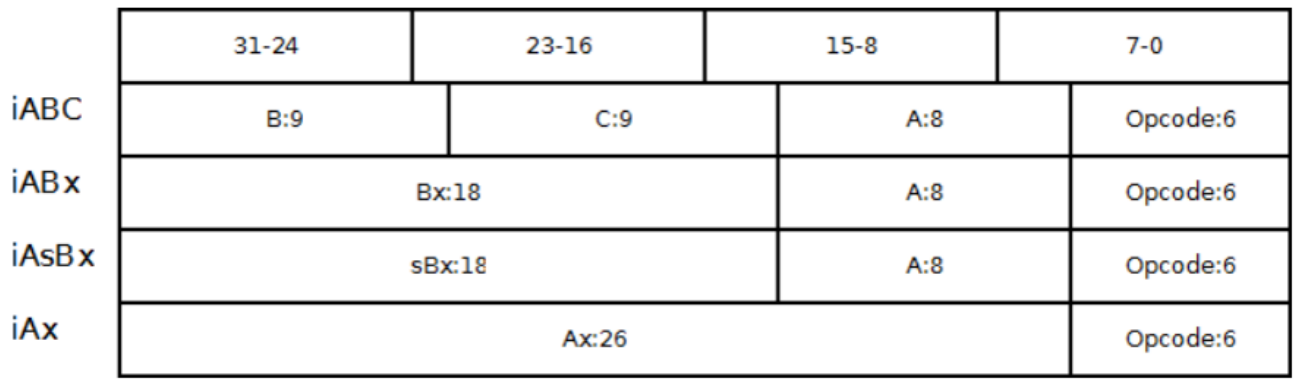
```
dofile(..) = loadfile(..) || pcall(..)
```

loadfile()将文件加载并进行词法语法分析，生成Proto结构传给虚拟部分执行。为了提高运行速度，这里采用的是梯度下降法，调用函数是f_parser()。Proto包含了源文件的全部信息和生成的opcode，虚拟机则是一个for(;;)循环，从中依次取得指令并执行。

每个Lua虚拟机对应一个lua_State结构体，它使用Tvalue数组来模拟栈。使用CallInfo结构体来表示每个被调函数的信息，base_ci是CallInfo数组，ci指向当前执行的函数。函数有一个prev指针指向调用者，Proto*数组指向内嵌函数，完成上下文切换、变量查询等。

2.2 OpCode

Lua虚拟机的指令是32位的，



6为的操作数对应63条指令，但实际上只用了38条。相关信息在lopcodes.h中有定义：

```
#define SIZE_C      9
#define SIZE_B      9
#define SIZE_Bx     (SIZE_C + SIZE_B)
#define SIZE_A      8
#define SIZE_Ax     (SIZE_C + SIZE_B + SIZE_A)
```

```
#define SIZE_OP      6

#define POS_OP       0
#define POS_A        (POS_OP + SIZE_OP)
#define POS_C        (POS_A + SIZE_A)
#define POS_B        (POS_C + SIZE_C)
#define POS_Bx       POS_C
#define POS_Ax       POS_A
```

有了具体的位置信息，就可以获取或设置具体的数值：

```
#define GET_OPCODE(i)   (cast(OpCode, ((i)>>POS_OP) & MASK1(SIZE_OP,0)))
#define SET_OPCODE(i,o) ((i) = (((i)&MASK0(SIZE_OP,POS_OP)) | \
        ((cast(Instruction, o)<<POS_OP)&MASK1(SIZE_OP,POS_OP))))

#define getarg(i,pos,size)  (cast(int, ((i)>>pos) & MASK1(size,0)))
#define setarg(i,v,pos,size)    ((i) = (((i)&MASK0(size,pos)) | \
                ((cast(Instruction, v)<<pos)&MASK1(size,pos))))

#define GETARG_A(i) getarg(i, POS_A, SIZE_A)
#define SETARG_A(i,v)   setarg(i, v, POS_A, SIZE_A)

#define GETARG_B(i) getarg(i, POS_B, SIZE_B)
#define SETARG_B(i,v)   setarg(i, v, POS_B, SIZE_B)

#define GETARG_C(i) getarg(i, POS_C, SIZE_C)
#define SETARG_C(i,v)   setarg(i, v, POS_C, SIZE_C)

#define GETARG_Bx(i)    getarg(i, POS_Bx, SIZE_Bx)
#define SETARG_Bx(i,v)  setarg(i, v, POS_Bx, SIZE_Bx)

#define GETARG_Ax(i)    getarg(i, POS_Ax, SIZE_Ax)
#define SETARG_Ax(i,v)  setarg(i, v, POS_Ax, SIZE_Ax)

#define GETARG_sBx(i)   (GETARG_Bx(i)-MAXARG_sBx)
#define SETARG_sBx(i,b) SETARG_Bx((i),cast(unsigned int, (b)+MAXARG_sBx))
```

这里主要提一些我出乎我预料的指令

```
typedef enum {
/*----------------------------------------------------------------------
name        args    description
------------------------------------------------------------------------*/
OP_NEWTABLE,/*  A B C   R(A) := {} (size = B,C)             */

OP_LEN,/*   A B R(A) := length of R(B)              */

OP_TAILCALL,/*  A B C   return R(A)(R(A+1), ... ,R(A+B-1))      */

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

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

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

具体指令的执行部分还没有仔细看。

四、其他内容

3.1 GC

Lua中将可以被GC的数据类型定义为大于字符串的类型：

```
#define iscollectable(o)    (ttype(o) >= LUA_TSTRING)
```

在5.3中将其改为位运算加快速度。

具体而言需要GC的数据类型有

```
string
table
function
thread
proto
```

Lua在5.0时采用引用计数法作为GC算法，后来在5.1中改为了标记清除法。标记清除法的一个细节就是一共需要几种颜色来标记数据？

1. 双色法：白色、黑色 -> 不能被打断（下一轮扫描？）
2. 三色法：白色、灰色、黑色 -> 灰色是被扫描过但引用没被扫描过
3. 四色法：双白色 ->在回收阶段新注册的变量。

考虑到GC算法需要分布进行，Lua采用的是四色法。

但是这种方法也不是没有缺点，比如会造成内存碎片，需要清理。

自动GC的触发条件是totalbytes >= Gcthreshold，如果希望关闭自动GC或者想使用自己的GC函数，那么就需要将Gcthreshold设置成一个较大的数值。另外要注意的就是在手动执行GC之后需要再次关闭自动GC。

3.2 环境

Lua中有不同层次的环境，比如使用Global表存放全局变量，registry表是全局唯一的，env保存函数自己的环境，会被逐层继承，UpValue存储函数的静态变量。

这样在环境中查找就需要比较复杂，这一部分在函数idx2addr()中：

```
if (idx > 0): 相对栈顶向上的偏移量
if (idx > LUA_REGISTRYINDEX): 相对栈顶向下的偏移量
case LUA_REGISTRYINDEX: registry表
case LUA_ENVIRONINDEX: env表
case LUA_GLOBALSINDEX: global表

default: upvalue数组
```

3.3 模块与热更新

Lua中有一些默认加载的模块，比如：

```
static const luaL_Reg lualibs[] = {
  {"", luaopen_base},                          // 无需模块名即可访问，比如print
  {LUA_LOADLIBNAME, luaopen_package},
  {LUA_TABLIBNAME, luaopen_table},
  {LUA_IOLIBNAME, luaopen_io},
  {LUA_OSLIBNAME, luaopen_os},
  {LUA_STRLIBNAME, luaopen_string},
  {LUA_MATHLIBNAME, luaopen_math},           // math.sin()
  {LUA_DBLIBNAME, luaopen_debug},
  {LUA_STRUCTLIBNAME , luaopen_struct},
  {NULL, NULL}
};
```

新的模块加载需要进行模块名和函数的注册，需要同时维护两个表：

```
require(“XXX”)  -- lua5.3中不再使用module

G[“XXX”]= registry[“LOADED”][“XXX”]  = {}

G[“os”][“print”] = 函数指针
```

如果想实现热更新，那么就需要先让Lua虚拟机认为没有加载这个模块，并重新加载。可以考虑下面的代码：

```
package.loaded[_name]= nil
require(_name)
```

另外需要注意的是，如果只是函数更新，那么显然全局变量不应该因为被热更新掉，需要加一些保护。比如

```
a =a or 0
```

五、总结

这是我第一次去仔细去看一门语言的实现，学到了很多比较底层的细节。而且Lua高质量的代码也让我获益匪浅，包括C的一些用法和易读性的规范。对我来说，从虚拟机中学到最多的大概是它的指令集了。在此之前我贫瘠的想象力还是停留在MIPSCPU的七条指令上，这个对我以后的发展应该也很有帮助。再有就是看到了GC算法在工业产品上的应用，不再是停留在理论层面上。Lua中基本数据的表示也让我有了很多思考，从关键的问题抓起，层层抽象，同时兼顾不同的需求。

很棒的一次调研，除了时间比较接近考试周。