Lua 源码欣赏

云风

2012 年 11 月 8 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

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] Lua 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 5.2.1 版可以在 `http://www.lua.org/ftp/lua-5.2.1.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`

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

   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。

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，也搀杂了一些 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 后跟一个大写字母表识内部模块名，而后由下划线加若干小写字母描述方法名。

[5] Clean 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。

[6] K&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[7]。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。

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 luaD [8] 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 、luaH [9]、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，核心系统还用到两个基础设施：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

[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 。

[8]D 取 Do 之意。

[9]H 是意取 Hash 之意。

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 （luaY [10] 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。

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，生成为字节码。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。

[10] Y 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。

[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。

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渡设计了[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 linit.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

[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 5.2.1 版本的源代码分布在 58 个文件中，共 20128 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

的了解。

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]

在本书接下来的章节，将大致按以上次序来分析 Lua 5.2 的实现。

[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 5.2.1 的实现。

第二章　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。

2.1　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 2.1: lmathlib.c: mathlib

```
237 static const luaL_Reg mathlib[] = {
238   {"abs",   math_abs},
239   {"acos",  math_acos},
240   {"asin",  math_asin},
241   {"atan2", math_atan2},
242   {"atan",  math_atan},
243   {"ceil",  math_ceil},
244   {"cosh",   math_cosh},
245   {"cos",   math_cos},
246   {"deg",   math_deg},
247   {"exp",   math_exp},
248   {"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 2.2: lmathlib.c: luaopen_math

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

luaL_newlib 是定义在 lauxlib.h 里的一个宏，在源代码2.3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 2.3: 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

[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。

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2.4 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 2.4: lauxlib.c: luaL_setfuncs

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

2.2 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

[2]Lua 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。

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 2.5: lmathlib.c: l_tg

```
26 #if !defined(l_tg)
27 #define l_tg(x)         (x)
28 #endif
29
30
31
32 static int math_abs (lua_State *L) {
33   lua_pushnumber(L, l_tg(fabs)(luaL_checknumber(L, 1))
       );
34   return 1;
35 }
36
37 static int math_sin (lua_State *L) {
38   lua_pushnumber(L, l_tg(sin)(luaL_checknumber(L, 1)))
       ;
39   return 1;
40 }
41
42 static int math_sinh (lua_State *L) {
43   lua_pushnumber(L, l_tg(sinh)(luaL_checknumber(L, 1))
       );
44   return 1;
45 }
```

稍微值得留意的是，在这里还定义了一个宏 l_tg 。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 2.6: lmathlib.c: log

```
123 static int math_log (lua_State *L) {
124   lua_Number x = luaL_checknumber(L, 1);
125   lua_Number res;
126   if (lua_isnoneornil(L, 2))
127     res = l_tg(log)(x);
128   else {
129     lua_Number base = luaL_checknumber(L, 2);
130     if (base == 10.0) res = l_tg(log10)(x);
131     else res = l_tg(log)(x)/l_tg(log)(base);
132   }
133   lua_pushnumber(L, res);
134   return 1;
135 }
136
137 #if defined(LUA_COMPAT_LOG10)
138 static int math_log10 (lua_State *L) {
139   lua_pushnumber(L, l_tg(log10)(luaL_checknumber(L, 1)
        ));
140   return 1;
141 }
142 #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

[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。

[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

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 2.7: lmathlib.c: random

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

故，log(x)/log(10) 往往并不严格等于 log10(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。

```
221         lua_pushnumber(L, l_tg(floor)(r*(u-l+1)) + l);
                /* int in [l, u] */
222         break;
223       }
224       default: return luaL_error(L, "wrong number of
              arguments");
225     }
226     return 1;
227   }
```

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。

2.3 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 LPEG [6]。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 2.8: lstrlib.c: open

```
964 LUAMOD_API int luaopen_string (lua_State *L) {
965   luaL_newlib(L, strlib);
966   createmetatable(L);
967   return 1;
968 }
```

[5]C 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。

[6]LPEG 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/

这里多了一处 createmetatable 的调用，下面列出细节：

源代码 2.9: lstrlib.c: createmetatable

```
949 static void createmetatable (lua_State *L) {
950   lua_createtable(L, 0, 1);  /* table to be metatable
        for strings */
951   lua_pushliteral(L, "");  /* dummy string */
952   lua_pushvalue(L, -2);  /* copy table */
953   lua_setmetatable(L, -2);  /* set table as metatable
        for strings */
954   lua_pop(L, 1);  /* pop dummy string */
955   lua_pushvalue(L, -2);  /* get string library */
956   lua_setfield(L, -2, "__index");  /* metatable.
        __index = string */
957   lua_pop(L, 1);  /* pop metatable */
958 }
```

第一次看 Lua 的 API 使用流程，容易被 Lua Stack 弄晕。Lua 和 C 的交互就是通过这一系列的 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：

源代码 2.10: 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 2.11: 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 20 [8]。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 2.12: lstrlib.c: byte

```
141   if (posi + n <= pose)  /* (size_t -> int) overflow?
          */
142     return luaL_error(L, "string slice too long");
143   luaL_checkstack(L, n, "string slice too long");
144   for (i=0; i<n; i++)
145     lua_pushinteger(L, uchar(s[posi+i-1]));
```

[7]C 语言的 char 类型是 singed 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。

[8]LUA_MINSTACK 定义在 lua.h 中，默认值为 20 。

暂时不继续写这一章了。

2.4 暂且搁置

第三章　String 的实现

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，能更为高效的使用它。

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 /*
56 ** LUA_TSTRING variants */
57 #define LUA_TSHRSTR      (LUA_TSTRING | (0 << 4))   /*
       short strings */
58 #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，一个字符串一旦没有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

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

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，而是直接最加在 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

```
425 #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。

[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 。

[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。

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, l);  /* seed */
2   size_t step = (l>>5)+1;  /* if string is too long,
        don't hash all its chars */
3   size_t l1;
4   for (l1=l; 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于 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

[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。

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

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。

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 ^ l;
53   size_t l1;
54   size_t step = (l >> LUAI_HASHLIMIT) + 1;
55   for (l1 = l; 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) \
```

[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。

```
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, luaO_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 }
```

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;
```

[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 [9]。即，将哈系值相同的对象串在分别独立的链表中，实现起来更为简单。

```
139         o = gch(o)->next) {
140       TString *ts = rawgco2ts(o);
141       if (h == ts->tsv.hash &&
142           ts->tsv.len == l &&
143           (memcmp(str, getstr(ts), l * sizeof(char)) ==
                0)) {
144         if (isdead(G(L), o))  /* string is dead (but was
                not collected yet)? */
145           changewhite(o);  /* resurrect it */
146         return ts;
147       }
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 的垃圾收集过程是分步完成的。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)->strt;
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      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: createstrobj

```
98  static TString *createstrobj (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

```
434 typedef union Udata {
```

```
435   L_Umaxalign dummy;  /* ensures maximum alignment for
          'local' udata */
436   struct {
437     CommonHeader;
438     struct Table *metatable;
439     struct Table *env;
440     size_t len;  /* number of bytes */
441   } uv;
442 } 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　Table 的实现

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

[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

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

```
544 /*
545 ** Tables
546 */
547
548 typedef union TKey {
549   struct {
550     TValuefields;
551     struct Node *next;  /* for chaining */
552   } nk;
553   TValue tvk;
554 } TKey;
555
556
557 typedef struct Node {
558   TValue i_val;
559   TKey i_key;
560 } Node;
561
562
563 typedef struct Table {
564   CommonHeader;
565   lu_byte flags;  /* 1<<p means tagmethod(p) is not
        present */
566   lu_byte lsizenode;  /* log2 of size of 'node' array
        */
567   struct Table *metatable;
568   TValue *array;  /* array part */
569   Node *node;
570   Node *lastfree;  /* any free position is before this
```

```
        position */
571     GCObject *gclist;
572     int sizearray;  /* size of 'array' array */
573   } Table;
574
575
576
577   /*
578   ** 'module' operation for hashing (size is always a
        power of 2)
579   */
580   #define lmod(s,size) \
581           (check_exp((size&(size-1))==0, (cast(int, (s)
              & ((size)-1)))))
582
583
584   #define twoto(x)          (1<<(x))
585   #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

[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 。

```
67 #define dummynode                    (&dummynode_)
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       gnext(othern) = n;  /* redo the chain with 'n'
            in place of 'mp' */
425       *n = *mp;  /* copy colliding node into free pos.
             (mp->next also goes) */
426       gnext(mp) = NULL;  /* now 'mp' is free */
427       setnilvalue(gval(mp));
428     }
429     else {  /* colliding node is in its own main
            position */
430       /* new node will go into free position */
431       gnext(n) = gnext(mp);  /* chain new position */
432       gnext(mp) = n;
433       mp = n;
434     }
435   }
436   setobj2t(L, gkey(mp), key);
437   luaC_barrierback(L, obj2gco(t), key);
438   lua_assert(ttisnil(gval(mp)));
439   return gval(mp);
440 }
```

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

[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

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 LUAI_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 luaO_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_TNIL: return luaO_nilobject;
484     case LUA_TSHRSTR: return luaH_getstr(t, rawtsvalue
          (key));
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), nvalue(key))) /*
            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 luaO_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 (luai_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

[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

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   luai_hashnum(i, n);
83   if (i < 0) {
84     if (cast(unsigned int, i) == 0u - i)  /* use
         unsigned to avoid overflows */
85       i = 0;  /* handle INT_MIN */
86     i = -i;  /* must be a positive value */
87   }
88   return hashmod(t, i);
89 }
```

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 luai_hashnum 。这个函数默认定义在 llimits.h 中。

这里有两个预定义的版本，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 luai_hashnum(i,n)  \
233   { volatile union luai_Cast u; u.l_d = (n) + 1.0;  /*
        avoid -0 */ \
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]; };
```

[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

如果无法这种技巧来快速计算数字的哈希值，则改用一个性能不高，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 = 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

[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。

```
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 found */
164     }
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;
```

[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，但并非空值一定会截断长度统计。

```
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;
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",
38     "__concat", "__call"
```

[10]lua 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。

```
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]) :
        luaO_nilobject);
76  }
```

[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。

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",
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。

[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。

第五章　全局状态机及内存管理

Lua 可以方便的被嵌入 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。

5.1　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 5.1: lauxlib.c: luaL_newstate

```
918 static void *l_alloc (void *ud, void *ptr, size_t
      osize, size_t nsize) {
919   (void)ud; (void)osize;  /* not used */
920   if (nsize == 0) {
921     free(ptr);
922     return NULL;
923   }
924   else
925     return realloc(ptr, nsize);
926 }
927
928
929 static int panic (lua_State *L) {
930   luai_writestringerror("PANIC: unprotected error in
        call to Lua API (%s)\n",
931                         lua_tostring(L, -1));
932   return 0;  /* return to Lua to abort */
933 }
934
935
936 LUALIB_API lua_State *luaL_newstate (void) {
937   lua_State *L = lua_newstate(l_alloc, NULL);
938   if (L) lua_atpanic(L, &panic);
939   return L;
940 }
```

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 nsize 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 5.2: lmem.h: memory

```
17 #define luaM_reallocv(L,b,on,n,e) \
18         ((cast(size_t, (n)+1) > MAX_SIZET/(e)) ?  /*
               +1 to avoid warnings */ \
19                 (luaM_toobig(L), (void *)0) : \
20                 luaM_realloc_(L, (b), (on)*(e), (n)*(e
                       )))
21
22 #define luaM_freemem(L, b, s)   luaM_realloc_(L, (b),
       (s), 0)
23 #define luaM_free(L, b)         luaM_realloc_(L, (b),
       sizeof(*(b)), 0)
24 #define luaM_freearray(L, b, n)   luaM_reallocv(L, (b)
       , n, 0, sizeof((b)[0]))
25
26 #define luaM_malloc(L,s)        luaM_realloc_(L, NULL,
        0, (s))
27 #define luaM_new(L,t)           cast(t *, luaM_malloc(
       L, sizeof(t)))
28 #define luaM_newvector(L,n,t) \
```

```
29                  cast(t *, luaM_reallocv(L, NULL, 0, n,
                        sizeof(t)))
30
31 #define luaM_newobject(L,tag,s) luaM_realloc_(L, NULL,
       tag, (s))
32
33 #define luaM_growvector(L,v,nelems,size,t,limit,e) \
34           if ((nelems)+1 > (size)) \
35             ((v)=cast(t *, luaM_growaux_(L,v,&(size),
                   sizeof(t),limit,e)))
36
37 #define luaM_reallocvector(L, v,oldn,n,t) \
38    ((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。

源代码 5.3: lmem.c: realloc

```
75 void *luaM_realloc_ (lua_State *L, void *block, size_t
       osize, size_t nsize) {
76   void *newblock;
77   global_State *g = G(L);
78   size_t realosize = (block) ? osize : 0;
79   lua_assert((realosize == 0) == (block == NULL));
80 #if defined(HARDMEMTESTS)
81   if (nsize > realosize && g->gcrunning)
82     luaC_fullgc(L, 1);  /* force a GC whenever
         possible */
83 #endif
```

```
84    newblock = (*g->frealloc)(g->ud, block, osize, nsize
        );
85    if (newblock == NULL && nsize > 0) {
86      api_check(L, nsize > realosize,
87                      "realloc cannot fail when shrinking a
                          block");
88      if (g->gcrunning) {
89        luaC_fullgc(L, 1);  /* try to free some memory
              ... */
90        newblock = (*g->frealloc)(g->ud, block, osize,
              nsize);  /* try again */
91      }
92      if (newblock == NULL)
93        luaD_throw(L, LUA_ERRMEM);
94    }
95    lua_assert((nsize == 0) == (newblock == NULL));
96    g->GCdebt = (g->GCdebt + nsize) - realosize;
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 5.4: 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)",
53                      what, limit);
          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)))
```

5.2 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 5.5: 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 5.6: 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 }
```

5.2.1 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。

5.2.2 seed

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

seed 和字符串的 hash 算法相关，参见 3.1.1 节。

5.2.3 buff

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

buff 用于 Lua 源代码文本的解析过程。

5.2.4 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 5.7: lapi.c: lua_version

```
128 LUA_API const lua_Number *lua_version (lua_State *L) {
129   static const lua_Number version = LUA_VERSION_NUM;
130   if (L == NULL) return &version;
131   else return G(L)->version;
132 }
```

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)
```

[1] luaL_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。

源代码 5.8: lauxlib.c: luaL_checkversion

```
943 LUALIB_API void luaL_checkversion_ (lua_State *L,
      lua_Number ver) {
944   const lua_Number *v = lua_version(L);
945   if (v != lua_version(NULL))
946     luaL_error(L, "multiple Lua VMs detected");
947   else if (*v != ver)
948     luaL_error(L, "version mismatch: app. needs %f, 
        Lua core provides %f",
949                   ver, *v);
950   /* check conversions number -> integer types */
951   lua_pushnumber(L, -(lua_Number)0x1234);
952   if (lua_tointeger(L, -1) != -0x1234 ||
953       lua_tounsigned(L, -1) != (lua_Unsigned)-0x1234)
954     luaL_error(L, "bad conversion number->int;"
955                   " must recompile Lua with proper
                        settings");
956   lua_pop(L, 1);
957 }
```

5.2.5 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       luai_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 5.9: 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

[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。

[2]Lua 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。

[3]Continuation : `http://en.wikipedia.org/wiki/Continuation`

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

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5.2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 #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

```
391 union Value {
```

```
392   GCObject *gc;      /* collectable objects */
393   void *p;           /* light userdata */
394   int b;             /* booleans */
395   lua_CFunction f;   /* light C functions */
396   numfield           /* numbers */
397 };
```

从这里我们可以看到，引用类型用一个指针 GCObject * 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]

[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[8]。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。

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;
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  (LUAI_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 <= LUAI_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，由 func 引用正在调用的函数对象。了解这一点对调试嵌入式 Lua 代码有很大的好处。

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。

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[7]可以获

[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]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。

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。[8]

```
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
```

[8]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，或编写出试用于 Lua 其它版本的实现。

```
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
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[9]。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[10]，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[11]。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。[12]

[9] 这里涉及 C 调用层次上的系统堆栈保存和恢复问题。

[10] Java 就是这样，不用太考虑和原生代码的交互问题。

[11] 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。

[12] 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。

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_ULONGJMP)
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
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 用了一个内部 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->errorJmp) {  /* 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[13]。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。

[13]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。

公开 API 仅需要对它做一些封装即可。[14]

源代码 6.16: ldo.c: luaD_pcall

```
582 int luaD_pcall (lua_State *L, Pfunc func, void *u,
583                 ptrdiff_t old_top, ptrdiff_t ef) {
584   int status;
585   CallInfo *old_ci = L->ci;
586   lu_byte old_allowhooks = L->allowhook;
587   unsigned short old_nny = L->nny;
588   ptrdiff_t old_errfunc = L->errfunc;
589   L->errfunc = ef;
590   status = luaD_rawrunprotected(L, func, u);
591   if (status != LUA_OK) {  /* an error occurred? */
592     StkId oldtop = restorestack(L, old_top);
593     luaF_close(L, oldtop);  /* close possible pending
          closures */
594     seterrorobj(L, status, oldtop);
595     L->ci = old_ci;
596     L->allowhook = old_allowhooks;
597     L->nny = old_nny;
598     luaD_shrinkstack(L);
599   }
600   L->errfunc = old_errfunc;
601   return status;
602 }
```

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 、nny[15]、errfunc 都保存在 luaD_pcall 的 C 堆栈上，一旦 luaD_rawrunprotected 就可以正确恢复。

[14]luaD_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使用这个 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。

[15]nny 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

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[16]。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: restorestack

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。

[16]调用 luaF_close 涉及 upvalue 的 gc 流程，参见 GC 的章节。

```
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)
        */
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       if (L->hookmask & LUA_MASKCALL)
315         luaD_hook(L, LUA_HOOKCALL, -1);
316       lua_unlock(L);
317       n = (*f)(L);  /* do the actual call */
318       lua_lock(L);
319       api_checknelems(L, n);
320       luaD_poscall(L, L->top - n);
321       return 1;
322     }
323     case LUA_TLCL: {  /* Lua function: prepare its
          call */
```

```
324         StkId base;
325         Proto *p = clLvalue(func)->p;
326         luaD_checkstack(L, p->maxstacksize);
327         func = restorestack(L, funcr);
328         n = cast_int(L->top - func) - 1;  /* number of
                real arguments */
329         for (; n < p->numparams; n++)
330           setnilvalue(L->top++);  /* complete missing
                arguments */
331         base = (!p->is_vararg) ? func + 1 :
                adjust_varargs(L, p, n);
332         ci = next_ci(L);  /* now 'enter' new function */
333         ci->nresults = nresults;
334         ci->func = func;
335         ci->u.l.base = base;
336         ci->top = base + p->maxstacksize;
337         lua_assert(ci->top <= L->stack_last);
338         ci->u.l.savedpc = p->code;  /* starting point */
339         ci->callstatus = CIST_LUA;
340         L->top = ci->top;
341         if (L->hookmask & LUA_MASKCALL)
342           callhook(L, ci);
343         return 0;
344       }
345       default: {  /* not a function */
346         func = tryfuncTM(L, func);  /* retry with '
                function' tag method */
347         return luaD_precall(L, func, nresults);  /* now
                it must be a function */
348       }
349     }
350 }
```

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：Lua 函数需求的参数个数是明确的，所以需要先调整数据栈上的参数个数；CallInfo 里保存有字节码执行指针 savedpc ，将其指向 lua 函数对象中的字节指令区；之后初始化线程运行状态，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。[17]

源代码 6.19: 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。

[17]关于元表查询 luaT_gettmbyobj 的部分，参见 4.4 节。

luaD_poscall 做的工作也很简单，主要是数据栈的调整工作。

源代码 6.20: ldo.c: luaD_poscall

```
353 int luaD_poscall (lua_State *L, StkId firstResult) {
354   StkId res;
355   int wanted, i;
356   CallInfo *ci = L->ci;
357   if (L->hookmask & (LUA_MASKRET | LUA_MASKLINE)) {
358     if (L->hookmask & LUA_MASKRET) {
359       ptrdiff_t fr = savestack(L, firstResult);  /*
            hook may change stack */
360       luaD_hook(L, LUA_HOOKRET, -1);
361       firstResult = restorestack(L, fr);
362     }
363     L->oldpc = ci->previous->u.l.savedpc;  /* 'oldpc'
          for caller function */
364   }
365   res = ci->func;  /* res == final position of 1st
        result */
366   wanted = ci->nresults;
367   L->ci = ci = ci->previous;  /* back to caller */
368   /* move results to correct place */
369   for (i = wanted; i != 0 && firstResult < L->top; i
        --)
370     setobjs2s(L, res++, firstResult++);
371   while (i-- > 0)
372     setnilvalue(res++);
373   L->top = res;
374   return (wanted - LUA_MULTRET);  /* 0 iff wanted ==
        LUA_MULTRET */
375 }
```

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 6.21: ldo.c: luaD_call

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

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。[18]

钩子函数是一个 C 函数，用内部 API luaD_hook 封装起来。

源代码 6.22: 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);
```

[18]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 。

```
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。[19]

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

[19]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。

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 [20]这个项目。它用操作系统的协程来解决这个问题[21]，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

[20]CoCo - True C Coroutines for Lua `http://coco.luajit.org`

[21]CoCo 在 Windows 上使用了 Fiber。

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);
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 6.23: ldo.c: lua_yieldk

```
554 LUA_API int lua_yieldk (lua_State *L, int nresults,
      int ctx, lua_CFunction k) {
555   CallInfo *ci = L->ci;
556   luai_userstateyield(L, nresults);
557   lua_lock(L);
558   api_checknelems(L, nresults);
559   if (L->nny > 0) {
560     if (L != G(L)->mainthread)
561       luaG_runerror(L, "attempt to yield across
            metamethod/C-call boundary");
562     else
563       luaG_runerror(L, "attempt to yield from outside
            a coroutine");
564   }
565   L->status = LUA_YIELD;
566   ci->extra = savestack(L, ci->func);  /* save current
        'func' */
567   if (isLua(ci)) {  /* inside a hook? */
568     api_check(L, k == NULL, "hooks cannot continue
          after yielding");
569   }
570   else {
571     if ((ci->u.c.k = k) != NULL)  /* is there a
          continuation? */
572       ci->u.c.ctx = ctx;  /* save context */
573     ci->func = L->top - nresults - 1;  /* protect
```

```
        stack below results */
574     luaD_throw(L, LUA_YIELD);
575   }
576   lua_assert(ci->callstatus & CIST_HOOKED);  /* must
        be inside a hook */
577   lua_unlock(L);
578   return 0;  /* return to 'luaD_hook' */
579 }
```

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 6.24: ldo.c: lua_resume

```
523 LUA_API int lua_resume (lua_State *L, lua_State *from,
      int nargs) {
524   int status;
525   lua_lock(L);
526   luai_userstateresume(L, nargs);
527   L->nCcalls = (from) ? from->nCcalls + 1 : 1;
528   L->nny = 0;  /* allow yields */
529   api_checknelems(L, (L->status == LUA_OK) ? nargs + 1
        : nargs);
530   status = luaD_rawrunprotected(L, resume, L->top -
        nargs);
```

```
531   if (status == -1)  /* error calling 'lua_resume'? */
532     status = LUA_ERRRUN;
533   else {  /* yield or regular error */
534     while (status != LUA_OK && status != LUA_YIELD) {
          /* error? */
535       if (recover(L, status))  /* recover point? */
536         status = luaD_rawrunprotected(L, unroll, NULL)
              ;  /* run continuation */
537       else {  /* unrecoverable error */
538         L->status = cast_byte(status);  /* mark thread
              as 'dead' */
539         seterrorobj(L, status, L->top);
540         L->ci->top = L->top;
541         break;
542       }
543     }
544     lua_assert(status == L->status);
545   }
546   L->nny = 1;  /* do not allow yields */
547   L->nCcalls--;
548   lua_assert(L->nCcalls == ((from) ? from->nCcalls :
        0));
549   lua_unlock(L);
550   return status;
551 }
```

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 6.25: ldo.c: resume

```
484 static void resume (lua_State *L, void *ud) {
485   int nCcalls = L->nCcalls;
486   StkId firstArg = cast(StkId, ud);
```

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

```
513           firstArg = L->top - n;  /* yield results come
                  from continuation */
514         }
515         luaD_poscall(L, firstArg);  /* finish '
                luaD_precall' */
516       }
517       unroll(L, NULL);
518     }
519     lua_assert(nCcalls == L->nCcalls);
520   }
```

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 6.26: ldo.c: unroll

```
421 static void unroll (lua_State *L, void *ud) {
422   UNUSED(ud);
423   for (;;) {
424     if (L->ci == &L->base_ci)  /* stack is empty? */
425       return;  /* coroutine finished normally */
426     if (!isLua(L->ci))  /* C function? */
427       finishCcall(L);
428     else {  /* Lua function */
```

```
429         luaV_finishOp(L);  /* finish interrupted
                instruction */
430         luaV_execute(L);  /* execute down to higher C '
                boundary' */
431       }
432     }
433   }
```

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。顾需要先调用 luaV_finishOp ，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 6.27: ldo.c: finishCcall

```
400 static void finishCcall (lua_State *L) {
401   CallInfo *ci = L->ci;
402   int n;
403   lua_assert(ci->u.c.k != NULL);  /* must have a
          continuation */
404   lua_assert(L->nny == 0);
405   /* finish 'lua_callk' */
406   adjustresults(L, ci->nresults);
407   /* call continuation function */
408   if (!(ci->callstatus & CIST_STAT))  /* no call
          status? */
409     ci->u.c.status = LUA_YIELD;  /* 'default' status
            */
410   lua_assert(ci->u.c.status != LUA_OK);
411   ci->callstatus = (ci->callstatus & ~(CIST_YPCALL |
          CIST_STAT)) | CIST_YIELDED;
412   lua_unlock(L);
413   n = (*ci->u.c.k)(L);
414   lua_lock(L);
```

```
415   api_checknelems(L, n);
416   /* finish 'luaD_precall' */
417   luaD_poscall(L, L->top - n);
418 }
```

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[22]：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。[23]。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 6.28: ldo.c: recover

```
439 static CallInfo *findpcall (lua_State *L) {
440   CallInfo *ci;
441   for (ci = L->ci; ci != NULL; ci = ci->previous) {
        /* search for a pcall */
442     if (ci->callstatus & CIST_YPCALL)
443       return ci;
444   }
445   return NULL;  /* no pending pcall */
446 }
447
```

[22] 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。

[23] 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 。

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

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6.24 ，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。

源代码 6.29: lapi.c: lua_callk

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

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 6.30: lapi.c: lua_pcallk

```
925 struct CallS {  /* data to 'f_call' */
926   StkId func;
927   int nresults;
928 };
929
930
931 static void f_call (lua_State *L, void *ud) {
932   struct CallS *c = cast(struct CallS *, ud);
933   luaD_call(L, c->func, c->nresults, 0);
934 }
935
936
937
938 LUA_API int lua_pcallk (lua_State *L, int nargs, int
      nresults, int errfunc,
939                         int ctx, lua_CFunction k) {
940   struct CallS c;
941   int status;
942   ptrdiff_t func;
943   lua_lock(L);
944   api_check(L, k == NULL || !isLua(L->ci),
945     "cannot use continuations inside hooks");
946   api_checknelems(L, nargs+1);
947   api_check(L, L->status == LUA_OK, "cannot do calls
        on non-normal thread");
948   checkresults(L, nargs, nresults);
949   if (errfunc == 0)
950     func = 0;
951   else {
952     StkId o = index2addr(L, errfunc);
```

```
953     api_checkvalidindex(L, o);
954     func = savestack(L, o);
955   }
956   c.func = L->top - (nargs+1);  /* function to be
          called */
957   if (k == NULL || L->nny > 0) {  /* no continuation
          or no yieldable? */
958     c.nresults = nresults;  /* do a 'conventional'
            protected call */
959     status = luaD_pcall(L, f_call, &c, savestack(L, c.
            func), func);
960   }
961   else {  /* prepare continuation (call is already
          protected by 'resume') */
962     CallInfo *ci = L->ci;
963     ci->u.c.k = k;  /* save continuation */
964     ci->u.c.ctx = ctx;  /* save context */
965     /* save information for error recovery */
966     ci->extra = savestack(L, c.func);
967     ci->u.c.old_allowhook = L->allowhook;
968     ci->u.c.old_errfunc = L->errfunc;
969     L->errfunc = func;
970     /* mark that function may do error recovery */
971     ci->callstatus |= CIST_YPCALL;
972     luaD_call(L, c.func, nresults, 1);  /* do the call
            */
973     ci->callstatus &= ~CIST_YPCALL;
974     L->errfunc = ci->u.c.old_errfunc;
975     status = LUA_OK;  /* if it is here, there were no
            errors */
976   }
977   adjustresults(L, nresults);
978   lua_unlock(L);
```

```
979   return status;
980 }
```

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 6.31: 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     incr_top(L);
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 6.32: lapi.c: lua_error

```
1102 LUA_API int lua_error (lua_State *L) {
1103   lua_lock(L);
1104   api_checknelems(L, 1);
1105   luaG_errormsg(L);
1106   lua_unlock(L);
1107   return 0;  /* to avoid warnings */
1108 }
```

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. Indent style. `http://en.wikipedia.org/wiki/Indent_style`.

[8] Wikipedia. Nan. `http://en.wikipedia.org/wiki/NaN`.

[9] Wikipedia. Separate chaining. `http://en.wikipedia.org/wiki/Hash_table#Separate_chaining`.

源代码目录