« 共享 lua state 中的数据 | 返回首页 | 有关 Forth »

采访 Lua 发明人的一篇文章

《Masterminds of Programming: Conversations with the Creators of Major Programming Languages》是本相当不错的书。博文翻译出版了这本书，中文名叫做《编程之魂》。

书是好书，可惜翻译这本书需要对各种语言的深入研究，看起来译者有点力不从心。出版社打算重新做这本书。受编辑所托，我校对了其中第七章：有关 Lua 的一段。原文读下来拍案叫好。可惜译文许多地方看起来有些词不达意。许多在口语化交流中提到的术语被忽略了做了错误的翻译。有些部分应该是对 lua 理解不够而没能表达清楚。

仔细校对了两段后，我干脆放弃原译本，自己动手翻译了一份（保留了不到 1/4 原来的译文）。虽然个人能力有限，但也算是每句话自己都看明白了再译的。虽说有些地方没有直译，但也算没有夹带私货。

这里贴出一段，希望大家阅读愉快。

---

7. Lua

Lua 是一门非常之小，但五脏俱全的动态语言。它由 Roberto Ierusalimschy、Luiz Henrique de Figueiredo 和 Waldemar Celes在1993年创建。Lua 拥有一组精简的强大特性，以及容易使用的 C API ，这使得它易于嵌入与扩展来表达特定领域的概念。Lua在专有软件界声名显赫。例如，在诸多游戏中，比如 Blizzard（暴雪）公司的《魔兽世界》和 Crytek GmbH 公司的《孤岛危机》，还有 Adobe 的 Photoshop Lightroom ，都使用它来作脚本 和 UI 方面的工作。它继承了 Lisp 和 Scheme，或许还有 AWK 的血脉 ； 在设计上类似于 JavaScript、Icon 和 Tcl。

7.1 脚本的威力

你是如何定义 Lua 的？

LHF：一种可嵌入，轻量，快速，功能强大的脚本语言。

Roberto：不幸的是，越来越多的人们使用“脚本语言”作为“动态语言”的代名词。现在，甚至是 Erlang 或者 Scheme 都被称为脚本语言。这非常糟糕，因为我们无法精确的描述一类特定的动态语言。在最初的含义解释中，Lua 是一种脚本语言，这种语言通常用来控制其它语言编写的其他组件。

人们在使用Lua设计软件时，应该注意些什么呢？

Luiz：我想应该是用 Lua 的方式来做事。不建议去模拟出所有你在其它语言中用到的东西。你应该真的去用这个语言提供的特性，我想对于使用任何一门语言都是这样的。就 Lua 来讲，语言的特性主要指用 table 表示所有的东西，用 metamethod 做出优雅的解决方案。还有 coroutine 。

Lua 的用户应该是哪些人呢？

Roberto ：我认为大多数没有脚本功能的应用程序都能从 Lua 中受益。

Luiz：问题在于，大多数设计者很长时间都不会意识到有这种需求。当已经有了诸多用 C 或 C++ 编写的代码，为时已晚。应用程序设计者应该从一开始就考虑脚本。这会给它们带来更多的灵活性。而且这样做还可以更好的把握性能问题。因为这样做以后，会迫使他们去考虑程序中到底哪里是性能关键，而哪些地方无伤大雅。而这些性能不太重要之处，就交给脚本去处理，开发周期短，速度快。

从安全性的观点来看，Lua 能为程序员提供些什么呢？

Roberto：Lua 解释器的核心部分被构建为一个 “独立的应用程序(freestanding application)”。这个术语来自 ISO C，大意是说，这部分不使用任何跟外部环境有关的东西（不依赖 stdio、malloc 等）。所有那些功能都由扩展库来提供。使用这种体系结构，很容易让程序限制对外部资源的访问。具体来说，我们可以在 Lua 自身的内部创建出一个沙盒，把如何我们认为危险的操作从沙盒的外部环境中剔除。（比如打开文件等）

Luiz：Lua 还提供了用户自定义的调试钩子，用它可以监视 Lua 程序的执行。这样，在 lua 中运行时间过长或是使用了过多内存的时候，我们可以从外部中断它的执行。

Lua 有什么局限性？

Roberto：我认为 Lua 的主要局限是所有动态语言共有的。首先，即使是利用最先进的 JIT 技术（Lua 的 JIT 是所有动态语言 JIT 中最好的之一）也达不到优秀静态语言的性能。其次，一些复杂的程序从静态分析中受益匪浅（主要是静态类型）。

是什么促使你决定使用垃圾收集器？

Roberto：Lua 从第一天开始，就一直使用垃圾收集器。我想说，对于一种解释型语言来讲，垃圾收集器可以比引用计数更加紧凑和健壮，更不用说它没有把垃圾丢得到处都是。考虑到解释型语言通常已经有自描述数据（通过给值加上标签之类的东西），一个简单的标记清除（mark-and-sweep）收集器实现起来极其简单，而且几乎对解释器其余的部分不会产生什么影响。

对于无类型语言(untyped language)，引用计数会很重量。没有静态类型，每次赋值都可能会改变计数，对变量的新值和旧值都需要进行动态检查。后来尝试过在 Lua 中引入引用计数，并没有提高性能。

你对 Lua 处理数字的方式满意吗？

Roberto：从我的经验来看，计算机中的数字老是会给我们带来一些意外（因为它们也来至于计算机之外!）。至于说 Lua 使用 double 作为唯一的数字类型，我认为这是一种合理的折衷方案。我们已经考虑了很多其他可选方案，不过对于 Lua 来说，这些方案要么太慢，要么太复杂，要么太耗内存。对于嵌入式系统，甚至使用 double 也不是一种合理的选择，因此，我们可以使用一个备选的数值类型，比如说 long ，来编译解释器。

你为什么选择 table 作为 Lua 中的统一数据结构？

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，有一些东西吸引了我的兴趣。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。

程序员可以从 Lua 中的 first-class 函数中获得怎样的价值？

Roberto：50多年来，虽然名称各异：从子程序到方法，“函数” 已经成为编程语言的主要部分，因此，对函数的良好支持为所有语言必备。Lua 支持程序员使用函数式编程领域中的一些功能强大的技术，比如，把数据表示成函数。例如，一种形状可能用函数来表示，给定 x 和 y ，可以判断这个点是否在这个形状内。这种表示方式可以用于一些操作，比如联合和交集等。

你为什么要实现闭包（closure）？

Roberto：闭包自始至终我们都想在 Lua 中实现：它简单、灵活、功能强大。从第一版开始，Lua 就把函数做为一等值（first-class value）对待。这被证明非常有用，即使是对于没有函数式编程的“常规的”程序员来说也是一样。而不支持闭包的函数，其实用价值就会大打折扣。顺便说一句，闭包这个术语来源于一种实现技术，而不是指它本身的特性。从特性描述上来说，闭包相当于“带词法作用域的一等函数”，当然用闭包这个术语更为简短。

你打算如何处理并发问题？

Roberto：我们不信任基于抢占式内存共享的多线程技术。在 HOPL 论文中，我们写道：“我们仍然认为，如果在连 a=a+1 都没有确定结果的语言中，无人可以写出正确的程序。” 我们可以通过去掉抢占式这一点，或是不共享内存，就可以回避这个问题。而 Lua ，提供用这两种方式解决问题的支持。

使用协程（coroutine），我们可以共享内存，但不是抢占式的。不过这个技术利用不到多核机器。但在这类机器上，使用多“进程”就能极大的发挥其性能。这个我提到的“进程”是指在 C 里的一个线程，这个线程维护自己独立的 Lua 状态机。这样，在 Lua 层面上，就没有内存共享使用。在《Lua 程序设计第二版》[Lua.org] 中，我给出了这种方式的一个原型。最近我们已经看到有些库支持了这种方式（比如 Lua Lanes 以及 luaproc）。

你没有支持并发，但你为多任务实现了一个有趣的解决方案：非对称式协程。它们如何工作的？

Roberto：我有一些 Modula 2 语言的经验（我的妻子在她的硕士论文工作中为 M-code 编写了一个完整的解释器），使用协程作为协作式并发以及别的控制结构的基础设置是我一直偏爱的方法。然而，Modula 2 中那种对称式协程，在 Lua 中行不通。

Luiz：在我们的 HOPL 论文中，对那些设计决策全部做了极为详细的解释说明。

Roberto：我们最终选择了非对称式模型。它的基本思想非常简单。通过显式调用 coroutine.create 函数来创建一个协程，把一个函数作为协程主体来执行。当我们启动 (resume) 协程时，它开始运行函数体并且直到结束或者让出控制权

(yield) ；一个协程只有通过显式调用 yield 函数才会中断。以后，我们可以 resume 它，它将会从它停止的地方继续执行。

它的基本思想非常类似于 Python 的生成器，但有一个关键区别：Lua协程可以在嵌套调用中 yield，而在 Python 中，生成器只能从它的主函数中 yield。在实现上，这意味着每个协程像线程一样必须有独立堆栈。和“平坦”的生成器相比，“带堆栈”的协程发挥了不可思议的强大威力。例如，我们可以在它们的基础上实现一次性延续点 (one-shot continuations)。

7.2 经验

对于你做的这些，你如何定义成功？

Luiz：一种语言的成功，取决于使用该语言的程序员数量以及使用它的应用程序的成功。其实，到底有多少人在使用 Lua 编程，我们并没有确切的答案，不过毫无疑问的是，有很多成功使用 Lua 的应用程序，其中包括一些非常成功的游戏。同样地，使用 Lua 的应用程序的范围，从桌面图像处理到嵌入式机器人控制。这表明 Lua 具有一个非常明确的小众市场。最后，Lua 是唯一一种由发展中国家创建并在全球获得广泛应用的语言。它也是 ACM HOPL 唯一重点推介的语言。

Roberto：这很难定义。我曾经在多个领域工作过，在每个领域我从不同的方式在感受了成功。总之，我想说这些的共通之处在于：“被人知晓”。被认可，被公认，被人们推荐，这些都让人非常开心。

对于这门语言，你有什么遗憾吗？

Luiz：我确实没有任何遗憾。不过，事后回想起来，如果我们当初知道我们现在正在做的事情该怎么做的话，这些事情本可以早点完成！

Roberto：我不确信我有什么具体的遗憾，不过语言设计会牵涉到很多困难的决策。对我来说，最困难的决策是在易用性方面。Lua 的目标之一是让非专业程序员易于使用。我没有契合这种定位。因此，当我把自己当作用户，从这个视野来看，有关 Lua 语言的某些决策并非最佳。Lua 的语法就是一个典型的例子：虽然 Lua 的很多应用都得益于其冗长的语法，不过，就我自己的口味而言，我更偏爱紧凑的符号。

你在设计或实现时犯过错吗？

Luiz：我认为我们在设计或实现 Lua 时，并没有犯什么大错。我们学着如何发展一门语言。这绝不仅仅是定义它的语法和语义并将其实现。还有许多重要的社会问题，比如说创建并支持一个社区。这需要通过多种途径，编撰手册、写书、维护网站、邮件列表以及聊天室等。毫无疑问，我们认识到了支持一个社区的价值，明白了做这些工作需要极大的投入，并不亚于在设计和编码工作中的投入。

Roberto：我们很幸运，没有犯什么大错。我们在这个过程中还是出了许多小问题。作为 Lua 演化发展的一部分，我们有机会修正它们。当然，版本间的不兼容问题会让一些用户感到烦恼。好在 Lua 现在已经非常稳定了。

对于成为一名优秀的程序员，你有什么建议？

Luiz：永远不要害怕重新开始，这当然是说到容易做到难。永远不要低估需要注意的细节。你认为未来可能会用到的功能，就不要马上添加了：现在增加这个功能只会让你日后真的需要这个东西时，那些更好的特性很难加入。最后，永远追求更为简洁的解决方案。诚如爱因斯坦所言：尽量简洁，然过犹不及（As simple as possible, but not simpler.）。

Roberto：学习新的编程语言，不过一定要读好书！Haskell 是所有程序员都应该学会的一种语言。学习计算机科学：新算法、新形式体系（如果你还不了解，可以看一下 Lambda 演算，或是 pi 演算，CSP 等等）持续改进你的代码。

计算机科学的最大问题是什么？我们又如何教授呢？

Roberto：我想还没有什么能像“计算机科学”那样表达一种广为人知的知识集。并不是说计算机科学不是科学，而是说太难定义什么是计算机科学，什么不是（以及什么重要什么不重要）。计算机科学界的很多人都没有一个正规的计算机科学背景。

Luiz：我把自己当成是一名对计算机在数学中扮演什么角色感兴趣的数学家。当然，我非常喜欢计算机。:)

Roberto：即使是那些有正规计算机科学背景的人，也没有达成共识，我们缺乏一个交流的共同基础。很多人认为是 Java 创建了监视器、虚拟机以及接口（相对于类）等。

是不是有很多计算机科学学科仅仅只是一种职业训练？

Roberto：是的。而且，很多程序员甚至连计算机科学的学位都没有。

Luiz：我并不这么认为，但我不是作为一名程序员被雇用的。从另外一方面来说，我认为，要求程序员有计算机科学学位或是诸如此类的认证是错误的。计算机科学学位并不代表很好的编程能力。很多优秀的程序员也没有计算机科学学位（或许这只在我开始编程时成立；现在我可能是太老了）。我的观点是，一个人拥有计算机科学学位并不能保证他程序写得好。

Roberto：要求所有的专业人士都拥有学位是不对的。但我的意思是这个领域的“文化”太薄弱。几乎没什么东西需要人们必须知道。当然，雇主可以制定自己的要求，但不应该对学位有严格规定。

数学在计算机科学，特别是编程方面，起到一个什么作用？

Luiz：好吧，我是一位数学家。对我来说，数学无处不在。我之所以被编程所吸引，很可能是因为它具有数学的特性：精确、抽象和优雅。编写一个程序有如对一个复杂定理的证明，你可以持续不断地精炼和改进，而且它还能干点实际的事情！

当然，我在编程时根本没想这些，不过我认为，数学的学习对于编程是非常重要的。它有助于带你进入一种特定的心境当中。如果你习惯以抽象事物的自身法则去思考问题，编程就变得更简单。

Roberto：按照 Christos H. Papadimitriou 的说法，“计算机科学是新的数学”。一名程序员如果没有数学功底，就很难有大的作为。从更广的视野来看，数学和编程都具有一些共同的思想原则：抽象。它们都使用同一个关键工具：形式逻辑。优秀的程序员任何时候都在使用“数学”，利用它来确立 code invariants 以及接口模型等。

很多编程语言都是数学家创建的——或许这就是编程困难的原因所在!

Roberto：我会把这个问题留给我们的数学家。

Luiz：好的，此前我已经说过，编程绝对具有数学品质：精确、抽象、优雅。对我来说，设计编程语言就像是构建一种数学理论：你提供了功能强大的工具，其他人可以使用它来做很出色的工作。我一直被那些规模小而功能强的编程语言所吸引。强大的原语和结构之美如同强大的定义和基本理论之美。

你是如何区分出优秀的程序员的呢？

Luiz：你也知道。如今，糟糕的程序员更容易识别——不是因为他们的程序很糟糕（尽管那些程序通常非常复杂又混乱不堪），而是因为你可以感觉到，编程对他们来说并不愉悦，好像他们写的程序对他们自己来说是一个神秘事物，一种负担。

调试技能如何教授？

Luiz：我认为调试无法教授，至少不能正式地教授。不过当你跟别人，一个或许比你经验更丰富的人，一起调试的时候，你可以通过具体案例来学习。你可以从他们那里学习调试策略：如何去缩小问题范围，如何去做出预测和评估结果，判断哪些是没有用的，只是些噪音而已。

Roberto：调试本质上是在解决问题。它是一个需要来调动你已学会使用的一切工具的活动。当然存在一些实用的技巧（例如，如有可能，尽量不用调试器，在用 C 这样的底层语言编程时，使用内存检查器），不过，这些技巧只是调试的一小部分。必须像学习编程那样学习调试。

你如何测试和调试你的代码呢？

Luiz：我主要是一块一块的构建，分块测试。我很少使用调试器。即使用调试器，也只是调试 C 代码。我从不用调试器调试 Lua 代码。对于 Lua 来说，在适当的位置放几条打印语句通常就可以胜任了。

Roberto：我差不多也是这样。当我使用调试器时，通常只是用来查找代码在哪里崩溃了。对于 C 代码，有个像 Valgrind 或者 Purify 这样的工具是必要的。

源代码中的注释起到什么作用？

Roberto：用处不大。我通常认为，如果有什么需要注释的，那只是因为程序没写好。对于我来说，一条注释更像是打了个便签，它在说“以后记得重写这段代码”。我认为清晰的代码要比带注释的代码可读性更强。

Luiz：我同意。我一直坚持：注释应该用来表达代码不能清晰表达的东西。

一个项目应该如何文档化呢？

Roberto：强制执行。没有什么工具可以代替一份井井有条、深思熟虑的文档。

Luiz：但是，为一个项目的发展历程写出好的文档，唯一的可能就是从一开始就把这一点放在心上。Lua 并没有这样做；我们从来没想到 Lua 能发展这么快，并在今天获得这么广泛的应用。我们在撰写 HOPL 论文的日子里（这花了将

近两年时间！），我们发现已经很难记起当时是怎么做出一些设计决策的了。从另外一个角度来说，如果早期我们要求会议都有正式的会议记录，可能就会失去一些自发性，并错失一些乐趣。

在代码库的发展历程中，你需要权衡哪些因素？

Luiz：我会说“实现的简单性”。这样做的话，速度和正确性随之而来。同时，灵活性也是重点，这样，如果需要，你可以换一个实现方式。

可用的硬件资源如何影响程序员的心态？

Luiz：我是个老家伙了。我是在一台 IBM 370 上学习的编程。要花上几个小时来给卡片穿孔、提交给队列再等到打印输出。我见过各种各样的慢机器。我认为程序员应该体验一下这些机器，因为并不是世界上人人都有最快的机器。编写给大众使用的应用程序的人应该在慢机子上试一下，这样才可以获得更广泛的用户体验。当然，仅可能用最好的机器来开发：把大量时间花在等待完成编译上可一点也不有趣。在现在的全球因特网中，Web 开发者应该尝试慢速连接，而不是他们工作机上的超快连接速度。以平均水平的平台为目标，会让你的产品速度更快、更简单，而且更好。

就Lua来说，“硬件”是指 C 编译器。我们在实现 Lua 的过程中学会的一点就是：以可移植性为目标确实值得。几乎从一开始，我们就是用非常严格的ANSI/ISO C (C89) 来实现 Lua 的。这样一来，Lua 就可以在专用硬件上运行，比如机器人、打印机固件和网络路由器等，这些没有一个是我们当初的实际目标平台。

Roberto：你应该始终认为硬件资源有限，这是一条金科玉律。它们当然总是有限的。“自然厌恶真空”；任何程序都有扩展的趋势，直到它用完了所有的可用资源。此外，随着确定平台上的资源越来越便宜的同时，又会出现一些有严格限制的新平台。微型计算机是这样；移动电话是这样；一切都是这样。如果你想做成市场第一，你最好要时刻关注你的程序需要什么资源。

对于现在或者不久的将来开发计算机系统的人，你在发明、开发和完成你的语言方面，有什么经验可以说的吗？

Luiz：我认为，程序员应该始终记住：并非所有的应用程序都是运行在功能强大的台式机或者笔记本电脑上的。很多应用程序要运行在受限的设备上，比如说手机，甚至是更小的设备等。设计和实现软件工具的人们应该特别关注这个问题，因为没有人会告诉你，你的工具会在什么地方如何使用。因此，就应该为使用最小的资源而设计。你可能会惊奇地发现：很多环境使用了你的工具，而你并没有把这些环境作为主要的应用目标，你甚至都不知道它们的存在。Lua 就碰到过这种事！而且这很自然；我们内部有一个笑话，这其实不是一个真正的笑话：我们讨论在 Lua 中的一个特性的细节时，我们问自己，“好的，不过它会不会在微波炉上运行呢？”

7.3 语言设计

Lua 易于嵌入，而且要求的资源也非常少。你是如何设计的，使得它适应硬件、内存和软件资源都很有限的情况？

Roberto：开始时，我们并没有把这些目标搞得很明确。我们只是为了完成项目才不得已而为之。随着我们的发展，这些目标对我们来说变得更为清晰。现在，我想各方面的主要问题都始终是经济问题。例如，无论什么时候，有人建议一些新的特性，第一个问题就是需要多大的成本。

你有没有因为特性成本太高而拒绝添加它们呢？

Roberto：几乎所有的特性，相对于它们能带给语言的东西来说，都“成本太高”。举一个例子，甚至一个简单的 continue 语句都不符合我们的标准。

添加一个特性需要带来多大的收益才是值得的呢？

Roberto：没有固定的规范，不过看该特性是否能让我们感到“惊喜”是条好的判断标准；也就是说，不仅仅满足其初始其初始动机。这让我想起了另一条经验法则：多少用户会从该特性中受益。某些特性只对一小部分用户是有用的，而其他特性对于几乎所有人都是有用的。

你有例子说明一条新特性对很多人都有用吗？

Roberto：for 循环。我们甚至反对过这个特性，不过当它出现时，它改变了书中所有的例子! 弱表也是出奇地有用。使用它们的人并不多，不过他们应该试试。

在 1.0 版本之后的多年里，你都没有把 for 循环加上。是什么驱使你不加它？而又是什么使你最终加入了它？

Roberto：我们曾无法找到一种让循环通用而简洁的格式，以至于我们一直不肯加入它。当我们发现可以使用一个生成器函数这样一个不错的形式后，我们就把 for 循环加上了。实际上，闭包是使生成器简单通用的要素。因为把生成器函数做成闭包，可以在循环过程中保留其内部状态。

更新代码来获取新特性的优势，重新得到更好的编程实践经验，这些会引起大块费用吗？

Roberto：新特性不是必须使用的。

那么人们会选择一个 Lua 的版本一直用到整个项目的生命期结束，从不升级吗？

Roberto：我认为，在游戏领域大多数人确实是这样做的。而在其他领域，我认为有一些项目不断更新他们所用的 Lua 版本。不过有个反例，魔兽世界从 Lua 5.0 更新到了 5.1 ！请留意 Lua 现在要比早年的时候稳定多了。

你们在开发过程中是如何分工的，特别是在编写代码方面？

Luiz：Lua 第一版是由 Waldemar 在 1993 年编码的。自 1995 年左右以来，Roberto 编写和维护了主要代码。我负责一小部分：字节码 dump/undump 模块和独立编译器 luac 。我们一直在修改代码，并通过电子邮件向其他人发送代码修改建议，而且，我们就新特性及其实现开了很长时间的会议。

你从用户那里得到了很多有关语言和实现的反馈吗？对于在语言中加入用户反馈及其修改，你有一个正式的机制吗？

Roberto：我们开玩笑说：你要是忘了什么，那它肯定不重要。Lua 讨论列表非常活跃，不过一些人将开放软件和社区项目等同视之。有一次，我向 Lua 列表发送了以下消息，总结了我们的方法：

Lua 是一款开放软件，不过它从未进行过开放式开发。这并不意味着我们没有听取其他人的意见。实际上，我们几乎阅读了邮件列表中的每一条消息。Lua 里面的若干重要特性就起源或发展至外部的贡献（元表、协程，以及闭包的实现，这里仅举出几个重要的名字），不过，一切都要由我们来最终决定。我们这么做并非觉得我们的判断要比其他人的更好。而仅仅是因为我们想让 Lua 成为我们想要的语言，而不是世界上最流行的语言。

由于采用了这种开发风格，我们不愿意为 Lua 建一个公开的代码仓库。我们不想会我们做的每一处代码修改处处解释。不想为所有的更新保留文档。我们想在有些奇怪的想法时，有足够的自由来试一下，不满意的话就放弃掉，而不需要对每个行动都做一个解释。

为什么你喜欢获得建议和想法，而不是代码？我在想，或许你自己写代码能够让你学到关于问题（解决方案）的更多知识。

Roberto：差不多可以这么说。我们喜欢彻底搞清楚在 Lua 中发生了什么，因此，一段代码贡献不大。一段代码并不能解释为什么采用这种方式，但是，一旦我们理解了它的根本思想，编写代码就成了我们不想错过的乐事。

Luiz：我想对于引入第三方代码还有一个问题，我们无法确保其所有权。我们肯定不想溺死在要别人把代码授权给我们的合法化的过程中。

Lua 会不会达到这种状态：你已经添加了所有想要添加的特性，唯一需要的就是改进实现（例如，LuaJIT）？

Roberto: 我觉得现在就处于这种状态。我们已经添加的特性，即使不算是全部，也是我们想要添加的绝大部分。

你是如何操作冒烟测试和回归测试的？使用开放代码仓库的一大好处是，你可以让人们对几乎每一个修改进行自动测试。

Luiz：Lua 的发布并没有那么频繁，因此，发布一个版本时，已经进行过很多的测试。当这个版本已经相当可靠时我们才发布工作期版本（work version / pre-alpha 版），人们能够看中看到新添加的特性。

Roberto：我们确实进行了严格的回归测试。重点在于：因为我们的代码是用 ANSI C 编写的，基本上没有什么可移植性问题。我们没有必要在若干不同的机器上进行测试。一旦修改了代码，我就会执行所有的回归测试，不过这一切都是自动进行的。我要做的只是敲一下 test all 。

如果发现了一个反复出现的问题，到底是局部临时解决，还是全局通盘考虑，你如何判断哪一种是最佳解决方案？

Luiz：我们一直尽量做到一发现 bug 就修复它。不过，因为我们并不经常发布新的 Lua 版本。所以我们都是等到有足够的修复量才发布一个小版本。大版本做的都是改进工作而不是修复 bug 。 如果问题非常复杂（这种情况很罕见），我们会提供一个小版本作临时解决方案。而在下一个大版本中通盘考虑来解决它。

Roberto：通常，局部的权宜修复很快就可以完成。只有在确实不可能进行全局修复时，我们才会作局部的权宜方案。例如，如果某个全局修改需要一个新的不兼容接口。

从开始到现在，已经过去了这么多年，你仍然会为有限的资源而设计吗？

Roberto：当然会的，我们一直致力于此。我们甚至考虑过改变 C 结构内的字段顺序，以节省几个字节。:)

Luiz：相比于以前，现在有更多的人们把 Lua 语言运用到比以前更小的设备上面。

以用户视野来对简单性的追求怎样影响语言设计的？我想起了 Lua 对类的支持，让我想起了许多在 C 中实现面向对象的方式（不过没那么另人烦恼）。

Roberto：目前，我们有一个准则叫“机制而非法策”。它可以保证语言简洁，不过就像你说的，用户必须提供它自己的法则。就类这个问题来说，有很多方法实现它。有些用户会喜欢某种方式，而其他用户则可能痛恨它。

Luiz：这个确实赋予了 Lua 一种 DIY 的风格。

Tcl 也用了一种类似的方法，不过各家各有其法使它支离破碎。因为 Lua 有特定的目的，所以分裂对它不是啥严重问题吗？

Roberto: 对。有时这是个问题。但对于大量应用（比如说游戏）来说，这不是个问题。Lua 主要用来嵌入到别的应用程序中。而应用程序会提供一个坚固的框架来统一编程规范。你看到了 Lua/Lightroom, Lua/WoW, Lua/Wireshark —— 这个每个都有自己的内部文化。

你认为 Lua 这种“我们提供机制” 的展延性风格，给人带来巨大的好处吗？

Roberto：这么说并不确切。对于大多数事情来说，它是一种折衷处理。有时候，提供即刻可用的规范法则非常有用。“我们提供机制”更为灵活，但需要做更多的工作，并使得风格分裂。这最终也是个经济问题。

Luiz：另一方面，有时候这很难向用户解释。我的意思是，让他们理解是这些机制是什么，以及这些机制的原理。

这会使项目之间交流代码变得困难吗？

Roberto：没错，通常就是这样。它也阻碍了独立库的发展。例如，WoW 拥有大量的库（甚至连用遗传算法解决货郎担问题的库都有），不过在 WoW 之外却没人去用它们。

你担心 Lua 会因此分裂成 WoW/Lua，Lightroom/Lua 等分支吗？

Luiz：我们并不担心：语言还保持相同，只是可用的函数不同而已。我认为这些应用程序会在某些方面受益于此。

严肃的 Lua 用户会在 Lua 基础上编写他们自己的方言吗？

Roberto：很有可能。至少我们还没有宏。要是有宏的话，我认为你可以使用宏来创建一种真正的方言。

Luiz: 本质上还不算一种语言的方言。不过算是用函数来实现的一种特定领域语言。这曾是 Lua 的设计目的之一。当 Lua 仅仅用来作数据文件时，它看起来是一种方言，当然那些只是 Lua 表而已。有些项目或多或少实现了一些宏。比如我想起了 metalua 。这也是 Lisp 的一个问题。

你为何选择提供一种可扩充的语义？

Roberto：它开始是作为提供面向对象特性的一个方法。我们不想在 Lua 中添加 OO 机制， 但用户想要这些。我们想到这个方法，提供足够的机制让用户实现自己的 OO 机制。到现在我们也觉得这是一个正确的决策。然而，这使得用 Lua 的方式 OO 编程对于初学者来说更为困难。但它也给语言带来了大量的灵活度。特别是，当我们把 Lua 和其它语言混用（这是 Lua 的一个特色）时，这种灵活度使得程序员可以让 Lua 的对象模型去适应外部语言的对象模型。

目前的硬件、软件、服务和网络环境同你最初设计时的系统环境有何不同？这些变化对你的系统以及未来的改变有何影响？

Roberto：因为 Lua 是以极高的可移植性为目标，我认为目前的“环境”同以前的环境并没有什么不同。例如，我们开始开发 Lua 时，DOS/Windows 3 跑在 16 位机器上；一些老机器仍然是 8 位的。目前我们没有 16 位的台式机了，不过，若干使用 Lua 的平台（嵌入式系统）仍然是 16 位或者甚至是8位的。

最大的变化在于 C 语言。回头看 1993 年，当时我们刚开始做 Lua ，ISO (ANSI) C 还没有像今天这么成熟。很多平台仍然使用 K&R C 。很多应用程序写了一些很复杂的宏来使得程序通过 K&R C 和 ANSI C 两者的编译。主要的区别在函数头的声明。当时，坚持使用 ANSI C 是一个冒险的决定。

Luiz：我们仍未感觉到有必要转移到 C99 上面。Lua 是用 C89 实现的。如果过渡到 64 位机器上时出现些小毛病的话，或许我们必须使用 C99 的一部分（特别跟长度有关的类型定义），不过我并不希望出现任何问题。

如果能全部重新构建 Lua 的 VM 的话，你仍然会坚持使用 ANSI C 吗，或者你希望有一个更好的语言用于跨平台的底层开发？

Roberto：不。ANSI C 是我（目前）知道的可移植性最好的语言。

Luiz：有些杰出的ANSI C编译器，不过，即使是使用它们的扩展，也不会给我们带来很多性能提升。

Roberto：改进 ANSI C 并保持它的可移植性和性能并不容易。

顺便问一句，你是说 C89/90 吗？

Roberto：是的。C99 尚未确认好。

Luiz：再者，我不确定 C99 能给我们带来很多额外的特性。我还特别想到了 gcc 中使用的带标签的 goto 语句作为 switch 的一种替代方案（在虚拟机执行的主干里）。

Roberto：在很多机器中，这样做可以改进性能。

Luiz：我们早期对它作过测试，最近也有人也对它进行了测试，效果并不吸引人。

Roberto：部分原因在于我们基于寄存器的体系结构。它倾向于用较少的操作码，每个操作码分担更多的工作。这减少了分发器的负担。

你为什么要构建一个基于寄存器的 VM 呢？

Roberto：为了避免所有的 getlocal/setlocal 指令。我们也想去实践一下我们的想法。我们想啊，如果它运行得不好，至少我们还能写一些研究这个的论文。而最后，它运行得非常好，而我们也只写了一篇论文。:D

在 VM 上运行对调试有没有帮助？

Roberto：它没有提供“帮助”；它改变了整个调试的概念。既调试过编译型语言，又调试过解释型语言（比如 C 和 Java）的人都知道它们天差地别。好的VM 会让语言变得更安全，在某种意义上，该错误可以从语言层面上理解，而非机器层面（比如说段错误）。

如果语言是平台无关的，这对调试有何影响？

Roberto：通常它有利于调试，因为一种语言越是和平台无关，它就越需要可靠的抽象描述和行为。

考虑到我们是人，而人总会犯错。你是否曾经考虑过：为了在调试阶段有所帮助，需要向语言添加某种特性或是从中删除一些特性？

Roberto：当然了。辅助调试的第一步就是良好的错误消息。

Luiz：从初期版本开始，Lua 中的错误消息就在一直改进。我们已经从可怕的“调用操作对象不是一个函数”的错误消息（这条错误消息一直用到 Lua 3.2），变成了更好的错误消息：“试图调用全局 'f' (一个 nil 值)”。从 Lua 5.0 开始，我们使用对字节码的符号追踪 (Symbolic execution) 来试着提供更有用的错误消息。

Roberto：在语言自身的设计中，我们一直设法避免使用复杂的结构。如果它很难理解，就会更难调试。

在设计一门语言和设计用这种语言编写的程序之间，有什么联系？

Roberto：至少对我来说，设计一门语言的要点在于从用户的角度出发，也就是说，去考虑用户将怎样使用每一个特性，用户将会如何将这些特性和其它语言对比。程序员总会找到使用一种语言的新方式，优秀的语言应该允许那些意想不到的使用方法。不过，语言的“正常”用法应该遵从语言设计者的初衷。

语言的实现会在多大程度上影响语言的设计？

Roberto：这是一条双向道。实现会对语言产生巨大的影响：我们不应该设计无法高效实现的东西。一些人忘了这点。在设计任何软件时，效率一直是一个（或者是惟一的）主要约束条件。不过，设计也可能会对实现产生较大的影响。一眼看去，Lua 的几个特色之处都来自于它的实现（体积小、优秀的 C API ，以及可移植性），而 Lua 的设计在使这些实现变得可能中，起到了关键作用。

我读过你的一篇论文，《 Lua uses a handwritten scanner and a handwritten recursive descent parser（ Lua 使用一个手写扫描程序和一个手写的递归下降分析器）》。你是如何开始考虑手工构建一个分析器的？是不是从一开始就很清楚，这样做要比 yacc 生成的分析器要好？

Roberto：Lua 第一版使用了 lex 和 yacc 。不过，Lua 最初的主要目标之一是作为一种数据描述语言，和 XML 没什么不同。

Luiz：但是时间要更早一些。

Roberto：很快人们开始把 Lua 用于数兆字节的数据文件，此时 lex 生成的扫描器迅速变成了瓶颈。手写一个优秀的扫描器非常容易。而且只做了这么一点简单的改进后，我们就提高了 Lua 大约 30% 的性能。

决定从 yacc 改成手工编写解析器是很后来的事情，这个决定做得并不容易。这起源于几乎所有 yacc/bison 实现使用的主干代码的问题。

当时，它们的可移植性很差（例如，用了好多处的 malloc.h ，这是一个非 ANSI C 的头文件），而且，我们无法控制其整体质量（例如，控制堆栈溢出和内存分配错误等问题），而且它们也不是可重入的（比如要在解析代码的过程中调用

解析器）。另一方面，如果你想要像 Lua 那样及时生成代码，自底向上解析器也不如自顶向下的那么好。因为它难以处理“继承属性(Inherited attributes)”。我们改写之后，发现我们手写的解析器要比 yacc 生成的那个略小以及略快一点。不过这不是改写的主要原因。

Luiz：自顶向下分析器还能提供更好的错误消息。

Roberto：不过，我从不推荐为没有成熟语法的语言手写解析器。并可以肯定LR(1)（或是 LALR 甚至 SRL）会比 LL(1) 强大多了。甚至对于 Lua 这样的简单语法的语言来说，我们也必须使用一些技巧来构建一个像样的分析器。例如，处理二元表达式的程序并没有按原始语法去处理，而是用了一个聪明的基于优先级(priority-based)的递归方案。在我的编译器课上一直向我的学生推荐 yacc 。

你的教学生涯中有什么趣闻轶事吗？

Roberto：我刚开始教授编程时，供我们的学生使用的计算机设备是一台大型机。有一次，一个非常优秀的团队提交的一个程序作业，居然连编译都没通过。我找他们来谈话，他们发誓用好几个测试案例仔细的测试了程序。当然了，他们和我用的是同一台机器，完全相同的环境，都是在那台大型机上。这个神秘事件只到几周后才搞明白。原来机器上的 Pascal 编译器被升级了。升级刚好发生在学生完成任务和我开始批改作业之间。他们的程序有一个很小的词法错误（如果记得没错，是多了个分号），而老的编译器没有检测到！

云风 提交于 June 9, 2010 10:36 PM | 固定链接

COMMENTS

as simple as possible ,but not simpler

简单而不简陋

Posted by: zhb | (40) October 29, 2014 02:26 PM

你认为未来可能会用到的功能，就不要马上添加了：现在增加这个功能只会让你日后真的需要这个东西时，那些更好的特性很难加入。最后，永远追求更为简洁的解决方案。

我通常认为，如果有什么需要注释的，那只是因为程序没写好。对于我来说，一条注释更像是打了个便签，它在说“以后记得重写这段代码”。我认为清晰的代码要比带注释的代码可读性更强。

终于读完了，文章中提到的多处编程箴言，真是赞，顶；

Posted by: Anonymous | (39) November 23, 2013 05:48 PM

你认为未来可能会用到的功能，就不要马上添加了：现在增加这个功能只会让你日后真的需要这个东西时，那些更好的特性很难加入。最后，永远追求更为简洁的解决方案。

我通常认为，如果有什么需要注释的，那只是因为程序没写好。对于我来说，一条注释更像是打了个便签，它在说“以后记得重写这段代码”。我认为清晰的代码要比带注释的代码可读性更强。

终于读完了，文章中提到的多处编程箴言，真是赞，顶；

Posted by: Anonymous | (38) November 23, 2013 05:48 PM

只有一个分号，哈哈
As simple as possible,
but not simpler.

程序员的思考方式，数学之美

Posted by: c语言小程序 | (37) May 25, 2013 08:00 PM

我是北京友骐律师事务所负责人，做专利及合同领域法律服务，从97年开始该行业。业务范围涉及专利诉讼无效申请复审转让等。本人是律师，专利代理人资格，司法鉴定人
事务所网站有详情www.yqlaw.cn 含联系方式

Posted by: Anonymous | (36) May 23, 2013 09:23 PM

如果你想要像 Lua 那样及时生成代码
====
如果你想要像 Lua 那样“即时”生成代码
原文是when you want to generate code on the fly,，即动态生成或者运行时生成

Posted by: erazy0 | (35) April 18, 2012 09:39 PM

这个每个都有自己的内部文化
====
每种都有它们自己的内部文化

Posted by: erazy0 | (34) April 18, 2012 09:16 PM

用户必须提供它自己的法则
====
用户必须提供“他”自己的“策略”

Posted by: erazy0 | (33) April 18, 2012 09:13 PM

机制而非法策
====
直接用“机制而非策略”不好么

Posted by: erazy0 | (32) April 18, 2012 09:09 PM

我们不想会我们做的每一处代码修改处处解释
====
不想为

Posted by: erazy0 | (31) April 18, 2012 09:02 PM

仅可能用最好的机器来开发
====
尽可能

Posted by: erazy0 | (30) April 18, 2012 08:52 PM

程序员可以从 Lua 中的 first-class 函数中获得怎样的价值？
回答有两段，第二段没翻译。
====
Lua uses functions also in some unconventional ways, and the fact that they are first class simplifies those uses. For instance, every chunk (any piece of code that we feed to the interpreter) is compiled like a function body, so any conventional function definition in Lua is always nested inside an outer function. That means that even trivial Lua programs need first-class functions.
我翻译了一下，请指正：
Lua 还会以一些非常规方式来使用函数。而且函数作为first class简化了那些用法。例如，每个块(我们提供给解释器的任意代码片断)都可以像一个函数体那样被编译，因此，在Lua 中，任何常规的函数定义总是内嵌在一个外部函数中。这意味着即使是毫不起眼的Lua 程序也需要first-class函数。

Posted by: erazy0 | (29) April 18, 2012 08:35 PM

这种表示方式可以用于一些操作，比如联合和交集等。
====
应该是“并集和交集”

Posted by: erazy0 | (28) April 18, 2012 08:02 PM

因为它们也来至于计算机之外!
===

来自于

Posted by: erazy0 | (27) April 18, 2012 07:42 PM

偶像

Posted by: angzh | (26) September 5, 2011 03:55 PM

一个typo，最后一段SLR

Posted by: 素雪 | (25) January 8, 2011 02:55 PM

lua真的用了jit吗？？我翻遍了代码没找到。

Posted by: no name | (24) August 29, 2010 04:57 PM

都是高手云集的地方

Posted by: 我的大学 | (23) July 6, 2010 10:42 AM

@chunting

这里这篇就是我的译本。

那句我的版本是：
极尽简洁，然过犹不及(As simple as possible, but not simpler.)——爱因斯坦

Posted by: cloud | (22) June 28, 2010 06:54 PM

爱因斯坦的那句名言，后半句的翻译似乎应该重新考量。原文是 Everything should be made as simple as possible, but not simpler. 今天看到中文版 The art and science of C 里翻译为：任何事情都应该尽可能简单，【而不是较简单。】
云风认为呢？

Posted by: chunting | (21) June 28, 2010 06:02 PM

文笔真好呀

Posted by: mike | (20) June 13, 2010 07:37 PM

@ery

建议在 package.path 增加 /opt/monit/plugins/?.lua

否则考虑使用 require _PACKAGE.."plugin" 这样的形式去 require 相同目录下的 plugin.lua

Posted by: cloud | (19) June 12, 2010 05:11 PM

请教lua的path设置问题，我们系统是erlang中调用/opt/monit/plugins/check_xxx.lua,
其中check_xxx.lua都require同目录下的一个"plugin.lua"文件，同目录下执行插件无问题，但绝对路径执行就需要设置lua path，请问如何设置比较合理？

Posted by: ery | (18) June 12, 2010 03:56 PM

为什么一来这里心就沉稳了一些...虽然讲的是我不懂的lua,forth...不管怎么样来这里一次学习的心就坚定一分....感谢云风了...

Posted by: ucmvoliton | (17) June 12, 2010 12:28 PM

lua 是了不起的成就。
居然是某发展中国家的人搞出来的。
令人汗颜。

Posted by: mike | (16) June 11, 2010 11:48 AM

first-class 译成 第一级 怎样？看到很多人都这样翻译。

Posted by: chunting | (15) June 11, 2010 11:46 AM

文章虽然长,但很值得一读,对于lua开发者应该必读. 我摘录出几句印象最深,最值得深思的经验之谈:

应用程序设计者应该从一开始就考虑脚本。这会给它们带来更多的灵活性。而且这样做还可以更好的把握性能问题。因为这样做以后，会迫使他们去考虑程序中到底哪里是性能关键，而哪些地方无伤大雅。

我们不信任基于抢占式内存共享的多线程技术。我们仍然认为，如果在连 a=a+1 都没有确定结果的语言中，无人可以写出正确的程序。

你认为未来可能会用到的功能，就不要马上添加了：现在增加这个功能只会让你日后真的需要这个东西时，那些更好的特性很难加入。最后，永远追求更为简洁的解决方案。

我通常认为，如果有什么需要注释的，那只是因为程序没写好。对于我来说，一条注释更像是打了个便签，它在说“以后记得重写这段代码”。我认为清晰的代码要比带注释的代码可读性更强。

没有什么工具可以代替一份井井有条、深思熟虑的文档。

Posted by: dwing | (14) June 11, 2010 09:55 AM

first class是最难翻译的，怎么翻译都觉得不顺口。

Posted by: analyst | (13) June 10, 2010 06:11 PM

@zhaowe

昨天最早我用的头等，后来读起来不太顺，又改成了一等。

Posted by: cloud | (12) June 10, 2010 03:49 PM

first-class翻译成“头等”似乎比“一等”要好一些。

Posted by: zhaowe | (11) June 10, 2010 03:16 PM

真的很有感触。

作为一个初级程序员，我越发感觉到语言的重要性。老实说我最讨厌的语言，就是c++。

经常有人说，“语言不重要，思想才是重要的“。但其实语言是可以极大的影响人的思维方式的。就好像给你一把锤子，你多半会做出敲打的动作；而给你一把剪刀，你多半会做出剪裁的动作。

c++就好像给你一把很不好用的瑞士军刀，每个都不是那么好用，结果就是明明很简单可以处理的问题，用户可能巴不得用十八般兵器对处理。

我平时用的最多的还是c。前段时间，我试图考虑怎么去遍历c语言的结构体，然后我试着自己构建了一个结构体元信息表，里面用offsetof宏记录每个元素的偏移，但遗憾的是，由于c不是动态语言，我无法运行时实现每个元素的类型转换。但当我偶然知道python中有这样一种数据结构，它可以像数组一样使用，同时可以存储不同类型的元素时，我才明白，其实我并不是想遍历结构体，而是想获得一个类似python列表这样的数据结构。

以前觉得像ACE这样的库很牛逼，但是发现库牛X到一定程度，就要求你按照它的模式去使用它，很不舒服。其实设计一个号的库，真的不如设计一门新的语言。因为语言本身就是模式。

Posted by: thornyroad | (10) June 10, 2010 02:44 PM

不错，对lua的开发历史，开发模式，设计原则等又有了更多了解。

Posted by: phoolimin | (9) June 10, 2010 01:13 PM

内牛满面，才买了这本书。感觉翻译的是有些晦涩。

Posted by: xuruoji | (8) June 10, 2010 12:52 PM

“我们仍然认为，如果在连 a=a+1 都没有确定结果的语言中，无人可以写出正确的程序。”
+1

“优秀的语言应该允许那些意想不到的使用方法”
我可以认为C++也符合这个标准吗？ XD 至少从这个角度讲，C++是可爱的。

Posted by: sjinny | (7) June 10, 2010 09:50 AM

是什么促使你决定使用垃圾搜集器==>垃圾收集器
闭包至始至终我们都想在 Lua 中实现==>自始至终
上班时间，没来得及仔细看……

Posted by: xbeggar | (6) June 10, 2010 09:21 AM

翻译的不错，不过有个别地方有错别字...

Posted by: firejacky | (5) June 10, 2010 08:59 AM

@Cofyc

老版译文删漏了。

Posted by: cloud | (4) June 10, 2010 12:53 AM

感觉翻译！ -> 感谢翻译！

囧……

Posted by: Cofyc | (3) June 10, 2010 12:46 AM

收获很多。感觉翻译！

P.S. 这两段似乎重复了？
* Luiz：是的，它本质上不是一种语言分支，而是.... 例如，我想起了metalua。这是Lisp的一个问题。（译注[7]）
* Luiz: 本质上还不算一种语言的方言。不过算是... 比如我想起了 metalua 。这也是 Lisp 的一个问题。

Posted by: Cofyc | (2) June 10, 2010 12:44 AM

重做版本吗?我还正准备买呢...有没有说什么时候出新版本?

Posted by: DJ | (1) June 10, 2010 12:36 AM

POST A COMMENT

非这个主题相关的留言请到：留言本

名字：

Email 地址：

为了验证您是人类，请将六加一的结果（阿拉伯数字七）填写在下面：

URL:

记住我的信息？

留言：
（不欢迎在留言中粘贴程序代码）

提交