Lua源码剖析（三）：VM

lua VM的代码

lua的VM执行代码是从lvm.c中的 `void luaV_execute(lua_State *L)` 开始：

```
void luaV_execute (lua_State *L) {
  CallInfo *ci = L->ci;
  LClosure *cl;
  TValue *k;
  StkId base;
 newframe:  /* reentry point when frame changes (call/return) */
  lua_assert(ci == L->ci);
  cl = clLvalue(ci->func);
  k = cl->p->k;
  base = ci->u.l.base;
  /* main loop of interpreter */
  for (;;) {
    Instruction i = *(ci->u.l.savedpc++);
    StkId ra;
    if ((L->hookmask & (LUA_MASKLINE | LUA_MASKCOUNT)) &&
        (--L->hookcount == 0 || L->hookmask & LUA_MASKLINE)) {
```

```
  Protect(traceexec(L));
}
/* WARNING: several calls may realloc the stack and invalidate `ra' */
ra = RA(i);
lua_assert(base == ci->u.l.base);
lua_assert(base <= L->top && L->top < L->stack + L->stacksize);
vmdispatch (GET_OPCODE(i)) {
  vmcase(OP_MOVE,
    setobjs2s(L, ra, RB(i));
  )
  vmcase(OP_LOADK,
    TValue *rb = k + GETARG_Bx(i);
    setobj2s(L, ra, rb);
  )
  vmcase(OP_LOADKX,
    TValue *rb;
    lua_assert(GET_OPCODE(*ci->u.l.savedpc) == OP_EXTRAARG);
    rb = k + GETARG_Ax(*ci->u.l.savedpc++);
    setobj2s(L, ra, rb);
  )
  vmcase(OP_LOADBOOL,
    setbvalue(ra, GETARG_B(i));
    if (GETARG_C(i)) ci->u.l.savedpc++;  /* skip next instruction (if C) */
  )
  vmcase(OP_LOADNIL,
    int b = GETARG_B(i);
    do {
      setnilvalue(ra++);
```

```
  } while (b--);
)
vmcase(OP_GETUPVAL,
  int b = GETARG_B(i);
  setobj2s(L, ra, cl->upvals[b]->v);
)
vmcase(OP_GETTABUP,
  int b = GETARG_B(i);
  Protect(luaV_gettable(L, cl->upvals[b]->v, RKC(i), ra));
)
vmcase(OP_GETTABLE,
  Protect(luaV_gettable(L, RB(i), RKC(i), ra));
)
vmcase(OP_SETTABUP,
  int a = GETARG_A(i);
  Protect(luaV_settable(L, cl->upvals[a]->v, RKB(i), RKC(i)));
)
vmcase(OP_SETUPVAL,
  UpVal *uv = cl->upvals[GETARG_B(i)];
  setobj(L, uv->v, ra);
  luaC_barrier(L, uv, ra);
)
vmcase(OP_SETTABLE,
  Protect(luaV_settable(L, ra, RKB(i), RKC(i)));
)
vmcase(OP_NEWTABLE,
  int b = GETARG_B(i);
  int c = GETARG_C(i);
```

```
  Table *t = luaH_new(L);
  sethvalue(L, ra, t);
  if (b != 0 || c != 0)
    luaH_resize(L, t, luaO_fb2int(b), luaO_fb2int(c));
  checkGC(L, ra + 1);
)
vmcase(OP_SELF,
  StkId rb = RB(i);
  setobjs2s(L, ra+1, rb);
  Protect(luaV_gettable(L, rb, RKC(i), ra));
)
vmcase(OP_ADD,
  arith_op(luai_numadd, TM_ADD);
)
vmcase(OP_SUB,
  arith_op(luai_numsub, TM_SUB);
)
vmcase(OP_MUL,
  arith_op(luai_nummul, TM_MUL);
)
vmcase(OP_DIV,
  arith_op(luai_numdiv, TM_DIV);
)
vmcase(OP_MOD,
  arith_op(luai_nummod, TM_MOD);
)
vmcase(OP_POW,
  arith_op(luai_numpow, TM_POW);
```

```
)
vmcase(OP_UNM,
  TValue *rb = RB(i);
  if (ttisnumber(rb)) {
    lua_Number nb = nvalue(rb);
    setnvalue(ra, luai_numunm(L, nb));
  }
  else {
    Protect(luaV_arith(L, ra, rb, rb, TM_UNM));
  }
)
vmcase(OP_NOT,
  TValue *rb = RB(i);
  int res = l_isfalse(rb);  /* next assignment may change this value */
  setbvalue(ra, res);
)
vmcase(OP_LEN,
  Protect(luaV_objlen(L, ra, RB(i)));
)
vmcase(OP_CONCAT,
  int b = GETARG_B(i);
  int c = GETARG_C(i);
  StkId rb;
  L->top = base + c + 1;  /* mark the end of concat operands */
  Protect(luaV_concat(L, c - b + 1));
  ra = RA(i);  /* 'luav_concat' may invoke TMs and move the stack */
  rb = b + base;
  setobjs2s(L, ra, rb);
```

```
  checkGC(L, (ra >= rb ? ra + 1 : rb));
  L->top = ci->top;  /* restore top */
)
vmcase(OP_JMP,
  dojump(ci, i, 0);
)
vmcase(OP_EQ,
  TValue *rb = RKB(i);
  TValue *rc = RKC(i);
  Protect(
    if (cast_int(equalobj(L, rb, rc)) != GETARG_A(i))
      ci->u.l.savedpc++;
    else
      donextjump(ci);
  )
)
vmcase(OP_LT,
  Protect(
    if (luaV_lessthan(L, RKB(i), RKC(i)) != GETARG_A(i))
      ci->u.l.savedpc++;
    else
      donextjump(ci);
  )
)
vmcase(OP_LE,
  Protect(
    if (luaV_lessequal(L, RKB(i), RKC(i)) != GETARG_A(i))
      ci->u.l.savedpc++;
```

```
    else
      donextjump(ci);
  )
)
vmcase(OP_TEST,
  if (GETARG_C(i) ? l_isfalse(ra) : !l_isfalse(ra))
      ci->u.l.savedpc++;
    else
    donextjump(ci);
)
vmcase(OP_TESTSET,
  TValue *rb = RB(i);
  if (GETARG_C(i) ? l_isfalse(rb) : !l_isfalse(rb))
    ci->u.l.savedpc++;
  else {
    setobjs2s(L, ra, rb);
    donextjump(ci);
  }
)
vmcase(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 */
  if (luaD_precall(L, ra, nresults)) {  /* C function? */
    if (nresults >= 0) L->top = ci->top;  /* adjust results */
    base = ci->u.l.base;
  }
  else {  /* Lua function */
```

```
      ci = L->ci;
      ci->callstatus |= CIST_REENTRY;
      goto newframe;  /* restart luaV_execute over new Lua function */
    }
  )
  vmcase(OP_TAILCALL,
    int b = GETARG_B(i);
    if (b != 0) L->top = ra+b;  /* else previous instruction set top */
    lua_assert(GETARG_C(i) - 1 == LUA_MULTRET);
    if (luaD_precall(L, ra, LUA_MULTRET))  /* C function? */
      base = ci->u.l.base;
    else {
      /* tail call: put called frame (n) in place of caller one (o) */
      CallInfo *nci = L->ci;  /* called frame */
      CallInfo *oci = nci->previous;  /* caller frame */
      StkId nfunc = nci->func;  /* called function */
      StkId ofunc = oci->func;  /* caller function */
      /* last stack slot filled by 'precall' */
      StkId lim = nci->u.l.base + getproto(nfunc)->numparams;
      int aux;
      /* close all upvalues from previous call */
      if (cl->p->sizep > 0) luaF_close(L, oci->u.l.base);
      /* move new frame into old one */
      for (aux = 0; nfunc + aux < lim; aux++)
        setobjs2s(L, ofunc + aux, nfunc + aux);
      oci->u.l.base = ofunc + (nci->u.l.base - nfunc);  /* correct base */
      oci->top = L->top = ofunc + (L->top - nfunc);  /* correct top */
      oci->u.l.savedpc = nci->u.l.savedpc;
```

```
    oci->callstatus |= CIST_TAIL;  /* function was tail called */
    ci = L->ci = oci;  /* remove new frame */
    lua_assert(L->top == oci->u.l.base + getproto(ofunc)->maxstacksize);
    goto newframe;  /* restart luaV_execute over new Lua function */
  }
)
vmcasenb(OP_RETURN,
  int b = GETARG_B(i);
  if (b != 0) L->top = ra+b-1;
  if (cl->p->sizep > 0) luaF_close(L, base);
  b = luaD_poscall(L, ra);
  if (!(ci->callstatus & CIST_REENTRY))  /* 'ci' still the called one */
    return;  /* external invocation: return */
  else {  /* invocation via reentry: continue execution */
    ci = L->ci;
    if (b) L->top = ci->top;
    lua_assert(isLua(ci));
    lua_assert(GET_OPCODE(*((ci)->u.l.savedpc - 1)) == OP_CALL);
    goto newframe;  /* restart luaV_execute over new Lua function */
  }
)
vmcase(OP_FORLOOP,
  lua_Number step = nvalue(ra+2);
  lua_Number idx = luai_numadd(L, nvalue(ra), step); /* increment index */
  lua_Number limit = nvalue(ra+1);
  if (luai_numlt(L, 0, step) ? luai_numle(L, idx, limit)
                             : luai_numle(L, limit, idx)) {
    ci->u.l.savedpc += GETARG_sBx(i);  /* jump back */
```

```
      setnvalue(ra, idx);  /* update internal index */
      setnvalue(ra+3, idx);  /* and external index */
    }
  )
  vmcase(OP_FORPREP,
    const TValue *init = ra;
    const TValue *plimit = ra+1;
    const TValue *pstep = ra+2;
    if (!tonumber(init, ra))
      luaG_runerror(L, LUA_QL("for") " initial value must be a number");
    else if (!tonumber(plimit, ra+1))
      luaG_runerror(L, LUA_QL("for") " limit must be a number");
    else if (!tonumber(pstep, ra+2))
      luaG_runerror(L, LUA_QL("for") " step must be a number");
    setnvalue(ra, luai_numsub(L, nvalue(ra), nvalue(pstep)));
    ci->u.l.savedpc += GETARG_sBx(i);
  )
  vmcasenb(OP_TFORCALL,
    StkId cb = ra + 3;  /* call base */
    setobjs2s(L, cb+2, ra+2);
    setobjs2s(L, cb+1, ra+1);
    setobjs2s(L, cb, ra);
    L->top = cb + 3;  /* func. + 2 args (state and index) */
    Protect(luaD_call(L, cb, GETARG_C(i), 1));
    L->top = ci->top;
    i = *(ci->u.l.savedpc++);  /* go to next instruction */
    ra = RA(i);
    lua_assert(GET_OPCODE(i) == OP_TFORLOOP);
```

```
  goto l_tforloop;
)
vmcase(OP_TFORLOOP,
  l_tforloop:
  if (!ttisnil(ra + 1)) {  /* continue loop? */
    setobjs2s(L, ra, ra + 1);  /* save control variable */
     ci->u.l.savedpc += GETARG_sBx(i);  /* jump back */
  }
)
vmcase(OP_SETLIST,
  int n = GETARG_B(i);
  int c = GETARG_C(i);
  int last;
  Table *h;
  if (n == 0) n = cast_int(L->top - ra) - 1;
  if (c == 0) {
    lua_assert(GET_OPCODE(*ci->u.l.savedpc) == OP_EXTRAARG);
    c = GETARG_Ax(*ci->u.l.savedpc++);
  }
  luai_runtimecheck(L, ttistable(ra));
  h = hvalue(ra);
  last = ((c-1)*LFIELDS_PER_FLUSH) + n;
  if (last > h->sizearray)  /* needs more space? */
    luaH_resizearray(L, h, last);  /* pre-allocate it at once */
  for (; n > 0; n--) {
    TValue *val = ra+n;
    luaH_setint(L, h, last--, val);
    luaC_barrierback(L, obj2gco(h), val);
```

```
  }
  L->top = ci->top;  /* correct top (in case of previous open call) */
)
vmcase(OP_CLOSURE,
  Proto *p = cl->p->p[GETARG_Bx(i)];
  Closure *ncl = getcached(p, cl->upvals, base);  /* cached closure */
  if (ncl == NULL)  /* no match? */
    pushclosure(L, p, cl->upvals, base, ra);  /* create a new one */
  else
    setclLvalue(L, ra, ncl);  /* push cashed closure */
  checkGC(L, ra + 1);
)
vmcase(OP_VARARG,
  int b = GETARG_B(i) - 1;
  int j;
  int n = cast_int(base - ci->func) - cl->p->numparams - 1;
  if (b < 0) {  /* B == 0? */
    b = n;  /* get all var. arguments */
    Protect(luaD_checkstack(L, n));
    ra = RA(i);  /* previous call may change the stack */
    L->top = ra + n;
  }
  for (j = 0; j < b; j++) {
    if (j < n) {
      setobjs2s(L, ra + j, base - n + j);
    }
    else {
      setnilvalue(ra + j);
```

```
          }
        }
      )
      vmcase(OP_EXTRAARG,
        lua_assert(0);
      )
    }
  }
}
```

此函数先从 `CallInfo` 中取出运行的lua closure，取出这个closure的寄存器的 `base` 指针和closure的函数 `Proto` 的常量列表 `k` 。

debug hook

进入for循环开始执行代码，先取出当前指令 `Instruction` ，根据Lua State的hook mask来判断是否需要hook代码执行，这个hook代码执行就是lua提供给外界调试代码的库，我们可以使用这个debug库实现自己的调试器，两年前我使用这个debug实现过一个简单的lua调试器。（博客：http://www.cppblog.com/airtrack/archive/2011/01/01/137825.html 代码放在github上：https://github.com/airtrack/lua-debugger）

lua提供了四种 `hookmask` ，分别是：

```
#define LUA_MASKCALL     (1 << LUA_HOOKCALL)
#define LUA_MASKRET     (1 << LUA_HOOKRET)
#define LUA_MASKLINE     (1 << LUA_HOOKLINE)
#define LUA_MASKCOUNT     (1 << LUA_HOOKCOUNT)
```

`LUA_MASKCALL`表示每次调用函数的时候hook； `LUA_MASKRET`表示每次函数返回的时候hook； `LUA_MASKLINE`表示每行执行的时候hook；

`LUA_MASKCOUNT`表示每执行count条lua指令hook一次，这里的count是`debug.sethook ([thread,] hook, mask [, count])`中传递的。

`LUA_MASKLINE`和`LUA_MASKCOUNT`类型的hook是在函数的开头这段代码里hook：

```
if ((L->hookmask & (LUA_MASKLINE | LUA_MASKCOUNT)) &&
    (--L->hookcount == 0 || L->hookmask & LUA_MASKLINE)) {
  Protect(traceexec(L));
}
```

而`LUA_MASKCALL`和`LUA_MASKRET`类型的hook则分别在call和return的时候hook，具体是在ldo.c中的`luaD_precall`和`luaD_poscall`中hook。

如果设置了debug hook，那执行指令的时候就会检测一下是否需要调用hook函数。若需要`LUA_MASKLINE`或`LUA_MASKCOUNT`的hook则调用lvm.c中的`traceexec`函数，而`traceexec`函数通过调用ldo.c中的`luaD_hook`函数完成；若需要`LUA_MASKCALL`或`LUA_MASKRET`的hook则

ldo.c中的 `luaD_precall` 和 `luaD_poscall` 会对hook进行检测，最终还是调用到ldo.c中的 `luaD_hook` 函数完成。

```
void luaD_hook (lua_State *L, int event, int line) {
  lua_Hook hook = L->hook;
  if (hook && L->allowhook) {
    CallInfo *ci = L->ci;
    ptrdiff_t top = savestack(L, L->top);
    ptrdiff_t ci_top = savestack(L, ci->top);
    lua_Debug ar;
    ar.event = event;
    ar.currentline = line;
    ar.i_ci = ci;
    luaD_checkstack(L, LUA_MINSTACK);  /* ensure minimum stack size */
    ci->top = L->top + LUA_MINSTACK;
    lua_assert(ci->top <= L->stack_last);
    L->allowhook = 0;  /* cannot call hooks inside a hook */
    ci->callstatus |= CIST_HOOKED;
    lua_unlock(L);
    (*hook)(L, &ar);
    lua_lock(L);
    lua_assert(!L->allowhook);
    L->allowhook = 1;
    ci->top = restorestack(L, ci_top);
    L->top = restorestack(L, top);
    ci->callstatus &= ~CIST_HOOKED;
  }
}
```

我们发现这个调用的hook函数是注册在 `L->hook` 中的C函数指针，我们通过 `debug.sethook` 注册的hook函数是lua的函数，那这个注册的C函数肯定是用来完成lua函数与C函数之间的转换。

`L->hook` 这个函数指针的注册是通过ldebug.c中的 `lua_sethook` 函数完成：

```
LUA_API int lua_sethook (lua_State *L, lua_Hook func, int mask, int count) {
  if (func == NULL || mask == 0) {  /* turn off hooks? */
    mask = 0;
    func = NULL;
  }
  if (isLua(L->ci))
    L->oldpc = L->ci->u.l.savedpc;
  L->hook = func;
  L->basehookcount = count;
  resethookcount(L);
  L->hookmask = cast_byte(mask);
  return 1;
}
```

在ldblib.c中的 `db_sethook` 中调用了 `lua_sethook` 函数，这个hook函数是ldblib.c中的 `hookf` ：

```
static void hookf (lua_State *L, lua_Debug *ar) {
  static const char *const hooknames[] =
    {"call", "return", "line", "count", "tail call"};
```

```
  gethooktable(L);
  lua_pushthread(L);
  lua_rawget(L, -2);
  if (lua_isfunction(L, -1)) {
    lua_pushstring(L, hooknames[(int)ar->event]);
    if (ar->currentline >= 0)
      lua_pushinteger(L, ar->currentline);
    else lua_pushnil(L);
    lua_assert(lua_getinfo(L, "lS", ar));
    lua_call(L, 2, 0);
  }
}
```

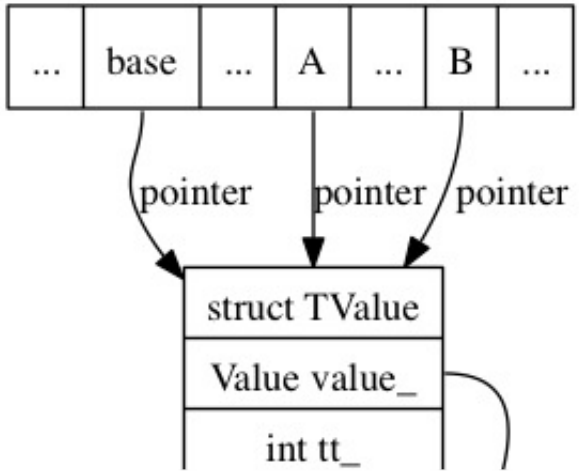
这个函数就把注册的lua hook函数取出来然后调用，传递的hook类型作为hook函数的第一参数，分别是 `{"call", "return", "line", "count", "tail call"}` 。

寄存器结构

lua是寄存器虚拟机，它为每个函数在运行时期最多分配250个寄存器。函数运行时都是通过这些寄存器来操作数据，指令操作寄存器的参数都是记录着相应寄存器的下标。在for循环中，通过 `RA(i)` 获取到指令 `i` 的参数 `A` 的寄存器，lua指令格式在上一篇中有介绍， `RA` 宏获得 `A` 参数的寄存器下标，再加上当前运行函数的 `base` 指针，就可以得出相应的寄存器。再之后通过 `GET_OPCODE(i)` 获得 `opcode` 并进入 `switch-case` ，分别针对每条指令类型取出相应的其它

指令参数并执行。

lua寄存器结构如图：



对每条指令分别根据指令类型操作`A`、`B`、`C`、`Ax`、`Bx`、`sBx`参数，参数可以是寄存器的下标，也可以是`Proto`的常量列表`k`的下标。`case`的第一条指令`OP_MOVE`就是最简单的指令，从指令`i`中取出参数`B`，然后把`B`指向的`TValue`赋值给`A`指向的`TValue`。从常量列表中把`TValue` load到寄存器中的指令有两种，分别

是`OP_LOADK`和`OP_LOADKX`。在`OP_LOADK`中，参数`Bx`就是`Proto`的常量列表的下标，然后简单的将这个`TValue` load到寄存器`RA(i)`中，如果一个函数的常量很多，个数超过了，参数`Bx`（14~31bits，共18位）的表示范围，这时候就要使用`OP_LOADKX`指令表示。在`OP_LOADKX`指令中，会继续读取下一条指令，下一条指令的类型是`OP_EXTRAARG`，它的参数是`Ax`（6~31bits，共26位）来表示`Proto`的常量列表的下标，这样常量的个数就扩大到了26位的表示范围。

函数调用的栈结构

lua的函数调用指令是`OP_CALL`和`OP_TAILCALL`，实际上函数调用是通过`luaD_precall`完成，这个函数判断被调用的函数是否是C函数，如果是C函数的话那就将函数执行完返回，如果不是则准备好一些基本数据，并把指令切换到被调用的lua函数的指令地址上，然后执行被调用函数的指令。

```
int luaD_precall (lua_State *L, StkId func, int nresults) {
  lua_CFunction f;
  CallInfo *ci;
  int n;  /* number of arguments (Lua) or returns (C) */
  ptrdiff_t funcr = savestack(L, func);
  switch (ttype(func)) {
    case LUA_TLCF:  /* light C function */
      f = fvalue(func);
      goto Cfunc;
```

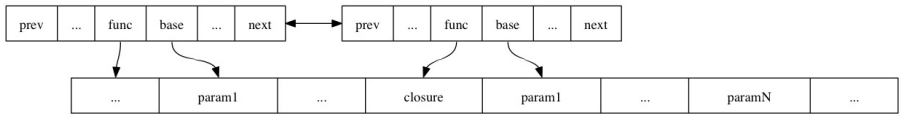
```
case LUA_TCCL: {  /* C closure */
  f = clCvalue(func)->f;
 Cfunc:
  luaD_checkstack(L, LUA_MINSTACK);  /* ensure minimum stack size */
  ci = next_ci(L);  /* now 'enter' new function */
  ci->nresults = nresults;
  ci->func = restorestack(L, funcr);
  ci->top = L->top + LUA_MINSTACK;
  lua_assert(ci->top <= L->stack_last);
  ci->callstatus = 0;
  if (L->hookmask & LUA_MASKCALL)
    luaD_hook(L, LUA_HOOKCALL, -1);
  lua_unlock(L);
  n = (*f)(L);  /* do the actual call */
  lua_lock(L);
  api_checknelems(L, n);
  luaD_poscall(L, L->top - n);
  return 1;
}
case LUA_TLCL: {  /* Lua function: prepare its call */
  StkId base;
  Proto *p = clLvalue(func)->p;
  luaD_checkstack(L, p->maxstacksize);
  func = restorestack(L, funcr);
  n = cast_int(L->top - func) - 1;  /* number of real arguments */
  for (; n < p->numparams; n++)
    setnilvalue(L->top++);  /* complete missing arguments */
  base = (!p->is_vararg) ? func + 1 : adjust_varargs(L, p, n);
```

```
      ci = next_ci(L);  /* now 'enter' new function */
      ci->nresults = nresults;
      ci->func = func;
      ci->u.l.base = base;
      ci->top = base + p->maxstacksize;
      lua_assert(ci->top <= L->stack_last);
      ci->u.l.savedpc = p->code;  /* starting point */
      ci->callstatus = CIST_LUA;
      L->top = ci->top;
      if (L->hookmask & LUA_MASKCALL)
        callhook(L, ci);
      return 0;
    }
    default: {  /* not a function */
      func = tryfuncTM(L, func);  /* retry with 'function' tag method */
      return luaD_precall(L, func, nresults);  /* now it must be a function */
    }
  }
}
```

lua的函数调用栈是通过一个`CallInfo`的链表来表示，每一个`CallInfo`链表元素表示一层函数调用，每个`CallInfo`通过`prev`和`next`指针分别指向前面的函数和后面的函数。`CallInfo`中的`base`和`top`分别指向这个调用栈帧的起始地址和结束地址，`base`到`top`这些栈空间在函数运行内部就是可用的寄存器。`func`则指向这个被调用函数的closure所在lua栈中的地址。

函数`CallInfo`链表结构与lua的栈的格式的关系有如下3种：

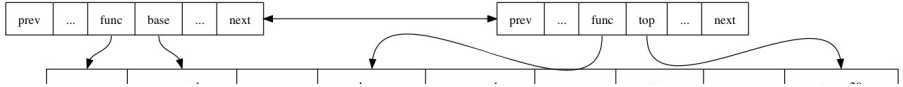
1.被调用函数为普通lua函数时，调用者把被调用函数的closure放到栈中，然后把传入函数的参数依次放入栈中。被调用者的 `CallInfo` 中的 `func` 指针指向它所属的closure，并把这个运行时期的 `base` 指针指向传进来的第一参数，如下图：



2.被调用函数是 `vararg` （变参）的lua函数时，被调用者的 `CallInfo` 的 `func` 还是指向相应的closure，固定参数则会复制一份，并把原来的设置为 `nil` ，而多出的参数则保留在原始位置，并将 `base` 指针指向复制的第一个实参。这样， `base` 指针前面的就是多出的参数，即固定的参数是从 `base` 指针指向的地方开始，而变参数则在 `base` 指针前面，这样可以保证后续的指令访问固定的参数跟非可变参数函数（第一种情况）时一致。

例如：一个变参函数 `function f(a, b, ...) end` ，这样调用 `f(1, 2, 3, 4)` ，那么会把 `1` 和 `2` 复制一份，分别作为 `a` 和 `b` 的实参， `3` 和 `4` 则保留在原始位置，也就是在 `base` 指针之前。

3.被调用的函数是C函数的时候， `CallIInfo` 的 `top` 指向 `L->top + LUA_MINSTACK(20)` ，为C函数操作lua栈预留的最小stack空间，在被调用的C函数中若使用的lua栈空间比较多时，需要调用 `lua_checkstack` 来向lua申请保证有足够的栈空间使用，不然就会出现lua stack overflow的错误。



函数调用完成后，lua通过指令`OP_RETURN`返回，这时候，最后一个`CallIInfo`就回收了。在回收之前，通过`luaD_poscall`来将函数的返回值复制到相应的位置，函数返回值复制到的位置的起点就是closure的位置，把closure覆盖掉。若调用的`CallInfo`表示的是C函数时，也是通过`luaD_poscall`完成返回值的复制。

```
int luaD_poscall (lua_State *L, StkId firstResult) {
  StkId res;
  int wanted, i;
  CallInfo *ci = L->ci;
  if (L->hookmask & (LUA_MASKRET | LUA_MASKLINE)) {
    if (L->hookmask & LUA_MASKRET) {
      ptrdiff_t fr = savestack(L, firstResult);  /* hook may change stack */
      luaD_hook(L, LUA_HOOKRET, -1);
      firstResult = restorestack(L, fr);
    }
    L->oldpc = ci->previous->u.l.savedpc;  /* 'oldpc' for caller function */
  }
  res = ci->func;  /* res == final position of 1st result */
  wanted = ci->nresults;
  L->ci = ci = ci->previous;  /* back to caller */
  /* move results to correct place */
  for (i = wanted; i != 0 && firstResult < L->top; i--)
    setobjs2s(L, res++, firstResult++);
  while (i-- > 0)
```

```
    setnilvalue(res++);
  L->top = res;
  return (wanted - LUA_MULTRET);  /* 0 iff wanted == LUA_MULTRET */
}
```

尾递归

lua对于递归会有尾递归优化，如果一个函数调用是尾递归的话，那么函数的调用栈是不会增长的。lua通过 `OP_TAILCALL` 指令完成尾递归调用，这条指令的前面一段跟 `OP_CALL` 相似，通过 `luaD_precall` 增加函数调用栈信息 `CallInfo` 。当 `luaD_precall` 返回时，调用的不是C函数，则会将新增的 `CallInfo` 与上一个 `CallInfo` 栈帧合并，然后把新增的 `CallInfo` 移除掉，这样的尾递归调用就不会导致栈帧增长了。

lua的其它指令就是很明确的操作一些寄存器和常量来完成代码执行。