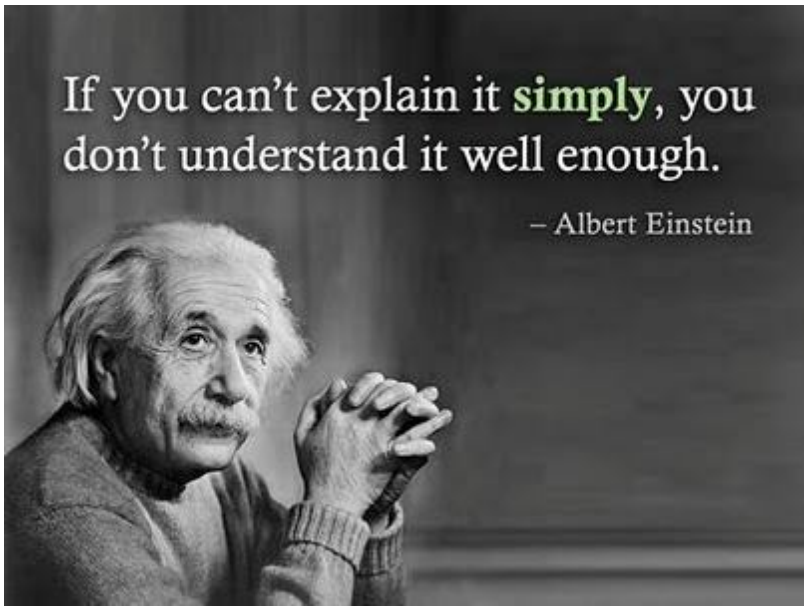
lua 源码分析(上)

有分享交流才有进步，永远不要固步自封



未来是低级语言的，也是高级语言的，但终归是 web 的

2012/2/1

Kuafu

目 录

1 参考资料

- 官网

http://www.lua.org/

- lua 程序设计

Roberto Ierusalimschy 著,周惟迪译

- Lua 5.1 参考手册

云风译
http://www.codingnow.com/2000/download/lua_manual.html

- The Lua Architecture

Advanced Topics in Software Engineering

- Lua 5.2 Reference Manual

refman-5.0.pdf
http://www.lua.org/manual/5.2/manual.html

- Lua 源码欣赏

Lua 源码欣赏.pdf
云风著

- Lua Source

Lua 代码结构
http://lua-users.org/wiki/LuaSource

- lua 编译笔记

http://www.cppblog.com/flyindark/archive/2011/05/01/145475.aspx

- lua-c 笔记

http://www.cppblog.com/flyindark/archive/2011/07/01/149937.html

- 代码注释

https://github.com/davidm/lua-annotate
http://stevedonovan.github.com/lua-5.1.4/
对源码进行了注释

- Sample Code

http://lua-users.org/wiki/SampleCode
官网提供的各种代码片段
Technical Notes
老的一些技术文档、已经不再维护
http://www.lua.org/notes/

lua 编程精粹
Programming Gems
http://www.lua.org/gems/

2 阅读说明

本文作为学习笔记边学习的同时边整理边深入而完成，很多概念并非一步到位的作出了正确的解释。Thread 和 gc 这两类高级主题不做介绍。

本文非常多的地方参考和引用了参考资料，并做了适当的引用说明，但是没有特意保证所有引用都有说明。

本文 lua 源码分析由于最近比较忙，已经有 2 个多月未继续，暂且先叫上部分吧，其实还未真正深入分析 lua 的实现机制，希望有时间后真正将 lua 的实现机制搞清楚，最终能熟练且灵活的将 lua 中很多高效灵活的机制运用到项目开发中去。

2.1 阅读本文的方式

本文没有固定阅读顺序，一些猜想、闲谈、宏观分析的小结可以先读。

2.2 源码阅读顺序[1]

本文遵从以下顺序分析源码，但是各个章节的组织顺序不是如此。

- lmathlib.c, lstrlib.c: get familiar with the external C API. Don't bother with the pattern matcher though. Just the easy functions.
- lapi.c: Check how the API is implemented internally. Only skim this to get a feeling for the code. Cross-reference to lua.h and luaconf.h as needed.
- lobject.h: tagged values and object representation. skim through this first. you'll want to keep a window with this file open all the time.
- lstate.h: state objects. ditto.
- lopcodes.h: bytecode instruction format and opcode definitions. easy.
- lvm.c: scroll down to luaV_execute, the main interpreter loop. see how all of the instructions are implemented. skip the details for now. reread later.

[1] 参考
http://www.reddit.com/comments/63hth/ask_reddit_which_oss_codebases_out_there_are_so/c02pxbp

- ldo.c: calls, stacks, exceptions, coroutines. tough read.
- lstring.c: string interning. cute, huh?
- ltable.c: hash tables and arrays. tricky code.
- ltm.c: metamethod handling, reread all of lvm.c now.
- You may want to reread lapi.c now.
- ldebug.c: surprise waiting for you. abstract interpretation is used to find object names for tracebacks. does bytecode verification, too.
- lparser.c, lcode.c: recursive descent parser, targetting a register-based VM. start from chunk() and work your way through. read the expression parser and the code generator parts last.
- lgc.c: incremental garbage collector. take your time.
- Read all the other files as you see references to them. Don't let your stack get too deep though.

读的精髓在浅尝辄止，雁过留痕，囫囵吞枣最好！

2.3 Lapi

Lua 是 lua c api 部分，排在阅读顺序的第二次序。这个 api 中的所有函数均有注释。参考官方文档和云风等中文译者的文档注释[2]:

API 列表

lua_State *()	lua_newstate (lua_Alloc f, void *ud)
void()	lua_close (lua_State *L)
lua_State *()	lua_newthread (lua_State *L)
lua_CFunction()	lua_atpanic (lua_State *L, lua_CFunction panicf)
const lua_Number *()	lua_version (lua_State *L)
int()	lua_absindex (lua_State *L, int idx)
int()	lua_gettop (lua_State *L)
void()	lua_settop (lua_State *L, int idx)
void()	lua_pushvalue (lua_State *L, int idx)
void()	lua_remove (lua_State *L, int idx)
void()	lua_insert (lua_State *L, int idx)
void()	lua_replace (lua_State *L, int idx)
void()	lua_copy (lua_State *L, int fromidx, int toidx)
int()	lua_checkstack (lua_State *L, int sz)
void()	lua_xmove (lua_State *from, lua_State *to, int n)
int()	lua_isnumber (lua_State *L, int idx)
int()	lua_isstring (lua_State *L, int idx)
int()	lua_iscfunction (lua_State *L, int idx)
int()	lua_isuserdata (lua_State *L, int idx)
int()	lua_type (lua_State *L, int idx)
const char *()	lua_typename (lua_State *L, int tp)
lua_Number()	lua_tonumberx (lua_State *L, int idx, int *isnum)

[2] Ver5.1 和 5.2 有不同的手册文档

lua_Integer()	lua_tointegerx (lua_State *L, int idx, int *isnum)
lua_Unsigned()	lua_tounsignedx (lua_State *L, int idx, int *isnum)
int()	lua_toboolean (lua_State *L, int idx)
const char *()	lua_tolstring (lua_State *L, int idx, size_t *len)
size_t()	lua_rawlen (lua_State *L, int idx)
lua_CFunction()	lua_tocfunction (lua_State *L, int idx)
void *()	lua_touserdata (lua_State *L, int idx)
lua_State *()	lua_tothread (lua_State *L, int idx)
const void *()	lua_topointer (lua_State *L, int idx)
void()	lua_arith (lua_State *L, int op)
int()	lua_rawequal (lua_State *L, int idx1, int idx2)
int()	lua_compare (lua_State *L, int idx1, int idx2, int op)
void()	lua_pushnil (lua_State *L)
void()	lua_pushnumber (lua_State *L, lua_Number n)
void()	lua_pushinteger (lua_State *L, lua_Integer n)
void()	lua_pushunsigned (lua_State *L, lua_Unsigned n)
const char *()	lua_pushlstring (lua_State *L, const char *s, size_t l)
const char *()	lua_pushstring (lua_State *L, const char *s)
const char *()	lua_pushvfstring (lua_State *L, const char *fmt, va_list argp)
const char *()	lua_pushfstring (lua_State *L, const char *fmt,...)
void()	lua_pushcclosure (lua_State *L, lua_CFunction fn, int n)
void()	lua_pushboolean (lua_State *L, int b)
void()	lua_pushlightuserdata (lua_State *L, void *p)
int()	lua_pushthread (lua_State *L)
void()	lua_getglobal (lua_State *L, const char *var)
void()	lua_gettable (lua_State *L, int idx)
void()	lua_getfield (lua_State *L, int idx, const char *k)
void()	lua_rawget (lua_State *L, int idx)
void()	lua_rawgeti (lua_State *L, int idx, int n)
void()	lua_rawgetp (lua_State *L, int idx, const void *p)
void()	lua_createtable (lua_State *L, int narr, int nrec)
void *()	lua_newuserdata (lua_State *L, size_t sz)
int()	lua_getmetatable (lua_State *L, int objindex)
void()	lua_getuservalue (lua_State *L, int idx)
void()	lua_setglobal (lua_State *L, const char *var)
void()	lua_settable (lua_State *L, int idx)
void()	lua_setfield (lua_State *L, int idx, const char *k)
void()	lua_rawset (lua_State *L, int idx)
void()	lua_rawseti (lua_State *L, int idx, int n)
void()	lua_rawsetp (lua_State *L, int idx, const void *p)
int()	lua_setmetatable (lua_State *L, int objindex)
void()	lua_setuservalue (lua_State *L, int idx)
void()	lua_callk (lua_State *L, int nargs, int nresults, int ctx, lua_CFunction k)
int()	lua_getctx (lua_State *L, int *ctx)

int()	lua_pcallk (lua_State *L, int nargs, int nresults, int errfunc, int ctx, lua_CFunction k)
int()	lua_load (lua_State *L, lua_Reader reader, void *dt, const char *chunkname, const char *mode)
int()	lua_dump (lua_State *L, lua_Writer writer, void *data)
int()	lua_yieldk (lua_State *L, int nresults, int ctx, lua_CFunction k)
int()	lua_resume (lua_State *L, lua_State *from, int narg)
int()	lua_status (lua_State *L)
int()	lua_gc (lua_State *L, int what, int data)
int()	lua_error (lua_State *L)
int()	lua_next (lua_State *L, int idx)
void()	lua_concat (lua_State *L, int n)
void()	lua_len (lua_State *L, int idx)
lua_Alloc()	lua_getallocf (lua_State *L, void **ud)
void()	lua_setallocf (lua_State *L, lua_Alloc f, void *ud)
int()	lua_getstack (lua_State *L, int level, lua_Debug *ar)
int()	lua_getinfo (lua_State *L, const char *what, lua_Debug *ar)
const char *()	lua_getlocal (lua_State *L, const lua_Debug *ar, int n)
const char *()	lua_setlocal (lua_State *L, const lua_Debug *ar, int n)
const char *()	lua_getupvalue (lua_State *L, int funcindex, int n)
const char *()	lua_setupvalue (lua_State *L, int funcindex, int n)
void *()	lua_upvalueid (lua_State *L, int fidx, int n)
void()	lua_upvaluejoin (lua_State *L, int fidx1, int n1, int fidx2, int n2)
int()	lua_sethook (lua_State *L, lua_Hook func, int mask, int count)
lua_Hook()	lua_gethook (lua_State *L)
int()	lua_gethookmask (lua_State *L)
int()	lua_gethookcount (lua_State *L)

2.3.1 index2addr

将栈中 idx 位置处元素转为表类型。这是一个内部调用方法，仅在 lapi 域内有效。

```
static TValue *index2addr (lua_State *L, int idx) {
  CallInfo *ci = L->ci;
  if (idx > 0) {
    TValue *o = ci->func + idx;
    api_check(L, idx <= ci->top - (ci->func + 1), "unacceptable index");
    if (o >= L->top) return NONVALIDVALUE;
    else return o;
  }
  else if (idx > LUA_REGISTRYINDEX) {
    api_check(L, idx != 0 && -idx <= L->top - (ci->func + 1), "invalid index");
    return L->top + idx;
  }
  else if (idx == LUA_REGISTRYINDEX)
    return &G(L)->l_registry;
```

```
  else {  /* upvalues */
    idx = LUA_REGISTRYINDEX - idx;
    api_check(L, idx <= MAXUPVAL + 1, "upvalue index too large");
    if (ttislcf(ci->func))  /* light C function? */
      return NONVALIDVALUE;  /* it has no upvalues */
    else {
      CClosure *func = clCvalue(ci->func);
      return (idx <= func->nupvalues) ? &func->upvalue[idx-1] : NONVALIDVALUE;
    }
  }
}
```

REGISTRY:= LUA_REGISTRYINDEX

LUA_REGISTRYINDEX=(-LUAI_MAXSTACK - 1000)

	idx		
State1	>0	L->ci->func+idx	返回 L->ci->func 为栈基址的表元素中第 idx 个对象
	0	N/A	
State2	(REGISTRY ~0]		返回 L->top 为栈基址的表元素中的第 idx 个表对象
State3	REGISTRY	G(L)->l_registry	返回&L->l_G-> l_registry 表对象，用于表示全局表
State4	< REGISTRY		将 idx 转为 c 闭包 idxc，并将 L->ci->func 转为 c 闭包函数 func 取出闭包函数 func 在 idx 处的 upvalue

数轴

State4	State3	State2	State2	State1
< REGISTRY	**REGISTRY**	**(REGISTRY ~0)**	**0**	**>0**
	G(L)->l_registry		N/A	
闭包函数的 upvalue	全局表	常规堆栈中的对象	无效	Callfino 中的函数

State1：

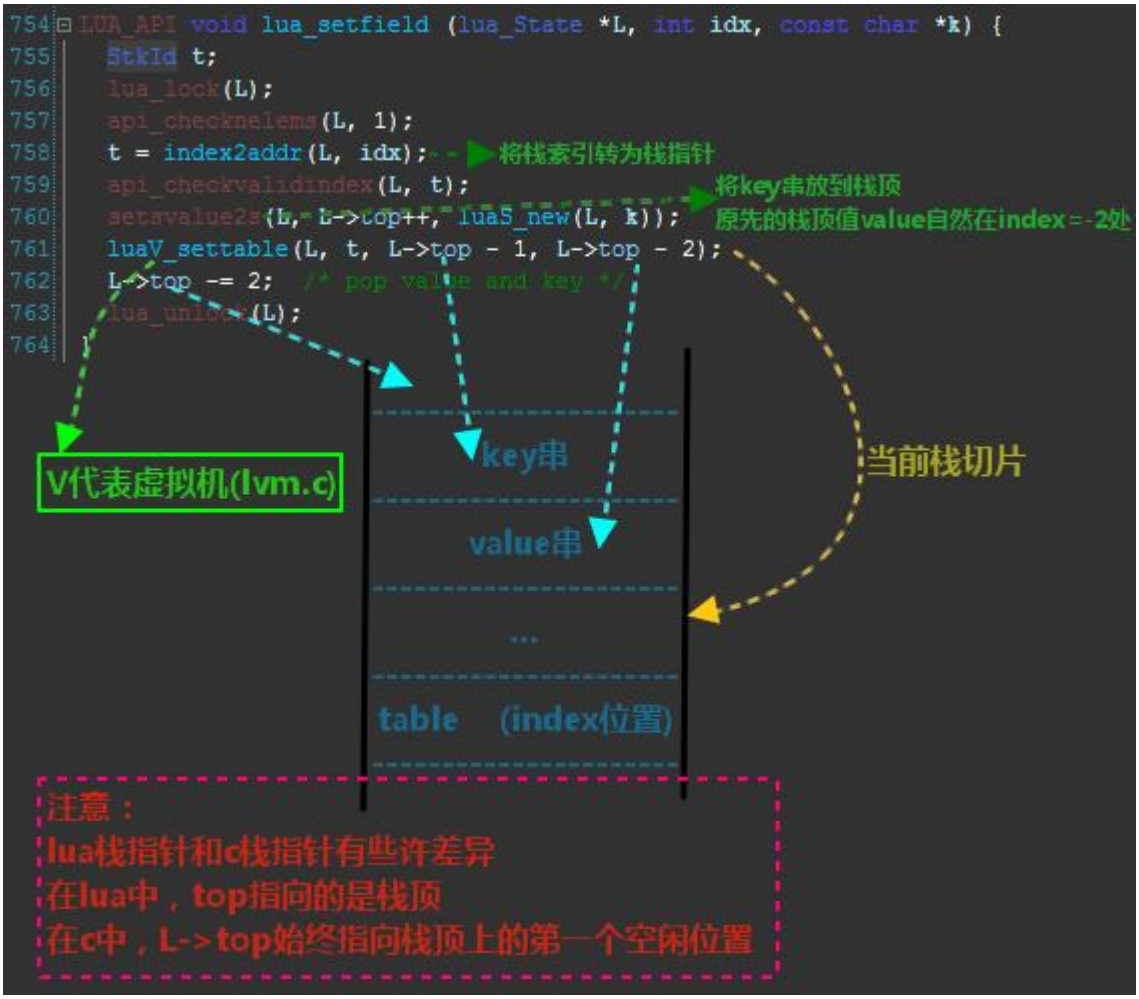
idx <= ci->top - (ci->func + 1)

```
static TValue *index2addr (lua_State *L, int idx) {
  CallInfo *ci = L->ci;
  if (idx > 0) {
    TValue *o = ci->func + idx;
    api_check(L, idx <= ci->top - (ci->func + 1), "unacceptable index");
    if (o >= L->top) return NONVALIDVALUE;
    else return o;
  }
  else if (idx > LUA_REGISTRYINDEX) {
    api_check(L, idx != 0 && -idx <= L->top - (ci->func + 1), "invalid index");
    return L->top + idx;
  }
  else if (idx == LUA_REGISTRYINDEX)
    return &G(L)->l_registry;
  else {  /* upvalues */
    idx = LUA_REGISTRYINDEX - idx;
    api_check(L, idx <= MAXUPVAL + 1, "upvalue index too large");
    if (ttislcf(ci->func))  /* light C function? */
      return NONVALIDVALUE;  /* it has no upvalues */
    else {
      CClosure *func = clCvalue(ci->func);
      return (idx <= func->nupvalues) ? &func->upvalue[idx-1] : NONVALIDVALUE;
    }
  }
}
```

2.3.2 lua_setfield

做一个等价于 t[k] = v 的操作， 这里 t 是给出的有效索引 index 处的值， 而 v 是栈顶的那个值。 这个函数将把这个值弹出堆栈。 跟在 Lua 中一样，这个函数可能触发一个 "newindex" 事件的元方法。

```
LUA_API void lua_setfield (lua_State *L, int idx, const char *k) {
  StkId t;
  lua_lock(L);
  api_checknelems(L, 1);
  t = index2addr(L, idx);
  api_checkvalidindex(L, t);
  setsvalue2s(L, L->top++, luaS_new(L, k));
  luaV_settable(L, t, L->top - 1, L->top - 2);
  L->top -= 2;  /* pop value and key */
  lua_unlock(L);
}
```



2.4 Lobject

标签值和对象。

2.4.1 Value

Value

```
union Value {
  GCObject *gc;    /* collectable objects */
  void *p;         /* light userdata */
```

```
  int b;              /* booleans */
  lua_CFunction f; /* light C functions */
  numfield            /* numbers */
};
```

Tvalue\ StkId

```
#define TValuefields    Value value_; int tt_
typedef struct lua_TValue TValue;
struct lua_TValue {
  TValuefields;
};
typedef TValue *StkId;   /* index to stack elements */
```

展开后

```
struct lua_TValue {
    union
    {
        struct{
            Value v__;
            int tt__;
        } i;
        double d__;
    }u;
};
```

2.4.2 TString

TString

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

2.4.3 Udata

2.4.4 Upvaldesc

2.4.5 LocVar

2.4.6 Proto

Proto

```
typedef struct Proto {
  CommonHeader;
  TValue *k;  /* constants used by the function */
  Instruction *code;
  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 */
  union Closure *cache;  /* last created closure with this prototype */
  TString  *source;  /* used for debug information */
  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;
  GCObject *gclist;
  lu_byte numparams;  /* number of fixed parameters */
  lu_byte is_vararg;
  lu_byte maxstacksize;  /* maximum stack used by this function */
} Proto;
```

2.4.7 UpVal

UpVal

```
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;
```

2.4.8 CClosure

2.4.9 LClosure

2.4.10 Closure

2.4.11 Tkey

2.4.12 Node

2.4.13 Table

Table

```
typedef struct Table {
  CommonHeader;
  lu_byte flags;  /* 1<<p means tagmethod(p) is not present */
  lu_byte lsizenode;  /* log2 of size of `node' array */
```

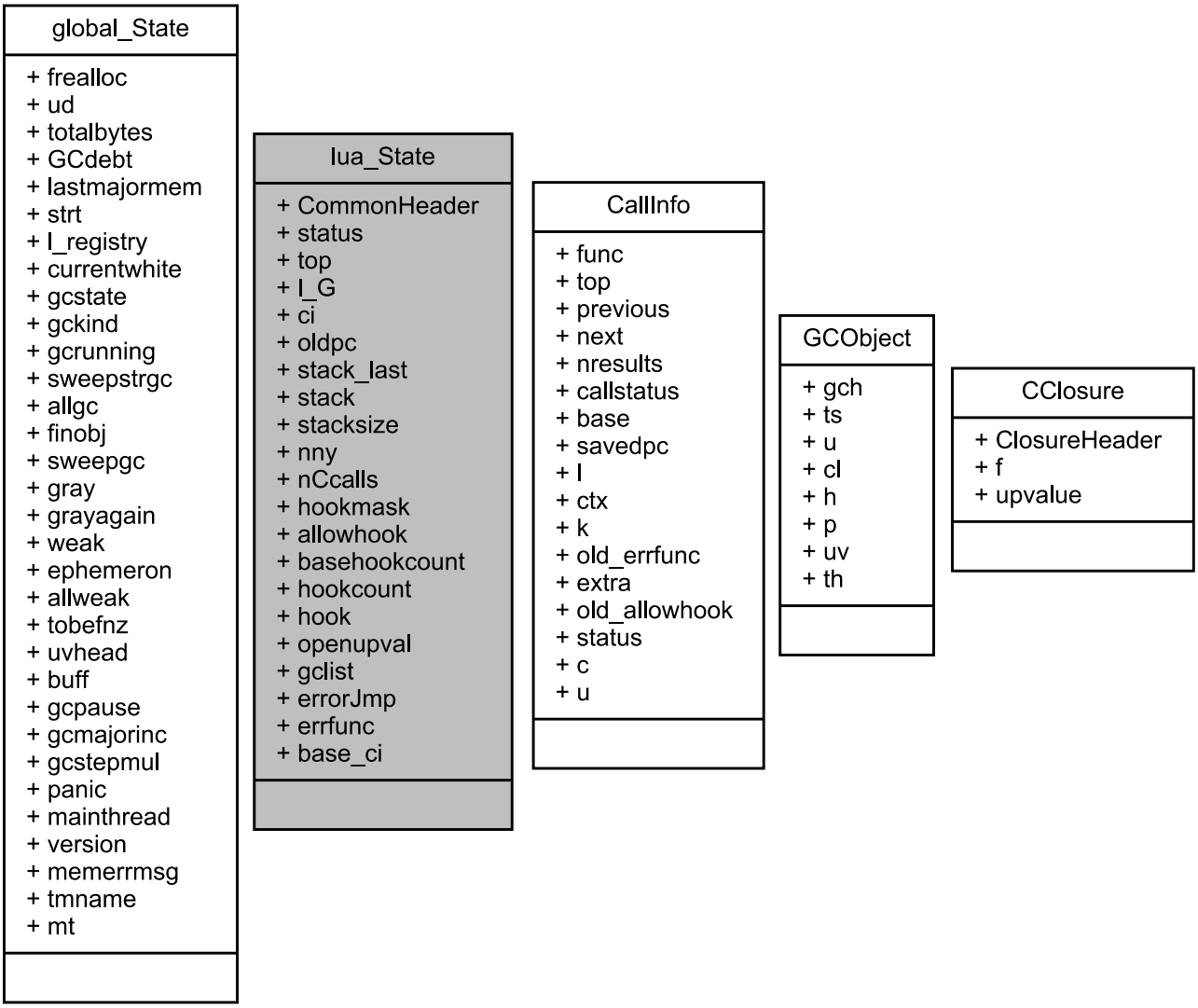
```
  struct Table *metatable;
  TValue *array;  /* array part */
  Node *node;
  Node *lastfree;  /* any free position is before this position */
  GCObject *gclist;
  int sizearray;  /* size of `array' array */
} Table;
```

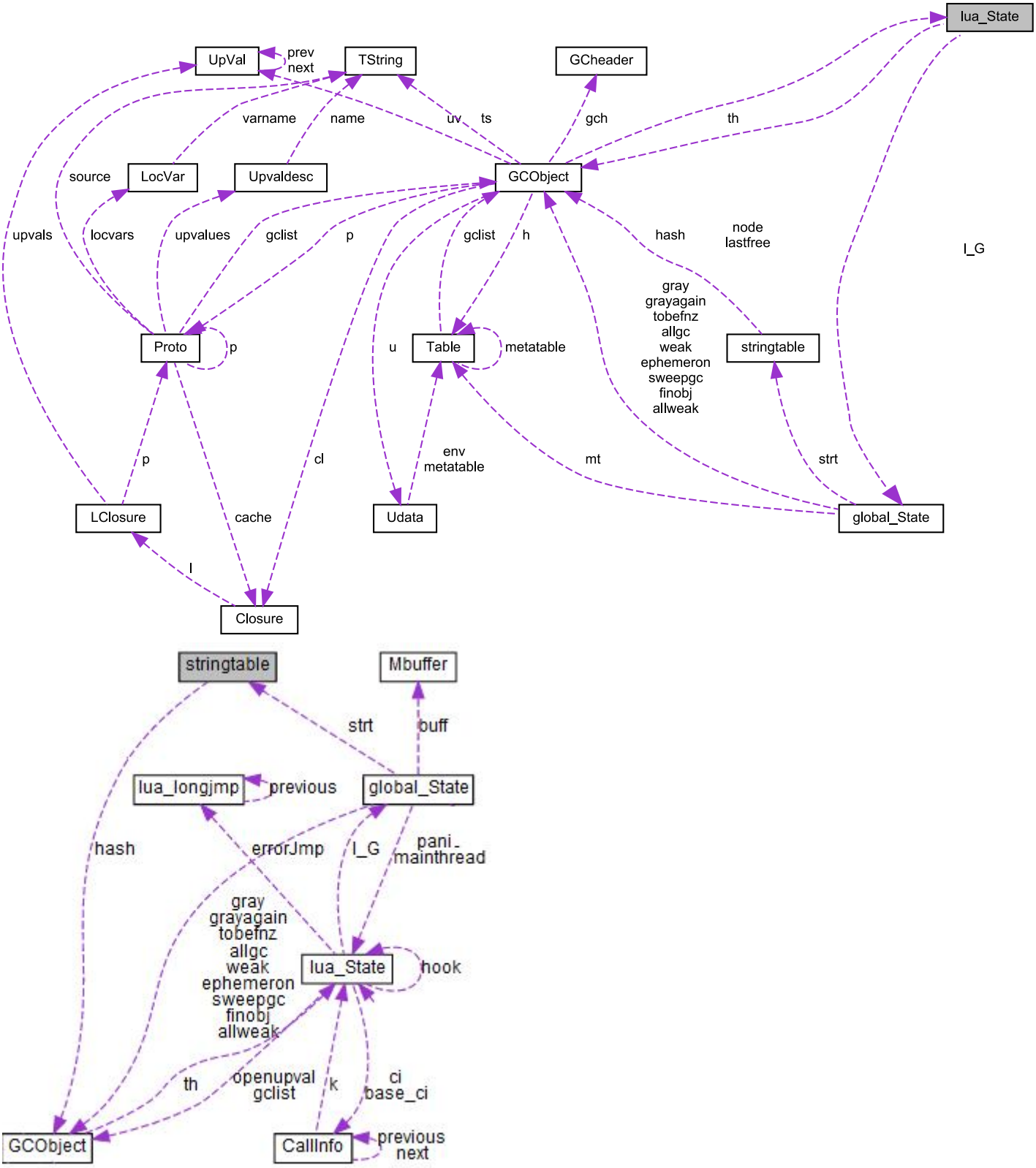
2.4.14 重要宏

```
/* Macros to access values */
#define nvalue(o)   check_exp(ttisnumber(o), num_(o))
#define gcvalue(o)  check_exp(iscollectable(o), val_(o).gc)
#define pvalue(o)   check_exp(ttislightuserdata(o), val_(o).p)
#define rawtsvalue(o)   check_exp(ttisstring(o), &val_(o).gc->ts)
#define tsvalue(o)  (&rawtsvalue(o)->tsv)
#define rawuvalue(o)    check_exp(ttisuserdata(o), &val_(o).gc->u)
#define uvalue(o)   (&rawuvalue(o)->uv)
#define clvalue(o)  check_exp(ttisclosure(o), &val_(o).gc->cl)
#define clLvalue(o) check_exp(ttisLclosure(o), &val_(o).gc->cl.l)
#define clCvalue(o) check_exp(ttisCclosure(o), &val_(o).gc->cl.c)
#define fvalue(o)   check_exp(ttislcf(o), val_(o).f)
#define hvalue(o)   check_exp(ttistable(o), &val_(o).gc->h)
#define bvalue(o)   check_exp(ttisboolean(o), val_(o).b)
#define thvalue(o)  check_exp(ttisthread(o), &val_(o).gc->th)
```

以上宏需要展开，待补充。

2.5 Lstate


global_State
+ frealloc
+ ud
+ totalbytes
+ GCdebt
+ lastmajormem
+ strt
+ l_registry
+ currentwhite
+ gcstate
+ gckind
+ gcrunning
+ sweepstrgc
+ allgc
+ finobj
+ sweepgc
+ gray
+ grayagain
+ weak
+ ephemeron
+ allweak
+ tobefnz
+ uvhead
+ buff
+ gcpause
+ gcmajorinc
+ gcstepmul
+ panic
+ mainthread
+ version
+ memerrmsg
+ tmname
+ mt
lua_State
+ CommonHeader
+ status
+ top
+ l_G
+ ci
+ oldpc
+ stack_last
+ stack
+ stacksize
+ nny
+ nCcalls
+ hookmask
+ allowhook
+ basehookcount
+ hookcount
+ hook
+ openupval
+ gclist
+ errorJmp
+ errfunc
+ base_ci
CallInfo
+ func
+ top
+ previous
+ next
+ nresults
+ callstatus
+ base
+ savedpc
+ l
+ ctx
+ k
+ old_errfunc
+ extra
+ old_allowhook
+ status
+ c
+ u
GCObject
+ gch
+ ts
+ u
+ cl
+ h
+ p
+ uv
+ th
CClosure
+ ClosureHeader
+ f
+ upvalue



2.5.1 不公开结构 LG

```
typedef struct LG {
  LX l;
  global_State g;
} LG;
```

2.5.2 CallInfo 结构

```
/*
** information about a call
*/
typedef struct CallInfo {
  StkId func;  /* function index in the stack */
  StkId top;  /* top for this function */
  struct CallInfo *previous, *next;  /* dynamic call link */
  short nresults;  /* expected number of results from this function */
  lu_byte callstatus;
  union {
    struct {  /* only for Lua functions */
      StkId base;  /* base for this function */
      const Instruction *savedpc;
    } l;
    struct {  /* only for C functions */
      int ctx;  /* context info. in case of yields */
      lua_CFunction k;  /* continuation in case of yields */
      ptrdiff_t old_errfunc;
      ptrdiff_t extra;
      lu_byte old_allowhook;
      lu_byte status;
    } c;
  } u;
} CallInfo;
```

2.5.3 global_State 结构

global_State

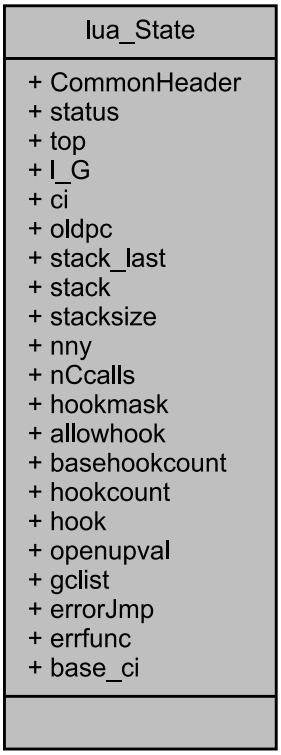
lua_Alloc	frealloc	function to reallocate memory
void *	ud	auxiliary data to `frealloc'

lu_mem	totalbytes	number of bytes currently allocated GCdebt
l_mem	GCdebt	bytes allocated not yet compensated by the collector
lu_mem	lastmajormem	memory in use after last major collection
stringtable	strt	hash table for strings
TValue	l_registry	TValue l_registry
lu_byte	currentwhite	lu_byte currentwhite
lu_byte	gcstate	state of garbage collector
lu_byte	gckind	kind of GC running
lu_byte	gcrunning	true if GC is running
int	sweepstrgc	position of sweep in `strt'
GCObject *	allgc	list of all collectable objects
GCObject *	finobj	list of collectable objects with finalizers
GCObject **	sweepgc	current position of sweep
GCObject *	gray	list of gray objects
GCObject *	grayagain	list of objects to be traversed atomically
GCObject *	weak	list of tables with weak values
GCObject *	ephemeron	list of ephemeron tables (weak keys)
GCObject *	allweak	list of all weak tables
GCObject *	tobefnz	list of userdata to be GC
UpVal	uvhead	head of double linked list of all open upvalues
Mbuffer	buff	temporary buffer for string concatenation
int	gcpause	size of pause between successive GCs
int	gcmajorinc	how much to wait for a major GC (only in gen. mode)
int	gcstepmul	GC `granularity'
lua_CFunction	panic	to be called in unprotected errors
struct lua_State *	mainthread	struct lua_State *mainthread;
const lua_Number *	version	pointer to version number
TString *	memerrmsg	memory error message
TString *	tmname [TM_N]	array with tag method names
struct Table *	mt [9]	metatables for basic types

2.5.4 lua_State 结构

lua_State

	Type	标识符	Remark
HEAD	GCObject *	next	
	lu_byte	tt	
	lu_byte	marked	
	lu_byte	status	
	StkId	top	first free slot in the stack
	global_State *	l_G	
	CallInfo *	ci	call info for current function
	const Instruction *	oldpc	
	StkId	stack_last	last free slot in the stack
	StkId	stack	stack base
	int	stacksize	
	unsigned short	nny	
	unsigned short	nCcalls	
	lu_byte	hookmask	
	lu_byte	allowhook	
	int	basehookcount	
	int	hookcount	
	lua_Hook	hook	
	GCObject *	openupval	list of open upvalues in this stack
	GCObject *	gclist	
	struct lua_longjmp *	errorJmp	
	ptrdiff_t	errfunc	
	CallInfo	base_ci	CallInfo for first level (C calling Lua)



union GCObject

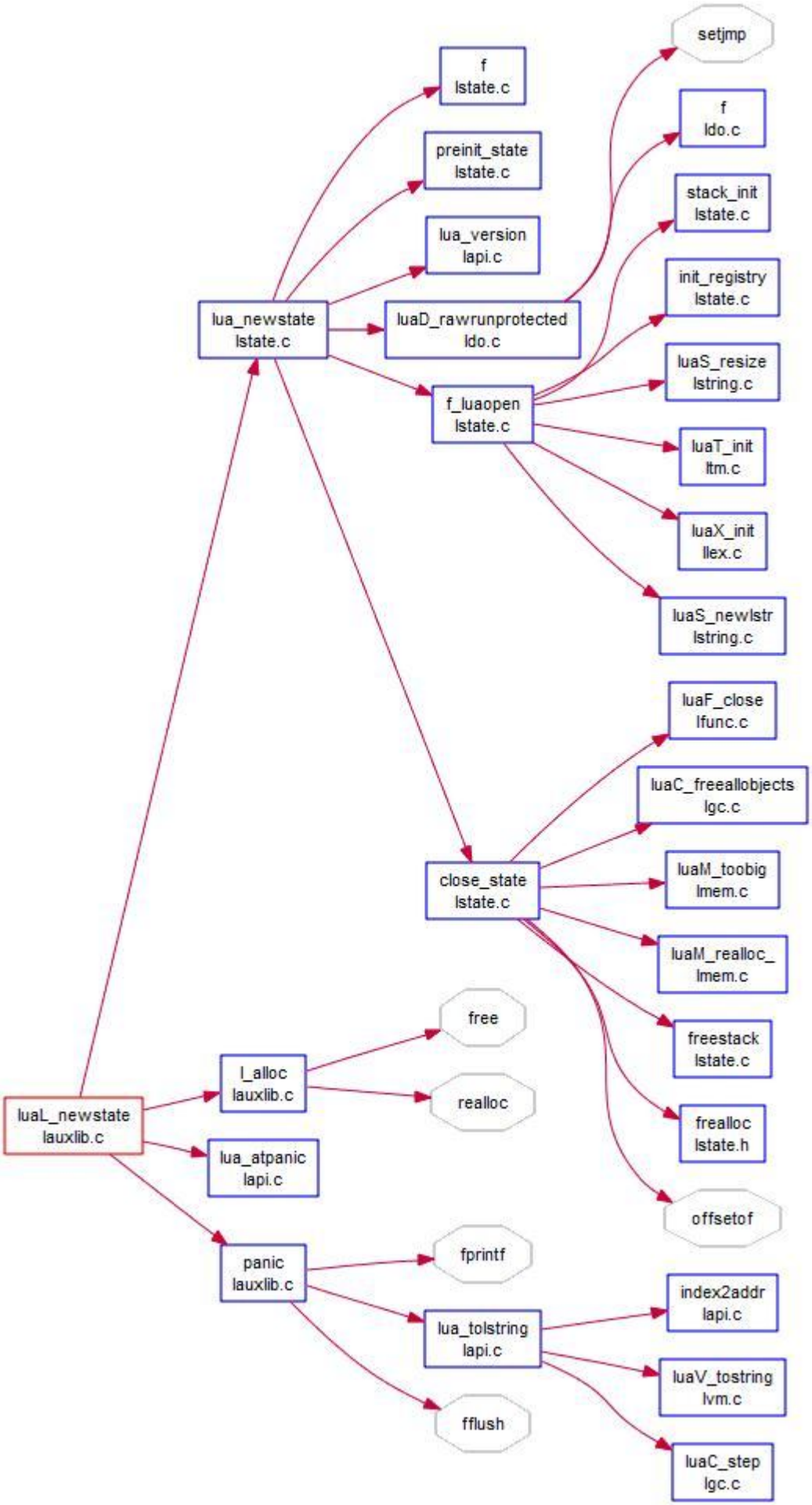
2.5.5 lua_newstate

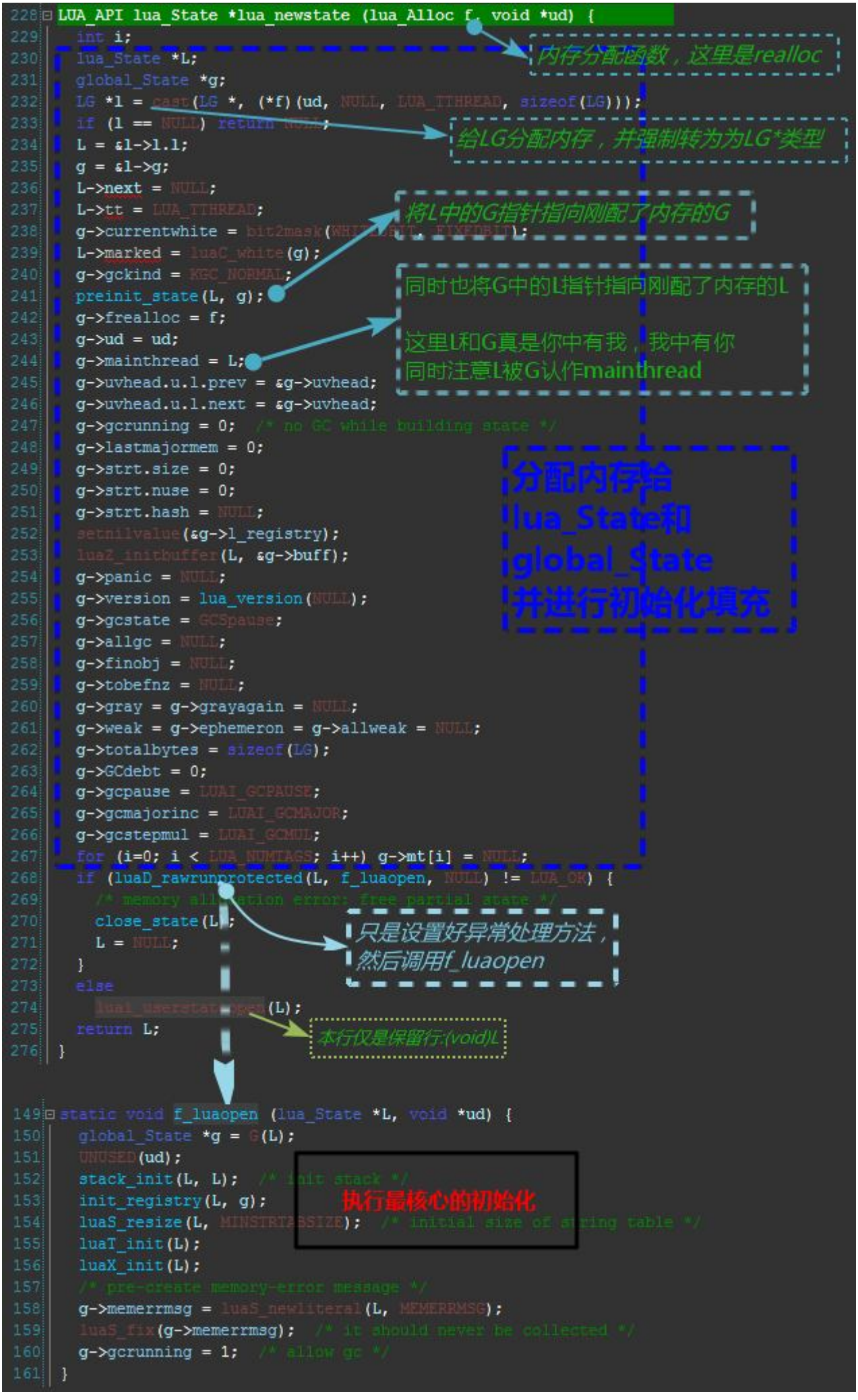
声明在 lua.h 中。

```
LUA_API lua_State *lua_newstate (lua_Alloc f, void *ud) {
  int i;
  lua_State *L;
  global_State *g;
  LG *l = cast(LG *, (*f)(ud, NULL, LUA_TTHREAD, sizeof(LG)));
  if (l == NULL) return NULL;
  L = &l->l.l;
  g = &l->g;
  L->next = NULL;
  L->tt = LUA_TTHREAD;
  g->currentwhite = bit2mask(WHITE0BIT, FIXEDBIT);
  L->marked = luaC_white(g);
```

```
  g->gckind = KGC_NORMAL;
  preinit_state(L, g);
  g->frealloc = f;
  g->ud = ud;
  g->mainthread = L;
  g->uvhead.u.l.prev = &g->uvhead;
  g->uvhead.u.l.next = &g->uvhead;
  g->gcrunning = 0;  /* no GC while building state */
  g->lastmajormem = 0;
  g->strt.size = 0;
  g->strt.nuse = 0;
  g->strt.hash = NULL;
  setnilvalue(&g->l_registry);
  luaZ_initbuffer(L, &g->buff);
  g->panic = NULL;
  g->version = lua_version(NULL);
  g->gcstate = GCSpause;
  g->allgc = NULL;
  g->finobj = NULL;
  g->tobefnz = NULL;
  g->gray = g->grayagain = NULL;
  g->weak = g->ephemeron = g->allweak = NULL;
  g->totalbytes = sizeof(LG);
  g->GCdebt = 0;
  g->gcpause = LUAI_GCPAUSE;
  g->gcmajorinc = LUAI_GCMAJOR;
  g->gcstepmul = LUAI_GCMUL;
  for (i=0; i < LUA_NUMTAGS; i++) g->mt[i] = NULL;
  if (luaD_rawrunprotected(L, f_luaopen, NULL) != LUA_OK) {
    /* memory allocation error: free partial state */
    close_state(L);
    L = NULL;
  }
  else
    luai_userstateopen(L);
  return L;
}
```

调用关系图:


luaL_newstate
lauxlib.c
l_alloc
lauxlib.c
free
realloc
lua_atpanic
lapi.c
panic
lauxlib.c
fprintf
fflush
lua_newstate
lstate.c
f
lstate.c
preinit_state
lstate.c
lua_version
lapi.c
luaD_rawrunprotected
ldo.c
setjmp
f
ldo.c
f_luaopen
lstate.c
stack_init
lstate.c
init_registry
lstate.c
luaS_resize
lstring.c
luaT_init
ltm.c
luaX_init
llex.c
luaS_newlstr
lstring.c
close_state
lstate.c
luaF_close
lfunc.c
luaC_freeallobjects
lgc.c
luaM_toobig
lmem.c
luaM_realloc_
lmem.c
freestack
lstate.c
frealloc
lstate.h
offsetof
lua_tolstring
lapi.c
index2addr
lapi.c
luaV_tostring
lvm.c
luaC_step
lgc.c


内存分配函数，这里是realloc
给LG分配内存，并强制转为为LG*类型
将L中的G指针指向刚配了内存的G
同时也将G中的L指针指向刚配了内存的L
这里L和G真是你中有我，我中有你
同时注意L被G认作mainthread
分配内存给
lua_State和
global_State
并进行初始化填充
只是设置好异常处理方法，
然后调用f_luaopen
本行仅是保留行:(void)L
执行最核心的初始化

```c
static void f_luaopen (lua_State *L, void *ud) {
  global_State *g = G(L);
  UNUSED(ud);
  stack_init(L, L);  /* init stack */
  init_registry(L, g);
  luaS_resize(L, MINSTRTABSIZE);  /* initial size of string table */
  luaT_init(L);
  luaX_init(L);
  /* pre-create memory-error message */
  g->memerrmsg = luaS_newliteral(L, MEMERRMSG);
  luaS_fix(g->memerrmsg);  /* it should never be collected */
  g->gcrunning = 1;  /* allow gc */
}

static void stack_init (lua_State *L1, lua_State *L) {
  int i; CallInfo *ci;
  /* initialize stack array */
  L1->stack = luaM_newvector(L, BASIC_STACK_SIZE, TValue);
  L1->stacksize = BASIC_STACK_SIZE;
  for (i = 0; i < BASIC_STACK_SIZE; i++)
    setnilvalue(L1->stack + i);  /* erase new stack */
  L1->top = L1->stack;
  L1->stack_last = L1->stack + L1->stacksize - EXTRA_STACK;
  /* initialize first ci */
  ci = &L1->base_ci;
  ci->next = ci->previous = NULL;
  ci->callstatus = 0;
  ci->func = L1->top;
  setnilvalue(L1->top++);  /* 'function' entry for this 'ci' */
  ci->top = L1->top + LUA_MINSTACK;
  L1->ci = ci;
}

static void init_registry (lua_State *L, global_State *g) {
  TValue mt;
  /* create registry */
  Table *registry = luaH_new(L);
  sethvalue(L, &g->l_registry, registry);
  luaH_resize(L, registry, LUA_RIDX_LAST, 0);
  /* registry[LUA_RIDX_MAINTHREAD] = L */
  setthvalue(L, &mt, L);
  luaH_setint(L, registry, LUA_RIDX_MAINTHREAD, &mt);
  /* registry[LUA_RIDX_GLOBALS] = table of globals */
  sethvalue(L, &mt, luaH_new(L));
  luaH_setint(L, registry, LUA_RIDX_GLOBALS, &mt);
}

void luaT_init (lua_State *L) {
  static const char *const luaT_eventname[] = {  /* ORDER TM */
    "__index", "__newindex",
    "__gc", "__mode", "__len", "__eq",
    "__add", "__sub", "__mul", "__div", "__mod",
    "__pow", "__unm", "__lt", "__le",
    "__concat", "__call"
  };
  int i;
  for (i=0; i<TM_N; i++) {
    G(L)->tmname[i] = luaS_new(L, luaT_eventname[i]);
    luaS_fix(G(L)->tmname[i]);  /* never collect these names */
  }
}
```

2.6 Llex

Lua 5.1.3 源代码分析之词法分析
http://xenyinzen.wordpress.com/2009/12/09/lua-5-1-3%E6%BA%90%E4%BB%A3%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B%E8%AF%8D%E6%B3%95%E5%88%86%E6%9E%90/

1.1 Lopcodes

字节码指令格式与 opcode 定义。本模块仅用于基础结构描述，不涉及字节码逻辑处理操作。

1.1.1 四种指令格式

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
OP(6bits)							A(8bits)								B(9bits)									C							
OP							A								Bx(18bits)															
OP							A								sBx															
OP							Ax																							

```
    `A' : 8 bits
    `B' : 9 bits
    `C' : 9 bits
    'Ax' : 26 bits ('A', 'B', and 'C' together)
    `Bx' : 18 bits (`B' and `C' together)
    `sBx' : signed Bx
```

简而言之其实就是指令+操作数。指令长度固定是 6 位，而操作数就是 A、B、C 的组合。sBx 只是有符号的 Bx。

1.1.2 指令集

OpCode 枚举定义了所有指令集，注释有指令集的描述。共 40 条指令。

```
/*
** grep "ORDER OP" if you change these enums
*/

typedef enum {
/*----------------------------------------------------------------------
name        args    description
------------------------------------------------------------------------*/
OP_MOVE,/*  A B R(A) := R(B)                    */
```

```
OP_LOADK,/* A Bx    R(A) := Kst(Bx)                 */
OP_LOADKX,/*    A   R(A) := Kst(extra arg)              */
OP_LOADBOOL,/*  A B C   R(A) := (Bool)B; if (C) pc++           */
OP_LOADNIL,/*   A B R(A), R(A+1), ..., R(A+B) := nil       */
OP_GETUPVAL,/*  A B R(A) := UpValue[B]              */
OP_GETTABUP,/*  A B C   R(A) := UpValue[B][RK(C)]          */
OP_GETTABLE,/*  A B C   R(A) := R(B)[RK(C)]             */
OP_SETTABUP,/*  A B C   UpValue[A][RK(B)] := RK(C)         */
OP_SETUPVAL,/*  A B UpValue[B] := R(A)              */
OP_SETTABLE,/*  A B C   R(A)[RK(B)] := RK(C)               */
OP_NEWTABLE,/*  A B C   R(A) := {} (size = B,C)            */
OP_SELF,/*  A B C   R(A+1) := R(B); R(A) := R(B)[RK(C)]       */
OP_ADD,/*   A B C   R(A) := RK(B) + RK(C)              */
OP_SUB,/*   A B C   R(A) := RK(B) - RK(C)              */
OP_MUL,/*   A B C   R(A) := RK(B) * RK(C)              */
OP_DIV,/*   A B C   R(A) := RK(B) / RK(C)              */
OP_MOD,/*   A B C   R(A) := RK(B) % RK(C)              */
OP_POW,/*   A B C   R(A) := RK(B) ^ RK(C)              */
OP_UNM,/*   A B R(A) := -R(B)                  */
OP_NOT,/*   A B R(A) := not R(B)                */
OP_LEN,/*   A B R(A) := length of R(B)              */
OP_CONCAT,/*    A B C   R(A) := R(B).. ... ..R(C)          */
OP_JMP,/*   A sBx   pc+=sBx; if (A) close all upvalues >= R(A) + 1 */
OP_EQ,/*    A B C   if ((RK(B) == RK(C)) ~= A) then pc++       */
OP_LT,/*    A B C   if ((RK(B) <  RK(C)) ~= A) then pc++       */
OP_LE,/*    A B C   if ((RK(B) <= RK(C)) ~= A) then pc++       */
OP_TEST,/*  A C if not (R(A) <=> C) then pc++           */
OP_TESTSET,/*   A B C   if (R(B) <=> C) then R(A) := R(B) else pc++    */
OP_CALL,/*  A B C   R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1)) */
OP_TAILCALL,/*  A B C   return R(A)(R(A+1), ... ,R(A+B-1))     */
OP_RETURN,/*    A B return R(A), ... ,R(A+B-2)  (see note)  */
OP_FORLOOP,/*   A sBx   R(A)+=R(A+2);
            if R(A) <?= R(A+1) then { pc+=sBx; R(A+3)=R(A) }*/
OP_FORPREP,/*   A sBx   R(A)-=R(A+2); pc+=sBx               */
OP_TFORCALL,/*  A C R(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2)); */
OP_TFORLOOP,/*  A sBx   if R(A+1) ~= nil then { R(A)=R(A+1); pc += sBx }*/
OP_SETLIST,/*   A B C   R(A)[(C-1)*FPF+i] := R(A+i), 1 <= i <= B   */
OP_CLOSURE,/*   A Bx    R(A) := closure(KPROTO[Bx])         */
OP_VARARG,/*    A B R(A), R(A+1), ..., R(A+B-2) = vararg       */
OP_EXTRAARG/*   Ax  extra (larger) argument for previous opcode    */
} OpCode;
```

```
MOVE        A B     R(A) := R(B)
LOADK       A Bx    R(A) := K(Bx)
LOADBOOL    A B C   R(A) := (Bool)B; if (C) PC++
LOADNIL     A B     R(A) := ... := R(B) := nil
GETUPVAL    A B     R(A) := U[B]
GETGLOBAL   A Bx    R(A) := G[K(Bx)]
GETTABLE    A B C   R(A) := R(B)[RK(C)]
SETGLOBAL   A Bx    G[K(Bx)] := R(A)
SETUPVAL    A B     U[B] := R(A)
SETTABLE    A B C   R(A)[RK(B)] := RK(C)
NEWTABLE    A B C   R(A) := {} (size = B,C)

SELF        A B C   R(A+1) := R(B); R(A) := R(B)[RK(C)]
ADD         A B C   R(A) := RK(B) + RK(C)
SUB         A B C   R(A) := RK(B) - RK(C)
MUL         A B C   R(A) := RK(B) * RK(C)
DIV         A B C   R(A) := RK(B) / RK(C)
POW         A B C   R(A) := RK(B) ^ RK(C)
UNM         A B     R(A) := -R(B)
NOT         A B     R(A) := not R(B)
CONCAT      A B C   R(A) := R(B) .. ... .. R(C)

JMP         sBx     PC += sBx
EQ          A B C   if ((RK(B) == RK(C)) ~= A) then PC++
LT          A B C   if ((RK(B) <  RK(C)) ~= A) then PC++
LE          A B C   if ((RK(B) <= RK(C)) ~= A) then PC++
TEST        A B C   if (R(B) <=> C) then R(A) := R(B) else PC++
CALL        A B C   R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
TAILCALL    A B C   return R(A)(R(A+1), ... ,R(A+B-1))
RETURN      A B     return R(A), ... ,R(A+B-2)      (see note)
TFORLOOP    A C     R(A+2), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
TFORPREP    A sBx   if type(R(A)) == table then R(A+1):=R(A), R(A):=next;

SETLIST     A Bx    R(A)[Bx-Bx%FPF+i] := R(A+i), 1 <= i <= Bx%FPF+1
SETLISTO    A Bx
CLOSE       A       close stack variables up to R(A)
CLOSURE     A Bx    R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))
```

1.1.3 指令定义

将指令掩码(OpArgMask)和指令模式(OpMode)进行排列组合，即生成了 8 位的指令码，指令码的低 2 位右移动了，所以就可以取高 6 位为有效。

```
/*
** masks for instruction properties. The format is:
** bits 0-1: op mode
** bits 2-3: C arg mode
** bits 4-5: B arg mode
** bit 6: instruction set register A
** bit 7: operator is a test (next instruction must be a jump)
*/

enum OpArgMask {
  OpArgN,  /* argument is not used */
  OpArgU,  /* argument is used */
  OpArgR,  /* argument is a register or a jump offset */
  OpArgK   /* argument is a constant or register/constant */
};

enum OpMode {iABC, iABx, iAsBx, iAx};  /* basic instruction format */

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

LUAI_DDEF 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, OpArgN, OpArgN, iABx)        /* OP_LOADKX */
 ,opmode(0, 1, OpArgU, OpArgU, iABC)        /* OP_LOADBOOL */
 ,opmode(0, 1, OpArgU, OpArgN, iABC)        /* OP_LOADNIL */
 ,opmode(0, 1, OpArgU, OpArgN, iABC)        /* OP_GETUPVAL */
 ,opmode(0, 1, OpArgU, OpArgK, iABC)        /* OP_GETTABUP */
 ,opmode(0, 1, OpArgR, OpArgK, iABC)        /* OP_GETTABLE */
 ,opmode(0, 0, OpArgK, OpArgK, iABC)        /* OP_SETTABUP */
 ,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, 0, OpArgN, 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(0, 0, OpArgN, OpArgU, iABC)       /* OP_TFORCALL */
 ,opmode(0, 1, OpArgR, OpArgN, iAsBx)      /* OP_TFORLOOP */
 ,opmode(0, 0, OpArgU, OpArgU, iABC)       /* OP_SETLIST */
 ,opmode(0, 1, OpArgU, OpArgN, iABx)       /* OP_CLOSURE */
 ,opmode(0, 1, OpArgU, OpArgN, iABC)       /* OP_VARARG */
 ,opmode(0, 0, OpArgU, OpArgU, iAx)    /* OP_EXTRAARG */
};
```

1.2 Lvm

此模块时 lua 虚拟机(Lua virtual machine)实现部分。luaV_execute 是主循环，执行所有的指令操作。此处先跳过细节，粗读即可。具体分析在后续的引擎核心章节解释。同时可以参考 A No-Frills Introduction to Lua 5.1 VM Instructions。

Lvm 预处理文件可以参考:
http://code.google.com/p/3dlearn/source/browse/trunk/scripts/luas/lua52/lvm2.i.c
http://code.google.com/p/3dlearn/source/browse/trunk/scripts/luas/lua52/lvm.i.c
其中 lvm.i.c 是完全展开所有预处理宏的文件，lvm2.i.c 只展开部分宏。

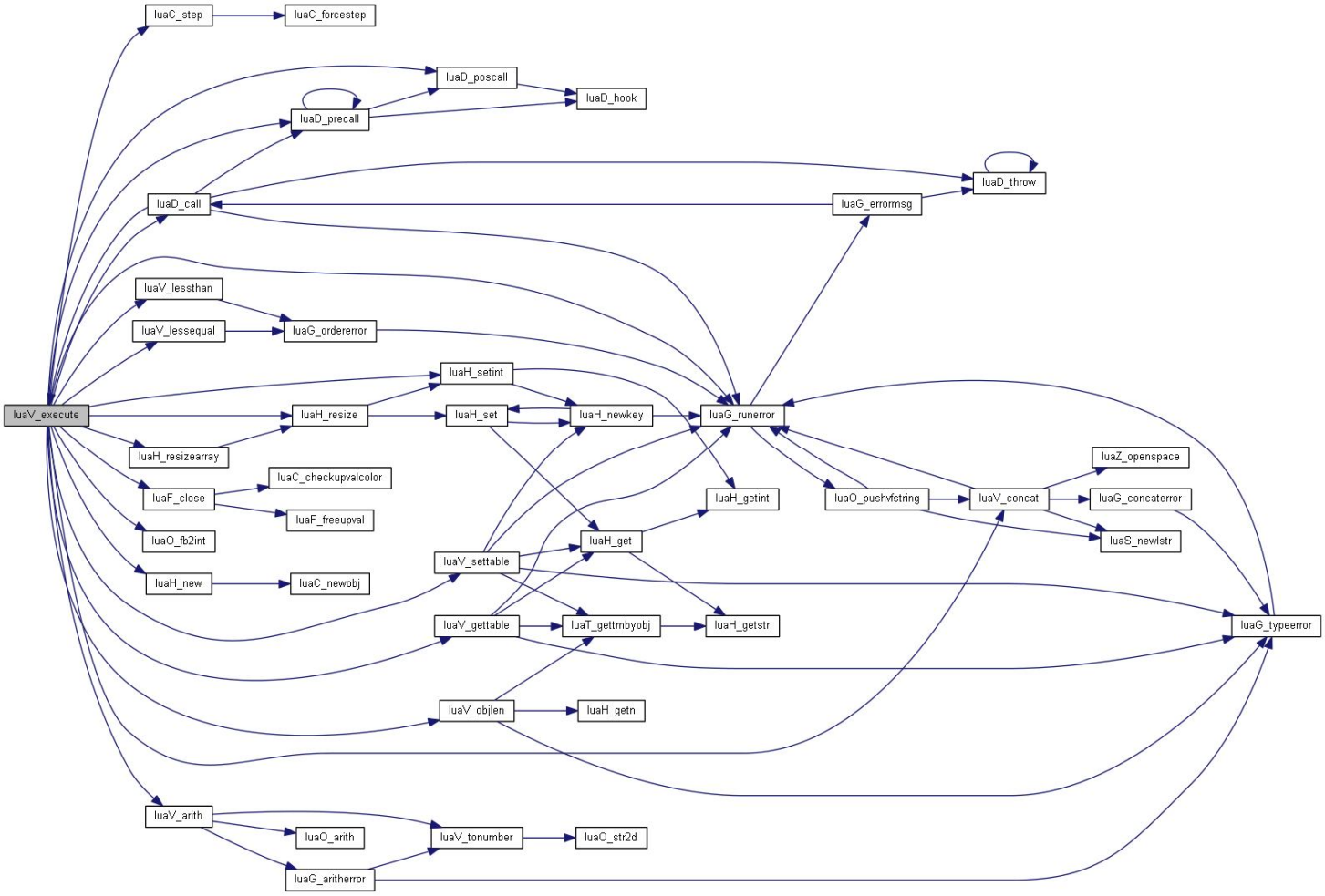
1.2.1 过程

共 12 个公开过程。

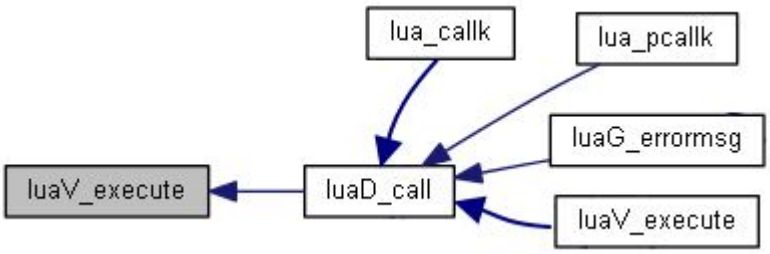
int	luaV_equalobj_ (lua_State *L, const TValue *t1, const TValue *t2)
int	luaV_lessthan (lua_State *L, const TValue *l, const TValue *r)
int	luaV_lessequal (lua_State *L, const TValue *l, const TValue *r)
const TValue *	luaV_tonumber (const TValue *obj, TValue *n)
int	luaV_tostring (lua_State *L, StkId obj)

void	luaV_gettable (lua_State *L, const TValue *t, TValue *key, StkId val)
void	luaV_settable (lua_State *L, const TValue *t, TValue *key, StkId val)
void	luaV_finishOp (lua_State *L)
void	luaV_execute (lua_State *L)
void	luaV_concat (lua_State *L, int total)
void	luaV_arith (lua_State *L, StkId ra const TValue *rb, const TValue *rc, TMS op)
void	luaV_objlen (lua_State *L, StkId ra, const TValue *rb)

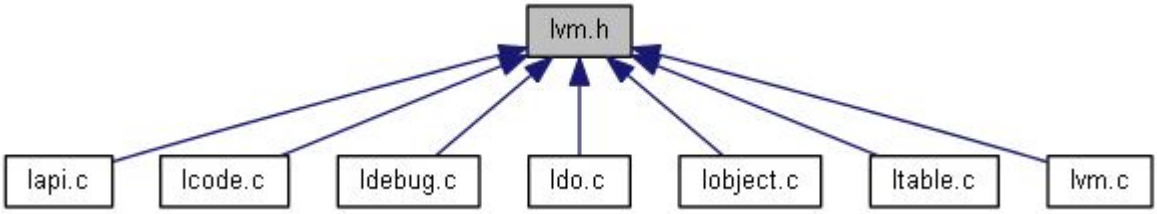
1.2.2 过程调用



被调用关系



文件引用



1.2.3 luaV_execute

可以参考 lvm2.i.c[3]预处理文件，以方便阅读。

```
void luaV_execute (lua_State *L) {
    CallInfo *ci = L->ci;
    LClosure *cl;
    TValue *k;
    StkId base;
newframe:
    ((void)0);
    cl = (&((ci->func)->u.i.v__).gc->cl.l);
    k = cl->p->k;
    base = ci->u.l.base;

    for (;;) {
        Instruction i = *(ci->u.l.savedpc++);
        StkId ra;
        if ((L->hookmask & ((1 << 2) | (1 << 3))) &&
            (--L->hookcount == 0 || L->hookmask & (1 << 2))) {
                { {traceexec(L);}; base = ci->u.l.base; };
        }

        ra = (base+(((int)(((i)>>(0 + 6)) & ((~((~(Instruction)0)<<(8)))<<(0))))));
        ((void)0);
        ((void)0);
        switch((((OpCode)(((i)>>0) & ((~((~(Instruction)0)<<(6)))<<(0)))))) {
      case OP_MOVE:
          {
              const TValue *o2_=base+(int)(((i)>>(((0 + 6) + 8) + 9)) &
((~((~(Instruction)0)<<9))<<0));
              TValue *o1_=(ra); o1_->u = o2_->u;
              ;
```

[3] http://code.google.com/p/3dlearn/source/browse/trunk/scripts/luas/lua52/lvm2.i.c

```
        } break;
    case OP_LOADK:
…
}
```

1.3 Ldo(Stack and Call structure)

调用，栈，异常，协程。粗读。

1.3.1 公开过程

int	luaD_protectedparser (lua_State *L, ZIO *z, const char *name, const char *mode)
void	luaD_hook (lua_State *L, int event, int line)
int	luaD_precall (lua_State *L, StkId func, int nresults)
void	luaD_call (lua_State *L, StkId func, int nResults, int allowyield)
int	luaD_pcall (lua_State *L, Pfunc func, void *u, ptrdiff_t oldtop, ptrdiff_t ef)
int	luaD_poscall (lua_State *L, StkId firstResult)
void	luaD_reallocstack (lua_State *L, int newsize)
void	luaD_growstack (lua_State *L, int n)
void	luaD_shrinkstack (lua_State *L)
void	luaD_throw (lua_State *L, int errcode)
int	luaD_rawrunprotected (lua_State *L, Pfunc f, void *ud)

1.3.2 luaD_pcall

```
int luaD_pcall (lua_State *L, Pfunc func, void *u,
                ptrdiff_t old_top, ptrdiff_t ef) {
  int status;
  CallInfo *old_ci = L->ci;
  lu_byte old_allowhooks = L->allowhook;
  unsigned short old_nny = L->nny;
  ptrdiff_t old_errfunc = L->errfunc;
  L->errfunc = ef;
  status = luaD_rawrunprotected(L, func, u);
  if (status != LUA_OK) {  /* an error occurred? */
    StkId oldtop = restorestack(L, old_top);
    luaF_close(L, oldtop);  /* close possible pending closures */
    seterrorobj(L, status, oldtop);
    L->ci = old_ci;
    L->allowhook = old_allowhooks;
    L->nny = old_nny;
    luaD_shrinkstack(L);
  }
  L->errfunc = old_errfunc;
  return status;
}
```

1.3.3 luaD_protectedparser

```
int luaD_protectedparser (lua_State *L, ZIO *z, const char *name,
```

```
                                const char *mode) {
  struct SParser p;
  int status;
  L->nny++;  /* cannot yield during parsing */
  p.z = z; p.name = name; p.mode = mode;
  p.dyd.actvar.arr = NULL; p.dyd.actvar.size = 0;
  p.dyd.gt.arr = NULL; p.dyd.gt.size = 0;
  p.dyd.label.arr = NULL; p.dyd.label.size = 0;
  luaZ_initbuffer(L, &p.buff);
  status = luaD_pcall(L, f_parser, &p, savestack(L, L->top), L->errfunc);
  luaZ_freebuffer(L, &p.buff);
  luaM_freearray(L, p.dyd.actvar.arr, p.dyd.actvar.size);
  luaM_freearray(L, p.dyd.gt.arr, p.dyd.gt.size);
  luaM_freearray(L, p.dyd.label.arr, p.dyd.label.size);
  L->nny--;
  return status;
}
```

1.4 Lstring

1.4.1 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;
  for (l1=l; l1>=step; l1-=step)  /* compute hash */
    h = h ^ ((h<<5)+(h>>2)+cast(unsigned char, str[l1-1]));
  for (o = G(L)->strt.hash[lmod(h, G(L)->strt.size)];
       o != NULL;
       o = gch(o)->next) {
    TString *ts = rawgco2ts(o);
    if (h == ts->tsv.hash &&
        ts->tsv.len == l &&
        (memcmp(str, getstr(ts), l * sizeof(char)) == 0)) {
      if (isdead(G(L), o))  /* string is dead (but was not collected yet)? */
        changewhite(o);  /* resurrect it */
      return ts;
    }
```

```
  }
  return newlstr(L, str, l, h);  /* not found; create a new string */
}
```

1.4.2 luaS_newudata

```
Udata *luaS_newudata (lua_State *L, size_t s, Table *e) {
  Udata *u;
  if (s > MAX_SIZET - sizeof(Udata))
    luaM_toobig(L);
  u = &luaC_newobj(L, LUA_TUSERDATA, sizeof(Udata) + s, NULL, 0)->u;
  u->uv.len = s;
  u->uv.metatable = NULL;
  u->uv.env = e;
  return u;
}
```

1.5 Ltable(hash tables)

Hash tables and arrays.table api 都是 H 打头，因为他们都是 hash 表。API 第一个参数都是 Table 对象，所以可以看成与状态机没有耦合关系的应用类。

1.6 Ltm(Tag methods)

metamethod handling, reread all of lvm.c now.

Tag 处理\元方法相关，API 用 T 打头。 涉及元方法处理。元方法在状态机初始化的同时被初始化。涉及的原方法在 luaT_init 初始化时被定义好。这些元方法在 lua 中类似一个运算符重载。

1.6.1 luaT_init

元方法的初始化

```
void luaT_init (lua_State *L) {
  static const char *const luaT_eventname[] = {  /* ORDER TM */
    "__index", "__newindex",
    "__gc", "__mode", "__len", "__eq",
    "__add", "__sub", "__mul", "__div", "__mod",
    "__pow", "__unm", "__lt", "__le",
    "__concat", "__call"
  };
  int i;
  for (i=0; i<TM_N; i++) {
    G(L)->tmname[i] = luaS_new(L, luaT_eventname[i]);
    luaS_fix(G(L)->tmname[i]);  /* never collect these names */
  }
}
```

1.6.2 luaT_gettm

取元方法

```
const TValue *luaT_gettm (Table *events, TMS event, TString *ename) {
  const TValue *tm = luaH_getstr(events, ename);
  lua_assert(event <= TM_EQ);
  if (ttisnil(tm)) {  /* no tag method? */
    events->flags |= cast_byte(1u<<event);  /* cache this fact */
    return NULL;
  }
  else return tm;
}
```

1.7 Ldebug

surprise waiting for you. abstract interpretation is used to find object names for tracebacks. does bytecode verification, too.

1.7.1 过程表

void	luaG_typeerror (lua_State *L, const TValue *o, const char *opname)
void	luaG_concaterror (lua_State *L, StkId p1, StkId p2)
void	luaG_aritherror (lua_State *L, const TValue *p1, const TValue *p2)

void	luaG_ordererror (lua_State *L, const TValue *p1, const TValue *p2)
void	luaG_runerror (lua_State *L, const char *fmt,...)
void	luaG_errormsg (lua_State *L)

<需要补充中文注释>

1.8 Lparser\lcode(递归下降分析器)

recursive descent parser, targetting a register-based VM. start from chunk() and work your way through. read the expression parser and the code generator parts last.

递归下降分析器(recursive descent parser)，基于寄存器的虚拟机[4]。

1.8.1 数据结构

struct	expdesc	
struct	Vardesc	
struct	Labeldesc	
struct	Labellist	
struct	Dyndata	
struct	FuncState	

<待补充>

1.9 Lgc(增量\渐进垃圾回收器)

Incremental garbage Collector[5].

<待补充>

[4] 见附录递归下降分析器

[5] 见附录垃圾回收器

2 编码规范与约定

“Lua 使用 CleanC 写的源代码，模块划分清晰，大部分模块被分在不同的.c 文件中实现，以同名的.h 文件描述模块导出的接口。”[6]

2.1 标识符惯用法

Name	Remark
cf	c function
lf	Lua function
ud	light userdata
ni	number input args
no	number output args
k	field index
n	Stack element at index n
n1	Stack element 1 at index n1
n2	Stack element 2 at index n2
value	Lua value
fmt	format string, see fprintf()
data	pointer to raw data
number	Lua number
ar	Pointer to debug structure
L, L1, L2	Lua state
B	Lua buffer
integer	Lua integer
t	lua type
ok	1=success
error	error code, 0=ok

2.2 接口代码约定

Code Conventions

Source: http://lua-users.org/wiki/LuaSource

The prefix of a external symbol indicates the module it comes from:

`luaA_ - lapi.c`

[6] Lua 源码欣赏，云风著.

```
luaB_ - lbaselib.c
luaC_ - lgc.c
luaD_ - ldo.c
luaE_ - lstate.c
luaF_ - lfunc.c
luaG_ - ldebug.c
luaH_ - ltable.c
luaI_ - lauxlib.c
luaK_ - lcode.c
luaL_ - lauxlib.c/h, linit.c (public functions)
luaM_ - lmem.c
luaO_ - lobject.c
luaP_ - lopcodes.c
luaS_ - lstring.c
luaT_ - ltm.c
luaU_ - lundump.c
luaV_ - lvm.c
luaX_ - llex.c
luaY_ - lparser.c
luaZ_ - lzio.c
lua_  - lapi.c/h + luaconf.h, debug.c
luai_ - luaconf.h
luaopen_ - luaconf.h + libraries (lbaselib.c, ldblib.c, liolib.c, lmathlib.c,
                                  loadlib.c, loslib.c, lstrlib.c, ltablib.c)
```

3 基础数据结构

基础数据结构大部分定义在 Lobject 中。本节主要是从 C 角度考虑 lua 数据结构。

3.1 等价 C 类型

本节介绍基础类型 string、array、table、function、userdata。其中 string、array 仍然是 table。

3.1.1 string

3.1.2 array

3.1.2.1 数组操作

数组操作 API

```
LUA_API void  (lua_rawgeti) (lua_State *L, int idx, int n);
LUA_API void  (lua_rawseti) (lua_State *L, int idx, int n);
```

解释:

直接对栈中 idx 处的数组操作，数组下标是 n。get 时将值压栈，set 时将栈顶值赋给数组元素。

理解:

rawgeti 表示 get raw index,不触发元操作，直接操作 t[key]=value。对于涉及 3 个数据的操作，例如这里的 t、key、value，lua 会接触栈顶将其中的 1 个或 2 个元素作为一个隐性的临时数据存放空间。

同时需要注意相近操作

Get 方法

```
LUA_API void  (lua_getglobal) (lua_State *L, const char *var);
LUA_API void  (lua_gettable) (lua_State *L, int idx);
LUA_API void  (lua_getfield) (lua_State *L, int idx, const char *k);
LUA_API void  (lua_rawget) (lua_State *L, int idx);
LUA_API void  (lua_rawgeti) (lua_State *L, int idx, int n);
LUA_API void  (lua_rawgetp) (lua_State *L, int idx, const void *p);
LUA_API void  (lua_createtable) (lua_State *L, int narr, int nrec);
LUA_API void *(lua_newuserdata) (lua_State *L, size_t sz);
LUA_API int   (lua_getmetatable) (lua_State *L, int objindex);
LUA_API void  (lua_getuservalue) (lua_State *L, int idx);
```

Set 方法

```
LUA_API void  (lua_setglobal) (lua_State *L, const char *var);
LUA_API void  (lua_settable) (lua_State *L, int idx);
LUA_API void  (lua_setfield) (lua_State *L, int idx, const char *k);
LUA_API void  (lua_rawset) (lua_State *L, int idx);
LUA_API void  (lua_rawseti) (lua_State *L, int idx, int n);
LUA_API void  (lua_rawsetp) (lua_State *L, int idx, const void *p);
LUA_API int   (lua_setmetatable) (lua_State *L, int objindex);
LUA_API void  (lua_setuservalue) (lua_State *L, int idx);
```

参考 case 章节的 set\get 方法辨析

4 全局变量

4.1 脚本层全局对象

4.1.1 查看全局变量

方法一

```
code
function printTable(t)
    if type(t) == "table" then
        local k ,v
        for k , v in pairs(t) do print("\t"  , k  , v ) end
    end

end



for k , v in pairs(_G) do
    print( type(v) , k  , v )
    printTable(v)
end
```

方法二
官网提供在线运行 demo
http://www.lua.org/cgi-bin/demo

4.1.2 全局变量

1	2	3
_G	write	pcall
_VERSION	ipairs	print
assert	load	rawequal
bit32	loadstring	rawget
arshift	math	rawlen
band	abs	rawset
bnot	acos	select
bor	asin	setmetatable
btest	atan	string
bxor	atan2	byte
extract	ceil	char
lrotate	cos	dump
lshift	cosh	find
replace	deg	format

rrotate	exp	gmatch
rshift	floor	gsub
collectgarbage	fmod	len
coroutine	frexp	lower
create	huge	match
resume	ldexp	rep
running	log	reverse
status	log10	sub
wrap	max	upper
yield	min	table
debug	modf	concat
gethook	pi	insert
getinfo	pow	maxn
getlocal	rad	pack
getmetatable	random	remove
getupvalue	randomseed	sort
getuservalue	sin	unpack
sethook	sinh	tonumber
setlocal	sqrt	tostring
setmetatable	tan	type
setupvalue	tanh	unpack
setuservalue	module	xpcall
traceback	next	
upvalueid	os	
upvaluejoin	clock	
dump	date	
error	difftime	
getmetatable	exit	
io	setlocale	
	time	
	pairs	

4.2 引擎全局对象

此处指 lua 核心实现中的全局对象。

global_State

5 架构

分析引擎架构

Source:
云风，lua 源码赏析.pdf
http://lua-users.org/wiki/LuaSource

5.1 文件结构

代码统计

空行	3,118
类	0
代码行	11,610
注释行	4,053
注释率	0.35
函数	782
总行	18,884

共 59 个代码文件

5.1.1 文件结构

C 文件

1.	lapi.c
2.	lauxlib.c
3.	lbaselib.c
4.	lbitlib.c
5.	lcode.c
6.	lcorolib.c
7.	lctype.c
8.	ldblib.c
9.	ldebug.c
10.	ldo.c
11.	ldump.c
12.	lfunc.c
13.	lgc.c
14.	linit.c
15.	liolib.c
16.	llex.c
17.	lmathlib.c
18.	lmem.c
19.	loadlib.c

20.	lobject.c
21.	lopcodes.c
22.	loslib.c
23.	lparser.c
24.	lstate.c
25.	lstring.c
26.	lstrlib.c
27.	ltable.c
28.	ltablib.c
29.	ltm.c
30.	lua.c
31.	lua.hpp
32.	luac.c
33.	lundump.c
34.	lvm.c
35.	lzio.c

其中核心文件 20 个，非核心文件 15 个。

5.1.1.1 分类

1. utility functionality
2. basic data types
3. parsing and code generation
4. bytecodes
5. standard libraries
6. C API

5.1.1.2 分类表

C 接口	lapi.c	C 语言接口
	linit.c	内嵌库的初始化
	lctype.c	C 标准库中 ctype 相关实现
实用功能	ldebug.c	debug 接口
	lgc.c	垃圾回收
	lmem.c	内存管理接口
	lzio.c	输入流接口
	lauxlib.c	库编写用到的辅助函数库
执行字节码	lopcodes.c	虚拟机的字节码定义
	ltm.c	元方法
	lvm.c	虚拟机
	ldo.c	函数调用以及栈管理
数据结构	lobject.c	对象操作的一些函数
	lstate.c	全局状态机
	lstring.c	字符串池

	ltable.c	表类型的相关操作
	lfunc.c	函数原型及闭包管理
脚本解析与字节码生成	lcode.c	代码生成器
	ldump.c	序列化预编译的扌扵扡字节码
	llex.c	词法分析器
	lparser.c	解析器
	lundump.c	还原预编译的字节码
标准库	lbaselib.c	基础库
	lstrlib.c	字符串库
	ltablib.c	表处理库
	lmathlib.c	数学库
	loslib.c	执打库
	liolib.c	io 库
	ldblib.c	debug 库
	loadlib.c	动态扩展库管理
	lbitlib.c	位操作库
	lcorolib.c	协程库
	lua.c	解释器
	luac.c	字节码编译器

这里也有个略有区别的文件划分
http://stevedonovan.github.com/lua-5.1.4/

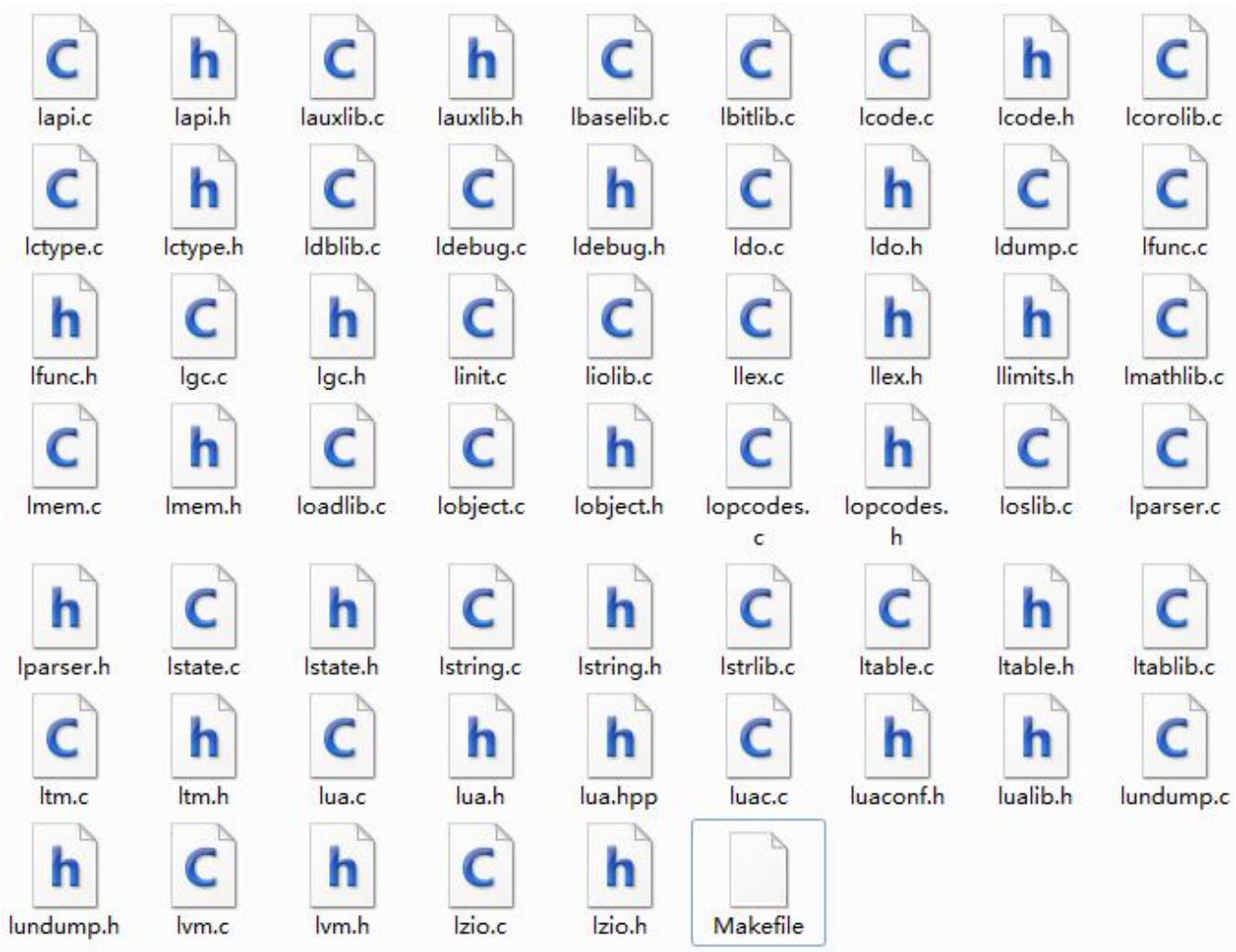
5.1.1.3 核心文件
核心文件是在 c 文件最开始定义了 LUA_CORE 的文件，共 22 个。

Index	File	宏定义所在行
1	ldo.c	13
2	lopcodes.c	8
3	ltm.c	11
4	lvm.c	13
5	lapi.c	12
6	lctype.c	8
7	lfunc.c	11
8	lobject.c	13
9	lstate.c	11
10	lstring.c	11
11	ltable.c	24
12	lcode.c	11
13	ldump.c	10
14	llex.c	12
15	lparser.c	11
16	lundump.c	10
17	ldebug.c	14

18	lgc.c	10
19	lmem.c	11
20	lzio.c	11

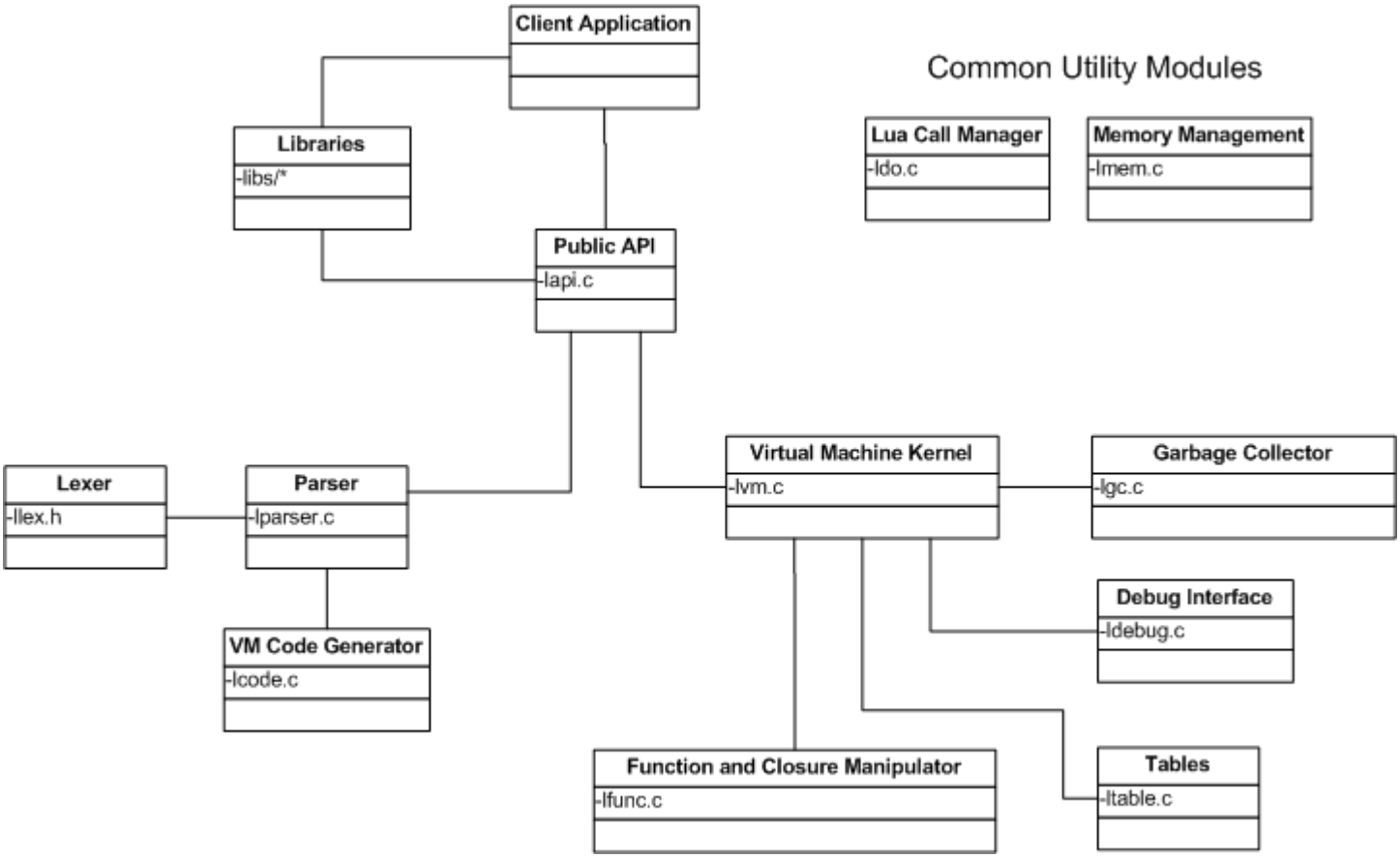
5.2 文件点评

本节主要从大体上点评下



5.3 模块划分

Source: The Lua Architecture,Module Decomposition



<需补充引用资料>

6 引擎核心

此处指 lua 实现的状态机-虚拟机-数据流机制的核心逻辑。

6.1 缘起缘灭[7]

所谓旁观者清，在了解引擎核心之前，似乎茫然无从入手，只能找个最相近的参考模型去想象。CPU 似乎是个不错的选择，这里就想当然找了个映射关系:

1.	LVM	CPU
2.	脚本	C/C++
3.	字节码	机器码
4.	基本数据类型	机器内存数据

既然如此，问题就来了。

数据放哪里？

[7] 本节纯属扯淡

答:引擎(lua 核心)维护一个全局的数据空间，它自己在任何时候都知道如何操纵一个标识符，不管这个标识符是表、函数、简单类型或其他对象。

脚本如何编译成字节码？
答:不知道。很多人都不知道 C 如何编译成汇编，照样玩得很 happy。

字节码如何操纵 C/C++对象(数据、函数调用等)?
答:不知道。将内存数据地址强制转换后传给引擎，引擎应该可以直接或间接操纵这个内存地址中的数据。但是操纵 C/C++函数就不知道如何幻想了，引擎至少有一个机制，这个机制可以让引擎调用一个“天使函数”，这个函数遵从引擎的使用机制(好比天使，至少有翅膀能飞，才能说明是上帝的使者,可以听从上帝的命令)，同事必须是个 C/C++函数的样子(好比天使，有个人行，可以同人交流)。

```
static int average(lua_State * L)
{
    /* get number of arguments */
    int n = lua_gettop(L);
    int sum=0;
    /* loop through each argument */
    for (int i = 1; i <= n; i++)
    {
        /* total the arguments */
        sum += lua_tonumber(L, i);
    }

    lua_pushnumber(L, sum / n);
    /* return the number of results */
    printf("c average called. [ok]\n");
    return 1;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int error;
    // 创建 Lua 接口指针
    lua_State* L = lua_open();
    // 加载 Lua 基本库
    luaopen_base(L);
    // 加载 Lua 通用扩展库
    luaL_openlibs(L);
    lua_register(L, "average", average);
    /* load the script */
    error = luaL_dofile(L, "luacallc.lua");
    lua_close( L);
    return 0;
}
```

如此这般，可以认为 lua_register 是天使函数吧？似乎不行。average 如何被调用依旧不明。拍个快照看看

```
demo_d.exe!average(lua_State * L=0x00394c40)
lua520_d.dll!luaD_precall(lua_State * L=0x00394c40, lua_TValue * func=0x00394e08, int nresults=0x00000001)
lua520_d.dll!luaV_execute(lua_State * L=0x00394c40)
lua520_d.dll!luaD_call (lua_State * L=0x00394c40, lua_TValue * func=0x00394e00, int nResults=0xffffffff, int
allowyield=0x00000000)
lua520_d.dll!f_call(lua_State * L=0x00394c40, void * ud=0x0015fa1c)
lua520_d.dll!luaD_rawrunprotected(lua_State * L=0x00394c40, void (lua_State *, void *)* f=0x68876df0, void
* ud=0x0015fa1c)
lua520_d.dll!luaD_pcall(lua_State * L=0x00394c40, void (lua_State *, void *)* func=0x68876df0, void *
u=0x0015fa1c, int old_top=0x00000010, int ef=0x00000000)
lua520_d.dll!lua_pcallk(lua_State * L=0x00394c40, int nargs=0x00000000, int nresults=0xffffffff, int
errfunc=0x00000000, int ctx=0x00000000, int (lua_State *)* k=0x00000000)
demo_d.exe!main(int argc=0x00000001, char * * argv=0x00394bc0)
```

	demo_d.exe	average	
	lua520_d.dll	luaD_precall	
	lua520_d.dll	luaV_execute	
	lua520_d.dll	luaD_call	
	lua520_d.dll	f_call	
	lua520_d.dll	luaD_rawrunprotected	
	lua520_d.dll	luaD_pcall	
	lua520_d.dll	lua_pcallk	
	demo_d.exe	main	

如此这般，luaD_precall[8]就是这最接近神的人

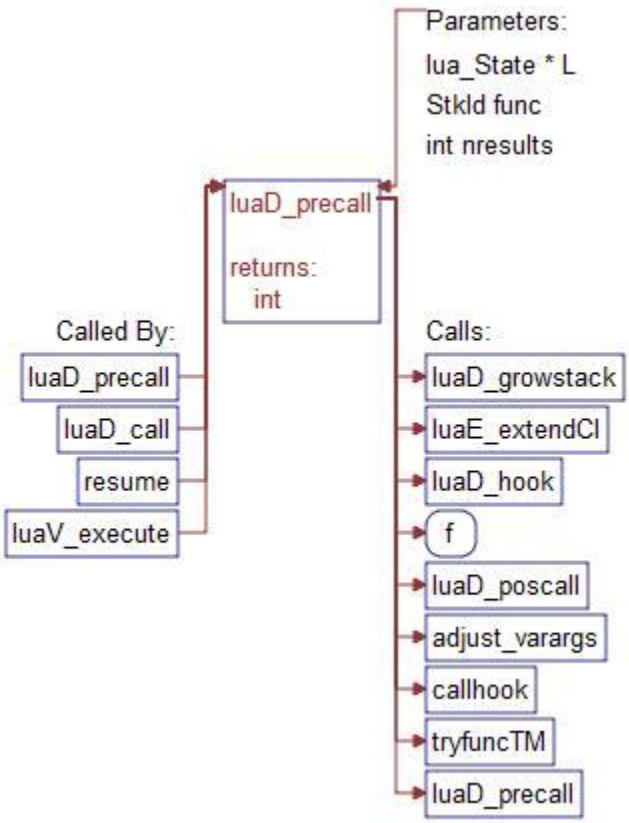
luaD_precall

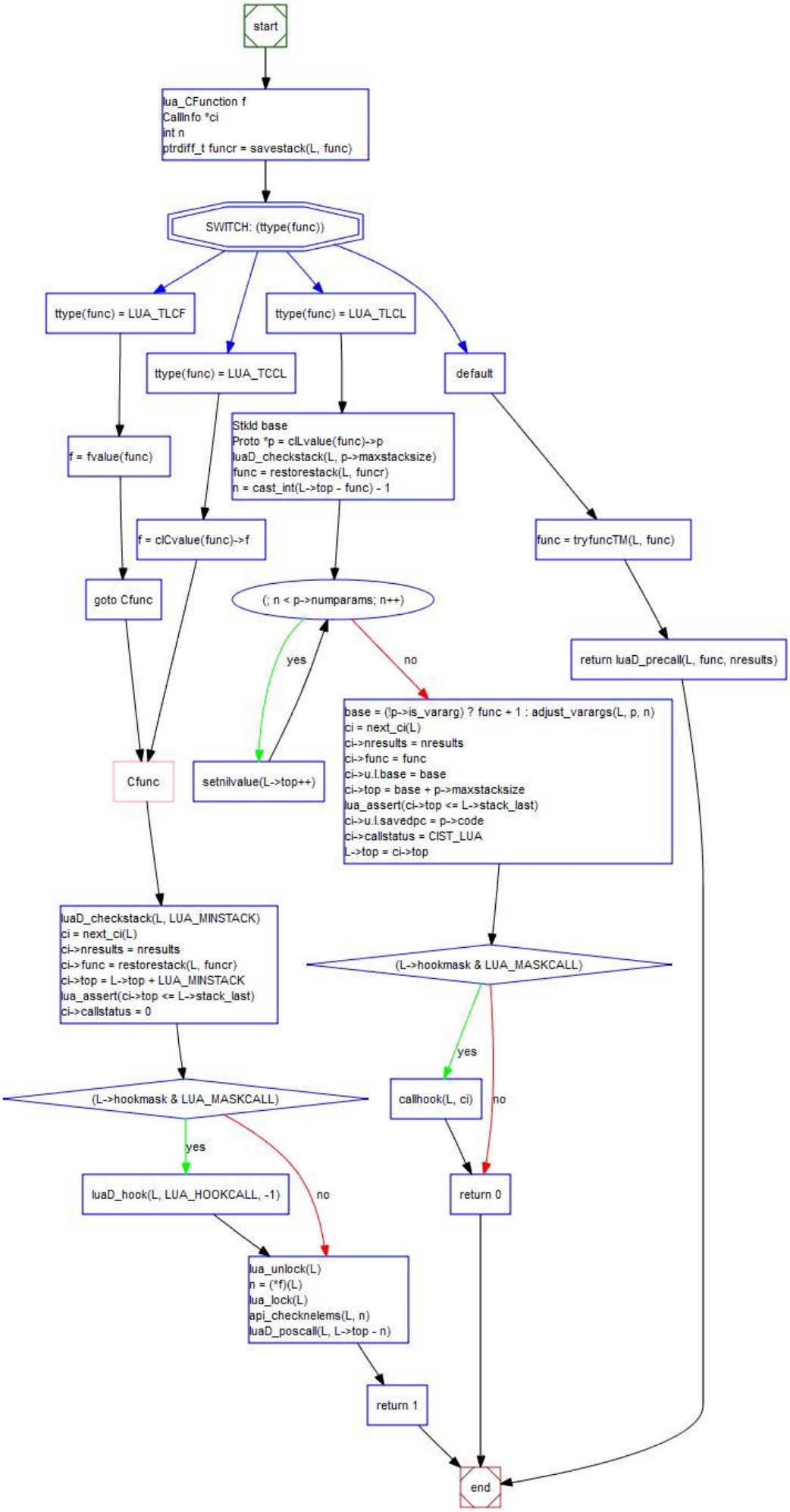
```
int luaD_precall (lua_State *L, StkId func, int nresults) {
  lua_CFunction f;
  CallInfo *ci;
  int n;  /* number of arguments (Lua) or returns (C) */
  ptrdiff_t funcr = savestack(L, func);
  switch (ttype(func)) {
    case LUA_TLCF:  /* light C function */
      f = fvalue(func);
      goto Cfunc;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

[8] ldo.c 函数调用以及栈管理

```
            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 */
        }
      }
    }
```

听其言观其行




start
lua_CFunction f
CallInfo *ci
int n
ptrdiff_t funcr = savestack(L, func)
SWITCH: (ttype(func))
ttype(func) = LUA_TLCF
ttype(func) = LUA_TCCL
ttype(func) = LUA_TLCL
default
f = fvalue(func)
f = clCvalue(func)->f
StkId base
Proto *p = clLvalue(func)->p
luaD_checkstack(L, p->maxstacksize)
func = restorestack(L, funcr)
n = cast_int(L->top - func) - 1
func = tryfuncTM(L, func)
goto Cfunc
(; n < p->numparams; n++)
yes
no
return luaD_precall(L, func, nresults)
Cfunc
setnilvalue(L->top++)
base = (!p->is_vararg) ? func + 1 : adjust_varargs(L, p, n)
ci = next_ci(L)
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
ci->callstatus = CIST_LUA
L->top = ci->top
luaD_checkstack(L, LUA_MINSTACK)
ci = next_ci(L)
ci->nresults = nresults
ci->func = restorestack(L, funcr)
ci->top = L->top + LUA_MINSTACK
lua_assert(ci->top <= L->stack_last)
ci->callstatus = 0
(L->hookmask & LUA_MASKCALL)
yes
no
callhook(L, ci)
(L->hookmask & LUA_MASKCALL)
yes
no
luaD_hook(L, LUA_HOOKCALL, -1)
return 0
lua_unlock(L)
n = (*f)(L)
lua_lock(L)
api_checknelems(L, n)
luaD_poscall(L, L->top - n)
return 1
end

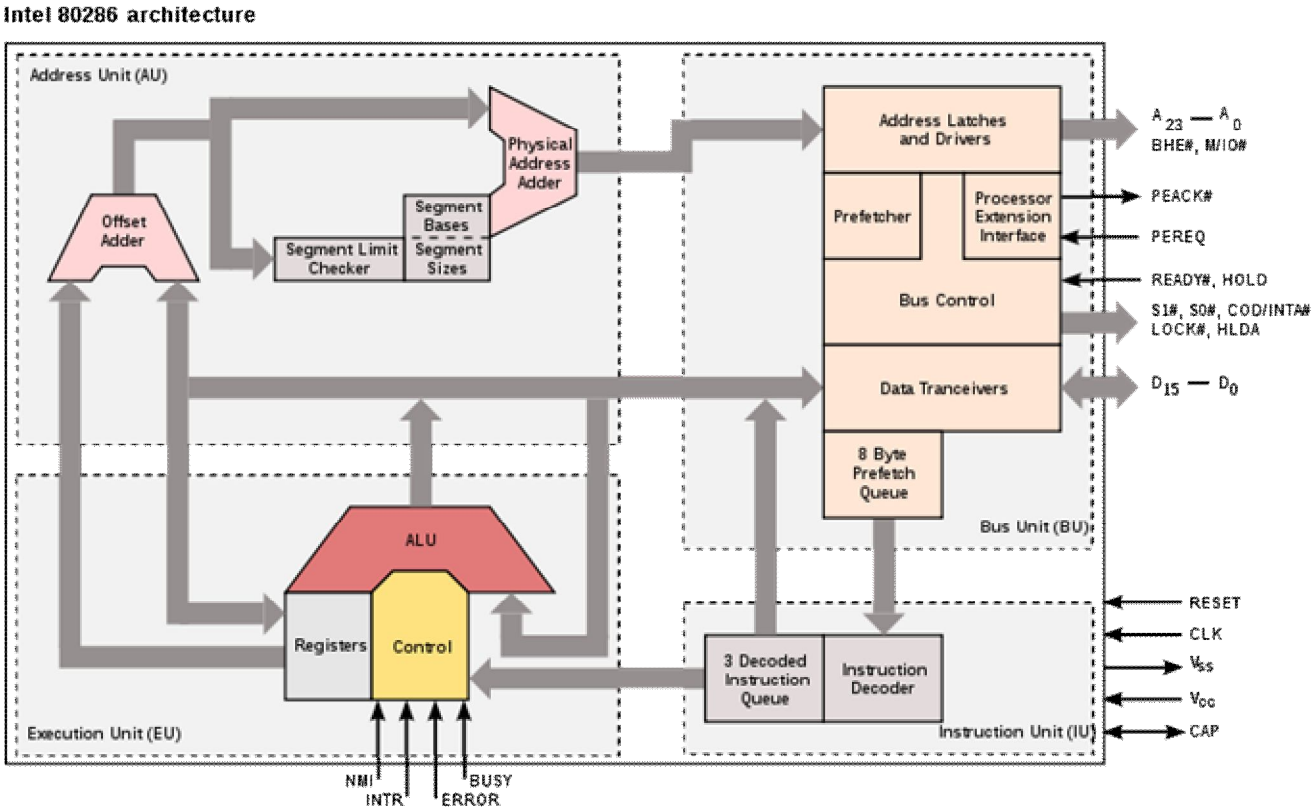
(可将图片复制到图像编辑软件中查看详细)

发现了 3 中函数类型

```
#define LUA_TLCL(LUA_TFUNCTION | (0 << 4))  /* Lua closure */
#define LUA_TLCF(LUA_TFUNCTION | (1 << 4))  /* light C function */
#define LUA_TCCL     (LUA_TFUNCTION | (2 << 4))  /* C closure */
```

分别是 lua 闭包、c 函数、c 闭包。杯具的是上面代码的 average 被当做了 c 闭包?不解中。姑妄听之。言归正传，果真如此，引擎还是保持的 c 对象的地址(这里是函数地址)，命令天使函数去执行这个函数地址的函数(代码中注释/* do the actual call */的这一行)。

LVM 如何工作?
答:不知道。先搞个 X86 CPU 架构[9]看看吧。



CPU 核心如何上机器码、如何上内存数据、如何对数据进行算术逻辑运算，神马总线、神马寄存器还算了解。引擎(lua 核心)呢？栈模型是有的，字节码摆在那里了，一个类似 IP 的东西索引对象是该有的，基本数据对象寻址是该有的。字节码指令集是会有的，控制器是该有的。

至此扯淡完毕[10]，可以开始按部就班的边看资料边对照代码分析了。

[9] http://en.wikipedia.org/wiki/Microarchitecture

[10] 本节仅从宏观上对引擎进行认识

6.1.1 展开为非核心对象

StkId 被用于表示栈元素的索引值，展开为

```
union Value {
      GCObject *gc;      /* collectable objects */
      void *p;           /* light userdata */
      int b;             /* booleans */
      lua_CFunction f; /* light C functions */
      numfield           /* numbers */
};

#define TValuefields    Value value_; int tt_

struct lua_TValue {
      TValuefields;
};

typedef struct lua_TValue TValue;

typedef TValue *StkId;   /* index to stack elements */

StkId top;   /* first free slot in the stack */
```

即 StkId 是

```
typedef struct
{
 union Value {
      GCObject *gc;      /* collectable objects */
      void *p;           /* light userdata */
      int b;             /* booleans */
      lua_CFunction f; /* light C functions */
      numfield           /* numbers */
 } value_;
 int tt_
}* StkId;
```

Lua 栈的本质是表的集合。栈中每一项均是表指针。
疑问:如何区分栈中表和普通类型？(见疑问章节)

6.1.2 展开为核心对象

```
/* little endian */
#define TValuefields  \
union { struct { Value v__; int tt__; } i; double d__; } u
#define NILCONSTANT {{NULL}, tag2tt(LUA_TNIL)}}
/* field-access macros */
#define v_(o)       ((o)->u.i.v__)
```

```
#define d_(o)       ((o)->u.d__)
#define tt_(o)      ((o)->u.i.tt__)

//#define TValuefields  Value value_; int tt_

struct lua_TValue {
    TValuefields;
};
```

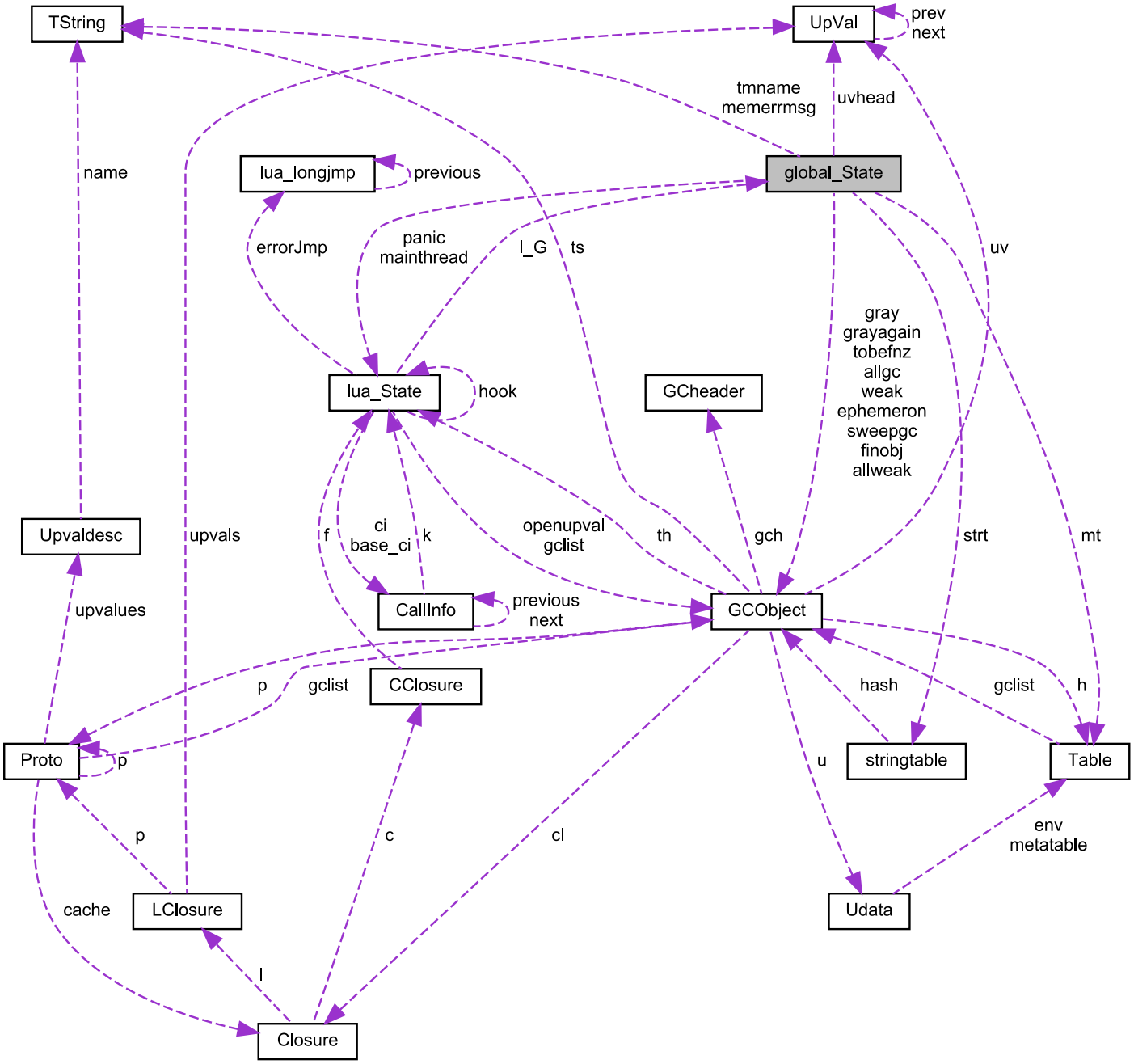
lua_TValue

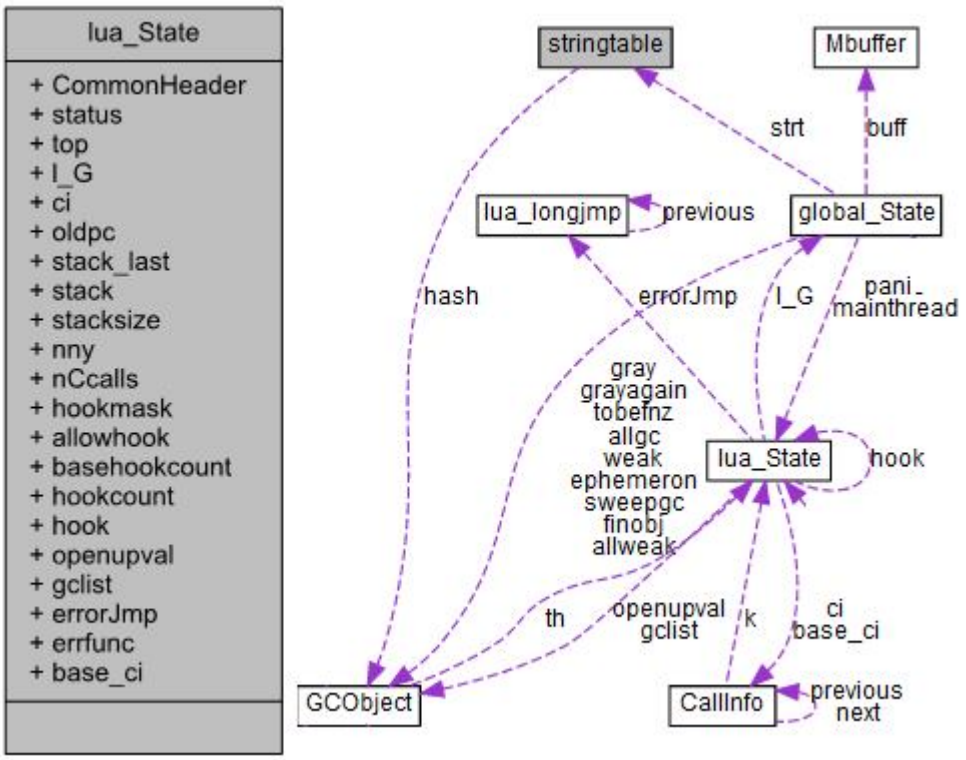
```
struct lua_TValue {
    union
    {
        struct{
            Value v__;
            int tt__;
        } i;
        double d__;
    }u;
};
```

```
union Value {
  GCObject *gc;    /* collectable objects */
  void *p;         /* light userdata */
  int b;           /* booleans */
  lua_CFunction f; /* light C functions */
  numfield         /* numbers */
};
```

6.2 状态机

Lua 是一个独立而完整的状态机，为整个系统提供一个模式一致的低耦合的对外接口做出了杰出贡献。先看全貌，


TString
UpVal
prev
next
tmname
memerrmsg
uvhead
name
lua_longjmp
previous
global_State
errorJmp
panic
mainthread
l_G
ts
uv
gray
grayagain
tobefnz
allgc
weak
ephemeron
sweepgc
finobj
allweak
lua_State
hook
GCheader
Upvaldesc
upvals
f
ci
base_ci
k
openupval
gclist
th
gch
strt
mt
upvalues
CallInfo
previous
next
GCObject
p
gclist
CClosure
hash
gclist
h
Proto
p
u
stringtable
Table
p
c
cl
env
metatable
cache
LClosure
Udata
l
Closure



6.2.1 CallInfo(L->ci)

CallInfo 是一个双向链表。

StkId	func	/* function index in the stack */
StkId	top	/* top for this function */
struct CallInfo *	previous	
struct CallInfo *	next	
short	nresults	函数期望值
lu_byte	callstatus	CallInfo 位状态
struct	l	Lua 函数
struct	c	C 函数

6.2.2 CallInfo 位状态(Bits in CallInfo status)

7	6	5	4	3	2	1	0
-	CIST_TAIL	CIST_STAT	CIST_YPCALL	CIST_YIELDED	CIST_REENTRY	CIST_HOOKED	CIST_LUA
	尾调用	有错误码	让步调用	从挂起恢复	被递归调用	debug	c

例如，判断一个调用是否是 lua 调用，可以

#define isLua(ci) ((ci)->callstatus & CIST_LUA)

位状态的原始定义

#define CIST_LUA (1<<0) /* call is running a Lua function */

```
#define CIST_HOOKED (1<<1)  /* call is running a debug hook */
#define CIST_REENTRY(1<<2)  /*call is running on same invocation of   luaV_execute */
#define CIST_YIELDED (1<<3)  /* call reentered after suspension */
#define CIST_YPCALL  (1<<4)  /* call is a yieldable protected call */
#define CIST_STAT  (1<<5)  /* call has an error status (pcall) */
#define CIST_TAIL  (1<<6)  /* call was tail called */
```

7 虚拟机

参考
A No-Frills Introduction to Lua 5.1 VM Instructions.pdf
by Kein-Hong Man, esq. <khman AT users.sf.net>

8 闭包

9 垃圾回收

10 Case

10.1 初始化和加载脚本

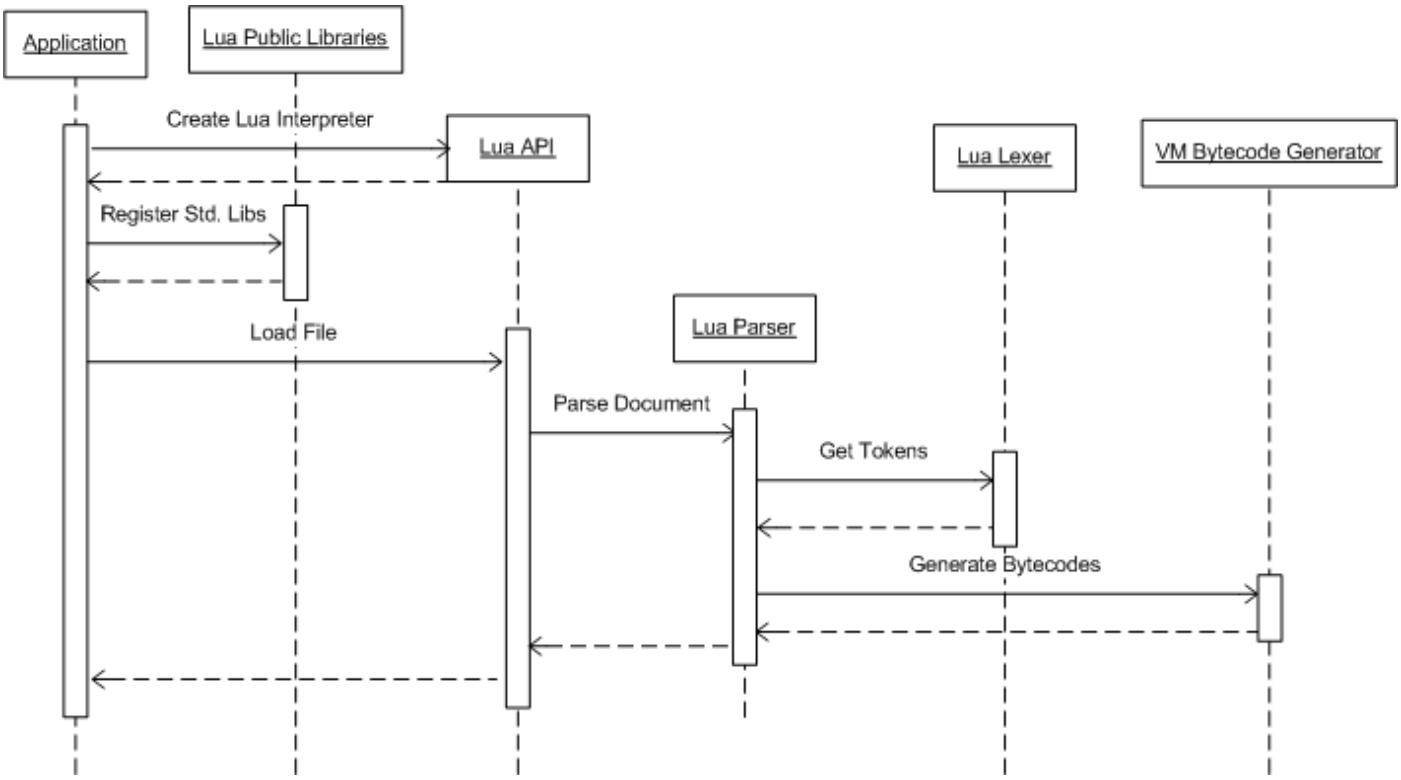


Figure 1: process of initializing Lua and loading a script file

10.2 Lua runtime code reading Lua 运行期源代码分析阅读

http://sunxiunan.com/?p=1477

You could download the project for VC2008 in http://groups.google.com/group/lua5/web/lua_vc2008.rar
The project will use parameter like "c:\test.lua", and the lua script like above image.

The code starting point is pmain().

lua parser will parse the code file, and use LexState *ls to store the information.

After the binary code generating, opcodes will run in function luaV_execute().

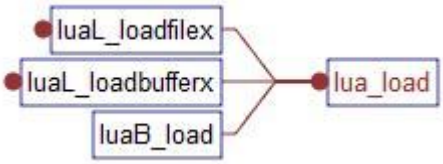
for the code in standard library (c function), it will be called in function luaD_precall().

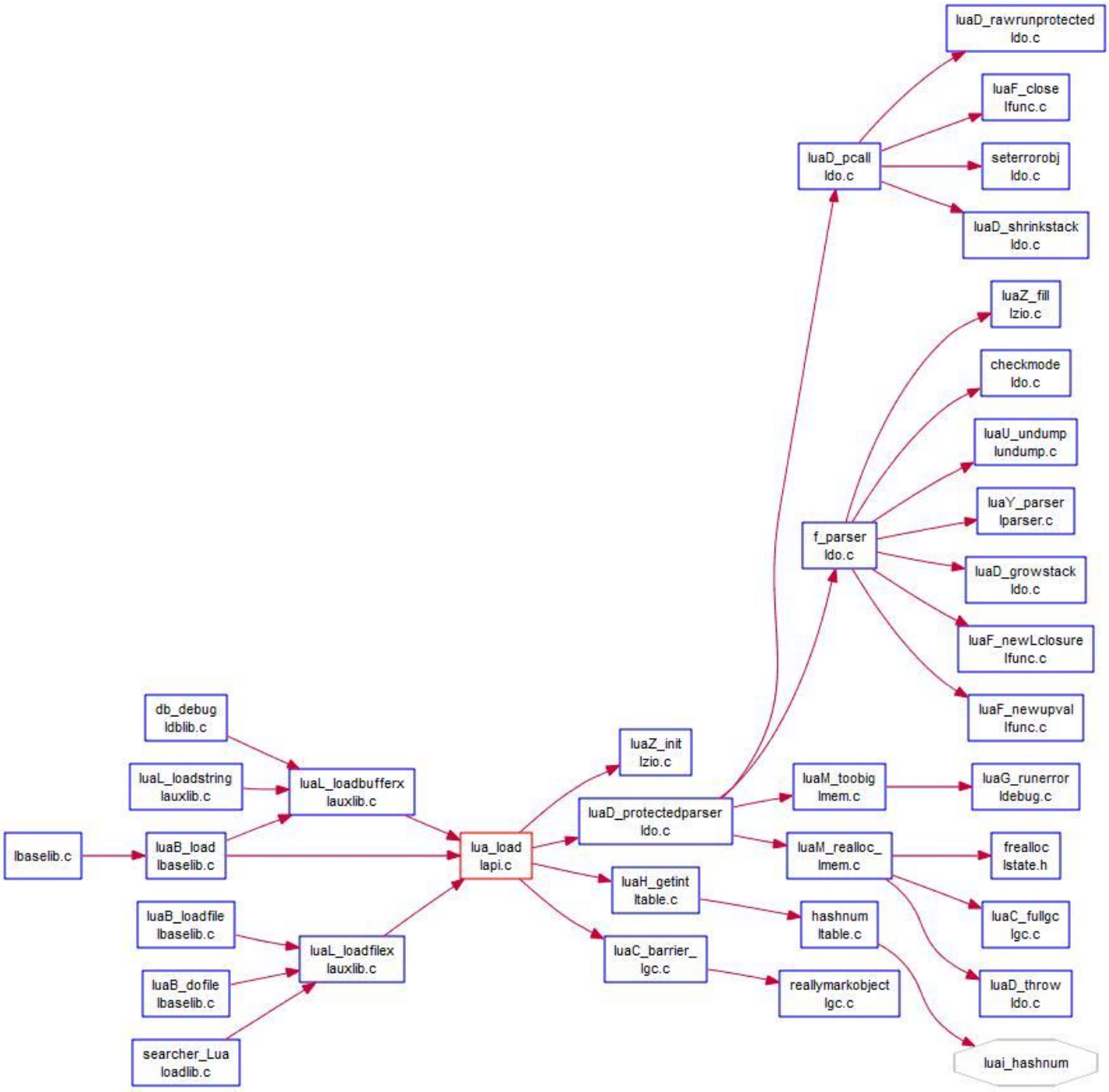
Output like:

```
 [luaD_precall] CFunc:0044D0E0 L->base:00393368 L->top:00393378 ci->func:00393358   ##
[pmain] 00393190 BEGIN ##
[luaL_openlibs] lib->name:[] func:0041E4C9 ##
[lua_pushcclosure] fn:0041E4C9 n:0 L->top:00393378 — cl:00396B00 L->top:00393388 ##
[luaD_precall] CFunc:0041E4C9 L->base:00393388 L->top:00393398 ci->func:00393378   ##
[lua_pushcclosure] fn:004265E0 n:0 L->top:003933A8 — cl:00396D10 L->top:003933B8 ##
[luaI_openlib] libname:[_G] [assert] func:004265E0 ##
```

10.3 加载脚本

lua_load





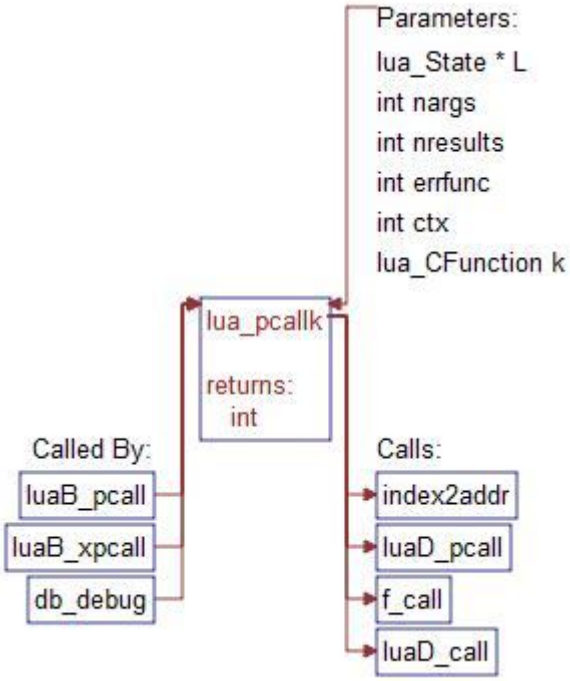
lua_load

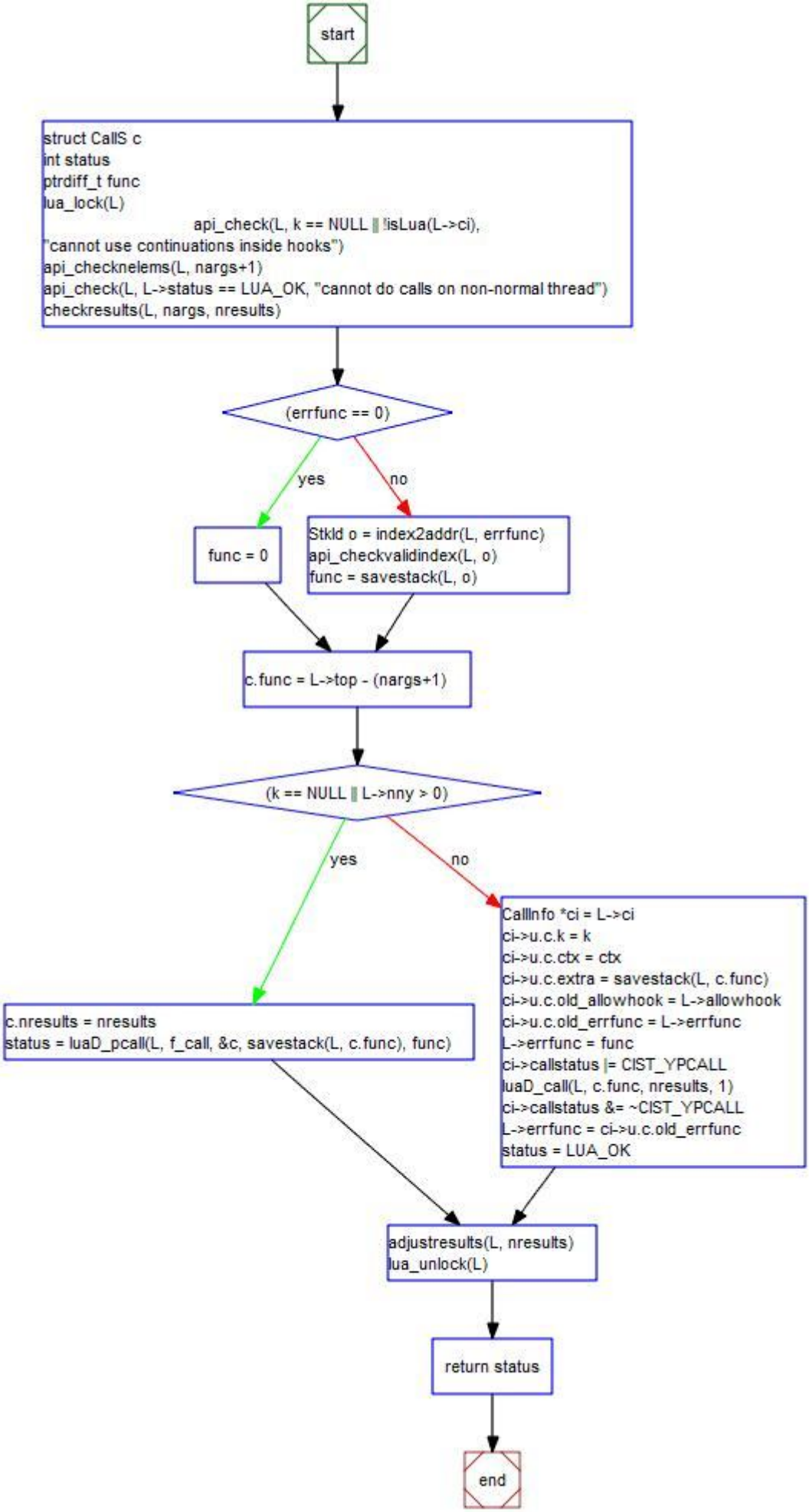
```
LUA_API int lua_load (lua_State *L, lua_Reader reader, void *data,
                      const char *chunkname, const char *mode) {
  ZIO z;
  int status;
  lua_lock(L);
  if (!chunkname) chunkname = "?";
  luaZ_init(L, &z, reader, data);
  status = luaD_protectedparser(L, &z, chunkname, mode);
  if (status == LUA_OK) {  /* no errors? */
    LClosure *f = clLvalue(L->top - 1);  /* get newly created function */
    if (f->nupvalues == 1) {  /* does it have one upvalue? */
      /* get global table from registry */
      Table *reg = hvalue(&G(L)->l_registry);
```

```
      const TValue *gt = luaH_getint(reg, LUA_RIDX_GLOBALS);
      /* set global table as 1st upvalue of 'f' (may be LUA_ENV) */
      setobj(L, f->upvals[0]->v, gt);
      luaC_barrier(L, f->upvals[0], gt);
    }
  }
  lua_unlock(L);
  return status;
}
```

10.4 执行字节码

lua_pcallk


Parameters:
lua_State * L
int nargs
int nresults
int errfunc
int ctx
lua_CFunction k
lua_pcallk
returns:
int
Called By:
luaB_pcall
luaB_xpcall
db_debug
Calls:
index2addr
luaD_pcall
f_call
luaD_call


start
struct CallS c
int status
ptrdiff_t func
lua_lock(L)
api_check(L, k == NULL || !isLua(L->ci),
"cannot use continuations inside hooks")
api_checknelems(L, nargs+1)
api_check(L, L->status == LUA_OK, "cannot do calls on non-normal thread")
checkresults(L, nargs, nresults)
(errfunc == 0)
yes
no
func = 0
StkId o = index2addr(L, errfunc)
api_checkvalidindex(L, o)
func = savestack(L, o)
c.func = L->top - (nargs+1)
(k == NULL || L->nny > 0)
yes
no
CallInfo *ci = L->ci
ci->u.c.k = k
ci->u.c.ctx = ctx
ci->u.c.extra = savestack(L, c.func)
ci->u.c.old_allowhook = L->allowhook
ci->u.c.old_errfunc = L->errfunc
L->errfunc = func
ci->callstatus |= CIST_YPCALL
luaD_call(L, c.func, nresults, 1)
ci->callstatus &= ~CIST_YPCALL
L->errfunc = ci->u.c.old_errfunc
status = LUA_OK
c.nresults = nresults
status = luaD_pcall(L, f_call, &c, savestack(L, c.func), func)
adjustresults(L, nresults)
lua_unlock(L)
return status
end

10.5 创建 math 库

在“阅读源码顺序”一节中，第一个阅读源码对象是 lmathlib.c，有必要看清楚。

```
static int math_atan2 (lua_State *L) {
  lua_pushnumber(L, l_tg(atan2)(luaL_checknumber(L, 1),
                                luaL_checknumber(L, 2)));
  return 1;
}
```

检查栈上第一个和第二个参数是否为数字
使用 c 的数学函数计算结果
结果放到栈顶上

```
static const luaL_Reg mathlib[] = {
  {"abs",    math_abs},
  {"acos",   math_acos},
  {"asin",   math_asin},
  {"atan2", math_atan2},
  {"atan",   math_atan},
  {"ceil",   math_ceil},
  {"cosh",    math_cosh},
  {"cos",    math_cos},
  {"deg",    math_deg},
  {"exp",    math_exp},
  {"floor", math_floor},
  {"fmod",    math_fmod},
  {"frexp", math_frexp},
  {"ldexp", math_ldexp},
#if defined(LUA_COMPAT_LOG10)
  {"log10", math_log10},
#endif
  {"log",    math_log},
  {"max",    math_max},
  {"min",    math_min},
  {"modf",    math_modf},
  {"pow",    math_pow},
  {"rad",    math_rad},
  {"random",      math_random},
  {"randomseed", math_randomseed},
  {"sinh",    math_sinh},
  {"sin",    math_sin},
  {"sqrt",   math_sqrt},
  {"tanh",    math_tanh},
  {"tan",    math_tan},
  {NULL, NULL}
};
```

注册到 mathlib 的所有函数

```
LUAMOD_API int luaopen_math (lua_State *L) {
```

```
  luaL_newlib(L, mathlib);
  lua_pushnumber(L, PI);
  lua_setfield(L, -2, "pi");
  lua_pushnumber(L, HUGE_VAL);
  lua_setfield(L, -2, "huge");
  return 1;
}
```

注册一个库。

```
275 LUAMOD_API int luaopen_math (lua_State *L) {
276   luaL_newlib(L, mathlib);
277   lua_pushnumber(L, PI);
278   lua_setfield(L, -2, "pi");
279   lua_pushnumber(L, HUGE_VAL);
280   lua_setfield(L, -2, "huge");
281   return 1;
282 }

#define luaL_newlibtable(L,l)   \
  lua_createtable(L, 0, sizeof(l)/sizeof((l)[0]) - 1)

#define luaL_newlib(L,l)    (luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

api.c

```
666 LUA_API void lua_createtable (lua_State *L, int narray, int nrec) {
667   Table *t;
668   lua_lock(L);
669   luaC_checkGC(L);
670   t = luaH_new(L);
671   sethvalue(L, L->top, t);
672   api_incr_top(L);
673   if (narray > 0 || nrec > 0)
674     luaH_resize(L, t, narray, nrec);
675   lua_unlock(L);
676 }

104 #define luaC_condGC(L,c) \
105     {if (G(L)->GCdebt > 0) {c;}; condchangemem(L);}
106 #define luaC_checkGC(L)     luaC_condGC(L, luaC_step(L);)

464 LUA_API void lua_pushnumber (lua_State *L, lua_Number n) {
465   lua_lock(L);
466   setnvalue(L->top, n);
467   luai_checknum(L, L->top,
468     luaG_runerror(L, "C API - attempt to push a signaling NaN"));
469   api_incr_top(L);
470   lua_unlock(L);
471 }

753 LUA_API void lua_setfield (lua_State *L, int idx, const char *k) {
754   StkId t;
755   lua_lock(L);
756   api_checknelems(L, 1);
757   t = index2addr(L, idx);
758   api_checkvalidindex(L, t);
759   setsvalue2s(L, L->top++, luaS_new(L, k));
760   luaV_settable(L, t, L->top - 1, L->top - 2);
761   L->top -= 2;  /* pop value and key */
762   lua_unlock(L);
763 }
```

ltable.c

```
Table *luaH_new (lua_State *L) {
  Table *t = &luaC_newobj(L, LUA_TTABLE, sizeof(Table), NULL, 0)->h;
  t->metatable = NULL;
  t->flags = cast_byte(~0);
  t->array = NULL;
  t->sizearray = 0;
  setnodevector(L, t, 0);
  return t;
}
```

lgc.c

```
GCObject *luaC_newobj (lua_State *L, int tt, size_t sz, GCObject **list
                       int offset) {
  global_State *g = G(L);
  GCObject *o = obj2gco(cast(char *, luaM_newobject(L, tt, sz)) + offse
  if (list == NULL)
    list = &g->allgc;  /* standard list for collectable objects */
  gch(o)->marked = luaC_white(g);
  gch(o)->tt = tt;
  gch(o)->next = *list;
  *list = o;
  return o;
}
```

同时可以参考《Lua 源码欣赏.pdf》[11]。

10.6 set\get 方法辨析

Get 方法

```
LUA_API void  (lua_getglobal) (lua_State *L, const char *var);
LUA_API void  (lua_gettable) (lua_State *L, int idx);
LUA_API void  (lua_getfield) (lua_State *L, int idx, const char *k);
LUA_API void  (lua_rawget) (lua_State *L, int idx);
LUA_API void  (lua_rawgeti) (lua_State *L, int idx, int n);
LUA_API void  (lua_rawgetp) (lua_State *L, int idx, const void *p);
LUA_API void  (lua_createtable) (lua_State *L, int narr, int nrec);
LUA_API void *(lua_newuserdata) (lua_State *L, size_t sz);
LUA_API int   (lua_getmetatable) (lua_State *L, int objindex);
LUA_API void  (lua_getuservalue) (lua_State *L, int idx);
```

Set 方法

```
LUA_API void  (lua_setglobal) (lua_State *L, const char *var);
```

[11] math 模块注册机制,Lua 源码欣赏.pdf,云风著

```
LUA_API void  (lua_settable) (lua_State *L, int idx);
LUA_API void  (lua_setfield) (lua_State *L, int idx, const char *k);
LUA_API void  (lua_rawset) (lua_State *L, int idx);
LUA_API void  (lua_rawseti) (lua_State *L, int idx, int n);
LUA_API void  (lua_rawsetp) (lua_State *L, int idx, const void *p);
LUA_API int   (lua_setmetatable) (lua_State *L, int objindex);
LUA_API void  (lua_setuservalue) (lua_State *L, int idx);
```

<待补充>

11 调试与分析

http://www.tecgraf.puc-rio.br/~lhf/ftp/lua/#tokenf

11.1 VM Code

test.lua

```
y = 5
print(y)
```

```
E:\GameDev\Script\luas\analyse>luac -p -l test.lua

main <test.lua:0,0> (6 instructions, 24 bytes at 004F7AF0)
0+ params, 2 slots, 0 upvalues, 0 locals, 3 constants, 0 functions
        1       [1]     LOADK           0 -2    ; 5
        2       [1]     SETGLOBAL       0 -1    ; y
        3       [2]     GETGLOBAL       0 -3    ; print
        4       [2]     GETGLOBAL       1 -1    ; y
        5       [2]     CALL            0 2 1
        6       [2]     RETURN          0 1
```

main <test.lua:0,0> (6 instructions, 24 bytes at 005F7AF0)

0+ params, 2 slots, 0 upvalues, 0 locals, 3 constants, 0 functions

1	[1]	LOADK	0 -2	; 5
2	[1]	SETGLOBAL	0 -1	; y
3	[2]	GETGLOBAL	0 -3	; print
4	[2]	GETGLOBAL	1 -1	; y
5	[2]	CALL	0 2 1	

6	[2]	RETURN	0 1	

11.2 LuaCov

一个分析工具，可以标记 lua 文件中那些代码行被执行了，那些没有被执行
http://luacov.luaforge.net/

示例
运行命令行
`lua -lluacov test.lua`
会输出分析结果 lcov.report.out

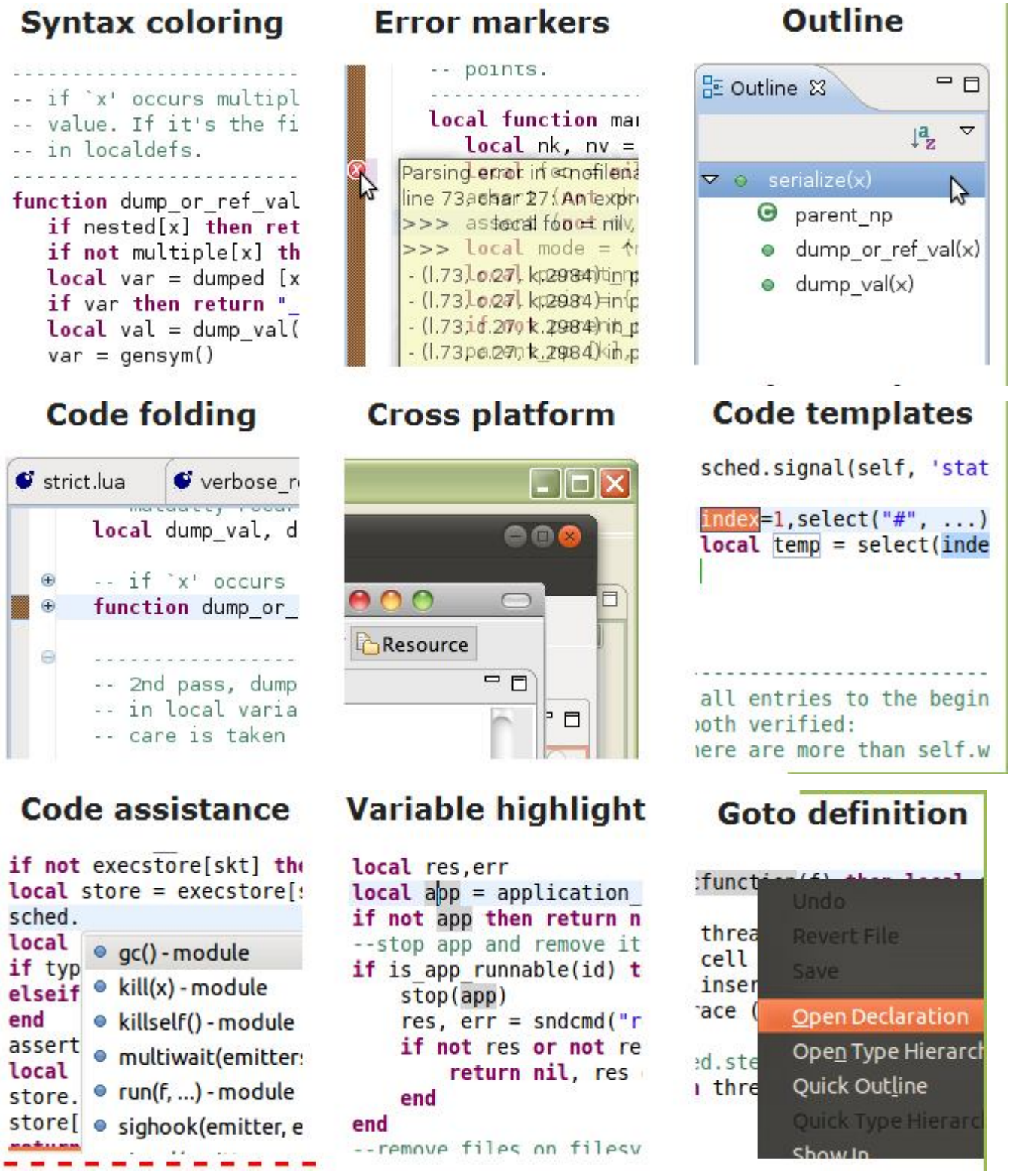
```
==============================================================
../test.lua
==============================================================

       -- Which branch will run?
1      if 10 > 100 then
0        print("I don't think this line will execute.")
0      else
1        print("Hello, LuaCov!")
1      end
```

11.3 LDT

作为 Eclipse 的一个插件使用，开源。
http://eclipse.org/koneki/ldt/
源码下载地址:
http://git.eclipse.org/c/koneki/org.eclipse.koneki.ldt.git/

LDT is about providing Lua developers with a proper user assistance. A user experience as comfortable

as the one common static languages users are used to.



11.4 lua 调试器：运行时的值查看

参考资料

12 基础模型

13 语法糖

13.1 类实现

用 C++类来认识 C 方式的实现，仅用户辅助分析代码，只是在抽象层面帮助记忆用。最终是要 OO，不是要 OO C++或 OO C。只是熟悉 C++的缘故，C++ OO 比较好记忆和查看。

文件就是类(对象)
H 头文件中声明的 struct 的对象是类的变量
文件靠前的 static 函数是类的私有函数
文件靠后的 LUA_API 打头的全局函数是类的公开函数

考虑 lua_State

```
struct lua_State {
  CommonHeader;
  lu_byte status;
  StkId top;  /* first free slot in the stack */
  global_State *l_G;
  CallInfo *ci;  /* call info for current function */
  const Instruction *oldpc;  /* last pc traced */
```

```
  StkId stack_last;  /* last free slot in the stack */
  StkId stack;  /* stack base */
  int stacksize;
  unsigned short nny;  /* number of non-yieldable calls in stack */
  unsigned short nCcalls;  /* number of nested C calls */
  lu_byte hookmask;
  lu_byte allowhook;
  int basehookcount;
  int hookcount;
  lua_Hook hook;
  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) */
  CallInfo base_ci;  /* CallInfo for first level (C calling Lua) */
};
```

派生自类 CommonHeader

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

构造函数是

```
LUA_API lua_State *lua_newstate (lua_Alloc f, void *ud);
```

析构函数是

```
LUA_API void lua_close (lua_State *L);
```

公开方法是

lstate.c

```
LUA_API lua_State *lua_newthread (lua_State *L)
void luaE_freeCI (lua_State *L) {
CallInfo *luaE_extendCI (lua_State *L) {
```

lapi.c

int	lua_gettop (lua_State *L)
void	lua_settop (lua_State *L, int idx)
void	lua_remove (lua_State *L, int idx)
…	…

私有方法是

```
static void close_state (lua_State *L) {
static void preinit_state (lua_State *L, global_State *g) {
static void f_luaopen (lua_State *L, void *ud) {
static void init_registry (lua_State *L, global_State *g) {
static void freestack (lua_State *L) {
static void stack_init (lua_State *L1, lua_State *L) {
```

如此这般，一个用 c 实现的类 lua_State 就浮现出来了。这将是一个巨类，几百的公开方法。这显然是极端的做法，仗着 C 语言语法特点的灵活性就玩命了。

一个理想的 C++实现方式(这里是从抽象层面假设将 c 实现的 lua 源码用 c++思维方式考虑)是，一个文件对应一个类。由于所有的 LUA_API 都将 lua_State 当做第一个参数，实现 lua_State 就非常简洁明了，按上文的方式实现即可。考虑其他类时，按功能划分类，这里一个文件即是一个类，文件名本身就描述的次文件的功能，然后将文件中的类成员和类对象抽离出来。对于函数，可以从第一个参数 L 顺藤摸瓜，找到方法中类的主体。例如 lvm，其核心是对栈的维护，根据核心功能和文件内容，将 class

Lvm 具体化。

伪代码:

```
Class Lvm
{
	core_statck statck;
	CallInfo calls; //
	TValue *tm;  //table
private:
	static void callTM (lua_State *L, const TValue *f, const TValue *p1,

public:
	void luaV_execute (lua_State *L) {
};
```

当然这是理所当然的想法。没有熟悉引擎核心是无法抽象和理解所有对象的依赖关系以及各个对象的行为的。

至此，在围绕对象和语言的角度上，该幻想的都幻想完了。幻想就是幻想，不能当真。

13.2 类型转换

```
#define cast(t, exp)	((t)(exp))
#define cast_byte(i)	cast(lu_byte, (i))
#define cast_num(i)	cast(lua_Number, (i))
#define cast_int(i)	cast(int, (i))
#define cast_uchar(i)	cast(unsigned char, (i))
```

13.3 LUA_CORE

此部分为核心文件，共 20 个。可以参考文件结构一章。文件头定义了 LUA_CORE 宏即为核心文件。

```
#define LUA_CORE
```

LUA_CORE 打开了一些列核心引擎所需的结构，下面分别有说明，其中最重要的是打开了一个核心数据封装的方式，以便在数据使用上有最优表现。

核心文件列表

Index	File	宏定义所在行

1	ldo.c	13
2	lopcodes.c	8
3	ltm.c	11
4	lvm.c	13
5	lapi.c	12
6	lctype.c	8
7	lfunc.c	11
8	lobject.c	13
9	lstate.c	11
10	lstring.c	11
11	ltable.c	24
12	lcode.c	11
13	ldump.c	10
14	llex.c	12
15	lparser.c	11
16	lundump.c	10
17	ldebug.c	14
18	lgc.c	10
19	lmem.c	11
20	lzio.c	11

13.3.1 定义动态库

```
#if defined(LUA_BUILD_AS_DLL)

#if defined(LUA_CORE) || defined(LUA_LIB)
#define LUA_API __declspec(dllexport)
#else
#define LUA_API __declspec(dllimport)
#endif

#else

#define LUA_API		extern

#endif
```

由于此处定义了动态库，则 LUA_API 被定义为

```
__declspec(dllexport)
```

13.3.2 定义标准操作

```
/* these are quite standard operations */
#if defined(LUA_CORE)
#define luai_numadd(L,a,b)	((a)+(b))
```

```
#define luai_numsub(L,a,b)     ((a)-(b))
#define luai_nummul(L,a,b)     ((a)*(b))
#define luai_numdiv(L,a,b)((a)/(b))
#define luai_numunm(L,a) (-(a))
#define luai_numeq(a,b)        ((a)==(b))
#define luai_numlt(L,a,b)  ((a)<(b))
#define luai_numle(L,a,b)  ((a)<=(b))
#define luai_numisnan(L,a)(!luai_numeq((a), (a)))
#endif
```

直接进行立即数操作，似乎没什么特别的。

13.3.3 打开 MS compiler 汇编

此处打开了 MS_ASMTRICK 宏，

```
#if defined(LUA_CORE)       /* { */
#if defined(LUA_NUMBER_DOUBLE) && !defined(LUA_ANSI)   /* { */
/* On a Microsoft compiler on a Pentium, use assembler to avoid clashes
   with a DirectX idiosyncrasy */
#if defined(LUA_WIN) && defined(_MSC_VER) && defined(_M_IX86) /* { */
#define MS_ASMTRICK
#else               /* }{ */
/* the next definition uses a trick that should work on any machine
   using IEEE754 with a 32-bit integer type */
#define LUA_IEEE754TRICK

/*
@@ LUA_IEEEENDIAN is the endianness of doubles in your machine
** (0 for little endian, 1 for big endian); if not defined, Lua will
** check it dynamically.
*/
/* check for known architectures */
#if defined(__i386__) || defined(__i386) || defined(__X86__) || \
    defined (__x86_64)
#define LUA_IEEEENDIAN 0
#elif defined(__POWERPC__) || defined(__ppc__)
#define LUA_IEEEENDIAN 1
#endif

#endif              /* } */
#endif          /* } */
#endif          /* } */
```

则 MS_ASMTRICK 被打开

```
#if defined(MS_ASMTRICK)    /* { */
/* trick with Microsoft assembler for X86 */
```

```
#define lua_number2int(i,n)  __asm {__asm fld n   __asm fistp i}
#define lua_number2integer(i,n)    lua_number2int(i, n)
#define lua_number2unsigned(i,n)  \
  {__int64 l; __asm {__asm fld n   __asm fistp l} i = (unsigned int)l;}


#elif defined(LUA_IEEE754TRICK)    /* }{ */
...
#endif             /* } */
```

13.3.4 数据包装技巧

一个将所有类型包装到一个 double 值的小技巧，打开了 LUA_NANTRICK_LE

```
/*
@@ LUA_NANTRICK_LE/LUA_NANTRICK_BE controls the use of a trick to
** pack all types into a single double value, using NaN values to
** represent non-number values. The trick only works on 32-bit machines
** (ints and pointers are 32-bit values) with numbers represented as
** IEEE 754-2008 doubles with conventional endianess (12345678 or
** 87654321), in CPUs that do not produce signaling NaN values (all NaNs
** are quiet).
*/
#if defined(LUA_CORE) && \
   defined(LUA_NUMBER_DOUBLE) && !defined(LUA_ANSI)    /* { */
/* little-endian architectures that satisfy those conditions */
#if defined(__i386__) || defined(__i386) || defined(__X86__) || \
   defined(_M_IX86)
#define LUA_NANTRICK_LE
#endif
#endif                          /* } */
```

于是 LUA_NANTRICK_LE 打开了一处重要的数据结构

```
#if defined(LUA_NANTRICK_LE)


/* little endian */
#define TValuefields  \
    union { struct { Value v__; int tt__; } i; double d__; } u
#define NILCONSTANT {{NULL}, tag2tt(LUA_TNIL)}}
/* field-access macros */
#define v_(o)       ((o)->u.i.v__)
#define d_(o)       ((o)->u.d__)
#define tt_(o)      ((o)->u.i.tt__)
```

```
#elif defined(LUA_NANTRICK_BE)

/* big endian */
#define TValuefields  \
    union { struct { int tt__; Value v__; } i; double d__; } u
#define NILCONSTANT {{tag2tt(LUA_TNIL), {NULL}}}
/* field-access macros */
#define v_(o)       ((o)->u.i.v__)
#define d_(o)       ((o)->u.d__)
#define tt_(o)      ((o)->u.i.tt__)

#elif !defined(TValuefields)
#error option 'LUA_NANTRICK' needs declaration for 'TValuefields'

#endif
```

此处的关键是打开了如下定义块

```
/* little endian */
#define TValuefields  \
    union { struct { Value v__; int tt__; } i; double d__; } u
#define NILCONSTANT {{NULL}, tag2tt(LUA_TNIL)}}
/* field-access macros */
#define v_(o)       ((o)->u.i.v__)
#define d_(o)       ((o)->u.d__)
#define tt_(o)      ((o)->u.i.tt__)
```

于是 tValue 就比较容易看懂了。

14 疑问

14.1 Tvaluefields 为何重复定义？

在 lobject.h 中，Tvaluefields 被重复定义:

Line 96

```
#define TValuefields	Value value_; int tt_
```

Line 285

```
#undef TValuefields
#undef NILCONSTANT

#if defined(LUA_NANTRICK_LE)

/* little endian */
#define TValuefields	\
	union { struct { Value v__; int tt__; } i; double d__; } u
#define NILCONSTANT	{{{NULL}, tag2tt(LUA_TNIL)}}
/* field-access macros */
#define v_(o)		((o)->u.i.v__)
#define d_(o)		((o)->u.d__)
#define tt_(o)		((o)->u.i.tt__)
```

但是在 285 行之前，似乎没有地方需要用到 TValuefields

14.2 如何区分栈中表和普通类型？

= 附录 =

15 元编程 MetaProgramming

http://en.wikipedia.org/wiki/Metaprogramming

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime. In some cases, this allows programmers to minimize the number of lines of code to express a solution (hence reducing development time), or it gives programs greater flexibility to efficiently handle new situations without recompilation.

The language in which the metaprogram is written is called the metalanguage. The language of the programs that are manipulated is called the *object language*. The ability of a programming language to be its own metalanguage is called*reflection* or *reflexivity*.

http://zh.wikipedia.org/wiki/元编程

元编程是指某类计算机程序的编写，这类计算机程序编写或者操纵其它程序（或者自身）作为它们的数据，或者在运行时完成部分本应在编译时完成的工作。

编写元程序的语言称之为元语言。被操纵的程序的语言称之为目标语言。一门编程语言同时也是自身的元语言的能力称之为反射或者自反。

16 闭包 Closure

http://en.wikipedia.org/wiki/Closure_(computer_science)

In computer science, a closure (also lexical closure, function closure, function value or functional value) is a function together with a referencing environment for the non-local variables of that function.[1] A closure allows a function to access variables outside its typical scope. Such a function is said to be "closed over" its free variables.

http://zh.wikipedia.org/wiki/闭包

在计算机科学中，闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。

在一些语言中，在函数中定义另一个函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。运行时，一旦外部的 函数被执行，一个闭包就形成了，闭包中包含了内部函数的代码，以及所需外部函数中的变量的引用。其中所引用的变量称作上值(upvalue)。

16.1 C 闭包

C 语言 (使用 LLVM 编译器或苹果修改版的 GCC)支持块。闭包变量用__block 标记。同时，这个扩展也可以应用到 Objective-C 与 C++中。

```
typedef int (^IntBlock)();

IntBlock downCounter(int start) {
        __block int i = start;
        return Block_copy( ^int() {
                return i--;
        });
 }

IntBlock f = downCounter(5);
printf("%d", f());
printf("%d", f());
printf("%d", f());
Block_release(f);
```

16.2 C++闭包

C++允许通过重载 operator()来定义函数对象。这种对象的行为在某种程度上与函数式编程语言中的函数类似。它们可以在运行时创建，保存状态，但是不能如闭包一般隐式获取局部变量。C++标准委员会正在考虑两种在 C++中引入闭包的建议（它们都称为 lambda 函数）[1], [2]。这些建议间主要的区别在于一种默认在闭包中储存全部局部变量的拷贝，而另一种只存储这些变量的引用。这两种建议都提供了可以覆盖默认行为的选项。若这两种建议之一被接受，则可以写如下代码

```
void foo(string myname)
{
    typedef std::vector< string > names;

    int y;
    names n;
    // ...
    names::iterator i =
        find_if(n.begin(), n.end(), [&](const string& s)
    {
        return s != myname && s.size() > y;
    });
    // 'i' is now either 'n.end()' or points to the first string in 'n'
    // 'i' 现在是'n.end()'或指向'n'中第一个
    // 不等于'myname'且长度大于'y'的字符串
}
```

至少两种 C++编译器，Visual C++ 2010（或 Visual C++ 10.0）与 gcc-4.5 已经支持了这种特性。

17 基础数据类型数据长度

http://msdn.microsoft.com/en-us/library/cc953fe1(v=VS.80).aspx

Type	*Size*
bool	1 byte
char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
long, unsigned long	4 bytes
float	4 bytes
double	8 bytes
long double1	8 bytes
long long	Equivalent to __int64.

18 The Complete Syntax of Lua

[12]Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::= varlist1 `=´ explist1 |
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** explist1 **do** block **end** |
function funcname funcbody |
local function Name funcbody |
local namelist [`=´ explist1]

laststat ::= **return** [explist1] | **break**

funcname ::= Name {`.´ Name} [`:´ Name]

varlist1 ::= var {`,´ var}

var ::= Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

[12] http://www.codingnow.com/2000/download/lua_manual.html

explist1 ::= {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 ::= `(´ [explist1] `)´ | tableconstructor | String

function ::= **function** funcbody

funcbody ::= `(´ [parlist1] `)´ block **end**

parlist1 ::= namelist [`,´ `...´] | `...´

tableconstructor ::= `{´ [fieldlist] `}´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp

fieldsep ::= `,´ | `;´

binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |
`<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |
and | **or**

unop ::= `-´ | **not** | `#´

19 递归下降分析器(Recursive descent parser)

Recursive descent parser
http://en.wikipedia.org/wiki/Recursive_descent_parser
In computer science, a recursive descent parser is a kind of top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

20 垃圾回收器

垃圾回收 (计算机科学)
http://zh.wikipedia.org/wiki/垃圾回收 (计算机科学)

20.1 基础算法

基础的算法有下面几种方式，参考记数、追踪收集、标记清除、复制收集、堆积压缩、标记压缩。

- ✧ 参考记数
- ✧ 追踪收集
- ✧ 标记清除
- ✧ 复制收集
- ✧ 堆积压缩
- ✧ 标记压缩

20.2 贝姆垃圾收集器

http://zh.wikipedia.org/wiki/贝姆垃圾收集器
Boehm-Demers-Weiser garbage collector，也就是著名的 Boehm GC，是计算机应用在 C/C++语言上的一个保守的垃圾回收器（garbage collector），可应用于许多经由 C\C++开发的专案, 同时也适用于其它执行环境的各类编程语言，包括了 GNU 版 Java 编译器执行环境，以及 Mono 的 Microsoft .NET 移植平台。

21 Lua 5.1 C API[13]

21.1 Push data

Push data	*Remark*
lua_pushboolean	push a boolean value
lua_pushinteger	push integer
lua_pushnumber	push Lua number (double)
lua_pushliteral	push string literal
lua_pushstring	push C string
lua_pushlstring	push string of given length
lua_pushfstring	push a sprintf() formatted string and return also pointer to result
lua_pushvfstring	push a vsprintf() formatted string and return also pointer to result
lua_pushcfunction	push a C function
lua_pushcclosure	push a C closure with n upvalues (value 1 being pushed first)
lua_pushlightuserdata	push light user data
lua_pushthread	push the thread of the current stack, return 1 f
lua_pushnil	push nil

21.2 Check data

Check data	*Remark*
lua_isboolean	return true if value at position n is a boolean
lua_isnumber	return true if value at position n is a number
lua_isstring	return true if value at position n is a string
lua_istable	return true if value at position n is a table
lua_isfunction	return true if value at position n is a Lua function
lua_iscfunction	return true if value at position n is a C function
lua_islightuserdata	return true if value at position n is a light user data
lua_isuserdata	return true if value at position n is a light/full user data
lua_isoneornil	return true if value at position n is nil or outside current stack
lua_isnone	return true if value at position n is outside current stack
lua_isnil	return true if value at position n is nil
lua_isthread	return true if value at position n is a thread

21.3 Get data checked

[13] lua_capi.pdf

Get data checked	*Remark*
luaL_checkany	check if value at position n is a valid value
luaL_checkinteger	check for integer and return int at position n
luaL_checkint	check for number and return int at position n
luaL_checklong	check for number and return long at position n
luaL_checkstring	check for string and return string at position n
luaL_checklstring	check for string and return string at position n and actual length
luaL_checknumber	check for number and return int at position n
luaL_checktype	check for Lua type t at position n
luaL_checkudata	check for userdata name and return its pointer at position n
luaL_checkoption	search index of n (or pc) in list ppc

21.4 Get data converted

Get data converted	*Remark*
lua_toboolean	convert value at position n to bool
lua_tocfunction	convert value at position n to a C function
lua_tointeger	convert value at position n to integer
lua_tostring	convert value at position n to C string, return pointer
lua_tolstring	convert value at position n to C string, return pointer and actual length
lua_tonumber	convert value at position n to a Lua number
lua_topointer	convert value at position n to pointer
lua_tothread	convert value at position n to thread
lua_touserdata	convert value at position n to light userdata

21.5 Get data with defaults

Get data with defaults	*Remark*
luaL_optint	check for number at position n, return n if number or d otherwise
luaL_optinteger	check for number at position n, return n if number or d otherwise
luaL_optlong	check for number at position n, return n if number or d otherwise
luaL_optnumber	check for number at position n, return n if integer or d otherwise
luaL_optlstring	check for string at position n, return n if string or pc with len otherwise
luaL_optstring	check for string at position n, return n if string or pc otherwise

21.6 Stack operator

Stack operator	Remark
lua_gettop	return the current size of the stack
lua_settop	set stack size to n

lua_insert	moves top element to position n
lua_pop	pop n values from stack
lua_pushvalue	push value at position n
lua_remove	remove value at position n
lua_replace	pop value and replace value at position n
lua_xmove	pop n values from L1, push to L2

21.7 Value operator

Value operator	Remark
lua_equal	return true if values at position n1 and n2 are equal
lua_lessthan	return true if value at position n1 is smaller than value at position n2
lua_rawequal	return true if value at position n1 is smaller than value at position n2 (without me
luaL_gsub	push copy of pc with all patt replaced by rep
lua_concat	pop n values and push concatenated strings 1 .. n

21.8 Table

Table	Remark
lua_createtable	create and push a new table with pre-allocated space
lua_newtable	create and push a new empty table
lua_getfield	push value of table at position n with field name
lua_setfield	pop value and store in table at position n with field key
lua_rawget	push value of table at position n with field at top (without metamethods)
lua_rawset	pop value and store in table at position n with field key (without metamethods)
lua_rawgeti	Remark
lua_rawseti	create and push a new table with pre-allocated space
lua_gettable	create and push a new empty table
lua_settable	push value of table at position n with field name
lua_getmetatable	pop value and store in table at position n with field key
lua_setmetatable	push value of table at position n with field at top (without metamethods)
LuaL_newmetatable	pop value and store in table at position n with field key (without metamethods)
lualL_getmetatable	Remark
lua_next	create and push a new table with pre-allocated space
lua_objlen	create and push a new empty table
luaL_getmetafield	push value of table at position n with field name

21.9 Global data

Global data	Remark
lua_setglobal	store a global value

lua_getglobal	push a global value
lua_setfenv	set environment table of value at position n
lua_getfenv	push environment table of value at position n
lua_register	register C function in global table with name
luaL_register	open library with list elements in table on stack, if name is != 0 create and push new table

21.10 Call function

Call function	Remark
lua_call	call Lua function func with ni input values and no expected return values
lua_pcall	call Lua function func in protected mode, if f != 0 call function at position f
lua_cpcall	call C function in protected mode, if error != 0 return errobj
luaL_callmeta	call metatable of value at position n with field name if possible: v = n.__call(n)

21.11 Load or call Lua code

Load or call Lua code	Remark
lua_load	load a Lua chunk name by repeatedly calling reader(data), push the resulting function
luaL_loadbuffer	load a Lua chunk with name at given buffer and length, push the resulting function
luaL_dofile	load and run Lua file, push return values
luaL_dostring	load and run Lua chunk in memory, push return values
luaL_loadfile	load a Lua file, push the resulting function
luaL_loadstring	load a Lua chunk in memory, push the resulting function

21.12 Debugging

Debugging	Remark
ua_gethook	return hook function
ua_gethookcount	return hook count
ua_gethookmask	return hook mask
ua_sethook	set hook function, mask and count
ua_getinfo	return specific information, see manual for details
ua_getlocal	get information for local variable
ua_setlocal	pop and store as value of local variable n
ua_getupvalue	get information for upvalue n in function at position f
ua_setupvalue	pop and store as value of upvalue n in function at position f
ua_dump	dump function as binary data to writer
ua_error	generates a Lua error, never returns
ua_getstack	get information of runtime stack at level n
ua_checkstack	ensure remaining stack space of at least n values

ua_type	return Lua type of value at position n
ua_typename	return typename of type number t
ua_atpanic	set panic function and return previous one
uaL_argcheck	If cond is not true: raise argument error with text based on n and msg
uaL_argerror	raise argument error with text based on n and msg
uaL_typerror	raise type error with text based on n and name
uaL_error	raise error with a sprintf() formatted message
uaL_checkstack	ensure remaining stack space of at least n values, raise error with text including msg
uaL_where	push a string describing the current programm position

21.13 Buffer

Buffer	Remark
luaL_buffinit	initialise a buffer
luaL_prepbuffer	return intermediate space
luaL_addvalue	pop value and copy resulting string to buffer
luaL_addchar	add character c to buffer
luaL_addlstring	add string with len l to buffer
luaL_addstring	add C string to buffer
luaL_addsize	add intermediate space with given size to buffer
luaL_pushresult	finish buffer and push result

21.14 Thread

Thread	Remark
lua_yield	suspend a coroutine
lua_resume	resume a coroutine
lua_status	return status of thread L

21.15 Library

Library	Remark
lua_close	close Lua library
luaopen_base	open base library
luaopen_debug	open debug library
luaopen_io	open io library
luaopen_math	open math library
luaopen_os	open os library
luaopen_package	open package library
luaopen_string	open string library
luaopen_table	open table library

luaL_openlibs	open all the above standard libraries

21.16 Misc

Misc	Remark
lua_newthread	create and push a new thread
lua_newuserdata	allocate and push user data with given size
lua_newstate	create a new Lua state with given allocator and user data
luaL_newstate	create a new Lua state with defaults
lua_gc	control garbage collector
lua_getallocf	get memory allocator and user data
lua_setallocf	set memory allocator and user data
luaL_ref	create unique key in table at position n
luaL_unref	release key in table at position n

luaL_findtable	deprecated:
luaL_setn	deprecated:
luaL_getn	deprecated:
luaL_openlib	open

21.17 Basic types

Basic types	*Value*	*Type name*
LUA_TNONE	(-1)	n/a
LUA_TNIL	0	nil
LUA_TBOOLEAN	1	boolean
LUA_TLIGHTUSERDATA	2	n/a
LUA_TNUMBER	3	number
LUA_TSTRING	4	string
LUA_TTABLE	5	table
LUA_TFUNCTION	6	function
LUA_TUSERDATA	7	n/a
LUA_TTHREAD	8	thread

22 Add-on

Visual Studio - Lua Language Support

http://vslua.codeplex.com/

This addon for Visual Studio 2008 allows for syntax coloring, and error checking of the Lua script language.

NEW: Visual Studio 2010 support has been added!