

# Lua 源码欣赏

云风

2012 年 1 月 28 日



# 序

长达五年半的等待，Lua 5.2 终于在 2011 年末正式发布了。



# 目录

<b>第一章 概览</b>	<b>1</b>
1.1 源文件划分 . . . . .	1
1.2 代码风格 . . . . .	3
1.3 Lua 核心 . . . . .	4
1.4 代码翻译及预编译字节码 . . . . .	5
1.5 内嵌库 . . . . .	5
1.6 独立解析器及字节码编译器 . . . . .	6
1.7 阅读源代码的次序 . . . . .	6
<b>第二章 内置库的实现</b>	<b>9</b>
2.1 从 math 模块看 Lua 的模块注册机制 . . . . .	9
2.2 math 模块 API 的实现 . . . . .	11
2.3 string 模块 . . . . .	15



# 第一章 概览

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

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

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

下面，我们按这个划分来分拆解析。

## 1.1 源文件划分

从官网下载到 Lua 5.2 的源代码后<sup>3</sup>，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

---

<sup>1</sup>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>

<sup>2</sup> 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。

<sup>3</sup>本书讨论的 Lua 5.2 版可以在 <http://www.lua.org/ftp/lua-5.2.0.tar.gz> 下载获得

<sup>4</sup>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]<sup>5</sup> 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<sup>6</sup>，并有些修改，例如函数定义的开花括号没有另起一行。同时，也掺杂了一些 GNU 风格，比如采用了双空格缩进、在函数名和小括号间加有空格。代码缩进风格没有好坏，但求统一。

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

<sup>5</sup>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。

<sup>6</sup>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[6]。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。如果要参与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 `luaL.h` 中）外，其它都属于内部 API，禁止被外部程序使用<sup>7</sup>。

## 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 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` 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`<sup>8</sup>、`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。我们将在后面的章节来剖析其中的细节。

<sup>7</sup>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。

<sup>8</sup>H 是意取 Hash 之意。

## 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<sup>9</sup> 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。<sup>10</sup>

## 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。<sup>11</sup>

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

---

<sup>9</sup>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。

<sup>10</sup>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。

<sup>11</sup>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。

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。

12

## 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，不到两万行 C 代码<sup>13</sup>。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<sup>14</sup>：

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

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

之后、可以开始了解 Lua VM 的实现。

---

<sup>12</sup>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。

<sup>13</sup>Lua 5.2.0 版本的源代码分布在 58 个文件中，共 19808 行

<sup>14</sup>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\\_well\\_designed\\_that\\_you\\_would\\_consider\\_them\\_must\\_reads/](http://www.reddit.com/comments/63hth/ask_reddit_which_oss_codebases_out_there_are_so_well_designed_that_you_would_consider_them_must_reads/)

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。<sup>15</sup>

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。

---

<sup>15</sup>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](http://blog.codingnow.com/2011/04/lua_gc_multithreading.html)，在本书的后面，会重新领略 Lua 5.2.0 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 LUAMODAPI 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`<sup>1</sup>

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

<sup>1</sup>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]。不过要注意：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<sup>2</sup>。

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

```

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

```

## 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

<sup>2</sup>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。

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         );
35     return 1;
36 }
37
38 static int math_sin (lua_State *L) {
39     lua_pushnumber(L, l_tg(sin)(luaL_checknumber(L, 1)))
40         ;
41     return 1;
42 }
43
44 static int math_sinh (lua_State *L) {
45     lua_pushnumber(L, l_tg(sinh)(luaL_checknumber(L, 1))
46         );
47     return 1;
48 }
```

稍微值得注意的是，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`<sup>3</sup>。

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_COMPATLOG10)
138 static int math_log10 (lua_State *L) {
139     lua_pushnumber(L, l_tg(log10)(luaL_checknumber(L, 1)
140         ));
141     return 1;
142 }
143 #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<sup>4</sup>。早期的 Lua 函数看起来

<sup>3</sup> 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。

<sup>4</sup>因为人类更习惯用十进制计数，而计算机则在内部运算采用的是二进制。由于浮点数表示误差的缘故，`log(x)/log(10)` 往往并不严格等于 `log10(x)`，而是有少许误差。在我们常用的 X86 浮点指令集中，

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
204        needed also because on
205        some systems (SunOS!) 'rand()' may return a value
206        larger than RANDMAX */
207     lua_Number r = (lua_Number)(rand()%RANDMAX) / (
208         lua_Number)RANDMAX;
209     switch (lua_gettop(L)) { /* check number of
210        arguments */
211     case 0: { /* no arguments */
212         lua_pushnumber(L, r); /* Number between 0 and 1
213            */
214         break;
215     }
216     case 1: { /* only upper limit */
217         lua_Number u = luaL_checknumber(L, 1);
218         luaL_argcheck(L, 1.0 <= u, 1, "interval is empty
219            ");
220         lua_pushnumber(L, l_tg(floor)(r*u) + 1.0); /*
221            int in [1, u] */
222         break;
223     }
224     case 2: { /* lower and upper limits */
225         lua_Number l = luaL_checknumber(L, 1);
226         lua_Number u = luaL_checknumber(L, 2);
227         luaL_argcheck(L, l <= u, 2, "interval is empty")
228         ;
229         lua_pushnumber(L, l_tg(floor)(r*(u-l+1)) + l);

```

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。

```

                /* 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<sup>5</sup>。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<sup>6</sup>。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 }

```

<sup>5</sup>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。

<sup>6</sup>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<sup>7</sup>，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。

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<sup>8</sup>。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 2.11: 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]));

```

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

<sup>8</sup>LUA\_MINSTACK 定义在 lua.h 中，默认值为 20。



## 参考文献

- [1] Iso/iec 9899:201x. In *Programming languages - C*, chapter 6.5.11 Generic selection.
- [2] Iso/iec 9899:201x. In *Programming languages - C*, chapter 6.2.5 Types.
- [3] Roberto Ierusalimschy. The novelties of lua 5.2. 2011. <http://www.inf.puc-rio.br/~roberto/talks/novelties-5.2.pdf>.
- [4] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. The evolution of lua. *Proceedings of ACM HOPL III (2007) 2-1-2-26*. <http://www.lua.org/doc/hopl.pdf>.
- [5] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. *Lua 5.2 Reference Manual*, 2011. <http://www.lua.org/manual/5.2/manual.html>.
- [6] Wikipedia. Indent style. [http://en.wikipedia.org/wiki/Indent\\_style](http://en.wikipedia.org/wiki/Indent_style).



# 源代码目录

2.1	lmathlib.c: mathlib . . . . .	9
2.2	lmathlib.c: luaopen_math . . . . .	10
2.3	lauxlib.h: luaL_newlib . . . . .	10
2.4	lauxlib.c: luaL_setfuncs . . . . .	11
2.5	lmathlib.c: l_tg . . . . .	12
2.6	lmathlib.c: log . . . . .	13
2.7	lmathlib.c: random . . . . .	14
2.8	lstrlib.c: open . . . . .	15
2.9	lstrlib.c: createmetatable . . . . .	16
2.10	lstrlib.c: uchar . . . . .	16
2.11	lstrlib.c: byte . . . . .	17