(http://twitter.com/kenlistdev) (http://github.com/kenlist)

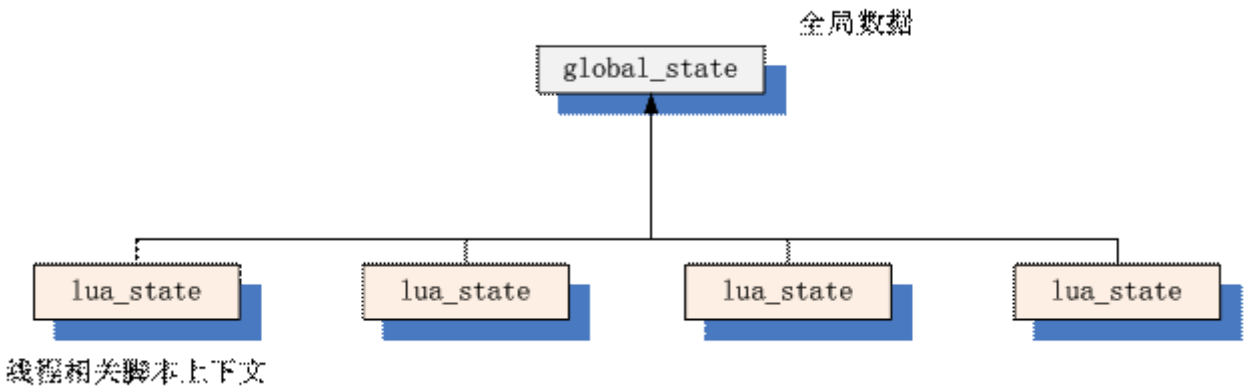
(http://geekluo.com/) (mailto:rijian.lrj@alibaba-inc.com)

Lua数据结构 -- lua_State（六）

2014/04/12

前面各种Lua的数据类型基本都说得差不多了，剩下最后一个数据类型：**lua_State**，我们可以认为是**"脚本上下文"**，主要是包括当前脚本环境的运行状态信息，还会有gc相关的信息。

Lua这门语言考虑了多线程的情况，在脚本空间中能够开多个线程相关脚本上下文，而大家会共用一个全局脚本状态数据，如下：



全局数据global_state的数据结构如下：

```
typedef struct global_State {
  stringtable strt;  /* hash table for strings */
  lua_Alloc frealloc;  /* function to reallocate memory */
  void *ud;         /* auxiliary data to `frealloc' */
  lu_byte currentwhite;
  lu_byte gcstate;  /* state of garbage collector */
  int sweepstrgc;  /* position of sweep in `strt' */
  GCObject *rootgc;  /* list of all collectable objects */
  GCObject **sweepgc;  /* position of sweep in `rootgc' */
  GCObject *gray;  /* list of gray objects */
  GCObject *grayagain;  /* list of objects to be traversed atomically */
  GCObject *weak;  /* list of weak tables (to be cleared) */
  GCObject *tmudata;  /* last element of list of userdata to be GC */
  Mbuffer buff;  /* temporary buffer for string concatentation */
  lu_mem GCthreshold;
  lu_mem totalbytes;  /* number of bytes currently allocated */
  lu_mem estimate;  /* an estimate of number of bytes actually in use */
  lu_mem gcdept;  /* how much GC is `behind schedule' */
  int gcpause;  /* size of pause between successive GCs */
  int gcstepmul;  /* GC `granularity' */
  lua_CFunction panic;  /* to be called in unprotected errors */
  TValue l_registry;
  struct lua_State *mainthread;
  UpVal uvhead;  /* head of double-linked list of all open upvalues */
  struct Table *mt[NUM_TAGS];  /* metatables for basic types */
  Tstring *tmname[TM_N];  /* array with tag-method names */
} global_State;
```

global_state主要是用于GC的数据链表，下面简要说明几个:

1. stringtable strt：这个是在TString那章说到的全局字符串哈希表
2. TValue l_registry：对应LUA_REGISTRYINDEX的全局table.
3. TString *tmname[TM_N]：元方法的名称字符串。
4. Table *mt[NUM_TAGS]：基本类型的元表，这是Lua5.0的特性。

mt成员在作者介绍文章中说到:

```
It only takes a few more lines of code to add this support, and what a world of difference it
 makes.
```

这个特性其实非常有用，我们来看看下面例子：

```
local a = 33
local b = a.tostring()
```

在上面代码中，我们看到a支持一个tostring的方法，a是数值类型，我们可以为数值类型添加任意的方法。Lua文章中说到一个用途，就是对于unicode和gbk的字符串的len方法能自己实现。

其它成员就不一一介绍了，下面来介绍与线程相关的脚本上下文lua_State：

```
struct lua_State {
  CommonHeader;
  lu_byte status;
  StkId top;  /* first free slot in the stack */
  StkId base;  /* base of current function */
  global_State *l_G;
  CallInfo *ci;  /* call info for current function */
  const Instruction *savedpc;  /* `savedpc' of current function */
  StkId stack_last;  /* last free slot in the stack */
  StkId stack;  /* stack base */
  CallInfo *end_ci;  /* points after end of ci array*/
  CallInfo *base_ci;  /* array of CallInfo's */
  int stacksize;
  int size_ci;  /* size of array `base_ci' */
  unsigned short nCcalls;  /* number of nested C calls */
  unsigned short baseCcalls;  /* nested C calls when resuming coroutine */
  lu_byte hookmask;
  lu_byte allowhook;
  int basehookcount;
  int hookcount;
  lua_Hook hook;
  TValue l_gt;  /* table of globals */
  TValue env;  /* temporary place for environments */
  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) */
};
```

我们看到，lua_State也带有CommonHeader头，在第一章中也提到了GCObject中有lua_State th这个成员，由此可见lua_State也会是被回收的对象之一。

考虑回一个线程中的脚本上下文，我们再来逐个分析每个成员：

- lu_byte status：线程脚本的状态，线程可选状态如下：

```
/* thread status; 0 is OK */
#define LUA_YIELD         1
#define LUA_ERRRUN        2
#define LUA_ERRSYNTAX     3
#define LUA_ERRMEM        4
#define LUA_ERRERR        5
```

0：表示线程正在正常运行。
LUA_YIELD：表明线程处于挂起的状态，整个线程上下文会被挂起，可以通过调用lua_yield来挂起线程。
LUA_ERRRUN：所有运行错误发生之后，线程状态会变成LUA_ERRRUN。
LUA_ERRSYNTAX：语法分析错误后，线程状态会变成LUA_ERRSYNTAX。
LUA_ERRMEM：内存分配错误后，线程状态会变成LUA_ERRMEM。
LUA_ERRERR：其它一些溢出的错误会使线程变成LUA_ERRERR状态。

- StkId top：指向当前线程栈的栈顶指针，typedef TValue *StkId
- StkId base：指向当前函数运行的相对基位置，具体可参考第四章的闭包
- global_State *l_G：指向全局状态的指针
- CallInfo *ci：当前线程运行的函数调用信息
- const Instruction *savedpc：函数调用前，记录上一个函数的pc位置

- StkId stack_last：栈的实际最后一个位置（栈的长度是动态增长的）
- StkId stack：栈底
- CallInfo *end_ci：指向函数调用栈的栈顶
- CallInfo *base_ci：指向函数调用栈的栈底
- int stacksize：栈的大小
- int size_ci：函数调用栈的大小
- unsigned short nCcalls：当前C函数的调用的深度
- unsigned short baseCcalls：用于记录每个线程状态的C函数调用深度的辅助成员
- lu_byte hookmask：支持哪些hook能力，有下列可选的

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

LUA_MASKCALL：每当调用函数前一瞬间，都会先调用用户注册的hook方法
LUA_MASKRET：在Lua正要离开函数一瞬间，调用用户注册的hook方法
LUA_MASKLINE:每执行一行代码前，都会先执行用户注册的hook方法
LUA_MASKCOUNT：可以设置执行多少条指令后调用，调用用户注册的hooK方法
```

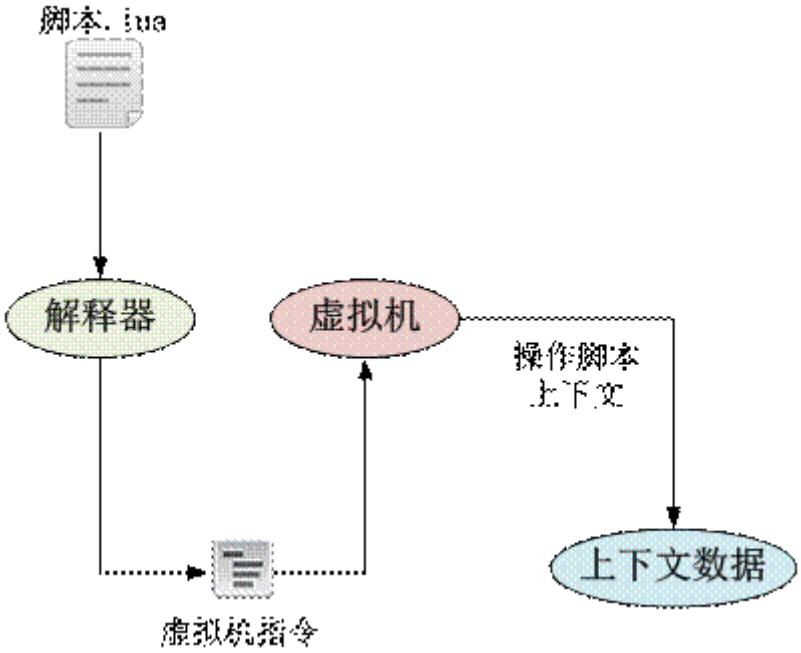
- lu_byte allowhook：是否允许hook
- int basehookcount：用户设置的执行指令数(LUA_MASKCOUNT下有效)
- int hookcount：运行时，跑了多少条指令（LUA_MASKCOUNT下有效）
- lua_Hook：用户注册的hook回调函数
- TValue l_gt：当前线程的全局的环境表
- TValue env：当前运行的环境表
- GCObject *openupval、gclist：用于gc，详细将会在GC一章细说
- struct lua_longjmp *errorJmp：发生错误的长跳转位置，用于记录当函数发生错误时跳转出去的位置。

```
/* chain list of long jump buffers */
struct lua_longjmp {
                struct lua_longjmp *previous;
                luai_jmpbuf b;
                volatile int status;  /* error code */
};
ptrdiff_t errfunc：用户注册的错误回调函数
```

本系列总结：

整个系列文章回答了我们对Lua中最基本的一个问题：“一个Lua变量究竟是什么？”。由此我们深入并引申出各种知识，在脚本中我们觉得弱类型变量用起来很痛快，而其实它的内部实现其实是如此的复杂。

对于实现一门脚本语言，必须实现的是解释器、虚拟机、上下文数据3大部分：



上下文数据这一层是脚本最基础，最底层的东西，它决定了这门脚本究竟能做什么。抛开解释器和虚拟机，我们依然可以单纯地通过C接口，在C++这一层就能操作脚本的上下文数据。

有空再研究一下Lua的GC，解释器等等。

lua (6) (http://geekluo.com/tags.html#lua-ref)

← Previous (http://geekluo.com/contents/2014/04/12/5-lua-closure-structure.html)

Next → (http://geekluo.com/contents/2014/04/12/7-lua-udata-structure.html)