Lua 的演进

lua 的优点：

- 可移植性
- 容易嵌入
- 体积小
- 高效率

这些优点都来自于 lua 的设计目标：简洁。从 Scheme 获得了很多灵感，包括匿名函数，合理的语义域概念

3 lua 前身

巴西被商贸限制，引入计算机软件和硬件受限，巴西人有强烈的民族情绪去创造自己的软件。三名作者都是同一个实验室 Tecgraf 的，这个实验室与很多工业实体有合作关系。成立的头十年，重点是创造交互性的图形软件，帮助合作伙伴进行设计。巴西石油公司是其中一个重要伙伴。有大量的遗留数据需要处理。于是诞生了 DEL，一个领域专用语言，主要用来描述数据流图的数据的。

后来人们对 DEL 需求越来越多，不止是一门简单的数据描述语言可以解决的了。

lua 为解决实际问题而生，受到三位作者学过的语言影响。自己创造的是..（两个句号）连接字符串。自豪的是，在 13 年的演进里，lua 的类型系统只修改了两次。lua 诞生的时候，基本类型包括 nil，number，string，table，C function，Lua function，userdata。97 年的时候，Lua3.0 将 C function 和 Lua function 合并了。03 年的时候，提出了 boolean 值类型，增加了 thread 协程类型。

1993 年，第一版 Lua 由 Waldemar 在 Roberto 指导下完成。词法分析用了 Unix 上经典的 yacc 和 lex。解释器将 lua 代码翻译成针对一个基于栈的虚拟机的指令。C API 很容易扩展，因此最早只有 5 个函数（print,tonumber,type,next,nextvar）和 3 个库（input/output, string, math)。

5 Lua 的历史

lua 的发布模式和其他社区不一样。alpha 版本已经相当稳定，beta 版本几乎可以作为 final 版，除非是用来修复 bug。这个发布模式对于 lua 的稳定性有很大帮助，但不利于尝试新特性。因此，从 5.0 版本开始，添加了新的 work 版本。work 版本是 lua 当前开发版的 snapshot，有助于 lua 社区迈向开源社区的哲学：早发布、多发布。

lua 的标准库被刻意保持在一个很小的范围，因为大部分需要的功能都会由宿主或第三方库提供。4.0 对 C API 重新设计了，C API 有很大改动，之后就向着完美一点点前进了。结果是不再有任何内置函数，所有标准库都是基于 C API 实现，没有通过特别的后门去访问 Lua 内部结构。

Lua 的 vm 在 4.0 版本以前一直是基于栈的。在 3.1 版本，我们对多个指令添加了变量来提高性能。后来发现这个实现复杂度很高，性能提升不明显，于是在 Lua 3.2 版本去掉了。从 Lua 5.0 开始，vm 改为基于寄存器。代码生成器因此有更多机会去优化和减少常见 Lua 程序的指令数了。

5.1 Lua 1

Lua 的成功使得 Lua 被大规模用在数据描述上，而预编译 Lua 代码为 VM 字节码的特性，迫使 Lua 编译器必须非常快，即使是对于大型项目。通过将 lex 产生的 scanner 替换为一个手写的版本，我们几乎让 Lua 编译器的速度翻了一倍。同时，我们修改了 Lua VM 的 table 构造函数，构造一个大型的表不再需要一条条指令传参数进去了，可以一次调用完成。从那以后，优化重点就变成了预编译时间。

1994 年，我们发布了带有这些优化的 lua 版本：Lua1.1。这次发布和第一篇描述 Lua 的设计和实现的文章是重合的。之前从未公开发布过的版本，就被称为 Lua 1.0（Lua1.0 的一个 1993 年 7 月的 snapshot 在 2003 年 10 月发布，以此庆祝 Lua 十周年）

Lua 1.1 最早以源码方式在 ftp 提供下载。这远在开源运动兴盛蓬勃之前。

Lua1.1 有限制性的用户协议，对于学术用途是免费的，但是商业用途则需要协商。那部分协议没有凑效：尽管我们有一些初始联系人，从没有商业使用是经过协商的。其它脚本语言（比如 Tcl）的免费促使我们意识到，限制商用甚至会反过来影响学术用途的发展，因为一些学术项目是打算最终走向市场的。因此，当发布 Lua2.1 的时候，我们将 Lua 作为无限制免费软件发布了。我们天真的以一种学院气的口吻对之前的协议重新措辞，并觉得新协议写的很直观了。稍后，随着开源许可证的散播，我们的协议文本变成了干扰着一些用户的噪音：我们的协议没有清晰的说明是否和 GPL 协议相容。在 2002 年 5 月，在邮件列表里进行了冗长的讨论后，我们决定将来发布的 Lua 版本（从 Lua 5.0 开始）都使用非常清晰直观的 MIT 协议。2002 年 7 月，自由软件基金会（FSF）确认了我们之前的协议是 GPL 兼容的。但我们已经决定采纳 MIT 协议了。对我们协议的疑问从此都消失了。

5.2 Lua 2

1990 年的时候，面向对象迈向巅峰，对于 Lua 没有面向对象的支持，我们受到了很大的压力。我们不想将 Lua 变成面向对象，因为我们不想“修复”一种编程范式（fix a programming paradigm）。特别是，我们不觉得 Lua 需要将对象和类作为基础语言概念，我们觉得可以透过 table 来实现（table 可以保存方法和数据，因为函数是第一类对象）。直到今天，Lua 也没有强加任何对象和类模型给用户，我们初心不变。很多用户建议和实现了面向对象模型；面向对象也是邮件列表里经常讨论的问题，我们觉得这是健康的。

另一方面，我们希望允许 Lua 可以面向对象编程。我们决定提供一套灵活的机制，让用户可以选择对应用来说合适的模型，而不是修复模型。1995 年 2 月发布的 Lua2.1，标志着这些灵活语义的问世，极大的增加了 Lua 的表达能力，从此，灵活语义就变成了 Lua 的标志。

灵活语义的一个目标是允许 table 作为对象和类的基础。为了实现这个目标，我们需要实现 table 的继承。另一个目标是将 userdata 变成应用数据的天然代理，可以作为函数参数而不只是一个句柄。我们希望能够索引 userdata，就好像他们只是一个 table，可供调用他们身上的方法。这让 Lua 可以更为自然的实现自己的主要设计目标：通过提供脚本访问到应用服务和数据，从而扩展应用。我们决定实现一套 fallback 机制，让 Lua 把未定义行为交给程序员处理，而不是直接在语言本身实现这些特性。

在 Lua2.1 的时候，我们提供了 fallback 机制，支持以下行为：table 索引，算术操作符，字符串拼接，顺序比较，函数调用。当这些操作应用到“错误”的类型上，对应的 fallback 就会被调用到，允许程序员决定 Lua 如何处理。table 索引 fallback 允许 userdata 和其它值类型表现的跟表一样。我们也定义了当 Key 不在 table 时的 fallback，从而实现多种形式的继承（通过委托）。为了完善面

向对象编程，我们添加了两个语法糖：`function a:foo(…)`就好比 `function a.foo(self,…)`一样，以及 `a:foo(…)`作为 `a.foo(a, …)`的语法糖。在 6.8 节我们会讨论 fallback 的细节。

从 Lua1.0 开始，我们就提供了值类型的内省函数（introspective functions）：type，可以用来获取 Lua 值的类型；next，可以用来遍历一个 table；以及 nextvar，可以遍历全局环境。（正如第四章所述，这是为了实现类似 SOL 的类型检查）为了应付用户对完整调试设施的强烈需求，1995 年 12 月发布的 Lua2.2 引入了一个 debug API 来获取运行中的函数信息。这个 API 为用户提供了以 C 语言编写自己的内省函数工具链的手段，比如编写自己的调试器和性能分析工具。debug API 刚开始的时候相当简洁：debug 库允许访问 Lua 的调用栈，访问当前执行的代码行行数，以及一个可以查找指定值的变量名的函数。根据 M.Sc.的 Tomas Gorham 的工作，debug API 在 1996 年 5 月发布的 Lua2.4 版本里得到了完善，提供了函数访问局部变量，提供了钩子在行数变化和函数调用时触发。

因为 Lua 在 Tecgraf 的广泛使用，很多大型的图形源文件都是用 Lua 写的，作为图形编辑器的输出格式。加载这些源文件会随着文件大小变得越来越大和越来越复杂而变的越来越长。从第一版开始，Lua 就预编译所有程序到字节码，再执行字节码。大型代码文件的加载时间可以通过保存 bytecode 来缩减。这和处理图形源文件特别有关系。所以在 Lua 2.4 版本，我们引入了一个外部编译器 Luac，可以编译一个 Lua 文件为字节码并保存为二进制文件。这个二进制文件的格式经过精心选择，可以轻松加载同时体积小巧。通过 luac，程序员可以在运行期避免词法分析和代码生成，这些操作早期还是比较耗时的。除了更快的加载，luac 还允许离线的语法检查，以及随意的用户改动。很多产品（比如模拟人生和 Adobe 的 Lightroom）都是透过预编译格式发布 Lua 脚本的。

在 luac 的实现过程里，我们开始将 Lua 的核心重构成清晰的分离模块。于是，我们现在可以轻易移除词法解析的模块（词法解析器，语法解析器和代码生成器），这些部分占了大约 35%的核心代码，剩下的部分可以加载预编译的 Lua 脚本。这种代码剪裁，对于在移动设备、机器人和感应器这些小设备里嵌入 Lua，是有显著意义的。

从第一版开始，Lua 就自带有一个库来进行字符串处理。这个库在 Lua2.4 之前功能有限。但是，随着 Lua 的成熟，Lua 需要进行更重量级的字符串处理。我们认为，沿革 Snobol，Icon，Awk 和 Perl 的传统，为 Lua 添加模式匹配是自然而然的。但是，我们不想将第三方的模式匹配引擎打包到 Lua 里面去，因为这些引擎通常都很大，我们也希望避开因为引入第三方库带来的代码版权问题。

1995 年的第二学期，作为 Roberto 指导下的学生项目，Milton Jonathan，Pedro Miller Rabinovitch，Pedro Willemsens 和 Vinicius Almendra 为 Lua 写出了一个模式匹配库。那个设计的经验引导我们写出了我们自己的模式匹配引擎。1996 年的 12 月，我们在 Lua2.5 中添加了两个函数：strfind（最早职能查找纯文本）和新的 gsub 函数（名字来源于 Awk）。gsub 函数可以用来全局替换符合指定模式的子串。它接受一个新的字符串或者一个函数作为参数，函数参数会在每次遇到匹配时调用，并预期该函数返回新子串以供替换。为了缩小实现的规模，我们没有支持完整的正则表达式。我们支持的模式包括字符类，重复，以及捕获（但是没有可选或组匹配）除了简洁性，这种模式匹配还十分强大，是 Lua 的一个强有力的补充。

那一年是 Lua 历史上的转折点，因为 Lua 获得了全球的曝光。在 1996 年 6 月，我们在《Software：Practice & Experience》杂志上发布了一篇 Lua 的文章，为 Lua 带来了外部的关注，至少是学术圈子的关注。在 1996 年的 12 月，Lua 2.5 刚刚发布后，杂志 Dr.Dobb's Journal 也发表了 Lua 的文章。Dr.Dobb's Journal 是一本面向程序员的流行刊物，那篇文章为 Lua 带去了软件工业界的关

注。在那次发布之后，我们收到了很多消息，其中一条是 1997 年 1 月收到的，来自 Bret Mogilefsky，LucasArts 出品的冒险游戏——Grim Fandango 的首席程序员。Bret 告诉我们，他从 Dr.Dobb's 上读到了 Lua，他们打算用 Lua 代替自己写的脚本语言。1998 年 10 月，Grim Fandango 发布，1999 年 5 月 Bret 告诉我们"大量的游戏都是用 Lua 写的"。那个时候，Bret 参加了 GDC（Game Developers' Conference, 游戏开发者会议）的有关游戏脚本的圆桌会议。他谈到了 Grim Fandango 应用 Lua 的成功经验。很多我们认识的开发者，都是在那次事件里认识到 Lua 的。在那以后，Lua 在游戏程序员之间口耳相传，变成了游戏工业里可资销售的技能。

因为 Lua 的国际化曝光，向我们提问 Lua 的信息越来越多。1997 年 2 月，我们建立了邮件列表，好更有效率的处理这些问题，让其他人也能帮忙回答问题，并开始建设 Lua 社区。这个列表至今发布了超过 38000 条消息。很多热门游戏对 Lua 的使用，吸引了很多人到邮件列表里。现在已经有超过 1200 个订阅者了。Lua 列表十分友善，同时又富有技术性，我们对此深感幸运。邮件列表逐渐变成了 Lua 社区的焦点所在，也是 Lua 演进的动力源泉。所有的重大事件，都是首先在邮件列表里出现的：发布重大通知，请求新特性，bug 报告等等。

在 Usenet 新闻组里建立 comp.lang.lua 讨论组曾经在邮件列表里讨论过两次，分别是 1998 年的 4 月和 1999 年的 7 月。两次的结论都是邮件列表的流量不能保证创建一个新闻组。而且，更多人倾向于邮件列表。随着多个阅读和搜索完整邮件存档的 web 界面问世，创建新闻组变得无关紧要了。

5.3 Lua 5.3

Lua2.1 里引入的 fallback 机制，可以很好的支持灵活扩展的语义，但这个机制是全局的：每个事件只有一个钩子。这让共享或重用代码变的很艰难，因为同一事件的 fallback 在模块里只能定义一次，不能共存。1996 年 12 月，听取了 Stephan Herrmann 的建议后，我们在 1997 年 7 月发布的 Lua 3.0 中，我们解决了 fallback 冲突问题。我们将 fallback 替换为 tag 方法：钩子是以(event, tag)的形式挂在字典里的。Tags 是在 Lua2.1 引入的整数标签，可以附在 userdata 上。最初的动机是希望同类的 C 对象，在 Lua 里都有相同的 tag(不过，Lua 没有强迫要对 Tag 提供解释)。Lua3.0 里，我们对所有值类型提供了 tag 支持，以支持对应的 tag 方法。6.8 节里还会继续讨论 fallback 的演进。

1998 年 7 月 Lua 3.1 引入了函数式编程，Lua 拥有了匿名函数和 upvalue 支持的函数闭包。引入闭包是被高阶函数的存在所启发的，例如 gsub，可以使用函数作为参数。在 Lua 3.1 点工作中，邮件列表里讨论了多线程和协作式多任务，这些讨论主要源于 Bret Mogilefsky 在 Grim Fandango 中，对 Lua2.5 和 Lua 3.1 alpha 版所作的改动。虽然没有最终定论，但这个话题一直很热门。协作式多任务在 2003 年 4 月发布的 Lua5.0 里提供了，详见 6.7 节。

从 Lua 1.0 到 Lua 3.2，C API 总体上没有变化，都是针对一个隐式 Lua 虚拟机进行操作的。但是，新的应用程序比如 web，需要多状态支持。为了解决这一问题，Lua 3.1 引入了多个独立 Lua 状态的设计，可以在运行时自由切换。而完全可重入的 API 则需要等到 Lua4.0。同时，两个非官方的 Lua3.2 修改版出现了，它们都带有显式的 Lua 状态，一个是 1998 年，Roberto Ierusalimschy 和 Anna Hester 为 CGILua 写的、基于 Lua 3.2 alpha 的版本。另一个是 1999 年，Erik Hougaard 基于 Lua 3.2 final 写的版本。Erik 的版本是公开发布的，并应用

在 Crazy Ivan robot 里面。CGILua 是作为 CGILua 发行版的一部分发布，从未以独立包的形式出现。

1999 年 7 月，Lua 3.2 主要是一个维护性的版本。没有新奇的特性，除了一个可以用 Lua 编写调试器相关代码的 debug 库。但是，从那时起，Lua 就处在一个相对稳定的版本，因此 Lua 3.2 有很长的生命期。因为下一个版本 Lua 4.0 是用的全新的不兼容 API，很多用户就只停留在 Lua 3.2，没有再迁移到 4.0 版本了。比如 Tecgraf 就从来没有迁移到 4.0 版本，打算直接上 Lua 5.0.很多 Tecgraf 的产品还是用的 Lua 3.2

5.4 Lua 4

2000 年 12 月，Lua 4.0 正式发布。正如上文所述，4.0 的主要改变是完全可重入的 API，为那些需要多份 Lua state 的应用而设计。因为改造 API 为完全可重入已经是主要改动，我们借此机会重新设计了 API，依赖清晰的堆栈实现与 C 层的值交换。这是 Reuben Thomas 在 2000 年 7 月提出的。

Lua 4 同时引入了 for 语句，这是邮件列表中的日常话题和大部分 Lua 用户最想要的特性。我们早期没有引入 for 语句，是因为 while 循环更为一般化。但是，用户总是投诉忘记在 while 循环的尾部更新控制变量，从而引起死循环。而且，我们在好语法这一点上没有达成一致。我们觉得 Modula 语言的 for 语句限制性太大了，因为它既不能迭代 table 里的元素，也不能迭代文件里的行。C 传统上的 for 循环也不适合 Lua。基于 Lua 3.1 引入的闭包和匿名函数，我们决定使用高阶函数来实现迭代。所以，Lua 3.1 提供了一个高阶函数来迭代

table，可以对 table 的每一对键值对回调用户自定义函数。比如，要打印 table t 里面的每一个键值对，只需要写 `foreach(t,print)`。

在 Lua 4.0 我们终于设计了一个 for 循环，它有两种方式：一个数字式的循环以及一个表遍历的循环（1997 年 10 月由 Michael Spalinski 提出）。这两种方式覆盖了大部分常用应用环境；对于更一般化的循环，依然有 while 循环可供使用。要打印 table t 里的所有键值对，可以这样写：

```
for k, v in t do
  print(k, v)
end
```

添加 for 循环只是一个很简单的一个改动，但它的确改变了 Lua 程序的外观。Roberto 不得不重写了 Programming in Lua 草稿里的许多例子。Roberto 在 1998 年开始写书了，但是他从来都没写完，因为 Lua 是一个不断变动的目标。随着 Lua 4.0 的发布，书里大部分内容需要改写，几乎所有的代码块都要重写了。

Lua 4.0 发布后，我们开始为 Lua 4.1 工作。在 Lua 4.1 版本，我们面临的主要挑战，大概是要不要支持、如何支持多线程吧，这在当时是一个大问题。随着 Java 的用户增长以及 Pthreads 的出现，很多程序员开始考虑多线程，将其作为编程语言的关键特性进行考量。但是，对于我们来说，Lua 支持多线程需要考虑几个严肃的问题。首先，在 C 层面支持多线程就会用到非 ANSI C 的原语——尽管 Pthread 很流行，但仍然有很多平台（当时和现在）都缺乏这个库的支持。第二，更重要的是，我们不相信标准多线程模型：那个共享内存的抢占式并发模型。我们仍然认为，对于 `a=a+1` 都没有确定结果的语言，没人能写出正确的程序。

对于 Lua 4.1，我们开始用 Lua 的经典方式解决这些难题：我们只实现了一个简单的多栈共存的机制，我们称为 threads（线程）。外部的库可以使用这些 Lua

线程来实现多线程，比如基于 Pthreads 来实现。同样的机制也能用来实现协程，一个协作式的、非抢占的多线程模型。2001 年 7 月，Lua 4.1 alpha 发布，带有额外的多线程和协程库支持；同时也引入了弱表，以及标志性的基于寄存器的虚拟机。我们一直很想用基于寄存器的虚拟机进行实验。

Lua 4.1 alpha 发布后的第二天，John D.Ramsdell 在邮件列表里开始了关于语法域的大讨论。经过数十封邮件的讨论，我们清晰的发现，Lua 需要完全词法作用域的支持，而不是 Lua 3.1 开始的 upvalue 机制。在 2001 年 10 月，我们想到了实现高效的完全词法作用域的支持，并在同年的 12 月发布了一个可以工作的版本。那个版本还引入了新的 table 混合模型，可以在合适的时候将 table 作为数组实现。因为那个版本实现了新的基础算法，我们决定作为可工作版本来发布，尽管我们已经为 Lua 4.1 发布了一个 alpha 版本。

2002 年 2 月，我们为 Lua 4.1 发布了一个新的工作版本，带有三个相对新颖的特性：一个基于迭代函数的通用的 for loop，取代 tags 和 fallbacks 的元表和元方法，以及协程。在那次发布后，我们意识到 Lua 4.1 带来了太多的巨变——可能 Lua 5.0 比较适合作为下个版本的版本号。

5.5 Lua 5

对于 Lua 4.1 这个版本号来说，最后的改动来自于 Christian Lindig 和 Norman Ramsey 在哈佛大学召开的 Lua 库设计会议。这次会议的一个主要结论是，Lua 需要某种模块系统。我们一直认为模块可以透过 table 来完成，但是连标准库都没有按这种方式实现。于是我们决定下个版本迈出这一步。

将库函数放入 table 里面是一个巨大的冲击，因为这会影响到所有使用了至少一个库函数的程序。比如，老版本的 strfind 函数现在叫 string.find(在 string 库

里的 find 域，存储在叫 string 的 table 里)；openfile 变成了 io.open；sin 变成了 math.sin，诸如此类。为了让转换变的简单点，我们提供了一个兼容脚本，在里面定义了老函数去应用新函数：

```
strfind = string.find
openfile = io.open
sin = math.sin
…
```

但是，将库打包到 table 里面去始终是一个大改动。2002 年 6 月，当我们放出了带有这一改动的可工作版本时，我们放弃了"Lua 4.1"这个名字，并将其命名为"Lua 5.0 work0"。最终版本的进展从那时开始就相当稳定了，2003 年 4 月，我们发布了 Lua 5.0。这次发布让 Lua 的特性得以稳定下来，让 Roberto 可以完成他的新书了，新书在 2003 年 12 月发布。

Lua 5.0 发布后不久，我们立即开始 Lua 5.1 的工作。最初的动机是实现增量式的垃圾回收系统，以满足游戏程序员的需要。Lua 使用的是传统的标记-清除垃圾回收算法，直到 Lua 5.0 为止，垃圾回收都是原子执行的。副作用是对于某些应用，可能会在垃圾回收时有很长的暂停。在那个时间点，我们的主要考虑是，添加写屏障需要实现增量式垃圾回收，对于 Lua 的性能会有负面影响。为了补偿这一劣势，我们也常侍了分代回收。我们也希望保留老一套的自适应行为，可以根据总内存占用调节垃圾收集的频率。而且我们希望回收器保持简洁的特点，就像 Lua 的其他部分一样。

我们在增量式分代 GC 上花了超过一年的时间。但因我们没有接触过对内存有强烈需求的应用（比如游戏），所以很难在真实场景中测试收集器的表现。从 2004 年的 3 月到 12 月，我们放出了数个可工作版本，试图得到收集器在应用中表现的具体反馈。我们最终收到了奇怪的内存分配行为的报告，并成功重现了但没有解释。2005 年 1 月，一名 Lua 社区的活跃成员 Mike Pall，通过内存分配图解释了问题的根源：在某些特定场景，增量行为、分代行为和自适应行

为之间会有微妙的交互，导致收集器“自适应”到越来越低频率的收集上去。因为太复杂和不可预测，我们放弃了分代方式，并在 Lua 5.1 实现了更简单的增量回收。

在这段时间里，程序员尝试了 Lua 5.0 的模块系统。新包开始涌现，老包也开始转移到新系统去。包作者们希望指导构建模块的最佳实践。2005 年 7 月，在 Lua 5.1 的开发期间，一场由 Mark Hamburg 组织的国际性 Lua 会议，在 San Jose 的 Adobe 召开。其中一场演讲是关于 Lua 5.1 的新特性，引发了关于模块和包的冗长讨论。结果是我们对模块系统作出了一些细微但收效甚大的改变。尽管我们对 Lua 有“机制而非策略”的指南，我们还是定义了一系列的策略来编写模块和加载包，并做了些小改动让这些策略跑的更好。2006 年 2 月，Lua 5.1 发布。尽管 Lua 5.1 的初衷是增量垃圾收集，模块系统的改进可能是最容易留意到的。另一方面，增量垃圾收集没有被注意到，说明了它成功的避免了长时间的停顿。

6 特性演进

在这一节，我们会详细讨论 Lua 的一些特性的演进。

6.1 类型

Lua 的类型系统是相对稳定的。很长时间里，Lua 只有 6 个基本类型：nil，number，string，table，function 和 userdata（实际上，直到 Lua 3.0 为止，之前的 C 函数和 Lua 函数有不同的内部类型，但是这个差异对于调用者是透明的）。唯一真正的改变来自于 Lua 5.0，这个版本引入了两个新类型：thread 和 boolean。

引入的类型 thread 用于表示协程。像其他 Lua 值类型一样，thread 是第一类的值。为了避免创造新的语法，所有针对 thread 的基础操作都是通过 library 来提供的。

很长时间里，我们都很抗拒引入布尔值类型：nil 就是 false，其他所有一切都是 true。这种情况对于我们的目的是简洁够用的。但是，nil 也作为 table 里元素缺失的值，以及未定义变量的默认值。在某些应用里，需要一个 table 的域被置为 false 但仍然可见表示存在，一个显式的 false 可以表示这种情况。Lua 5.0 里，我们最终引入了布尔值类型 true 和 false。nil 还是作为 false。回顾这个设计，可能在布尔表达式里引用 nil 报错会是更好的解决办法，就跟其他表达式里一样。这样，nil 扮演未定义值类型的代理，就会更加具有一致性。但是，这个改动很可能会导致很多现存的程序报错。LISP 有类似的问题，空列表代表了 nil 和 false。Scheme 显式使用 false，并将空列表作为 true。但一些 Scheme 实现仍然将空列表作为 false。

6.2 Table

Lua 1.1 有三种语法来创建一个 table：`@()`，`@[]`和`@{}`。最简单的形式是`@()`，用于创建一个空表。可以在创建时提供一个可选的长度参数，作为优化性能的提示信息。`@[]`这种形式是用来构建数组的，比如`@[2,4,9,16,25]`。在这种 table 里，key 是隐式的从 1 开始的自然数。`@{}`这种形式是用来构造记录的，比如`@{name="John",age=35}`。这种 table 是键值对的集合，键是显式的字符串。一个 table 可以用这三种的其中一种方式构造，构造后可以动态修改，不因构造方式受限。此外，还能在构造列表和记录的时候提供用户函数调用，比如`@foo[]`或者`@foo{}`。这个语法是从 SOL 里继承的，是过程数据表述的表达式，Lua 的一

个主要特性(参见第二节)。语法是构建好 table 后，调用对应的函数，将构造好的 table 作为单参数传入函数。函数允许随意检查和修改 table，但是函数的返回值会被忽略：table 就是表达式的最终值。

Lua 2.1 里，table 创建的语法是统一的，并得到简化：开头的@被移除了，唯一的构造方式是{…}。Lua 2.1 也允许混合的构造方式，比如

```
grades{8.5, 6.0, 9.2; name="John", major="math"}
```

数组的部分和记录的部分被分号分割开。最后，`foo{...}`变成了`foo({...})`的语法糖。也就是说，函数加上表构造器，变成了普通的函数调用。所以，函数需要显式返回 table（或其他值）。从构造器里拿掉@是一个平凡的改动，但它的确改变了语言的感觉，不仅仅是它的外观。改变语言观感的平凡改动不应该被忽视。

然而，这个语法的简化和 table 构造器的语义改变带来了副作用。Lua 1.1 里，`=`用来判断相等。Lua 2.1 里 table 构造器统一之后，表达式`{a=3}`变成有歧义了，因为它可以表示```("a", 3)````这个键值对，或者(1, b)，其中 b 是表达式 a=3 的值。为了解决二义性，Lua 2.1 用`==`取代了`=`。改动后，`{a=3}`表示一个含有键值对`("a", 3)`的 table，而`{a==3}`表示`(1, b)`。

这些改变让 Lua 2.1 变得不兼容 Lua 1.1 了（所以大版本号也变了）。但是，因为那时候实际上所有的 Lua 用户都是 Tecgraf 的，这并不是一个致命的改变：现存的代码可以经过我们编写的特定工具进行转换。table 构造器的语法从那时起几乎不变了，唯一的例外是 Lua 3.1 引入的：记录的 key 可以用上任意表达式了，比如`{[10*x+f(y)]=47}`。尤其是这一改动允许了 key 为任意的字符串了，包括保留字和带有空格的字符串。因此，`{function=1}`是无效的，因为`function`是一个保留字，但是`{["function"]=1}`是有效的。从 Lua 5.0 开始，也可以自由的混合数组部分和记录部分了，因此没有必要再在 table 构造器里使用分号了。

虽然 table 的语法演进了，但是 table 的语义得到完整保留：table 依然是关联数组，可以存放任意键值对。但是，实践中经常使用的 table 要么是单纯的数组（连续的整型 key）或者记录（字符串作为 key）。因为 table 是 Lua 里唯一的数据结构机制，我们投入了大量的努力去实现 table，让 table 在 Lua 的核心代码里高效运行。直到 Lua 4.0 为止，table 都是作为纯哈希表实现的，所有的键值对都是显式存储的。在 Lua 5.0 版本我们引入了 table 的混合表示：每个 table 包含了一个哈希部分和一个数组部分，两个部分都可以是空的。Lua 检测一个 table 是不是作为一个数组来使用，并自动将数字索引的值移动到数组部分，而非原本的存储在哈希部分。这种分裂只在底层实现层次进行；访问 table 域是透明的，即使是对虚拟机来说。table 会自动根据内容使用两个部分。

这个混合机制有两个优点。第一，访问整型 key 的操作会变得更快了，因为不再需要哈希。第二，更重要的是，数组部分只占原来哈希部分的一半大小，因为哈希部分需要同时存储 key 和 value，而数组部分的 key 已经隐含在下标了。结果是，如果一个 table 是作为数组使用的，它的表现就像数组一样，只要它的整型 key 是密集分布的。而且，哈希部分没有内存或者时间的代价，因为作为数组使用时，哈希部分不存在。反过来说，如果 table 是作为记录使用而非数组，那么数组部分就是空的。这些节省下来的内存是重要的，因为对于 Lua 程序来说，创建大量小 table 是很常见的（比如用 table 来表示 object）。Lua 的 table 也能优雅的处理稀疏数组：语句 `a={[1000000000]=1}`在哈希部分创建了一个键值对，而非一个 10 亿元素的数组。

另一个打造高效 table 实现的原因，是我们可以使用 table 实现各种各样的任务。比如，在 Lua 5.0 版本里的标准库函数，从 Lua 1.1 开始就作为全局变量存

在，现在移动到 table 中去了。最近，Lua 5.1 带来了完整的包和模块系统，这些系统都是基于 table 实现的。

table 扮演了 Lua 核心的一个突出角色。有两个场景我们直接用 table 代替了核心代码的特殊数据结构：Lua 4.0 版本，用 table 来表示全局环境（用于保存全局变量），Lua 5.0 实现了可扩展语义（参见 6.8）。从 Lua 4.0 开始，全局变量就存储在普通的 Lua table 里，称为全局 table，这是 John Belmonte 在 2000 年 4 月提出的简化建议。在 Lua 5.0 我们用元表和元方法取代了 tag 和 tag 方法（Lua 3.0 引入的）。元表是普通的 Lua table，元方法是作为元表的域存储的。Lua 5.0 也引入了环境 table，可以附加到 Lua 函数上；它们就是 Lua 函数索引的全局环境。Lua 5.1 将环境变量 table 扩展到 C 函数、userdata 和协程，取代了全局的环境变量。这些改动简化了 Lua 的实现、Lua 和 C 程序员所用的 API，因为全局变量和元方法可以在 Lua 里操控，不再需要特殊函数了。

6.3 字符串

字符串在一门脚本语言里扮演了主要的角色。因此，创建和处理字符串的设施是脚本语言易用性的重要部分。

文本字符串的语法有一个有趣的演进过程。从 Lua 1.1 开始，一个文本串可以用单引号或者双引号来定义，也可以包含像 C 一样的转义序列。同时使用单引号和双引号来定义字符串，并且具有相同语义，在当时有点不寻常。（比如，在脚本语言的传统里，Perl 是会展开双引号字符串里的变量的，而单引号字符串则保持不变）这两种引号表示方式允许字符串包含其中一种引号，而不需要使用转义符。但是任意文本的字符串还是需要转义符序列的。

Lua 2.2 引入了长字符串，传统编程语言所没有，却普遍存在于脚本语言的一项特性。长字符串可以有许多行，不需要解析其中的转义符序列；它们提供了一

个灵活的方式，将任意文本作为字符串，不再需要担心其中的内容（是否需要转义符）。但是，要为长字符串定义一个好的语法，没有想象中的简单，尤其是它们会被用于包含任意的程序文本（其中可能包含其他的长字符串）。这让我们思考两个问题：长字符串应该如何结束，它们是否可以嵌套。直到 Lua 5.0 为止，长字符串都是被包含在一对匹配的`[[...]]`中，并且可以包含嵌套的长字符串。可惜，结束分隔符`]]`可以是有效 Lua 程序的一部分，以一种不平衡的方式出现：`a[b[i]]`，或者在其他上下文环境里，比如 XML 的`<[!CDATA[...]]>`。

Lua 5.1 引入了长字符串的新形式：用`[===[…]===]`包围文本，`=`的数量可以是任意的（包括 0）。这种新的长字符串不嵌套：一个长字符串遇到匹配数量的`=`就会结束。不管怎么说，现在包裹任意文本变得更简单了，即使文本包含其他长字符串或者不匹配的`]=…=]`序列：只要使用适当数量的`=`字符。

6.4 块注释

Lua 的注释是–开始，到行尾结束。这是最简单的注释，非常有效。很多语言使用单行注释，不过它们用的符号不一样。使用–表示注释的语言包括 Ada 和 Haskell。

我们从未觉得需要多行注释，或者块注释，除非是作为关闭一块代码的快捷方式。使用什么语法永远是一个问题：使用 C 语言的熟悉的`/*...*/`语法，或者其他语言的，都和 Lua 的单行注释格式不搭配。同样，块注释是否能嵌套也是一个问题，困扰着用户，并影响到词法分析器的复杂度。嵌套的块注释的场景，一般是程序员需要注释掉某些代码块。自然，它们希望代码块里的注释可以正确处理，这只有在块注释可以嵌套的情况下才能发生。

ANSI C 支持块注释，但是不允许嵌套。C 程序员通常使用 C 预处理器的惯用法来关闭代码块：`#if 0 …#endif`。这种方式有清晰的优势，它可以优雅的处理好被关闭代码里的注释。带着同样的动机和启发，我们强调了关闭 Lua 代码块的需要——不是块注释的——通过在 Lua 3.0 引入类似 C 语言的 pragma 进行条件编译。尽管块注释可以用上条件编译，我们不觉得这是它的正确用法。在 Lua 4.0 的改造工作中，我们认为条件编译带来的词法器复杂度太大，不值得继

续支持，同时对于用户来说，它的语义对于用户有一定复杂性，对于完整宏支持的设施也没有达成共识（参见第 7 章）。所以，Lua 4.0 里我们去掉了条件编译的支持，Lua 仍然没有支持块注释。

块注释是在 Lua 5.0 才被最终引入的，形式是`--[[...]]`。因为他们故意模仿了长字符串（参见 6.3）的语法，所以很容易修改词法器进行支持。这种相似性也帮助用户掌握概念和语法。块注释也可以被用于关闭代码：惯用法是在需要关闭代码的前后加上两行，分别包含`--[[`和`--]]`。要打开代码，只需要在第一行添加一个`-`：这两行便都变成单行注释。
像长字符串一样，块注释也可以嵌套，同样有长字符串所遇到的问题。特别是包含不平衡的`]]`有效的 Lua 代码，比如`a[b[i]]`，不能在 Lua 5.0 被可靠的注释掉。Lua 5.1 长字符串的新语法也能用于块注释，形式是`--[===[…]===]`，这个问题因此得到简洁稳定的解决方案。

6.5 函数

Lua 里的函数一直是第一类对象。一个函数可以在运行时创建，通过编译和执行包含其定义的字符串实现。随着 Lua 3.1 引入的匿名函数和 upvalue，程序员可以在运行时创建函数，不再需要从文本里编译了。

Lua 的函数无论是 C 或者 Lua 的，都没有声明。被调用时，他们会接受不定数量的参数：多余的参数会被丢掉，缺失的参数会被赋予 nil 值。（这吻合了多重赋值的语义）。C 函数一直都能处理不同数量的参数。Lua 2.5 介绍了可变参数类型 Lua 函数，标志是参数列表以…结尾（只在 Lua 3.0 出现的实验特性）。当一个变参函数被调用，对应…的参数将会收集到一个叫 `arg` 的 table 里。这种方式虽然很简单便捷，但是要把这些参数传给另一个函数，就需要解包这个 table。因为程序员经常将参数传递给另一个函数，Lua 5.1 允许`...`用于参数列表和赋值表达式的右值。这避免了没必要的创建 `arg` table。

Lua 执行的基本单位是代码块（chunk）；它只是一个语句序列。Lua 的一个代码块就像其他语言的 main 程序：它可以包含函数定义和可执行代码。（实际上，一个函数定义就是可执行代码：一个赋值）同时，一个代码块和一个普通的 Lua 函数非常相似。比如，代码块和普通 Lua 函数一直有着同一类字节码。

但是，在 Lua 5.0 之前，代码块需要一些内部的“魔法”来开始执行。Lua 2.2 开始，代码块开始类似普通函数，那时候，作为未记录在文档的特性，函数外可以定义局部变量了（只在 Lua 3.1 变成了官方发布）。Lua 2.5 允许代码块返回值。Lua 3.0 代码块变成了内部的函数，只是他们在被编译后立即执行；他们不以函数形式暴露给用户层。最终，在 Lua 5.0 迈出了这一步，将代码块的加载和执行变成了两步，来为宿主程序员提供更好的控制性和错误报告。结果，Lua 5.0 里代码块变成了日常无参数的匿名函数。Lua 5.1 里代码块变成了匿名变参函数，因为可以在运行时进行值传递。这些值都可以透过新的...机制进行传递。从别的角度看，代码块就像其他语言的模块一样：他们一般用来提供全局环境的函数和变量。最初，我们没想过 Lua 会用于大规模编程，所以我们不觉得需要显式的模块。而且，我们觉得 table 就足够应付模块的需要了。在 Lua 5.0 里，我们将所有标准库打包进 table 里了，以此说明 table 足够应付模块。这鼓励了其他人将库打包进 table 里，让共享库变得更为简单。我们现在觉得 Lua 可以用来大规模编程了，特别是在 Lua 5.1 带来了基于 table 的包系统和模块系统之后。

6.6 词法范围

在 Lua 开发的早期，我们开始考虑全词法域的第一类函数。这是一个优雅的构造，完美演绎了 Lua 的少而精哲学，提供少量但强大的构造方式。同时也让 Lua 适合进行函数式编程。但是，我们找不到全词法的合理实现方式。一开始，Lua 就使用了一个简单的基于数组的栈来记录活跃记录（所有临时变量和

局部变量存储的地方）。这个实现简单高效，我们找不到理由推翻它。当我们允许嵌套函数和全词法域之后，一个内部函数使用的变量，可能有着比创建它的函数更长的生命期，所以我们不能用栈的方式去处理这些变量。

简单的 Scheme 实现在堆上分配栈帧。1987 年，Dybvig[20]描述了如何使用栈去分配栈帧，前提是栈帧不包含嵌套函数使用的变量。他的方法需要编译器提前知道，一个变量是不是嵌套函数的自由变量。这不适合 Lua 编译器，因为 Lua 编译器解析完表达式后就立即生成操作变量的代码，它无法得知任一变量是否会在稍后的运行中用作嵌套函数的自由变量。我们希望实现 Lua 时保持这个设计，因此没有使用 Dybvig 的方法。基于相同的原因，我们也不能使用先进的编译器技巧，比如数据流分析。

现在，有很多优化策略去避免使用堆来存储栈帧（e.g.,[21]），但它们全都需要编译器有中间表达方式，Lua 编译器却没有。McDermott 的基于栈的栈帧分配建议[36]，特别强调是针对解释器的，是我们所知唯一的不需要中间表示方式的代码生成。就像我们的实现一样【31】，他的提案将变量放在栈上，如果它们被嵌套的闭包使用又超出了词法域，就将它们移动到堆上。但是，他的提案假设环境是以关联表的方式组织的。所以，将一个环境移动到堆上，解释器只需要修正列表头部，所有的局部变量自动到堆上了。Lua 使用真实的记录作为活跃记录，局部变量的访问转译成直接访问栈顶对应偏移地址，所以不能使用 McDermott 的方法。

在很长一段时间里，这些困难一直困扰着我们，无法在 Lua 里引入嵌套的第一类函数，并实现全词法域支持。最终，在 Lua 3.1 里，我们接受了一个称为 upvalue 的妥协实现。在这个方案里，内部函数不能访问和修改运行时的外部变量，但可以访问函数创建时的变量。这些变量称为 upvalue。upvalue 的主要优势，在于可以简单的实现：所有局部变量在栈上；创建一个函数时，函数被打包在一个闭包之中，闭包包含了函数使用的外部变量的副本。换句话说，upvalue 是外部变量的冻结值。为了避免误解，我们创建了新的语法来访问

upvalue：`%varname`。这个语法清晰表明代码是在访问变量的冻结值，而不是变量本身。upvalue 虽然不能修改，但依然非常有用。有需要的时候，我们可以使用 table 作为 upvalue 来模拟可变外部变量：尽管我们不能改变变量指向的 table，我们仍可以改变 table 的域本身。这个特性在以下场景特别有用：匿名函数作为高阶函数的参数，用于 table 遍历和模式匹配。

2000 年 12 月，Roberto 在他的书【27】的第一版手稿里写到"Lua 通过 upvalue 提供了一种合适的词法作用域形式"。在 2001 年 7 月，John D. Ramsdell 在邮件列表中争论到，"一门语言要么是词法作用域的，要么不是；在词法作用域前添加'合适的'这个形容词毫无意义"。那条消息促使我们寻找更好的解决方案，以及支持全词法域的方式。2001 年 10 月，我们做出了了全词法域支持的初步实现，并发布到邮件列表中。构想是每个 upvalue 都是间接访问的，当变量在词法域的时候指向栈；在词法域结束的时候，会将变量值移到堆区，并将指针指向堆区。开放闭包（带有指向栈的 upvalue）被保存在一个列表里，允许重定向和重用开放的 upvalue。重用对于获得正确语义有其必要性。如果两个闭包，共享一个外部变量，各自有自身的 upvalue，那么在词法域结束的时候，每个闭包都有自己的变量拷贝了，但正确的语义要求它们共享变量。为了保证重用，创建闭包的算法是这样做的：对于闭包使用的每个外部变量，首先搜索开放闭包的列表，如果发现一个 upvalue 指向同样的外部变量，就重用那个 upvalue；否则，创建新的 upvalue。

Edgar Toering，Lua 社区的活跃成员，误解了我们对词法域的描述。结果是他理解的比我们的最初的构想要更好：只保留开放 upvalue 的列表，而不是开放闭包的列表。因为闭包使用的局部变量的数量，一般远小于使用它们的闭包的数量（局部变量的数量被代码文本静态限制了），他的实现比我们的更加高效。而且也更容易适配到协程中（基本上同一时间实现），因为我们可以为每个栈保留一个独立的 upvalue 列表。我们在 Lua 5.0 添加了全词法域支持，用的

正是这个算法，因为它符合了我们的所有期望：它可以用一遍编译器实现；没有让只访问内部局部变量的函数增加运行代价，因为它们依然在栈上处理所有局部变量；而且访问外部局部变量的代价仅仅是一次额外的中转【31】。

6.7 协程

我们一直在为 Lua 寻找某种形式的第一类延续性。这种寻找是受 Scheme 的第一类延续的启发诞生的（Scheme 真是我们的灵感源泉），也是游戏程序员对于"软"多线程的需要（一般描述为"以某种方式挂起一个角色，并且稍候继续"）。

2000 年的时候，Maria Julia de Lima 在 Lua 4.0 alpha 版实现了完整的第一类延续性，作为她在读 PhD 的部分工作成果【35】。她使用了一个简单的实现，因为就像词法域问题一样，以更高明的技巧实现协程，对于 Lua 总体上的简洁而言，实在太复杂了。对于她的实验来说，结果是令人满意的，但是作为最终产品，就显得太慢了。无论如何，她的实现发现了 Lua 的一个奇特问题。因为 Lua 是一门可扩展语言，有可能（而且很普遍）会从 C 调用 Lua，或者从 Lua 调用 C。因此，在一个 Lua 程序执行的任意点，当前延续性通常都是有部分是 Lua，部分是 C。尽管可以操作 Lua 的延续性（特别是通过操控 Lua 调用栈），却不能操作 C 的延续性，尤其是在 ANSI C 标准下。那时候，我们对这个问题没有很深的了解。特别是，我们找不到 C 调用的准确限制。Lima 的实现只是简单的禁止了所有 C 调用。再次强调，这对于她的实验而言是可用的，但是对于 Lua 的官方发行版来说，完全无法接受。因为 Lua 和 C 代码可以轻易融合正是 Lua 的一个标记。

2001 年 12 月，Thatcher Ulrich 在没有理解这一困难的前提下，在邮件列表里宣布： 我已经为 Lua 4.0 创建了一个补丁，让 Lua 到 Lua 的调用变成非递归的（比如"无堆栈"(stackless)）。这带来了 `sleep()`调用的实现， 允许从宿主程序退出【省略】，并让 Lua state 可以通过新的 API 函数 lua_resume 回到刚刚的执行环境。
换句话说，他提议了一个非对称的协程机制，基于两个原语：yield（他称为 sleep）和 resume。他的补丁遵循了邮件列表中，Bret Mogilefsky 提到的，Grim Fandango 公司针对 Lua 2.5 和 3.1 添加协作式多任务，这一实现的高阶描述（Bret 不能提供细节，因为是公司私有的）。

在这次公告之后，2002 年 2 月在哈佛举行了 Lua 库设计大会，有了一些关于 Lua 的第一类延续性的讨论。有人认为，如果第一类延续性太复杂，我们可以实现一次性延续性。其他人认为实现对称的协程更好。但我们找不到这些机制的合适实现，可以解决 C 调用的问题。

我们花了不少时间研究，才意识到为什么在 Lua 实现对称协程那么困难，为什么 Ulrich 的基于非对称协程的实现，避开了我们遇到的难点。一次性延续性和对称协程都涉及到操作全延续性。所以，只要这些延续性包含了任何 C 的部分，就不可能捕捉到他们（除非用 ANSI C 以外的工具）。相反，一个基于 yield 和 resume 的非对称的协程机制，只操作偏序延续性：yield 捕获了当前调用点到对应 resume 之间的延续性【19】。在非对称协程上，当前延续性可以包含 C 的部分，只要他们在被捕获的延续性之外。换句话说，唯一的限制是我们不能跨 C 调用的 yield。

意识到这一点之后，基于 Ulrich 的概念验证实现，我们在 Lua 5.0 实现了非对称协程。主要的改动是为虚拟机执行指令的解释器循环，不再能递归调用了。在之前的版本里，当解释器循环执行一个 CALL 指令，他会递归调用自己来执行被调用的函数。Lua 5.0 里，解释器更像是一个真实的 CPU：当他执行一个 CALL 指令的时候，他将一些上下文信息推到栈上，然后处理被调用函数，被调用函数返回的时候，再恢复上下文。改成这样以后，实现协程变得很直接了。

不像大多数非对称协程的实现，Lua 里协程是 stackfull【19】的。通过协程，我们可以实现对称协程，甚至是 Scheme 的 call / 1cc 操作符（调用当前一次性延续）。但是，使用 C 函数在这些实现里还是受到严重限制。

我们希望 Lua 5.0 引入的协程成为一个标志，标志着协程作为强有力的控制结构的复兴。

6.8 可扩展语义

在 5.2 节，我们介绍了可扩展语义的一个例子，Lua 2.1 里允许程序员使用 fallback 作为通用机制，处理 Lua 未定义的场景。fallback 因此提供了一种可恢复异常处理的受限形式。尤其是，通过使用 fallback，我们可以让一个值对不是为其设计的操作进行响应，或者让一个值表现得像另一个值一样。例如，我们可以让 userdata 和 table 响应算术运算符，userdata 可以表现得像 table 一样，字符串表现得像函数一样，诸如此类。此外，我们可以让一个 table 响应不存在的 key，这是实现继承的基石。通过 table 索引的 fallback 机制，以及一点点定义和调用方法的语法糖，面向对象编程及继承在 Lua 里成为了现实。

尽管对象、类和继承不是 Lua 的核心概念，他们可以直接在 Lua 里实现，并根据应用程序的需要采取多种多样的方式实现。换句话说，Lua 提供了机制，而非策略——我们由始至终都紧密追随的宗旨。

最简单的继承是通过委托继承，Self 率先引入这机制并被其他基于原型的语言所采用，比如 NewtonScript 和 JavaScript。下面的代码展示了 Lua 2.1 里通过委托实现继承的一个实现。

```
function Index(a, i)
  if i == "parent" then
    return nil
  end
  local p = a.parent
  if type(p) == "table" then
    return p[i]
  else
    return nil
  end
end
setfallback("index", Index)
```

当访问一个 table 的缺失域（比如属性或者方法），就会触发索引的 fallback。实现继承就是将索引的 fallback 指向一条向上的祖系链，并有可能再次触发 fallback，直到遇到一个拥有指定域的 table，或者这条链完结。

在设置了索引 fallback 以后，以下代码会打印 `red`，虽然 `b` 没有 `color` 域：

```
a=Window{x=100,y=200,color="red"}
b=Window{x=300,y=400,parent=a}
print(b.color)
```

通过 `parent` 域进行委托，这过程中没有黑魔法或者硬编码。程序员拥有完整的自由：他们可以使用 `parent` 或者其他名字，可以通过尝试一系列的父系函数实现多继承，诸如此类。我们没有硬编码任一行为的决定，引出了 Lua 的主要设计概念：元机制。我们提供了方法让用户可以编码他们需要的特性，以他们想要的方式实现，并只实现他们需要的特性，而非将语言本身塞满特性。

Fallback 机制极大扩展了 Lua 的表达能力。但是，fallback 是全局的句柄：对于每个事件，只有一个函数可以被执行。结果就是，很难在一个程序里混合不同的继承机制，因为只有一个钩子来实现继承（只有索引 fallback）。对于一个单一的开发组，基于自己的对象系统进行开发，这可能不是什么严重的问题，但是一个组尝试使用另一个组的代码，就会变成问题，因为他们的对象系统并不一定一致。不同机制的钩子可以串在一起，但是链式调用很慢，很复杂，充满错误，并且不礼貌。fallback 链不鼓励代码共享和重用；实际上几乎没有人使用。这让使用第三方库变得很难。

Lua 2.1 允许标记 userdata。Lua 3.0 我们将标记扩展到所有值，并用标记方法（tag methods）取代了 fallback 机制。标记方法是只在带有指定标记的值上执行的 fallback。这让实现独立的继承机制成为可能。这种情况下不需要链式调用，因为一个标记的方法不会影响另一个标记的方法。

标记方法机制工作的很好，一直存续到 Lua 5.0 为止，我们在 Lua 5.0 实现了元表和元方法来取代标记和标记方法。元表只是普通的 Lua table，所以可以用 Lua 直接操作，不需要特殊函数。就像标记一样，元表可以用来表示 userdata 和 table 的用户定义类型：所有“同类”对象应该共享同一个元表。不像标记，元表和他们的内容会在所有引用消失后自动被回收掉。（相反，标记和标记方法会等到程序结束才会被回收。）元表的引入同时简化了实现：标记方法需要在 Lua 核心代码里添加特殊的私有表示方法，元表主要就是标准的 table 机制。

下面的代码展示了 Lua 5.0 里，继承是如何实现的。index 元方法取代了 index 标记，元表里则是用__index 域来表示。代码通过将 b 的元表里的__index 域指向 a，实现了 b 继承 a。（一般情况下，index 元方法都是函数，但我们允许它设为 table，以直接支持简单的委托继承。）

```
a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400}
setmetatable(b, {__index = a})
print(b.color) ->red
```

6.9 C API

Lua 提供了 C 函数库和宏，允许宿主和 Lua 通信。这层 Lua 和 C 之间的 API 是 Lua 的一个主要组成部分；它让 Lua 变成了一门嵌入式语言。

就像 Lua 的其他部分一样，API 在 Lua 的演进里发生了很多变化。可惜，和语言本身相比，API 设计很少受到外来影响，主要是因为这方面很少研究活动。

API 一直是双向的，因为从 Lua 1.0 开始，我们就考虑了从 C 调用 Lua，以及从 Lua 调用 C，并认为两者同样重要。能从 C 调用 Lua 让 Lua 变成了扩展语言，即一门用于通过配置、宏和其他终端用户定制功能，来扩充应用程序特性的语言。能从 Lua 调用 C 让 Lua 变成了可扩展语言，因为我们可以用 C 函数扩展 Lua，为其提供新设施。(这就是为什么我们说 Lua 是一门可扩展的扩展语言【30】)这两个视角的共同点是，API 必须处理好 C 和 Lua 之间的失配：C 是静态类型的，Lua 是动态类型的，C 是人工管理内存，Lua 是自动进行垃圾回收的。

目前，C API 通过引入一个虚拟栈来交换 Lua 和 C 的数据，解决了这两个失配问题。每个 Lua 的 C 函数调用都会使用一个新的栈帧，包含了函数调用的参数。如果 C 函数需要给 Lua 返回值，它就在返回前把这些值压到栈上。

每个栈槽可以保存任一 Lua 值类型。每个 Lua 类型都有 C 的唯一表示（比如字符串和数字），目前有两个 API 函数：一个是注入函数，用于压栈一个 Lua 值，对应给出的 C 值；一个是投影函数，可以返回一个 C 值，对应指定栈位置的 Lua 值。Lua 值类型里没有对应 C 表示的（比如 table 和函数），可以通过 API 及他们的栈位置来操作。

实际上，所有的 API 函数都从栈上获得他们的操作数，并将结果放到栈上。因为栈可以存放任意 Lua 值类型，这些 API 函数都可以操作任意 Lua 类型，因此解决了类型失配问题。为了避免 C 使用中的 Lua 值被回收掉，栈上的所有值都不会被回收。当一个 C 返回，它的 Lua 栈帧消失了，所有的 C 函数使用的 Lua 值都会被释放。如果没有其他引用，这些值最终都会被回收。这就解决了内存管理的失配问题。

我们花了好多时间才实现成现在这套 API 的样子。为了讨论 API 的演进，我们以等价于如下 Lua 函数的 C 代码进行说明：

```
function foo(t)
    return t.x
end
```

换句话说，这个函数接受一个单参数，参数类型应该是 table，并返回 table 里储存在 x 域的值。尽管很简单，这个例子足够说明 API 里的三个重要问题：如何获取参数，如何索引 table，如何返回值。

在 Lua 1.0 里，foo 函数的 C 代码如下：

```
void foo_l(void) {
    lua_Object t = lua_getparam(1);
    lua_Object r = lua_getfield(t, "x");
    lua_pushobject(r);
}
```

注意，我们所索引的值是存储在字符串索引 x，因为 t.x 是 t["x"]的语法糖。而且所有的 API 都是以 lua_(或者 LUA_)开头的，以此避免和其他 C 库的命名冲突。

为了暴露这个 C 函数给 Lua，并使用 foo 作为函数名，我们会这样做

```
lua_register("foo", foo_l);
```

之后，foo 就能像普通 Lua 函数一样从 C 调用了：

```
t = {x = 200}
print(foo(t))        -> 200
```

API 的一个关键部件是 lua_Object 类型，定义如下：

```
typedef struct Object *lua_Object;
```

简而言之，lua_Object 是一个抽象类型，用于在 C 里表示 Lua 值。传递给 C 函数的参数，通过调用 lua_getparam 获得，这个函数会返回 lua_Object。在上述例子里，我们调用一次 lua_getparam 来获得 table，这个 table 应当是 foo 函数的第一个参数。（其余的参数会自动略过。）当 table 在 C 里变成可用的（作为 lua_Object），我们就可以通过调用 lua_getfield 函数获取 x 域的值。这个值在 C 里也是表示为 lua_Object，最后会通过 lua_pushobject 压栈，送回到 Lua 中去。

栈也是 API 的另一个关键部件。它用来从 C 里传值给 Lua。每个 Lua 类型都有对应的 C 版本 push 函数：number 类型有 lua_pushnumber，string 类型有 lua_pushstring，特殊值 nil 可以用 lua_pushnil。也有允许 C 回传任意 Lua 值到 Lua 的 lua_pushobject 函数。当一个 C 函数返回，所有栈上的值都会返回给 Lua，作为 C 函数的结果（Lua 的函数可以返回多个值）。

概念上，lua_Object 是一个 union 类型，因为它可以指代任意 Lua 值。很多脚本语言，比如 Perl，Python 和 Ruby，依然使用 union 类型来在 C 层表示它们的值。这种表示方法的主要弊端，在于很难为语言设计垃圾回收。没有额外信息的情况下，垃圾回收器不可能知道，一个值是否被 C 代码里的一个 union 值任用。没有这个信息，垃圾回收器有可能回收值，导致 union 变成了悬空指针。即使这个 union 是 C 函数的局部变量，这个 C 函数还是能再次调用 Lua，并触发垃圾回收流程。

Ruby 通过检查 C 栈来解决这个问题，这种方法难以移植。Perl 和 Python 则是以另一种方式实现，通过为 union 类型提供显式引用计数函数来解决这个问题。一旦你增加了一个值的引用计数，垃圾回收器就不会回收那个值了，直到你将计数减为 0。但是，让程序员保证引用计数的正确性并不容易。很容易会产生错误，但后期很难查出来（对内存泄漏和悬空指针排查过的人都可以作证）。此外，引用计数解决不了循环引用的问题。

Lua 从来没有提供过这类引用计数函数。Lua 2.1 之前，为了保证一个未被引用的 lua_Object 不被回收，你可以做的事情不多，顶多就是自己持有这个对象的引用，并避免调用 Lua。（只要你可以保证 union 指向的值同样储存在一个 Lua 变量里，你就是安全的。）Lua 2.1 带来了一个重要的变化：它跟踪了所有传给 C 的 lua_Object，保证它们在 C 函数活跃的时候不会被回收掉。当 C 函数返回到 Lua 之后，（只有在这个时候）所有这些 lua_Object 的值引用都会被释放，所以它们可以被回收。【JNI 使用一个类似的方法来处理局部引用】

更具体地说，Lua 2.1 里，一个 lua_Object 不再是一个指向 Lua 内部数据结构的指针，而是一个内部数组的索引，这个内部数组存储了所有传给 C 的值：

```
typedef unsigned int lua_Object;
```

这个改动让 lua_Object 的使用变得可靠了：当一个值在数组里，Lua 就不会回收它。当 C 函数返回了，整个数组就被释放，并回收掉没有其他引用的函数所使用的值。（这个改动也给实现垃圾回收带来了更多的灵活性，因为它可以按需移动对象了；但是，我们没有这么做）

对于简单的使用，Lua 2.1 的行为是非常实际的：这种方案安全，C 程序员也不用担心引用计数。每个 lua_Object 就像 C 里的一个局部变量：对应的 Lua 值会保证处理它的 C 函数的生命期内都存活。对于复杂的使用场景：这个简单的方

法有两个缺陷，需要额外的机制保证：有时候，lua_Object 的值需要锁定一段比 C 函数生命期更长的时间；有时候，需要更短的锁定时间。

第一种情况有一个简单的解决办法：Lua 2.1 引入了引用系统。函数 lua_lock 从栈上取一个 Lua 值，并返回一个引用。这个引入是一个数字，可以以后任意时间使用来获得 Lua 值，使用 lua_getlocked 函数即可。（同时还有 lua_unlock 函数，用于销毁一个引用）通过这种引用方法，要保存非本地的 C 变量就变得简单了。

第二种情况更为微妙。存在内部数组的对象只有函数返回的时候才会被释放。如果函数使用了太多的值，就会发生数组越界，或者内存不足错误。比如，考虑以下的高阶迭代函数，会迭代调用函数并打印结果，直到调用返回 nil：

```
void l_loop(void) {
  lua_Object f = lua_getparam(1);
  for (;;) {
    lua_Object res;
    lua_callfunction(f);
    res = lua_getresult(1);
    if (lua_isnil(res)) break;
    printf("%s\n", lua_getstring(res));
  }
}
```

这段代码的问题是，每个调用返回的字符串都不能回收，直到循环的底部（整个函数结束为止），因此有可能发生数组越界或者内存耗尽。这种错误很难追踪，因此 Lua 2.1 的实现设置了内部数组保存 lua_Object 的上限。Lua 会报错：“C 函数里太多对象”而非泛泛的“内存耗尽”错误，让错误变得容易跟踪，但没有完全避免这一问题。

为了解决这个问题，Lua 2.1 的 API 提供了两个函数，lua_beginblock 和 lua_endblock，为 lua_Object 的值创建动态范围（blocks）；所有在 lua_beginblock 之后创建的值，都会在对应的 lua_endblock 调用后，从内部数组里被移除。但是，因为块原则（block discipline）不能强制让 C 程序员遵

守，很容易就会忘记使用这些区块。而且，这些显式作用域控制用起来有点棘手。比如，一个欠周到的修复前述例子的方法，是用块包裹住 for 循环的循环体，但这样做存在问题：我们需要在 break 语句所在位置，也调用 lua_endblock。这个 Lua 对象扩大生命周期的困难，经历了数个版本，最后在 Lua 4.0 才解决，我们重构了整个 API。无论如何，就像我们之前提到的那样，对于普通应用，API 是非常易用的，而且大部分程序员永远不会遇到这里描述的情景。更重要的是，API 是安全的。错误用法会产生定义好的错误，但不会产生悬空指针或者内存泄露。

Lua 2.1 还为 API 带来了其他改变。其中之一是引入了 lua_getsubscript，允许使用任何值来索引 table。这个函数没有显式参数：它从栈上取出 table 和 key。老的 lua_getfield 被重定义为宏，以实现兼容：

```
#define lua_getfield(o,f) \
    (lua_pushobject(o), lua_pushstring(f), \
     lua_getsubscript())
```

（C API 的向后兼容性通常用宏来实现，只要代价可以接受。）

除了上述修改，从语法上讲，API 从 Lua 1 到 Lua 2 的变化是很少的。比如说，我们的说明函数 foo 可以直接用 Lua 1.0 的版本，在 Lua 2.0 依然可以运行。lua_Object 的含义变了，lua_getfield 用新的原语实现了，但对于普通用户来说，就好像没有变化一样。因此，API 一直很稳定，持续到 Lua 4.0 版本。

Lua 2.4 扩展了引用机制以支撑弱引用。Lua 程序的一个常见设计，是使用一个 Lua 对象（通常是一个 table）作为一个 C 对象的代理。常见的场景是，C 对象必须知道它的代理是什么，并保留对代理的引用。但是，那个引用会阻止对代理对象的回收，即使对象从 Lua 里不能再访问了。在 Lua 2.4 版本，就可以创

建一个弱引用指向 proxy；那个引用不会阻止代理对象的回收。获取一个被回收的引用将返回一个特殊值 LUA_NOOBJECT。

Lua 4.0 为 C API 带来了两个新颖的设计：支持多个 Lua 虚拟机和用于 C 和 Lua 交换值的虚拟栈。要实现多个独立的 Lua 虚拟机，就需要去掉所有全局的虚拟机。直到 Lua 3.0 为止，都只有一个 Lua 虚拟机存在，而且使用了很多静态变量，分散在代码里。Lua 3.1 引入了多个独立 Lua 虚拟机；所有静态变量都归集到一个单一的 C 结构体里去。添加了一个新的 API 来切换虚拟机，但任意时刻只有一个 Lua 虚拟机是活跃的。所有其他 API 函数都是针对活跃虚拟机操作的，而活跃虚拟机并没有出现在调用里。Lua 4.0 在 API 引入了显式的 Lua 虚拟机。这引起了和前述版本的不兼容。【17】所有和 Lua 通讯的 C 代码（尤其是向 Lua 注册的 C 函数）需要在 C API 中添加一个显式的虚拟机变量。既然所有 C 函数都必须重写，我们借此机会在 Lua 4.0 里大改了 C-Lua 交流方式：我们替换了 lua_Object 的概念，代之以显式的虚拟栈，用来在 Lua 和 C 之间双向通信。这个栈也能用来存储临时值。

在 Lua 4.0 里，我们的 foo 函数可以改写如下：

```
int foo_l (lua_State *L) {
  lua_pushstring(L, "x");
  lua_gettable(L, 1);
  return 1;
}
```

第一个区别是函数签名：foo_l 现在接受一个 Lua 虚拟机，用于接收操作数和返回函数返回值。在之前版本里，函数调用完毕后，所有残留在栈上的值都会返回 Lua。现在，因为所有操作都使用栈，它可以包含除返回值外的中间值，所以函数需要告诉 Lua，栈上到底有几个元素是返回值。另一个区别是，不再需

要 lua_getparam 了，因为函数参数在函数开始时就在栈上，并可以透过他们的 index 直接访问，就好像其他栈上的值一样。

最后的区别是 lua_gettable 的使用，这个函数用于替代 lua_getsubscript，用于访问 table 里的域。lua_gettable 接受一个栈索引（而不是一个 Lua 对象），索引指向所操作的 table，从栈顶弹出 key，并将结果压栈。该函数不改变 table 在栈中的位置，因为 table 经常重复索引。在函数 foo_l 里，lua_gettable 使用的 table 在栈位置 1，因为他是函数的第一个参数，key 是字符串"x"，需要在调用 lua_gettable 之前压栈。这个函数调用将栈中的 key，换成了对应的 table 里的值。所以，在 lua_gettable 之后，栈上有两个值：栈位置 1 的 table，和栈位置 2 上 key 索引到的结果。这个 C 函数返回 1，告诉 Lua 使用栈顶的值作为函数的唯一返回值。

为了更清楚的说明新的 API，这是我们循环例子的 Lua 4.0 实现：

```
int l_loop (lua_State *L) {
  for (;;) {
    lua_pushvalue(L, 1);
    lua_call(L, 0, 1);
    if (lua_isnil(L, -1)) break;
    printf("%s\n", lua_tostring(L, -1));
    lua_pop(L, 1);
  }
  return 0;
}
```

为了调用一个 Lua 函数，我们将它压到栈上，并压入其参数（上面的例子没有）。然后我们调用 lua_call，告诉 lua 要从栈上获取多少个参数（因此隐式的表达了函数在栈上的位置），要在调用里获取多少个结果。在例子里，我们没有使用参数，期待一个结果。lua_call 函数将函数和参数从栈上移除，并压入指定数目的结果。调用 lua_pop 会从栈上去掉一个返回值，让栈在循环开始时处

在相同的位置。为方便起见，我们可以用正数索引栈，表示从底部开始的位置，或者负数来表示从栈顶开始的位置。在示例中，我们使用-1 作为索引，让 lua_isnil 和 lua_tostring 从栈顶开始索引元素，该位置包含了函数调用的结果。

后见之明，在 API 里使用一个单独的栈是很明显的简化措施，但是 Lua 4.0 发布的时候，很多用户都抱怨新 api 的复杂性。尽管 Lua 4.0 的 API 有更简洁的概念模型，直接操作栈还是需要一些思考来保证正确性。很多用户都宁愿用前一版的 API，即使它没有任何清晰的概念模型来表达其原理。简单的任务不需要概念模型，之前的 API 因此工作的很好。更复杂的任务经常会打破用户自定义的自有模式，因为大部分的用户从未用 C 来编码复杂项目。所以，新 API 第一眼看上去太复杂了。但是，这些怀疑论最后都消失了，因为用户理解并肯定了新模型，这个模型证明了自身的简洁、不易出错。

Lua 4.0 多个虚拟机共存的可能性，导致了有关引用机制的意料之外的问题。之前，一个 C 库如果需要保留固定对象，可以创建这个对象的引用，并存储在全局 C 变量里。在 Lua 4.0，如果一个 C 库是和多个虚拟机协作的，它就需要为每个虚拟机保留单独的引用，不能用全局 C 变量来表示了。为了解决这个难题，Lua 4.0 引入了注册表（registry），注册表就是一个普通的 Lua 表，只在 C 层使用。有了注册表，C 库想要保留 Lua 对象，就能选择唯一的 key 索引，并在注册表里用这个 key 来关联对象。因为每个独立的 Lua 虚拟机都有自己的注册表，C 库可以在所有虚拟机里都使用同样的 key 来操作对应的对象。

我们可以轻易的在注册表上实现之前的引用机制，只需要使用整数 key 来表示索引即可。要创建一个新的索引，我们只需要找到没使用的整数 key，并在那个 key 存储值。获取一个引用就是一个简单的 table 访问。但是，我们不能使用注册表实现弱引用。所以，Lua 4.0 保持了之前的引用机制。Lua 5.0 里，由

于语言本身引入了弱表，我们终于可以将核心里的引用机制干掉，并移到库里。

C API 在向着完整性缓慢演进。从 Lua 4.0 开始，所有标准库函数都可以只使用 C API 来编写。在 Lua 4.0 之前，Lua 有数目可观的内置函数（Lua 1.1 有 7 个，Lua 3.2 有 35 个），大部分可以使用 C API 编写，但因为追求速度，我们并没有这样做。有少量内置函数是不能用 C API 来编写的，因为 C API 并不完整。比如，Lua 3.2 之前是不可能用 C API 来遍历 table 的内容的，尽管在 Lua 里可以使用内置函数 next 去实现。目前，C API 也尚不完整，不是所有 Lua 能干的都可以用 C API 完成；比如，C API 还缺少对 Lua 值进行算术运算的函数。我们计划下个版本解决这个问题。

6.10 userdata

从第一版开始，Lua 的一个重要特性就是可以直接操作 C 数据，这个特性是通过提供 userdata 这种特殊 Lua 数据类型来实现的。这种能力是 Lua 可扩展性的基础。

对于 Lua 程序来说，userdata 类型在 Lua 的演进里从未改变：尽管是第一类值类型，userdata 还是透明的类型，在 Lua 里唯一有效的操作就是相等性判断。其他针对 userdata 的操作（比如创建，探测，修改）都需要 C 函数提供。

对于 C 函数而言，userdata 类型在 Lua 的演进里经历很多变更。在 Lua 1.0 里，一个 userdata 值只是一个简单的 void*指针。这种简化方法等主要问题是，C 库没有办法检查一个 userdata 是不是有效的。尽管 Lua 代码不能创建 userdata 值，但可以将一个库创建的 userdata 传给另一个库，而另一个库期待的其实是指向不同结构的指针。因为 C 函数没有机制检查这种失配，指针适配

往往对应用程序造成致命后果。我们一直认为，Lua 程序导致宿主程序崩溃是不可接受的。Lua 应该是一门安全的语言。

为了克服指针失配的问题，Lua 2.1 引入了标记的概念（也是 Lua 3.0 里标记方法的概念来源）。一个标记只是一个任意的整数值，关联到一个 userdata 上。一个 userdata 的标记只能在创建时设置一次，假设每个 C 库都使用自己的独特的标记，C 代码可以检查 userdata 的标记，确保 userdata 就是所期待的类型。（一个库作者如何选择不会和别人重复的 tag 还是一个问题。这个问题只有在 Lua 3.0 才得到解决，在这个版本 Lua 提供了 lua_newtag 作标记管理）

Lua 2.1 更大的问题是 C 资源的管理。userdata 往往指向一块在 C 里动态分配的内存，当对应 userdata 在 Lua 里被回收，这块内存也需要释放掉。但是，userdata 是值，不是对象。因此，他们不会被回收（就像 number 不会被回收一样）。为了克服这个限制，一个典型的设计是利用 table 作为 C 结构在 Lua 中的代理，将实际的 userdata 存储在预定义的域里。当 table 被回收，它的析构函数就释放对应的 C 结构。

这种简单的解决方案会产生潜在问题。因为 userdata 是存在代理 table 的一个普通域里面，一个恶意的用户有可能在 Lua 里污染它。特别是，一个用户可以创建 userdata 的副本，并在 table 被回收后继续使用这个副本。这个时候，对应的 C 结构已经被销毁了，userdata 就变成悬空指针了，这会带来灾难性后果。为了改善对 userdata 生命周期的控制，Lua 3.0 将 userdata 从值类型改为对象类型，以便垃圾回收。用户可以使用 userdata 析构函数（垃圾回收的标记

方法）来释放对应的 C 结构。Lua 的垃圾回收器的正确性，保证了 userdata 不能在回收后再被使用。

但是，userdata 作为对象会带来一致性问题。给出一个 userdata，要获得对应的指针是很简单的，但是我们经常需要做相反的事：给出一个 C 指针，获得对应的 userdata。在 Lua 2 中，两个具有相同指针和相同标记的 userdata 就是相同的；相等性是基于它们的值。所以，给出指针和标记，我们就能获得 userdata。在 Lua 3，因为 userdata 变成了对象，所以相等性变成了一致性：两个 userdata 只有在两者同一的情况下才会相同（也就是说，这两个 userdata 是同一个）。每个 userdata 的创建都不一样。因此，一个指针和标记不足以获得对应的 userdata。

为了解决这一困难，并解决与 Lua 2 的不兼容问题，Lua 3 采用了下面的语意来将一个 userdata 压到栈上：如果 Lua 已经有一个 userdata，拥有给出的指针和标记，那么那个 userdata 就会被推到栈上；否则就创建一个新的 userdata 并压入栈。所以，C 代码要翻译一个 C 指针倒对应的 userdata 就变简单了。（实际上，C 代码可以沿用 Lua 2 的。）

但是，Lua 3 的行为有一个主要缺陷：它将两个基本操作合成了一个原语（lua_pushuserdata）：搜索 userdata 和创建 userdata。比如，没有办法在不创建 userdata 的情况下，检查一个给定的 C 指针有没有对应的 userdata。而且，也不能不管 C 指针，直接创建一个新的 userdata。如果 Lua 已经有对应值的 userdata，就不能再创建新的 userdata 了。

Lua 4 引入了新函数 lua_newuserdata 以缓解这个问题。不像 lua_pushuserdata，这个函数总是会创建一个新的 userdata。而且，更重要的

是，这些 userdata 可以存储任意 C 数据，而不仅仅是指针。用户可以告诉 lua_newuserdata 需要分配多少内存，lua_newuserdata 会返回一个指针，指向所分配的内存。通过让 Lua 给用户分配内存，很多常见的与 userdata 相关的任务都简化了。比如，C 代码不需要处理内存分配错误，因为 Lua 已经接手了。更重要的是，C 代码不需要处理内存释放：这种 userdata 使用的内存会在 userdata 被回收的时候，被 Lua 自动释放。

但是，Lua 4 依然没有提供一个漂亮的解决方案来应对搜索问题（即通过给定 C 指针寻找一个 userdata）。所以，它保留了 lua_pushuserdata 的行为，结果产生了一个混合的系统。只有在 Lua 5 的时候，我们才删除了 lua_pushuserdata，并去掉 userdata 创建和搜索的关联。实际上，Lua 5 将整个搜索设施都移除了。Lua 5 也引入了轻 userdata（light userdata），只用来存放 C 指针，跟 Lua 1 的普通 userdata 完全一致。应用程序可以使用弱表来关联 C 指针（light userdata 所包含的）和对应的“重”userdata。

正如其他 Lua 演进一样，Lua 5 的 userdata 比 Lua 4 的更加灵活；也更容易解释和实现。对于简单的用例，比如只需要存储一个 C 结构，Lua 5 的 userdata 就很易用。对于更为复杂的需求，比如需要将一个 C 指针映射到一个 Lua 的 userdata，Lua 5 提供了机制（light userdata 和弱表）给用户来实现适合宿主应用的策略。

6.11 反射性

Lua 从刚开始的版本就支持一些反射设施。提供支持的主要原因是，Lua 的建议用法是作为替换 SOL 的配置语言。正如第四章所述，我们的设想是，程序员若有需要，可以使用语言本身，写日常的类型检查。

比如，如果一个用户写了下面的代码：

```
T = @track{ y=9, x=10, id="1992-34" }
```

我们希望可以检查到，track 的确有一个域 y，这个域就是一个数字。我们也希望可以校验出 track 没有外来的域（比如检查出打字输入错误）。为了完成这两个任务，我们需要访问到 Lua 值的类型，以及遍历一个 table 的机制，并访问所有键值对。

Lua 1.0 用两个函数完成了所需的功能，这两个函数延续到今天：type 和 next。type 函数返回一个描述给定值的类型的字符串（比如 number, nil, table 之类）。next 函数接受一个 table 和一个键，并返回一个 table 里的“下一个”键（以任意顺序）。以 next(t, nil)调用会返回“第一个”键。通过 next 函数，我们可以遍历一个 table 并处理所有的键值对。比如，下面的代码可以打印 table t 的所有键值对：

```
k = next(t, nil)
while k do
    print(k, t[k])
    k = next(t,k)
end
```

所有这些函数都有一个简单的实现：type 检查给定值的内部标记，并返回对应的字符串；next 根据内部 table 的表示，在 table 里寻找给定的 key 并访问下一个 key。

在 Java 和 Smalltalk 这些语言里，反射需要具体的概念比如类、方法、实例变量。而且，这种具像化还需要新的概念比如元类（具体类的类）。Lua 不需要

这些东西。Lua 里，大部分类似 Java 反射包提供的设施都是免费的：类和模块都是 table，方法就是函数。所以，Lua 不需要任何特别的机制来具像化；它们就是朴素的程序值。类似的，Lua 不需要特殊机制来在运行时建立方法调用（因为函数是第一类值，而且 Lua 的参数传递机制天然支持可变数量参数传递），也不需要特殊的机制来访问一个全局变量或者根据一个给定的名字访问一个实例变量（因为它们都是普通的 table 域）。

7 回顾 这一节我们会给出一个 Lua 演进的简短评论，讨论其中做得好的，我们后悔的，以及我们不后悔，但是觉得可以以不同方式实现的。

最高明的决策，莫过于 Lua 1.0 的时候决定使用 table 作为 Lua 唯一的数据结构机制。table 被证明是强大而高效的。table 在语言里的核心地位和它的实现，是 Lua 的主要特性。我们顶住了用户要求添加其他数据结构的压力，主要是添加“真正的”数组和元祖，对此我们表现得十分顽固，但我们也提供了高效实现的 table 和灵活的设计。比如，我们可以实现一个 set（集合），只需要将它的元素作为 table 的 key 即可。全赖 Lua 的 table 接受任意值作为 key，这个特性才得以实现。

另一个则是我们对可移植性的坚持，最初是因为 Tecgraf 客户处在大量不同的平台上。这允许 Lua 运行在我们从未梦想得到支持的平台上。特别是，可移植性是 Lua 被游戏开发广泛采用的原因之一。受限的环境，比如游戏主机，一般不支持完整语意的全套标准 C 库。最终，通过减少 Lua 核心对标准 C 库的依赖，我们将 Lua 核心打造成只依赖独立的 ANSI C 实现标准。这一举措主要是为了嵌入的灵活性，但同样增加了可移植性。比如，从 Lua 3.1 开始，只要摆弄代码里少数几个宏，就可以让 Lua 使用一个应用程序自定义的内存分配器，

而不是直接使用 malloc 等内存分配函数。从 Lua 5.1 开始，内存分配器可以在创建 Lua 虚拟机的时候动态提供。

事后来看，我们认为在一个小型委员会的抚养下成长，对于 Lua 的演进是十分积极的。由一个庞大委员会设计的语言，倾向于变得十分复杂，从未完全实现过它们支持者的期望。大部分成功的语言都是被抚养长大，而非设计出来的。它们遵循着一个缓慢的自下而上的过程，以一门小语言开始，带着谦虚的目标。语言的成长是真实用户实际反馈的成果，这个过程里设计渐渐浮出水面，实际有效的新特性得到认同。这个描述正是对 Lua 演进的真实写照。我们倾听用户的反馈和建议，但我们只在三人同意的前提下才添加一个新的特性；否则，就留待未来决定。稍后再添加特性比移除特性要容易得多。这个开发过程对于保持语言的简洁性至关重要，而简洁性是我们最重要的资本。绝大部分 Lua 的其他特性——速度、体积小、可移植性——都是从简洁性而来。

从第一个版本开始，Lua 就有真实用户了，也即是除了我们自己以外的用户，他们根本不关心语言本身，只关心如何高效率的使用这门语言。用户总是为语言做出重要贡献，通过建议，抱怨，使用报告，和问题等形式。我们的小小委员会在管理这类反馈的时候再一次扮演了重要角色：它的结构给我们足够的惯性，去贴近用户、倾听用户，但不完全遵循它们的建议。

Lua 的最佳描述是：一个封闭开发的开源项目。这意味着，尽管源码对于审查和采用是免费可得的，Lua 还是没有以合作的方式开发。我们会采纳用户建议，但从未逐字逐句采纳他们的代码。我们总是尝试实现我们自己的设计。

Lua 演进的另一个不寻常的方面，是我们处理不兼容修改的方式。很长一段时间里，我们都认为简洁和优雅比兼容前一个版本更为重要。当一个老的特性被新特性取代，我们就删掉老的特性。我们经常（但非总是）提供一些兼容性工具，比如一个兼容的库，一个转换脚本，或者编译时的选项来保留老特性。在这些情形下，用户需要在转移到新版本的时候采取一定措施。

我们并未真正后悔这种演进方式。但是，我们最终变得更为保守了。不仅因为我们的用户和代码基变得比以前更大，还因为我们觉得作为一门语言，Lua 变得更成熟了。

我们应该从一开始就引入布尔类型，但我们希望从最为简洁的语言开始。没有一开始就引入布尔类型，带来了一些不幸的副作用。其中一个是我们现在有两

个 false 值了：nil 和 false。另一个是，Lua 函数用于警告调用者错误发生的常见用法，是返回 nil 值接着一段错误信息。如果用 false 而非 nil，就更适合这个场景了，nil 可保留作为缺少值的含义。

在算术运算时，自动将字符串转化为数字，是我们从 Awk 里参考的，其实可以忽略。（在字符串操作中自动将数字转换为字符串则很方便，也更少问题。）

除了我们的"机制，而非策略"的规则外——我们觉得在 Lua 演进里这个规则很有价值——我们应该为模块和包提供一个准确的指导方针。缺乏通用的构建模块和安装包的策略，限制了不同工作组共享代码，不利于发展整个社区的代码基。Lua 5.1 提供了模块和包的指导方针，我们希望能够补救现存的局面。

正如 6.4 节提到的，Lua 3.0 引入了条件编译的支持，主要是为了提供一种方式来关闭代码。我们收到很多请求，要求加强 Lua 的条件编译，这些请求甚至来自于不使用条件编译的用户！目前最大的需求是一个完整的宏处理器，就像 C 的预处理器一样。提供这样一个宏处理器，跟我们提供扩展机制的哲学是一致的。但是，我们希望宏也能用 Lua 编写，而不是其他特殊语言。我们不希望直接在词法器上添加宏设施，以免其变得臃肿不堪，拖慢编译过程。更重要的是，那个时候 Lua 语法分析器还不是完全可重入的，所以没有办法在词法器里调用 Lua。（这个限制在 Lua 5.1 去掉了）所以邮件列表里和 Lua 开发者里有无穷无尽的讨论。我们依然希望能够为 Lua 提供一个宏系统：它会给 Lua 提供更灵活的语法，支持更灵活的语义。

8 结论

Lua 在很多大型公司里大获成功，比如 Adobe，Bombardier，Disney，Electronic Arts，Intel，LucasArts，Microsoft，Nasa，Olivetti 和 Philips。这些公司很多产品都直接在商用产品里嵌入 Lua，并向最终用户暴露 Lua 脚本。

Lua 在游戏领域特别成功。最近流行一个说法：“Lua 正迅速变成游戏脚本的事实标准”【37】。两个在 gamedev.net 进行的非正式的投票【5，6】，分别在 2003 年 9 月和 2006 年 6 月结束，结果显示 Lua 正是最炙手可热的游戏开发脚本语言。GDC 专门针对游戏开发中的 Lua 的圆桌会议举行了两次，分别是 2004 年和 2006 年。很多著名游戏都使用 Lua：Baldur’s Gate, Escape from Monkey Island, FarCry, Grim Fandango, Homeworld 2, Illarion, Impossible Creatures, Psychonauts, The Sims, World of Warcraft。现在有两本使用 Lua 进行游戏开发的书【42，25】，而且很多其他游戏开发的书都会开辟专门的章节来讲述 Lua【23，44，41，24】。

Lua 在游戏界的广泛使用是我们的惊喜收获。我们并没有为 Lua 考虑过游戏开发这个目标。（Tecgraf 主要考虑的是科学软件）事后看来，这种成功是可以理解的，因为所有让 Lua 变得与众不同的特质，对游戏开发都十分重要：

可移植性：很多游戏跑在非传统的平台上，比如游戏机上，这些平台需要特殊的开发工具。构建 Lua 只需要一个 ANSI C 编译器。

容易嵌入：游戏是性能敏感型应用。他们既需要性能，来应付图形和模拟，还需要灵活性，以应对创意的部分。很多游戏都用至少两种语言开发，这并非偶然，一种语言用于脚本编写，另一种用于构建引擎。在这个框架内，轻松集成 Lua 到其他语言（在游戏界主要是 C++）是一个巨大的优势。

简洁：很多游戏设计师，脚本编写者，关卡作者都不是专业的程序员。对他们来说，一门语法简洁，语义清晰的语言尤其重要。

高效率，小体积：游戏是性能敏感应用程序；分配给脚本运行的事件通常都很小。Lua 是最快的脚本语言之一【1】。游戏机是受限环境。脚本解释器应该节约资源。Lua 核心的体积只有 100K。

对代码的控制：不像其他大部分的软件企业，游戏产品很少演化。很多时候，一个游戏发布之后，就没有更新或者新版本了，只有新游戏。所以，冒险在游戏里使用一门新的脚本语言更容易。一门脚本语言是否会演进，如何演进，对于游戏开发者来说不是一个很关键的点。他们所需要的就只是在游戏里使用的版本。因为他们对 Lua 源码有完全访问的权限，他们可以选择永远保持在同一个 Lua 版本。

自由的协议：很多商业游戏是不开源的。很多游戏公司甚至拒绝使用任何开源代码。因为竞争非常激烈，所以游戏公司倾向于保守技术上的秘密。对他们来说，像 Lua 一样的自由协议是非常方便的。

协程：如果脚本语言支持多任务，编写脚本游戏会变的更简单，因为一个角色或者活动可以被暂停，并稍后恢复。Lua 通过协程支持协作式多任务【14】。

过程式数据文件：Lua 的原始设计目标是提供威力强大的数据描述设施，这允许游戏使用 Lua 组织数据文件，替换特殊格式的文本数据文件，带来诸多益处，尤其是同源性和表达力。

致谢

Lua 是在大家的帮助下成长起来的。Tecgraf 的所有人都以不同形式为 Lua 贡献——使用这门语言，讨论它，向 Tecgraf 以外传播它。特别鸣谢 Marcelo Gattass，作为 Tecgraf 的主管，总是鼓励我们并给予我们完全的自由，去演进 Lua 这门语言和它的实现。Lua 不再是 Tecgraf 的一个产品，但依然在 PUC-Rio 内开发，开发组属于 2004 年 5 月建成的 LabLua 实验室。

没有用户的话，Lua 就只是另外一门语言，注定被遗忘。用户和他们的使用是一门语言的试金石。特别鸣谢我们邮件列表的成员，感谢他们的建议、抱怨和耐心。邮件列表相对较小，但非常友善，并有一些 Lua 开发团队以外的技术水平很高的人士参与，他们慷慨的向整个社区分享自己的专业知识，让人获益匪浅。

我们感谢 Norman Ramsey 的牵头，Norman Ramsey 建议我们在 HOPL III 上发表一篇有关 Lua 的论文，并帮助我们联系到会议管理团队。我们感谢 Julia Lawall，帮助我们通读数遍这篇文章的草稿，并代表 HOPL III 委员会仔细处理本文。我们感谢 Norman Ramsey，Julia Lawall，Brent Hailpern， Barbara Ryder，以及未具名的编辑，感谢他们详细的评论和富有建设性的建议。

我们也向 Andre ´Carregal, Anna Hester, Bret Mogilefsky, Bret Victor, Daniel Collins, David Burgess, Diego Nehab, Eric Raible, Erik Hougaard, Gavin Wraith, John Belmonte, Mark Hamburg, Peter Sommerfeld, Reuben Thomas, Stephan Herrmann, Steve Dekorte, Taj Khattra 和 Thatcher Ulrich 致谢，感谢他们对 Lua 发展历程的补充，并对相关文本进行润色。Katrina Avery 做了很好的拷贝-编辑工作。

最后，我们感谢 PUC-Rio，IMPA 和 CNPq 对我们 Lua 工作一如既往的支持，以及 FINEP 和微软研究院对数个 Lua 相关项目的支持。

相关文献：

[1] The computer language shootout benchmarks. http://shootout.alioth.debian.org/.
[2] Lua projects. http://www.lua.org/uses.html.
[3] The MIT license. http://www.opensource.org/licenses/mit-license.html.
[4] Timeline of programming languages. http://en.wikipedia.org/wiki/Timeline of programming languages.
[5] Which language do you use for scripting in your game engine? http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163, Sept. 2003.
[6] Which is your favorite embeddable scripting language? http://www.gamedev.net/gdpolls/viewpoll.asp?ID=788, June 2006.
[7] K. Beck. Extreme Programming Explained: Embrace

Change. Addison-Wesley, 2000.
[8] G. Bell, R. Carey, and C. Marrin. The Virtual Reality Modeling Language Specification—Version 2.0. http://www.vrml.org/VRML2.0/FINAL/, Aug. 1996. (ISO/IEC CD 14772).
[9] J. Bentley. Programming pearls: associative arrays. Communications of the ACM, 28(6):570–576, 1985.
[10] J. Bentley. Programming pearls: little languages. Communications of the ACM, 29(8):711–721, 1986.
[11] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In SIGPLAN Conference on Programming Language Design and Implementation, pages 99–107, 1996.
[12] W. Celes, L. H. de Figueiredo, and M. Gattass. EDG: uma ferramenta para criac¸ ˜ao de interfaces gr´aficas interativas. In Proceedings of SIBGRAPI ’95 (Brazilian Symposium on Computer Graphics and Image Processing), pages 241–248, 1995.
[13] B. Davis, A. Beatty, K. Casey, D. Gregg, and J.Waldron. The case for virtual register machines. In Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators, pages 41–49. ACM Press, 2003.
[14] L. H. de Figueiredo, W. Celes, and R. Ierusalimschy. Programming advanced control mechanisms with Lua coroutines. In Game Programming Gems 6, pages 357–369. Charles River Media, 2006.
[15] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. The design and implementation of a language for extending applications. In Proceedings of XXI SEMISH (Brazilian Seminar on Software and Hardware), pages 273–284, 1994.
[16] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. Dr. Dobb’s Journal, 21(12):26–33, Dec. 1996.
[17] L. H. de Figueiredo, C. S. Souza, M. Gattass, and L. C. G. Coelho. Gerac¸ ˜ao de interfaces para captura de dados sobre desenhos. In Proceedings of SIBGRAPI ’92 (Brazilian Symposium on Computer Graphics and Image Processing), pages 169–175, 1992.
[18] A. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. Journal of Universal Computer Science, 10(7):910–925, 2004.
[19] A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. MCC 15/04, PUC-Rio, 2004.
[20] R. K. Dybvig. Three Implementation Models for Scheme.

PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1987. Technical Report #87-011.
[21] M. Feeley and G. Lapalme. Closure generation based on viewing LAMBDA as EPSILON plus COMPILE. Journal of Computer Languages, 17(4):251–267, 1992.
[22] T. G. Gorham and R. Ierusalimschy. Um sistema de depurac¸ ˜ao reflexivo para uma linguagem de extens˜ao. In Anais do I Simp´osio Brasileiro de Linguagens de Programac¸ ˜ao, pages 103–114, 1996.
[23] T. Gutschmidt. Game Programming with Python, Lua, and Ruby. Premier Press, 2003.
[24] M. Harmon. Building Lua into games. In Game Programming Gems 5, pages 115–128. Charles River Media, 2005.
[25] J. Heiss. Lua Scripting f¨ur Spieleprogrammierer. Hit the Ground with Lua. Stefan Zerbst, Dec. 2005.
[26] A. Hester, R. Borges, and R. Ierusalimschy. Building flexible and extensible web applications with Lua. Journal of Universal Computer Science, 4(9):748–762, 1998.
[27] R. Ierusalimschy. Programming in Lua. Lua.org, 2003.
[28] R. Ierusalimschy. Programming in Lua. Lua.org, 2nd edition, 2006.
[29] R. Ierusalimschy, W. Celes, L. H. de Figueiredo, and R. de Souza. Lua: uma linguagem para customizac¸ ˜ao de aplicac¸ ˜oes. In VII Simp´osio Brasileiro de Engenharia de Software – Caderno de Ferramentas, page 55, 1993.
[30] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua: an extensible extension language. Software: Practice & Experience, 26(6):635–652, 1996.
[31] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. Journal of Universal Computer Science, 11(7):1159–1176, 2005.
[32] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua 5.1 Reference Manual. Lua.org, 2006.
[33] K. Jung and A. Brown. Beginning Lua Programming. Wrox, 2007.
[34] L. Lamport. LATEX: A Document Preparation System. Addison-Wesley, 1986.
[35] M. J. Lima and R. Ierusalimschy. Continuac¸ ˜oes em Lua. In VI Simp´osio Brasileiro de Linguagens de Programac¸ ˜ao, pages 218–232, June 2002.
[36] D. McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped LISP. In ACM conference on LISP and functional programming, pages 154–

162, 1980.
[37] I. Millington. Artificial Intelligence for Games. Morgan Kaufmann, 2006.
[38] B. Mogilefsky. Lua in Grim Fandango. http://www.grimfandango.net/?page=articles&pagenumber=2, May 1999.
[39] Open Software Foundation. OSF/Motif Programmer's Guide. Prentice-Hall, Inc., 1991.
[40] J. Ousterhout. Tcl: an embeddable command language. In Proc. of the Winter 1990 USENIX Technical Conference. USENIX Association, 1990.
[41] D. Sanchez-Crespo. Core Techniques and Algorithms in Game Programming. New Riders Games, 2003.
[42] P. Schuytema and M. Manyen. Game Development with Lua. Delmar Thomson Learning, 2005.
[43] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. SIGPLAN Notices, 35(6):26–36, 2000.
[44] A. Varanese. Game Scripting Mastery. Premier Press, 2002.