Lua 源码分析 -- 对象表示

Lua 是动态类型的语言，即是说类型附着于值而不变量[1]. Lua 的八种基本类型空，布尔，数值，字符串，表，函数和用户数据. 所有类似的值都是虚拟机的第一类值. Lua 解释器将其表示成为标签联合(*tagged union*). 如下面代码示例所示:

```
lobject.h : 56
/*
** Union of all Lua values
*/
typedef union {
    GCObject *gc;
    void *p;
    lua_Number n;
    int b;
} Value;

/*
** Tagged Values
*/

#define TValuefields Value value; int tt

typedef struct lua_TValue {
    TValuefields;
} TValue;

lstate.h : 132
/*
** Union of all collectable objects
*/
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 */
```

```
};

lobject.h : 39
/*
** Common Header for all collectable objects (in macro form, to be
** included in other objects)
*/
#define CommonHeader GCObject *next; lu_byte tt; lu_byte marked


/*
** Common header in struct form
*/
typedef struct GCheader {
    CommonHeader;
} GCheader;
```

首先看到的一个 TValue 结构,它是由一个 Value 类型的字段 value 和 int 类型字段 tt 组成,它由于一个宏定义出来.很显然,这里的 tt 就是用于表示这个值的类型,这也是之前所说的,Lua 的类型是附着于值上的原因.

接 下来,再打量打量 Value 的定义,它被定义为 union.这样做的目的是让这一个类型可以表示多个类型.从这个定义中可以看出这样一点:Lua 的值可以 分成两类,第一类是可以被垃圾回收机制回收的对象,它们统一使用 GCObject 的指针来表示;另一类是原始类型,直接使用 C 语言的类型来表示相应类型, 如:用 void *来表示 lightuesrdata, 用 lua_Number 来表示数值,用 int 来表示 boolean.这里需要注意的是 lua_Number 是在如下两个文件定义出来的.由于 Lua 是易于 嵌入的语言,在某些特定的环境下,所有数值都用双精度浮点来表示并不合适,因此,在 Lua 的配置文件上使用宏来定义数值类型.这使得要改变 Lua 的数值类 型变得非常简单.

```
lua.h:98
/* type of numbers in Lua */
typedef LUA_NUMBER lua_Number;
```

```
luaconf.h:504
#define LUA_NUMBER double
```

接 下来继续看 GCObject 的定义,这个类型中的字段在这里并不做详细展开,只是说明是用于表示什么类型的.TString,UData,Table, lua_State 分别用于表示字符串,用户数据,表和协程.而Closure,Proto,UpVal 都是用于表示第一类的函数的. 基于栈的,词法定界的第一类函数在实现上是有一些难度的,看看如下代码:

```
function foo()
    local a
    return function() return a end
end
```

由 于Lua是词法定界的,局部变量a只在函数 foo 中有效,所以它可以保存在 foo 的栈中,因此当 foo 执行完毕 a 而就随着栈的销毁而成为垃圾; 但问题是 foo 返回的函数还在引用着它, 这个函数会在栈销毁后继续存在,当它返回 a 的时候又拿什么返回呢? 这个问题将在函数的实现中介绍. 这也是为什么实现函数用了三个类型的原因.

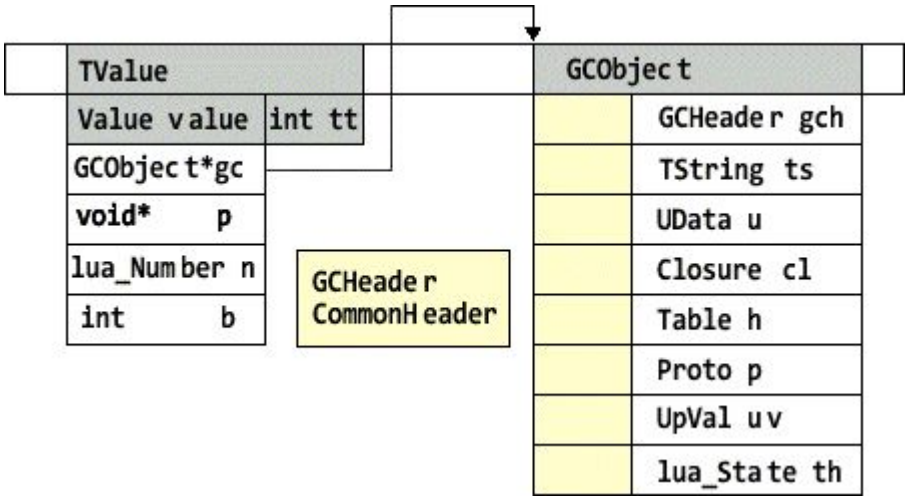
另外, 这些类型的开头都是 GCHeader, 它的所有字段由宏 CommonHeader 给出来了. 字段 next 说明可回收对象是可以放到链表中去的, 而 marked 是在 GC 中用于标记的. 具体的 GC 算法在这一章就不做介绍了.

值得注意是在 CommonHeader 中还有一个 tt 用于表示值的类型, 在 TValue 中不是有一个吗? 这样数据不是冗余了? 我是这样看这个问题的:

第一: TValue 是所有值的集合, 而 GC 中如果每个对象都要判断是否是可回收的, 必然会非常影响效率, 因此将 GCObject 独立出来. 可以省去这一层判断.

第二：对于基本类型来说，所需要的空间相对较小，如果将复杂的对象也做为一union放在一起，就会使得空间效率低，因此在 TValue 中只使用了一个指针来表示 GCObject. 这样在 GC 对看到的对象就不再是 TValue 了，所以对应的类型标识也不在了，所以在 CommonHeader 中加了一个字段来表示类型.

最后，给出一副图来表于 Lua 的内存表示：




Lua 源码分析 -- 虚拟机

Lua 首先将源程序编译成为字节码，然后交由虚拟机解释执行. 对于每一个函数，Lua 的编译器将创建一个原型(prototype)，它由一组指令及其使用到的常量组成[1]. 最初的 Lua 虚拟机是基于栈的. 到 1993 年，Lua5.0 版本，采用了基于寄存器的虚拟机，使得 Lua 的解释效率得到提升，

**体系结构与指令系统**

与虚拟机和指令相关的文件主要有两个： lopcodes.c 和 lvm.c. 从名称可以看出来，这两个文件分别用于描述操作码(指令)和虚拟机.

首先来看指令：

Lua 共有 38 条指令，在下面两处地方分别描述了这些指令的名称和模式，如下：

lopcodes.c:16

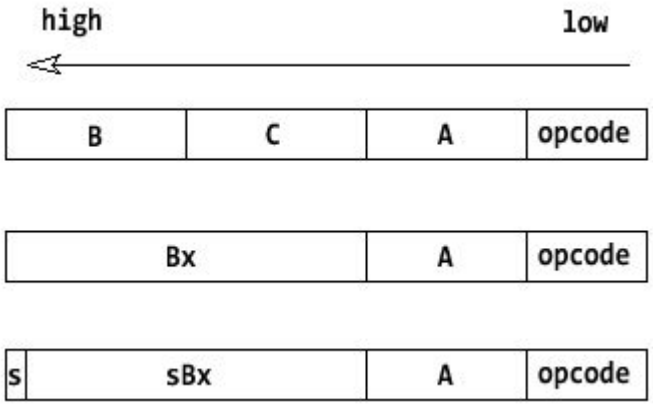
```
const char *const luaP_opnames[NUM_OPCODES+1] = {
  "MOVE",
  "LOADK",
  "LOADBOOL",
  "LOADNIL",
  "GETUPVAL",
  "GETGLOBAL",
  "GETTABLE",
  "SETGLOBAL",
  "SETUPVAL",
  "SETTABLE",
  "NEWTABLE",
  "SELF",
  "ADD",
  "SUB",
  "MUL",
  "DIV",
  "MOD",
  "POW",
  "UNM",
  "NOT",
  "LEN",
  "CONCAT",
  "JMP",
  "EQ",
  "LT",
  "LE",
  "TEST",
  "TESTSET",
  "CALL",
  "TAILCALL",
  "RETURN",
  "FORLOOP",
  "FORPREP",
  "TFORLOOP",
  "SETLIST",
  "CLOSE",
  "CLOSURE",
  "VARARG",
  NULL
};
```

```
#define opmode(t,a,b,c,m) (((t)<<7) | ((a)<<6) | ((b)<<4) | ((c)<<2)
| (m))

const lu_byte luaP_opmodes[NUM_OPCODES] = {
/*       T  A    B       C     mode           opcode    */
  opmode(0, 1, OpArgR, OpArgN, iABC)         /* OP_MOVE */
 ,opmode(0, 1, OpArgK, OpArgN, iABx)        /* OP_LOADK */
 ,opmode(0, 1, OpArgU, OpArgU, iABC)        /* OP_LOADBOOL */
 ,opmode(0, 1, OpArgR, OpArgN, iABC)        /* OP_LOADNIL */
 ,opmode(0, 1, OpArgU, OpArgN, iABC)        /* OP_GETUPVAL */
 ,opmode(0, 1, OpArgK, OpArgN, iABx)        /* OP_GETGLOBAL */
 ,opmode(0, 1, OpArgR, OpArgK, iABC)        /* OP_GETTABLE */
 ,opmode(0, 0, OpArgK, OpArgN, iABx)        /* OP_SETGLOBAL */
 ,opmode(0, 0, OpArgU, OpArgN, iABC)        /* OP_SETUPVAL */
 ,opmode(0, 0, OpArgK, OpArgK, iABC)        /* OP_SETTABLE */
 ,opmode(0, 1, OpArgU, OpArgU, iABC)        /* OP_NEWTABLE */
 ,opmode(0, 1, OpArgR, OpArgK, iABC)        /* OP_SELF */
 ,opmode(0, 1, OpArgK, OpArgK, iABC)        /* OP_ADD */
 ,opmode(0, 1, OpArgK, OpArgK, iABC)        /* OP_SUB */
 ,opmode(0, 1, OpArgK, OpArgK, iABC)        /* OP_MUL */
 ,opmode(0, 1, OpArgK, OpArgK, iABC)        /* OP_DIV */
 ,opmode(0, 1, OpArgK, OpArgK, iABC)        /* OP_MOD */
 ,opmode(0, 1, OpArgK, OpArgK, iABC)        /* OP_POW */
 ,opmode(0, 1, OpArgR, OpArgN, iABC)        /* OP_UNM */
 ,opmode(0, 1, OpArgR, OpArgN, iABC)        /* OP_NOT */
 ,opmode(0, 1, OpArgR, OpArgN, iABC)        /* OP_LEN */
 ,opmode(0, 1, OpArgR, OpArgR, iABC)        /* OP_CONCAT */
 ,opmode(0, 0, OpArgR, OpArgN, iAsBx)        /* OP_JMP */
 ,opmode(1, 0, OpArgK, OpArgK, iABC)        /* OP_EQ */
 ,opmode(1, 0, OpArgK, OpArgK, iABC)        /* OP_LT */
 ,opmode(1, 0, OpArgK, OpArgK, iABC)        /* OP_LE */
 ,opmode(1, 1, OpArgR, OpArgU, iABC)        /* OP_TEST */
 ,opmode(1, 1, OpArgR, OpArgU, iABC)        /* OP_TESTSET */
 ,opmode(0, 1, OpArgU, OpArgU, iABC)        /* OP_CALL */
 ,opmode(0, 1, OpArgU, OpArgU, iABC)        /* OP_TAILCALL */
 ,opmode(0, 0, OpArgU, OpArgN, iABC)        /* OP_RETURN */
 ,opmode(0, 1, OpArgR, OpArgN, iAsBx)        /* OP_FORLOOP */
 ,opmode(0, 1, OpArgR, OpArgN, iAsBx)        /* OP_FORPREP */
 ,opmode(1, 0, OpArgN, OpArgU, iABC)        /* OP_TFORLOOP */
 ,opmode(0, 0, OpArgU, OpArgU, iABC)        /* OP_SETLIST */
 ,opmode(0, 0, OpArgN, OpArgN, iABC)        /* OP_CLOSE */
 ,opmode(0, 1, OpArgU, OpArgN, iABx)        /* OP_CLOSURE */
 ,opmode(0, 1, OpArgU, OpArgN, iABC)        /* OP_VARARG */
```

};

前面一个数组容易理解，表示了每条指令的名称．后面一个数组表示的是指令的模式．奇怪的符号让人有些费解．在看模式之前，首先来看 Lua 指令的格式：




如上图，Lua 的指令可以分成三种形式．即在上面的模式数组中也可以看到的 iABC，iABx 和 iAsBx．对于三种形式的指令来说，前两部分都是一样的，分别是 6 位的操作码和 8 位 A 操作数；区别在于，后面部是分割成为两个长度为 9 位的操作符(B，C)，一个无符号的 18 位操作符 Bx 还是有符号的 18 位操作符 sBx．这些定义的代码如下：

```
lopcodes.c : 34
/*
** size and position of opcode arguments.
*/
#define SIZE_C        9
#define SIZE_B        9
#define SIZE_Bx       (SIZE_C + SIZE_B)
#define SIZE_A        8

#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
```

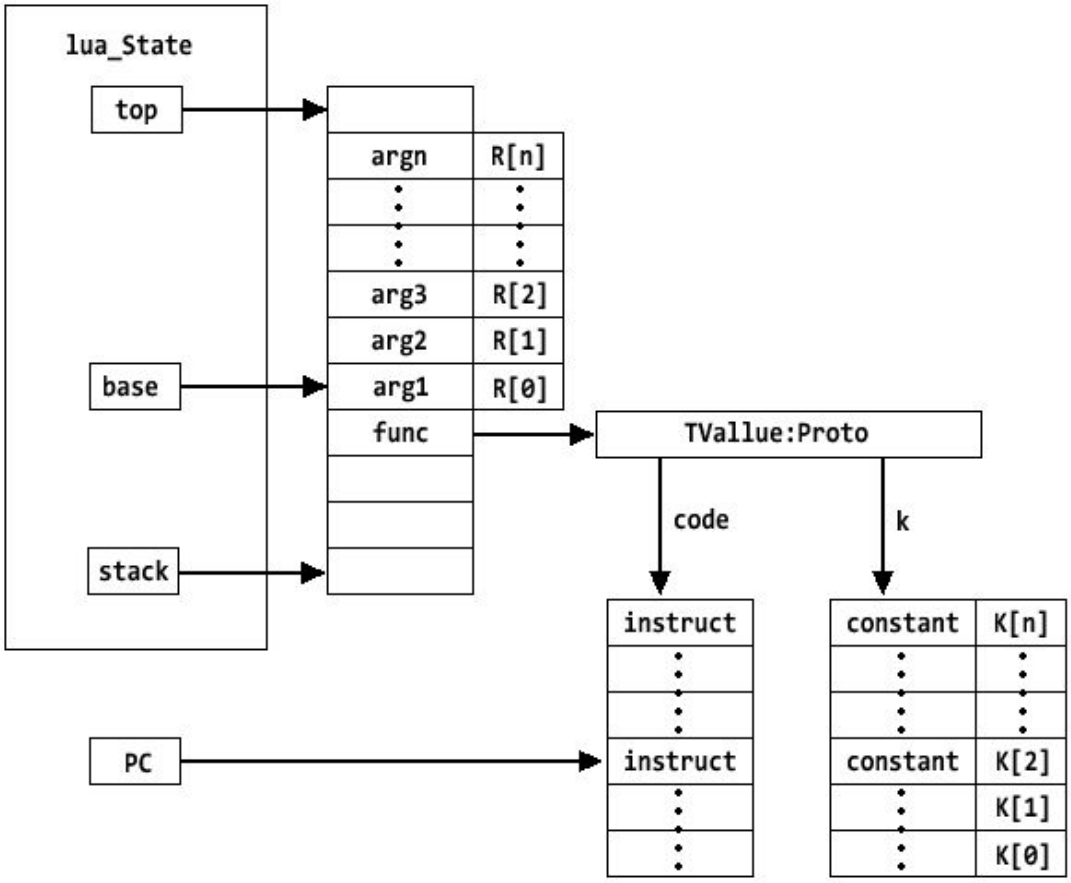
再来看指令的操作模式，Lua 使用一个字节来表示指令的操作模式．具体的含义如下：

1.使用最高位来表示是否是一条测试指令．之所以将这一类型的指令特别地标识出来，是因为 Lua 的指令长度是 32 位,对于分支指令来说，要想在这 32 位中既表示两个操作数来做比较，同时还要表示一个跳转的地址，是很困难的．因此将这种指令分成两条，第一条是测试指令，紧接着一条无条件跳转．如果判断条件成立则将 PC(Program Counter，指示下一条要执行的指令)加一，跳过下一条无条件跳转指令，继续执行；否则跳转．

2．第二位用于表示 A 操作数是否被设置

3．接下来的二位用于表示操作数 B 的格式,OpArgN 表示操作数未被使用，OpArgU 表示操作数被使用(立即数?)，OpArgR 表示表示操作数是寄存器或者跳转的偏移量，OpArgK 表示操作数是寄存器或者常量.

最后，给出 Lua 虚拟机的体系结构图(根据源代码分析得出)：



首先，我们注意到，Lua 的解释器还是一个以栈为中心的结构．在 lua_State 这个结构中，有许多个字段用于描述这个结构．stack 用于指向绝对栈底，而 base 指向了当前正在执行的函数的第一个参数，而 top 指向栈顶的第一个空元素．

我们可以看到，这个体系结构中并没有独立出来的寄存器．从以下代码来看：

```
lvm.c:343
#define RA(i)    (base+GETARG_A(i))
/* to be used after possible stack reallocation */
#define RB(i)       check_exp(getBMode(GET_OPCODE(i)) == OpArgR,
base+GETARG_B(i))
#define RC(i)       check_exp(getCMode(GET_OPCODE(i)) == OpArgR,
base+GETARG_C(i))
#define RKB(i)    check_exp(getBMode(GET_OPCODE(i)) == OpArgK, /
    ISK(GETARG_B(i)) ? k+INDEXK(GETARG_B(i)) : base+GETARG_B(i))
#define RKC(i)    check_exp(getCMode(GET_OPCODE(i)) == OpArgK, /
    ISK(GETARG_C(i)) ? k+INDEXK(GETARG_C(i)) : base+GETARG_C(i))
#define KBx(i)      check_exp(getBMode(GET_OPCODE(i)) == OpArgK,
k+GETARG_Bx(i))
```

当指令操作数的类型是寄存器时，它的内容是以 base 为基址在栈上的索引值．如图所示．寄存器实际是 base 之上栈元素的别名；当指令操作数的类型的常数时，它首先判断 B 操作数的最位是否为零．如果是零，则按照和寄存器的处理方法一样做，如果不是零，则在常数表中找相应的值．

我们知道 Lua 中函数的执行过程是这样的．首先将函数压栈，然后依次将参数压栈，形成图中所示的栈的内容．因此 R[0]到 R[n]也分别表示了 Arg[1]到 Arg[N+1]．在第一个参数之下，就是当前正在执行的函数，对于 Lua 的函数（相对 C 函数）来说，它是指向类型为 Prototype 的 TValue，在 Prototype 中字段 code 指向了一个数组用来表示组成这个函数的所有指令，字段 k 指向一个数组来表示这个函数使用到的所有常量．最后，Lua 在解释执行过程中有专门的变量 pc 来指向下一条要执行的指令．

**指令解释器**

有了前面对指令格式和体系结构的介绍,现在我们可以进入正题, 来看看 Lua 的指令是如何执行的了.主函数如下:

```
lvm.c:373
void luaV_execute (lua_State *L, int nexeccalls) {
  LClosure *cl;
  StkId base;
  TValue *k;
  const Instruction *pc;
 reentry:  /* entry point */
  lua_assert(isLua(L->ci));
  pc = L->savedpc;
  cl = &clvalue(L->ci->func)->l;
  base = L->base;
  k = cl->p->k;
```

这是最开始的初始化过程.其中, pc 被初始化成为了 L->savedpc,base 被初始化成为了 L->base, 即程序从 L->savedpc 开始执行 (在下一篇专题中,将会介绍到 L->savedpc 在函数调用的预处理过程中指向了当前函数的 code),而 L->base 指向栈中当前函数的下一个位置.cl 表示当前正在执行闭包(当前可以理解成为函数),k 指向当前闭包的常量表.

接下来(注意,为了专注主要逻辑, 我将其中用于 Debugger 支持,断言等代码省略了):

```
  /* main loop of interpreter */
  for (;;) {
    const Instruction i = *pc++;
    StkId ra;
    /* 省略 Debugger 支持和 Coroutine 支持*/
    /* warning!! several calls may realloc the stack and invalidate
`ra' */
    ra = RA(i);
    /* 省略断言 */
    switch (GET_OPCODE(i)) {
```

进 入到解释器的主循环,处理很简单,取得当前指令,pc 递增,初始化 ra,然后根据指令的

操作码进行选择. 接下来的代码是什么样的, 估计大家都能想到,一大串的case来指示每条指令的执行.具体的实现可以参考源码, 在这里不对每一条指令展开, 只是对其中有主要的几类指令进行说明:

传值类的指令,与MOVE为代表:

```
lvm.c:403
      case OP_MOVE: {
        setobjs2s(L, ra, RB(i));
        continue;
      }
lopcodes:154
OP_MOVE,/*    A B    R(A) := R(B)                    */
lobject.h:161
#define setobj(L,obj1,obj2) /
  { const TValue *o2=(obj2); TValue *o1=(obj1); /
    o1->value = o2->value; o1->tt=o2->tt; /
    checkliveness(G(L),o1); }


/*
** different types of sets, according to destination
*/

/* from stack to (same) stack */
#define setobjs2s    setobj
```

从注释来看, 这条指令是将操作数A,B都做为寄存器,然后将B的值给A. 而实现也是简单明了,只使用了一句. 宏展开以后, 可以看到, R[A],R[B]的类型是TValue, 只是将这两域的值传过来即可. 对于可回收对象来说,真实值不会保存在栈上,所以只是改了指针,而对于非可回收对象来说,则是直接将值从R[B]赋到R[A].

数值运算类指令,与ADD为代表:

```
lvm.c:470
      case OP_ADD: {
        arith_op(luai_numadd, TM_ADD);
        continue;
```

```
      }
lvm.c:360
#define arith_op(op,tm) { /
        TValue *rb = RKB(i); /
        TValue *rc = RKC(i); /
        if (ttisnumber(rb) && ttisnumber(rc)) { /
          lua_Number nb = nvalue(rb), nc = nvalue(rc); /
          setnvalue(ra, op(nb, nc)); /
        } /
        else /
          Protect(Arith(L, ra, rb, rc, tm)); /
      }
lopcodes.c:171
OP_ADD,/*    A B C    R(A) := RK(B) + RK(C)                */
```

如果两个操作数都是数值的话,关键的一行是:

```
setnvalue(ra,op(nb,nc));
```

即两个操作数相加以后,把值赋给 R[A].值得注意的是,操作数 B,C 都是 RK, 即可能是寄存器也可能是常量,这最决于最 B 和 C 的最高位是否为 1,如果是 1,则是常量,反之则是寄存器.具体可以参考宏 ISK 的实现.

如果两个操作数不是数值,即调用了 Arith 函数,它尝试将两个操作转换成数值进行计算,如果无法转换,则使用元表机制.该函数的实现如下:

```
lvm.c:313
static void Arith (lua_State *L, StkId ra, const TValue *rb,
                   const TValue *rc, TMS op) {
  TValue tempb, tempc;
  const TValue *b, *c;
  if ((b = luaV_tonumber(rb, &tempb)) != NULL &&
      (c = luaV_tonumber(rc, &tempc)) != NULL) {
    lua_Number nb = nvalue(b), nc = nvalue(c);
    switch (op) {
      case TM_ADD: setnvalue(ra, luai_numadd(nb, nc)); break;
      case TM_SUB: setnvalue(ra, luai_numsub(nb, nc)); break;
      case TM_MUL: setnvalue(ra, luai_nummul(nb, nc)); break;
      case TM_DIV: setnvalue(ra, luai_numdiv(nb, nc)); break;
      case TM_MOD: setnvalue(ra, luai_nummod(nb, nc)); break;
      case TM_POW: setnvalue(ra, luai_numpow(nb, nc)); break;
      case TM_UNM: setnvalue(ra, luai_numunm(nb)); break;
      default: lua_assert(0); break;
```

```
    }
  }
  else if (!call_binTM(L, rb, rc, ra, op))
    luaG_aritherror(L, rb, rc);
}
```

在上面 call_binTM 用于调用到元表中的元方法,因为在 Lua 以前的版本中元方法也被叫做 tag method, 所以函数最后是以 TM 结尾的.

```
lvm:163
static int call_binTM (lua_State *L, const TValue *p1, const TValue
*p2,
                       StkId res, TMS event) {
  const TValue *tm = luaT_gettmbyobj(L, p1, event);  /* try first
operand */
  if (ttisnil(tm))
    tm = luaT_gettmbyobj(L, p2, event);  /* try second operand */
  if (!ttisfunction(tm)) return 0;
  callTMres(L, res, tm, p1, p2);
  return 1;
}
```

在 这个函数中,试着从二个操作数中找到其中一个操作数的元方法(第一个操作数优先),这里 event 表示具体哪一个元方法,找到了之后,再使用函数 callTMres()去调用相应的元方法. callTMres()的实现很简单,只是将元方法,第一,第二操作数先后压栈,再调用并取因返回值.具体如下:

```
lvm.c:82
static void callTMres (lua_State *L, StkId res, const TValue *f,
                       const TValue *p1, const TValue *p2) {
  ptrdiff_t result = savestack(L, res);
  setobj2s(L, L->top, f);  /* push function */
  setobj2s(L, L->top+1, p1);  /* 1st argument */
  setobj2s(L, L->top+2, p2);  /* 2nd argument */
  luaD_checkstack(L, 3);
  L->top += 3;
  luaD_call(L, L->top - 3, 1);
  res = restorestack(L, result);
  L->top--;
  setobjs2s(L, res, L->top);
}
```

逻辑运算类指令,与 EQ 为代表:

```
lvm.c:541
      case OP_EQ: {
        TValue *rb = RKB(i);
        TValue *rc = RKC(i);
        Protect(
          if (equalobj(L, rb, rc) == GETARG_A(i))
            dojump(L, pc, GETARG_sBx(*pc));
        )
        pc++;
        continue;
      }
lopcodes.c:185
OP_EQ,/*    A B C    if ((RK(B) == RK(C)) ~= A) then pc++        */
```

在 这条指令实现的过程中,equalobj 与之前的算术运算类似,读者可以自行分析.关键看

它是如果实现中跳转的,如果 RK[B]==RK[C]并且 A 为 1 的情况下(即条件为真),则会使

用 pc 取出下一条指令,调用 dojump 进行跳转,否则 pc++,挂空紧接着的无条件跳转指令.

dojump 的实现如下:

```
lvm.c:354
#define dojump(L,pc,i)    {(pc) += (i); luai_threadyield(L);}
```

luai_threadyield 只是顺序地调用 lua_unlock 和 lua_lock,这里为释放一次锁,使

得别的线程可以得到调度.

函数调用类指令,与 CALL 为代表:

```
lvm.c:582
      case OP_CALL: {
        int b = GETARG_B(i);
        int nresults = GETARG_C(i) - 1;
        if (b != 0) L->top = ra+b;  /* else previous instruction set
top */
        L->savedpc = pc;
        switch (luaD_precall(L, ra, nresults)) {
          case PCRLUA: {
            nexeccalls++;
              goto reentry;  /* restart luaV_execute over new Lua
function */
```

```
          }
          case PCRC: {
             /* it was a C function (`precall' called it); adjust
results */
            if (nresults >= 0) L->top = L->ci->top;
            base = L->base;
            continue;
          }
          default: {
            return;  /* yield */
          }
        }
      }
lopcodes.c:192
OP_CALL,
/*    A B C    R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1)) */
```

这一条指令将在下一个介绍 Lua 函数调用规范的专题中详细介绍．在这里只是简单地说明 CALL 指令的 R[A]表示的是即将要调用的函数，而 B 和 C 则分别表示参数个数加 1，和返回值个数加 1．之所以这里需要加 1，其原因是：B 和 C 使用零来表示变长的参数和变长的返回值，而实际参数个数就向后推了一个．

指令的介绍就先到此为止了，其它的指令的实现也比较类似．仔细阅读源码就可很容易地分析出它的意义来．下一篇将是一个专题，详细地介绍 Lua 中函数的调用是如何实现的．