Lua Performance Tips

Lua性能建议——Roberto Lerusalimschy

就像在其他任何编程语言中一样，在Lua中，我们始终要遵循两条程序优化准则：

- 不要这么做
- 暂时不要这么做（仅对于专家而言）

这些准则在使用Lua编程时极为重要。Lua在众多脚本语言中以性能而闻名，而它也的确对得起它的名声。尽管如此，我们都知道性能是编程的关键要素。具有指数时间复杂度的问题被称为棘手的问题并非偶然。结果来的太迟还不如不来（原文直译：太迟的结果就是无用的结果）。因此每个优秀的程序员都应该始终在花费资源来优化一段代码的成本和运行该代码时节省资源的收益之间取得平衡。

好的程序员关于优化问的第一个问题总是："程序是否需要优化？"，如果答案是肯定的（也必然是肯定的），第二个问题就是："在哪里优化？"

我们需要一些仪器来回答这两个问题，不应在没有适当的评估的情况下尝试优化代码。有经验的程序员和新手间的区别不在于有经验者更擅长发现程序可能浪费时间的地方，而在于他们知道自己不擅长发现那些地方。

几年前，我和Noemi Rodriguez使用Lua开发了CORBAORB (Object Request Broker，对象请求代理)的原型，之后它发展为了OiL (Orb in Lua)。作为第一个原型，它的实现重在简化。为了避免对C代码库的过度依赖，该原型使用一些算术运算来序列化整数以隔离每个字节（转化为基数256）。它不支持浮点数运算。由于CORBA将字符串作为字符序列来处理，ORB首先将Lua字符串转化为字符序列（Lua table存储），然后再像其他地序列一样进行处理。

当该ORB原型完成时，我们将它的性能表现和使用C++实现的一个专业ORB进行了对比。我们希望的是我们的ORB会稍微慢一些，毕竟它是用Lua实现的，但令我们失望的是，它慢的有点多。起初，我们将责任归咎于Lua；但在那之后，我们怀疑罪魁祸首可能是序列化每个数字所需的那些操作。因此，我们决定在Profile（探查器）下运行该程序；我们使用了一个非常简单的探查器，探查的结果震惊了我们。

与我们所想的不同，数字的序列化对性能没有可衡量的影响，因为序列化的数字并不多。然而，对字符串的序列化，占据了总时间的很大一部分。即使我们没有明确地操作字符串，但几乎每条CORBA消息都有几个字符串：对象引用、方法名和一些其他的内部值也都被编码为字符串。每个字符串的序列化是一项昂贵的操作，因为这需要创建一个新的table，使用单独的字符将其填满，然后生成序列，这涉及到一个接一个地序列化每个字符。当我们重新实现了字符串在特殊情况（而不是对序列使用通用代码）下的序列化，我们立即得到了可观的加速。仅需要几行代码，它的性能就已经和C++媲美了。

因此，在优化程序性能时，我们应该一直进行测量。优化前测量，能让我们知道在那里进行优化；优化后测量，让我们知道优化是否真正改善了我们的代码。

一旦你决定你的Lua代码确实需要优化，本文会帮助你知道如何去进行优化，主要是展示Lua中什么操作比较快，什么比较慢。这里我们不讨论一般的优化技术，诸如更好的算法之类。当然，了解并使用这些技术是必需的，但是还存在其他的蹊径。本文中我仅讨论Lua特有的技术；我将不断评估小型程序的时间和空间性能。除非另有说明，否则所有的操作都将运行在Ubuntu 7.10和Lua 5.1.1的主内存为1 GB

的Pentium IV 2.9 GHz上。我经常会给出实际的衡量标准（例如7秒），但关键的是不同评估之间的关系。当我说一个程序比另一个程序"快X%倍"时，意味着它的运行时间减少了X%（一个程序快100%无需花费时间）。当我说一个程序比另一个程序"慢X%倍"时，我的意思是另一个程序快X%（程序慢50%意味着它花费了两倍的时间）。

Basic facts 基本事实

在运行任何代码之前，Lua先将源代码转换（预编译）为内置格式。该格式是针对虚拟机的一些列指令，与真实CPU的机器码相近。然后C代码会解释该内部结构；该C代码本质上是一个while循环，内部包含了一个很大的switch语句，每条指令对应了一个case。

也许你在什么地方了解到，从5.0版本开始，Lua就开始使用一种基于寄存器的虚拟机了。该虚拟机的寄存器不对应CPU中的寄存器，因为这种对应关系是不可移植的，并且极大程度受限于可用寄存器的数量。Lua使用栈（使用数组加一些索引）来容纳自己的寄存器。每个激活的函数都有自己的激活记录，该记录是一个栈的切片，函数在其中存储自己的寄存器，即每个函数都有自己的寄存器。每个函数最多可以使用250个寄存器，因为每条指令最多只能有8位来引用一个寄存器。

得益于大量的寄存器，Lua预编译器能够存储所有local变量到寄存器中，结果就是在Lua中，对local变量的访问非常快。举个例子，如果 a 和 b 是局部变量，`a = a + b` 这样的Lua语句将生成一条指令 `ADD 0 0 1`（假设 a 和 b 分别位于寄存器0和1中）。为了进行比较，如果 a 和 b 都是全局变量，则该加法的代码会变成这样：

```
GETGLOBAL       0 0     ; a
GETGLOBAL       1 1     ; b
ADD             0 0 1
SETGLOBAL       0 0     ; a
```

因此，很容易得出，提高Lua程序性能最重要的准则之一：使用local变量！

如果你要从程序中压缩性能，除了显而易见的地方外，还有几个地方可以优化。例如，如果你要在循环体中调用函数，则可以将该函数分配给局部变量。例如：

```
for i = 1, 1000000 do
    local x = math.sin(i)
end
```

比下边的代码慢30%

```
local sin = math.sin
for i = 1, 1000000 do
    local x = sin(i)
end
```

对外部local变量（对封闭的函数来说的local变量）的调用并不像对local变量的调用那么快，但是它仍然比调用全局变量要来得快，看下面的片段：

```
function foo (x)
    for i = 1, 1000000 do
        x = x + math.sin(i)
    end
    return x
end

print(foo(10))
```

我们可以通过在函数foo外声明函数sin来优化这段代码：

```
local sin = math.sin
function foo (x)
    for i = 1, 1000000 do
        x = x + sin(i)
    end
    return x
end

print(foo(10))
```

第二段代码大概比第一段代码快30%。

尽管和其他语言的编译器相比，Lua编译器非常高效，但编译是一项繁重的任务。因此，要尽可能避免在程序中编译代码（例如函数loadstring）。除非必需运行真正动态的代码（例如最终用户输入的代码），否则很少需要编译动态代码。

例如，下面的代码创建了一个表，其中存储了返回从1到100000的常量值的一系列函数：

```
local lim = 100000
local a = {}
for i = 1, lim do
    a[i] = loadstring(string.format("return %d", i))
end
print(a[10]())  --> 10
```

这段代码运行要花1.4秒，使用闭包，我们能避免这种动态编译。这段代码创建了相同的100000个函数，但是却只运行了十分之一的时间（0.14秒）。

```
function fk (k)
    return function () return k end
end

local lim = 100000
local a = {}
for i = 1, lim do a[i] = fk(i) end
print(a[10]())  --> 10
```

About tables 关于表

通常来说你不需要了解有关Lua如何实现表已使用它们的任何知识，实际上，Lua竭尽全力确保实现细节不向用户表现出来。因此要优化使用表的程序（实际上是任何Lua程序），最好对Lua如何实现表有所了解。

Lua中表的实现涉及一些十分聪明的算法。Lua中的每个表都有两部分：数组（array）部分和哈希（hash）部分。数组部分存储了一些条目和其1到n范围内的整数键。这个n的值是特定的，稍后我们将讨论如何计算这个n。所有其他的条目（包括越界的整数键）都被存储在哈辛部分中。

顾名思义，哈希部分使用哈希算法来存储和查找其键。它使用了所谓的开放地址表，这意味着所有的条目都存储在哈希数组本身中。哈希算法给出键的主索引，如果发生冲突（即如果两个键被散列到同一位置），则这些键被链接到一个列表中，每个元素占据一个条目。

当Lua需要向表中插入一个新键并且哈希数组已满时，Lua会进行一次Rehash。第一步是确定新的数组部分和哈希部分的大小。所以Lua会遍历所有条目，对他们进行计数与分类，然后选择数组部分的大小，要求是数组部分的一半以上都能被填充的2的最大幂。哈希部分的大小则是可以容纳所有剩余条目的2的最小幂（那些不适合数组部分的条目）。

当Lua创建新表时，两个部分的大小都是0，没有为它们分配数组。让我们看看运行下面的代码会发生什么：

```
local a = {}
for i = 1, 3 do
    a[i] = true
end
```

首先该段代码创建了一个空表a。在第一次循环中，赋值语句`a[1] = true`触发了一次Rehash；然后Lua设置表的数组部分的大小为1，并保持哈希部分为空。第二次循环中，赋值语句`a[2] = true`触发了另一次Rehash，现在数组部分的大小为2。最后，第三次循环又触发了一次Rehash，将数组部分的大小设为了4。

```
a = {}
a.x = 1; a.y = 2; a.z = 3
```

上面的代码页做了类似的事情，只不过其改变的是哈希部分的大小。

对于大型表来说，该初始开销将在整个创建过程中分摊，虽然包含三个元素的表需要进行三次哈希处理，具有一百万个元素的表却只需要进行二十次哈希处理。但是，当你创建数千个小表时，总开销会非常大。

老版本的Lua创建了带有些预先分配的条目的空表，如果没记错的话应该是4个。这些条目的目的是避免初始化小表时的这种开销。但是，这种方法浪费了内存。例如，你创建了百万个点（只含有两个值的表），而每个点使用的内存只是实际需要的两倍，这样的内存开销太大了。这就是最新版的Lua表没有预分配条目的原因。

如果你使用C语言进行编程，则可以用Lua API函数lua_createtable避免这些Rehash。它在无所不在的lua_State之后接收两个参数：新表的数组部分的初始大小和哈希部分的初始大小（要注意的是，尽管Rehash会将数组部分的大小设为2的幂次方，但是数组部分的大小可以为任何值；相反的，哈希部分的大小永远只能是2的幂次方，如果你给与了不合适的值，该函数会将哈希部分的大小设为不小于你给定的值的最小的2的幂次方）。通过为新表赋予适当的大小，避免这些初始Hash的过程十分轻松，只不过这样的话Lua只能在Rehash的时候收缩表了。因此，如果你的初始大小大于所需的大小，Lua可能永远不会纠正您浪费的空间。

在Lua中进行编程时，可以使用构造函数来避免那些最初的Rehash。当你编写`{true, true, true}`时，Lua事先知道该表的数组部分将需要三个条目，因此Lua会创建具有该大小的表。同样，如果你编写`{x = 1, y = 2, z = 3}`，Lua将创建一个表，该表的哈希部分有四个条目，例如，下个循环将在2.0秒内运行：

```
for i = 1, 1000000 do
    local a = {}
    a[1] = 1; a[2] = 2; a[3] = 3
end
```

如果我们创建带有初始大小的表，则这个循环只需要花费0.7秒的时间：

```
for i = 1, 1000000 do
    local a = {true, true, true}
    a[1] = 1; a[2] = 2; a[3] = 3
end
```

但是，如果您编写类似`{[1] = true，[2] = true，[3] = true}`的内容，则Lua不够聪明，无法检测到给定的表达式（在这种情况下为数字）描述数组索引， 因此它创建了一个表，该表的哈希部分有四个插槽，浪费了内存和CPU时间。

仅当表Rehash时，表的两部分的大小才会发生改变，即只有当表完全充满且Lua要插入新元素时才会发生。因此，遍历一个表并清除所有字段（设为nil）是不会减小表所占据的空间的。但是，如果在表中插入一些新元素，则表将不得不调整大小。这通常不是问题，如果你继续擦除元素并插入新元素（在许多程序中很常见），表的大小将保持稳定。但是，你不应期望通过擦除达标的字段来恢复内存，最好直接释放表本身。

强制Rehash的一个肮脏的技巧是在表中插入足够的nil元素，如下：

```
a = {}
lim = 10000000
for i = 1, lim do a[i] = i end            --  create a huge table
print(collectgarbage("count"))            --> 196626
for i = 1, lim do a[i] = nil end          --  erase all its elements
print(collectgarbage("count"))            --> 196626
for i = lim + 1, 2*lim do a[i] = nil end  --  create many nil elements
print(collectgarbage("count"))            --> 17
```

我并不推荐这个技巧，除非有特殊情况，原因很简单，对大表来说，这个操作太慢了，而且没有一个很简单的办法来指导“足够”的元素数量是多少。

你可能想知道为什么在插入nil值时Lua不收缩表。首先，避免测试我们要插入表中的内容，检测插入的值是否为nil会降低所有插入的速度；其次，也是更重要的，遍历表时允许赋nil值。看下面的循环体：

```
for k, v in pairs(t) do
    if some_property(v) then
        t[k] = nil      -- erase that element
    end
end
```

如果Lua在赋空值后Rehash整个表，会破坏整个遍历。如果你真的想清除表中所有的内容，正确的方法是进行一次简单的遍历：

```
for k in pairs(t) do
    t[k] = nil
end
```

更为“聪明”的做法是这样：

```
while true do
    local k = next(t)
    if not k then break end
    t[k] = nil
end
```

然而，像这样的循环对于大型表来说是很慢的。next函数在没有制定键的情况下被调用时，会返回表中的“第一个”值（然而这是以随机序列制定的）。为了这样做，next函数遍历整个表，寻找一个不为nil的值；由于该循环赋第一个值为nil，next要花费越来越长的时间来寻找第一个不为空的值。结果就是，这个看似“聪明”的循环花了近20秒来擦除一个大小为100000的表；而使用pairs函数的循环则只花了0.04秒。

About strings 关于字符串

和表一样，最好了解一下Lua是怎么实现字符串的，以便于更高效的使用它们。

Lua实现字符串的方式和其他大部分语言有两个重要方面的不同。

首先，Lua中所有的字符串都被内部化了，意思是Lua保留着任何字符串的单个副本。每当出现新的字符串时，Lua会检查是否已经有该字符串的副本，如果有，重新使用该副本。内部化（Internalization）使得想字符串对比和表索引之类的操作十分迅速，但是这降低了字符串的创建速度。

其次，Lua中的变量从不自己持有字符串，而仅引用它们。此实现加快了一些字符串操作的速度。举例来说，在Perl语言中，当你编写类似 `$x = $y` 的内容，其中的 `$y` 包含一个字符串时，该赋值操作将字符串从 `$y` 复制到 `$x` 中。如果这个字符串很长，该操作的消耗就会十分昂贵。而在Lua中，该赋值操作仅仅复制了一个指针。

不过这种实现模式减慢了特定形式的字符串拼接。在Perl中，操作 `$s = $s."x"` 和 `$s .= "x"` 完全不同，第一个操作你将获得 `$s` 的副本，并在其末尾添加“x”，在第二个操作中，“x”仅附加到 `$s` 变量所保存的内部缓冲区中。因此第二种形式与字符串大小无关（假设缓冲区中有空间来添加附加的内容）。如果在循环中包含这些命令，它们的区别在于线性算法和二次算法。例如，这个循环将花费将近五分钟来读取5MByte的文件：

```
$x = "";
while (<>) {
    $x = $x . $_;
}
```

如果将循环中的 `$x = $x . $_` 改成 `$x .= $_`，则只需要花费0.1秒的时间！

Lua不提供第二个操作这样的更快的选项，因为它的变量没有与之关联的缓存区。因此，我们需要使用显式缓存区：包含字符串的表可以完成此操作。这个循环将在0.28秒内读取相同的5MByte文件，速度不及Perl，但可以接受：

```
local t = {}
for line in io.lines() do
    t[#t + 1] = line
end
s = table.concat(t, "\n")
```

Reduce, reuse, recycle 减少，复用，回收

在处理Lua资源时，我们应该使用和地球资源一样的三个R。

Reduce是最简单的选择。有几种方法避免对新对象的需求。例如，如果你的程序使用了太多的表，则可以考虑更改其数据表示形式。举个简单的例子，如果你的程序操作折线，Lua中折线最自然的形式是点列表，如下：

```
polyline = { { x = 10.3, y = 98.5 },
             { x = 10.3, y = 18.3 },
             { x = 15.0, y = 98.5 },
             ...
}
```

尽管很自然，对于大型折现来说，这种表示方式并不是很经济，因为每个单点都需要一个table。第一个选择是将记录更改为数组，从而使用更小的内存：

```
polyline = { { 10.3, 98.5 },
             { 10.3, 18.3 },
             { 15.0, 98.5 },
             ...
}
```

对于有100W个点的折线，此更改将内存使用从95K降低到了65K，当然，这为可读性付出了代价——`p[i].x`比`p[i][1]`更容易理解。不过还有更经济的方式：

```
polyline = { x = { 10.3, 10.3, 15.0, ...},
             y = { 98.5, 18.3, 98.5, ...}
}
```

这样，原来的`p[i].x`就变成了`p.x[i]`，使用这种表现形式，同样的曲线就仅仅占用24K的内存了。

寻找减少垃圾制造机会的最好地方就是循环了。举例：如果一个不变的表在循环中被创建，你可以将它移到循环外，甚至直接移出包含它的函数。对比如下：

```
function foo (...)
    for i = 1, n do
        local t = {1, 2, 3, "hi"}
        -- do something without changing 't'
        ...
    end
end

local t = {1, 2, 3, "hi"} -- create 't' once and for all
function foo (...)
    for i = 1, n do
        -- do something without changing 't'
        ...
    end
end
```

同样的，对闭包来说，只要您不将它们移出所需的变量范围，就可以使用相同的技巧。 例如：

```
function changenumbers (limit, delta)
    for line in io.lines() do
        line = string.gsub(line, "%d+", function (num)
            num = tonumber(num)
            if num >= limit then return tostring(num + delta) end
            -- else return nothing, keeping the original number
        end)
        io.write(line, "\n")
    end
end
```

我们可以通过将内部函数移到循环外来避免每一行都创建新的闭包：

```
function changenumbers (limit, delta)
    local function aux (num)
        num = tonumber(num)
        if num >= limit then return tostring(num + delta) end
    end
    for line in io.lines() do
        line = string.gsub(line, "%d+", aux)
        io.write(line, "\n")
    end
end
```

不过函数aux不能移动到函数changenumbers之外，因为那里它无法访问limit和delta。

对很多字符串操作来说，我们可以通过在现有字符串上使用索引来减少对新字符串的需求。例如，string.find返回找到模版的位置，而不是匹配项。通过返回索引，它避免了为每个成功的匹配创建新的字符串。必要时，程序员可以通过调用函数string.sub来获得匹配的字符串。（标准库中添加一个比较子字符串功能的函数是一个好主意，这样我们就可以检查字符串中特定的值，而不必从字符串中提取改值创建一个新字符串）。

当我们不可避免地要使用新对象时，我们仍可以通过Reuse避免创建新对象。对字符串来说，重用不是必要的，因为Lua已经做了这方面的工作了：它总是内部化它使用的所有字符串，因此在可能的情况下都可以重用它们。但对表来说，重用可能非常有效。通常情况（一般指在循环中创建新table）下，表的内容不是恒定的，尽管如此，我们仍然可以在所有迭代中重用同一张表，只需要更改其内容即可：

```
local t = {}
for i = 1970, 2000 do
    t[i] = os.time({year = i, month = 6, day = 14})
end
-- 下面的操作是一样的，但是重用了表aux
local t = {}
local aux = {year = nil, month = 6, day = 14}
for i = 1970, 2000 do
    aux.year = i
    t[i] = os.time(aux)
end
```

实现重用的一种特别有效的方法是通过memoizing记忆。基本思想很简单：存储给定输入的某些计算结果，以便在再次提供相同输入时，程序仅重用先前的结果。

LPeg是Lua中用于模式匹配的新包，它有趣的使用了记忆。LPeg将每个模式编译成一个内部表示形式，这是给执行匹配的解析机使用的"程序"。和匹配本身相比，这个编译是十分昂贵的。因此LPeg会记住其编译结果以重复使用。一个简单的表将描述模式的字符串和其对应的内部表示形式相关联。

记忆的一个常见问题是，存储先前结果的空间成本可能超过重用这些结果的收益。为了在Lua中解决此问题，我们可以使用一个弱表来保留结果，以便最终将未使用的结果从表中删除。

在Lua中，使用高阶函数，我们可以定义一个通用的记忆函数：

```
function memoize (f)
    local mem = {}                          -- memoizing table
    setmetatable(mem, {__mode = "kv"})      -- make it weak
    return function (x)                     -- new version of 'f', with memoizing
        local r = mem[x]
        if r == nil then                    -- no previous result?
            r = f(x)                        -- calls original function
            mem[x] = r                      -- store result for reuse
        end
        return r
    end
end
```

给定任何函数f，`memoize(f)`返回一个新函数，该函数返回与f相同的结果，但会对齐进行记忆。例如，我们可以重定义loadstring的一个记忆版本：

```
loadstring = memoize(loadstring)
```

我们使用这个新函数的方式与旧函数完全相同，但是如果要加载的字符串中有很多重复的字符串，则可以大大提高性能。

如果您的程序创建并释放了过多的协程，则回收可能是提高其性能的一种选择。 当前的协程API不提供重用协程的直接支持，但我们可以规避此限制：

```
co = coroutine.create(function (f)
    while f do
        f = coroutine.yield(f())
    end
end
```

这个协程接收一个任务（要运行的函数），运行该任务，并在完成时等待下一个工作。

Lua中的大部分回收是由垃圾收集器自动完成的。Lua使用增量垃圾收集器，这意味着收集器在与程序执行交错的较小步骤中（逐步）执行其任务。这些步骤的速度与内存分配成正比：对Lua分配的每个内存，GC都按比列进行工作，程序消耗内存的速度越快，收集器尝试回收它的速度就越快。

如果我们将Reduce和Reuse准则用于我们的程序，通常收集器将没有太多工作要做。但有时我们无法避免产生大量垃圾，并且GC可能会变得太过于庞大。Lua中的GC对一般程序进行了调整，因此在大多数应用程序中性能都相当不错。不过，有时我们可以通过针对特定情况调整收集器来提高程序的性能。

我们可以使用Lua中函数collectgarbage（或者C中的lua_gc）来控制GC，尽管接口不同，它们都提供了基本相同的功能。在本次讨论中，我将使用Lua接口，但是通常使用C语言可以更好地完成这种操作。

函数collectgarbage提供了几种功能：停止并重启GC，强制执行完整的收集周期，强制执行收集步骤，获取Lua使用的总内存，并更改两个影响收集器运行速度的参数。在处理需要大量内存的程序时，它们都有自己的用途。

永久停止GC可能是某些批处理程序的一种选择，这些程序创建多个数据结构，根据这些结构产生一些输出，然后退出（例如，编译器）。对于那些程序，尝试收集垃GC圾可能会浪费时间，因为几乎没有垃圾可以收集，并且在程序完成时将释放所有内存。

对于非批处理程序，永远停止GC不被推荐。但是，在某些时间紧迫的时间段内停止收集器可能是有益的。如有必要，该程序可以通过始终保持GC停止，而只能通过显式强制执行或完全收集来使其运行的方式获取GC的全部控制权。例如，几个事件驱动平台提供了一个设置空闲函数的选项，该空闲函数在没有其他事件需要处理时调用。这是进行垃圾收集的最佳时机。 （在Lua 5.1中，每次在GC停止时强制进行一些收集时，它都会自动重新启动。因此，要使其保持停止，必须在强制进行一些收集后立即调用collectgarbage（“ stop”）。

最后，作为最后的选择，您可以尝试更改收集器参数。收集器具有两个控制其速度的参数。第一个称为pause暂停，它控制收集器在完成收集周期与开始下一个收集周期之间等待的时间。第二个参数称为stepmul（名字来源于step multiplier步骤乘数），它控制收集器在每个步骤中执行的工作量。大致而言，较小的停顿和较大的步进倍数会提高收集器的速度。这些参数对程序整体性能的影响很难预测。更快的收集器显然会浪费更多的CPU周期。但是，它可能会减少程序使用的总内存，从而可能减少分页。只有仔细实验才能为这些参数提供最佳值。

Final remarks 最后的提示

正如我们在简介中讨论的那样，优化是一项棘手的事情。有几点需要考虑，为首的就是程序是否需要任何优化。如果程序真的有性能问题，那么我们就要专注于在哪里优化、如何优化。

我们所讨论的技术既不是唯一的，也不是最重要的。因为有一些更通用的技术来源，我们在这里专注于Lua特有的技术。

在结束之前，我想提一下两个选项，它们是提高Lua程序性能的临界点，因为这两个选项都涉及Lua代码范围之外的更改。第一个是使用LuaJIT，这是由Mike Pall开发的Lua即时编译器。他一直做得非常出色，LuaJIT可能是当今动态语言中最快的JIT。缺点是它只能在x86架构上运行，并且需要非标准的Lua解释器（LuaJIT）来运行程序。优点是您可以使程序运行速度提高5倍，而无需对代码进行任何更改。

第二种选择是将部分代码移至C。毕竟，Lua标志之一是其与C代码互相提供接口的能力。在这种情况下，重要的一点是为C代码选择正确的粒度级别。一方面，如果仅将非常简单的功能移入C，则Lua和C之间的通信开销可能会扼杀这些功能的改进带来的任何收益。另一方面，如果将太大的函数移到C中，则会失去灵活性。

最后，请记住，这两个选项有些不兼容。您的程序拥有的C代码越多，LuaJIT对其进行优化的能力就越低。