### 使用local

1，编译器：翻译程序

作用：将源语言程序翻译为目标语言程序

源语言程序翻译:某程序语言写的，比如：C,C#，JAVA等语言

目标语言程序：2进制数表示的机器代码写的程序

2，先大概讲下解析器，与编译器功能有点类似

解释器在语义分析后选择了直接执行语句；编译器在语义分析后选择将将语义存储成某一种中间语言，之后通过不同的后端翻译成不同的机器语言（即可执行程序）

3，写好的运行前，lua会把源代码预编译为一种中间码（这种中间码一般会被进一步编译为可执行程序）。中间码会通过C语言的解析器进行解析（lua解析器用最纯的c来写的）。5.0以前类似java的jvm,基于栈的虚拟机

4，5.0后，栈的虚拟机--》寄存器虚拟机

寄存器：用于暂存指令，数据和地址

Lua用栈来存储其寄存器，每一个活动的函数lua都会分配一个栈，么一个函数的栈可以存储至少250个寄存器。Lua预编译器能把所有的**local变量存储在寄存器中**，这使lua在获取local变量时效率很高。**也就是说local的能存储在寄存器中，而全局不能。**

**存储在寄存器的好处：减少了预编译产生的指令**

简单例子：

a和b为local ,a+b

a：寄存 b:寄存

ADD 0 0 1 ：一条指令

a和b不为local ,a+b

GETGLOBAL 0 0 ;get a

GETGLOBAL 1 1 ;get b

ADD 0 0 1 ;do add

SETGLOBAL 0 0 ;set a

4条指令

### 表（Table）

Lua表可以分为2部分组成：数组部分和哈希hash部分。

数组部分包含所有从 1 到 n 的整数键，其他的所有键都储存在哈希表部分中

当我们把一个新键值赋值给表中，若数组和哈希表已经满了，则会触发一个再哈希，再哈希代价很高。首先再内存分配一个新的长度数组然后原来的记录转到先数组中，新哈希表长度接近与所有元素数目的2的乘方。

空表{}，数组和哈希的长度都为0，即不会为他们初始化任何数组，

结论：你有很多非常多的很小的表需要创建时，你可以将其预先填充以避免 rehash(哈希部分，)。

先进行填充再改变即避免rehash

例子：

for i = 1,2000000 do

local a = {}

a[1] = 1; a[2] = 2; a[3] = 3

end

b = os.clock()

print(b-a) --1.528293

--优化

a = os.clock()

for i = 1,2000000 do

local a = {1,1,1}

a[1] = 1; a[2] = 2; a[3] = 3

end

b = os.clock()

print(b-a) --0.746453

### 字符串

所有的字符串在lua中都只存储一份拷贝，lua检查是否有其相同的拷贝（类似java中的字符串常量池吧），若找不到则创建。

字符串变量保存的只是其引用（与java一样），这样做在赋值个复制操作很快，改变的只是其引用所指向的地址即可。但是降低了拼接字符串的速度

--每次进行拼接都会创建新的字符串对象，像java中的string一样，拼接后的字符串在池中没有。Java使用stringbuffer（可变对象）,stringbuilder来解决这个问题。而lua则使用表（table）的concat（模拟）解决

例子：

local s = ''

local t = {}

for i = 1,300000 do

t[#t + 1] = 'a'

end

s = table.concat( t, '')

就是把要拼接的字符串存储进table，再使用concat连接

结论：在大字符串连接中，我们应避免..。应用 table 来模拟 buffer，然后 concat 得到最终字符串。

### 四，3R

3R 原则（the rules of 3R）是：减量化（reducing），再利用（reusing）和再循

环（recycling）三种原则的简称。

**reducing**减量化

例子：polyline = {

{ x = 1.1, y = 2.9 },

{ x = 1.1, y = 3.7 },

{ x = 4.6, y = 5.2 },

...

}

以上的数据结构十分自然，便于理解。但是每一个顶点都需要一个哈希部分来储存。

如果放置在数组部分中，则会减少内存的占用: 因为再哈希代价大之前说过

1

2

3

4

5

polyline = {

{ 1.1, 2.9 },

{ 1.1, 3.7 },

{ 4.6, 5.2 },

缺点是：可读性差了

这个很好理解

for i=1,n do

local t = {1,2,3,'hi'}

--执行逻辑，但 t 不更改

...

end

我们应该把在循环中不变的东西放到循环外来创建：

1

2

3

4

5

local t = {1,2,3,'hi'}

for i=1,n do

--执行逻辑，但 t 不更改

...

end

**再利用reusing**

如果无法避免创建新对象，我们需要考虑重用旧对象。

例子：

local t = {}

for i = 1970, 2000 do

t[i] = os.time({year = i, month = 6, day = 14})

end

在每次循环迭代中，都会创建一个新表{year = i, month = 6, day = 14}，但是只

有 year 是变量。

下面这段代码重用了表：

local t = {}

local aux = {year = nil, month = 6, day = 14}

for i = 1970, 2000 do

aux.year = i;

t[i] = os.time(aux)

end

**再循 环recycling**

Lua 自带垃圾回收器，所以我们一般不需要考虑垃圾回收的问题。

了解 Lua 的垃圾回收能使得我们编程的自由度更大。这个后面可以再看，现在用不到

**九、结语**

我们应该在写代码时，按照高标准去写，尽量避免在事后进行优化。

如果真的有性能问题，我们需要用工具量化效率，找到瓶颈，然后针对其优化。当

然优化过后需要再次测量，查看是否优化成功。

在优化中，我们会面临很多选择：代码可读性和运行效率，CPU 换内存，内存换

CPU 等等。需要根据实际情况进行不断试验，来找到最终的平衡点。

第一、使用 LuaJIT，LuaJIT 可以使你在不修改代码的情况下获得平均约 5 倍的加

速。查看 LuaJIT 在 x86/x64 下的性能提升比。

第二、将瓶颈部分用 C/C++来写。因为 Lua 和 C 的天生近亲关系，使得 Lua 和 C 可以混合编程。但是 C 和 Lua 之间的通讯会抵消掉一部分 C 带来的优势。

注意：这两者并不是兼容的，你用 C 改写的 Lua 代码越多，LuaJIT 所带来的优化

幅度就越小。