0726

About VM

原文

虚拟机基本概念

虚拟机指借助软件系统对物理机器指令执行进行的一种模拟。
首先，对于物理机器的执行，主要是机器从内存中fetch指令，通过总线传输到CPU，然后进行译码、执行、结果存储等步骤。既然虚拟机是对其进行的一种模拟，那么也逃不过以下几个特点：

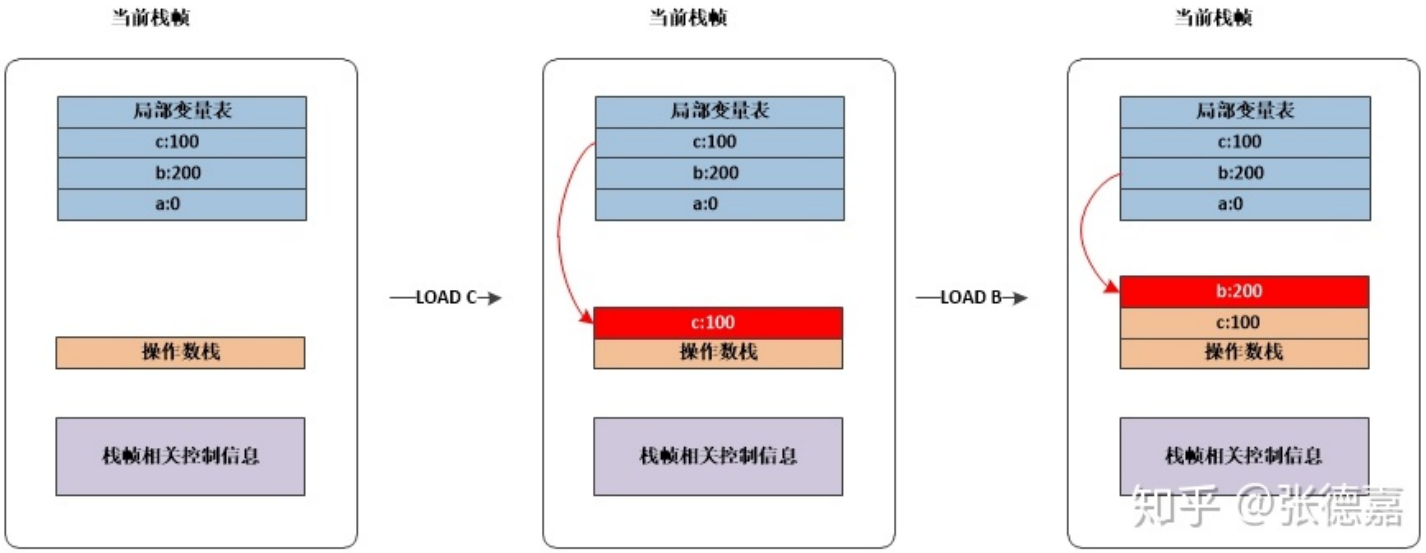
1. 将源码编译成VM所能执行的字节码。
2. 字节码格式(指令格式)，例如三元式、四元式、波兰式等。
3. 函数调用的相关栈结构，函数的出入口和传参方式。
4. 指令指针，类似于物理机的指令寄存器(EIP)
5. 虚拟CPU。 instruction dispatcher。
    1. 取指：通过IP fetch下一条指令
    2. 译码：对指令进行翻译，得到指令类型，并且解析其操作数
    3. 执行：跳到对应逻辑块进行执行。

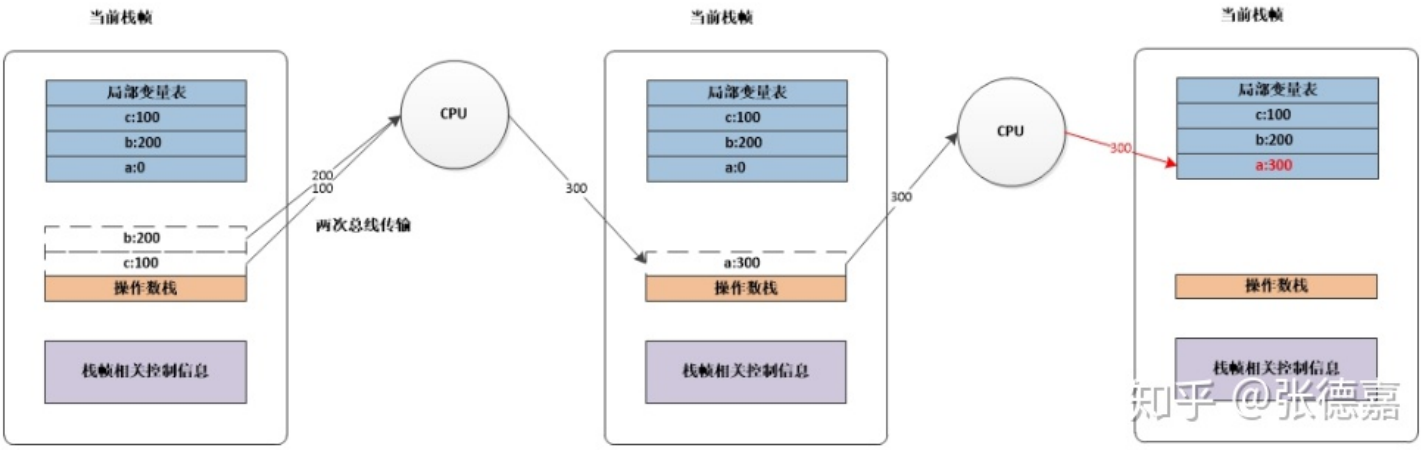
栈式虚拟机和寄存器式虚拟机

虽然虚拟机的实现都逃不过以上几步，但是以具体实现来看，又分为两大类：栈式和寄存器式。

栈式虚拟机

采用栈式虚拟机的语言有JVM、CPython以及.Net CLR等。 它的概念很简单，就是所有的指令执行，都是基于一个操作数栈的。你想要执行任何指令时，对不起，得先入栈，然后算完了再给我出栈。
流程如下图：





总的来说，就是抽象出了一个高度可移植的操作数栈，所有代码都会被编译成字节码，然后字节码就是在玩这个栈。 好处是实现简单，移植性强。坏处是指令条数比较多，数据转移次数比较多，因为每一次入栈出栈都牵涉数据的转移。

寄存器式虚拟机

采用寄存器式的虚拟机有lua和Dalvik等。 这种实现没有操作数栈这一概念，但是会有许多的虚拟寄存器。这类虚拟寄存器有别于CPU的寄存器，因为CPU寄存器往往是定址的(比如DX本身就是能存东西)，而寄存器式的虚拟机中的寄存器通常有两层含义：

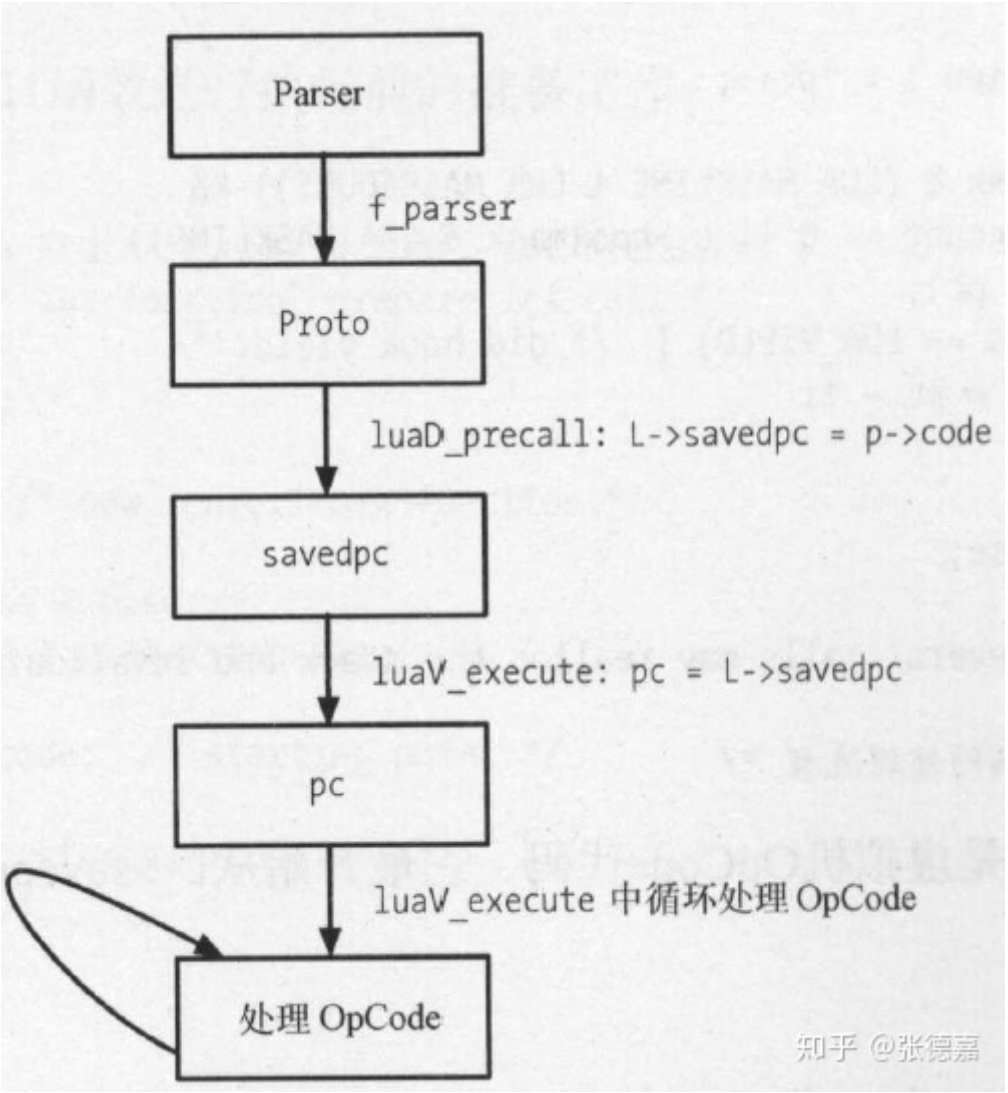
(1)寄存器别名(比如lua里的RA、RB、RC、RBx等)，它们往往只是起到一个地址映射的功能，它会根据指令中跟操作数相关的字段计算出操作数实际的内存地址，从而取出操作数进行计算；

(2)实际寄存器，有点类似操作数栈，也是一个全局的运行时栈，只不过这个栈是跟函数走的，一个函数对应一个栈帧，栈帧里每个slot就是一个寄存器，第1步中通过别名映射后的地址就是每个slot的地址。具体的栈帧可以参考后文讲CallInfo时的栈帧图。 好处是指令条数少，数据转移次数少。坏处是单挑指令长度较长。

具体来看，lua里的实际寄存器数组是用TValue结构的栈来模拟的，这个栈也是lua和C进行交互的虚拟栈。 lua里的字节码叫做opcode，本文正文将对"源码->字节码生成->字节码执行"这整个流程进行介

绍，并对其中的关键函数和数据结构进行源码级别的剖析。

虚拟机执行流程图



About lua

原文

为了达到较高的执行效率，lua代码并不是直接被Lua解释器解释执行，而是会先编译为字节码，然后再交给lua虚拟机去执行

lua代码称为chunk，编译成的字节码则称为二进制chunk（Binary chunk）

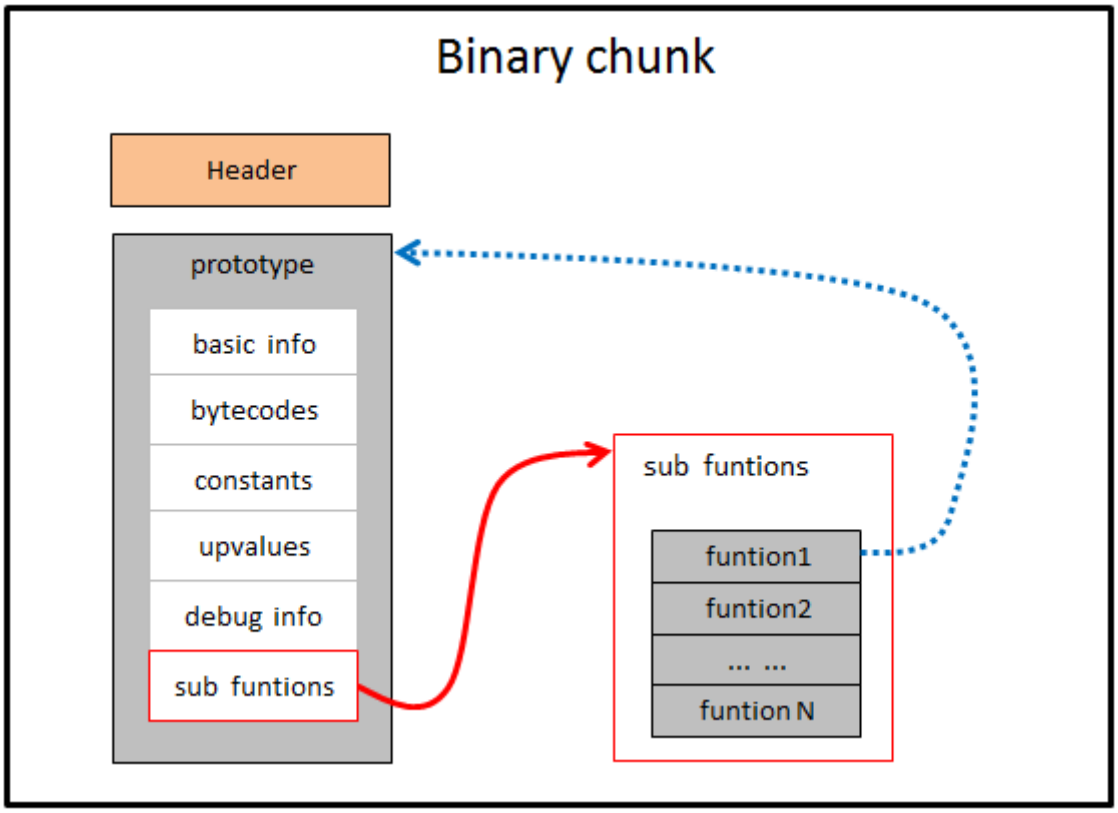
lua.exe、wlua.exe解释器可直接执行lua代码（解释器内部会先将其编译成字节码），也可执行使用luac.exe将lua代码预编译（Precompiled）为字节码

使用预编译的字节码并不会加快脚本执行的速度，但可以加快脚本加载的速度，并在一定程度上保护源代码

luac.exe可作为编译器，把lua代码编译成字节码，同时可作为反编译器，分析字节码的内容

```
luac.exe -v  // 显示luac的版本号
luac.exe Hello.lua  // 在当前目录下，编译得到Hello.lua的二进制chunk文件 luac.out（默认含调试符号）
luac.exe -o Hello.out Hello1.lua Hello2.lua // 在当前目录下，编译得到Hello1.lua和Hello2.lua的二进制
luac.exe -s -o d:\Hello.out Hello.lua  // 编译得到Hello.lua的二进制chunk文件d:\Hello.out（去掉调试
luac.exe -p Hello1.lua Hello2.lua  // 对Hello1.lua和Hello2.lua只进行语法检测（注：只会检查语法规则，
```

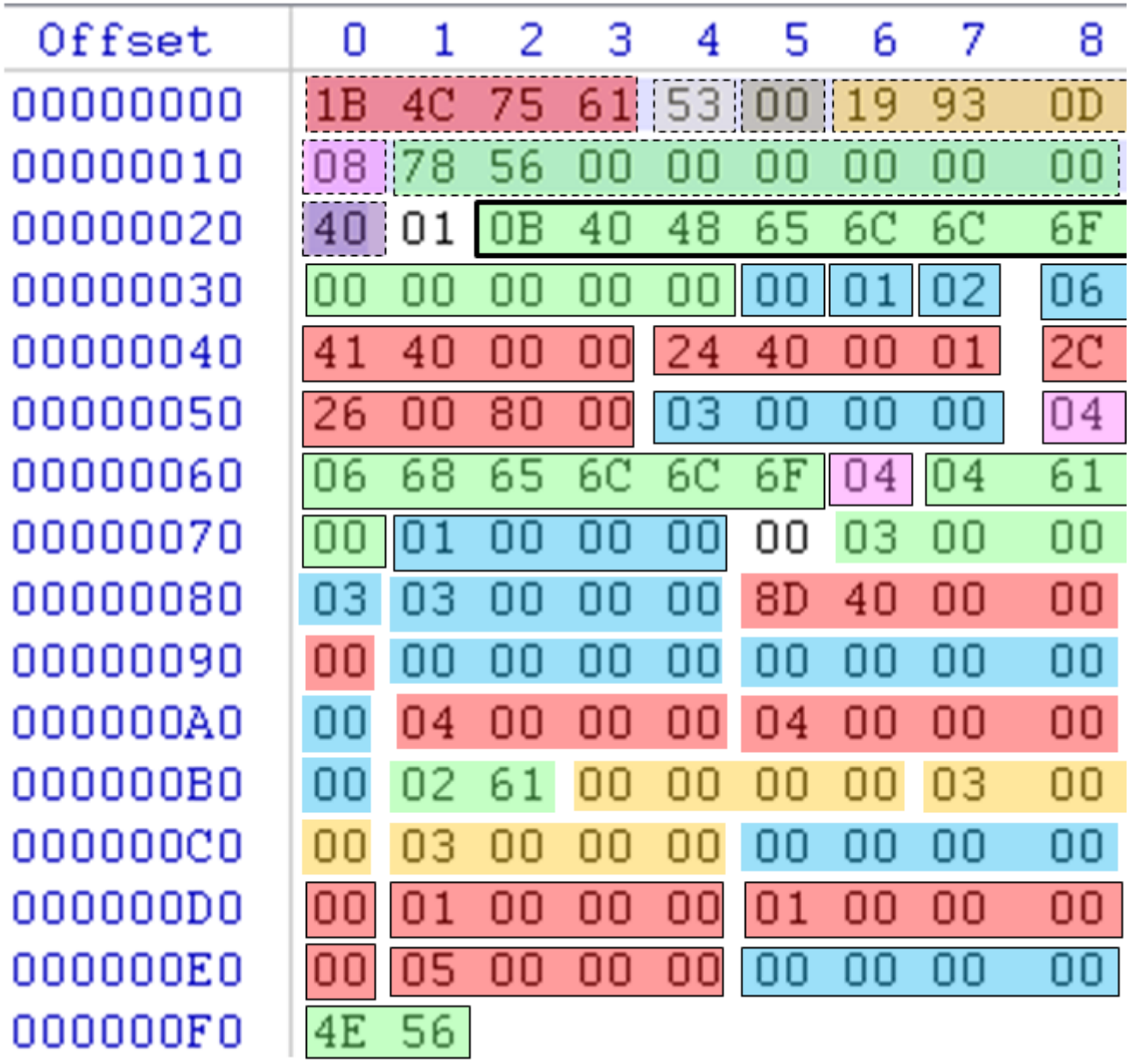
lua编译器以函数为单位对源代码进行编译，每个函数会被编译成一个称之为原型（Prototype）的结构
原型主要包含6部分内容：函数基本信息（basic info：含参数数量、局部变量数量等信息）、字节码（bytecodes）、常量（constants）表、upvalue（闭包捕获的非局部变量）表、调试信息（debug info）、子函数原型列表（sub functions）
原型结构使用这种嵌套递归结构，来描述函数中定义的子函数

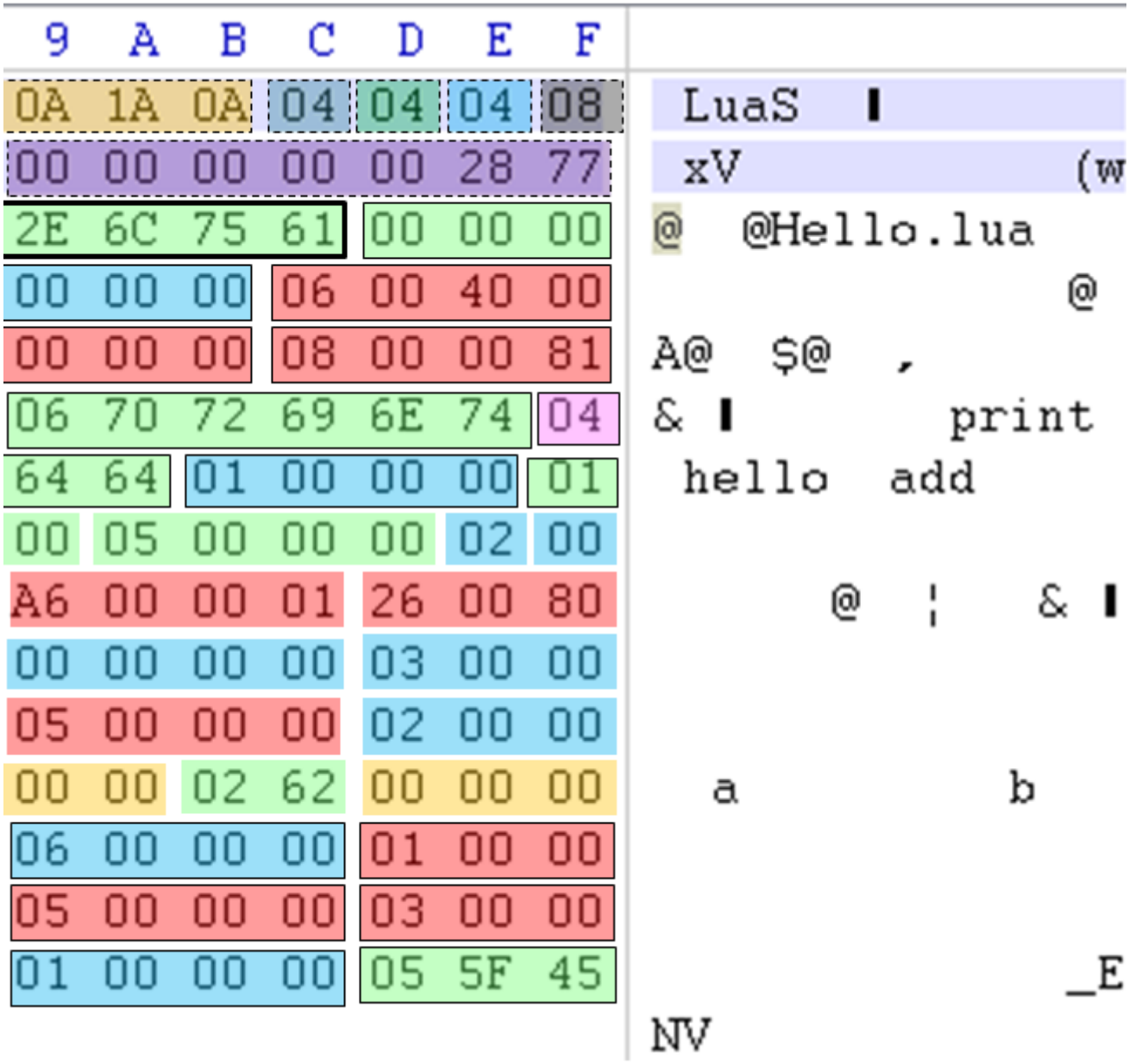


注：lua允许开发者可将语句写到文件的全局范围中，这是因为lua在编译时会将整个文件放到一个称之为main函数中，并以它为起点进行编译
Hello.lua源代码如下：

```
print ("hello")
function add(a, b)
   return a+b
end
```

编译得到的Hello.out的二进制为：


Offset 0 1 2 3 4 5 6 7 8
00000000 1B 4C 75 61 53 00 19 93 0D
00000010 08 78 56 00 00 00 00 00 00
00000020 40 01 0B 40 48 65 6C 6C 6F
00000030 00 00 00 00 00 00 01 02 06
00000040 41 40 00 00 24 40 00 01 2C
00000050 26 00 80 00 03 00 00 00 04
00000060 06 68 65 6C 6C 6F 04 04 61
00000070 00 01 00 00 00 00 03 00 00
00000080 03 03 00 00 00 8D 40 00 00
00000090 00 00 00 00 00 00 00 00 00
000000A0 00 04 00 00 00 04 00 00 00
000000B0 00 02 61 00 00 00 00 03 00
000000C0 00 03 00 00 00 00 00 00 00
000000D0 00 01 00 00 00 01 00 00 00
000000E0 00 05 00 00 00 00 00 00 00
000000F0 4E 56



二进制chunk（Binary chunk）的格式并没有标准化，也没有任何官方文档对其进行说明，一切以lua官方实现的源代码为准。

其设计并没有考虑跨平台，对于需要超过一个字节表示的数据，必须要考虑大小端（Endianness）问题。

lua官方实现的做法比较简单：编译lua脚本时，直接按照本机的大小端方式生成二进制chunk文件，当加载二进制chunk文件时，会探测被加载文件的大小端方式，如果和本机不匹配，就拒绝加载.

二进制chunk格式设计也没有考虑不同lua版本之间的兼容问题，当加载二进制chunk文件时，会检测其版本号，如果和当前lua版本不匹配，就拒绝加载

另外，二进制chunk格式设计也没有被刻意设计得很紧凑。在某些情况下，一段lua代码编译成二进制chunk后，甚至会被文本形式的源代码还要大。

预编译成二进制chunk主要是为了提升加载速度，因此这也不是很大的问题.