(http://twitter.com/kenlistdev) (http://github.com/kenlist)

(http://geekluo.com/) (mailto:rijian.lrj@alibaba-inc.com)

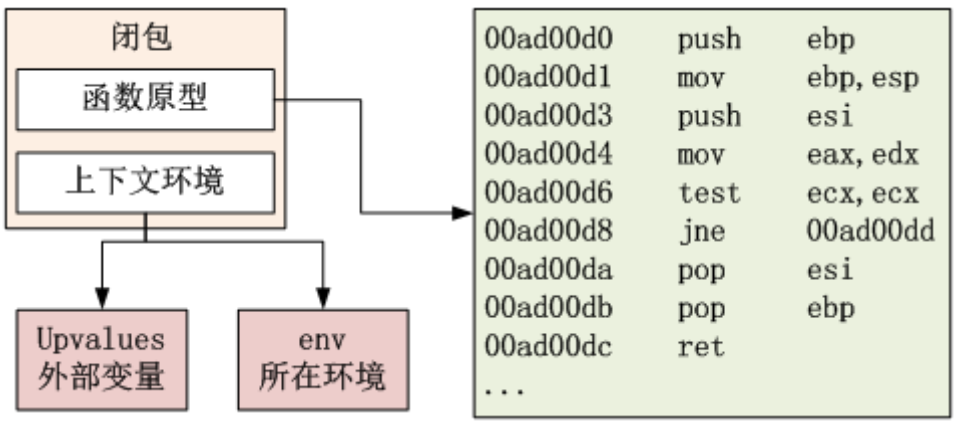
Lua数据结构 -- 闭包（四）

2014/04/12

前面几篇文章已经说明了Lua里面很常用的几个数据结构，这次要分享的也是常用的数据结构之一 – **函数的结构。**函数在Lua里也是一种变量，但是它却很特殊，能存储执行语句和被执行，本章主要描述Lua是怎么实现这种函数的。

在脚本世界里，相信**闭包**这个词大家也不陌生，闭包是由函数与其相关引用环境组成的实体。可能有点抽象，下面详细说明：

##一、 闭包的组成



闭包主要由以下2个元素组成：

1. **函数原型**：上图意在表明是一段可执行代码。在Lua中可以是lua_CFunction，也可以是lua自身的虚拟机指令。
2. **上下文环境**：在Lua里主要是Upvalues和env，下面会有说明Upvalues和env。 在Lua里，我们也从闭包开始，逐步看出整个结构模型，下面是Closure的数据结构：(lobject.h 291-312)

```
#define ClosureHeader \
        CommonHeader; lu_byte isC; lu_byte nupvalues; GCObject *gclist; \
        struct Table *env

typedef struct CClosure {
  ClosureHeader;
  lua_CFunction f;
  TValue upvalue[1];
} CClosure;

typedef struct LClosure {
  ClosureHeader;
  struct Proto *p;
  UpVal *upvals[1];
} LClosure;

typedef union Closure {
  CClosure c;
  Lclosure l;
} Closure;
```

不难发现，Lua的闭包分成2类，一类是CClosure，即**luaC函数的闭包。**另一类是LClosure，是**Lua里面原生的函数的闭包。**下面先讨论2者都有相同部分ClosureHeader：

1. CommonHeader：和与TValue中的GCHeader能对应起来的部分
2. isC：是否CClosure
3. nupvalues：外部对象个数
4. gclist：用于GC销毁，超出本章话题，在GC章节将详细说明
5. env：函数的运行环境，下面会有补充说明

对于CClosure数据结构：

1. lua_CFunction f：函数指针，指向自定义的C函数
2. TValue upvalue[1]：C的闭包中，用户绑定的任意数量个upvalue

对于LClosure数据结构：

1. Proto *p：Lua的函数原型，在下面会有详细说明
2. UpVal *upvals：Lua的函数upvalue，这里的类型是UpVal，这个数据结构下面会详细说明，这里之所以不直接用TValue是因为具体实现需要一些额外数据。

##二、 闭包的UpVal实现

究竟什么是UpVal呢？先来看看代码：

```
function FunA(a)
    local c = 10
    function FuncB(b)
        return a+b+c
    end
    return FuncB
end

local testA = FuncA(3)
Local testB = testA(5)
```

分析一下上面这段代码，最终testB的值显然是3+5+10=18。当调用testA(5)的时候，其实是在调用FuncB(5)，但是这个FuncB知道a = 3，这个是由**FuncA调用时**，记录到FuncB的**外部变量**，我们把a和c称为FuncB的upvalue。那么Lua是如何实现upvalue的呢？ 以上面这段代码为例，从虚拟机的角度去分析实现流程：

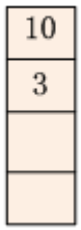
####1) FuncA(3)执行流程

- 把3这个常量放到栈顶，执行FuncA

虚拟机操作：（帮助理解，与真实值有差别）

```
LOADK top 3                //把3这个常量放到栈顶
CALL  top FuncA nresults   //调用对应的FuncA函数
```

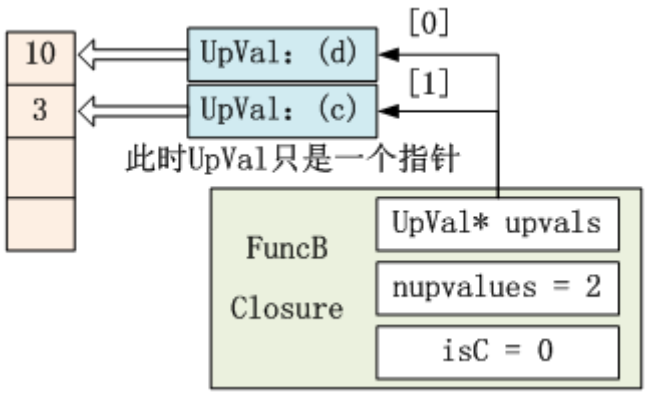
- 虚拟机的**pc已经在FuncA里面**了，FuncA中的**局部变量都是放到栈中的**，所以第一句loacl c = 10是把10放到栈顶（这里假设先放到栈顶简化一些复杂细节问题，下同）



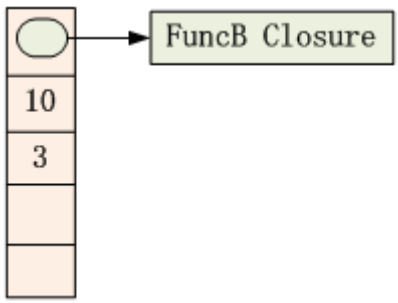
虚拟机操作：

```
LOADK top 10                          //local c = 10
```

- 遇到Function FuncB这个语句，会**生成FuncB的闭包**，这个过程同时会绑定upval到这个闭包上，但这是值还在栈上，**upval只是个指针。**



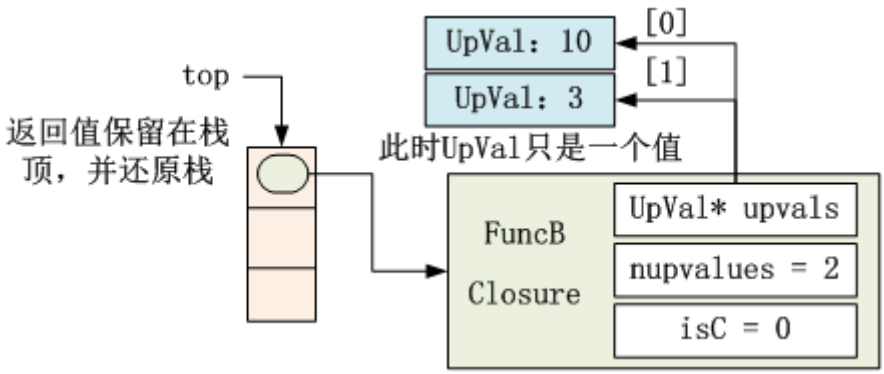
上面生成一个闭包之后，因为在Lua里，函数也是一个变量，上面的语句等价于**local FuncB = function() … end**，所以也会生成一个临时的FuncB到栈顶。



虚拟机操作：

```
NEWCLOSURE ra preclosure  //ra寄存器指定的位置就是栈顶，用来存放生成的      FuncB Closure，preclos
ure是上一个闭包，就是FuncA的闭包
GETUPVAL ra    // FuncB闭包的upvals添加一个指针指向d
GETUPVAL ra    // FuncB闭包的upvals添加一个指针指向c
```

- 最后return FuncB，就会把这个**闭包关闭**并返回出去，同时会把所有的upval进行unlink操作，让**upval本身保存值。**



虚拟机操作：

```
RETURN ra        // 当前闭包关闭并退出，清栈并把返回值ra放到栈顶
```

####2) FuncB的执行过程 到了FuncB执行的时候，参数b=5已经放到栈顶，然后执行FuncB。语句比较简单和容易理解，return a+b+c 虚拟机操作如下：

```
GETUPVAL rb 0        //获取当前闭包(FuncB)的upvals[0]，放到rb
LOADK ra stack[0]    //把栈顶的值放到ra，即参数b=5放到ra
ADD ra rb                        //ra = ra + rb
GETUPVAL rb 1              //获取当前闭包(FuncB)的upvals[1]，放到rb
ADD ra rb                        //ra = ra + rb
RETURN ra                        //闭包关闭，并把ra的值放到栈顶后返回
```

到这里UpVal的创建和使用也在上面给出事例说明，总结一下UpVal的实现：

- UpVal是在函数**闭包生成的时候（运行到function时）绑定**的。
- UpVal在**闭包还没关闭前**（即函数返回前），是**对栈的引用**，这样做的目的是可以在函数里修改对应的值从而修改UpVal的值，比如：

lua code:

```
function funcA()
        local d = 10
        function funcB()
                return d
        end
        d = 100
        local testB = funcB()
        return funcB
end
```

在上面的例子中，`testB = 100`，因为函数还没有关闭，所以在`funcB`查找`upVal`会找到对应修改后的引用

- **闭包关闭后**（即函数退出后），UpVal不再是指针，而是**值。** 知道UpVal的原理后，就只需要简要叙述一下UpVal的数据结构：（lobject.h 274 - 284）

```
typedef struct UpVal {
  CommonHeader;
  TValue *v;  /* points to stack or to its own
value */
  union {
    TValue value;  /* the value (when closed)
*/
    struct {  /* double linked list (when open)
*/
      struct UpVal *prev;
      struct UpVal *next;
    } l;
  } u;
} UpVal;
```

1. CommHeader： UpVal也是可回收的类型，一般有的CommHeader也会有
2. TValue* v：当函数打开时是指向对应stack位置值，当关闭后则指向自己
3. TValue value：函数关闭后保存的值
4. UpVal* prev、UpVal* next：用于GC，全局绑定的一条UpVal回收链表

##三、 函数原型

之前说的，函数原型是表明一段可执行的代码或者操作指令。在绑定到Lua空间的C函数，**函数原型就是lua_CFunction的一个函数指针**，指向用户绑定的C函数。下面描述一下**Lua中的原生函数的函数原型**，即**Proto数据结构**（lobject.h 231-253）：

```
/*
** Function Prototypes
*/
typedef struct Proto {
  CommonHeader;
  TValue *k;  /* constants used by the function
*/
  Instruction *code;
  struct Proto **p;  /* functions defined
inside the function */
  int *lineinfo;  /* map from opcodes to source
lines */
  struct LocVar *locvars;  /* information about
local variables */
  TString **upvalues;  /* upvalue names */
  TString  *source;
  int sizeupvalues;
  int sizek;  /* size of `k' */
  int sizecode;
  int sizelineinfo;
  int sizep;  /* size of `p' */
  int sizelocvars;
  int linedefined;
  int lastlinedefined;
  GCObject *gclist;
  lu_byte nups;  /* number of upvalues */
  lu_byte numparams;
  lu_byte is_vararg;
  lu_byte maxstacksize;
} Proto;
```

1. CommonHeader：Proto也是需要回收的对象，也会有与GCHeader对应的CommonHeader
2. TValue* k：函数使用的常量数组，比如local d = 10，则会有一个10的数值常量
3. Instruction *code：虚拟机指令码数组
4. Proto **p：函数里定义的函数的函数原型，比如funcA里定义了funcB，在funcA的5. Proto中，这个指针的[0]会指向funcB的Proto
5. int *lineinfo：主要用于调试，每个操作码所对应的行号
6. LocVar *locvars：主要用于调试，记录每个本地变量的名称和作用范围
7. TString **upvalues：一来用于调试，二来用于给API使用，记录所有upvalues的名称
8. TString *source：用于调试，函数来源，如c:\t1.lua@ main
9. sizeupvalues： upvalues名称的数组长度
10. sizek：常量数组长度
11. sizecode：code数组长度
12. sizelineinfo：lineinfo数组长度
13. sizep：p数组长度
14. sizelocvars：locvars数组长度
15. linedefined：函数定义起始行号，即function语句行号
16. lastlinedefined：函数结束行号，即end语句行号
17. gclist：用于回收
18. nups：upvalue的个数，其实在Closure里也有nupvalues，这里我也不太清楚为什么要弄两个，nups是语法分析时会生成的，而nupvalues是动态计算的。
19. numparams：参数个数
20. is_vararg：是否参数是”…”（可变参数传递）
21. maxstacksize：函数所使用的stacksize

Proto的所有参数都是在**语法分析和中间代码生成时获取的**，相当于编译出来的汇编码一样是不会变的，动态性是在Closure中体现的。

##四、 闭包运行环境

在前面说到的闭包数据结构中，有一个成员env，是一个Table*指针，用于指向当前闭包运行环境的Table。

什么是闭包运行环境呢？以下面代码举例：

```
function funcA()
        d = 20                  //在运行环境的table中设置d变量，即env[“d”] = 20
        local d = 50    //在本地创建一个变量叫d
        local c = d             //由于函数创建了本地变量d，所以会在本地读取d变量
end
```

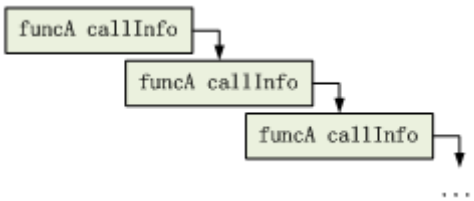
上面代码中的d = 20，其实就是在**环境变量中取env[“d”]**，所以env一定是个table，而当定义了本地变量之后，之后的所有变量都对从本地变量中操作。

##五、 函数调用信息

函数调用相当于一个**状态信息**，每次函数调用都会生成一个状态，比如递归调用，则会有一个栈去记录每个函数调用状态信息，比如说下面这段没有意义的代码：

```
function funcA()
        funcA()
end
```

那么每次调用将会生成一个调用状态信息，上面代码会无限生成下去：

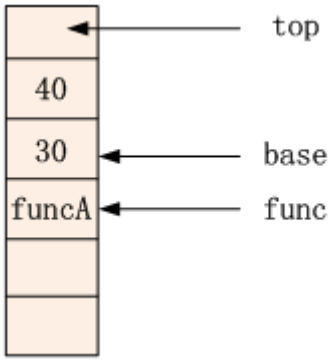


究竟一个CallInfo要记录哪些状态信息呢？下面来看看CallInfo的数据结构：

```
/*
** informations about a call
*/
typedef struct CallInfo {
  StkId base;  /* base for this function */
  StkId func;  /* function index in the stack
*/
  StkId  top;  /* top for this function */
  const Instruction *savedpc;
  int nresults;  /* expected number of results
from this function */
  int tailcalls;  /* number of tail calls lost
under this entry */
} CallInfo;
```

1. Instruction *savedpc：如果这个调用被中断，则用于记录当前闭包执行到的pc位置
2. nresults：返回值个数，-1为任意返回个数
3. tailcalls：用于调试，记录尾调用次数信息，关于尾调用下面会有详细解释

4. base、func、top：如下：



##六、 函数调用的栈操作

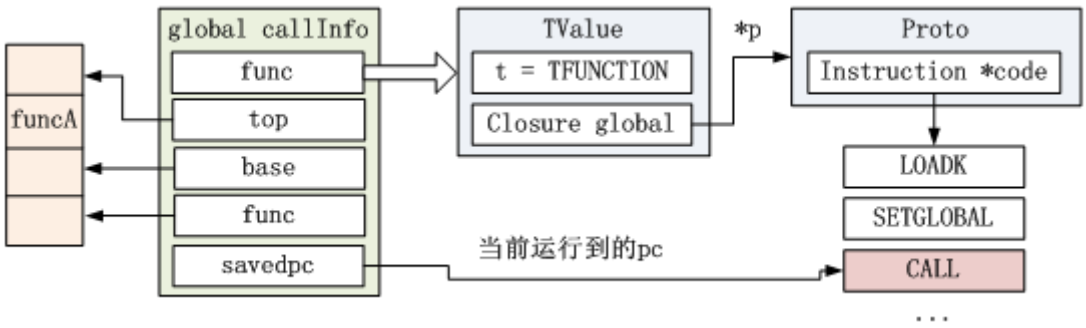
上面描述的CallInfo信息，具体整个流程是怎么走的，结合下面代码详细地叙述整个调用过程，栈是怎么变化的：

```
function global()
        function funcA(a, b)
                return a + b + 10
        end
        funcA(30, 40)
end
```

假设现在走到了funcA(30, 40)这个语句，在执行前已经存在了global这个闭包和funcA这个闭包，在调用global这个闭包时，已经生成了一个global的CallInfo。

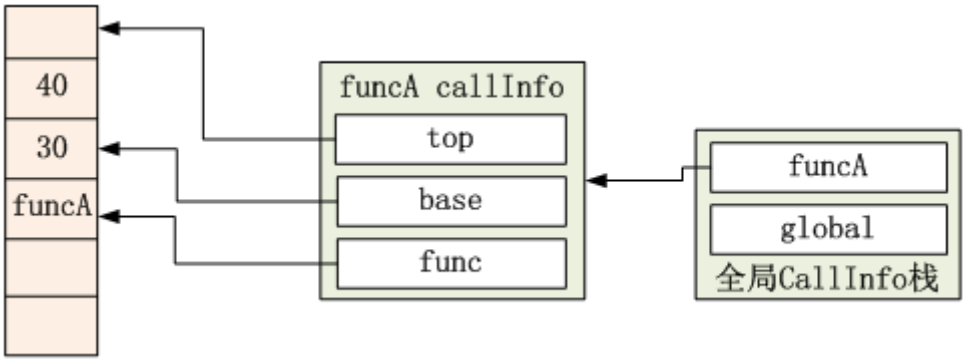
####1） 函数调用的栈操作：（OP_CALL lvm.c 582-601）

- global的CallInfo信息记录，并把funcA放到栈顶

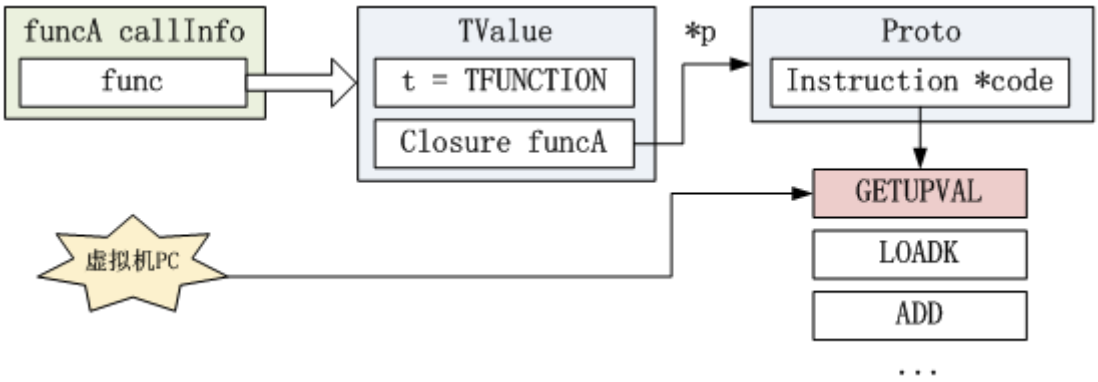


当前虚拟机的pc指针，指向global函数原型中的CALL指令，这时global的CallInfo的savedpc就会保存当前pc。然后会把要执行的funcA的闭包放到栈顶。

- 参数分别放到栈顶（从左到右分别进栈），生成funcA的CallInfo，并把完成对应CallInfo栈操作

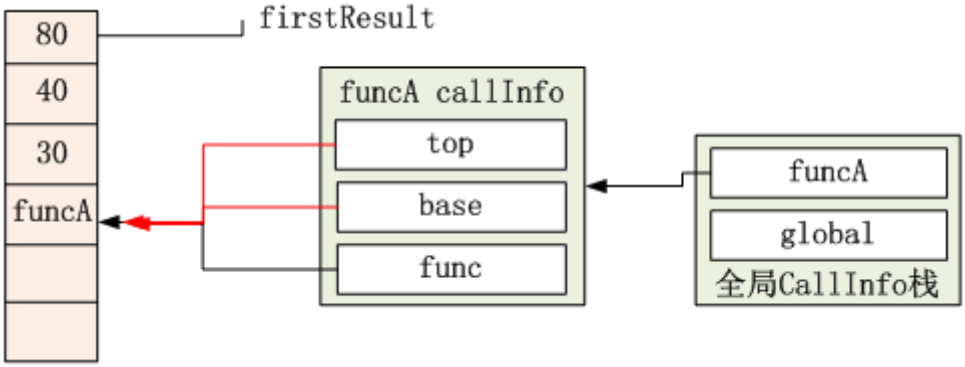


- 设置虚拟机pc到funcA闭包第一条虚拟机Instruction，并继续执行虚拟机

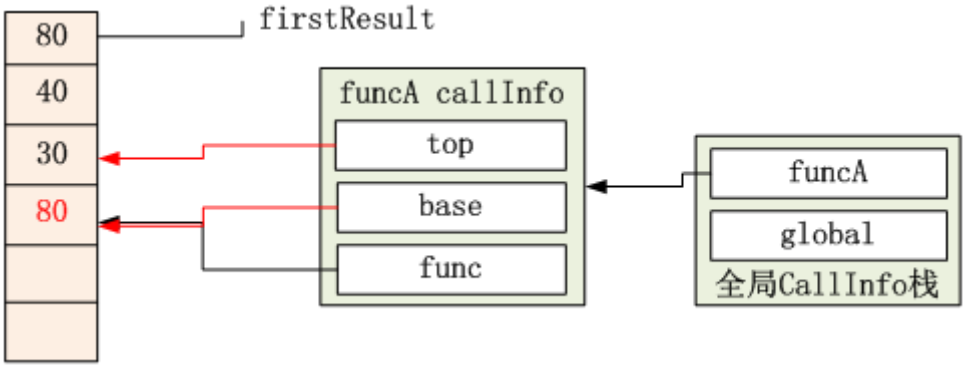


####2） 函数返回的栈操作：（OP_RETURN lvm.c 635-648）

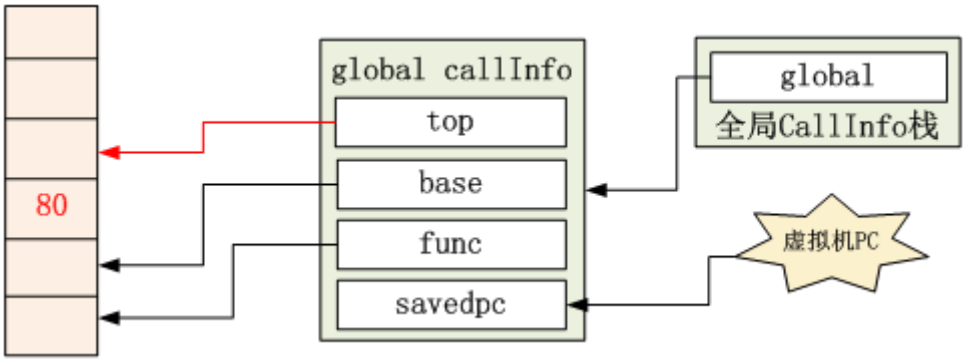
- 记录第一个返回值的位置到firstResult，把栈中的funcA位置设置为base和top



- 把返回值根据nresult参数重新push到栈



- 从全局CallInfo栈弹出funcA，并还原虚拟机pc到global的savedpc和栈信息



- 继续执行虚拟机

##七、 尾调用（TAILCALL）

看看下面这段经典的累加代码：

```
function Recursion(a)
        if a == 1 then
        return 1
        end
        return Recursion(a - 1) + a
end 如果是Recursion(10)的话，能正确计算出1+2+…+10的值，但是如果是Recursion(20000)，这样毫无疑问会导致**stack overflow**。
```

就像上一节说到的，每个递归调用会**生成一个CallInfo**，全局CallInfo栈的大小是有限的，基于乘2增长可以知道**lua的栈最大深度是16834（2^14）**：

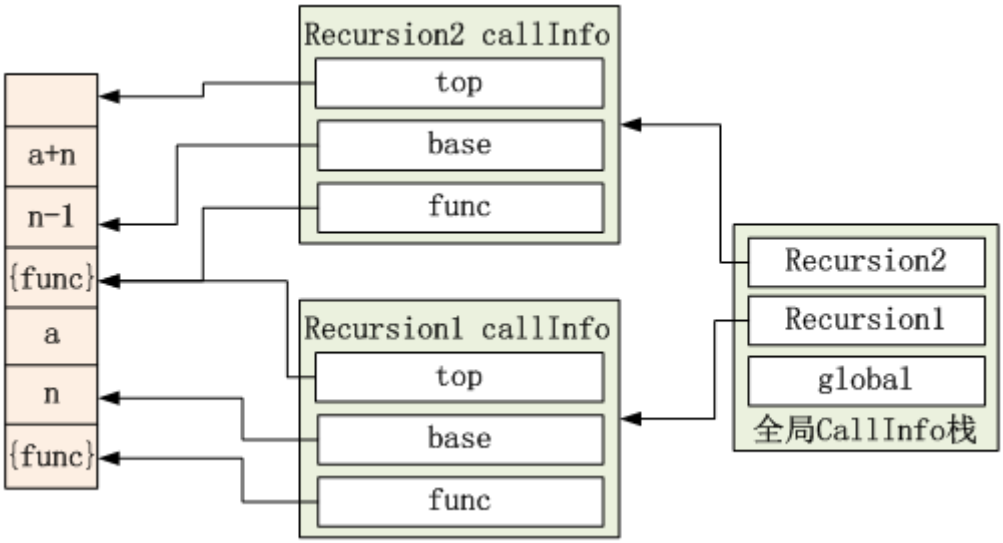
```
#define LUAI_MAXCALLS   20000 （luaconf.h 435） **尾调用是一种对函数解释的优化方法**，对于上面代码，改造成下面代码后，则不会出现stack overflow：

function Recursion(n, a)
        if n == 1 then
        return a + 1
        end
        if a == nil then
        a = 0
        end
        return Recursion(n - 1, a + n)
end
```

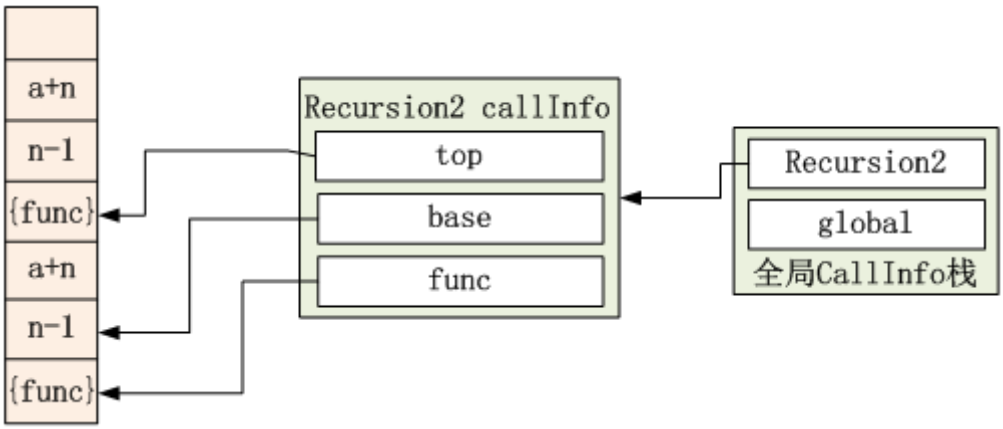
上面的Recursion方法不会出现stack overflow错误，也能顺利算出Recursion(20000) = 200010000。尾调用的使用方法十分简单，就是**在return后直接调用函数，不能有其它操作**，这样的写法即会进入尾调用方式。

那究竟lua是如何实现这种尾调用优化的呢？尾调用是在编译时分析出来的，有独立的操作码OP_TAILCALL，在虚拟机中的执行代码在lvm.c 603-634，具体原理如下：

1）首先像普通调用一样，准备调用Recursion函数



2）关闭Recursion1的调用状态，把Recursion2的对应栈数据下移，然后重新执行



本质优化思想：**先关闭前一个函数，销毁CallInfo，再调用新的CallInfo，这样就会避免全局CallInfo栈溢出。**

##八、 总结 本文讨论了闭包、UpVal、函数原型、环境、栈操作、尾调用等相关知识，基本上把大部分的知识点和细节也囊括了，另外还有2大块知识：函数原型的生成和闭包GC可能迟些再分享。

lua (6) (http://geekluo.com/tags.html#lua-ref)

← Previous (http://geekluo.com/contents/2014/04/11/4-lua-tstring-structure.html)

Next → (http://geekluo.com/contents/2014/04/12/6-lua-state-structure.html)

评论需要翻墙 for disqus