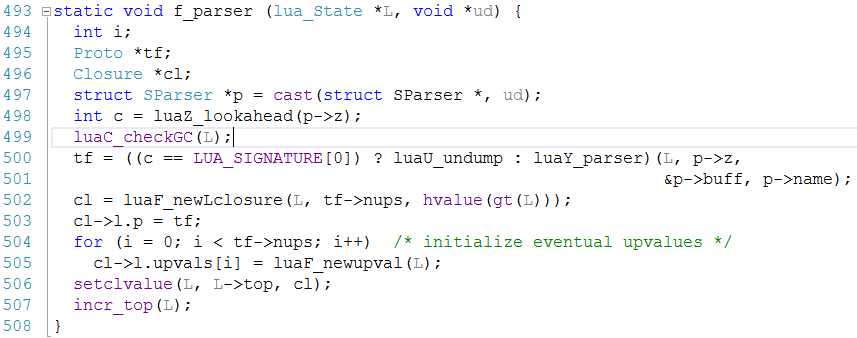
执行lua文件调用的是luaL\_dofile函数，它实际上是个宏，内部首先调luaL\_loadfile函数，再低啊用lua\_pcall函数：

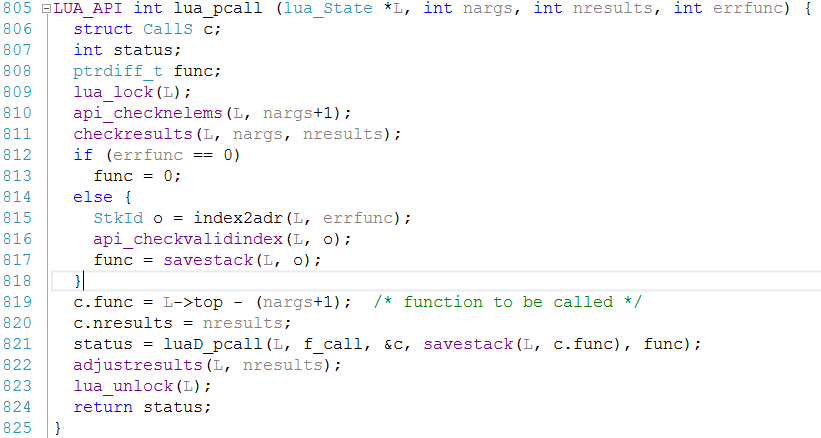
（lauxlib.h文件）



其中luaL\_loadfile函数用于进行词法和语法分析，lua\_pcall用于将第一步中分析的结果（也就是字节码）放到虚拟机中执行，下面收看看luaL\_loadfile函数，它最终会调用f\_parser函数，这是对代码进行分析的入口函数（ldo.c）：



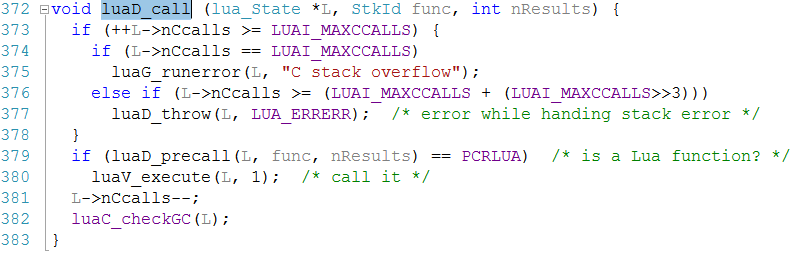
这里仅看函数对外的输出就可以看到：完成词法分析之后，返回了Proto类型的指针tf，然后将其绑定在新创建的Closure指针上，初始化Upvalu，最后压入栈中。不难想象，词法分析之后产生的字节码等相关数据都在这个Proto类型的结构体中，而这个结构又作为Closure保存了下来，留待下一步使用。接下来看看lua\_pcall函数是如何将产生的字节码放入虚拟机中执行的：



可以看到，首先获取需要调用的函数指针c.func = L->top - (nargs+1); /\* function to be called \*/，这里的nargs是由函数参数传入的，在luaL\_dofile中低啊用lua\_pcall时，这里传入的参数是0，换句话说，这里得到的函数对象指针就是前面f\_parser函数中最后两句代码放入lua栈的Closure指针：



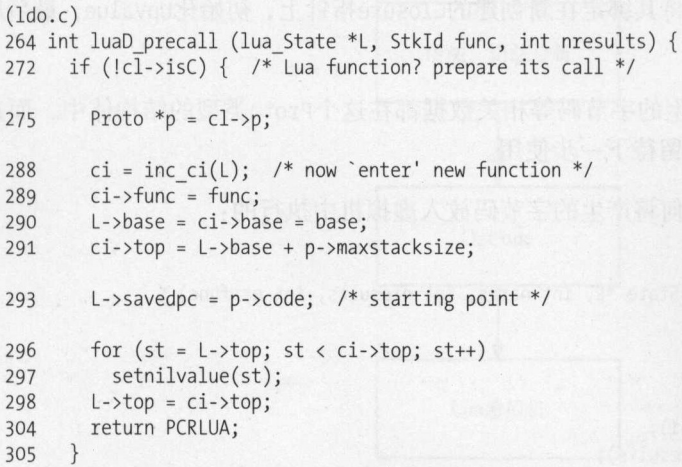
接下来会执行luaD\_pcall（ldo.c），最终会执行到luaD\_call（ldo.c）函数



其中我们看到下面这个调用：

if (luaD\_precall(L, func, nResults) == PCRLUA) /\* is a Lua function? \*/

luaV\_execute(L, 1); /\* call it \*/



将关键代码挑出来看：

1. 从Lua\_State的CallInfo驻足中得到一个新的CallInfo结构体，设置它的func、base、top指针
2. 从前面分析阶段生成的Closure指针中，去除保存下来的Proto结构体，前面提到过，这个结构体中保存的是分析过程完结之后生成的字节码等信息
3. 将这里创建的CallInfo指针的top/base指针赋值给lua\_State结构体的top、base指针，将Proto结构体的code成员赋值给lua\_State指针的savedpc字段，code成员保留的就是字节码
4. 将多余的函数参数赋值为nil，比如一个函数定义中需要的是两个参数，实际传入的只有一个，那么多出来的那个参数就会被赋值为nil

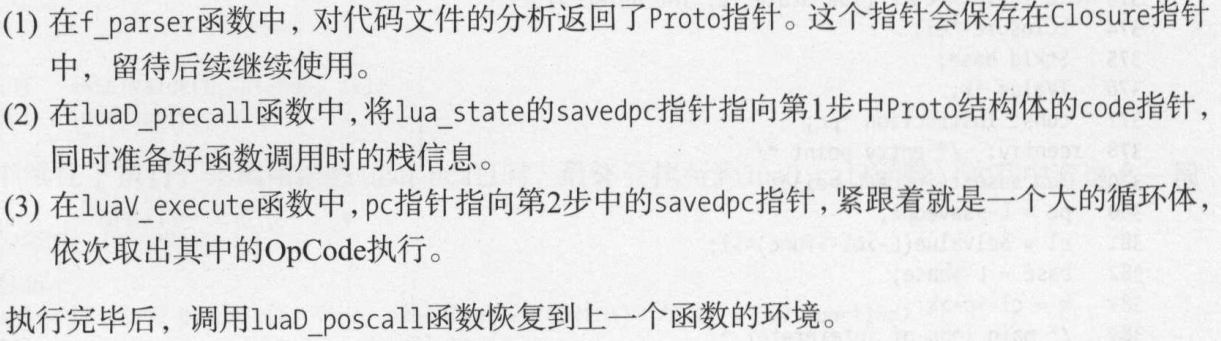
回到前面的luaD\_call函数，调用完luaD\_precall函数之后，就会进入luaV\_execute（lvm.c）函数，该函数的pc指存放的是虚拟机

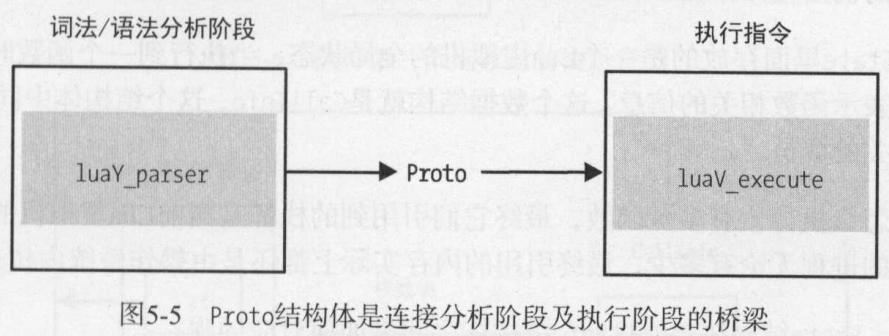
OpCode代码，它最开始从L->savepc初始化而来，而L->savepc在luaD\_precall中赋值：

（ldo.c）

L->savedpc = p->code /\* starting point \*/

可以看到，luaV\_execute函数最主要的作用就是一个大循环，将当前传入的指令一次执行，最后执行完毕后，还会调用luaD\_poscall函数恢复到上一次函数调用的环境：在OP\_RETURN的时候调用





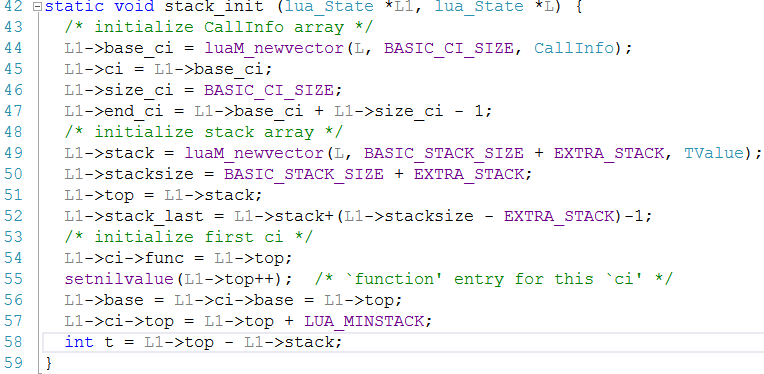
Lua解释器中的内存部分就存放在lua栈中，lua中也是把栈的某一个位置称为寄存器，每个lua虚拟机对应一个lua\_State结构体，它只用TValue数组来模拟栈，其中包括几个与栈相关的成员：

Stack：栈数组的起始位置

Base：当前函数栈的基地址

Top：当前栈的下一个可用位置

这些成员的初始化操作在stack\_init函数中完成，然而lua\_State里面存放在是一个Lua虚拟机的全局状态，当执行到一个函数时，需要有对应的数据结构来表示函数相关的信息，这个数据结构就是CallInfo，这个结构体中同样有top、base这两个与栈相关的成员。



无论函数怎么执行，有多少函数，最终它们引用到的栈都是单钱lua虚拟机的栈，这好比一个操作系统中的进程无论有多少，最终引用的内存实际上都还是由操作系统内核来管理的。那么，lua\_state结构体与CallInfo结构体之间是如何对应的呢？

在lua\_state中，有一个base\_ci的CallInfo数组，存储的信息就是CallInfo信息，而另一个ci成员，指向的就是当前函数的CallInfo指针。在调用函数之前，一般会调用luaD\_precall函数，它主要完成以下几个操作：

1. 保存当前虚拟机执行的指令savedpc到当前CallInfo的savedpc中，这里保存下来是为了后面调用完毕之后恢复执行
2. 分别计算出待调用函数的base、top值，这些值的计算依赖于函数的参数数量
3. 从lua\_State的base\_ci数组中分配一个新的CallInfo指针，存储前面两步计算出来的信息，切换到这个函数中准备调用。

可以看到，lua\_State结构体中的top、base指针是与函数执行相关的变量，在函数执行前后都会有变化，从下面的图中可以看到，两个函数

执行期间CallInfo指针分别指向lua\_State指针分配的栈数组的不同位置，而随着当前函数的变化，lua\_State结构体中的top和base指针的指向也发生了变化，这两个变量之中指向当前执行函数的对应位置。

