

Lua 源码欣赏

云风

2013 年 4 月 17 日

注

这是这本书中其中部分章节的草稿。我不打算按顺序来编写这本书，而打算以独立章节的形式分开完成，到最后再做统一的调整。由于业余时间不多，所以产出也不固定。

目录

第一章 概览	1
1.1 源文件划分	1
1.2 代码风格	2
1.3 Lua 核心	3
1.4 代码翻译及预编译字节码	3
1.5 内嵌库	4
1.6 独立解析器及字节码编译器	4
1.7 阅读源代码的次序	4
第二章 全局状态机及内存管理	7
2.1 内存管理	7
2.2 全局状态机	10
2.2.1 garbage collect	12
2.2.2 seed	13
2.2.3 buff	13
2.2.4 version	13
2.2.5 防止初始化的意外	14
第三章 字符串	17
3.1 数据结构	17
3.1.1 Hash DoS	18
3.2 实现	20
3.2.1 字符串比较	20
3.2.2 短字符串的内部化	20
3.3 Userdata 的构造	22
第四章 表	25
4.1 数据结构	25
4.2 算法	27
4.2.1 短字符串优化	29
4.2.2 数字类型的哈希值	31
4.3 表的迭代	33
4.4 对元方法的优化	36

4.4.1	类型名字	37
第五章	函数与闭包	39
5.1	函数原型	39
5.2	Upvalue	41
5.3	闭包	42
5.3.1	Lua 闭包	43
5.3.2	C 闭包	46
5.3.3	轻量 C 函数	46
第六章	协程及函数的执行	49
6.1	栈与调用信息	49
6.1.1	数据栈	50
6.1.2	调用栈	54
6.1.3	线程	56
6.2	线程的执行与中断	58
6.2.1	异常处理	58
6.2.2	函数调用	60
6.2.3	钩子	66
6.2.4	从 C 函数中挂起线程	67
6.2.5	挂起与延续	68
6.2.6	lua_callk 和 lua_pcallk	73
6.2.7	异常处理	76
第七章	虚拟机	77
7.1	指令结构	77
7.1.1	常量	78
7.1.2	操作码分类校验	79
7.1.3	操作码列表	80
7.2	字节码的运行	81
7.2.1	luaV_execute	81
7.2.2	寄存器赋值	83
7.2.3	表处理	86
7.2.4	表达式运算	89
7.2.5	分支和跳转	95
7.2.6	函数调用	99
7.2.7	不定长参数	101
7.2.8	生成闭包	103
7.2.9	For 循环	106
7.2.10	协程的中断和延续	108

第八章	内置库的实现	111
8.1	从 math 模块看 Lua 的模块注册机制	111
8.2	math 模块 API 的实现	113
8.3	string 模块	115
8.4	暂且搁置	116

第一章 概览

Lua 是一门编程语言，Lua 官方网站¹提供了由语言发明者实现的官方版本²。虽然 Lua 有简洁清晰的语言标准，但我们不能将语言的标准制定和实现完全分开看待。事实上，随着官方实现版本的不断更新，Lua 语言标准也在不断变化。

本书试图向读者展现 Lua 官方实现的细节。在开始前，先从宏观上来看，实现这门语言需要完成那些部分的工作。

Lua 作为一门动态语言，提供了一个虚拟机。Lua 代码最终都是以字节码的形式由虚拟机解释执行的。把外部组织好的代码置入内存，让虚拟机解析运行，需要有一个源代码解释器，或是预编译字节码的加载器。而只实现语言特性，几乎做不了什么事。所以 Lua 的官方版本还提供了一些库，并提供一系列 C API，供第三方开发。这样，Lua 就可以借助这些外部库，做一些我们需要的工作。

下面，我们按这个划分来分拆解析。

1.1 源文件划分

从官网下载到 Lua 5.2 的源代码后³，展开压缩包，会发现源代码文件全部放在 src 子目录下。这些文件根据实现功能的不同，可以分为四部分。⁴

1. 虚拟机运转的核心功能

lapi.c C 语言接口

lctype.c C 标准库中 ctype 相关实现

ldebug.c Debug 接口

ldo.c 函数调用以及栈管理

lfunc.c 函数原型及闭包管理

lgc.c 垃圾回收

lmem.c 内存管理接口

lobject.c 对象操作的一些函数

lopcodes.c 虚拟机的字节码定义

lstate.c 全局状态机

¹Lua 是一个以 MIT license 发布的开源项目，你可以自由的下载、传播、使用它。它的官方网站是：<http://www.lua.org>

² Lua 官方实现并不是对 Lua 语言的唯一实现。另外比较流行的 Lua 语言实现还有 LuaJIT (<http://www.luajit.org>)。由于采用了 JIT 技术，运行性能更快。除此之外，还能在互联网上找到其它一些不同的实现。

³本书讨论的 Lua 5.2.2 版可以在 <http://www.lua.org/ftp/lua-5.2.2.tar.gz> 下载获得

⁴在 Lua Wiki 上有一篇文章介绍了 Lua 源代码的结构：<http://lua-users.org/wiki/LuaSource>

lstring.c 字符串池

ltable.c 表类型的相关操作

ltm.c 元方法

lvm.c 虚拟机

lzio.c 输入流接口

2. 源代码解析以及预编译字节码

lcode.c 代码生成器

ldump.c 序列化预编译的 Lua 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

3. 内嵌库

lauxlib.c 库编写用到的辅助函数库

lbaselib.c 基础库

lbitlib.c 位操作库

lcorolib.c 协程库

ldblib.c Debug 库

linit.c 内嵌库的初始化

liolib.c IO 库

lmathlib.c 数学库

loadlib.c 动态扩展库管理

loslib.c OS 库

lstrlib.c 字符串库

ltablib.c 表处理库

4. 可执行的解析器，字节码编译器

lua.c 解释器

luac.c 字节码编译器

1.2 代码风格

Lua 使用 Clean C [5]⁵ 编写的源代码模块划分清晰，大部分模块被分解在不同的 .c 文件中实现，以同名的 .h 文件描述模块导出的接口。比如，lstring.c 实现了 Lua 虚拟机中字符串池的相关功能，而这部分的内部接口则在 lstring.h 中描述。

⁵Clean C 是标准 C/C++ 的一个子集。它只包含了 C 语言中的一些必要特性。这样方便把 Lua 发布到更多的可能对 C 语言支持不完整的平台上。比如，对于没有 ctype.h 的 C 语言编译环境，Lua 提供了 lctype.c 实现了一些兼容函数。

它使用的代码缩进风格比较接近 K&R 风格⁶，并有些修改，例如函数定义的开花括号没有另起一行。同时，也掺杂了一些 GNU 风格，比如采用了双空格缩进、在函数名和小括号间加有空格。代码缩进风格没有好坏，但求统一。

Lua 的内部模块暴露出来的 API 以 luaX_xxx 风格命名，即 lua 后跟一个大写字母标识内部模块名，而后由下划线加若干小写字母描述方法名。

供外部程序使用的 API 则使用 lua_xxx 的命名风格。这些在 Lua 的官方文档里有详细描述。定义在 lua.h 文件中。此外，除了供库开发用的 luaL 系列 API（定义在 lauxlib.h 中）外，其它都属于内部 API，禁止被外部程序使用⁷。

1.3 Lua 核心

Lua 核心部分仅包括 Lua 虚拟机的运转。Lua 虚拟机的行为是由一组组 opcode 控制的。这些 opcode 定义在 lopcodes.h 及 lopcodes.c 中。而虚拟机对 opcode 的解析和运作在 lvm.c 中，其 API 以 luaV 为前缀。

Lua 虚拟机的外在数据形式是一个 lua_State 结构体，取名 State 大概意为 Lua 虚拟机的当前状态。全局 State 引用了整个虚拟机的所有数据。这个全局 State 的相关代码放在 lstate.c 中，API 使用 luaE 为前缀。

函数的运行流程：函数调用及返回则放在 ldo.c 中，相关 API 以 luaD⁸ 为前缀。

Lua 中最复杂和重要的三种数据类型 function、table、string 的实现分属在 lfunc.c、ltable.c、lstring.c 中。这三组内部 API 分别以 luaF、luaH⁹、luaS 为前缀命名。不同的数据类型最终被统一定义为 Lua Object，相关的操作在 lobject.c 中，API 以 luaO 为前缀。

Lua 从第 5 版后增加了元表，元表的处理在 ltm.c 中，API 以 luaT 为前缀。

另外，核心系统还用两个基础设施：内存管理 lmem.c，API 以 luaM 为前缀；带缓冲的流处理 lzio.c，API 以 luaZ 为前缀。

最后是核心系统里最为复杂的部分，垃圾回收部分，在 lgc.c 中实现，API 以 luaC 为前缀。

Lua 设计的初衷之一就为了最好的和宿主系统相结合。它是一门嵌入式语言[5]，所以必须提供和宿主系统交互的 API。这些 API 以 C 函数的形式提供，大多数实现在 lapi.c 中。API 直接以 lua 为前缀，可供 C 编写的程序库直接调用。

以上这些就构成了让 lua 运转起来的最小代码集合。我们将在后面的章节来剖析其中的细节。

1.4 代码翻译及预编译字节码

光有核心代码和一个虚拟机还无法让 Lua 程序运行起来。因为必须从外部输入将运行的 Lua 程序。Lua 的程序的人读形式是一种程序文本，需要经过解析得到内部的数据结构（常量和 opcode 的集合）。这个过程是通过 parser：lparser.c（luaY¹⁰ 为前缀的 API）及词法分析 llex.c（luaX 为前缀的 API）。

解析完文本代码，还需要最终生成虚拟机理解的数据，这个步骤在 lcode.c 中实现，其 API 以 luaK 为前缀。

⁶K&R 风格将左花括号放在行尾，而右花括号独占一行。只对函数定义时有所例外。GNU 风格左右花括号均独占一行，且把 TAB 缩进改为两个空格[8]。不同的代码缩进风格还有许多其它细微差异。阅读不同开源项目的代码将会有所体会。如果要参于到某个开源项目的开发，通常应尊重该项目的代码缩进风格，新增和修改保持一致。

⁷理论上 luaL 系列 API 属于更高层次，对于 Lua 的第三方库开发也不是必须的。这些 API 全部由 Lua 标准 API 实现，没有使用任何其它内部 API。

⁸D 取 Do 之意。

⁹H 是意取 Hash 之意。

¹⁰Y 可能是取 yacc 的含义。因为 Lua 最早期版本的代码翻译是用 yacc 和 lex 这两个 Unix 工具实现的[4]，后来才改为手写解析器。

为了满足某些需求，加快代码翻译的流程。还可以采用预编译的过程。把运行时编译的结果，生成字节码。这个过程以及逆过程由 `ldump.c` 和 `lundump.c` 实现。其 API 以 `luaU` 为前缀。¹¹

1.5 内嵌库

作为嵌入式语言，其实完全可以不提供任何库及函数。全部由宿主系统注入到 `State` 中即可。也的确有许多系统是这么用的。但 Lua 的官方版本还是提供了少量必要的库。尤其是一些基础函数如 `pairs`、`error`、`setmetatable`、`type` 等等，完成了语言的一些基本特性，几乎很难不使用。

而 `coroutine`、`string`、`table`、`math` 等等库，也很常用。Lua 提供了一套简洁的方案，允许你自由加载你需要的部分，以控制最终执行文件的体积和内存的占用量。主动加载这些内建库进入 `lua_State`，是由在 `luaLib.h` 中的 API 实现的。¹²

在 Lua 5.0 之前，Lua 并没有一个统一的模块管理机制。这是由早期 Lua 仅仅定位在嵌入式语言决定的。这些年，由更多的人倾向于把 Lua 作为一门独立编程语言来使用，那么统一的模块化管理就变得非常有必要。这样才能让丰富的第三方库可以协同工作。即使是当成嵌入式语言使用，随着代码编写规模的扩大，也需要合理的模块划分。

Lua 5.1 引入了一个官方推荐的模块管理机制。使用 `require` / `module` 来管理 Lua 模块，并允许从 C 语言编写的动态库中加载扩展模块。这个机制被作者认为有点过度设计了[3]。在 Lua 5.2 中又有所简化。我们可以在 `loadlib.c` 中找到实现。内建库的初始化 API 则在 `limit.c` 中可以找到。

其它基础库可以在那些以 `lib.c` 为后缀的源文件中，分别找到它们的实现。

1.6 独立解析器及字节码编译器

Lua 在早期几乎都是被用来嵌入到其它系统中使用，所以源代码通常被编译成动态库或静态库被宿主系统加载或链接。但随着 Lua 的第三方库越来越丰富，人们开始倾向于把 Lua 作为一门独立语言来使用。Lua 的官方版本里也提供了一个简单的独立解析器，便是 `lua.c` 所实现的这个。并有 `luac.c` 实现了一个简单的字节码编译器，可以预编译文本的 Lua 源程序。¹³

1.7 阅读源代码的次序

Lua 的源代码有着良好的设计，优美易读。其整体篇幅不大，仅两万行 C 代码左右¹⁴。但一开始入手阅读还是有些许难度的。

从易到难，理清作者编写代码的脉络非常重要。LuaJIT 的作者 Mike Pall 在回答“哪一个开源代码项目设计优美，值得阅读不容错过”这个问题时，推荐了一个阅读次序¹⁵：

首先、阅读外围的库是如何实现功能扩展的，这样可以熟悉 Lua 公开 API。不必陷入功能细节。

然后、阅读 API 的具体实现。Lua 对外暴露的 API 可以说是一个对内部模块的一层封装，这个层次尚未触及核心，但可以对核心代码有个初步的了解。

¹¹ 极端情况下，我们还可以对 Lua 的源码做稍许改变，把 `parser` 从最终的发布版本中裁减掉，让虚拟机只能加载预编译好的字节码。这样可以减少执行代码的体积。Lua 的代码解析部分与核心部分之间非常独立，做到这一点所需修改极少。但这种做法并不提倡。

¹² 如果你静态链接 Lua 库，还可以通过这些 API 控制最终链入执行文件的代码体积。

¹³ 笔者倾向于在服务器应用中使用独立的 Lua 解析器。这样会更加灵活，可以随时切换其它 Lua 实现（例如采用性能更高的 LuaJIT），并可以方便的使用第三方库。

¹⁴ Lua 5.2.2 版本的源代码分布在 58 个文件中，共 20220 行 C 代码。

¹⁵ Ask Reddit: Which OSS codebases out there are so well designed that you would consider them 'must reads'? http://www.reddit.com/comments/63hth/ask_reddit_which_oss_codebases_out_there_are_so/c02pxbp

之后、可以开始了解 Lua VM 的实现。

接下来就是分别理解函数调用、返回，string、table、metatable 等如何实现。

debug 模块是一个额外的设施，但可以帮助你理解 Lua 内部细节。

最后是 parser 等等编译相关的部分。

垃圾收集将是最难的部分，可能会花掉最多的时间去理解细节。¹⁶

本书接下来的章节，并未按照以上次序来撰写。但大多数篇章能独立阅读，相互参考之处均有附注。读者可根据需要，自由选择阅读次序。

¹⁶笔者曾经就 Lua 5.1.4 的 gc 部分做过细致的剖析。相关文章可以在这里找到：http://blog.codingnow.com/2011/04/lua_gc_multithreading.html，在本书的后面，会重新领略 Lua 5.2.2 的实现。

第二章 全局状态机及内存管理

Lua 可以方便地被嵌入 C 程序中使用。

你可以很容易的创建出一个 Lua 虚拟机对象，不同的 Lua 虚拟机之间的工作是线程安全的，因为一切和虚拟机相关的内存操作都被关联到虚拟机对象中，而没有利用任何其它共享变量。

Lua 的虚拟机核心部分，没有任何的系统调用，是一个纯粹的黑盒子，正确的使用 Lua，不会对系统造成任何干扰。这其中最关键的一点是，Lua 让用户自行定义内存管理器，在创建 Lua 虚拟机时传入，这保证了 Lua 的整个运行状态是用户可控的。

2.1 内存管理

Lua 要求用户给出一个内存管理函数，在 Lua 创建虚拟机的时候传入。。

```
1 typedef void * (*lua_Alloc) (void *ud, void *ptr, size_t osize, size_t  
   nsize);  
2  
3 LUA_API lua_State *(lua_newstate) (lua_Alloc f, void *ud);
```

虽然许多时候，我们并不直接使用 lua_newstate 这个 API，而是用另一个更方便的版本 luaL_newstate。

但从 API 命名就可以看出，后者不输入核心 API，它是利用前者实现的。它利用 C 标准库中的函数实现了一个默认的内存管理器，这也可以帮助我们理解这个内存管理器的语义。

源代码 2.1: lauxlib.c: luaL_newstate

```
919 static void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {  
920     (void)ud; (void)osize; /* not used */  
921     if (nsize == 0) {  
922         free(ptr);  
923         return NULL;  
924     }  
925     else  
926         return realloc(ptr, nsize);  
927 }  
928  
929  
930 static int panic (lua_State *L) {
```

```

931     luaL_writestringerror("PANIC: unprotected error in call to Lua API (%s)\n",
932                           lua_tostring(L, -1));
933     return 0; /* return to Lua to abort */
934 }
935
936
937 LUALIB_API lua_State *luaL_newstate (void) {
938     lua_State *L = lua_newstate(l_alloc, NULL);
939     if (L) luaL_atpanic(L, &panic);
940     return L;
941 }

```

Lua 定义的内存管理器仅有一个函数，虽然接口上类似 `realloc` 但是和 C 标准库中的 `realloc` 有所区别。它需要在 `nsiz` 为 0 时，提供释放内存的功能。

和标准库中的内存管理接口不同，Lua 在调用它的时候会准确给出内存块的原始大小。对于需要为 Lua 定制一个高效的内存管理器来说，这个信息很重要。因为大多数内存管理的算法，都需要在释放内存的时候了解内存块的尺寸。而标准库的 `free` 和 `realloc` 函数却不给出这个信息。所以大多数内存管理模块在实现时，在内存块的前面加了一个 `cookie`，把内存块尺寸存放在里面。Lua 在实现时，刻意提供了这个信息。这样我们不必额外储存内存块尺寸，这对大量使用小内存块的环境可以节约不少内存。

另外，这个内存管理接口接受一个额外的指针 `ud`。这可以让内存管理模块工作在不同的堆上。恰当的定制内存管理器，就可以回避线程安全问题。不考虑线程安全的因素，我们可以让内存管理工作更为高效。

另一个可以辅助我们做出更有效的内存管理模块的机制是在 Lua 5.2 时增加的。当 `ptr` 传入 `NULL` 时，`osize` 的含义变成了对象的类型。这样，分配器就可以得知我们是在分配一个什么类型的对象，可以针对它做一些统计或优化工作。

Lua 使用了一组宏来管理不同类别的内存：单个对象、数组、可变长数组等。这组宏定义在 `lmem.h` 中。

源代码 2.2: `lmem.h`: memory

```

24 #define luaL_reallocv(L,b,on,n,e) \
25     (cast(void, \
26           (cast(size_t, (n)+1) > MAX_SIZET/(e)) ? (luaM_toobig(L), 0) : 0), \
27       luaL_realloc_(L, (b), (on)*(e), (n)*(e)))
28
29 #define luaM_freemem(L, b, s)    luaL_realloc_(L, (b), (s), 0)
30 #define luaM_free(L, b)          luaL_realloc_(L, (b), sizeof(*(b)), 0)
31 #define luaM_freearray(L, b, n)  luaL_reallocv(L, (b), n, 0, sizeof((b)
32                                   [0]))
33
34 #define luaM_malloc(L,s)         luaL_realloc_(L, NULL, 0, (s))
35 #define luaM_new(L,t)            cast(t *, luaL_malloc(L, sizeof(t)))
36 #define luaM_newvector(L,n,t) \

```



```

36         cast(t *, luaM_reallocv(L, NULL, 0, n, sizeof(t)))
37
38 #define luaM_newobject(L,tag,s) luaM_realloc_(L, NULL, tag, (s))
39
40 #define luaM_growvector(L,v,nelems,size,t,limit,e) \
41     if ((nelems)+1 > (size)) \
42         ((v)=cast(t *, luaM_growaux_(L,v,&(size),sizeof(t),limit,e)))
43
44 #define luaM_reallocvector(L, v,oldn,n,t) \
45     ((v)=cast(t *, luaM_reallocv(L, v, oldn, n, sizeof(t))))

```

这组宏的核心在一个内部 API：luaM_realloc_，它不会被直接调用。其实现在 lmem.c 中。它调用保存在 global_State 中的内存分配器管理内存。这些工作不仅仅是分配新的内存，释放不用的内存，扩展不够用的内存。Lua 也会通过 realloc 试图释放掉预申请过大的内存的后半部分，当然，这取决于用户提供的内存管理器能不能缩小内存块了。之所以用宏来实现这组 API，是因为内存管理会被高频调用，而 luaM_reallocv 这样的 API 的传入参数中经常出现常数，实现为宏可以保证常量计算在编译时进行。

源代码 2.3: lmem.c: realloc

```

75 void *luaM_realloc_ (lua_State *L, void *block, size_t osize, size_t nsize)
76 {
77     void *newblock;
78     global_State *g = G(L);
79     size_t realsize = (block) ? osize : 0;
80     lua_assert((realsize == 0) == (block == NULL));
81 #if defined(HARDMEMTESTS)
82     if (nsize > realsize && g->gcrunning)
83         luaC_fullgc(L, 1); /* force a GC whenever possible */
84 #endif
85     newblock = (*g->frealloc)(g->ud, block, osize, nsize);
86     if (newblock == NULL && nsize > 0) {
87         api_check(L, nsize > realsize,
88             "realloc cannot fail when shrinking a block");
89         if (g->gcrunning) {
90             luaC_fullgc(L, 1); /* try to free some memory... */
91             newblock = (*g->frealloc)(g->ud, block, osize, nsize); /* try again
92             */
93         }
94         if (newblock == NULL)
95             luaD_throw(L, LUA_ERRMEM);
96     }
97     lua_assert((nsize == 0) == (newblock == NULL));
98     g->GCdebt = (g->GCdebt + nsize) - realsize;

```

```

97     return newblock;
98 }

```

从代码中可以看到, `luaM_realloc_` 根据传入的 `osize` 和 `nsize` 调整内部感知的内存大小 (设置 `GCdebt`), 在内存不够用的时候会主动尝试做 GC 操作。

`lmem.c` 中还有另一个内部 API `luaM_growaux_`, 它是用来管理可变长数组的。其主要策略是当数组空间不够时, 扩大为原来的两倍。

源代码 2.4: `lmem.c: growvector`

```

46 void *luaM_growaux_ (lua_State *L, void *block, int *size, size_t
    size_elems,
47                     int limit, const char *what) {
48     void *newblock;
49     int newsize;
50     if (*size >= limit/2) { /* cannot double it? */
51         if (*size >= limit) /* cannot grow even a little? */
52             luaG_runerror(L, "too many %s (limit is %d)", what, limit);
53         newsize = limit; /* still have at least one free place */
54     }
55     else {
56         newsize = (*size)*2;
57         if (newsize < MINSIZEARRAY)
58             newsize = MINSIZEARRAY; /* minimum size */
59     }
60     newblock = luaM_reallocv(L, block, *size, newsize, size_elems);
61     *size = newsize; /* update only when everything else is OK */
62     return newblock;
63 }

```

```

1 #define luaM_growvector(L,v,nelems,size,t,limit,e) \
2     if ((nelems)+1 > (size)) \
3         ((v)=cast(t *, luaM_growaux_(L,v,&(size),sizeof(t),limit,e)))

```

2.2 全局状态机

从 Lua 的使用者的角度看, `global_State` 是不可见的。我们无法用公开的 API 取到它的指针, 也不需要引用它。但分析 Lua 的实现就不能绕开这个部分。

`global_State` 里面有对主线程的引用, 有注册表管理所有全局数据, 有全局字符串表, 有内存管理函数, 有 GC 需要的把所有对象串联起来的相关信息, 以及一切 Lua 在工作时需要的工作内存。

通过 `lua_newstate` 创建一个新的 Lua 虚拟机时, 第一块申请的内存将用来保存主线程和这个全局状态机。Lua 的实现尽可能的避免内存碎片, 同时也减少内存分配和释放的次数。它采用了一个小技巧, 利用一个 LG 结构, 把主线程 `lua_State` 和 `global_State` 分配在一起。

源代码 2.5: lstate.c: LG

```

59 typedef struct LX {
60 #if defined(LUAI_EXTRASPACE)
61     char buff[LUAI_EXTRASPACE];
62 #endif
63     lua_State l;
64 } LX;
65
66
67 /*
68 ** Main thread combines a thread state and the global state
69 */
70 typedef struct LG {
71     LX l;
72     global_State g;
73 } LG;

```

这里，主线程必须定义在结构的前面，否则关闭虚拟机的时候（如下代码）就无法正确的释放内存。

```

1 (*g->frealloc)(g->ud, fromstate(L), sizeof(LG), 0); /* free main block */

```

这里的 LG 结构是放在 C 文件内部定义的，而不存在公开的 H 文件中，是仅供这一个 C 代码了解的知识，所以这种依赖数据结构内存布局的用法负作用不大。

lua_newstate 这个公开 API 定义在 lstate.c 中，它初始化了所有 global_State 中将引用的数据。阅读它的实现可以了解全局状态机中到底包含了哪些东西。

源代码 2.6: lstate.c: lua_newstate

```

262 LUA_API lua_State *lua_newstate (lua_Alloc f, void *ud) {
263     int i;
264     lua_State *L;
265     global_State *g;
266     LG *l = cast(LG *, (*f)(ud, NULL, LUA_TTHREAD, sizeof(LG)));
267     if (l == NULL) return NULL;
268     L = &l->l.l;
269     g = &l->g;
270     L->next = NULL;
271     L->tt = LUA_TTHREAD;
272     g->currentwhite = bit2mask(WHITE0BIT, FIXEDBIT);
273     L->marked = luaC_white(g);
274     g->gckind = KGC_NORMAL;
275     preinit_state(L, g);
276     g->frealloc = f;
277     g->ud = ud;

```

```

278 g->mainthread = L;
279 g->seed = makeseed(L);
280 g->uvhead.u.l.prev = &g->uvhead;
281 g->uvhead.u.l.next = &g->uvhead;
282 g->gcrunning = 0; /* no GC while building state */
283 g->GCestimate = 0;
284 g->strt.size = 0;
285 g->strt.nuse = 0;
286 g->strt.hash = NULL;
287 setnilvalue(&g->l_registry);
288 luaZ_initbuffer(L, &g->buff);
289 g->panic = NULL;
290 g->version = lua_version(NULL);
291 g->gcstate = GCSpause;
292 g->allgc = NULL;
293 g->finobj = NULL;
294 g->tobefnz = NULL;
295 g->sweepgc = g->sweepfin = NULL;
296 g->gray = g->grayagain = NULL;
297 g->weak = g->ephemeron = g->allweak = NULL;
298 g->totalbytes = sizeof(LG);
299 g->GCdebt = 0;
300 g->gcpause = LUAI_GCPAUSE;
301 g->gcmajorinc = LUAI_GCMAJOR;
302 g->gcstepmul = LUAI_GCMUL;
303 for (i=0; i < LUA_NUMTAGS; i++) g->mt[i] = NULL;
304 if (luaD_rawrunprotected(L, f_luaopen, NULL) != LUA_OK) {
305     /* memory allocation error: free partial state */
306     close_state(L);
307     L = NULL;
308 }
309 else
310     luai_userstateopen(L);
311 return L;
312 }

```

2.2.1 garbage collect

大部分内容和 GC 有关。因为内存管理和不用的内存回收都是基于整个虚拟机的，采用根扫描的 GC 算法的 LUA 实现自然把所有 GC 管理下内存的根与 GC 过程的关联信息都保存在了这里。有关 GC 的代码是 Lua 源码中最复杂的部分，将单列一章来分析。

2.2.2 seed

```
1 g->seed = makeseed(L);
```

seed 和字符串的 hash 算法相关，参见 3.1.1 节。

2.2.3 buff

```
1 luaZ_initbuffer(L, &g->buff);
```

buff 用于 Lua 源代码文本的解析过程，以及字符串处理需要的临时空间。Lua 的单个 State 工作在单线程模式下，所以在内部需要时，总是重复利用这个指针指向的临时空间。

2.2.4 version

```
1 g->version = lua_version(NULL);
```

这是 Lua 5.2 新增加的特性。

让我们先看一下 lua_version 的实现。

源代码 2.7: lapi.c: lua_version

```
137 LUA_API const lua_Number *lua_version (lua_State *L) {
138     static const lua_Number version = LUA_VERSION_NUM;
139     if (L == NULL) return &version;
140     else return G(L)->version;
141 }
```

这里把一个全局变量地址赋给了 version 域。这样可以起到一个巧妙的作用：用户可以在运行时获得传入参数 L 中的 version 地址，与自身链入的 lua 虚拟机实现代码中的这个变量，知道创建这个 L 的代码是否和自身的代码版本是否一致。这是比版本号比较更强的检查。

我们再看看 luaL_checkversion 的实现就能明白 lua_version 这个 API 为何不返回一个版本号数字，而是给出一个保存了版本号数字的内存指针了。¹

```
1 #define luaL_checkversion(L)    luaL_checkversion_(L, LUA_VERSION_NUM)
```

源代码 2.8: lauxlib.c: luaL_checkversion

```
944 LUALIB_API void luaL_checkversion_ (lua_State *L, lua_Number ver) {
945     const lua_Number *v = lua_version(L);
946     if (v != lua_version(NULL))
947         luaL_error(L, "multiple_Lua_VMs_detected");
948     else if (*v != ver)
949         luaL_error(L, "version_mismatch: app_needs_%f, Lua_core_provides_%f",
```

¹luaL_checkversion 一个重要的功能是检测多重链入的 lua 虚拟机。这固然是错误使用 lua 造成的，并非 lua 自身的缺陷。但把 lua 作用一门嵌入式语言而不是独立语言使用是 lua 的设计初衷，这个 bug 非常常见且不易发觉，所以给出一个 api 来检测错误的发生是个不错的设计。具体分析可以参考笔者的一篇 blog http://blog.codingnow.com/2012/01/lua_link_bug.html。同时在 4.1 节也有进一步的讨论。

```

950         ver, *v);
951  /* check conversions number -> integer types */
952  lua_pushnumber(L, -(lua_Number)0x1234);
953  if (lua_tointeger(L, -1) != -0x1234 ||
954      lua_tounsigned(L, -1) != (lua_Unsigned)-0x1234)
955      luaL_error(L, "bad conversion number->int;"
956                  " must recompile Lua with proper settings");
957  lua_pop(L, 1);
958 }

```

2.2.5 防止初始化的意外

Lua 在处理虚拟机创建的过程非常的小心。

由于内存管理器是外部传入的，不可能保证它的返回结果。到底有多少内存可供使用也是未知数。为了保证 Lua 虚拟机的健壮性，需要检查所有的内存分配结果。Lua 自身有完整的异常处理机制可以处理这些错误。所以 Lua 的初始化过程是分两步进行的，首先初始化不需要额外分配内存的部分，把异常处理机制先建立起来，然后去调用可能引用内存分配失败导致错误的初始化代码。

```

1  if (luaD_rawrunprotected(L, f_luaopen, NULL) != LUA_OK) {
2      /* memory allocation error: free partial state */
3      close_state(L);
4      L = NULL;
5  }
6  else
7      lua_i_userstateopen(L);

```

在 6.2.1 节，会展示 luaD_rawrunprotected 怎样截获内存分配的异常情况。我们现在仅关注 f_luaopen 函数。

源代码 2.9: lstate.c: f_luaopen

```

183 static void f_luaopen (lua_State *L, void *ud) {
184     global_State *g = G(L);
185     UNUSED(ud);
186     stack_init(L, L); /* init stack */
187     init_registry(L, g);
188     luaS_resize(L, MINSTRTABSIZE); /* initial size of string table */
189     luaT_init(L);
190     luaX_init(L);
191     /* pre-create memory-error message */
192     g->memerrmsg = luaS_newliteral(L, MEMERRMSG);
193     luaS_fix(g->memerrmsg); /* it should never be collected */
194     g->gcrunning = 1; /* allow gc */
195 }

```

这里初始化了主线程的数据栈²、初始化注册表、给出一个基本的字符串池³、初始化元表用的字符串⁴、初始化词法分析用的 token 串⁵、初始化内存错误信息。

²主线程是独立于其它线程的创建过程直接创建出来的。并没有调用 `lua_newthread`。

³`luaS_resize` 参见3.2.2 节

⁴`luaT_init` 参见4.4 节

⁵`luaX_init`。

第三章 字符串

Lua 中的字符串可以包含任何 8 位字符，包括了 C 语言中标示字符串结束的 `\0`。它以带长度的内存块的形式在内部保存字符串，同时在和 C 语言做交互时，又能保证在每个内部储存的字符串末尾添加 `\0` 以兼容 C 库函数。这使得 Lua 的字符串应用范围相当广。Lua 管理及操作字符串的方式和 C 语言不太相同，通过阅读其实现代码，可以加深对 Lua 字符串的理解，能更为高效地使用它。

3.1 数据结构

从 Lua 5.2.1 开始，字符串保存在 Lua 状态机内有两种内部形式，短字符串及长字符串。

源代码 3.1: lobject.h: variant string

```
55 /* Variant tags for strings */
56 #define LUA_TSHRSTR      (LUA_TSTRING | (0 << 4)) /* short strings */
57 #define LUA_TLNGSTR      (LUA_TSTRING | (1 << 4)) /* long strings */
```

这个小类型区分放在类型字节的高四位，所以为外部 API 所不可见。对于最终用户来说，他们只见到 `LUA_TSTRING` 一种类型。区分长短字符串的界限由定义在 `luaconf.h` 中的宏 `LUAI_MAXSHORTLEN` 来决定。其默认设置为 40 字节。由 Lua 的实现决定了，`LUAI_MAXSHORTLEN` 不可以设置少于 10 字节¹。

字符串一旦创建，则不可被改写。Lua 的值对象若为字符串类型，则以引用方式存在。属于需被垃圾收集器管理的对象。也就是说，一个字符串一旦被任何地方引用就可以回收它。

字符串类型 `TString` 定义在 `lobject.h` 里。

源代码 3.2: lobject.h: tstring

```
410 typedef union TString {
411     L_Umaxalign dummy; /* ensures maximum alignment for strings */
412     struct {
413         CommonHeader;
414         lu_byte extra; /* reserved words for short strings; "has hash" for
                        longs */
415         unsigned int hash;
416         size_t len; /* number of characters in string */
417     } tsv;
418 } TString;
```

¹元方法名和保留字必须是短字符串，所以短字符串长度不得短于最长的元方法名 `__newindex` 和保留字 `function`。

除了用于 GC 的 CommonHeader 外，有 extra hash 和 len 三个域。extra 用来记录辅助信息² 记录字符串的 hash 可以用来加快字符串的匹配和查找；由于 Lua 并不以 \0 结尾来识别字符串的长度，故需要一个 len 域来记录其长度。

字符串的数据内容并没有被分配独立一块内存来保存，而是直接加在 TString 结构的后面。用 getstr 这个宏就可以取到实际的 C 字符串指针。

源代码 3.3: lobject.h: getstr

```
422 #define getstr(ts)      cast(const char *, (ts) + 1)
```

所有短字符串均被存放在全局表（global_State）的 strt 域中。strt 是 string table 的简写，它是一个哈希表。

源代码 3.4: lstate.h: stringtable

```
59 typedef struct stringtable {
60     GCObject **hash;
61     lu_int32 nuse; /* number of elements */
62     int size;
63 } stringtable;
```

相同的短字符串在同一个 Lua State 中只存在唯一一份，这被称为字符串的内部化³。

长字符串则独立存放，从外部压入一个长字符串时，简单复制一遍字符串，并不立刻计算其 hash 值，而是标记一下 extra 域。直到需要对字符串做键匹配时，才惰性计算 hash 值，加快以后的键比较过程⁴。

3.1.1 Hash DoS

在 Lua 5.2.0 之前，字符串是不分长短一律内部化后放在字符串表中的。

对于长字符串，为了加快内部化的过程，计算长字符串哈希值是跳跃进行的。下面是 Lua 5.2.0 中的字符串哈希值计算代码：

```
1 unsigned int h = cast(unsigned int, 1); /* seed */
2 size_t step = (1>>5)+1; /* if string is too long, don't hash all its
   chars */
3 size_t l1;
4 for (l1=1; l1<=step; l1+=step) /* compute hash */
5     h = h ^ ((h<<5)+(h>>2)+cast(unsigned char, str[l1-1]));
```

Lua 5.2.0 发布后不久，有人在邮件列表中提出，Lua 的这个设计有可能对其给予 Hash DoS 攻击的机会⁵。攻击者可以轻易构造出上千万拥有相同哈希值的不同字符串，以此数十倍的降低 Lua 从外部压入字符串进入内部字符串表的效率[6]。当 Lua 用于大量依赖字符串处理的诸如 HTTP 服务的处理时，输入的字符串不可控制，很容易被人恶意利用。

²对于短字符串 extra 用来记录这个字符串是否为保留字，这个标记用于词法分析器对保留字的快速判断；对于长字符串，可以用于惰性求哈希值。

³合并相同的字符串可以大量减少内存占用，缩短比较字符串的时间。因为相同的字符串只需要保存一份在内存中，当用这个字符串做键匹配时，比较字符串只需要比较地址是否相同就够了，而不必逐字节比较。

⁴关于长字符串惰性求哈希值的过程，参见 4.2.1 节。

⁵Real-World Impact of Hash DoS in Lua. <http://lua-users.org/lists/lua-l/2012-01/msg00497.html>

Lua 5.2.1 为了解决这个问题，把长字符串独立出来。大量文本处理中的输入的字符串不再通过哈希内部化进入全局字符串表。同时，使用了一个随机种子用于哈希值的计算，使攻击者无法轻易构造出拥有相同哈希值的不同字符串。

源代码 3.5: lstring.c: stringhash

```

51 unsigned int luaS_hash (const char *str, size_t l, unsigned int seed) {
52     unsigned int h = seed ^ cast(unsigned int, 1);
53     size_t l1;
54     size_t step = (l >> LUAH_HASHLIMIT) + 1;
55     for (l1 = 1; l1 >= step; l1 -= step)
56         h = h ^ ((h<<5) + (h>>2) + cast_byte(str[l1 - 1]));
57     return h;
58 }

```

这个随机种子是在 Lua State 创建时放在全局表中的，它利用了构造状态机的内存地址随机性以及用户可配置的一个随机量同时来决定⁶。

源代码 3.6: lstate.c: makeseed

```

80 /*
81 ** Compute an initial seed as random as possible. In ANSI, rely on
82 ** Address Space Layout Randomization (if present) to increase
83 ** randomness..
84 */
85 #define addbuff(b,p,e) \
86     { size_t t = cast(size_t, e); \
87       memcpy(buff + p, &t, sizeof(t)); p += sizeof(t); }
88
89 static unsigned int makeseed (lua_State *L) {
90     char buff[4 * sizeof(size_t)];
91     unsigned int h = luai_makeseed();
92     int p = 0;
93     addbuff(buff, p, L); /* heap variable */
94     addbuff(buff, p, &h); /* local variable */
95     addbuff(buff, p, lua0_nilobject); /* global variable */
96     addbuff(buff, p, &lua_newstate); /* public function */
97     lua_assert(p == sizeof(buff));
98     return luaS_hash(buff, p, h);
99 }

```

⁶用户可以在 luaconf.h 中配置 luai_makeseed 定义自己的随机方法，默认是利用 time 函数获取时间构造种子。值得注意的是，使用系统当前时间来构造随机种子这种行为，有可能给调试带来一些困扰。因为字符串 hash 值的不同，会让程序每次运行过程中的内部布局有一些细微变化。好在字符串池使用的是开散列算法（参见3.2.2），这个影响非常的小。但如果你希望让嵌入 lua 的程序，每次运行都严格一致，最好自己定义 luai_makeseed 函数。

3.2 实现

3.2.1 字符串比较

比较两个字符串是否相同，需要区分长短字符串。子类型不同自然不是相同的字符串。

源代码 3.7: lstring.c: eqstr

```
45 int luaS_eqstr (TString *a, TString *b) {
46     return (a->tsv.tt == b->tsv.tt) &&
47         (a->tsv.tt == LUA_TSHRSTR ? eqshrstr(a, b) : luaS_eqlngstr(a, b));
48 }
```

长字符串比较，当长度不同时，自然是不同的字符串。而长度相同时，则需要逐字节比较：

源代码 3.8: lstring.c: eqlngstr

```
33 int luaS_eqlngstr (TString *a, TString *b) {
34     size_t len = a->tsv.len;
35     lua_assert(a->tsv.tt == LUA_TLNGSTR && b->tsv.tt == LUA_TLNGSTR);
36     return (a == b) || /* same instance or... */
37         ((len == b->tsv.len) && /* equal length and ... */
38         (memcmp(getstr(a), getstr(b), len) == 0)); /* equal contents */
39 }
```

短字符串因为经过的内部化，所以不必比较字符串内容，而仅需要比较对象地址即可。Lua 用一个宏来高效的实现它：

源代码 3.9: lstring.h: eqshrstr

```
34 #define eqshrstr(a,b)    check_exp((a)->tsv.tt == LUA_TSHRSTR, (a) == (b))
```

3.2.2 短字符串的内部化

所有的短字符串都被内部化放在全局的字符串表中。这张表是用一个哈希表来实现⁷。

源代码 3.10: lstring.c: internshrstr

```
133 static TString *internshrstr (lua_State *L, const char *str, size_t l) {
134     GCObject *o;
135     global_State *g = G(L);
136     unsigned int h = luaS_hash(str, l, g->seed);
137     for (o = g->strt.hash[lmod(h, g->strt.size)];
138         o != NULL;
139         o = gch(o)->next) {
140         TString *ts = rawgco2ts(o);
```

⁷字符串表和 Lua 表中的哈希部分（源代码4.4）不同，需求更简单。它不必迭代。全局只有一个，不用太考虑空间利用率。所以这里使用的是开散列算法。开散列也被称为 Separate chaining [10]。即，将哈希值相同的对象串在分别独立的链表中，实现起来更为简单。

```

141     if (h == ts->tsv.hash &&
142         l == ts->tsv.len &&
143         (memcmp(str, getstr(ts), l * sizeof(char)) == 0)) {
144         if (isdead(G(L), o)) /* string is dead (but was not collected yet)?
145             */
146             changewhite(o); /* resurrect it */
147         return ts;
148     }
149     return newshrstr(L, str, l, h); /* not found; create a new string */
150 }

```

这是一个开散列的哈希表实现。一个字符串被放入字符串表的时候，先检查一下表中有没有相同的字符串。如果有，则复用已有的字符串；没有则创建一个新的。碰到哈希值相同的字符串，简单的串在同一个哈希位的链表上即可。

注意 144-145 行，这里需要检查表中的字符串是否是死掉的字符串。这是因为 Lua 的垃圾收集过程是逐步完成的。而向字符串池添加新字符串在任何步骤之间都可能发生。有可能在标记完字符串后发现有些字符串没有任何引用，但在下个步骤中又产生了相同的字符串导致这个字符串复活。

当哈希表中字符串的数量 (nuse 域) 超过预定容量 (size 域) 时。可以预计 hash 冲突必然发生。这个时候就调用 `luaS_resize` 方法把字符串表的哈希链表数组扩大，重新排列所有字符串的位置。这个过程和 Lua 表（参见源代码4.5）的处理类似，不过里面涉及垃圾收集的一些细节，不在本节分析。

源代码 3.11: `lstring.c: resize`

```

64 void luaS_resize (lua_State *L, int newsize) {
65     int i;
66     stringtable *tb = &G(L)->strtbl;
67     /* cannot resize while GC is traversing strings */
68     luaC_runtilstate(L, ~bitmask(GCSsweepstring));
69     if (newsize > tb->size) {
70         luaM_reallocvector(L, tb->hash, tb->size, newsize, GCObject *);
71         for (i = tb->size; i < newsize; i++) tb->hash[i] = NULL;
72     }
73     /* rehash */
74     for (i=0; i<tb->size; i++) {
75         GCObject *p = tb->hash[i];
76         tb->hash[i] = NULL;
77         while (p) { /* for each node in the list */
78             GCObject *next = gch(p)->next; /* save next */
79             unsigned int h = lmod(gco2ts(p)->hash, newsize); /* new position */
80             gch(p)->next = tb->hash[h]; /* chain it */
81             tb->hash[h] = p;
82             resetoldbit(p); /* see MOVE OLD rule */

```

```

83     p = next;
84 }
85 }
86 if (newsize < tb->size) {
87     /* shrinking slice must be empty */
88     lua_assert(tb->hash[newsize] == NULL && tb->hash[tb->size - 1] == NULL)
89     ;
89     luaM_reallocvector(L, tb->hash, tb->size, newsize, GCObject *);
90 }
91 tb->size = newsize;
92 }

```

每在 Lua 状态机内部创建一个字符串，都会按 C 风格字符串存放，以兼容 C 接口。即在字符串的末尾加上一个 `\0`，这在 `lstring.c` 的 108 行可以见到。这样不违背 Lua 自己用内存块加长度的方式储存字符串的规则，在把 Lua 字符串传递出去和 C 语言做交互时，又不必做额外的转换。

源代码 3.12: `lstring.c`: `createtrobj`

```

98 static TString *createtrobj (lua_State *L, const char *str, size_t l,
99                             int tag, unsigned int h, GCObject **list) {
100     TString *ts;
101     size_t totalsize; /* total size of TString object */
102     totalsize = sizeof(TString) + ((l + 1) * sizeof(char));
103     ts = &luaC_newobj(L, tag, totalsize, list, 0)->ts;
104     ts->tsv.len = l;
105     ts->tsv.hash = h;
106     ts->tsv.extra = 0;
107     memcpy(ts+1, str, l*sizeof(char));
108     ((char *) (ts+1))[l] = '\0'; /* ending 0 */
109     return ts;
110 }

```

3.3 Userdata 的构造

Userdata 在 Lua 中并没有太特别的地方，在储存形式上和字符串相同。可以看成是拥有独立元表，不被内部化处理，也不需要追加 `\0` 的字符串。在实现上，只是对象结构从 `TString` 换成了 `UData`。所以实现代码也被放在 `lstring.c` 中，其 `api` 也以 `luaS` 开头。

源代码 3.13: `lobject.h`: `udata`

```

431 typedef union Udata {
432     L_Umaxalign dummy; /* ensures maximum alignment for 'local' udata */
433     struct {
434         CommonHeader;

```

```
435     struct Table *metatable;
436     struct Table *env;
437     size_t len;  /* number of bytes */
438 } uv;
439 } Udata;
```

Userdata 的数据部分和字符串一样，紧接在这个结构后面，并没有单独分配内存。Userdata 的构造函数如下：

源代码 3.14: lstring.c: newudata

```
175 Udata *luaS_newudata (lua_State *L, size_t s, Table *e) {
176     Udata *u;
177     if (s > MAX_SIZET - sizeof(Udata))
178         luaM_toobig(L);
179     u = &luaC_newobj(L, LUA_TUSERDATA, sizeof(Udata) + s, NULL, 0)->u;
180     u->uv.len = s;
181     u->uv.metatable = NULL;
182     u->uv.env = e;
183     return u;
184 }
```


第四章 表

Lua 使用 table 作为统一的数据结构。在一次对 Lua 作者的采访中¹，他们这样说道：

Roberto: 从我的角度，灵感来自于 VDM（一个主要用于软件规范的形式化方法），当我们开始创建 Lua 时，有一些东西引起了我的兴趣。VDM 提供三种数据聚合的方式：*set*、*sequence* 和 *map*。不过，*set* 和 *sequence* 都很容易用 *map* 来表达，因此我有了用 *map* 作为统一结构的想法。Luiz 也有他自己的原因。

Luiz: 没错，我非常喜欢 AWK，特别是它的联合数组。

4.1 数据结构

用 table 来表示 Lua 中的一切数据结构是 Lua 语言的一大特色。为了效率，Lua 的官方实现，又把 table 的储存分为数组部分和哈希表部分。数组部分从 1 开始作整数数字索引。这可以提供紧凑且高效的随机访问。而不能被储存在数组部分的数据全部放在哈希表中，唯一不能做哈希键值的是 nil，这个限制可以帮助我们发现许多运行期错误。Lua 的哈希表有一个高效的实现，几乎可以认为操作哈希表的时间复杂度为 $O(1)$ 。

这样分开的储存方案，对使用者是完全透明的，并没有强求 Lua 语言的实现一定要这样分开。但在 lua 的基础库中，提供了 pairs 和 ipairs 两个不同的 api 来实现两种不同方式的 table 遍历方案；用 # 对 table 取长度时，也被定义成和整数下标有关，而非整个 table 的尺寸。在 Lua 的 C API 中，有独立的 api 来操作数组的整数下标部分。也就是说，Lua 有大量的特性，提供了对整数下标的高效访问途径。

Table 的内部数据结构被定义在 lobject.h 中，

源代码 4.1: lobject.h: table

```
545 typedef union TKey {
546     struct {
547         TValuefields;
548         struct Node *next; /* for chaining */
549     } nk;
550     TValue tvk;
551 } TKey;
552
553
554 typedef struct Node {
```

¹见《Masterminds of Programming: Conversations with the Creators of Major Programming Languages》的第 7 章，中译名《编程之魂》。笔者曾做过这一章的翻译：http://blog.codingnow.com/2010/06/masterminds_of_programming_7_lua.html

```

555     TValue i_val;
556     TKey i_key;
557 } Node;
558
559
560 typedef struct Table {
561     CommonHeader;
562     lu_byte flags; /* 1<<p means tagmethod(p) is not present */
563     lu_byte lsizenode; /* log2 of size of 'node' array */
564     struct Table *metatable;
565     TValue *array; /* array part */
566     Node *node;
567     Node *lastfree; /* any free position is before this position */
568     GCObject *gclist;
569     int sizearray; /* size of 'array' array */
570 } Table;
571
572
573
574 /*
575 ** 'module' operation for hashing (size is always a power of 2)
576 */
577 #define lmod(s,size) \
578     (check_exp((size&(size-1))==0, (cast(int, (s) & ((size)-1)))))
579
580
581 #define twoto(x)      (1<<(x))
582 #define sizenode(t)   (twoto((t)->lsizenode))

```

Table 的数组部分被储存在 `TValue *array` 中，其长度信息存于 `int sizearray`。哈希表储存在 `Node *node`，哈希表的大小用 `lu_byte lsizenode` 表示，由于哈希表的大小一定为 2 的整数次幂，所以这里的 `lsizenode` 表示的是幂次，而不是实际大小。

每个 table 结构，最多会由三块连续内存构成。一个 Table 结构，一块存放了连续整数索引的数组，和一块大小为 2 的整数次幂的哈希表。哈希表的最小尺寸为 2 的 0 次幂，也就是 1。为了减少空表的维护成本，Lua 在这里做了一点优化。它定义了一个不可改写的空哈希表：`dummynode`。让空表被初始化时，`node` 域指向这个 `dummy` 节点。它虽然是一个全局变量，但因为对其访问是只读的，所以不会引起线程安全问题。²

源代码 4.2: ltable.c: dummynode

```

67 #define dummynode      (&dummynode_)

```

²不当的链接 lua 库，有可能造成错误。如果你错误的链接了两份相同的 Lua 库实现到你的进程中，大多数情况下，代码可以安全的运行。但在清除空 table 时，会因为无法正确的识别 `dummynode` 而程序崩溃。建议在 Lua 扩展库的实现时调用 `luaL_checkversion` 做一下检查。更详细的分析参见笔者的一篇 blog http://blog.codingnow.com/2012/01/lua_link_bug.html。

```

68
69 #define isdummy(n)                ((n) == dummynode)
70
71 static const Node dummynode_ = {
72     {NILCONSTANT}, /* value */
73     {{NILCONSTANT, NULL}} /* key */
74 };

```

阅读 luaH_new 和 luaH_free 两个 api 的实现，可以了解这一层次的数据结构。

源代码 4.3: ltable.c: new

```

368 Table *luaH_new (lua_State *L) {
369     Table *t = &luaC_newobj(L, LUA_TTABLE, sizeof(Table), NULL, 0)->h;
370     t->metatable = NULL;
371     t->flags = cast_byte(~0);
372     t->array = NULL;
373     t->sizearray = 0;
374     setnodevector(L, t, 0);
375     return t;
376 }
377
378
379 void luaH_free (lua_State *L, Table *t) {
380     if (!isdummy(t->node))
381         luaM_freearray(L, t->node, cast(size_t, sizenode(t)));
382     luaM_freearray(L, t->array, t->sizearray);
383     luaM_free(L, t);
384 }

```

其中 setnodevector 用来初始化哈希表部分。内存管理部分则使用了 luaM 相关 api。

4.2 算法

Table 按照 lua 语言的定义，需要实现四种基本操作：读、写、迭代和获取长度。lua 中并没有删除操作，而仅仅是把对应键位的值设置为 nil。

写操作被实现为查询已有键位，若不存在则创建新键。得到键位后，写入操作就是一次赋值。所以，在 table 模块中，实际实现的基本操作为：创建、查询、迭代和获取长度。

创建操作的 api 为 luaH_newkey，阅读它的实现就能对整个 table 有一个全面的认识。它只负责在哈希表中创建出一个不存在的键，而不关数组部分的工作。因为 table 的数组部分操作和 C 语言数组没有什么不同，不需要特别处理。

源代码 4.4: ltable.c: newkey

```

405 TValue *luaH_newkey (lua_State *L, Table *t, const TValue *key) {

```

```

406 Node *mp;
407 if (ttisnil(key)) luaG_runerror(L, "table_index_is_nil");
408 else if (ttisnumber(key) && luai_numisnan(L, nvalue(key)))
409     luaG_runerror(L, "table_index_is_NaN");
410 mp = mainposition(t, key);
411 if (!ttisnil(gval(mp)) || isdummy(mp)) { /* main position is taken? */
412     Node *othern;
413     Node *n = getfreepos(t); /* get a free place */
414     if (n == NULL) { /* cannot find a free place? */
415         rehash(L, t, key); /* grow table */
416         /* whatever called 'newkey' take care of TM cache and GC barrier */
417         return luaH_set(L, t, key); /* insert key into grown table */
418     }
419     lua_assert(!isdummy(n));
420     othern = mainposition(t, gkey(mp));
421     if (othern != mp) { /* is colliding node out of its main position? */
422         /* yes; move colliding node into free position */
423         while (gnext(othern) != mp) othern = gnext(othern); /* find previous
424         */
425         gnext(othern) = n; /* redo the chain with 'n' in place of 'mp' */
426         *n = *mp; /* copy colliding node into free pos. (mp->next also goes)
427         */
428         gnext(mp) = NULL; /* now 'mp' is free */
429         setnilvalue(gval(mp));
430     }
431     else { /* colliding node is in its own main position */
432         /* new node will go into free position */
433         gnext(n) = gnext(mp); /* chain new position */
434         gnext(mp) = n;
435         mp = n;
436     }
437 }
438 setobj2t(L, gkey(mp), key);
439 luaC_barrierback(L, obj2gco(t), key);
440 lua_assert(ttisnil(gval(mp)));
441 return gval(mp);
442 }

```

lua 的哈希表以闭散列³方式实现。每个可能的键值，在哈希表中都有一个主要位置，称作 mainposition。创建一个新键时，检查 mainposition，若无人使用，则可以直接设置为这个新键。若之前有其它键占据了

³用闭散列方法解决哈希表的键冲突，往往可以让哈希表内数据更为紧凑，有更高的空间利用率。关于其算法，可以参考：http://en.wikipedia.org/wiki/Open_addressing

这个位置，则检查占据此位置的键的主位置是不是这里。若两者位置冲突，则利用 Node 结构中的 next 域，以一个单向链表的形式把它们链起来；否则，新键占据这个位置，而老键更换到新位置并根据它的主键找到属于它的链的那条单向链表中上一个结点，重新链入。

无论是哪种冲突情况，都需要在哈希表中找到一个空闲可用的结点。这里是在 getfreepos 函数中，递减 lastfree 域来实现的。lua 也不会再在设置键位的值为 nil 时而回收空间，而是在预先准备好的哈希空间使用完后惰性回收。即在 lastfree 递减到哈希空间头时，做一次 rehash 操作。

源代码 4.5: ltable.c: rehash

```

343 static void rehash (lua_State *L, Table *t, const TValue *ek) {
344     int nasize, na;
345     int nums[MAXBITS+1]; /* nums[i] = number of keys with 2^(i-1) < k <= 2^i
                          */
346     int i;
347     int totaluse;
348     for (i=0; i<=MAXBITS; i++) nums[i] = 0; /* reset counts */
349     nasize = numusearray(t, nums); /* count keys in array part */
350     totaluse = nasize; /* all those keys are integer keys */
351     totaluse += numusehash(t, nums, &nasize); /* count keys in hash part */
352     /* count extra key */
353     nasize += countint(ek, nums);
354     totaluse++;
355     /* compute new size for array part */
356     na = computesizes(nums, &nasize);
357     /* resize the table to new computed sizes */
358     luaH_resize(L, t, nasize, totaluse - na);
359 }

```

rehash 的主要工作是统计当前 table 中到底有多少有效键值对，以及决定数组部分需要开辟多少空间。其原则是最终数组部分的利用率需要超过 50%。

lua 使用一个 rehash 函数中定义在栈上的 nums 数组来做这个整数键统计工作。这个数组按 2 的整数幂次来分开统计各个区段间的整数键个数。统计过程的实现见 numusearray 和 numusehash 函数。

最终，computesizes 函数计算出不低于 50% 利用率下，数组该维持多少空间。同时，还可以得到有多少有效键将被储存在哈希表里。

根据这些统计数据，rehash 函数调用 luaH_resize 这个 api 来重新调整数组部分和哈希部分的大小，并把不能放在数组里的键值对重新塞入哈希表。

查询操作 luaH_get 的实现要简单的多。当查询键为整数键且在数组范围内时，在数组部分查询；否则，根据键的哈希值去哈希表中查询。拥有相同哈希值的冲突键值对，在哈希表中由 Node 的 next 域单向链起来，所以遍历这个链表就可以了。

4.2.1 短字符串优化

以短字符串为键相当常见，lua 对此做了一点优化。

Lua 5.2.1 对短于 `LUA_MAXSHORTLEN`（默认设置为 40 字节）的短字符串会做内部唯一化处理。相同的短字符串在同一个 State 中只会存在一份。这可以简化字符串的比较操作。在 Lua 5.2.0 之前，则会对所有的字符串不论长短做此处理。但在大多数应用场合，长字符串都是文本处理的对象，而不会做比较操作，内部唯一化处理将带来额外开销。而且对长字符串的部分哈希可能被用于 DoS 攻击，参见 3.1.1。

在 Lua 5.2.1 中，长字符串并不做内部唯一化，且其哈希值也是惰性计算的。我们在 `mainposition` 函数中的 101-106 行可以看到这段惰性计算的代码。

源代码 4.6: `ltable.c: mainposition`

```

97 static Node *mainposition (const Table *t, const TValue *key) {
98     switch (ttype(key)) {
99         case LUA_TNUMBER:
100             return hashnum(t, nvalue(key));
101         case LUA_TLNGSTR: {
102             TString *s = rawtsvalue(key);
103             if (s->tsv.extra == 0) { /* no hash? */
104                 s->tsv.hash = luaS_hash(getstr(s), s->tsv.len, s->tsv.hash);
105                 s->tsv.extra = 1; /* now it has its hash */
106             }
107             return hashstr(t, rawtsvalue(key));
108         }
109         case LUA_TSHRSTR:
110             return hashstr(t, rawtsvalue(key));
111         case LUA_TBOOLEAN:
112             return hashboolean(t, bvalue(key));
113         case LUA_TLIGHTUSERDATA:
114             return hashpointer(t, pvalue(key));
115         case LUA_TLCF:
116             return hashpointer(t, fvalue(key));
117         default:
118             return hashpointer(t, gcvalue(key));
119     }
120 }

```

从 `luaH_get` 的实现中可以看到，遇到短字符串查询时，就会去调用 `luaH_getstr` 回避逐字节的字符串比较操作。

源代码 4.7: `ltable.c: get`

```

463 /*
464 ** search function for short strings
465 */
466 const TValue *luaH_getstr (Table *t, TString *key) {
467     Node *n = hashstr(t, key);
468     lua_assert(key->tsv.tt == LUA_TSHRSTR);

```

```

469 do { /* check whether 'key' is somewhere in the chain */
470     if (ttisshrstring(gkey(n)) && eqshrstr(rawtsvalue(gkey(n)), key))
471         return gval(n); /* that's it */
472     else n = gnext(n);
473 } while (n);
474 return lua0_nilobject;
475 }
476
477
478 /*
479 ** main search function
480 */
481 const TValue *luaH_get (Table *t, const TValue *key) {
482     switch (ttype(key)) {
483         case LUA_TSHRSTR: return luaH_getstr(t, rawtsvalue(key));
484         case LUA_TNIL: return lua0_nilobject;
485         case LUA_TNUMBER: {
486             int k;
487             lua_Number n = nvalue(key);
488             lua_number2int(k, n);
489             if (luai_numeq(cast_num(k), n)) /* index is int? */
490                 return luaH_getint(t, k); /* use specialized version */
491             /* else go through */
492         }
493         default: {
494             Node *n = mainposition(t, key);
495             do { /* check whether 'key' is somewhere in the chain */
496                 if (luaV_rawequalobj(gkey(n), key))
497                     return gval(n); /* that's it */
498                 else n = gnext(n);
499             } while (n);
500             return lua0_nilobject;
501         }
502     }
503 }

```

4.2.2 数字类型的哈希值

当数字类型为键，且没有置入数组部分时，我们需要对它们取哈希值，便于放进哈希表内。

在 Lua 5.1 之前，对数字类型的哈希运算非常简单。就是对其占用的内存块的数据，按整数形式相加，代码如下：

```

1  #define numints          cast_int(sizeof(lua_Number)/sizeof(int))
2
3  static Node *hashnum (const Table *t, lua_Number n) {
4      unsigned int a[numints];
5      int i;
6      if (lua_i_numeq(n, 0)) /* avoid problems with -0 */
7          return gnode(t, 0);
8      memcpy(a, &n, sizeof(a));
9      for (i = 1; i < numints; i++) a[0] += a[i];
10     return hashmod(t, a[0]);
11 }

```

由于 Lua 的数字类型是允许用户配置的，有人为了让它可以处理 64 位整数，将其配置成 long double。这使得上面这段代码引发了一个 bug，他将这个 bug 报告到 lua 邮件列表中⁴。LuaJIT 的作者 Mike Pall 指出，在 64 位系统下，long double 被认为是 16 字节而不是 32 位系统下的 12 字节，以保持对齐。但其数值本身还是 12 字节，另 4 字节被填入随机的垃圾数据。如果依旧用上面的算法计算数字的哈希值，则有可能导致相同的数字拥有不同的哈希值⁵。

从严谨角度上来说，上面的 hashnum 算法也是值得商榷的。所以到了 Lua 5.2 以后，这个函数被修改为如下的实现：

源代码 4.8: ltable.c: hashnum

```

80  static Node *hashnum (const Table *t, lua_Number n) {
81      int i;
82      lua_i_hashnum(i, n);
83      if (i < 0) {
84          if (cast(unsigned int, i) == 0u - i) /* use unsigned to avoid
85              overflows */
86              i = 0; /* handle INT_MIN */
87              i = -i; /* must be a positive value */
88      }
89      return hashmod(t, i);
90 }

```

这里把数字哈希算法函数提取出来，变成了用户可配置的函数 `lua_i_hashnum`。这个函数默认定义在 `llimits.h` 中。

这里有两个预定义版本，当用户维持数字类型为 `double`，且目标机器使用 IEEE754 标准的浮点数时，使用这样一个版本：

源代码 4.9: llimits.h: hashnum1

```

232 #define lua_i_hashnum(i,n) \
233 { volatile union lua_i_Cast u; u.l_d = (n) + 1.0; /* avoid -0 */ \

```

⁴见 2009 年 10 月，围绕 Crash in luaL_loadfile in 64-bit x86_64 的讨论。<http://lua-users.org/lists/lua-l/2009-10/msg00642.html>

⁵在 64 位系统下，用 `lightuserdata` 来处理 64 位整数是更好的选择，笔者实现了一个简单的扩展库：<https://github.com/cloudwu/lua-int64>


```
234 (i) = u.l_p[0]; (i) += u.l_p[1]; } /* add double bits for his hash */
```

它使用一个联合体来取出浮点数所占内存中的数据。

```
1 union luai_Cast { double l_d; LUA_INT32 l_p[2]; };
```

如果无法用这种技巧来快速计算数字的哈希值，则改用性能不高，但更为通用，符合标准的算法：

源代码 4.10: llimits.h: hashnum2

```
281 #include <float.h>
282 #include <math.h>
283
284 #define luai_hashnum(i,n) { int e; \
285     n = l_mathop(frexp)(n, &e) * (lua_Number)(INT_MAX - DBL_MAX_EXP); \
286     lua_number2int(i, n); i += e; }
287
288 #endif
```

4.3 表的迭代

在 Lua 中，并没有提供一个自维护状态的迭代器。而是给出了一个 `next` 方法。传入上一个键，返回下一个键值对。这就是 `luaH_next` 所要实现的。

源代码 4.11: ltable.c: next

```
169 int luaH_next (lua_State *L, Table *t, StkId key) {
170     int i = findindex(L, t, key); /* find original element */
171     for (i++; i < t->sizearray; i++) { /* try first array part */
172         if (!ttisnil(&t->array[i])) { /* a non-nil value? */
173             setnvalue(key, cast_num(i+1));
174             setobj2s(L, key+1, &t->array[i]);
175             return 1;
176         }
177     }
178     for (i -= t->sizearray; i < sizenode(t); i++) { /* then hash part */
179         if (!ttisnil(gval(gnode(t, i)))) { /* a non-nil value? */
180             setobj2s(L, key, gkey(gnode(t, i)));
181             setobj2s(L, key+1, gval(gnode(t, i)));
182             return 1;
183         }
184     }
185     return 0; /* no more elements */
186 }
```

它尝试返回传入的 key 在数组部分中的下一个非空值。当超出数组部分后，则检索哈希表中的对应位置，并返回哈希表中对应节点在储存空间分布上的下一个节点处的键值对。

在大多数其它语言中，遍历一个无序集合的过程中，通常不允许对这个集合做任何修改。即使允许，也可能产生未定义的结果。在 lua 中也一样，遍历一个 table 的过程中，向这个 table 插入一个新键这个行为，将无法预测后续的遍历行为⁶但是，lua 却允许在遍历过程中，修改 table 中已存在的键对应的值⁷。由于 lua 没有显式的从 table 中删除键的操作，只能对不需要的键设为空。

一旦在迭代过程中发生垃圾收集，对键值赋值为空的操作就有可能导致垃圾收集过程中把这个键值对标记为死键⁸。所以，在 next 操作中，从上一个键定位下一个键时，需要支持检索一个死键，查询这个死键的下一个键位。具体代码见 findindex 的实现：

源代码 4.12: ltable.c: findindex

```

144 static int findindex (lua_State *L, Table *t, StkId key) {
145     int i;
146     if (ttisnil(key)) return -1; /* first iteration */
147     i = arrayindex(key);
148     if (0 < i && i <= t->sizearray) /* is 'key' inside array part? */
149         return i-1; /* yes; that's the index (corrected to C) */
150     else {
151         Node *n = mainposition(t, key);
152         for (;;) { /* check whether 'key' is somewhere in the chain */
153             /* key may be dead already, but it is ok to use it in 'next' */
154             if (luaV_rawequalobj(gkey(n), key) ||
155                 (ttisdeadkey(gkey(n)) && iscollectable(key) &&
156                  deadvalue(gkey(n)) == gcvalue(key))) {
157                 i = cast_int(n - gnode(t, 0)); /* key index in hash table */
158                 /* hash elements are numbered after array ones */
159                 return i + t->sizearray;
160             }
161             else n = gnext(n);
162             if (n == NULL)
163                 luaG_runerror(L, "invalid key to " LUA_QL("next")); /* key not
164                                     found */
165         }
166     }
}

```

lua 的 table 的长度定义只对序列表有效⁹。所以，在实现的时候，仅需要遍历 table 的数组部分。只有

⁶从实现上看，在遍历过程中插入一个不存在的键，并不会让程序崩溃，但有可能在当次遍历过程中，无法遍历到这个新插入的键。更严重的后果是，新插入的键触发了 rehash 过程，很有可能遍历到曾经遍历过的节点。

⁷在 Lua 手册[5] 关于 next 的描述中，这样写道：The behavior of next is undefined if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

⁸死键在 rehash 后会从哈希表中清除，而不添加新键就不会 rehash 表而导致死键消失。在这个前提下，遍历 table 是安全的。

⁹在 Lua 5.1 时，对 table 的获取长度定义更为严格一些。一个 table t 的长度 n，要保证 t[n] 不为空，且 t[n+1] 一定为空。t[1] 为空的 table 的长度可定义为零。到了 lua 5.2 后，在手册[5] 的 3.4.6 节简化了定义。在循序序列中出现空洞（即有空值）有可能影响长度计算，但并非空值一定

当数组部分填满时才需要进一步的去检索哈希表。它使用二分法，来快速在哈希表中快速定位一个非空的整数键的位置。

源代码 4.13: ltable.c: getn

```

532 static int unbound_search (Table *t, unsigned int j) {
533     unsigned int i = j; /* i is zero or a present index */
534     j++;
535     /* find 'i' and 'j' such that i is present and j is not */
536     while (!ttisnil(luaH_getint(t, j))) {
537         i = j;
538         j *= 2;
539         if (j > cast(unsigned int, MAX_INT)) { /* overflow? */
540             /* table was built with bad purposes: resort to linear search */
541             i = 1;
542             while (!ttisnil(luaH_getint(t, i))) i++;
543             return i - 1;
544         }
545     }
546     /* now do a binary search between them */
547     while (j - i > 1) {
548         unsigned int m = (i+j)/2;
549         if (ttisnil(luaH_getint(t, m))) j = m;
550         else i = m;
551     }
552     return i;
553 }
554
555
556 /*
557 ** Try to find a boundary in table 't'. A 'boundary' is an integer index
558 ** such that t[i] is non-nil and t[i+1] is nil (and 0 if t[1] is nil).
559 */
560 int luaH_getn (Table *t) {
561     unsigned int j = t->sizearray;
562     if (j > 0 && ttisnil(&t->array[j - 1])) {
563         /* there is a boundary in the array part: (binary) search for it */
564         unsigned int i = 0;
565         while (j - i > 1) {
566             unsigned int m = (i+j)/2;
567             if (ttisnil(&t->array[m - 1])) j = m;
568             else i = m;

```

会截断长度统计。

```

569     }
570     return i;
571 }
572 /* else must find a boundary in hash part */
573 else if (isdummy(t->node)) /* hash part is empty? */
574     return j; /* that is easy... */
575 else return unbound_search(t, j);
576 }

```

4.4 对元方法的优化

Lua 实现复杂数据结构，大量依赖给 table 附加一个元表（metatable）来实现。故而 table 本身的一大作用就是作为元表存在。查询元表中是否存在一个特定的元方法就很容易成为运行期效率的热点。如果不能高效的解决这个热点，每次对带有元表的 table 的操作，都需要至少多作一次 hash 查询。但是，并非所有元表都提供了所有元方法的，对于不存在的元方法查询就是一个浪费了。

在 源代码4.1 中，我们可以看到，每个 Table 结构中都有一个 flags 域。它记录了那些元方法不存在。ltable.h 里定义了一个宏：

```

1 #define invalidateTmcache(t)    ((t)->flags = 0)

```

这个宏用来在 table 被修改时，清空这组标记位，强迫重新做元方法查询。只要充当元表¹⁰的 table 没有被修改，缺失元方法这样的查询结果就可以缓存在这组标记位中了。

源代码 4.14: ltm.h: fasttm

```

41 #define gfasttm(g,et,e) ((et) == NULL ? NULL : \
42     ((et)->flags & (1u<<(e))) ? NULL : luaT_gettm(et, e, (g)->tmname[e]))
43
44 #define fasttm(l,et,e)    gfasttm(G(l), et, e)

```

我们可以看到 fasttm 这个宏能够快速的剔除不存在的元方法。

另一个优化点是，不必在每次做元方法查询的时候都压入元方法的名字。在 state 初始化时，lua 对这些元方法生成了字符串对象：

源代码 4.15: ltm.c: init

```

32 void luaT_init (lua_State *L) {
33     static const char *const luaT_eventname[] = { /* ORDER TM */
34         "__index", "__newindex",
35         "__gc", "__mode", "__len", "__eq",
36         "__add", "__sub", "__mul", "__div", "__mod",
37         "__pow", "__unm", "__lt", "__le",

```

¹⁰lua 5.2 仅对 table 的元表做了这个优化，而没有理会其它类型的元表的元方法查询。这大概是因为，只有 table 容易缺失一些诸如 __index 这样的元方法，而使用 table 的默认行为。当 lua 代码把这些操作作用于其它类型如 userdata 时，它没有 table 那样的默认行为，故对应的元方法通常存在。

```

38     "__concat", "__call"
39 };
40 int i;
41 for (i=0; i<TM_N; i++) {
42     G(L)->tmname[i] = luaS_new(L, luaT_eventname[i]);
43     luaS_fix(G(L)->tmname[i]); /* never collect these names */
44 }
45 }

```

这样，在 table 查询这些字符串要比我们使用 `lua_getfield` 这样的外部 api 要快的多¹¹。通过调用 `luaT_gettmbyobj` 可以获得需要的元方法。

源代码 4.16: ltm.c: gettm

```

63 const TValue *luaT_gettmbyobj (lua_State *L, const TValue *o, TMS event) {
64     Table *mt;
65     switch (ttypenv(o)) {
66         case LUA_TTABLE:
67             mt = hvalue(o)->metatable;
68             break;
69         case LUA_TUSERDATA:
70             mt = uvalue(o)->metatable;
71             break;
72         default:
73             mt = G(L)->mt[ttypenv(o)];
74     }
75     return (mt ? luaH_getstr(mt, G(L)->tmname[event]) : lua0_nilobject);
76 }

```

4.4.1 类型名字

最后，有一段和元方法不太相关的代码也放在 ltm 模块中¹²。在 ltm.h / ltm.c 中还为每个 lua 类型提供了字符串描述。它用于输出调试信息以及作为 `lua_typename` 的返回值。这个字符串并未在其它场合用到，所以也没有为其预生成 string 对象。

源代码 4.17: ltm.c: typename

```

22 static const char udatatypename[] = "userdata";
23
24 LUAI_DDEF const char *const luaT_typenames_[LUA_TOTALTAGS] = {
25     "no_value",
26     "nil", "boolean", udatatypename, "number",
27     "string", "table", "function", udatatypename, "thread",

```

¹¹使用 `lua_getfield` 做字符串检索，需要先将字符串压入 state。这意味着需要把外部字符串在短字符串表池中做一次哈希查询。

¹²把这组字符串常量定义在 ltm 中，大概是因为这里同时定义了元方法的名字常量，实现比较类似罢了。

```
28     "proto", "upval" /* these last two cases are used for tests only */  
29 };
```

udatatypename 在这里单独并定义出来，多半出于严谨的考虑。让 userdata 和 lightuserdata 返回的 "userdata" 字符串指针保持一致。

第五章 函数与闭包

闭包（Closure）这个概念对 C 语言¹程序员比较陌生。但在函数式编程中却是一个重要概念。如果说 C++ 式的面向对象编程是把一组函数绑定到特定数据类型上的话，那么闭包可以说是把一组数据绑定到特定函数上。

一个简单的闭包是这样的：

```
1 function MakeCounter()  
2     local t = 0  
3     return function()  
4         t = t + 1  
5         return t  
6     end  
7 end
```

当调用 MakeCounter 后，会得到一个函数。这个函数每调用一次，返回值就会递增一。顾名思义，我们可以把这个返回的函数看作一个计数器。MakeCounter 可以产生多个计数器，每个都独立计数。也就是说，每个计数器函数都独享一个变量 t，相互不干扰。这个 t 被称作计数器函数的 upvalue，被绑定到计数器函数中。拥有了 upvalue 的函数就是闭包。

在 Lua 5.1 之前，所有的函数都实现成闭包。只是从语义上来说，upvalue 数量为零的闭包被称为函数。看这样一段代码：

```
1 function foobar()  
2     return function()  
3         return "Hello"  
4     end  
5 end
```

foobar 每次调用都会生成一个新的、没有 upvalue 的闭包，即便这些闭包在功能上完全相同。Lua 5.2 改进了这一点。它尽量复用拥有完全相同 upvalue 的闭包。对于没有 upvalue 的函数来说，也就不会再出现第二份重复的对象了。

5.1 函数原型

闭包是函数和 upvalue 的结合体。在 Lua 中，闭包统一被称为函数，而函数就是这里所指的函数原型。原型是不可以被调用的，只有和 upvalue 结合在一起才变成了 Lua 语言中供用户使用的函数对象。所

¹C 语言缺乏支持闭包的基本语法机制，但也有人人为它做了语法扩展。Apple 公司为 C 提供了一个叫作 Blocks 的非标准扩展，可以让 C 语言支持闭包[7]。

以，从公开的 API 定义中，是不存在函数原型这一数据类型的。它只存在于实现中，但和字符串、表这些数据类型一样，参于垃圾回收的过程。

```

1 function foobar()
2     return load[[
3         return "Hello"
4     ]]
5 end

```

这段代码看似会得到和前面提到的代码相同的结果：每次调用 foobar 都会生成一个返回 Hello 字符串的函数。但实际上却有所区别。load 不仅仅有编译代码的成本，而且每次会产生一个新的函数原型；而直接返回一个函数对象，无论是否需要绑定 upvalue，都会复用同一个函数原型。

生成函数原型有两个途径：其一是由源代码编译而来，其二可以从编译好的字节码加载得到。这些都不在本章节细述，我们仅讨论把函数原型和 upvalue 绑定在一起得到闭包的过程。

Proto 是一种 GCOBJECT，它的结构定义在 lobject.h 中。它的类型为 LUA_TPROTO，由于这并不是一个公开类型，最终用户无法得到一个 Proto 对象，所以 LUA_TPROTO 没有定义在 lua.h 中，而存在于 lobject.h。

```

1 #define LUA_TPROTO          LUA_NUMTAGS

```

源代码 5.1: lobject.h: Proto

```

467 typedef struct Proto {
468     CommonHeader;
469     TValue *k; /* constants used by the function */
470     Instruction *code;
471     struct Proto **p; /* functions defined inside the function */
472     int *lineinfo; /* map from opcodes to source lines (debug information)
473                    */
474     LocVar *locvars; /* information about local variables (debug information)
475                      */
476     Upvaldesc *upvalues; /* upvalue information */
477     union Closure *cache; /* last created closure with this prototype */
478     TString *source; /* used for debug information */
479     int sizeupvalues; /* size of 'upvalues' */
480     int sizek; /* size of 'k' */
481     int sizecode;
482     int sizelineinfo;
483     int sizep; /* size of 'p' */
484     int sizelocvars;
485     int linedefined;
486     int lastlinedefined;
487     GCOBJECT *gclist;
488     lu_byte numparams; /* number of fixed parameters */

```



```

487     lu_byte is_vararg;
488     lu_byte maxstacksize;  /* maximum stack used by this function */
489 } Proto;

```

从数据结构看，Proto 记录了函数原型的字节码、函数引用的常量表、调试信息、和其它一些基本信息：有多少个参数，调用这个函数需要多大的数据栈空间。参数个数和数据栈空间大小可以让虚拟机在运行的时候，可以精确预分配出需要的数据栈空间，这样就不必每次对栈访问的时候都去做越界检查了。这个结构的细节在第七章提及，还会讨论。

Lua 的函数不像 C 语言那样，只能平坦的定义²；Lua 可以在函数中再定义函数，Proto 也就必然被实现为有层级的。在 Proto 结构中，我们能看到对内层 Proto 的引用 p。

从 Lua 5.2 开始，Lua 将原型和变量绑定的过程，都尽量避免重复生成不必要的闭包。当生成一次闭包后，闭包将被 cache 引用，下次再通过这个原型生成闭包时，比较 upvalue 是否一致来决定复用。cache 是一个弱引用，一旦在 gc 流程发现引用的闭包已不存在，cache 将被置空。

Lua 的调试信息很丰富，调试信息占用的内存甚至多过字节码本身。每条 code 中的指令都对应着 line 数组中的源代码行号。局部变量和 upvalue 的调试信息，包括名字和在源代码中的作用域都记录在 Proto 结构里。从结构定义的名字中很好理解。

嵌套 Proto、源代码、变量名这些都由 GC 管理生命期。其它数据结构则和 Proto 本身紧密绑定。我们查看 luaF_freeproto 的源代码就能读到在释放一个 Proto 结构时，顺带回收了这些内部数据结构占用的内存。由于 Lua 的内存管理需要提供每个内存块的大小³，在 Proto 结构中也如实记录了它们。

源代码 5.2: lfunc.c: luaF_freeproto

```

135 void luaF_freeproto (lua_State *L, Proto *f) {
136     luaM_freearray(L, f->code, f->sizecode);
137     luaM_freearray(L, f->p, f->sizep);
138     luaM_freearray(L, f->k, f->sizek);
139     luaM_freearray(L, f->lineinfo, f->sizelineinfo);
140     luaM_freearray(L, f->locvars, f->sizelocvars);
141     luaM_freearray(L, f->upvalues, f->sizeupvalues);
142     luaM_free(L, f);
143 }

```

5.2 Upvalue

Upvalue⁴ 指在闭包生成的那一刻，与函数原型绑定在一起的那些外部变量。这些变量原本是上一层函数的局部变量或 upvalue，可以在上层返回返回后，继续被闭包引用。

理解 upvalue 并不太难，但 upvalue 的实现却复杂的多。

当外层函数并没有退出时，我们调用刚生成的闭包，这个时候闭包更像一个普通的内嵌函数。外层函数的局部变量只是数据栈上的一个普通变量，虚拟机用一个数据栈上的索引映射局部变量，内嵌函数可以通过

²GCC 对 C 语言的扩展允许嵌套定义函数。但嵌套函数不等于闭包。你可以从内层函数中访问外层函数中的局部变量，但如果你用函数指针去引用内层函数时，这些函数调用就不可靠了。因为它引用的外层变量可能并不存在或是不是想像中的值。

³参见 2.1 节。

⁴Upvalue 没有特别恰当的中译名。有人主张把 upvalue 直译为“上值”，略显古怪。笔者觉得译为“升量”也不错，意为被提升的变量，但恐怕也难被广泛接受。故在本书中全部保留英文。

数据栈自由访问它。而一旦外层函数返回，数据栈空间收缩，原有的局部变量消失了。这个时候闭包需要用其它方式访问这些 upvalue。

如果将数据栈上的每个变量都实现成一个独立的对象是没有必要的，尤其是数字、布尔量这类数据类型，没必要实现成复杂对象。Lua 没有采用这种低效的实现方法。它的 upvalue 从实现上来讲，更像是 C 语言中的指针。它引用了另一个对象。多个闭包可以共享同一个 upvalue，有如 C 语言中，可以有多份指针指向同一个结构体。

在 Lua 中，upvalue 是内部的一种独立的数据类型，表示对另一个 Lua 值的引用。

```
1 #define LUA_TUPVAL      (LUA_NUMTAGS+1)
```

UpVal 的结构定义在 lobject.h 中：

源代码 5.3: lobject.h: UpVal

```
496 typedef struct UpVal {
497     CommonHeader;
498     TValue *v; /* points to stack or to its own value */
499     union {
500         TValue value; /* the value (when closed) */
501         struct { /* double linked list (when open) */
502             struct UpVal *prev;
503             struct UpVal *next;
504         } l;
505     } u;
506 } UpVal;
```

它直接用一个 TValue 指针引用一个 Lua 值变量。当被引用的变量还在数据栈上时，这个指针直接指向栈上的地址。这个 upvalue 被称为开放的。由于 Lua 的数据栈的大小可扩展，当数据栈内存延展时，其内存地址会发生变化。这个时候需要主动修正 UpVal 结构中的指针。这个过程我们在 6.1.1 节提及，它由 ldo.c 中的 correctstack 函数来实现⁵。

遍历当前所有开放的 upvalue 利用的是当前线程中记录的链表 openupval⁶。这是一个双向链表，所以在 UpVal 结构中有两个指针指向前后节点。

链表指针保存在一个联合中。当 upvalue 被关闭后，就不再需要这两个指针了。所谓关闭 upvalue，就是当 upvalue 引用的数据栈上的数据不再存在于栈上时（通常是由申请局部变量的函数返回引起的），需要把 upvalue 从开放链表中拿掉，并把其引用的数据栈上的变量值换一个安全的地方存放。这个安全所在就是 UpVal 结构体内。

无须用特别的标记位区分一个 UpVal 在开放还是关闭状态。当 upvalue 关闭时，UpVal 中的指针 v 一定指向结构内部的 value。

5.3 闭包

Lua 支持支持两种闭包：由 Lua 语言实现的，以及用 C 语言实现的。从外部数据类型来看，它们同属于一种类型：函数（Lua.TFUNCTION）。从 GC 管理的角度看，是同一种 GCOBJECT 类型：

⁵参见源代码6.5。

⁶参见源代码6.3。

```

1  typedef union Closure {
2      CClosure c;
3      LClosure l;
4  } Closure;

```

根据 Lua 闭包或是 C 闭包的不同, Closure 定位为一个联合。CClosure 和 LClosure 共有一个相同的数据头:

```

1  #define ClosureHeader \
2      CommonHeader; lu_byte nupvalues; GCObject *gclist

```

5.3.1 Lua 闭包

Lua 闭包仅仅是原型 Proto 和 UpVal 的集合。

源代码 5.4: lobject.h: LClosure

```

523 typedef struct LClosure {
524     ClosureHeader;
525     struct Proto *p;
526     UpVal *upvals[1]; /* list of upvalues */
527 } LClosure;

```

源代码 5.5: lfunc.c: luaF_newLclosure

```

30 Closure *luaF_newLclosure (lua_State *L, int n) {
31     Closure *c = &luaC_newobj(L, LUA_TLCL, sizeLclosure(n), NULL, 0)->c1;
32     c->l.p = NULL;
33     c->l.nupvalues = cast_byte(n);
34     while (n--) c->l.upvals[n] = NULL;
35     return c;
36 }

```

构造 Lua 闭包的内部 API luaF_newLclosure 只绑定了 Proto 却不包括初始化 upvalue 对象的过程。这是因为构造 Lua 闭包有两种可能的途径。

第一、Lua 闭包一般在虚拟机运行的过程中被动态构造出来的。这时, 闭包需引用的 upvalue 都在当前的数据栈上。利用 luaF_findupval 这个 API 可以把数据栈上的值转换为 upvalue。

源代码 5.6: lfunc.c: luaF_findupval

```

47 UpVal *luaF_findupval (lua_State *L, StkId level) {
48     global_State *g = G(L);
49     GCObject **pp = &L->openupval;
50     UpVal *p;
51     UpVal *uv;

```

```

52 while (*pp != NULL && (p = gco2uv(*pp))->v >= level) {
53     GCObject *o = obj2gco(p);
54     lua_assert(p->v != &p->u.value);
55     lua_assert(!isold(o) || isold(obj2gco(L)));
56     if (p->v == level) { /* found a corresponding upvalue? */
57         if (isdead(g, o)) /* is it dead? */
58             changewhite(o); /* resurrect it */
59         return p;
60     }
61     pp = &p->next;
62 }
63 /* not found: create a new one */
64 uv = &luaC_newobj(L, LUA_TUPVAL, sizeof(UpVal), pp, 0)->uv;
65 uv->v = level; /* current value lives in the stack */
66 uv->u.l.prev = &g->uvhead; /* double link it in 'uvhead' list */
67 uv->u.l.next = g->uvhead.u.l.next;
68 uv->u.l.next->u.l.prev = uv;
69 g->uvhead.u.l.next = uv;
70 lua_assert(uv->u.l.next->u.l.prev == uv && uv->u.l.prev->u.l.next == uv);
71 return uv;
72 }

```

这个函数先在当前的 `openupval` 链表中寻找是否已经转化过，如果有就复用之。否则，构造一个新的 `upvalue` 对象，并把它串到 `openupval` 链表中。

当离开一个代码块时，这个代码块中定义的局部变量变得不可见。Lua 会调整数据栈指针，销毁掉这些变量。若这些栈值被某些闭包以开放 `upvalue` 的形式引用，就需要把它们关闭。`luaF_close` 实现了这一系列关闭操作：删掉已经无人引用的 `upvalue`、把数据从数据栈上复制到 `UpVal` 结构中，并修正 `UpVal` 中的指针 `v`。

源代码 5.7: `lfunc.c: luaF_close`

```

82 void luaF_freeupval (lua_State *L, UpVal *uv) {
83     if (uv->v != &uv->u.value) /* is it open? */
84         unlinkupval(uv); /* remove from open list */
85     luaM_free(L, uv); /* free upvalue */
86 }
87
88
89 void luaF_close (lua_State *L, StkId level) {
90     UpVal *uv;
91     global_State *g = G(L);
92     while (L->openupval != NULL && (uv = gco2uv(L->openupval))->v >= level) {
93         GCObject *o = obj2gco(uv);

```

```

94     lua_assert(!isblack(o) && uv->v != &uv->u.value);
95     L->openupval = uv->next;  /* remove from 'open' list */
96     if (isdead(g, o))
97         luaF_freeupval(L, uv);  /* free upvalue */
98     else {
99         unlinkupval(uv);  /* remove upvalue from 'uvhead' list */
100        setobj(L, &uv->u.value, uv->v);  /* move value to upvalue slot */
101        uv->v = &uv->u.value;  /* now current value lives here */
102        gch(o)->next = g->allgc;  /* link upvalue into 'allgc' list */
103        g->allgc = o;
104        luaC_checkupvalcolor(g, uv);
105    }
106 }
107 }

```

第二种生成 Lua 闭包的方式是加载一段 Lua 代码。每段 Lua 代码都会被编译成一个函数原型，但 Lua 的外部 API 是不返回函数原型对象的，而是把这个函数原型转换为一个 Lua 闭包对象。如果从源代码加载的话，不可能有用户构建出来的 upvalue 的。但是，任何一个代码块都至少有一个 upvalue：_ENV。⁷

如果用户试图 dump 一个拥有多个 upvalue 的 Lua 闭包，它会得到一个 upvalue 数量不为一的函数原型的二进制数据块。undump 这个数据块，就将生成多个 upvalue 的闭包。

在 ldo.c 中，我们能读到 f_parser 函数把加载进内存的函数原型和 upvalue 绑定起来。

源代码 5.8: ldo.c: f_parser

```

632 static void f_parser (lua_State *L, void *ud) {
633     int i;
634     Closure *cl;
635     struct SParser *p = cast(struct SParser *, ud);
636     int c = zgetc(p->z);  /* read first character */
637     if (c == LUA_SIGNATURE[0]) {
638         checkmode(L, p->mode, "binary");
639         cl = luaU_undump(L, p->z, &p->buff, p->name);
640     }
641     else {
642         checkmode(L, p->mode, "text");
643         cl = luaY_parser(L, p->z, &p->buff, &p->dyd, p->name, c);
644     }
645     lua_assert(cl->l.nupvalues == cl->l.p->sizeupvalues);
646     for (i = 0; i < cl->l.nupvalues; i++) {  /* initialize upvalues */
647         UpVal *up = luaF_newupval(L);
648         cl->l.upvals[i] = up;

```

⁷_ENV 从 Lua 5.2 开始引入 Lua 语言，同时废弃了之前版本中环境这个概念。Lua 的核心不再区分全局表、环境表这些。访问全局变量只是对 _ENV 这张表访问的语法糖。_ENV 必须被每一个 Lua 函数可见，所以被放在 Lua 代码块的第一个 upvalue 中。

```

649     luaC_objbarrier(L, cl, up);
650 }
651 }

```

这些 upvalue 并不源于数据栈，所以是用 luaF_newupval 新构造出来的。

源代码 5.9: lfunc.c: luaF_newupval

```

39 UpVal *luaF_newupval (lua_State *L) {
40     UpVal *uv = &luaC_newobj(L, LUA_TUPVAL, sizeof(UpVal), NULL, 0)->uv;
41     uv->v = &uv->u.value;
42     setnilvalue(uv->v);
43     return uv;
44 }

```

5.3.2 C 闭包

作为嵌入式语言，Lua 必须有和 C 程序方便交互的能力。C 函数可以以闭包的形式方便的嵌入 Lua。

和 Lua 闭包不同，在 C 函数中不会去引用外层函数中的局部变量。所以，C 闭包中的 upvalue 天生就是关闭状态的。Lua 也就不需要用独立的 UpVal 对象来引用它们。对于 C 闭包，upvalue 是直接存放在 CClosure 结构中的。

源代码 5.10: lobject.h: CClosure

```

516 typedef struct CClosure {
517     ClosureHeader;
518     lua_CFunction f;
519     TValue upvalue[1]; /* list of upvalues */
520 } CClosure;

```

C 闭包的构造流程也相对简单的多：

源代码 5.11: lfunc.c: luaF_newCclosure

```

23 Closure *luaF_newCclosure (lua_State *L, int n) {
24     Closure *c = &luaC_newobj(L, LUA_TCCL, sizeCclosure(n), NULL, 0)->cl;
25     c->c.nupvalues = cast_byte(n);
26     return c;
27 }

```

5.3.3 轻量 C 函数

当 C 闭包一个 upvalue 都没有时，可想而知，我们其实不必使用一个复杂的数据结构来表示它。直接用 C 函数指针即可，而且相同的 C 函数是可以复用的，不必每次构造出来时都生成一个新的 CClosure 对象。这就是 Lua 5.2 开始引入的轻量 C 函数的概念。这样的 C 函数不需要由 GC 来管理其生命期，不必为之构建出闭包结构。

我们可以阅读 lapi.c 中的 lua_pushcclosure 的实现来理解它的实现。

源代码 5.12: lapi.c: lua_pushcclosure

```

555 LUA_API void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n) {
556     lua_lock(L);
557     if (n == 0) {
558         setfvalue(L->top, fn);
559     }
560     else {
561         Closure *cl;
562         api_checknelems(L, n);
563         api_check(L, n <= MAXUPVAL, "upvalue_index_too_large");
564         luaC_checkGC(L);
565         cl = luaF_newCclosure(L, n);
566         cl->c.f = fn;
567         L->top -= n;
568         while (n--)
569             setobj2n(L, &cl->c.upvalue[n], L->top + n);
570         setclCvalue(L, L->top, cl);
571     }
572     api_incr_top(L);
573     lua_unlock(L);
574 }

```

对于轻量 C 函数，直接把函数指针用宏 setfvalue 压入堆栈即可。

```

1 #define setfvalue(obj,x) \
2   { TValue *io=(obj); val_(io).f=(x); sett_(io, LUA_TLCF); }

```

setfvalue 给 TValue 设置了类型 LUA_TLCF，它是 LUA_TFUNCTION 三个子类型中的一个，并未对外部 API 公开，定义在 lobject.h 中。

源代码 5.13: lobject.h: lua_TFUNCTION

```

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

```


第六章 协程及函数的执行

Lua 从第 5 版开始，支持了协程（coroutine）这一重要的语言设施，极大的扩展了 Lua 语言的用途。而 Lua 5.2 版，又对协程做了极大的完善，不再有之前的种种限制¹。这些看似微小的进步，背后却历尽十年的艰辛²。

若 Lua 仅作为一种独立语言，支持协程可能并不算麻烦。可困难在于 Lua 生来以一门嵌入式语言[5]存在，天生需要大量与宿主系统 C 语言做交互。典型的应用环境是由 C 语言开发的系统，嵌入 Lua 解析器，加载 Lua 脚本运行。同时注入一些 C 函数供 Lua 脚本调用。Lua 作为控制脚本，并不直接控制外界的模块，做此桥梁的正是那些注入的 C 接口。在较为复杂的应用环境中，这些注入的 C 函数还需要有一些回调方法。当我们企图用 Lua 脚本去定制这些回调行为时，就出现了 C 函数调用 Lua 函数，Lua 函数再调用 C 函数，这个 C 函数又调用 Lua 函数的层层嵌套的过程。

C 语言本身并不支持协程或延续点³，一旦中断 Lua 协程，就面临 C 语言中调用栈如何处理的难题。

作为对比，可以参考期望达到同样功能的，给 Python 提供轻量线程的 Stackless Python。其作者写道⁴：

We do no longer support a native, compatible Stackless implementation but a hardware/platform dependant version only. This gives me the most effect with minimum maintenance work.

直接操作 C 层面的堆栈，可以较为容易的作到协程的切换。但这样做，会和硬件平台绑定。这是一个在 C 中实现延续点的不错的方法。但这个做法不符合 Lua 的设计原则。Lua 为了解决这个问题，对 Lua 语言的实现以及和 C 交互的接口设计上做了大量的努力。最终使用标准的 C 语言，实现了完整功能的 Lua 协程。

6.1 栈与调用信息

刚接触 Lua 时，从 C 层面看待 Lua，Lua 的虚拟机对象就是一个 lua_State。但实际上，真正的 Lua 虚拟机对象被隐藏起来了。那就是 lstate.h 中定义的结构体 global_State（（参见 2.2 节）。

lua_State 是暴露给用户的数据类型。从名字上看，它想表示一个 Lua 程序的执行状态，在官方文档中，它指代 Lua 的一个线程。每个线程拥有独立的数据栈以及函数调用链，还有独立的调试钩子和错误处理设施。所以我们不应当简单的把 lua_State 看成一个静态的数据集，它是一组 Lua 程序的执行状态机。所有的 Lua C API 都是围绕这个状态机，改变其状态的：或把数据压入堆栈，或取出，或执行栈顶的函数，或继续上次被中断的执行过程。

同一 Lua 虚拟机中的所有执行线程，共享了一块全局数据 global_State。在 Lua 的实现代码中，需要访问这个结构体的时候，会调用宏：

¹在 Lua 5.1 版之前，协程不能从元方法内跳出，也不能以保护方式运行。

²Lua 第 4 版于 2000 年底发布，到 2002 年做了一些小修正。首次支持协程的第 5 版正式版 2003 年面世，而让协程功能完善的 5.2 版直到 2011 年底才迟迟到来。

³Continuation：http://en.wikipedia.org/wiki/Continuation

⁴http://www.stackless.com/browser/Stackless/readme.txt

```
1 #define G(L)      (L->l_G)
```

忽略 lstate.h 中涉及 gc 的复杂部分，我们可以先看一眼 lua_State 的数据结构。

源代码 6.1: lstate.h: lua_State

```
154 struct lua_State {
155     CommonHeader;
156     lu_byte status;
157     StkId top; /* first free slot in the stack */
158     global_State *l_G;
159     CallInfo *ci; /* call info for current function */
160     const Instruction *oldpc; /* last pc traced */
161     StkId stack_last; /* last free slot in the stack */
162     StkId stack; /* stack base */
163     int stacksize;
164     unsigned short nny; /* number of non-yieldable calls in stack */
165     unsigned short nCcalls; /* number of nested C calls */
166     lu_byte hookmask;
167     lu_byte allowhook;
168     int basehookcount;
169     int hookcount;
170     lua_Hook hook;
171     GCObject *openupval; /* list of open upvalues in this stack */
172     GCObject *gclist;
173     struct lua_longjmp *errorJmp; /* current error recover point */
174     ptrdiff_t errfunc; /* current error handling function (stack index) */
175     CallInfo base_ci; /* CallInfo for first level (C calling Lua) */
176 };
```

这个数据结构是围绕程序如何执行来设计的，数据栈之外的数据储存并不体现在这个结构中。在 Lua 源代码的其它部分，经常会用到 lua_State，但不是所有代码都需要了解结构的细节。一般提到这类数据的地方，都用变量名 L 来指代它。

接下来，我们来分析 lua_State 中重要的两个数据结构：数据栈和调用链。

6.1.1 数据栈

Lua 中的数据可以这样分为两类：值类型和引用类型。值类型可以被任意复制，而引用类型共享一份数据，由 GC 负责维护生命期。Lua 使用一个联合 union Value 来保存数据。

源代码 6.2: lobject.h: Value

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

```

391     int b;                /* booleans */
392     lua_CFunction f; /* light C functions */
393     numfield          /* numbers */
394 };

```

从这里我们可以看到，引用类型用一个指针 `GObject * gc` 来间接引用，而其它值类型都直接保存在联合中。为了区分联合中存放的数据类型，再额外绑定一个类型字段。

```

1  #define TValuefields    Value value_; int tt_
2
3  typedef struct lua_TValue TValue;
4
5  struct lua_TValue {
6      TValuefields;
7  };

```

这里，使用繁杂的宏定义，`TValuefields` 以及 `numfield` 是为了应用一个被称为 NaN Trick 的技巧。⁵ `lua_State` 的数据栈，就是一个 `TValue` 的数组。代码中用 `StkId` 类型来指代对 `TValue` 的引用。

```

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

```

在 `lstate.c` 中，我们可以读到对堆栈的初始化及释放的代码。

源代码 6.3: `lstate.c`: stack

```

133 static void stack_init (lua_State *L1, lua_State *L) {
134     int i; CallInfo *ci;
135     /* initialize stack array */
136     L1->stack = luaM_newvector(L, BASIC_STACK_SIZE, TValue);
137     L1->stacksize = BASIC_STACK_SIZE;
138     for (i = 0; i < BASIC_STACK_SIZE; i++)
139         setnilvalue(L1->stack + i); /* erase new stack */
140     L1->top = L1->stack;
141     L1->stack_last = L1->stack + L1->stacksize - EXTRA_STACK;
142     /* initialize first ci */
143     ci = &L1->base_ci;

```

⁵在不修改 Lua 的默认配置的情况下，一个 Lua 数据需要 8 个字节来存放，即一个 `double` 的长度。但是，类型信息需要额外的一个字节。由于大多数平台需要数据对齐以保证数据访问的效率，所以 Lua 在默认实现中，使用 `int` 来保存数据类型信息。这样，每个 `TValue` 的数据长度就需要 12 字节，增加了 50%。这看起来是一个不小的开销，所以，在 Lua 5.2 中，可以选择打开 NaN Trick 来把数据类型信息压缩进 8 字节的数据段。

所谓 NaN Trick，就是指在现行的浮点数二进制标准 IEEE 754 中，指数位全 1 时，表示这并不是一个数字。它用来表示无穷大，以及数字除 0 的结果[9]。也就是说，一个浮点数是不是数字，只取决于它的指数部分，和尾数部分无关。现实中的处理器，只会产生一种尾数全为 0 的 NaN。所有尾数不为 0 的 NaN 值，都可以看成是刻意构造出来的值。换句话说，处理器只会产生值为 `0xfff8000000000000` 的 NaN，大于它的值都可以用作其它用途，且能和正常的浮点数区分开。

`Double` 类型的尾数位有 52 位，而在 32 位平台上，仅需要 32 位即可表示 Lua 支持的除数字以外的所有类型（主要是指针），剩下的位置来保存类型信息足够了。当 Lua 5.2 开启 NaN Trick 编译选项时，简单的把前 24 位设置为 `7FF7A5` 来标识非数字类型，留下 8 位储存类型信息。

对于目前 Lua 5.2 的实现，NaN Trick 对于 64 位平台是没有意义的。因为指针和 `double` 同样占用 8 字节的空间。不过，64 位平台上，地址指针有效位只有 48 位，理论上也是可以利用 NaN Trick 的，但这样会增加代码复杂度，且在 64 位平台上节约内存的意义相对较小，Lua 5.2 的实现就没有这么做了。倒是在 LuaJIT 2.0 中，对 64 位平台，同样采用了这个技巧。

```

144     ci->next = ci->previous = NULL;
145     ci->callstatus = 0;
146     ci->func = L1->top;
147     setnilvalue(L1->top++); /* 'function' entry for this 'ci' */
148     ci->top = L1->top + LUA_MINSTACK;
149     L1->ci = ci;
150 }
151
152
153 static void freestack (lua_State *L) {
154     if (L->stack == NULL)
155         return; /* stack not completely built yet */
156     L->ci = &L->base_ci; /* free the entire 'ci' list */
157     luaE_freeCI(L);
158     luaM_freearray(L, L->stack, L->stacksize); /* free stack array */
159 }

```

一开始，数据栈的空间很有限，只有 2 倍的 LUA_MINSTACK 的大小。

```

1 #define BASIC_STACK_SIZE      (2*LUA_MINSTACK)

```

LUA_MINSTACK 默认为 20，配置在 lua.h 中。任何一次 Lua 的 C 函数调用，都只保证有这么大的空闲数据栈空间可用。

Lua 供 C 使用的栈相关 API 都是不检查数据栈越界的，这是因为通常我们编写 C 扩展都能把数据栈空间的使用控制在 LUA_MINSTACK 以内，或是显式扩展。对每次数据栈访问都强制做越界检查是非常低效的。

数据栈不够用的时候，可以扩展。这种扩展是用 realloc 实现的，每次至少分配比原来大一倍的空间，并把旧的数据复制到新空间。

源代码 6.4: ldo.c: growstack

```

157 /* some space for error handling */
158 #define ERRORSTACKSIZE      (LUA_MAXSTACK + 200)
159
160
161 void luaD_reallocstack (lua_State *L, int newsize) {
162     TValue *oldstack = L->stack;
163     int lim = L->stacksize;
164     lua_assert(newsize <= LUA_MAXSTACK || newsize == ERRORSTACKSIZE);
165     lua_assert(L->stack_last - L->stack == L->stacksize - EXTRA_STACK);
166     luaM_reallocvector(L, L->stack, L->stacksize, newsize, TValue);
167     for (; lim < newsize; lim++)
168         setnilvalue(L->stack + lim); /* erase new segment */
169     L->stacksize = newsize;

```

```

170     L->stack_last = L->stack + newsize - EXTRA_STACK;
171     correctstack(L, oldstack);
172 }
173
174
175 void luaD_growstack (lua_State *L, int n) {
176     int size = L->stacksize;
177     if (size > LUAI_MAXSTACK) /* error after extra size? */
178         luaD_throw(L, LUA_ERRERR);
179     else {
180         int needed = cast_int(L->top - L->stack) + n + EXTRA_STACK;
181         int newsize = 2 * size;
182         if (newsize > LUAI_MAXSTACK) newsize = LUAI_MAXSTACK;
183         if (newsize < needed) newsize = needed;
184         if (newsize > LUAI_MAXSTACK) { /* stack overflow? */
185             luaD_reallocstack(L, ERRORSTACKSIZE);
186             luaG_runerror(L, "stack overflow");
187         }
188         else
189             luaD_reallocstack(L, newsize);
190     }
191 }

```

数据栈扩展的过程，伴随着数据拷贝。这些数据都是可以直接值复制的，所以不需要在扩展之后修正其中的指针。但，有些外部结构对数据栈的引用需要修正为正确的新地址。这些需要修正的位置包括 upvalue 以及执行栈对数据栈的引用。这个过程由 correctstack 函数实现。

源代码 6.5: ldo.c: correctstack

```

142 static void correctstack (lua_State *L, TValue *oldstack) {
143     CallInfo *ci;
144     GCObject *up;
145     L->top = (L->top - oldstack) + L->stack;
146     for (up = L->openupval; up != NULL; up = up->gch.next)
147         gco2uv(up)->v = (gco2uv(up)->v - oldstack) + L->stack;
148     for (ci = L->ci; ci != NULL; ci = ci->previous) {
149         ci->top = (ci->top - oldstack) + L->stack;
150         ci->func = (ci->func - oldstack) + L->stack;
151         if (isLua(ci))
152             ci->u.l.base = (ci->u.l.base - oldstack) + L->stack;
153     }
154 }

```

6.1.2 调用栈

Lua 把调用栈和数据栈分开保存。调用栈放在一个叫做 CallInfo 的结构中，以双向链表的形式储存在线程对象里。

源代码 6.6: lstate.h: CallInfo

```

69 typedef struct CallInfo {
70     StkId func; /* function index in the stack */
71     StkId top; /* top for this function */
72     struct CallInfo *previous, *next; /* dynamic call link */
73     short nresults; /* expected number of results from this function */
74     lu_byte callstatus;
75     ptrdiff_t extra;
76     union {
77         struct { /* only for Lua functions */
78             StkId base; /* base for this function */
79             const Instruction *savedpc;
80         } l;
81         struct { /* only for C functions */
82             int ctx; /* context info. in case of yields */
83             lua_CFunction k; /* continuation in case of yields */
84             ptrdiff_t old_errfunc;
85             lu_byte old_allowhook;
86             lu_byte status;
87         } c;
88     } u;
89 } CallInfo;

```

CallInfo 保存着正在调用的函数的运行状态。状态标示存放在 callstatus 中。部分数据和函数的类型有关，以联合形式存放。C 函数与 Lua 函数的结构不完全相同。callstatus 中保存了一位标志用来区分是 C 函数还是 Lua 函数。⁶

```

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

```

正在调用的函数一定存在于数据栈上，在 CallInfo 结构中，func 指向正在执行的函数在数据栈上的位置。需要记录这个信息，是因为如果当前是一个 Lua 函数，且传入的参数个数不定的时候，需要用这个位置和当前数据栈底的位置相减，获得不定参数的准确数量⁷。

同时，func 还可以帮助我们调试嵌入式 Lua 代码。

在用 gdb 这样的调试器调试代码时，可以方便的查看 C 中的调用栈信息，但一旦嵌入 Lua，我们很难理解运行过程中的 Lua 代码的调用栈。不理解 Lua 的内部结构，就可能面对一个简单的 lua_State L 变量束手无策。

⁶在 Lua 5.1 中，没有 callstatus 这个域。而是通过访问 func 引用的函数对象来了解函数是 C 函数还是 Lua 函数。有 callstatus 后，目前 5.2 的版本少一次间接内存访问，要略微高效一些。

⁷参见：7.2.7 节。

实际上，遍历 L 中的 ci 域指向的 CallInfo 链表可以获得完整的 Lua 调用链。而每一级的 CallInfo 中，都可以进一步的通过 func 域取得所在函数的更详细信息。当 func 为一个 Lua 函数时，根据它的函数原型⁸可以获得源文件名、行号等诸多调试信息。

下面展示的是一个粗略的 gdb 脚本，可以利用当前栈上的 L 分析出 Lua 的调用栈。⁹

```

1  define btlua
2      set $p = L.ci
3      while ($p != 0 )
4          set $tt = ($p.func.tt_ & 0x3f)
5          if ( $tt == 0x06 )
6              set $proto = $p.func.value_.gc.cl.l.p
7              set $filename = (char*)&($proto.source.tsv) + 1)
8              set $lineno = $proto.lineinfo[ $p.u.l.savedpc - $proto.code -1 ]
9              printf "0x%x_LUA_FUNCTION:_%4d%s\n", $p, $lineno, $filename
10
11             set $p = $p.previous
12             loop_continue
13         end
14
15         if ( $tt == 0x16 )
16             printf "0x%x_LIGHT_C_FUNCTION", $p
17             output $p.func.value_.f
18             printf "\n"
19
20             set $p = $p.previous
21             loop_continue
22         end
23
24         if ( $tt == 0x26 )
25             printf "0x%x_C_FUNCTION", $p
26             output $p.func.value_.gc.cl.c.f
27             printf "\n"
28
29             set $p = $p.previous
30             loop_continue
31         end
32
33         printf "0x%x_LUA_BASE\n", $p
34         set $p = $p.previous
35     end

```

⁸在 gdb 中，可以通过 func.value_.gc.cl.l.p 获得等价于 C 代码中 clLvalue(func).p 这个宏的功能。

⁹这段 gdb 脚本是由我的同事阿楠在工作中创作，用于我们一个 C 与 Lua 5.2 混合编程项目的调试。功能虽不算太完善，但基本够用。一旦了解其原理，很容易在其它调试器下实现同样的功能，或编写出适用于 Lua 其它版本的实现。

```
36 end
```

CallInfo 是一个标准的双向链表结构，不直接被 GC 模块管理。这个双向链表表达的是一个逻辑上的栈，在运行过程中，并不是每次调入更深层次的函数，就立刻构造出一个 CallInfo 节点。整个 CallInfo 链表会在运行中被反复复用。直到 GC 的时候才清理那些比当前调用层次更深的无用节点。

lstate.c 中有 luaE_extendCI 的实现：

源代码 6.7: lstate.c: luaE_extendCI

```
112 CallInfo *luaE_extendCI (lua_State *L) {
113     CallInfo *ci = luaM_new(L, CallInfo);
114     lua_assert(L->ci->next == NULL);
115     L->ci->next = ci;
116     ci->previous = L->ci;
117     ci->next = NULL;
118     return ci;
119 }
```

但具体执行部分并不直接调用这个 API 而是使用另一个宏：

源代码 6.8: ldo.c: next_ci

```
289 #define next_ci(L) (L->ci = (L->ci->next ? L->ci->next : luaE_extendCI(L)))
```

也就是说，调用者只需要把 CallInfo 链表当成一个无限长的堆栈使用即可。当调用层次返回，之前分配的节点可以被后续调用行为复用。在 GC 的时候只需要调用 luaE_freeCI 就可以释放过长的链表。

源代码 6.9: lstate.c: luaE_freeCI

```
122 void luaE_freeCI (lua_State *L) {
123     CallInfo *ci = L->ci;
124     CallInfo *next = ci->next;
125     ci->next = NULL;
126     while ((ci = next) != NULL) {
127         next = ci->next;
128         luaM_free(L, ci);
129     }
130 }
```

6.1.3 线程

把数据栈和调用栈合起来就构成了 Lua 中的线程。在同一个 Lua 虚拟机中的不同线程因为共享了 global_State 而很难做到真正意义上的并发。它也绝非操作系统意义上的线程，但在行为上很相似。用户可以 resume 一个线程，线程可以被 yield 打断。Lua 的执行过程就是围绕线程进行的。

我们从 lua_newthread 阅读起，可以更好的理解它的数据结构。

源代码 6.10: lstate.c: lua_newthread

```

233 LUA_API lua_State *lua_newthread (lua_State *L) {
234     lua_State *L1;
235     lua_lock(L);
236     luaC_checkGC(L);
237     L1 = &luaC_newobj(L, LUA_TTHREAD, sizeof(LX), NULL, offsetof(LX, l))->th;
238     setthvalue(L, L->top, L1);
239     api_incr_top(L);
240     preinit_state(L1, G(L));
241     L1->hookmask = L->hookmask;
242     L1->basehookcount = L->basehookcount;
243     L1->hook = L->hook;
244     resethookcount(L1);
245     luai_userstatethread(L, L1);
246     stack_init(L1, L); /* init stack */
247     lua_unlock(L);
248     return L1;
249 }

```

这里我们能发现，内存中的线程结构并非 lua_State，而是一个叫 LX 的东西。

下面来看一看 LX 的定义：

源代码 6.11: lstate.c: LX

```

59 typedef struct LX {
60     #if defined(LUAI_EXTRASPACE)
61         char buff[LUAI_EXTRASPACE];
62     #endif
63     lua_State l;
64 } LX;

```

在 lua_State 之前留出了大小为 LUAI_EXTRASPACE 字节的空间。面对外部用户操作的指针是 L 而不是 LX，但 L 所占据的内存块的前面却是有所保留的。

这是一个有趣的技巧。用户可以在拿到 L 指针后向前移动指针，取得一些 EXTRASPACE 中额外的数据。把这些数据放在前面而不是 lua_State 结构的后面避免了向用户暴露结构的大小。

这里，LUAI_EXTRASPACE 是通过编译配置的，默认为 0。开启 EXTRASPACE 后，还需要定义下列宏来配合工作。

```

1  luai_userstateopen(L)
2  uai_userstateclose(L)
3  luai_userstatethread(L,L1)
4  luai_userstatefree(L,L1)
5  luai_userstateresume(L,n)
6  luai_userstateyield(L,n)

```

给 L 附加一些用户自定义信息在追求性能的环境很有意义。可以在为 Lua 编写的 C 模块中，直接偏移 L 指针来获取一些附加信息。这比去读取 L 中的注册表要高效的多。另一方面，在多线程环境下，访问注册表本身会改变 L 的状态，是线程不安全的。

6.2 线程的执行与中断

Lua 的程序运行是以线程为单位的。每个 Lua 线程可以独立运行直到自行中断，把中断的信息留在状态机中。每条线程的执行互不干扰，可以独立延续之前中断的执行过程。Lua 的线程和系统线程无关，所以不会为每条 Lua 线程创建独立的系统堆栈，而是利用自己维护的线程栈，内存开销也就远小于系统线程。但 Lua 又是一门嵌入式语言，和 C 语言混合编程是一种常态。一旦 Lua 调用的 C 库中企图中断线程，延续它就是一个巨大的难题¹⁰。Lua 5.2 通过一些巧妙的设计绕过了这个问题。

6.2.1 异常处理

如果 Lua 被实现为一个纯粹的运行在字节码虚拟机上的语言¹¹，只要不出虚拟机，可以很容易的实现自己的线程和异常处理。事实上，Lua 函数调用层次上只要没有 C 函数，是不会在 C 层面的调用栈上深入下去的¹²。但当 Lua 函数调用了 C 函数，而这个 C 函数中又进一步的回调了 Lua 函数，这个问题就复杂的多。考虑一下 lua 中的 pairs 函数，就是一个典型的 C 扩展函数，却又回调了 Lua 函数。

Lua 底层把异常和线程中断用同一种机制来处理，也就是使用了 C 语言标准的 longjmp 机制来解决这个问题。

为了兼顾 C++ 开发，在 C++ 环境下，可以把它配置为使用 C++ 的 try / catch 机制。¹³ 这些是通过宏定义来切换的。

源代码 6.12: ldo.c: LUAI_THROW

```

51 #if defined(__cplusplus) && !defined(LUA_USE_LONGJMP)
52 /* C++ exceptions */
53 #define LUAI_THROW(L,c)          throw(c)
54 #define LUAI_TRY(L,c,a) \
55     try { a } catch(...) { if ((c)->status == 0) (c)->status = -1; }
56 #define luai_jmpbuf              int /* dummy variable */
57
58 #elif defined(LUA_USE_ULONGLONG)
59 /* in Unix, try _longjmp/_setjmp (more efficient) */
60 #define LUAI_THROW(L,c)          _longjmp((c)->b, 1)
61 #define LUAI_TRY(L,c,a)          if (_setjmp((c)->b) == 0) { a }
62 #define luai_jmpbuf              jmp_buf
63
64 #else

```

¹⁰这里涉及 C 调用层次上的系统堆栈保存和恢复问题。

¹¹Java 就是这样，不用太考虑和原生代码的交互问题。

¹²一次纯粹的 Lua 函数调用，不会引起一次虚拟机上的 C 函数调用。

¹³用 C++ 开发的 Lua 扩展库，并不一定要链接到 C++ 编译的 Lua 虚拟机才能正常工作。如果你的库不必回调 Lua 函数，就不会有这个烦恼。使用 try/catch 不会比 longjmp 更安全。如果你需要在 C++ 扩展库中调用 lua_error 或 lua_yield 中断 C 函数的运行，那么考虑到可能发生的内存泄露问题，就有必要换成 try / catch 的版本。

```

65 /* default handling with long jumps */
66 #define LUAI_THROW(L,c)      longjmp((c)->b, 1)
67 #define LUAI_TRY(L,c,a)      if (setjmp((c)->b) == 0) { a }
68 #define luai_jmpbuf          jmp_buf
69
70 #endif

```

每条线程 L 中保存了当前的 longjmp 返回点: errorJmp, 其结构定义为 struct lua_longjmp。这是一条链表, 每次运行一段受保护的 Lua 代码, 都会生成一个新的错误返回点, 链到这条链表上。

下面展示的是 lua_longjmp 的结构:

源代码 6.13: ldo.c: lua_longjmp

```

77 struct lua_longjmp {
78     struct lua_longjmp *previous;
79     luai_jmpbuf b;
80     volatile int status; /* error code */
81 };

```

设置 longjmp 返回点是由 luaD_rawrunprotected 完成的。

源代码 6.14: ldo.c: luaD_rawrunprotected

```

125 int luaD_rawrunprotected (lua_State *L, Pfunc f, void *ud) {
126     unsigned short oldnCcalls = L->nCcalls;
127     struct lua_longjmp lj;
128     lj.status = LUA_OK;
129     lj.previous = L->errorJmp; /* chain new error handler */
130     L->errorJmp = &lj;
131     LUAI_TRY(L, &lj,
132         (*f)(L, ud);
133     );
134     L->errorJmp = lj.previous; /* restore old error handler */
135     L->nCcalls = oldnCcalls;
136     return lj.status;
137 }

```

这段代码很容易理解: 设置新的 jmpbuf, 串到链表上, 调用函数。调用完成后恢复进入时的状态。如果想回直接返回到最近的错误恢复点, 只需要调用 longjmp。Lua 用了内部 API luaD_throw 封装了这个过程。

源代码 6.15: ldo.c: luaD_throw

```

103 l_noret luaD_throw (lua_State *L, int errcode) {
104     if (L->errorJmp) { /* thread has an error handler? */
105         L->errorJmp->status = errcode; /* set status */
106         LUAI_THROW(L, L->errorJmp); /* jump to it */

```

```

107 }
108 else { /* thread has no error handler */
109     L->status = cast_byte(errcode); /* mark it as dead */
110     if (G(L)->mainthread->errorJump) { /* main thread has a handler? */
111         setobjs2s(L, G(L)->mainthread->top++, L->top - 1); /* copy error obj
        . */
112         luaD_throw(G(L)->mainthread, errcode); /* re-throw in main thread */
113     }
114     else { /* no handler at all; abort */
115         if (G(L)->panic) { /* panic function? */
116             lua_unlock(L);
117             G(L)->panic(L); /* call it (last chance to jump out) */
118         }
119         abort();
120     }
121 }
122 }

```

考虑到新构造的线程可能在不受保护的情况下运行¹⁴。这时的任何错误都必须被捕获，不能让程序崩溃。这种情况合理的处理方式就是把正在运行的线程标记为死线程，并且在主线程中抛出异常。

6.2.2 函数调用

函数调用分为受保护调用和不受保护的调用。

受保护的函数调用可以看成是一个 C 层面意义上完整的过程。在 Lua 代码中，pcall 是用函数而不是语言机制完成的。受保护的函数调用一定在 C 层面进出了一次调用栈。它使用独立的一个内部 API luaD_pcall 来实现。公开 API 仅需要对它做一些封装即可。¹⁵

源代码 6.16: ldo.c: luaD_pcall

```

587 int luaD_pcall (lua_State *L, Pfunc func, void *u,
588                ptrdiff_t old_top, ptrdiff_t ef) {
589     int status;
590     CallInfo *old_ci = L->ci;
591     lu_byte old_allowhooks = L->allowhook;
592     unsigned short old_nny = L->nny;
593     ptrdiff_t old_errfunc = L->errfunc;
594     L->errfunc = ef;
595     status = luaD_rawrunprotected(L, func, u);
596     if (status != LUA_OK) { /* an error occurred? */

```

¹⁴这一般是错误的使用 Lua 导致的。正确使用 Lua 线程的方式是把主函数压入新线程，然后调用 lua_resume 启动它。而 lua_resume 可以保护线程的主函数运行。

¹⁵luaD_pcall 主要被用来实现外部 API lua_pcallk。在 lua 虚拟机的实现中并不需要这个 API，这是因为 pcall 是作用标准库函数引入 lua 的，并非一个语言特性。但也并非 lua 核心代码不需要 luaD_pcall，在 GC 过程中释放对象，调用对象的 GC 元方法时需要使用 luaD_pcall。因为 GC 的发生时刻不可预知，不能让 GC 流程干扰了正常代码的运行，必须把对 GC 元方法的调用保护起来。

```

597     StkId oldtop = restorestack(L, old_top);
598     luaF_close(L, oldtop); /* close possible pending closures */
599     seterrorobj(L, status, oldtop);
600     L->ci = old_ci;
601     L->allowhook = old_allowhooks;
602     L->nny = old_nny;
603     luaD_shrinkstack(L);
604 }
605 L->errfunc = old_errfunc;
606 return status;
607 }

```

从这段代码我们可以看到 pcall 的处理模式：用 C 层面的堆栈来保存和恢复状态。ci、allowhooks、nny¹⁶、errfunc 都保存在 luaD_pcall 的 C 堆栈上，一旦 luaD_rawrunprotected 就可以正确恢复。

luaD_rawrunprotected 没有正确返回时，需要根据 old_top 找到堆栈上刚才调用的函数，给它做收尾工作¹⁷。因为 luaD_rawrunprotected 调用的是一个函数对象，而不是数据栈上的索引，这就需要额外的变量来定位了。

这里使用 restorestack 这个宏来定位栈上的地址，是因为数据栈的内存地址是会随着数据栈的大小而变化。保存地址是不可能的，而应该记住一个相对量。savestack 和 restorestack 两个宏就是做这个工作的。

源代码 6.17: ldo.h: restorestack

```

22 #define savestack(L,p)          (((char *) (p)) - ((char *) L->stack))
23 #define restorestack(L,n)      (((TValue *) ((char *) L->stack + (n))))

```

一般的 Lua 层面的函数调用并不对应一个 C 层面上函数调用行为。对于 Lua 函数而言，应该看成是生成新的 CallInfo，修正数据栈，然后把字节码的执行位置跳转到被调用的函数开头。而 Lua 函数的 return 操作则做了相反的操作，恢复数据栈，弹出 CallInfo，修改字节码的执行位置，恢复到原有的执行序列上。

理解了这一点，就能明白，在底层 API 中，为何分为 luaD_precall 和 luaD_poscall 两个了。

luaD_precall 执行的是函数调用部分的工作，而 luaD_poscall 做的是函数返回的工作。从 C 层面看，层层函数调用的过程并不是递归的。对于 C 类型的函数调用，整个函数调用是完整的，不需要等待后续再调用 luaD_poscall，所以 luaD_precall 可以代其完成，并返回 1；而 Lua 函数，执行完 luaD_precall 后，只是切换了 lua_State 的执行状态，被调用的函数的字节码尚未运行，luaD_precall 返回 0。待到虚拟机执行到对应的 return 指令时，才会去调用 luaD_poscall 完成整次调用。

我们先看看 luaD_precall 的源码：

源代码 6.18: ldo.c: luaD_precall

```

295 int luaD_precall (lua_State *L, StkId func, int nresults) {
296     lua_CFunction f;
297     CallInfo *ci;
298     int n; /* number of arguments (Lua) or returns (C) */

```

¹⁶nny 的全称是 number of non-yieldable calls。C 语言本身无法提供延续点的支持，所以 Lua 也无法让所有函数都是 yieldable 的。当一级函数处于 non-yieldable 状态时，更深的层次都无法 yieldable。这个变量用于监督这个状态，在错误发生时报告。每级 C 调用是否允许 yield 取决于是否有设置 C 延续点，或是 Lua 内核实现时认为这次调用在发生 yield 时无法正确处理。这些都由 luaD_call 的最后一个参数来制定。

¹⁷调用 luaF_close 涉及 upvalue 的 gc 流程，参见 GC 的章节。

```

299 ptrdiff_t funcr = savestack(L, func);
300 switch (ttype(func)) {
301     case LUA_TLCF: /* light C function */
302         f = fvalue(func);
303         goto Cfunc;
304     case LUA_TCCL: { /* C closure */
305         f = clCvalue(func)->f;
306         Cfunc:
307         luaD_checkstack(L, LUA_MINSTACK); /* ensure minimum stack size */
308         ci = next_ci(L); /* now 'enter' new function */
309         ci->nresults = nresults;
310         ci->func = restorestack(L, funcr);
311         ci->top = L->top + LUA_MINSTACK;
312         lua_assert(ci->top <= L->stack_last);
313         ci->callstatus = 0;
314         luaC_checkGC(L); /* stack grow uses memory */
315         if (L->hookmask & LUA_MASKCALL)
316             luaD_hook(L, LUA_HOOKCALL, -1);
317         lua_unlock(L);
318         n = (*f)(L); /* do the actual call */
319         lua_lock(L);
320         api_checknelems(L, n);
321         luaD_poscall(L, L->top - n);
322         return 1;
323     }
324     case LUA_TLCL: { /* Lua function: prepare its call */
325         StkId base;
326         Proto *p = clLvalue(func)->p;
327         luaD_checkstack(L, p->maxstacksize);
328         func = restorestack(L, funcr);
329         n = cast_int(L->top - func) - 1; /* number of real arguments */
330         for (; n < p->numparams; n++)
331             setnilvalue(L->top++); /* complete missing arguments */
332         base = (!p->is_vararg) ? func + 1 : adjust_varargs(L, p, n);
333         ci = next_ci(L); /* now 'enter' new function */
334         ci->nresults = nresults;
335         ci->func = func;
336         ci->u.l.base = base;
337         ci->top = base + p->maxstacksize;
338         lua_assert(ci->top <= L->stack_last);
339         ci->u.l.savedpc = p->code; /* starting point */

```

```

340     ci->callstatus = CIST_LUA;
341     L->top = ci->top;
342     luaC_checkGC(L); /* stack grow uses memory */
343     if (L->hookmask & LUA_MASKCALL)
344         callhook(L, ci);
345     return 0;
346 }
347 default: { /* not a function */
348     func = tryfuncTM(L, func); /* retry with 'function' tag method */
349     return luaD_precall(L, func, nresults); /* now it must be a function
        */
350 }
351 }
352 }

```

Light C function 和 C closure 仅仅是在储存上有所不同，处理逻辑是一致的：压入新的 CallInfo，把数据栈栈顶设置好。调用 C 函数，然后 luaD_poscall。

Lua 函数要复杂一些：先通过传入函数对象在数据栈上的位置和栈顶差，计算出数据栈上的调用参数个数 n 。如果 Lua 函数对输入参数个数有明确的最小要求，这点可以通过查询函数原型的 numparams 字段获知；若栈上提供的参数数量不足，就需要把不足的部分补为 nil。当调用函数需要可变参数的时候，还需要进一步处理：

源代码 6.19: ldo.c: adjust_varargs

```

257 static StkId adjust_varargs (lua_State *L, Proto *p, int actual) {
258     int i;
259     int nfixargs = p->numparams;
260     StkId base, fixed;
261     lua_assert(actual >= nfixargs);
262     /* move fixed parameters to final position */
263     fixed = L->top - actual; /* first fixed argument */
264     base = L->top; /* final position of first argument */
265     for (i=0; i<nfixargs; i++) {
266         setobjs2s(L, L->top++, fixed + i);
267         setnilvalue(fixed + i);
268     }
269     return base;
270 }

```

变长参数表这个概念，只在 Lua 函数中出现。当一个函数接收变长参数时，这部分的参数是放在上一级数据栈帧尾部的。adjust_varargs 将需要个固定参数复制到被调用的函数的新一级数据栈帧上，而变长参数留在原地。

接下来，要构造出新的一层调用栈 CallInfo。这个结构需要初始化字节码执行指针 savedpc，将其指向 lua 函数对象中的字节指令区。Lua 函数整体所需要的栈空间是在生成字节码时就已知的，所以可以用

luaD_checkstack 一次性分配好。CallInfo 中的栈顶可以直接调整到位¹⁸。CallInfo 中的返回参数个数，所引用的函数对象——初始化完毕，最后初始化线程运行状态 callstatus，标记上 CIST_LUA 就够了。

真正如何运行 Lua 函数，是由调用 luaD_precall 者决定的。

有些对象是通过元表驱动函数调用的行为的，这时需要通过 tryfuncTM 函数取得真正的调用函数。¹⁹

源代码 6.20: ldo.c: tryfuncTM

```

273 static StkId tryfuncTM (lua_State *L, StkId func) {
274     const TValue *tm = luaT_gettmbyobj(L, func, TM_CALL);
275     StkId p;
276     ptrdiff_t funcr = savestack(L, func);
277     if (!ttisfunction(tm))
278         luaG_typeerror(L, func, "call");
279     /* Open a hole inside the stack at 'func' */
280     for (p = L->top; p > func; p--) setobjs2s(L, p, p-1);
281     incr_top(L);
282     func = restorestack(L, funcr); /* previous call may change stack */
283     setobj2s(L, func, tm); /* tag method is the new function to be called */
284     return func;
285 }

```

根据 lua 的定义，通过元方法进行的函数调用和原生的函数调用有所区别。通过元方法进行的函数调用，会将对象自身作为第一个参数传入。这就需要移动数据栈，把对象插到第一个参数的位置。这个过程在源码中有清晰的展示。

luaD_poscall 做的工作也很简单，主要是数据栈的调整工作。

源代码 6.21: ldo.c: luaD_poscall

```

355 int luaD_poscall (lua_State *L, StkId firstResult) {
356     StkId res;
357     int wanted, i;
358     CallInfo *ci = L->ci;
359     if (L->hookmask & (LUA_MASKRET | LUA_MASKLINE)) {
360         if (L->hookmask & LUA_MASKRET) {
361             ptrdiff_t fr = savestack(L, firstResult); /* hook may change stack
362                                                         */
363             luaD_hook(L, LUA_HOOKRET, -1);
364             firstResult = restorestack(L, fr);
365         }
366         L->oldpc = ci->previous->u.l.savedpc; /* 'oldpc' for caller function
367                                                         */
368     }

```

¹⁸Lua 5 以后的虚拟机采用寄存器式设计，把栈空间当成若干个寄存器使用，所以在虚拟机运行一个代码块的过程中，几乎没有修改栈顶指针的操作。

¹⁹关于元表查询 luaT_gettmbyobj 的部分，参见 4.4 节。


```

367     res = ci->func;  /* res == final position of 1st result */
368     wanted = ci->nresults;
369     L->ci = ci = ci->previous;  /* back to caller */
370     /* move results to correct place */
371     for (i = wanted; i != 0 && firstResult < L->top; i--)
372         setobjs2s(L, res++, firstResult++);
373     while (i-- > 0)
374         setnilvalue(res++);
375     L->top = res;
376     return (wanted - LUA_MULTRET);  /* 0 iff wanted == LUA_MULTRET */
377 }

```

根据 `luaD_precall` 设置在 `CallInfo` 里的返回参数的个数 `nresults`，以及数据栈上在这次函数调用中实际新增的数据个数，需要对数据栈做一次调整。多余的抛弃，不足的补为 `nil`。

读完这些，再来看 `luaD_call` 就很清晰了。

`luaD_call` 主要用来实现外部 API `lua_callk`。它调用完 `luaD_precall` 后，接着调用 `luaV_execute` 完成对函数本身的字节码执行。`luaD_call` 的最后一个参数用来标示这次调用是否可以在其中挂起。因为在 lua 虚拟机执行期间，有许多情况都会引起新的一层 C 层面的函数调用。Lua 线程并不拥有独立的 C 堆栈，所以对于发生在 C 函数内部的线程挂起操作，不是所有的情况都正确处理的。是否接受 `yield` 操作，只有调用 `luaD_call` 才清楚。

源代码 6.22: `ldo.c: luaD_call`

```

386 void luaD_call (lua_State *L, StkId func, int nResults, int allowyield) {
387     if (++L->nCcalls >= LUAI_MAXCCALLS) {
388         if (L->nCcalls == LUAI_MAXCCALLS)
389             luaG_runerror(L, "C_stack_overflow");
390         else if (L->nCcalls >= (LUAI_MAXCCALLS + (LUAI_MAXCCALLS >> 3)))
391             luaD_throw(L, LUA_ERRERR);  /* error while handing stack error */
392     }
393     if (!allowyield) L->nny++;
394     if (!luaD_precall(L, func, nResults))  /* is a Lua function? */
395         luaV_execute(L);  /* call it */
396     if (!allowyield) L->nny--;
397     L->nCcalls--;
398 }

```

如前面所述，lua 虚拟机在解析字节码执行的过程中，对 Lua 函数的调用并不直接使用 `luaD_call`。它不会产生 C 层面的函数调用行为，就可以尽量不引起 C 函数中挂起线程的问题。但在某些情况上的处理，也这么做的话会让虚拟机的实现变得相当复杂。这些情况包括 `for` 语句引起的函数调用以及触发元方法引起的函数调用。Lua 利用 `luaD_call`，可以简化实现。

6.2.3 钩子

Lua 可以为每个线程设置一个钩子函数，用于调试、统计和其它一些特殊用法。²⁰

钩子函数是一个 C 函数，用内部 API `luaD_hook` 封装起来。

源代码 6.23: ldo.c: `luaD_hook`

```

217 void luaD_hook (lua_State *L, int event, int line) {
218     lua_Hook hook = L->hook;
219     if (hook && L->allowhook) {
220         CallInfo *ci = L->ci;
221         ptrdiff_t top = savestack(L, L->top);
222         ptrdiff_t ci_top = savestack(L, ci->top);
223         lua_Debug ar;
224         ar.event = event;
225         ar.currentline = line;
226         ar.i_ci = ci;
227         luaD_checkstack(L, LUA_MINSTACK); /* ensure minimum stack size */
228         ci->top = L->top + LUA_MINSTACK;
229         lua_assert(ci->top <= L->stack_last);
230         L->allowhook = 0; /* cannot call hooks inside a hook */
231         ci->callstatus |= CIST_HOOKED;
232         lua_unlock(L);
233         (*hook)(L, &ar);
234         lua_lock(L);
235         lua_assert(!L->allowhook);
236         L->allowhook = 1;
237         ci->top = restorestack(L, ci_top);
238         L->top = restorestack(L, top);
239         ci->callstatus &= ~CIST_HOOKED;
240     }
241 }
242
243
244 static void callhook (lua_State *L, CallInfo *ci) {
245     int hook = LUA_HOOKCALL;
246     ci->u.l.savedpc++; /* hooks assume 'pc' is already incremented */
247     if (isLua(ci->previous) &&

```

这里做的事情并不复杂：保存数据栈、构造调试信息、通过设置 `allowhook` 禁掉钩子的递归调用，然后调用 C 版本的钩子函数，完毕后恢复这些状态。²¹

²⁰我们可以给一个线程按指令执行条数设置钩子，并在钩子函数中 `yield` 出线程。只需要在外部做一个线程调度器，这样就用 lua 模拟了一个抢先式的多线程模型。每个 `coroutine` 并不需要主动调用 `yield` 挂起自己、钩子会定期做 `yield` 操作。笔者曾经在 Lua 5.2 的 beta 期，尝试过这个想法，细节可以参考 blog http://blog.codingnow.com/2011/11/ameba_lua_52.html。

²¹此处遗留了一个小问题，是关于 lua 的尾调用优化时，钩子的正确处理。

6.2.4 从 C 函数中挂起线程

Lua 5.2 中挂起一个线程和延续之前挂起线程的过程，远比 lua 5.2 之前的版本要复杂的多。在阅读这部分代码前，必须先展开一个话题：

Lua 5.2 解决的一个关键问题是，如何让 C 函数正确的配合工作。C 语言是不支持延续点这个特性的。如果你从 C 函数中利用 `longjmp` 跳出，就再也回不到跳出点了。这对 Lua 工作在虚拟机字节码上的大部分特性来说，都不是问题。但是，`pcall` 和元表都涉及 C 函数调用，有这样的限制，让 lua 不那么完整。Lua 5.2 应用一系列的技巧，绕开了这个限制，支持了 `yieldable pcall and metamethods`。

在 Lua 5.2 的文档中，我们可以找到这么一小节：`Handling Yields in C` 就是围绕解决这个难题展开的。首先我们来看看问题的产生：

`resume` 的发起总是通过一次 `lua_resume` 的调用，在 Lua 5.1 以前，`yield` 的调用必定结束于一次 `lua_yield` 调用，而调用它的 C 函数必须立刻返回。中间不能有任何 C 函数执行到中途的状态。这样，Lua VM 才能正常工作。

(C)lua_resume → Lua functions → `coroutine.yield` → (C)lua_yield → (C) return

在这个流程中，无论 Lua functions 有多少层，都被 `lua_State` 的调用栈管理。所以当最后 C return 返回到最初 resume 点，都不存在什么问题，可以让下一次 resume 正确继续。也就是说，在 yield 时，lua 调用栈上可以有没有执行完的 lua 函数，但不可以有没有执行完的 C 函数。

如果我们写了这么一个 C 扩展，在 C function 里回调了传入的一个 Lua 函数。情况就变得不一样了。

(C)lua_resume → Lua function → C function → (C) lua_call → Lua function → `coroutine.yield` → (C)lua_yield

C 通过 `lua_call` 调用的 Lua 函数中再调用 `coroutine.yield` 会导致在 yield 之后，再次 resume 时，不再可能从 `lua_call` 的下一行继续运行。lua 在遇到这种情况时，会抛出一个异常 “attempt to yield across metamethod/C-call boundary”。

在 5.2 之前，有人试图解决这个问题，去掉 `coroutine` 的这些限制。比如 `Coco`²² 这个项目。它用操作系统的协程来解决这个问题²³，即给每个 lua `coroutine` 真的附在一个 C 协程上，独立一个 C 堆栈。

这样的方案开销较大，且依赖平台特性。

那么，在 C 和 Lua 的边界，如果在 yield 之后，resume 如何继续运行 C 边界之后的 C 代码？所有 Lua 线程共用了一个 C 堆栈。可以使用 `longjmp` 从调用深处跳出来，却无法回到那个位置。因为一旦跳出，堆栈就被破坏。C 进入 Lua 的边界一共有四个 API：`lua_call`，`lua_pcall`，`lua_resume` 和 `lua_yield`。其中要解决的关键问题在于调用一个 lua 函数，却可能有两条返回路径。

lua 函数的正常返回应该执行 `lua_call` 调用后面的 C 代码，而中途如果 yield 发生，会导致执行序回到前面 `lua_resume` 调用处的下一行 C 代码执行。对于后一种，再次调用 `lua_resume`，还需要回到 `lua_call` 之后完成后续的 C 执行逻辑。C 语言是不允许这样做的，因为当初的 C 堆栈已经不存在了。

Lua 5.2 改造了 API `lua_callk` 来解决这个问题。既然在 yield 之后，C 的执行序无法回到 `lua_callk` 的下一行代码，那么就让 C 语言使用者自己提供一个 Continuation 函数 k 来继续。

我们可以这样理解 k 这个参数：当 `lua_callk` 调用的 lua 函数中没有发生 yield 时，它会正常返回。一旦发生 yield，调用者要明白，C 代码无法正常延续，而 lua vm 会在需要延续时调用 k 来完成后续工作。k 会得到正确的 L 保持正确的 lua state 状态，看起来就好像用一个新的 C 执行序替代掉原来的 C 执行序一样。

一个容易理解的用法就是在一个 C 函数调用的最后使用 `lua_callk`：

```
1 lua_callk(L, 0, LUA_MULTRET, 0, k);
```

²²CoCo - True C Coroutines for Lua <http://coco.luajit.org>

²³CoCo 在 Windows 上使用了 Fiber。

```
2 return k(L);
```

也就是把 `callk` 后面的执行逻辑放在一个独立 C 函数 `k` 中，分别在 `callk` 后调用它，或是传递给框架，让框架在 `resume` 后调用。这里 lua 状态机的状态被正确保存在 `L` 中，而 C 函数堆栈会在 `yield` 后被破坏掉。如果我们需要在 `k` 中得到延续点前的 C 函数状态怎么办呢？lua 提供了 `ctx` 用于辅助记录 C 中的状态。在 `k` 中，可以通过 `lua_getctx` 获得最近一次边界调用时传入的 `ctx`。

同时，`lua_getctx` 还会返回原始函数应该返回的值。这是因为切换到延续点函数中后，原始函数就无法返回了。利用 `lua_getctx` 就取到 `lua_pcallk` 原本应该返回的值。对于没有返回值的原始函数 `lua_callk`，它可以返回 `LUA_YIELD` 用来标识执行序曾被切出，又回到了延续状态中。

Lua 的线程结构 `L` 中保有完整的 `CallInfo` 调用栈。当 C 层面的调用栈被破坏时，尚未返回的 C 函数会于切入 lua 虚拟机前在 `CallInfo` 中留下延续点函数。原本在 C 层面利用原生代码和系统提供的 C 堆栈维系的 C 函数调用线索，被平坦化为 `L` 里 `CallInfo` 中的一个个延续点函数。想延续一个 C 调用栈被破坏掉的 lua 线程，只需要依次调用 `CallInfo` 中的延续点函数就能完成同样的执行逻辑。

6.2.5 挂起与延续

理解了 Lua 5.2 对线程挂起和延续的处理方式，再来看相关代码要容易理解一些。

中断并挂起一个线程和线程的执行发生异常，这两种情况对 lua 虚拟机的执行来说是类似的。都是利用 `luaD_throw` 回到最近的保护点。不同的是，线程的状态不同。主动挂起需要调用 API `lua_yieldk`，把当前执行处的函数保存到 `CallInfo` 的 `extra` 中，并设置线程状态为 `LUA_YIELD`，然后抛出异常。和异常抛出不同，`yield` 只可能被 `lua_yieldk` 触发，这是一个 C API，而不是 lua 的虚拟机的指令。也就是说，`yield` 必然来源于某次 C 函数调用，从 `luaD_call` 或 `luaD_pcall` 中退出的。这比异常的发生点要少的多。

下面先看看 `lua_yieldk` 的实现。

源代码 6.24: `ldo.c: lua_yieldk`

```
559 LUA_API int lua_yieldk (lua_State *L, int nresults, int ctx, lua_CFunction
    k) {
560     CallInfo *ci = L->ci;
561     luai_userstateyield(L, nresults);
562     lua_lock(L);
563     api_checknelems(L, nresults);
564     if (L->nny > 0) {
565         if (L != G(L)->mainthread)
566             luaG_runerror(L, "attempt to yield across a C-call boundary");
567         else
568             luaG_runerror(L, "attempt to yield from outside a coroutine");
569     }
570     L->status = LUA_YIELD;
571     ci->extra = savestack(L, ci->func); /* save current 'func' */
572     if (isLua(ci)) { /* inside a hook? */
573         api_check(L, k == NULL, "hooks cannot continue after yielding");
574     }
575     else {
```

```

576     if ((ci->u.c.k = k) != NULL) /* is there a continuation? */
577         ci->u.c.ctx = ctx; /* save context */
578     ci->func = L->top - nresults - 1; /* protect stack below results */
579     luaD_throw(L, LUA_YIELD);
580 }
581 lua_assert(ci->callstatus & CIST_HOOKED); /* must be inside a hook */
582 lua_unlock(L);
583 return 0; /* return to 'luaD_hook' */
584 }

```

不是所有的 C 函数都可以正常恢复，只要调用层次上面有一个这样的 C 函数，yield 就无法正确工作。这是由 nny 的值来检测的。关于 nny 参见 6.2.2 节的脚注。

lua_yieldk 是一个公开 API，只用于给 Lua 程序编写 C 扩展模块使用。所以处于这个函数内部时，一定处于一个 C 函数调用中。但钩子函数的运行是个例外。钩子函数本身是一个 C 函数，但是并不是通常正常的 C API 调用进来的。在 Lua 函数中触发钩子会认为当前状态是处于 Lua 函数执行中。这个时候允许 yield 线程，但无法正确的处理 C 层面的延续点。所以禁止传入延续点函数。而对于正常的 C 调用，允许修改延续点 k 来改变执行流程。这里只需要简单的把 k 和 ctx 设入 L，其它的活交给 resume 去处理就可以了。

lua_resume 的过程要复杂的多。先列出代码，再慢慢分析。

源代码 6.25: ldo.c: lua_resume

```

528 LUA_API int lua_resume (lua_State *L, lua_State *from, int nargs) {
529     int status;
530     lua_lock(L);
531     luai_userstateresume(L, nargs);
532     L->nCcalls = (from) ? from->nCcalls + 1 : 1;
533     L->nny = 0; /* allow yields */
534     api_checknelems(L, (L->status == LUA_OK) ? nargs + 1 : nargs);
535     status = luaD_rawrunprotected(L, resume, L->top - nargs);
536     if (status == -1) /* error calling 'lua_resume'? */
537         status = LUA_ERRRUN;
538     else { /* yield or regular error */
539         while (status != LUA_OK && status != LUA_YIELD) { /* error? */
540             if (recover(L, status)) /* recover point? */
541                 status = luaD_rawrunprotected(L, unroll, NULL); /* run
                    continuation */
542             else { /* unrecoverable error */
543                 L->status = cast_byte(status); /* mark thread as 'dead' */
544                 seterrorobj(L, status, L->top);
545                 L->ci->top = L->top;
546                 break;
547             }
548         }

```

```

549     lua_assert(status == L->status);
550 }
551 L->nny = 1;  /* do not allow yields */
552 L->nCcalls--;
553 lua_assert(L->nCcalls == ((from) ? from->nCcalls : 0));
554 lua_unlock(L);
555 return status;
556 }

```

lua_resume 开始运行时，等价于一次保护性调用。所以它是允许直接调用的 C 函数 yield 的。这里把 nny 设置为 0 开启。然后利用对 resume 函数的保护调用来进行前半段工作。

源代码 6.26: ldo.c: resume

```

489 static void resume (lua_State *L, void *ud) {
490     int nCcalls = L->nCcalls;
491     StkId firstArg = cast(StkId, ud);
492     CallInfo *ci = L->ci;
493     if (nCcalls >= LUAI_MAXCCALLS)
494         resume_error(L, "C_stack_overflow", firstArg);
495     if (L->status == LUA_OK) { /* may be starting a coroutine */
496         if (ci != &L->base_ci) /* not in base level? */
497             resume_error(L, "cannot_resume_non-suspended_coroutine", firstArg);
498         /* coroutine is in base level; start running it */
499         if (!luaD_precall(L, firstArg - 1, LUA_MULTRET)) /* Lua function? */
500             luaV_execute(L); /* call it */
501     }
502     else if (L->status != LUA_YIELD)
503         resume_error(L, "cannot_resume_dead_coroutine", firstArg);
504     else { /* resuming from previous yield */
505         L->status = LUA_OK;
506         ci->func = restorestack(L, ci->extra);
507         if (isLua(ci)) /* yielded inside a hook? */
508             luaV_execute(L); /* just continue running Lua code */
509         else { /* 'common' yield */
510             if (ci->u.c.k != NULL) { /* does it have a continuation? */
511                 int n;
512                 ci->u.c.status = LUA_YIELD; /* 'default' status */
513                 ci->callstatus |= CIST_YIELDED;
514                 lua_unlock(L);
515                 n = (*ci->u.c.k)(L); /* call continuation */
516                 lua_lock(L);
517                 api_checknelems(L, n);

```



```

518     firstArg = L->top - n;  /* yield results come from continuation */
519 }
520     luaD_poscall(L, firstArg);  /* finish 'luaD_precall' */
521 }
522     unroll(L, NULL);
523 }
524     lua_assert(nCcalls == L->nCcalls);
525 }

```

如果 resume 是重新启动一个函数，那么只需要按和 luaD_call 相同的正常的调用流程进行。

若需要延续之前的调用，如上文所述，之前只可能从一次 C 调用中触发 lua_yieldk 挂起。但钩子函数是一个特殊情况，它是一个 C 函数，却看起来在 Lua 中。这时从 CallInfo 中的 extra 取出上次运行到的函数，可以识别出这个情况。当它是一个 Lua 调用，那么必然是从钩子函数中切出的，不会有被打断的虚拟机指令，直接通过 luaV_execute 继续它的字节码解析执行流程。若是 C 函数，按照延续点的约定，调用延续点 k，之后经过 luaD_poscall 完成这次调用。

这所有事情做完之后，不一定完成了所有的工作。这是因为之前完整的调用层次，包含在 L 的 CallInfo 中，而不是存在于当前的 C 调用栈上。如果检查到 lua 的调用栈上有未尽的工作，必须完成它。这项工作可通过 unroll 函数完成。

源代码 6.27: ldo.c: unroll

```

426 static void unroll (lua_State *L, void *ud) {
427     UNUSED(ud);
428     for (;;) {
429         if (L->ci == &L->base_ci)  /* stack is empty? */
430             return;  /* coroutine finished normally */
431         if (!isLua(L->ci))  /* C function? */
432             finishCcall(L);
433         else {  /* Lua function */
434             luaV_finishOp(L);  /* finish interrupted instruction */
435             luaV_execute(L);  /* execute down to higher C 'boundary' */
436         }
437     }
438 }

```

unroll 发现 L 中的当前函数如果是一个 Lua 函数时，由于字节码的解析过程也可能因为触发元方法等情况调用 luaD_call 而从中间中断。故需要先调用 luaV_finishOp²⁴，再交到 luaV_execute 来完成未尽的字节码。

当执行流中断于一次 C 函数调用，finishCcall 函数能完成当初执行了一半的 C 函数的剩余工作。

源代码 6.28: ldo.c: finishCcall

```

401 static void finishCcall (lua_State *L) {
402     CallInfo *ci = L->ci;

```

²⁴参见??一节。

```

403     int n;
404     lua_assert(ci->u.c.k != NULL); /* must have a continuation */
405     lua_assert(L->nny == 0);
406     if (ci->callstatus & CIST_YPCALL) { /* was inside a pcall? */
407         ci->callstatus &= ~CIST_YPCALL; /* finish 'lua_pcall' */
408         L->errfunc = ci->u.c.old_errfunc;
409     }
410     /* finish 'lua_callk'/'lua_pcall' */
411     adjustresults(L, ci->nresults);
412     /* call continuation function */
413     if (!(ci->callstatus & CIST_STAT)) /* no call status? */
414         ci->u.c.status = LUA_YIELD; /* 'default' status */
415     lua_assert(ci->u.c.status != LUA_OK);
416     ci->callstatus = (ci->callstatus & ~(CIST_YPCALL | CIST_STAT)) |
        CIST_YIELDED;
417     lua_unlock(L);
418     n = (*ci->u.c.k)(L);
419     lua_lock(L);
420     api_checknelems(L, n);
421     /* finish 'luaD_precall' */
422     luaD_poscall(L, L->top - n);
423 }

```

前面曾提到过，此时线程一定处于健康的状态。那么之前的工作肯定中止于 `lua_callk` 或 `lua_pcallk`。这时，先应该完成 `lua_callk` 没完成的工作²⁵：`adjustresults(L, ci->nresults);`；然后调用 C 函数中设置的延续点函数；由于这是一次未完成的 C 函数调用，那么一定来源于一次被中断的 `luaD_precall`，收尾的工作还剩下 `luaD_poscall(L, L->top - n);`。

当 `resume` 这前半段工作完成，结果要么是一切顺利，状态码为 `LUA_OK` 结束或是 `LUA_YIELD` 主动挂起。那么就没有太多剩下的工作。L 的状态是完全正常的。可当代码中有错误发生时，问题就复杂一些。

从定义上说，`lua_resume` 需要具有捕获错误的能力。同样有这个能力的还有 `lua_pcallk`。如果在调用栈上，有 `lua_pcallk` 优先于它捕获错误，那么执行流应该交到 `lua_pcallk` 之后，也就是 `lua_pcallk` 设置的延续点函数。²⁶对 `lua_resume` 来说，错误被 `lua_pcallk` 捕获了，程序应该继续运行。它就有责任完成延续点的约定。这是用 `recover` 和 `unroll` 函数完成的。

源代码 6.29: `ldo.c: recover`

```

444 static CallInfo *findpcall (lua_State *L) {
445     CallInfo *ci;
446     for (ci = L->ci; ci != NULL; ci = ci->previous) { /* search for a pcall
        */

```

²⁵`lua_callk` 和 `lua_pcallk` 在调用完 `luaD_call` 后，后续的代码没有区别，都是 `adjustresults(L, ci->nresults);`，可以一致对待。

²⁶被 `resume` 保护起来的执行流程中再调用 `lua_pcallk`，它是不能直接使用 `setjmp` 设置保护点的。因为如果在这里设置 `setjmp`，有更深层次的函数调用了 `yield` 的话。再次 `resume` 时，就在丢失了最初的 C 调用栈同时，丢失了之前由 `lua_pcallk` 设置的保护点。缺乏这个设计，导致在 Lua 5.1 以前无法在 `pcall` 的函数中使用 `yield`。


```

447     if (ci->callstatus & CIST_YPCALL)
448         return ci;
449 }
450 return NULL; /* no pending pcall */
451 }
452
453
454 static int recover (lua_State *L, int status) {
455     StkId oldtop;
456     CallInfo *ci = findpcall(L);
457     if (ci == NULL) return 0; /* no recovery point */
458     /* "finish" luaD_pcall */
459     oldtop = restorestack(L, ci->extra);
460     luaF_close(L, oldtop);
461     seterrorobj(L, status, oldtop);
462     L->ci = ci;
463     L->allowhook = ci->u.c.old_allowhook;
464     L->nny = 0; /* should be zero to be yieldable */
465     luaD_shrinkstack(L);
466     L->errfunc = ci->u.c.old_errfunc;
467     ci->callstatus |= CIST_STAT; /* call has error status */
468     ci->u.c.status = status; /* (here it is) */
469     return 1; /* continue running the coroutine */
470 }

```

recover 用来把错误引导到调用栈上最近的 lua_pcallk 的延续点上。它首先回溯 CallInfo 栈，找到从 C 中调用 lua_pcallk 的位置。这次 lua_pcallk 一定从 luaD_pcall 中被打断，接下来就必须完成 luaD_pcall 本应该完成却没有机会去做的事情。所以我们会看到，接下来的代码和 luaD_pcall 的后半部分非常相似。最后需要把线程运行状态设上 CIST_STAT 标记让 unroll 函数正确的设置线程状态。

接下来只需要保护调用 unroll 来依据 lua 调用栈执行逻辑上后续的流程。

最后回到源代码6.25，可以看到 lua.resume 比 Lua 5.1 之前的版本多了一个参数 from，它是用来更准确的统计 C 调用栈的层级的。nCcalls 的意义在于当发生无穷递归后，Lua 虚拟机可以先于 C 层面的堆栈溢出导致的毁灭性错误之前，捕获到这种情况，安全的抛出异常。由于现在可以在 C 函数中切出，那么发起 resume 的位置可能处于逻辑上调用层次较深的位置。这就需要调用者传入 resume 的调用来源线程，正确的计算 nCcalls。

6.2.6 lua_callk 和 lua_pcallk

有了这些基础，公开的 API lua_callk 和 lua_pcallk 就能理解清楚了。

lua_callk 只是对 luaD_call 的简单封装。在调用之前，根据需要把延续点 k 以及 ctx 设置到当前的 CallInfo 结构中。

```

889 LUA_API void lua_callk (lua_State *L, int nargs, int nresults, int ctx,
890                          lua_CFunction k) {
891     StkId func;
892     lua_lock(L);
893     api_check(L, k == NULL || !isLua(L->ci),
894              "cannot use continuations inside hooks");
895     api_checknelems(L, nargs+1);
896     api_check(L, L->status == LUA_OK, "cannot do calls on non-normal thread")
897     ;
898     checkresults(L, nargs, nresults);
899     func = L->top - (nargs+1);
900     if (k != NULL && L->nny == 0) { /* need to prepare continuation? */
901         L->ci->u.c.k = k; /* save continuation */
902         L->ci->u.c.ctx = ctx; /* save context */
903         luaD_call(L, func, nresults, 1); /* do the call */
904     }
905     else /* no continuation or no yieldable */
906         luaD_call(L, func, nresults, 0); /* just do the call */
907     adjustresults(L, nresults);
908     lua_unlock(L);
909 }

```

lua_pcallk 差不了多少。如果不需要延续点的支持或是处于不能被挂起的状态，那么，简单的调用 luaD_pcall 就可以了。否则不能设置保护点，而改在调用前设置好延续点以及 ctx，并将线程状态标记为 CIST_YPCALL。这样在 resume 过程中被 recover 函数找到。

源代码 6.31: lapi.c: lua_pcallk

```

928 struct CallS { /* data to 'f_call' */
929     StkId func;
930     int nresults;
931 };
932
933
934 static void f_call (lua_State *L, void *ud) {
935     struct CallS *c = cast(struct CallS *, ud);
936     luaD_call(L, c->func, c->nresults, 0);
937 }
938
939
940
941 LUA_API int lua_pcallk (lua_State *L, int nargs, int nresults, int errfunc,
942                        int ctx, lua_CFunction k) {

```

```

943 struct CallS c;
944 int status;
945 ptrdiff_t func;
946 lua_lock(L);
947 api_check(L, k == NULL || !isLua(L->ci),
948   "cannot use continuations inside hooks");
949 api_checknelems(L, nargs+1);
950 api_check(L, L->status == LUA_OK, "cannot do calls on non-normal thread")
951   ;
952 checkresults(L, nargs, nresults);
953 if (errfunc == 0)
954   func = 0;
955 else {
956   StkId o = index2addr(L, errfunc);
957   api_checkstackindex(L, errfunc, o);
958   func = savestack(L, o);
959 }
960 c.func = L->top - (nargs+1); /* function to be called */
961 if (k == NULL || L->nny > 0) { /* no continuation or no yieldable? */
962   c.nresults = nresults; /* do a 'conventional' protected call */
963   status = luaD_pcall(L, f_call, &c, savestack(L, c.func), func);
964 }
965 else { /* prepare continuation (call is already protected by 'resume')
966   */
967   CallInfo *ci = L->ci;
968   ci->u.c.k = k; /* save continuation */
969   ci->u.c.ctx = ctx; /* save context */
970   /* save information for error recovery */
971   ci->extra = savestack(L, c.func);
972   ci->u.c.old_allowhook = L->allowhook;
973   ci->u.c.old_errfunc = L->errfunc;
974   L->errfunc = func;
975   /* mark that function may do error recovery */
976   ci->callstatus |= CIST_YPCALL;
977   luaD_call(L, c.func, nresults, 1); /* do the call */
978   ci->callstatus &= ~CIST_YPCALL;
979   L->errfunc = ci->u.c.old_errfunc;
980   status = LUA_OK; /* if it is here, there were no errors */
981 }
982 adjustresults(L, nresults);
983 lua_unlock(L);

```

```

982     return status;
983 }

```

6.2.7 异常处理

lua 的内部运行期异常，即错误码为 `LUA_ERRRUN` 的那个，都是直接或间接的由 `luaG_errormsg` 抛出的。按 lua 的约定，这类异常会在数据栈上留下错误信息，或是调用一个用户定义的错误处理函数。

`luaG_errormsg` 实现在 `ldebug.c` 中。

源代码 6.32: `ldebug.c: luaG_errormsg`

```

560 l_noret luaG_errormsg (lua_State *L) {
561     if (L->errfunc != 0) { /* is there an error handling function? */
562         StkId errfunc = restorestack(L, L->errfunc);
563         if (!ttisfunction(errfunc)) luaD_throw(L, LUA_ERRERR);
564         setobjs2s(L, L->top, L->top - 1); /* move argument */
565         setobjs2s(L, L->top - 1, errfunc); /* push function */
566         L->top++;
567         luaD_call(L, L->top - 2, 1, 0); /* call it */
568     }
569     luaD_throw(L, LUA_ERRRUN);
570 }

```

它尝试从 `L` 中读出 `errfunc`，并使用 `luaD_call` 调用它。如果在 `errfunc` 里再次出错，会继续调用自己。这样就有可能会在错误处理函数中递归下去。但调用达到一定层次后，`nCcalls` 会超过上限最终产生一个 `LUA_ERRERR` 中止这个过程。

公开的 API `lua_error` 是对它的简单封装。

源代码 6.33: `lapi.c: lua_error`

```

1105 LUA_API int lua_error (lua_State *L) {
1106     lua_lock(L);
1107     api_checknelems(L, 1);
1108     luaG_errormsg(L);
1109     /* code unreachable; will unlock when control actually leaves the kernel
1110        */
1110     return 0; /* to avoid warnings */
1111 }

```

第七章 虚拟机

Lua 是一门解释型语言。但在运行时，并不直接解释运行 Lua 的源代码，而是先翻译成一种字节码，这些字节码运行在 Lua 的虚拟机上。

从 Lua 第 5 版开始，将之前的堆栈式虚拟机重新设计后，重新实现为一种新的基于寄存器的虚拟机¹。同时大量减少了虚拟机指令数量²。这使得实现 Lua 虚拟机的复杂度也大大降低。除了官方采用的 C 实现外，采用 Java 或 C# 等实现 Lua 虚拟机都不是特别困难的事情。

7.1 指令结构

Lua 中的每条指令都由一个 32 位无符号整数表示，指令种类和操作对象都被编码进这个数字。Lua 为它定义了一个专门的类型叫做 Instruction。

```
1 /*
2 ** type for virtual-machine instructions
3 ** must be an unsigned with (at least) 4 bytes (see details in lopcodes.h)
4 */
5 typedef lu_int32 Instruction;
```

表示指令种类的部分叫做 opcode，由于操作种类只有 40 种，所以仅需要 6 位编码，目前尚留有很大的扩展空间。

每条指令，大都是对一个对象作出影响。这个受影响的对象被称为 A。它由一个 8 位整数编码进指令中。A 通常是一个寄存器的索引值；也可能是对 upvalue 的操作。而作用到 A 的参数一般有两个，每个参数就由 9 位表示，分别称为 B 和 C。

下表可以看到指令按位域的布局。

有部分指令不需要两个操作参数，这时，可以把 B 和 C 合并为一个 18 位的整数 Bx 看待，适应更大的范围。

当操作符涉及指令跳转时，例如 OP_JMP，这个参数表示跳转偏移量。向前跳转需要设定偏移量为一个负数。这类指令需要参数带有符号信息，此时记作 sBx。Lua 采用和浮点数的指数表示方式相同的移码来表示有符号的数字。例如，0 被表示为 2^{17} ，而 -1 被表示为 $2^{17} - 1$ 。

¹大部分虚拟机都被实现为堆栈式的，例如传统的 Java 虚拟机、.Net 虚拟机、Python 虚拟机等等。从历史上看，Lua 5 是第一个被广泛使用的寄存器式虚拟机；之后诞生的 Perl 6 以及 Google 为 Andriod 实现的 Delvik 虚拟机也采用了基于寄存器的设计。相比于堆栈式设计，基于寄存器的虚拟机可以用更少的指令数量来表达要进行的操作，但由于需要讲寄存器号编码进指令，所以单条指令的长度也需要更长。不过 Lua 的虚拟机指令设计的非常巧妙，用等长的 32 位数字就可以表示绝大部分指令。

²Lua 虚拟机指令数量一度在 3.1 版达到了 128 条，到 5.0 发布时，重新设计过的 Lua 虚拟机指令数仅需要 35 条[4]。到 Lua 5.2.2 为止，虚拟机一共有 40 条指令。

A B C 在指令中所占位数以及在指令码中的位偏移量, Lua 都在 `lopcodes.h` 中以宏定义的形式给出, 可以由程序员配置³。

源代码 7.1: `lopcodes.h`: `opcodearg`

```
35  /*
36  ** size and position of opcode arguments.
37  */
38  #define SIZE_C      9
39  #define SIZE_B      9
40  #define SIZE_Bx     (SIZE_C + SIZE_B)
41  #define SIZE_A      8
42  #define SIZE_Ax     (SIZE_C + SIZE_B + SIZE_A)
43
44  #define SIZE_OP     6
45
46  #define POS_OP      0
47  #define POS_A       (POS_OP + SIZE_OP)
48  #define POS_C       (POS_A + SIZE_A)
49  #define POS_B       (POS_C + SIZE_C)
50  #define POS_Bx      POS_C
51  #define POS_Ax      POS_A
```

7.1.1 常量

在源代码5.1中, 我们可以看到, 每个函数原型都绑定有这个函数用到的所有常量。Lua 虚拟机在运行时, 会将需要的常量加载到寄存器中, 然后利用这些寄存器做相应的工作。加载常量的 `opcode` 为 `LOADK`, 它有两个参数。这个操作把 Bx 所指的常量加载到 A 所指的寄存器中。Lua 虚拟机同时支持 256 个寄存器⁴, 而这个操作最多可以索引 2^{18} 个常量。

在 Lua 5.1 中, 单个函数的常量个数受这个 `opcode` 的限制。由程序员手写的代码很少超过这个限制, 但某些机器生成的代码则不然。Lua 5.2 将这个限制放宽到了 2^{26} 。它增加了 `LOADKX` 指令, 把常量索引号

³比如, 你可以把 B 和 C 参数在指令码中的位置交换用于你的嵌入环境。这样, 标准的 Lua 字节码反编译程序就会得到错误的结果。当你不想让别人轻易看到你的软件中嵌入的 Lua 字节码的含义, 这可能是个简单的办法。当然, 对于读过 Lua 源码的程序员来说, 没有太大的意义。

⁴Lua 的寄存器指向当前栈空间。

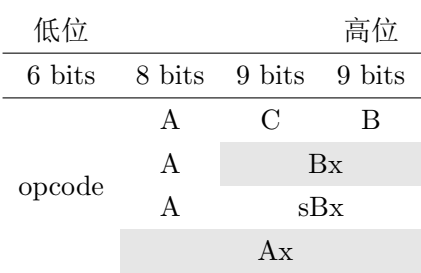


表 7.1: 指令布局

放在接下来的一条 EXTRAARG 指令中。EXTRAARG 把操作码以外的所有 26 位都用于参数表示，把它称为 Ax。我们可以看成这是 Lua 虚拟机对 32 位指令长度限制的一个扩展。

而当函数引用的常量较少时，Lua 则采用了另一种优化方案：不必将常量加载到寄存器中再做后续处理，而是让具体指令直接去访问常量表。某些指令用一种特殊的方式使用 B 或 C 参数。这个参数即可以是表示对寄存器号的索引，又可以表示对常量表的索引。只是，为了区分参数表示的是寄存器号，还是常量编号，需要占用参数中的一个位。去掉这一位后，B 和 C 还能表示 0 到 255 的索引，和 A 的长度吻合。下面看一下相关的宏⁵：

源代码 7.2: lopcodes.h: rk

```

129  /*
130  ** Macros to operate RK indices
131  */
132
133  /* this bit 1 means constant (0 means register) */
134  #define BITRK          (1 << (SIZE_B - 1))
135
136  /* test whether value is a constant */
137  #define ISK(x)          ((x) & BITRK)
138
139  /* gets the index of the constant */
140  #define INDEXK(r)       ((int)(r) & ~BITRK)
141
142  #define MAXINDEXRK      (BITRK - 1)
143
144  /* code a constant index as a RK value */
145  #define RKASK(x)         ((x) | BITRK)

```

7.1.2 操作码分类校验

按参数个数和种类不同，Lua 把 40 个 opcode 分为 4 个种类：iABC, iABx, iAsBx, iAx。在 lopcode.c 中，定义了一个叫做 luaP_opmodes 的数组，描述了每条操作码的类型，以及所涉及参数的具体用法。用法涉及参数是否有使用，或使用是用于寄存器还是常量索引等。

这些分类信息，用于 luac 反编译字节码时的输出，对于 Lua 运行期没有实际意义。如果定义了 lua_assert 这个宏的话，Lua 虚拟机还会根据其中的信息去校验字节码的相关代码的实现是否正确⁶。

这个检查应该是用于 Lua 的开发期。不对 Lua 虚拟机的实现做修改的话，这些检查是没有意义的。所以，所有这些校验代码，都是以宏的形式存在，默认处于关闭状态。

luaP_opmodes 中还有两个标记位。

第 6 位表示当前指令是否会修改寄存器 A，这个标记对调试模块有所帮助。它用于跟踪最后改变寄存器内容的指令位置，帮助生成调试信息。

⁵BITRK 这个宏使用了 SIZE_B，但这个机制可被同时应用于参数 B 和参数 C。所以，B 和 C 的长度需要一致。

⁶比如，把 B 作为寄存器索引时，利用了 RB() 这个宏。这个宏会间接检查当前指令的操作码对应的模式信息中，B 参数是否代表了一个寄存器索引。

第 7 位表示当前指令是否涉及一次条件跳转。之所以需要这个标记，是因为 Lua 中所有涉及条件分支的地方，实际上都在分支指令后紧随着一条 JMP 指令。Lua 没有为布尔运算单独设计操作码，它让所有的布尔运算都以分支执行流的形式出现⁷。这样做简化了指令集，但也使得单条分支指令无法承担全部操作的内容。分支指令和之后的跳转指令是一体的，是因为 32 位无法全部描述才分拆为两条指令。这个标记可以用来检测分支指令这一大类别。当遇到跳转指令时，可以回溯到前面一条指令来分辨是否是一次条件跳转。这对于 Lua 的字节码生成模块，是有必要的。

7.1.3 操作码列表

在 lopcodes.h 中，定义了全部 40 个操作码的枚举量。并以注释形式给出了每个操作码的含义。对于理解 Lua 虚拟机的操作码来说，这是一段非常重要的信息。这里把这些内容整理在下表。

名字	参数	描述
MOVE	A B	$R(A) := R(B)$
LOADK	A Bx	$R(A) := Kst(Bx)$
LOADKX	A	$R(A) := Kst(extra\ arg)$
LOADBOOL	A B C	$R(A) := (Bool)B; \text{ if } (C) \text{ pc}++$
LOADNIL	A B	$R(A), R(A+1), \dots, R(A+B) := nil$
GETUPVAL	A B	$R(A) := UpValue[B]$
GETTABUP	A B C	$R(A) := UpValue[B][RK(C)]$
GETTABLE	A B C	$R(A) := R(B)[RK(C)]$
SETTABUP	A B C	$UpValue[A][RK(B)] := RK(C)$
SETUPVAL	A B	$UpValue[B] := R(A)$
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	$R(A) := RK(B) / RK(C)$
MOD	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	$R(A) := \text{not } R(B)$
LEN	A B	$R(A) := \text{length of } R(B)$
CONCAT	A B C	$R(A) := R(B) \dots R(C)$
JMP	A sBx	$\text{pc} += sBx; \text{ if } (A) \text{ close all upvalues } \geq R(A) + 1$
EQ	A B C	$\text{if } ((RK(B) == RK(C)) = A) \text{ then pc}++$
LT	A B C	$\text{if } ((RK(B) < RK(C)) = A) \text{ then pc}++$
LE	A B C	$\text{if } ((RK(B) \leq RK(C)) = A) \text{ then pc}++$
TEST	A C	$\text{if not } (R(A) <=> C) \text{ then pc}++$

⁷Lua 的布尔运算 And 和 Or 支持短路法则，所以它在虚拟机中以分支跳转的形式实现。在 Lua 虚拟机的操作码中，看不到 And 操作和 Or 操作。

名字	参数	描述
TESTSET	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)
FORLOOP	A sBx	R(A)+=R(A+2); if R(A) <= R(A+1) ⁸ then { pc+=sBx; R(A+3)=R(A) }
FORPREP	A sBx	R(A)-=R(A+2); pc+=sBx
TFORCALL	A C	R(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
TFORLOOP	A sBx	if R(A+1) = nil then { R(A)=R(A+1); pc += sBx }
SETLIST	A B C	R(A)[(C-1)*FPF+i] := R(A+i), 1 ≤ i ≤ B
CLOSURE	A Bx	R(A) := closure(KPROTO[Bx])
VARARG	A B	R(A), R(A+1), ..., R(A+B-2) = vararg
EXTRAARG	Ax	extra (larger) argument for previous opcode

表 7.2: Opcode 列表

R() 表示这是一个寄存器索引；RK() 表示这可能是一个寄存器索引，也可能是一个常量索引，由参数的高位决定是哪种。

lopcodes.h 中的枚举量加有 OP_ 前缀。枚举量的次序可以调整，但需要小心。如果你调整了它们的次序，要找到源代码中所有提到 ORDER OP 的地方，把它们的次序调整为一致的⁹。

7.2 字节码的运行

在第六章，我们讨论了 C 函数在 Lua 中的运行过程。在这一章，我们来看看 Lua 函数是如何工作的。

如果仅看 Lua 虚拟机配备的操作码，实现这个虚拟机是非常容易的。Lua 支持的数据类很简单，所以虚拟机指令也只有简单的四则运算、字符串连接、生成闭包、分支比较、对表结构的读写、For 循环的处理、函数调用及返回，等等。

如果熟悉 Lua 语言，可以把 Lua 有限的语言特性一一对应到字节码的操作中。元表和协程这两个由 Lua 第 5 版引入的特性，在操作码中找不到对应物，它们以 C 库的形式提供融入到虚拟机的实现中。假设我们重新实现一个削减版的 Lua，去掉协程甚至元表这两个特性，也不考虑自动垃圾收集的话¹⁰，仅需要用一个简单的循环依次解析字节码的操作类型，做出相应的操作就够了。

7.2.1 luaV_execute

luaV_execute 是 Lua 虚拟机执行一段字节码的入口。如果把 Lua 虚拟机看成一个状态机¹¹，它就是从当前调用栈上次运行点开始解释字节码指令，直到下一个 C 边界跳出点。所谓 C 边界跳出点，可以是函数执行完毕，也可以是一次协程 yield 操作。

⁸当 R(A+2) 为负数时，这个比较关系应该颠倒过来。
⁹有的软件不想让别人分析内嵌的 Lua 字节码，会打乱这些操作码的次序，使得标准的 Lua 反编译工具难以解析出正确的结果。
¹⁰用其它语言实现 Lua 的需求，一般出现在希望把 Lua 嵌入到 Java，C# 这样的环境中。这些环境大多原生支持自动垃圾收集，不必特别去实现。
¹¹lua_State 这个结构名暗示了“状态机”这个含义。

在6.1.2一节，提到过 CallInfo 这个调用栈结构。第六章中，我们分析了 C 函数的相关结构，现在来看看关于 Lua 函数的部分。

```

1 struct { /* only for Lua functions */
2     StkId base; /* base for this function */
3     const Instruction *savedpc;
4 } l;
```

u.l.savedpc 保存有指向当前指令的指针，u.l.base 是当前函数的数据栈指针。

每次进入一层 Lua 函数，以及退出一层 Lua 函数，luaV_execute 并不对应的产生一次 C 层面的函数调用。也就是说，从 Lua 中调用一个 Lua 函数，并不会产生一次独立的 luaV_execute 调用。Lua 自己维护数据栈和调用栈，在解析字节码的时候，用 goto 来更新栈信息。我们可以看到，在 luaV_execute 的前半段，定义了 newframe 这个跳转标签，函数调用 OP_CALL OP_TAILCALL 以及函数返回 OP_RETURN 都会回到这里，更新栈帧继续运行。cl¹² 变量中放置调用栈中当前函数对象，k 是这个函数的指令序列，base 是当前数据栈底的位置。

源代码 7.3: lvm.c: luaV_execute

```

534 void luaV_execute (lua_State *L) {
535     CallInfo *ci = L->ci;
536     LClosure *cl;
537     TValue *k;
538     StkId base;
539     newframe: /* reentry point when frame changes (call/return) */
540     lua_assert(ci == L->ci);
541     cl = clLvalue(ci->func);
542     k = cl->p->k;
543     base = ci->u.l.base;
544     /* main loop of interpreter */
545     for (;;) {
546         Instruction i = *(ci->u.l.savedpc++);
547         StkId ra;
548         if ((L->hookmask & (LUA_MASKLINE | LUA_MASKCOUNT)) &&
549             (--L->hookcount == 0 || L->hookmask & LUA_MASKLINE)) {
550             Protect(traceexec(L));
551         }
552         /* WARNING: several calls may realloc the stack and invalidate 'ra' */
553         ra = RA(i);
554         lua_assert(base == ci->u.l.base);
555         lua_assert(base <= L->top && L->top < L->stack + L->stacksize);
```

luaV_execute 用一个死循环，依次解析字节码指令。当前指令 i 从 savedpc 中取出：

```

1 Instruction i = *(ci->u.l.savedpc++);
```

¹²cl 是 closure 的缩写。

所有的指令¹³都会操作寄存器 A。而对 Lua 虚拟机而言，寄存器即栈上的变量，所以可以将寄存器 A 所指变量预先取出放到局部变量 ra 中。

```
1  #define RA(i)    (base+GETARG_A(i))
2
3  ra = RA(i);
```

但某些操作在运行过程中，会改变数据栈的大小。而 ra 是一个指向数据栈的指针，而不是一个索引，这种情况下，一旦栈有可能发生变化，就需要重新取 ra 的值。

另外一个和数据栈指针有关的变量是 base，也会因为做某些操作和发生变化。这时就需要重新对 base 赋值。这个比 ra 重置要频繁的多，因为 ra 只在一次操作中有效，做下一个操作时，自然会重新读取。而 base 则一直需要使用，所以对 base 的重置被封装到了 Protect 宏里。把需要重置 base 的代码块包裹在这个宏中。

源代码 7.4: lvm.c: Protect

```
511 #define Protect(x)    { {x}; base = ci->u.l.base; }
```

luaV_execute 的内循环中，对于每种操作码编写了一小段代码。它用的 C 语言中的 switch/case 结构。为了结构表达清晰，Lua 5.2 以后，定义了一组宏来表示这种控制结构。

源代码 7.5: lvm.c: vmdispatch

```
530 #define vmdispatch(o)    switch(o)
531 #define vmcase(l,b)      case l: {b}    break;
532 #define vmcasenb(l,b)    case l: {b}          /* nb = no break */
```

7.2.2 寄存器赋值

Lua 虚拟机是基于寄存器结构实现的，也就是说，每段 Lua 代码被翻译为一组对 256 个寄存器的操作指令。这有点类似我们为 Lua 编写 C 扩展。C 函数通常是从 L 中取出参数逐个记录在 C 的局部变量中，然后利用 C 代码直接对这些值进行操作。可以把寄存器类比为 Lua 的寄存器，它们的确有相似之处，C 中的局部变量处于 C 堆栈上，而 Lua 的寄存器则处于 Lua 的数据栈中。

给局部变量赋值的过程，是由 MOVE LOADK LOADKX LOADBOOL LOADNIL GETUPVAL SETUPVAL 这组操作码完成的。

值的来源有三：

其一，其它寄存器，即局部变量。MOVE 可以完成这个工作。

源代码 7.6: lvm.c: OP_MOVE

```
557     vmcase(OP_MOVE,
558           setobjs2s(L, ra, RB(i));
559     )
```

其二，常量。

nil 和 bool 类型的数据比较短，可以通过指令直接加载，而无需通过常量表。

¹³这里所指所有指令不包括 OP_EXTRAARG。且 EXTRAARG 也不应该看作是一条独立的指令。因为它的含义由上一条指令决定，属于上一条指令的额外参数。之所以独立拥有一个操作码，是为了方便检查字节码的有效性。

源代码 7.7: lvm.c: OP_LOADNIL OP_LOADBOOL

```

570     vmcase(OP_LOADBOOL,
571           setbvalue(ra, GETARG_B(i));
572           if (GETARG_C(i)) ci->u.l.savedpc++; /* skip next instruction (if C
           ) */
573     )
574     vmcase(OP_LOADNIL,
575           int b = GETARG_B(i);
576           do {
577               setnilvalue(ra++);
578           } while (b--);
579     )

```

为了减少生成字节码的数量, Lua 让这两个操作中都承载了比字面上更多的含义。

LOADNIL 可以同时把多个变量初始化为 nil。设想一下, 在大量 Lua 代码中, 我们都会在函数内声明好几个局部变量。这些变量的默认值都为 nil。只要这些变量声明位置在一起, 那么在生成字节码时, 就可以用一条 LOADNIL 指令搞定它们。

LOADBOOL 则不仅仅用于简单的将一个布尔常量赋给寄存器。因为在 Lua 代码中, 用一个常量 true 或是 false 初始化局部变量是比较少的。更多的是针对逻辑表达式的处理。比如我们写下面这行代码时候,

```
1 local t = (a == b)
```

表面上看, `a == b` 是一个表达式。但是 Lua 虚拟机并没有针对 `==` 运算的对应字节代码。所有逻辑表达式都被翻译为 if then else 的分支结构了。上面的代码实际更接近这样:

```

1 local t
2 if a == b then
3     t = true
4 else
5     t = false
6 end

```

LOADBOOL 可以根据 C 参数来决定是否跳过下一条指令, 就可以避免为这种结构生成过多的分支跳转指令。只需要四条, 而不是五条指令来完成上述操作。

```

1 EQ          1 0 1
2 JMP         0 1      ; to 4
3 LOADBOOL    0 0 1
4 LOADBOOL    0 1 0

```

对于数字或字符串这样的常量, 不可能直接把值编码进指令中。所以 Lua 为每个函数原型保留有一张常量表, 引用这些常量时, 只需要给出在常量表中的索引。LOADK 就可以把 Bx 参数索引的常量加载到 A 寄存器中。如果常量表过大, 索引号超过了 Bx 可以表达的范围, 就使用 LOADKX。

源代码 7.8: lvm.c: OP_LOADK

```

560     vmcase(OP_LOADK,
561           TValue *rb = k + GETARG_Bx(i);
562           setobj2s(L, ra, rb);
563     )
564     vmcase(OP_LOADKX,
565           TValue *rb;
566           lua_assert(GET_OPCODE(*ci->u.l.savedpc) == OP_EXTRAARG);
567           rb = k + GETARG_Ax(*ci->u.l.savedpc++);
568           setobj2s(L, ra, rb);
569     )

```

第三，一些既不是常量，又不在寄存器中的数据。

在 Lua 5.2 中，这类数据仅指 upvalue 或存在于某张表中的值。在历史上，Lua 有过全局表的概念¹⁴。现在全局变量仅仅是对一个特殊的 upvalue `_ENV` 的引用，可以看成是一个语法糖而不是语言的基础特性了。不过现实是，大量的 Lua 代码会直接引用 `_ENV` 中的变量，为这种访问方式设计一个独立的操作码有利于字节码的紧凑，提高性能。

源代码 7.9: lvm.c: OP_GETUPVAL

```

580     vmcase(OP_GETUPVAL,
581           int b = GETARG_B(i);
582           setobj2s(L, ra, cl->upvals[b]->v);
583     )
584     vmcase(OP_GETTABUP,
585           int b = GETARG_B(i);
586           Protect(luaV_gettable(L, cl->upvals[b]->v, RKC(i), ra));
587     )
588     vmcase(OP_GETTABLE,
589           Protect(luaV_gettable(L, RB(i), RKC(i), ra));
590     )
591     vmcase(OP_SETTABUP,
592           int a = GETARG_A(i);
593           Protect(luaV_settable(L, cl->upvals[a]->v, RKB(i), RKC(i)));
594     )
595     vmcase(OP_SETUPVAL,
596           UpVal *uv = cl->upvals[GETARG_B(i)];
597           setobj(L, uv->v, ra);
598           luaC_barrier(L, uv, ra);
599     )
600     vmcase(OP_SETTABLE,
601           Protect(luaV_settable(L, ra, RKB(i), RKC(i)));

```

¹⁴全局表的概念在 Lua 5.1 之前都一直存在。从 Lua 5.0 开始变为环境表。所以在之前的虚拟机操作码中，有一组独立的操作码 `GETGLOBAL` 和 `SETGLOBAL`。

```

602     )
603     vmcase(OP_NEWTABLE,
604         int b = GETARG_B(i);
605         int c = GETARG_C(i);
606         Table *t = luaH_new(L);
607         sethvalue(L, ra, t);
608         if (b != 0 || c != 0)
609             luaH_resize(L, t, lua0_fb2int(b), lua0_fb2int(c));
610         checkGC(L, ra + 1);
611     )
612     vmcase(OP_SELF,
613         StkId rb = RB(i);
614         setobjs2s(L, ra+1, rb);
615         Protect(luaV_gettable(L, rb, RKC(i), ra));
616     )

```

GETUPVAL 和 SETUPVAL 可以读写当前的 upvalue。

GETTABUP 和 SETTABUP 则用来读写 upvalue 所指的表中的条目¹⁵。

GETTABLE 和 SETTABLE 是类似的，但操作的表不在 upvalue 中，而在寄存器里。它们在实现上的区别仅在于，前者 B 是对 upvalue 的索引，而后者则表示寄存器号。

从文档上看，SELF 在 Lua 语法中是一个语法糖。但 Lua 虚拟机的确为它做了优化。`a:f()` 和 `local a = a;a.f(a)` 看起来完整一致，但它们对应的字节码却有区别。

1	GETTABUP	0 0 -1	; _ENV "a"
2	SELF	0 0 -2	; "f"
3	CALL	0 2 1	

如果没有 SELF 操作，就需要用两条指令来替代它：

1	GETTABUP	0 0 -1	; _ENV "a"
2	GETTABLE	1 0 -2	; "f"
3	MOVE	2 0	
4	CALL	1 2 1	

7.2.3 表处理

luaV_gettable 是对表结构读操作的封装。在基础的表访问的基础上，增加了元方法的处理。

源代码 7.10: lvm.c: luaV_gettable

```

110 void luaV_gettable (lua_State *L, const TValue *t, TValue *key, StkId val)
111 {
112     int loop;
113     for (loop = 0; loop < MAXTAGLOOP; loop++) {

```

¹⁵GETTABUP 和 SETTABUP 可以看作是过去 GETGLOBAL 和 SETGLOBAL 指令的替代品，且用途更为广泛。

```

113     const TValue *tm;
114     if (ttistable(t)) { /* 't' is a table? */
115         Table *h = hvalue(t);
116         const TValue *res = luaH_get(h, key); /* do a primitive get */
117         if (!ttisnil(res) || /* result is not nil? */
118             (tm = fasttm(L, h->metatable, TM_INDEX)) == NULL) { /* or no TM? */
119             setobj2s(L, val, res);
120             return;
121         }
122         /* else will try the tag method */
123     }
124     else if (ttisnil(tm = luaT_gettmbyobj(L, t, TM_INDEX)))
125         luaG_typeerror(L, t, "index");
126     if (ttisfunction(tm)) {
127         callTM(L, tm, t, key, val, 1);
128         return;
129     }
130     t = tm; /* else repeat with 'tm' */
131 }
132 luaG_runerror(L, "loop in gettable");
133 }

```

元表的处理最多处理 MAXTAGLOOP¹⁶层，几乎没有程序会用到这么深的层次，那样会导致性能极其低下。设置这个上限，主要是为了避免元表的循环引用导致的死循环。在源代码4.14 中，我们知道，操作元方法比从外部的公开 API 操作元表要快的多。

当操作对象不可以按表的模式去索引时，这里利用 luaG.typeerror 抛出异常¹⁷，中断 luaV_execute 的执行。

如果元表中的 index 并不对应一张表，而是一个函数时，就会引发一次元方法调用。它是由 callTM 实现的。

源代码 7.11: lvm.c: callTM

```

93 static void callTM (lua_State *L, const TValue *f, const TValue *p1,
94                    const TValue *p2, TValue *p3, int hasres) {
95     ptrdiff_t result = savestack(L, p3);
96     setobj2s(L, L->top++, f); /* push function */
97     setobj2s(L, L->top++, p1); /* 1st argument */
98     setobj2s(L, L->top++, p2); /* 2nd argument */
99     if (!hasres) /* no result? 'p3' is third argument */
100         setobj2s(L, L->top++, p3); /* 3rd argument */

```

¹⁶MAXTAGLOOP 默认为 100。

¹⁷异常处理见第6.2.1节。

```

101  /* metamethod may yield only when called from Lua code */
102  luaD_call(L, L->top - (4 - hasres), hasres, isLua(L->ci));
103  if (hasres) { /* if has result, move it to its place */
104      p3 = restorestack(L, result);
105      setobjs2s(L, p3, --L->top);
106  }
107 }

```

所有元方法都有三个参数¹⁸。参数一一定是对象本身，而参数二则根据元方法的不同有所差异。对于表操作，第二个参数是 key 值；而二元运算操作则是第二个参数数。无论如何，前两个参数一定是输入值，不可以被修改。而第三个参数则可以是输入，也可以作为输出使用。callTM 的 hasres 参数表示是否需要输出。有输出时，元方法只有两个输入参数，第三个参数为输出。

luaV_settable 是对表结构写操作的封装。同样可能触发元方法。

源代码 7.12: lvm.c: luaV_settable

```

136 void luaV_settable (lua_State *L, const TValue *t, TValue *key, StkId val)
137 {
138     int loop;
139     for (loop = 0; loop < MAXTAGLOOP; loop++) {
140         const TValue *tm;
141         if (ttistable(t)) { /* 't' is a table? */
142             Table *h = hvalue(t);
143             TValue *oldval = cast(TValue *, luaH_get(h, key));
144             /* if previous value is not nil, there must be a previous entry
145              in the table; moreover, a metamethod has no relevance */
146             if (!ttisnil(oldval) ||
147                 /* previous value is nil; must check the metamethod */
148                 ((tm = fasttm(L, h->metatable, TM_NEWINDEX)) == NULL &&
149                  /* no metamethod; is there a previous entry in the table? */
150                  (oldval != luaO_nilobject) ||
151                  /* no previous entry; must create one. (The next test is
152                   always true; we only need the assignment.) */
153                  (oldval = luaH_newkey(L, h, key), 1)))) {
154                 /* no metamethod and (now) there is an entry with given key */
155                 setobj2t(L, oldval, val); /* assign new value to that entry */
156                 invalidateTMcache(h);
157                 luaC_barrierback(L, obj2gco(h), val);
158                 return;
159             }
160             /* else will try the metamethod */

```

¹⁸call 元方法是个例外，它不是在这里处理的。源代码6.19 的 luaD_precall 函数会尝试把对一个对象的函数调用行为，转换为对其 call 元方法对应函数的调用。


```

160     }
161     else /* not a table; check metamethod */
162         if (ttisnil(tm = luaT_gettmbyobj(L, t, TM_NEWINDEX)))
163             luaG_typeerror(L, t, "index");
164     /* there is a metamethod */
165     if (ttisfunction(tm)) {
166         callTM(L, tm, t, key, val, 0);
167         return;
168     }
169     t = tm; /* else repeat with 'tm' */
170 }
171 luaG_runerror(L, "loop_in_settable");
172 }

```

值得注意的是这么两行：

```

1 invalidateTmcache(h);
2 luaC_barrierback(L, obj2gco(h), val);

```

Lua 对元表的方法做了优化，invalidateTmcache 在4.4 节有介绍。

由于表内容的更改有可能导致 Lua 内其它对象的生命期变化，所以需要调用 luaC_barrierback 。这涉及垃圾收集模块的工作。

7.2.4 表达式运算

Lua 支持加减乘除四则运算，以及乘方和取模这些二元运算；同时还有取负、取反、对象取长度这几个一元运算操作；另外，针对字符串类型，有字符串连接操作。

源代码 7.13: lvm.c: expressions

```

617     vmcase(OP_ADD,
618         arith_op(luaI_numadd, TM_ADD);
619     )
620     vmcase(OP_SUB,
621         arith_op(luaI_numsub, TM_SUB);
622     )
623     vmcase(OP_MUL,
624         arith_op(luaI_nummul, TM_MUL);
625     )
626     vmcase(OP_DIV,
627         arith_op(luaI_numdiv, TM_DIV);
628     )
629     vmcase(OP_MOD,
630         arith_op(luaI_nummod, TM_MOD);
631     )

```

```

632     vmcase(OP_POW,
633         arith_op(luaI_numpow, TM_POW);
634     )
635     vmcase(OP_UNM,
636         TValue *rb = RB(i);
637         if (ttisnumber(rb)) {
638             lua_Number nb = nvalue(rb);
639             setnvalue(ra, luaI_numunm(L, nb));
640         }
641         else {
642             Protect(luaV_arith(L, ra, rb, rb, TM_UNM));
643         }
644     )
645     vmcase(OP_NOT,
646         TValue *rb = RB(i);
647         int res = l_isfalse(rb); /* next assignment may change this value
        */
648         setbvalue(ra, res);
649     )
650     vmcase(OP_LEN,
651         Protect(luaV_objlen(L, ra, RB(i)));
652     )
653     vmcase(OP_CONCAT,
654         int b = GETARG_B(i);
655         int c = GETARG_C(i);
656         StkId rb;
657         L->top = base + c + 1; /* mark the end of concat operands */
658         Protect(luaV_concat(L, c - b + 1));
659         ra = RA(i); /* 'luav_concat' may invoke TMs and move the stack */
660         rb = b + base;
661         setobjs2s(L, ra, rb);
662         checkGC(L, (ra >= rb ? ra + 1 : rb));
663         L->top = ci->top; /* restore top */
664     )

```

这些操作很类似，都是以一个或两个对象为操作对象，经过运算后，把结果置入寄存器 A 中。

对于数值类型之间的运算，Lua 做了一些优化，不会判断和触发元方法。这样可以提高处理效率。这部分是用宏来实现的。

源代码 7.14: lvm.c: arith_op

```

520 #define arith_op(op,tm) { \
521     TValue *rb = RKB(i); \

```

```

522     TValue *rc = RKC(i); \
523     if (ttisnumber(rb) && ttisnumber(rc)) { \
524         lua_Number nb = nvalue(rb), nc = nvalue(rc); \
525         setnvalue(ra, op(L, nb, nc)); \
526     } \
527     else { Protect(luaV_arith(L, ra, rb, rc, tm)); } }

```

而对于其它类型，则会调用 luaV_arith。

源代码 7.15: lvm.c: luaV_arith

```

360 void luaV_arith (lua_State *L, StkId ra, const TValue *rb,
361                 const TValue *rc, TMS op) {
362     TValue tempb, tempc;
363     const TValue *b, *c;
364     if ((b = luaV_tonumber(rb, &tempb)) != NULL &&
365         (c = luaV_tonumber(rc, &tempc)) != NULL) {
366         lua_Number res = luaO_arith(op - TM_ADD + LUA_OPADD, nvalue(b), nvalue(
367             c));
368         setnvalue(ra, res);
369     }
370     else if (!call_binTM(L, rb, rc, ra, op))
371         luaG_aritherror(L, rb, rc);
372 }

```

二元运算的元方法触发规则略微复杂，所以用 call_binTM 对 callTM 做了一些封装。

源代码 7.16: lvm.c: call_binTM

```

175 static int call_binTM (lua_State *L, const TValue *p1, const TValue *p2,
176                       StkId res, TMS event) {
177     const TValue *tm = luaT_gettmbyobj(L, p1, event); /* try first operand
178                                                         */
179     if (ttisnil(tm))
180         tm = luaT_gettmbyobj(L, p2, event); /* try second operand */
181     if (ttisnil(tm)) return 0;
182     callTM(L, tm, p1, p2, res, 1);
183     return 1;
184 }

```

先判断第一个对象是否有需要的元方法，如果找不到，则去第二个对象上面查找。倘若找不到元方法，又不是基本的数值类型，luaV_arith 会用 luaG_aritherror 抛出异常。

UNM 这个取负操作略微不同，它是一个一元运算。但处理方式没有太大区别，只需要把参数复制一遍即可。

LEN 操作用于取对象长度，根据 Lua 的定义：对于字符串取串的长度；对于表，再没有定义元方法时，取数组部分长度；其它情况调用元 len 方法。这里实现了 luaV_objlen。这个内部 API 还用于和 C 交互的 API lua_len 的实现。

源代码 7.17: lvm.c: call_binTM

```

335 void luaV_objlen (lua_State *L, StkId ra, const TValue *rb) {
336     const TValue *tm;
337     switch (ttypenv(rb)) {
338         case LUA_TTABLE: {
339             Table *h = hvalue(rb);
340             tm = fasttm(L, h->metatable, TM_LEN);
341             if (tm) break; /* metamethod? break switch to call it */
342             setnvalue(ra, cast_num(luaH_getn(h))); /* else primitive len */
343             return;
344         }
345         case LUA_TSTRING: {
346             setnvalue(ra, cast_num(tsvalue(rb)->len));
347             return;
348         }
349         default: { /* try metamethod */
350             tm = luaT_gettmbyobj(L, rb, TM_LEN);
351             if (ttisnil(tm)) /* no metamethod? */
352                 luaG_typeerror(L, rb, "get_length_of");
353             break;
354         }
355     }
356     callTM(L, tm, rb, rb, ra, 1);
357 }

```

和 LEN 类似的操作是字符串连接操作 CONCAT，但做的工作要复杂许多。CONCAT 操作从语义上来说，是把 R(B) 到 R(C) 之间的所有值，都以字符串方式连接起来，把结果放到 R(A) 中。这个连接过程是通过 luaV_concat 函数完成的。Lua 另有一个公开 API lua_concat，需要完成类似的工作。但是 lua_concat 的语义要简单一些，只需要把栈顶的若干变量连接起来，并将这些变量弹出，然后将结果压入堆栈。

两者相较，后者可以实现的高效一些。所以 CONCAT 遵循了后一种语义。所以这个操作和前面提到的其它运算不同，它可能会改变 R(B) 到 R(C) 之间寄存器的值，甚至 R(C) 之后寄存器的值。其实际的工作方式是，把通过临时修改栈顶地址为 C，然后连接 R(B) 到 R(C) 的值，其结果存于 R(B)。再将 R(B) 复制到 R(A)，最后将栈顶位置调整回去。至于 R(B) 到 R(C) 以及之后的寄存器，不能被后续指令读取。换句话说，R(B) 到 R(C) 寄存器必须工作在栈顶。这一点是由 Lua 的字节码编译部分保证的。

例如这样一段 Lua 代码：

```

1 local a = "1"
2 local b = "2"
3 local c = a .. b

```

就需要先把 a 和 b 复制到栈顶寄存器，然后再执行 CONCAT 操作。Lua 为它生成的字节码如下：

```

1  LOADK          0 -1      ; "1"
2  LOADK          1 -2      ; "2"
3  MOVE           2 0
4  MOVE           3 1
5  CONCAT         2 2 3

```

由于字符串连接操作，可能触发元方法，进而导致堆栈空间扩展。在 luaV_concat 调用完毕后，ra 可能不再指向原来的位置。故而需要重新获取 ra = RA(i)。对于这种需要额外申请内存的操作，需要用 checkGC 做步进回收工作。

在 CONCAT 操作的最后，重置了数据栈的栈顶。

我们应该这样理解数据栈的栈顶：Lua 字节码以寄存器的方式来理解数据栈空间，大多数情况下，用到多少寄存器是在生成字节码的编译期决定的。所以在函数原型结构里有 maxstacksize 这个信息，同时在运行时，会把这段空间的顶端记录在 CallInfo 的 top 中。虚拟机运行是随机访问这段栈空间的。但 Lua 虚拟机在运行时，也会以堆栈的方式利用这个数据栈，这里的 luaV_concat 就是这样。这种以栈形式利用数据堆栈都是临时行为，使用完毕后应该重置数据栈顶。

最后，来看看 luaV_concat 的实现：

源代码 7.18: lvm.c: luaV_concat

```

293 void luaV_concat (lua_State *L, int total) {
294     lua_assert(total >= 2);
295     do {
296         StkId top = L->top;
297         int n = 2; /* number of elements handled in this pass (at least 2) */
298         if (!(ttisstring(top-2) || ttisnumber(top-2)) || !tostring(L, top-1)) {
299             if (!call_binTM(L, top-2, top-1, top-2, TM_CONCAT))
300                 luaG_concaterror(L, top-2, top-1);
301         }
302         else if (tsvalue(top-1)->len == 0) /* second operand is empty? */
303             (void)tostring(L, top - 2); /* result is first operand */
304         else if (ttisstring(top-2) && tsvalue(top-2)->len == 0) {
305             setobjs2s(L, top - 2, top - 1); /* result is second op. */
306         }
307         else {
308             /* at least two non-empty string values; get as many as possible */
309             size_t t1 = tsvalue(top-1)->len;
310             char *buffer;
311             int i;
312             /* collect total length */
313             for (i = 1; i < total && tostring(L, top-i-1); i++) {
314                 size_t l = tsvalue(top-i-1)->len;
315                 if (l >= (MAX_SIZET/sizeof(char)) - t1)

```

```

316         luaG_runerror(L, "string_length_overflow");
317         t1 += 1;
318     }
319     buffer = luaZ_openspace(L, &G(L)->buff, t1);
320     t1 = 0;
321     n = i;
322     do { /* concat all strings */
323         size_t l = tsvalue(top-i)->len;
324         memcpy(buffer+t1, svalue(top-i), l * sizeof(char));
325         t1 += l;
326     } while (--i > 0);
327     setsvalue2s(L, top-n, luaS_newlstr(L, buffer, t1));
328 }
329 total -= n-1; /* got 'n' strings to create 1 new */
330 L->top -= n-1; /* popped 'n' strings and pushed one */
331 } while (total > 1); /* repeat until only 1 result left */
332 }

```

它每次至少处理两个元素。如果其中有数值类型，就地通过 `tostring` 转换为字符串。`tostring` 是一个宏，当寄存器内值类型为数字时，调用 `luaV_tostring` 将其转换为字符串。

源代码 7.19: `lvm.c: luaV_tostring`

```

47 int luaV_tostring (lua_State *L, StkId obj) {
48     if (!ttisnumber(obj))
49         return 0;
50     else {
51         char s[LUA_MAXNUMBER2STR];
52         lua_Number n = nvalue(obj);
53         int l = lua_number2str(s, n);
54         setsvalue2s(L, obj, luaS_newlstr(L, s, l));
55         return 1;
56     }
57 }

```

当这两个元素中有至少有一个即不是字符串，又不是数字的话，就会触发 `concat` 元方法。否则就继续向下检测，统计后续的字符串或数字的长度。多个这两种类型的值，可以一次连接在一起，连接过程不会产生函数调用。

统计尽可能多的连续可连接量，统计它们的总长度，然后利用 `luaZ_openspace` 在 `G(L)` 的 `buff` 域中分配一块临时的空间¹⁹，足够存放下结果串。最后，在这个空间上做字符串连接操作即可。重复这个过程，就可以把所有要求的寄存器内容以字符串形式连接在一起。

¹⁹在2.2.3节，介绍了这个临时空间。

7.2.5 分支和跳转

Lua 虚拟机定义了 EQ LT LE TEST TESTSET 五个分支操作。前面提到过，我们不能以单条 32 位的指令来看待条件分支指令。而应该把分支指令之后的跳转指令 JMP 看作是一体的。即，当条件成立时，继续运行；条件不成立时，跳转到指定位置。JMP 也可以单独做无条件跳转指令使用。跳转地址使用的是相对量，负数表示向前跳转，零表示下一条指令，依次类推。

无条件跳转和条件跳转分别用 dojump 及 donextjump 两个宏实现。

源代码 7.20: lvm.c: dojump

```
501 /* execute a jump instruction */
502 #define dojump(ci,i,e) \
503     { int a = GETARG_A(i); \
504       if (a > 0) luaF_close(L, ci->u.l.base + a - 1); \
505       ci->u.l.savedpc += GETARG_sBx(i) + e; }
```

donextjump 读出下一条指令，其必定是 JMP。这里并没有立刻递增 savedpc 的值。而是让随后的 dojump 对 savedpc 的偏移多加 1。由于 dojump 是用宏实现的，可以认为多传递一个参数 e 并不会影响效率。而合并对 savedpc 的修改执行效率要略微高一点。

dojump 除了偏移 savedpc 以跳转执行流以外，当 A 大于 0 时，还会调用 luaF_close 关闭 A 层次的 upvalue²⁰。在 Lua 5.1 以前，JMP 操作并无这个职责，它仅仅修改 savedpc。但 Lua 5.1 有另一个操作码 CLOSE。若 JMP 操作会跳出一个代码块时，就生成一条 CLOSE 操作的指令来调用 luaF_close。CLOSE 操作总是伴随着 JMP，Lua 5.2 对虚拟机指令集做了优化，去掉了 CLOSE，把这个操作合并到了 JMP 中。

JMP 指令就是简单的调用 dojump。

源代码 7.21: lvm.c: OP_JMP

```
665     vmcase(OP_JMP,
666           dojump(ci, i, 0);
667     )
```

EQ LT LE 的处理非常类似，它们都有对应的 C API，所以在 lvm.c 中都以 luaV_ API 的形式实现。在虚拟机执行函数中，就是对这些内部 API 的调用判断条件是否成立。当条件不成立时，则执行 donextjump。

源代码 7.22: lvm.c: OP_EQ

```
668     vmcase(OP_EQ,
669           TValue *rb = RKB(i);
670           TValue *rc = RKC(i);
671           Protect(
672             if (cast_int(equalobj(L, rb, rc)) != GETARG_A(i))
673               ci->u.l.savedpc++;
674           else
675             donextjump(ci);
```

²⁰upvalue 的关闭操作，参见 5.3.1 节。

```

676     )
677 )
678 vmcase(OP_LT,
679     Protect(
680         if (luaV_lessthan(L, RKB(i), RKC(i)) != GETARG_A(i))
681             ci->u.l.savedpc++;
682         else
683             donextjump(ci);
684     )
685 )
686 vmcase(OP_LE,
687     Protect(
688         if (luaV_lessequal(L, RKB(i), RKC(i)) != GETARG_A(i))
689             ci->u.l.savedpc++;
690         else
691             donextjump(ci);
692     )
693 )

```

对于 EQ 操作，类型不同则不同，类型相同则用 luaV_equalobj_ 判定。

源代码 7.23: lvm.h: equalobj

```

20 #define equalobj(L,o1,o2)  (ttisequal(o1, o2) && luaV_equalobj_(L, o1, o2))

```

源代码 7.24: lvm.c: luaV_equalobj_

```

257 /*
258 ** equality of Lua values. L == NULL means raw equality (no metamethods)
259 */
260 int luaV_equalobj_ (lua_State *L, const TValue *t1, const TValue *t2) {
261     const TValue *tm;
262     lua_assert(ttisequal(t1, t2));
263     switch (ttype(t1)) {
264         case LUA_TNIL: return 1;
265         case LUA_TNUMBER: return luai_numeq(nvalue(t1), nvalue(t2));
266         case LUA_TBOOLEAN: return bvalue(t1) == bvalue(t2); /* true must be 1
                !! */
267         case LUA_TLIGHTUSERDATA: return pvalue(t1) == pvalue(t2);
268         case LUA_TLCF: return fvalue(t1) == fvalue(t2);
269         case LUA_TSHRSTR: return eqshrstr(rawtsvalue(t1), rawtsvalue(t2));
270         case LUA_TLNGSTR: return luaS_eqlngstr(rawtsvalue(t1), rawtsvalue(t2));
271         case LUA_TUSERDATA: {
272             if (uvalue(t1) == uvalue(t2)) return 1;

```



```

273     else if (L == NULL) return 0;
274     tm = get_equalTM(L, uvalue(t1)->metatable, uvalue(t2)->metatable,
        TM_EQ);
275     break; /* will try TM */
276 }
277 case LUA_TTABLE: {
278     if (hvalue(t1) == hvalue(t2)) return 1;
279     else if (L == NULL) return 0;
280     tm = get_equalTM(L, hvalue(t1)->metatable, hvalue(t2)->metatable,
        TM_EQ);
281     break; /* will try TM */
282 }
283 default:
284     lua_assert(iscollectable(t1));
285     return gcvalue(t1) == gcvalue(t2);
286 }
287 if (tm == NULL) return 0; /* no TM? */
288 callTM(L, tm, t1, t2, L->top, 1); /* call TM */
289 return !l_isfalse(L->top);
290 }

```

这个函数严格按 Lua 手册中的定义实现，对每种类型分别比较。

从代码中可以看出，nil number boolean lightuserdata thread function 类型都是值比较²¹，可以在 O(1) 下完成。

字符串被分为短字符串和长字符串分别处理，短字符串直接比较指针，长字符串则可能触发完整的比较操作²²。

userdata 和 table 都有可能触发比较元方法。按 Lua 手册中的定义，EQ 操作的元方法触发原则是：被比较的两个对象比较有相同的元表，否则认为它们不相等。这个过程由 get_equalTM 函数保证。

源代码 7.25: lvm.c: get_equalTM

```

186 static const TValue *get_equalTM (lua_State *L, Table *mt1, Table *mt2,
187                                   TMS event) {
188     const TValue *tm1 = fasttm(L, mt1, event);
189     const TValue *tm2;
190     if (tm1 == NULL) return NULL; /* no metamethod */
191     if (mt1 == mt2) return tm1; /* same metatables => same metamethods */
192     tm2 = fasttm(L, mt2, event);
193     if (tm2 == NULL) return NULL; /* no metamethod */
194     if (luaV_rawequalobj(tm1, tm2)) /* same metamethods? */
195         return tm1;

```

²¹thread 和 function 未在 case 中出现，它们走的 default 的 gc 对象指针的比较流程。

²²字符串比较的细节参见 3.2.1 节。

```

196     return NULL;
197 }

```

luaV_equalobj_ 同时兼任了 rawequalobj 的职责，当不需要触发元方法时，将 L 参数传空即可。

源代码 7.26: lvm.h: luaV_rawequalobj

```

22 #define luaV_rawequalobj(o1,o2)          equalobj(NULL,o1,o2)

```

LT 操作由 luaV_lessthan 完成，LE 操作由 luaV_lessequal 完成。之所以没有大于操作，是因为大于可以由小于等于取反得到。

源代码 7.27: lvm.c: luaV_lessthan luaV_lessequal

```

231 int luaV_lessthan (lua_State *L, const TValue *l, const TValue *r) {
232     int res;
233     if (ttisnumber(l) && ttisnumber(r))
234         return luai_numlt(L, nvalue(l), nvalue(r));
235     else if (ttisstring(l) && ttisstring(r))
236         return l_strcmp(rawtsvalue(l), rawtsvalue(r)) < 0;
237     else if ((res = call_orderTM(L, l, r, TM_LT)) < 0)
238         luaG_ordererror(L, l, r);
239     return res;
240 }
241
242
243 int luaV_lessequal (lua_State *L, const TValue *l, const TValue *r) {
244     int res;
245     if (ttisnumber(l) && ttisnumber(r))
246         return luai_numle(L, nvalue(l), nvalue(r));
247     else if (ttisstring(l) && ttisstring(r))
248         return l_strcmp(rawtsvalue(l), rawtsvalue(r)) <= 0;
249     else if ((res = call_orderTM(L, l, r, TM_LE)) >= 0) /* first try 'le' */
250         return res;
251     else if ((res = call_orderTM(L, r, l, TM_LT)) < 0) /* else try 'lt' */
252         luaG_ordererror(L, l, r);
253     return !res;
254 }

```

Lua 对数字类型直接处理，所以无法通过修改数字类型的元表来改变其行为。userdata 和 table 至少有一个对象定义有需要的元方法。LE 元方法不是必须的，如果没有 LE，Lua 会尝试用 LT 元方法替代。

字符串比较也是直接编码在比较函数里的，无法通过修改字符串类型的元表来改变。字符串比较算法可能比你想象的要复杂。这是因为 Lua 语言定义的字符串比较并不是简单的 memcmp，而要考虑系统 locale 定义。它封装在 l_strcmp 函数中，通过调用 strcoll 来实现²³。

²³如果你的系统安装了 locale-pinyin 包，并把 locale 设置为 zh_CN@pinyin.utf8，lua 就能按拼音次序为中文做正确的排序了。

源代码 7.28: lvm.c: l_strcmp

```

209 static int l_strcmp (const TString *ls, const TString *rs) {
210     const char *l = getstr(ls);
211     size_t ll = ls->tsv.len;
212     const char *r = getstr(rs);
213     size_t lr = rs->tsv.len;
214     for (;;) {
215         int temp = strcoll(l, r);
216         if (temp != 0) return temp;
217         else { /* strings are equal up to a '\0' */
218             size_t len = strlen(l); /* index of first '\0' in both strings */
219             if (len == lr) /* r is finished? */
220                 return (len == ll) ? 0 : 1;
221             else if (len == ll) /* l is finished? */
222                 return -1; /* l is smaller than r (because r is not finished) */
223             /* both strings longer than 'len'; go on comparing (after the '\0')
                */
224             len++;
225             l += len; ll -= len; r += len; lr -= len;
226         }
227     }
228 }

```

由于 Lua 的字符串是 '\0' 安全的，字符串中也可能包含 '\0'。一次 strcoll 调用并不一定得到结果。每次用 strcoll 比较完，假若字符串相等，就要跳过 '\0' 继续比较，直到达到 TString 中记录的字符串长度为止。

7.2.6 函数调用

Lua 中的函数调用有两种，一种是标准的函数调用，它会需要生成新的一层调用栈，执行函数流程，然后弹出调用栈返回。另一种叫做尾调用，它是对标准函数调用的优化。尾调用不生成新的调用栈，而不复用当前的。在大多数函数式编程语言中，都需要对尾调用做特别优化。因为函数式语言特别依赖函数的层层调用，甚至用尾调用的方式来做循环。传统方式每次函数调用都需要生成新的栈帧，容易造成栈溢出。

普通的函数调用，是用 CALL 操作实现的。

源代码 7.29: lvm.c: OP_CALL

```

709 vmcase(OP_CALL,
710     int b = GETARG_B(i);
711     int nresults = GETARG_C(i) - 1;
712     if (b != 0) L->top = ra+b; /* else previous instruction set top */
713     if (luaD_precall(L, ra, nresults)) { /* C function? */
714         if (nresults >= 0) L->top = ci->top; /* adjust results */
715         base = ci->u.l.base;

```

```

716     }
717     else { /* Lua function */
718         ci = L->ci;
719         ci->callstatus |= CIST_REENTRY;
720         goto newframe; /* restart luaV_execute over new Lua function */
721     }
722 )

```

函数调用的流程细节在6.2.2节有分析，这里只是从 B C 中取出参数和返回值的个数，然后调用 luaD_precall。B 为 0 时，表示传入参数是不定数量的，那么实际参数就由栈顶到函数对象的位置 A 的距离决定。当 B 大于 0 时，参数个数为 B - 1，此时需要临时调整数据栈顶指针为 ra+b，以适应 luaD_precall 的要求。

当 C 为 0 时，返回值是变长的，数量不可预期。Lua 把这种调用成为 open call。这种情况只发生在链式调用，即把一个函数的返回值，作为另一个接收变长参数的 Lua 函数的参数的调用；以及尾调用，还有利用函数返回值去初始化一张表的情况。若 C 大于 0，则明确接收函数产生的返回值中的 C - 1 个。

如果函数是一个 C 函数，那么在 luaD_precall 完成后，函数已经调用完毕。如果不是 open call，就需要把数据栈顶指针复位（对应前面修改数据栈顶指针的行为）。否则，留待后续的处理²⁴。

对 luaD_precall 的调用无法用 Protect 宏包裹起来（需要取得返回值），base 值有可能被修改，故需要显式写一行 base 变量的重置。

Lua 函数的调用很简单，前面的 luaD_precall 已经生成好新的 CallInfo 结构。只需要给其调用状态 callstatus 设置 CIST_REENTRY 标记，表示这次 Lua 函数是由 Lua 函数本身驱动的，然后跳转到 luaV_execute 开头继续运行即可。CIST_REENTRY 运行标记可以确保被调用的 Lua 函数在返回时，不会跳出 luaV_execute 的内部循环。

尾调用指函数最后以调用另一个函数的形式结束。这样另一个函数的返回值就可以看作当前函数的返回值。Lua 的编译模块在生成这类代码的字节码时，会专门为这种情况生成 TAILCALL 的操作码。单独为尾调用优化，可以节省最后一步参数传递的开销，而且一旦发生尾调用，当前函数已经不再需要数据栈和调用栈，新的调用层次直接复用它们即可。

源代码 7.30: lvm.c: OP_TAILCALL

```

723     vmcase(OP_TAILCALL,
724         int b = GETARG_B(i);
725         if (b != 0) L->top = ra+b; /* else previous instruction set top */
726         lua_assert(GETARG_C(i) - 1 == LUA_MULTRET);
727         if (luaD_precall(L, ra, LUA_MULTRET)) /* C function? */
728             base = ci->u.l.base;
729         else {
730             /* tail call: put called frame (n) in place of caller one (o) */
731             CallInfo *nci = L->ci; /* called frame */
732             CallInfo *oci = nci->previous; /* caller frame */
733             StkId nfunc = nci->func; /* called function */
734             StkId ofunc = oci->func; /* caller function */

```

²⁴在 SETLIST 操作的结束点上，重置了栈顶指针。如果是链式调用或尾调用，那么是不关心栈顶指针的（反正调用前都需要改写一次）。

```

735     /* last stack slot filled by 'precall' */
736     StkId lim = nci->u.l.base + getproto(nfunc)->numparams;
737     int aux;
738     /* close all upvalues from previous call */
739     if (cl->p->sizep > 0) luaF_close(L, oci->u.l.base);
740     /* move new frame into old one */
741     for (aux = 0; nfunc + aux < lim; aux++)
742         setobjs2s(L, ofunc + aux, nfunc + aux);
743     oci->u.l.base = ofunc + (nci->u.l.base - nfunc); /* correct base
744     */
745     oci->top = L->top = ofunc + (L->top - nfunc); /* correct top */
746     oci->u.l.savedpc = nci->u.l.savedpc;
747     oci->callstatus |= CIST_TAIL; /* function was tail called */
748     ci = L->ci = oci; /* remove new frame */
749     lua_assert(L->top == oci->u.l.base + getproto(ofunc)->
750         maxstacksize);
751     goto newframe; /* restart luaV_execute over new Lua function */

```

尾调用必须是一次 open call，所以 C 必须为 0。对 luaD_precall 的调用，返回值参数个数也就写死为 LUA_MULTRET 了。

被调用函数是一个 C 函数时，处理和 CALL 操作没有什么不同。当它是一个 Lua 函数时，就需要复用当前栈帧。

首先，关闭当前栈帧上的 upvalue，原本这个步骤应该由 RETURN 来完成的。但因为发生尾调用时，当前栈帧上的变量已经结束了它们的生命期，并将被新的函数复用空间，所以 luaF_close 这个操作是需要提前做的。

然后，将 luaD_precall 为新一层函数调用生成的调用栈，以及在新一层数据栈上准备好的参数，都复制到当前栈帧上。尾调用的行为还应该在 callstatus 上特别标记为 CIST_TAIL，这样调试钩子才能正确识别。做好这个工作后，把 ci 设回当前栈帧，然后跳转到 luaV_execute 开头继续运行就好了。

返回操作 RETURN 只针对 Lua 函数，和 C 函数无关。Lua 函数返回需要调用 luaF_close 关闭 open upvalue²⁵。

当返回值数量不定时，保留栈顶的位置；如果数量明确，则按返回参数数量调整栈顶位置，以便让 luaD_poscall 可以正确调整返回值在当前栈帧上的位置。这时，检查 CIST_REENTRY 标记，如果没有这个标记，表示当次调用是从 C 里面产生。这样，直接让 luaD_execute 返回即可。否则，在明确参数个数的情况下重置数据栈顶²⁶，然后跳转到 luaD_execute 开头继续循环即可。

7.2.7 不定长参数

不定长参数这个概念，只是针对 Lua 函数。因为 Lua 提供给 C 扩展用的 API 中，没有针对不定长参数的函数。

²⁵ 参见 5.3.1 节

²⁶ 返回参数不定时，需要保留当前栈顶位置，以便让后续代码知道有几个返回值。

在生成 Lua 函数的字节码时，编译器可以获知这个函数是否需要处理不定数量的参数。在函数原型 Proto 的数据结构中²⁷，is_vararg 这个字段标识了这个函数是否需要不定数量的参数。

用 luaD_precall 发起对一个 Lua 函数的调用时，发起调用者可能准备了不定数量的参数。这种情况往往是由于前几条指令产生了不定数量的参数导致的。这可能是由一次 open call，所调用的函数返回了不定数量的参数；也可以是在 Lua 中用 ... 引用不定数量的参数，它会生成 VARARG 这个操作的字节码。

但 luaD_precall 不用理会传入参数的个数约定，它只需要统计出真实参数个数即可。但无论是哪种情况，参数都在数据栈顶一直排列到 ra 处。

luaD_precall 查看被调用的函数，一旦发现它是一个 Lua 函数，并需要处理不定个数的参数的话，它就要把固定的参数个数复制到新创建出的栈帧上，保留不定数量的部分在上一个栈帧的末尾。细节见源代码 6.19。如果被调用函数是一个 C 函数，或它不需要这些，就不用做这些处理。

综合这些，可以知道，当一个函数被执行，它若需要引用 ... 的话，这些数据的位置就在上一层数据栈的末尾，即，当前栈帧的底 base 就处在这组参数的末端。由于当前的 CallInfo 结构里还记录了当前正在运行的函数在数据栈上的位置，它指向所有传入参数的开头。

VARARG 指令可以把 ... 参数中的若干个复制到当前栈帧。

源代码 7.31: lvm.c: OP_VARARG

```

842     vmcase(OP_VARARG,
843         int b = GETARG_B(i) - 1;
844         int j;
845         int n = cast_int(base - ci->func) - cl->p->numparams - 1;
846         if (b < 0) { /* B == 0? */
847             b = n; /* get all var. arguments */
848             Protect(luaD_checkstack(L, n));
849             ra = RA(i); /* previous call may change the stack */
850             L->top = ra + n;
851         }
852         for (j = 0; j < b; j++) {
853             if (j < n) {
854                 setobjs2s(L, ra + j, base - n + j);
855             }
856             else {
857                 setnilvalue(ra + j);
858             }
859         }
860     )

```

n 就是计算出的不定数量的参数的具体个数。B 大于 0 时，将指定个数 B 数量的参数复制到 ra 后续的寄存器，不足补 nil。当 B 为 0 时，表示要全部复制，这种情况应该调整当前数据栈顶以示复制了多少个。这样，操作结束后，数据栈处于在顶部存放有不定参数列表的状态，其数量是从 ra 一直排列到数据栈顶。这组值，可以用于函数返回，或是函数调用，还可以是用来初始化一张新表。

²⁷ 参见源代码 5.1。

... 还可以用来初始化一张新构建的表。这就需要把不定数量的参数依次置入表的数组部分。无论是固定数量还是不定数量的值去构建一张表，都是用 SETLIST 这个操作。

源代码 7.32: lvm.c: OP_SETLIST

```

811     vmcase(OP_SETLIST,
812           int n = GETARG_B(i);
813           int c = GETARG_C(i);
814           int last;
815           Table *h;
816           if (n == 0) n = cast_int(L->top - ra) - 1;
817           if (c == 0) {
818               lua_assert(GET_OPCODE(*ci->u.l.savedpc) == OP_EXTRAARG);
819               c = GETARG_Ax(*ci->u.l.savedpc++);
820           }
821           luai_runtimecheck(L, ttistable(ra));
822           h = hvalue(ra);
823           last = ((c-1)*LFIELDS_PER_FLUSH) + n;
824           if (last > h->sizearray) /* needs more space? */
825               luaH_resizearray(L, h, last); /* pre-allocate it at once */
826           for (; n > 0; n--) {
827               TValue *val = ra+n;
828               luaH_setint(L, h, last--, val);
829               luaC_barrierback(L, obj2gco(h), val);
830           }
831           L->top = ci->top; /* correct top (in case of previous open call)
                        */
832     )

```

B 是一次需要复制的数据的个数，C 是偏移量。SETLIST 是为了某些机器生成的代码制造的海量数据准备的，如果 C（只有 9 位）超过范围的话，可以利用接下来的 EXTRAARG 来获得更大范围的 C。

对于大量数据的初始化，是一段段的调用 SETLIST 的，这样可以避免单次占用太大的数据栈空间。所以在计算偏移量时，对 C 做了一个倍率 LFIELDS_PER_FLUSH²⁸。

当 B 为 0 时，即复制数据栈顶到 ra 的所有数据，这和前面处理不定长参数的约定一致。

SETLIST 最后一定会重置数据栈顶，如果之前有对不定参数的处理，那么它会安全的把数据栈顶指针复位。

7.2.8 生成闭包

在 Lua 中，函数是一等公民。一切代码都是函数，准确说是闭包。当我们执行一段程序时，其实是调用一个函数。加载一个库，也是调用一个函数。加载一个 Lua 程序文件，里面即使定义了许多 Lua 函数，但它的整体依旧是单个函数。

²⁸LFIELDS_PER_FLUSH 默认为 50。

所以，每段完整的字节码都是一个 Lua 函数。而每个函数里可以附有很多个函数原型²⁹。函数原型没有放在常量表里，而是单独成表。这是因为，它们不可以被 Lua 代码直接使用。只有函数原型和 upvalue 绑定在一起时，成为了闭包，才是 Lua 虚拟机可以处理的对象。

函数原型在生成包含它们的函数的代码被加载时，生成在内存中。单个函数原型可以被多次绑定，生成多个闭包对象。这个过程是由 CLOSURE 操作完成的。

源代码 7.33: lvm.c: OP_CLOSURE

```

833     vmcase(OP_CLOSURE,
834           Proto *p = cl->p->p[GETARG_Bx(i)];
835           Closure *ncl = getcached(p, cl->upvals, base); /* cached closure
                        */
836           if (ncl == NULL) /* no match? */
837               pushclosure(L, p, cl->upvals, base, ra); /* create a new one */
838           else
839               setclLvalue(L, ra, ncl); /* push cached closure */
840           checkGC(L, ra + 1);
841       )

```

在 Lua 5.1 以前，无论你是否绑定 upvalue，绑定怎样的 upvalue，每次都会生成新的闭包。Lua 5.2 中，做了一点优化。它缓存了上次生成的闭包，如果可能，就重复利用。这对函数式编程特别有效，因为当你返回一个没有任何 upvalue 的纯函数，或是只绑定有全部变量的函数时，不会生成新的实例。

getcached 用来检查是否有缓存过的闭包，以及这次需要绑定的 upvalue 是否和缓存体一致。

源代码 7.34: lvm.c: getcached

```

374 /*
375 ** check whether cached closure in prototype 'p' may be reused, that is,
376 ** whether there is a cached closure with the same upvalues needed by
377 ** new closure to be created.
378 */
379 static Closure *getcached (Proto *p, UpVal **encup, StkId base) {
380     Closure *c = p->cache;
381     if (c != NULL) { /* is there a cached closure? */
382         int nup = p->sizeupvalues;
383         Upvaldesc *uv = p->upvalues;
384         int i;
385         for (i = 0; i < nup; i++) { /* check whether it has right upvalues */
386             TValue *v = uv[i].instack ? base + uv[i].idx : encup[uv[i].idx]->v;
387             if (c->l.upvals[i]->v != v)
388                 return NULL; /* wrong upvalue; cannot reuse closure */
389         }
390     }
391     return c; /* return cached closure (or NULL if no cached closure) */

```

²⁹ 参见 5.1 节

392 }

为了让 upvalue 可比较，函数原型中记录了 upvalue 的描述信息 Upvaldesc 结构。

源代码 7.35: lobject.h: Upvaldesc

```

446 typedef struct Upvaldesc {
447     TString *name; /* upvalue name (for debug information) */
448     lu_byte instack; /* whether it is in stack */
449     lu_byte idx; /* index of upvalue (in stack or in outer function's list)
450                  */
451 } Upvaldesc;

```

instack 描述了函数将引用的这个 upvalue 是否恰好处于定义这个函数的函数中。这时，upvalue 是这个外层函数的局部变量，它位于数据栈上。

idx 指的是 upvalue 的序号。对于关闭的 upvalue，已经无法从栈上取到，idx 指外层函数的 upvalue 表中的索引号；对于在数据栈上的 upvalue，序号即变量对应的寄存器号。

在 Lua 5.1 以前，闭包不被缓存，所以无须对 upvalue 做比较，也就不需要这些信息。在 Proto 结构中，仅保存有供调试信息用的 upvalue 名；Lua 5.2 则把名字信息转移到 Upvaldesc 结构内了。

比较引用的 upvalue 是否相同，按 instack 标记分开处理就好了。相同的 upvalue 地址也一定相同。全部 upvalue 都一致的话，缓存内的闭包就可以复用。此时，调用 setclLvalue 把它赋给 ra 即可。

否则，需要调用 pushclosure 生成一个新的闭包，并更新缓存。

源代码 7.36: lvm.c: pushclosure

```

395 /*
396 ** create a new Lua closure, push it in the stack, and initialize
397 ** its upvalues. Note that the call to 'luaC_barrierproto' must come
398 ** before the assignment to 'p->cache', as the function needs the
399 ** original value of that field.
400 */
401 static void pushclosure (lua_State *L, Proto *p, UpVal **encup, StkId base,
402                        StkId ra) {
403     int nup = p->sizeupvalues;
404     Upvaldesc *uv = p->upvalues;
405     int i;
406     Closure *ncl = luaF_newLclosure(L, nup);
407     ncl->l.p = p;
408     setclLvalue(L, ra, ncl); /* anchor new closure in stack */
409     for (i = 0; i < nup; i++) { /* fill in its upvalues */
410         if (uv[i].instack) /* upvalue refers to local variable? */
411             ncl->l.upvals[i] = luaF_findupval(L, base + uv[i].idx);
412         else /* get upvalue from enclosing function */
413             ncl->l.upvals[i] = encup[uv[i].idx];
414     }

```

```

415     luaC_barrierproto(L, p, ncl);
416     p->cache = ncl;  /* save it on cache for reuse */
417 }

```

绑定 upvalue 生成闭包的过程，我们在5.3.1节讨论过。pushclosure 的最后，在更新 cache 指针前，需要调用 luaC_barrierproto，这是因为更换缓存对象，可能引起旧的被缓存闭包的生命期变更。关于垃圾收集的细节不在这里展开。

7.2.9 For 循环

Lua 支持两种 for 循环。一种比较简单，就是简单的数字循环；另一种可以支持迭代器。

```

1  for v = e1, e2, e3 do block end

```

Lua 手册中这样就是这类循环的实现：

```

1  do
2      local var, limit, step = tonumber(e1), tonumber(e2), tonumber(e3)
3      if not (var and limit and step) then error() end
4      while (step > 0 and var <= limit) or (step <= 0 and var >= limit)
5          do
6              local v = var
7              block
8              var = var + step
9          end
10     end

```

固然可以用已有的操作去模拟这套实现，但不够高效。Lua 的虚拟机提供了两个单独的操作为其提供支持。

在循环开始的时候，var limit 和 step 应处于 ra ra+1 ra+2 处。用一条 FORPREP 检查它们是否有效，然后立刻跳转到循环代码块尾部的 FORLOOP 指令上，检查结束条件。FORLOOP 在条件成立时，就会循环这个过程。

源代码 7.37: lvm.c: FORPREP

```

778     vmcase(OP_FORPREP,
779         const TValue *init = ra;
780         const TValue *plimit = ra+1;
781         const TValue *pstep = ra+2;
782         if (!tonumber(init, ra))
783             luaG_runerror(L, LUA_QL("for") " _initial_value_must_be_a_number");
784         ;
785         else if (!tonumber(plimit, ra+1))
786             luaG_runerror(L, LUA_QL("for") " _limit_must_be_a_number");
787         else if (!tonumber(pstep, ra+2))
788             luaG_runerror(L, LUA_QL("for") " _step_must_be_a_number");

```

```

788     setnvalue(ra, luai_numsub(L, nvalue(ra), nvalue(pstep)));
789     ci->u.l.savedpc += GETARG_sBx(i);
790 )

```

由于 FORLOOP 每次都会递增 var 值，所以 FORPREP 预先把 var 减去 step。

FORLOOP 会根据 step 是正还是负，来区别对待结束条件。由于循环变量 var 很可能用于代码内部，所以生成的循环体代码不直接使用 var 对应的寄存器 ra，而是由 FORLOOP 每次复制一份 var 到 ra+3，循环体就可以安全的修改它了。

源代码 7.38: lvm.c: FORLOOP

```

767     vmcase(OP_FORLOOP,
768         lua_Number step = nvalue(ra+2);
769         lua_Number idx = luai_numadd(L, nvalue(ra), step); /* increment
            index */
770         lua_Number limit = nvalue(ra+1);
771         if (luai_numlt(L, 0, step) ? luai_numle(L, idx, limit)
772             : luai_numle(L, limit, idx)) {
773             ci->u.l.savedpc += GETARG_sBx(i); /* jump back */
774             setnvalue(ra, idx); /* update internal index... */
775             setnvalue(ra+3, idx); /* ...and external index */
776         }
777     )

```

迭代器型的 for 循环要复杂一些。

Lua 手册里是这样解释的：

```

1  for var_1, ..., var_n in explist do block end

```

等价于

```

1  do
2      local f, s, var = explist
3      while true do
4          local var_1, ..., var_n = f(s, var)
5          if var_1 == nil then break end
6          var = var_1
7          block
8      end
9  end

```

实际生成的字节码，迭代器调用在代码块的尾部。在循环体开头，用一条 JMP 指令，直接跳转到尾部。

local var_1, ..., var_n = f(s, var) 这个过程对应于 TFORCALL 这个操作。

而后判断循环是否结束，并在未结束时移动 var 参数，并跳转到代码块开头继续循环的过程由 TFORLOOP 承担。这两条指令从流程上总是挨在一起的，但由于 TFORCALL 会引发 luaD_call 有可能被打断，所以分解成两个操作来完成更简单。

源代码 7.39: lvm.c: TFORCALL TFORLOOP

```

791     vmcasenb(OP_TFORCALL,
792             StkId cb = ra + 3; /* call base */
793             setobjs2s(L, cb+2, ra+2);
794             setobjs2s(L, cb+1, ra+1);
795             setobjs2s(L, cb, ra);
796             L->top = cb + 3; /* func. + 2 args (state and index) */
797             Protect(luaD_call(L, cb, GETARG_C(i), 1));
798             L->top = ci->top;
799             i = *(ci->u.l.savedpc++); /* go to next instruction */
800             ra = RA(i);
801             lua_assert(GET_OPCODE(i) == OP_TFORLOOP);
802             goto l_tforloop;
803     )
804     vmcase(OP_TFORLOOP,
805         l_tforloop:
806         if (!ttisnil(ra + 1)) { /* continue loop? */
807             setobjs2s(L, ra, ra + 1); /* save control variable */
808             ci->u.l.savedpc += GETARG_sBx(i); /* jump back */
809         }
810     )

```

从源代码可以看出，TFORCALL 做了一点优化。它顺带会承担 TFORLOOP 的工作。检查结束条件，如果没有结束，就跳转。当循环次数很多，且不被中断时，虚拟机只解释了 TFORCALL 而没有处理 TFORCALL 和 TFORLOOP 两条指令。

7.2.10 协程的中断和延续

在6.2.5节，我们讨论过协程可能在 C 层面中断 luaV_execute 的运行。也就是说，luaV_execute 除了正常的在 RETURN 操作中返回外，处理许多操作时，都可能直接跳出。

为了让 C 中的 yield 跳出协程后，还可以回来继续执行虚拟机中的字节码。光是依靠 savedpc 记住当前的指令位置是不够的。我们还需要利用 luaV_finishOp 来补全被中断的操作未做完的事情。

好在每个操作的跳出点大多是唯一的，就是那次对 luaD_call 的调用，所以处理起来还算简单。下面，我们来看看 luaV_finishOp 的实现。

源代码 7.40: lvm.c: luaV_finishOp

```

420 /*
421 ** finish execution of an opcode interrupted by an yield
422 */
423 void luaV_finishOp (lua_State *L) {
424     CallInfo *ci = L->ci;
425     StkId base = ci->u.l.base;
426     Instruction inst = *(ci->u.l.savedpc - 1); /* interrupted instruction */

```

```

427 OpCode op = GET_OPCODE(inst);
428 switch (op) { /* finish its execution */
429     case OP_ADD: case OP_SUB: case OP_MUL: case OP_DIV:
430     case OP_MOD: case OP_POW: case OP_UNM: case OP_LEN:
431     case OP_GETTABUP: case OP_GETTABLE: case OP_SELF: {
432         setobjs2s(L, base + GETARG_A(inst), --L->top);
433         break;
434     }
435     case OP_LE: case OP_LT: case OP_EQ: {
436         int res = !l_isfalse(L->top - 1);
437         L->top--;
438         /* metamethod should not be called when operand is K */
439         lua_assert(!ISK(GETARG_B(inst)));
440         if (op == OP_LE && /* "<=" using "<" instead? */
441             ttisnil(luaT_gettmbyobj(L, base + GETARG_B(inst), TM_LE)))
442             res = !res; /* invert result */
443         lua_assert(GET_OPCODE(*ci->u.l.savedpc) == OP_JMP);
444         if (res != GETARG_A(inst)) /* condition failed? */
445             ci->u.l.savedpc++; /* skip jump instruction */
446         break;
447     }
448     case OP_CONCAT: {
449         StkId top = L->top - 1; /* top when 'call_binTM' was called */
450         int b = GETARG_B(inst); /* first element to concatenate */
451         int total = cast_int(top - 1 - (base + b)); /* yet to concatenate */
452         setobj2s(L, top - 2, top); /* put TM result in proper position */
453         if (total > 1) { /* are there elements to concat? */
454             L->top = top - 1; /* top is one after last element (at top-2) */
455             luaV_concat(L, total); /* concat them (may yield again) */
456         }
457         /* move final result to final position */
458         setobj2s(L, ci->u.l.base + GETARG_A(inst), L->top - 1);
459         L->top = ci->top; /* restore top */
460         break;
461     }
462     case OP_TFORCALL: {
463         lua_assert(GET_OPCODE(*ci->u.l.savedpc) == OP_TFORLOOP);
464         L->top = ci->top; /* correct top */
465         break;
466     }
467     case OP_CALL: {

```

```
468     if (GETARG_C(inst) - 1 >= 0) /* nresults >= 0? */
469         L->top = ci->top; /* adjust results */
470     break;
471 }
472 case OP_TAILCALL: case OP_SETTABUP: case OP_SETTABLE:
473     break;
474 default: lua_assert(0);
475 }
476 }
```

这里面最为复杂的只有 CONCAT 的处理。这是因为 CONCAT 有可能引发多次中断。这里就可以解释，为什么在 luaV_concat 的循环体内，每做完一个步骤，都需要修正数据栈顶指针。而不是循环结束后，一次性修改。这样，才可以在函数运行被中断后，可以正确延续。

第八章 内置库的实现

Lua 5.2 自带了几个库，实现了一般应用最基本的需求。这些库的实现仅仅使用了 Lua 官方手册中提到的 API，对 Lua 核心部分的代码几乎没有依赖，所以最易于阅读。阅读这些库的实现，也可以加深对 Lua API 的印象，方便我们自己扩展 Lua。

Lua 5.2 简化了 Lua 5.1 中模块组织方式，这也使得代码更为简短。
这一章，就从这里开始。

8.1 从 math 模块看 Lua 的模块注册机制

数学库是最简单的一个。它导入了若干数学函数，和两个常量 pi 与 huge。我们先看看它如何把一组 API 以及常量导入 Lua 的。

源代码 8.1: lmathlib.c: mathlib

```
233 static const luaL_Reg mathlib[] = {
234     {"abs",    math_abs},
235     {"acos",   math_acos},
236     {"asin",   math_asin},
237     {"atan2",  math_atan2},
238     {"atan",   math_atan},
239     {"ceil",   math_ceil},
240     {"cosh",   math_cosh},
241     {"cos",    math_cos},
242     {"deg",    math_deg},
243     {"exp",    math_exp},
244     {"floor",  math_floor},
```

我没有列完这段代码，因为后面是雷同的。Lua 使用一个结构 `luaL_Reg` 数组来描述需要注入的函数和名字。结构体前缀是 `luaL` 而不是 `lua`，是因为这并非 Lua 的核心 API 部分。利用 `luaL_newlib` 可以把这组函数注入一个 table。代码见下面：

源代码 8.2: lmathlib.c: luaopen_math

```
271 LUAMOD_API int luaopen_math (lua_State *L) {
272     luaL_newlib(L, mathlib);
273     lua_pushnumber(L, PI);
274     lua_setfield(L, -2, "pi");
```

```

275     lua_pushnumber(L, HUGE_VAL);
276     lua_setfield(L, -2, "huge");
277     return 1;
278 }

```

`luaL_newlib` 是定义在 `lauxlib.h` 里的一个宏，在源代码8.3中我们将看到它仅仅是创建了一个 `table`，然后把数组里的函数放进去而已。这个 API 在 Lua 的公开手册里已有明确定义的。

源代码 8.3: `lauxlib.h`: `luaL_newlib`

```

108 #define luaL_newlibtable(L,l) \
109     lua_createtable(L, 0, sizeof(l)/sizeof((l)[0]) - 1)
110
111 #define luaL_newlib(L,l)      (luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))

```

注入这些函数使用的是 Lua 5.2 新加的 API `luaL_setfuncs`，引入这个 API 是因为 Lua 5.2 取消了环境。那么，为了让 C 函数可以有附加一些额外的信息，就需要利用 `upvalue`¹

Lua 5.2 简化了 C 扩展模块的定义方式，不再要求模块创建全局表。对于 C 模块，以 `luaopen` 为前缀导出 API，通常是返回一张存有模块内函数的表。这可以精简设计，Lua 中 `require` 的行为仅仅只是用来加载一个预定义的模块，并阻止重复加载而已；而不用关心载入的模块内的函数放在哪里²。

`luaL_setfuncs` 在源代码8.4 里列出了实现，正如手册里所述，它把数组 `l` 中的所有函数注册入栈顶的 `table`，并给所有的函数绑上 `nup` 个 `upvalue`。

源代码 8.4: `lauxlib.c`: `luaL_setfuncs`

```

848 LUALIB_API void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup) {
849     luaL_checkversion(L);
850     luaL_checkstack(L, nup, "too many upvalues");
851     for (; l->name != NULL; l++) { /* fill the table with given functions */
852         int i;
853         for (i = 0; i < nup; i++) /* copy upvalues to the top */
854             lua_pushvalue(L, -nup);
855         lua_pushcclosure(L, l->func, nup); /* closure with those upvalues */
856         lua_setfield(L, -(nup + 2), l->name);
857     }
858     lua_pop(L, nup); /* remove upvalues */
859 }

```

¹给 C 函数绑上 `upvalue` 取代之前给 C 函数使用的环境表，是 Lua 作者推荐的做法[3]。不过要留意：Lua 5.2 引入了轻量 C 函数的概念，没有 `upvalue` 的 C 函数将是一个和 `lightuserdata` 一样轻量的值。不给不必要的 C 函数绑上 `upvalue` 可以使 Lua 程序得到一定的优化。为了把需求不同的 C 函数区别对待，可以通过多次调用 `luaL_setfuncs` 来实现。

²Lua 5.1 引入了模块机制，要求编写模块的人提供模块名。对于 C 模块，模块名通过 `luaL_openlib` 设置，Lua 模块则是通过 `module` 函数。Lua 将以这个模块名在全局表中创建同名的 `table` 以存放模块内的 API。这些设计相对繁杂，在 5.2 版中已被废弃，代码和文档都因此简洁了不少。

8.2 math 模块 API 的实现

math 模块内的各个数学函数的实现中规中矩，就是使用的 Lua 手册里给出的 API 来实现的。

Lua 的扩展方式是编写一个原型为 `int lua_CFunction (lua_State *L)` 的函数。L 对于使用者来说，不必关心其内部结构。实际上，公开 API 定义所在的 `lua.h` 中也没有 `lua_State` 的结构定义。对于一个用 C 编写的系统，模块化设计的重点在于接口的简洁和稳定。数据结构的细节和内存布局最好能藏在实现层面，Lua 的 API 设计在这方面做了一个很好的示范。这个函数通常不会也不建议被 C 程序的其它部分直接调用，所以一般藏在源文件内部，以 `static` 修饰之。

Lua 的 C 函数以堆栈的形式和 Lua 虚拟机交换数据，由一系列 API 从 L 中取出值，经过一番处理，压回 L 中的堆栈。具体的使用方式见 Lua 手册[5]。阅读这部分代码也能增进了解。

源代码 8.5: `luaconf.h: lmathop`

```

408  /*
409  @@ l_mathop allows the addition of an 'l' or 'f' to all math operations
410  */
411  #define l_mathop(x)                (x)

```

稍微值得注意的是，在这里还定义了一个宏 `l_mathop`。可以看出 Lua 在可定制性上的考虑。当你想把 Lua 的 Number 类型修改为 long double 时，便可以通过修改这个宏定义，改变操作 Number 的 C 函数。比如使用 `sinl`（或是使用 `sinf` 操作 float 类型）而不是 `sin`³。

我们再看另一小段代码：`math.log` 的实现：

源代码 8.6: `lmathlib.c: log`

```

118  static int math_log (lua_State *L) {
119      lua_Number x = luaL_checknumber(L, 1);
120      lua_Number res;
121      if (lua_isnoneornil(L, 2))
122          res = l_mathop(log)(x);
123      else {
124          lua_Number base = luaL_checknumber(L, 2);
125          if (base == (lua_Number)10.0) res = l_mathop(log10)(x);
126          else res = l_mathop(log)(x)/l_mathop(log)(base);
127      }
128      lua_pushnumber(L, res);
129      return 1;
130  }
131
132  #if defined(LUA_COMPAT_LOG10)
133  static int math_log10 (lua_State *L) {
134      lua_pushnumber(L, l_mathop(log10)(luaL_checknumber(L, 1)));
135      return 1;

```

³ C 语言的最新标准 C11 中增加了 `_Generic` 关键字以支持泛型表达式[1]，可以更好的解决这个问题。不过，Lua 的实现尽量避免使用 C 标准中的太多特性，以提高可移植性。

```
136 }
137 #endif
```

这里可以看出 Lua 对 API 的锤炼, 以及对宿主语言 C 语言的逐步脱离。早期的版本中, 是有 `math.log` 和 `math.log10` 两个 API 的。目前 `log10` 这个版本仅仅考虑兼容因素时才存在。这缘于 C 语言中也有 `log10` 的 API。但从语义上来看, 只需要一个 `log` 函数就够了⁴。早期的 Lua 函数看起来更像是对 C 函数的直接映射、而这些年 Lua 正向独立语言而演变, 在更高的层面设计 API 就不必再表达实现层面的差别了。

这一小节最后一段值得一读的是 `math.random` 的实现:

源代码 8.7: `lmathlib.c: random`

```
198 static int math_random (lua_State *L) {
199     /* the '%' avoids the (rare) case of r==1, and is needed also because on
200        some systems (SunOS!) 'rand()' may return a value larger than RAND_MAX
201        */
202     lua_Number r = (lua_Number)(rand()%RAND_MAX) / (lua_Number)RAND_MAX;
203     switch (lua_gettop(L)) { /* check number of arguments */
204     case 0: { /* no arguments */
205         lua_pushnumber(L, r); /* Number between 0 and 1 */
206         break;
207     case 1: { /* only upper limit */
208         lua_Number u = luaL_checknumber(L, 1);
209         luaL_argcheck(L, (lua_Number)1.0 <= u, 1, "interval is empty");
210         lua_pushnumber(L, l_mathop(floor)(r*u) + (lua_Number)(1.0)); /* [1,
211            u] */
212         break;
213     case 2: { /* lower and upper limits */
214         lua_Number l = luaL_checknumber(L, 1);
215         lua_Number u = luaL_checknumber(L, 2);
216         luaL_argcheck(L, l <= u, 2, "interval is empty");
217         lua_pushnumber(L, l_mathop(floor)(r*(u-l+1)) + l); /* [l, u] */
218         break;
219     }
220     default: return luaL_error(L, "wrong number of arguments");
221     }
222     return 1;
223 }
```

⁴因为人类更习惯用十进制计数, 而计算机则在内部运算采用的是二进制。由于浮点数表示误差的缘故, $\log(x)/\log(10)$ 往往并不严格等于 $\log_{10}(x)$, 而是有少许误差。在我们常用的 X86 浮点指令集中, CPU 硬件支持以 10 为底的对数计算, 在 C 语言中也有独立的函数通过不同的机器指令来实现。

用参数个数来区分功能上的微小差异是典型的 Lua 风格，这是 Lua 接口设计上的一个惯例。另外，编码中考虑平台差异很考量程序员对各平台细节的了解，例如这里注释中提到的 SunOS 的 rand() 的小问题。

8.3 string 模块

Lua 的 string 库相较其它许多动态语言的 string 库来说，可谓短小精悍。不到千行 C 代码就实现了一个简单使用的字符串模式匹配模块。虽然功能上比正则表达式有所欠缺，但考虑到代码体积和功能比，这应该是一个相当漂亮的平衡⁵。若需要更强大的字符串处理功能，Lua 的作者之一 Roberto 给出了一个比正则表达式更强大的选择 LPEG⁶。有这一轻一重两大利器，在 Lua 社区中，很少有人再用正则表达式了。

string 模块实现在 lstrlib.c 中。

我们先看看 string 模块的注册部分，它和上节所讲的 math 模块稍有不同，准确说是略微复杂一点。

源代码 8.8: lstrlib.c: open

```

1014 LUAMOD_API int luaopen_string (lua_State *L) {
1015     luaL_newlib(L, strlib);
1016     createmetatable(L);
1017     return 1;
1018 }
```

这里多了一处 createmetatable 的调用，下面列出细节：

源代码 8.9: lstrlib.c: createmetatable

```

999 static void createmetatable (lua_State *L) {
1000     lua_createtable(L, 0, 1); /* table to be metatable for strings */
1001     lua_pushliteral(L, ""); /* dummy string */
1002     lua_pushvalue(L, -2); /* copy table */
1003     lua_setmetatable(L, -2); /* set table as metatable for strings */
1004     lua_pop(L, 1); /* pop dummy string */
1005     lua_pushvalue(L, -2); /* get string library */
1006     lua_setfield(L, -2, "__index"); /* metatable.__index = string */
1007     lua_pop(L, 1); /* pop metatable */
1008 }
```

第一次看 Lua 的 API 使用流程，容易被 Lua Stack 弄晕。Lua 和 C 的交互就是通过一系列的 API 操作 Lua 栈来完成的。如果你看不太明白，可以用笔在纸上画出 Lua 栈上数据的情况。进入这个函数时，栈顶有一个 table、即所有的 string API 存在的那张表。然后，余下的几行 API 创建了一个 metatable，使用这张表做索引表。这张元表最终被设置入 dummy 字符串中。

给字符串类型设置元表是 Lua 5.1 引入的特性，它并非给特定的字符串值赋予元方法，而是针对整个字符串类型。这个特性几乎不会给 Lua 的运行效率带来损失，但极大的丰富了 Lua 的表达能力。当然，处于语言上的严谨考虑，Lua 的 API setmetatable 禁止修改这个元表。我们将在 ?? 节中看到相关代码。

下面返回源文件开头，uchar 这个宏定义很能体现 Lua 源码风格：

⁵C 语言社区中常用的正则表达式库 PCRE 的个头比整个 Lua 5.2 的实现还要大好几倍。

⁶LPEG 全称 Parsing Expression Grammars For Lua，可以在这里下载到源代码：<http://www.inf.puc-rio.br/~roberto/lpeg/>

源代码 8.10: lstrlib.c: uchar

```

32  /* macro to 'unsign' a character */
33  #define uchar(c)          ((unsigned char)(c))

```

C 语言中，对 char 类型中保存数字是否有符号并无严格定义⁷，所以有统一把字符数字转换为无符号来处理的需求。Lua 的源码中，所有类型强制转换都很严谨的使用了宏以显式标示出来。因为仅在这一个源文件中需要处理字符这个类型，所以 uchar 这个宏被定义在 .c 文件中，而不存在于别的 .h 里。

Lua 在处理偏移量时，习惯用负数表示从尾部倒数。Lua API 设计中，对 Lua 栈的索引就是如此；处理字符串时，字符串索引也遵循同一规则。这里定义了一个内部函数 posrelat 方便转换这个索引值。

源代码 8.11: lstrlib.c: posrelat

```

46  static size_t posrelat (ptrdiff_t pos, size_t len) {
47      if (pos >= 0) return (size_t)pos;
48      else if (0u - (size_t)pos > len) return 0;
49      else return len - ((size_t)-pos) + 1;
50  }

```

lstrlib.c 中间的数百行代码大体分为三个部分。

第一部分，是一些简单的 API 实现，如 string.len、string.reverse、string.lower、string.upper 等等，实现的中规中矩，乏善可陈。str_byte 函数的实现中，有一行 luaL_checkstack 调用值得初学 Lua 的 C bindings 编写人员注意。Lua 的栈不像 C 语言的栈那样，不大考虑栈溢出的情况。Lua 栈给 C 函数留的默认空间很小，默认情况下只有 20⁸。当你要在 Lua 的栈上留下大量值时，务必用 luaL_checkstack 扩展堆栈。因为处于性能考虑，Lua 和栈有关的 API 都是不检查栈溢出的情况的。

源代码 8.12: lstrlib.c: byte

```

141  if (posi > pose) return 0; /* empty interval; return no values */
142  n = (int)(pose - posi + 1);
143  if (posi + n <= pose) /* (size_t -> int) overflow? */
144      return luaL_error(L, "string slice too long");
145  luaL_checkstack(L, n, "string slice too long");

```

暂时不继续写这一章了。

8.4 暂且搁置

⁷C 语言的 char 类型是 signed 还是 unsigned 依赖于实现[2]。我们常用的 C 编译器如 gcc 的 char 类型是有符号的，也有一些 C 编译器如 Watcom C 的 char 类型默认则是无符号的。

⁸LUA_MINSTACK 定义在 lua.h 中，默认值为 20。

参考文献

- [1] Iso/iec 9899:201x. In *Programming languages - C*, chapter 6.5.11 Generic selection.
- [2] Iso/iec 9899:201x. In *Programming languages - C*, chapter 6.2.5 Types.
- [3] Roberto Ierusalimschy. The novelties of lua 5.2. 2011. <http://www.inf.puc-rio.br/~roberto/talks/novelties-5.2.pdf>.
- [4] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. The evolution of lua. *Proceedings of ACM HOPL III (2007) 2-1-2-26*. <http://www.lua.org/doc/hopl.pdf>.
- [5] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. *Lua 5.2 Reference Manual*, 2011. <http://www.lua.org/manual/5.2/manual.html>.
- [6] Lua Wiki. Hash dos. <http://lua-users.org/wiki/HashDos>.
- [7] Wikipedia. Blocks (c language extension). [http://en.wikipedia.org/wiki/Blocks_\(C_language_extension\)](http://en.wikipedia.org/wiki/Blocks_(C_language_extension)).
- [8] Wikipedia. Indent style. http://en.wikipedia.org/wiki/Indent_style.
- [9] Wikipedia. Nan. <http://en.wikipedia.org/wiki/NaN>.
- [10] Wikipedia. Separate chaining. http://en.wikipedia.org/wiki/Hash_table#Separate_chaining.

源代码目录

2.1	lauxlib.c: luaL_newstate	7
2.2	lmem.h: memory	8
2.3	lmem.c: realloc	9
2.4	lmem.c: growvector	10
2.5	lstate.c: LG	11
2.6	lstate.c: lua_newstate	11
2.7	lapi.c: lua_version	13
2.8	lauxlib.c: luaL_checkversion	13
2.9	lstate.c: f_luaopen	14
3.1	lobject.h: variant string	17
3.2	lobject.h: tstring	17
3.3	lobject.h: getstr	18
3.4	lstate.h: stringtable	18
3.5	lstring.c: stringhash	19
3.6	lstate.c: makeseed	19
3.7	lstring.c: eqstr	20
3.8	lstring.c: eqlngstr	20
3.9	lstring.h: eqshrstr	20
3.10	lstring.c: internshrstr	20
3.11	lstring.c: resize	21
3.12	lstring.c: createtobj	22
3.13	lobject.h: udata	22
3.14	lstring.c: newudata	23
4.1	lobject.h: table	25
4.2	ltable.c: dummynode	26
4.3	ltable.c: new	27
4.4	ltable.c: newkey	27
4.5	ltable.c: rehash	29
4.6	ltable.c: mainposition	30
4.7	ltable.c: get	30
4.8	ltable.c: hashnum	32
4.9	llimits.h: hashnum1	32
4.10	llimits.h: hashnum2	33

4.11	ltable.c: next	33
4.12	ltable.c: findindex	34
4.13	ltable.c: getn	35
4.14	ltm.h: fasttm	36
4.15	ltm.c: init	36
4.16	ltm.c: gettm	37
4.17	ltm.c: typename	37
5.1	lobject.h: Proto	40
5.2	lfunc.c: luaF_freeproto	41
5.3	lobject.h: UpVal	42
5.4	lobject.h: LClosure	43
5.5	lfunc.c: luaF_newLclosure	43
5.6	lfunc.c: luaF_findupval	43
5.7	lfunc.c: luaF_close	44
5.8	ldo.c: f_parser	45
5.9	lfunc.c: luaF_newupval	46
5.10	lobject.h: CClosure	46
5.11	lfunc.c: luaF_newCclosure	46
5.12	lapi.c: lua_pushcclosure	47
5.13	lobject.h: lua_TFUNCTION	47
6.1	lstate.h: lua_State	50
6.2	lobject.h: Value	50
6.3	lstate.c: stack	51
6.4	ldo.c: growstack	52
6.5	ldo.c: correctstack	53
6.6	lstate.h: CallInfo	54
6.7	lstate.c: luaE_extendCI	56
6.8	ldo.c: next_ci	56
6.9	lstate.c: luaE_freeCI	56
6.10	lstate.c: lua_newthread	57
6.11	lstate.c: LX	57
6.12	ldo.c: LUA_THROW	58
6.13	ldo.c: lua_longjmp	59
6.14	ldo.c: luaD_rawrunprotected	59
6.15	ldo.c: luaD_throw	59
6.16	ldo.c: luaD_pcall	60
6.17	ldo.h: restorestack	61
6.18	ldo.c: luaD_precall	61
6.19	ldo.c: adjust_varargs	63
6.20	ldo.c: tryfuncTM	64
6.21	ldo.c: luaD_poscall	64

6.22 ldo.c: luaD_call	65
6.23 ldo.c: luaD_hook	66
6.24 ldo.c: lua_yieldk	68
6.25 ldo.c: lua_resume	69
6.26 ldo.c: resume	70
6.27 ldo.c: unroll	71
6.28 ldo.c: finishCcall	71
6.29 ldo.c: recover	72
6.30 lapi.c: lua_callk	73
6.31 lapi.c: lua_pcallk	74
6.32 ldebug.c: luaG_errormsg	76
6.33 lapi.c: lua_error	76
7.1 lopcodes.h: opcodearg	78
7.2 lopcodes.h: rk	79
7.3 lvm.c: luaV_execute	82
7.4 lvm.c: Protect	83
7.5 lvm.c: vmdispatch	83
7.6 lvm.c: OP_MOVE	83
7.7 lvm.c: OP_LOADNIL OP_LOADBOOL	84
7.8 lvm.c: OP_LOADK	84
7.9 lvm.c: OP_GETUPVAL	85
7.10 lvm.c: luaV_gettable	86
7.11 lvm.c: callTM	87
7.12 lvm.c: luaV_settable	88
7.13 lvm.c: expressions	89
7.14 lvm.c: arith_op	90
7.15 lvm.c: luaV_arith	91
7.16 lvm.c: call_binTM	91
7.17 lvm.c: call_binTM	92
7.18 lvm.c: luaV_concat	93
7.19 lvm.c: luaV_tostring	94
7.20 lvm.c: dojump	95
7.21 lvm.c: OP_JMP	95
7.22 lvm.c: OP_EQ	95
7.23 lvm.h: equalobj	96
7.24 lvm.c: luaV_equalobj_	96
7.25 lvm.c: get_equalTM	97
7.26 lvm.h: luaV_rawequalobj	98
7.27 lvm.c: luaV_lessthan luaV_lessequal	98
7.28 lvm.c: lstrcmp	99
7.29 lvm.c: OP_CALL	99

7.30	lvm.c: OP_TAILCALL	100
7.31	lvm.c: OP_VARARG	102
7.32	lvm.c: OP_SETLIST	103
7.33	lvm.c: OP_CLOSURE	104
7.34	lvm.c: getcached	104
7.35	lobject.h: Upvaldesc	105
7.36	lvm.c: pushclosure	105
7.37	lvm.c: FORPREP	106
7.38	lvm.c: FORLOOP	107
7.39	lvm.c: TFORCALL TFORLOOP	108
7.40	lvm.c: luaV_finishOp	108
8.1	lmathlib.c: mathlib	111
8.2	lmathlib.c: luaopen_math	111
8.3	lauxlib.h: luaL_newlib	112
8.4	lauxlib.c: luaL_setfuncs	112
8.5	luaconf.h: lmathop	113
8.6	lmathlib.c: log	113
8.7	lmathlib.c: random	114
8.8	lstrlib.c: open	115
8.9	lstrlib.c: createmetatable	115
8.10	lstrlib.c: uchar	116
8.11	lstrlib.c: posrelat	116
8.12	lstrlib.c: byte	116