深入理解Lua虚拟机

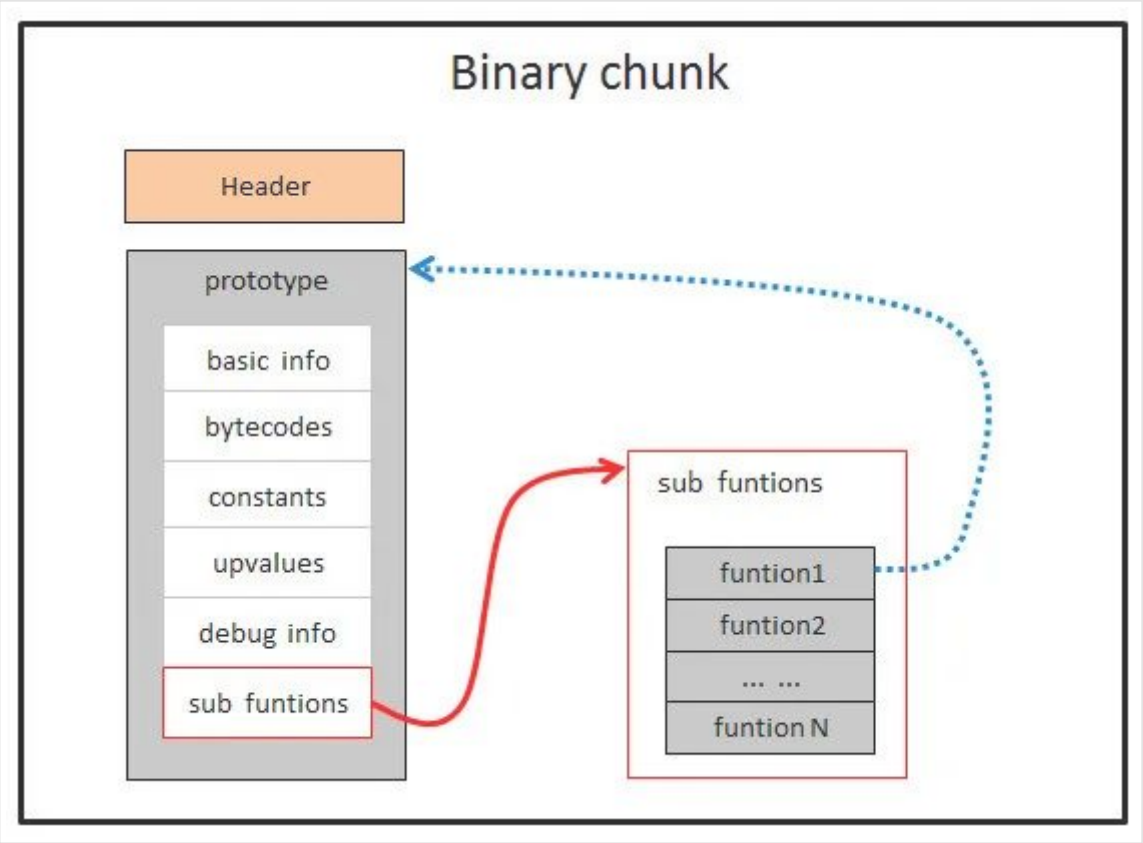
> 本文从一个简单示例入手，详细讲解 Lua 字节码文件的存储结构及各字段含义，进而引出 Lua 虚拟机指令集和运行时的核心数据结构 Lua State，最后解释 Lua 虚拟机的 47 条指令如何在 Lua State 上运作的。

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。

```
1  luac.exe -v  // 显示luac的版本号
2  luac.exe Hello.lua  //
3  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）
4
5  luac.exe -o Hello.out Hello1.lua Hello2.lua //
6  在当前目录下，编译得到Hello1.lua和Hello2.lua的二进制chunk文件Hello.out（默认含调试
7
8  luac.exe -s -o d:\\Hello.out Hello.lua  //
9  编译得到Hello.lua的二进制chunk文件d:\\Hello.out（去掉调试符号）
10
11 luac.exe -p Hello1.lua Hello2.lua  //
12 对Hello1.lua和Hello2.lua只进行语法检测（注：只会检查语法规则，不会检查变量、函数等
```

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：

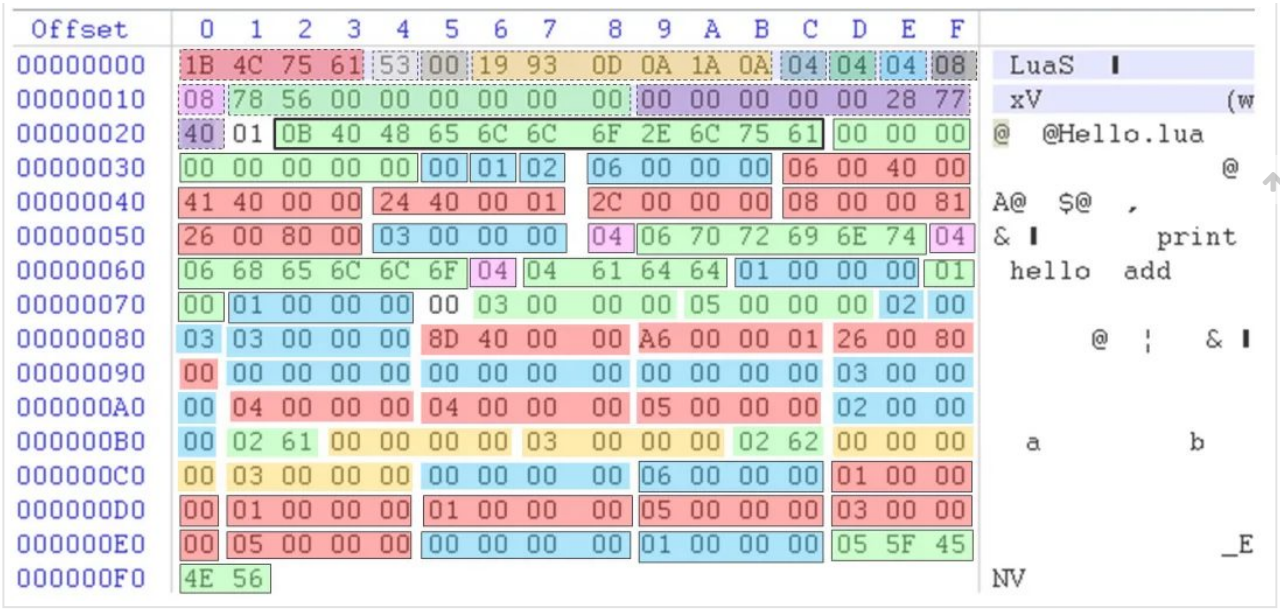


注：lua 允许开发者可将语句写到文件的全局范围中，这是因为 lua 在编译时会将整个文件放到一个称之为 main 函数中，并以它为起点进行编译。

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：



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。

头部字段：

字段	字节数	值	解释
签名（signature）	byte[4]	0x1B4C7561	二进制chunk文件的魔数，分别是ESC、L、u、a的ASCII码
版本号（version）	byte	0x53	lua语言的版本号由3部分组成：大版本号（Major Version）、小版本号（Minor Version）和发布号（Release Version） 比如：当前lua的版本号为5.3.5，那么大版本号为5，小版本号为3、发布号为5 由于发布号的增加只是为了修复bug，不会对二进制chunk文件格式进行调整，因此版本号（version）只存储了大版本号和小版本号
格式号（format）	byte	0x00	二进制chunk文件的格式号 lua虚拟机在加载二进制chunk时，会检查其格式号，如果和虚拟机本身的格式号不匹配，就会拒绝加载该文件
luacData	byte[6]	0x19930D0A1A0A	其中前两个字节0x1993，是lua1.0发布的年份；后四个字节依次是回车符（0x0D）、换行符（0x0A）、替换符（0x1A）和另一个换行符
cintSize	byte	0x04	cint数据类型size
sizeSize	byte	0x04	size_t数据类型size
instructionSize	byte	0x04	lua虚拟机指令size
luaIntegerSize	byte	0x08	lua整型size
luaNumberSize	byte	0x08	lua浮点数size
lua整数值（luacInt）	int64	0x7856000000000000	加载二进制chunk文件时，会用该字段检测其大小端和本机是否匹配
格式号（luacNum）	float64	0x0000000000287740	浮点数值为370.5 加载二进制chunk文件时，会用该字段检测其使用的浮点数格式

um)		287746	(目前主流平台一般都采用IEEE754浮点数格式)

嵌套的函数原型：

0B4048656C6C6F2E6C7561			源文件名Hello.lua；-s去掉调试信息后，该项数值为：00
	00000000 00000000		main函数起始行列号
	00		函数参数个数
	01		函数为不定参数
	02		寄存器数量
	06000000		函数指令数目
	06004000		第1条指令
	41400000		第2条指令
	24400001		第3条指令
	2C000000		第4条指令
	08000081		第5条指令
	26008000		第6条指令
	03000000		常量数目
	04		第1个常量tag，04表示字符串
	067072696E74		第1个常量内容
	04		第2个常量tag，04表示字符串
	0668656C6C6F		第2个常量内容
	04		第3个常量tag，04表示字符串
	04616464		第3个常量内容
	01000000		Upvalue数目
	0100		第1个Upvalue
	01000000		子函数原型数目
		03000000 05000000	add函数起始行列号
		02	函数参数个数
		00	函数为不定参数
		03	寄存器数量
		03000000	函数指令数目
		8D400000	第1条指令

main		A6000001	第2条指令
		26008000	第3条指令
		00000000	常量数目
		00000000	Upvalue数目
		00000000	子函数原型数目
		03000000	行号数目；-s去掉调试信息后，该项数值为：00000000；为0时下面的信息也都没有
	add	04000000	第1条指令行号
		04000000	第2条指令行号
		05000000	第3条指令行号
		02000000	局部变量数目；-s去掉调试信息后，该项数值为：00000000；为0时下面的信息也都没有
		0261	第1个局部变量名称
		00000000	第1个局部变量起始指令索引
		03000000	第1个局部变量终止指令索引
		0262	第2个局部变量名称
		00000000	第2个局部变量起始指令索引
		03000000	第2个局部变量终止指令索引
		00000000	Upvalue名称数目；-s去掉调试信息后，该项数值为：00000000；为0时下面的信息也都没有
	06000000		行号数目；-s去掉调试信息后，该项数值为：00000000；为0时下面的信息也都没有
	01000000		第1条指令行号
	01000000		第2条指令行号
	01000000		第3条指令行号
	05000000		第4条指令行号
	03000000		第5条指令行号
	05000000		第6条指令行号
	00000000		局部变量数目；-s去掉调试信息后，该项数值为：00000000；为0时下面的信息也都没有
	[illegible]		Upvalue名称数目；-s去掉调试信息后，该项数值为：

	01000000	00000000；为0时下面的信息也都没有
	055F454E56	第1个Upvalue名称

注 1：二进制 chunk 中的字符串分为三种情况：

1. ①NULL 字符串用 0x00 表示；
2. ② 长度小于等于 253（0xFD）的字符串，先用 1 个 byte 存储字符串长度+1 的数值，然后是字节数组；
3. ③ 长度大于等于 254（0xFE）的字符串，第一个字节是 0xFF，后面跟一个 8 字节 size_t 类型存储字符串长度+1 的数值，然后是字节数组。
4. 注 2：常量 tag 对应表

tag	lua字面量类型	存储类型
0x00	nil	不存储
0x01	boolean	字节（0、1）
0x03	number	lua浮点数
0x13	integer	lua整数
0x04	string	短字符串
0x14	string	长字符串

查看二进制 chunk 中的所有函数（精简模式）：

> luac.exe -l Hello.lua

> luac.exe -l Hello.out

```
main <Hello.lua:0,0> (6 instructions at 0046e528)
0+ params, 2 slots, 1 upvalue, 0 locals, 3 constants, 1 function
        1       [1]     GETTABUP        0 0 -1  ; _ENV "print"
        2       [1]     LOADK           1 -2    ; "hello"
        3       [1]     CALL            0 2 1
        4       [5]     CLOSURE         0 0     ; 0046e5b8
        5       [3]     SETTABUP        0 -3 0  ; _ENV "add"
        6       [5]     RETURN          0 1

function <Hello.lua:3,5> (3 instructions at 0046e5b8)
2 params, 3 slots, 0 upvalues, 2 locals, 0 constants, 0 functions
        1       [4]     ADD             2 0 1
        2       [4]     RETURN          2 2
        3       [5]     RETURN          0 1
```

> 注 1：每个函数信息包括两个部分：前面两行是函数的基本信息，后面是函数的指令列表。
> 注 2：函数的基本信息包括：函数名称、函数的起始行列号、函数包含的指令数量、函数地址。函数的参数 params 个数（0+表示函数为不固定参数）、寄存器 slots 数量、upvalue 数量、局部变量 locals 数量、常量 constants 数量、子函数 functions 数量。
> 注 3：指令列表里的每一条指令包含指令序号、对应代码行号、操作码和操作数。分号后为 luac 生成的注释，以便于我们理解指令。
> 注 4：整个文件内容被放置到了 main 函数中，并以它作为嵌套起点。

查看二进制 chunk 中的所有函数（详细模式）：

> luac.exe -l -l Hello.lua 注：参数为 2 个-l
> luac.exe -l -l Hello.out 注：详细模式下，luac 会把常量表、局部变量表和 upvalue 表的信息也打印出来

```
1   main <Test2.lua:0,0> (6 instructions at 0046e528)
2   0+ params, 2 slots, 1 upvalue, 0 locals, 3 constants, 1 function
3           序号     代码行     指令
4           1       [1]     GETTABUP        0 0 -1  ; _ENV "print"    //GETTAB
5           2       [1]     LOADK           1 -2    ; "hello"  //LOADK A Bx
6           3       [1]     CALL            0 2 1    ; //CALL A B C  //调用寄
7           4       [5]     CLOSURE         0 0     ; 0046e728      //CLOSURE
8           5       [3]     SETTABUP        0 -3 0  ; _ENV "add"    //SETTABUP
9           6       [5]     RETURN          0 1       ; //RETURN A B   //B:1
10  constants (3) for 0046e528:
```

```
        序号    常量名
        1       "print"
        2       "hello"
        3       "add"
locals (0) for 0046e528:
upvalues (1) for 0046e528:
        序号    upvalue名    是否为直接外围函数的局部变量    在外围函数调用帧
        0       _ENV        1                               0

function <Test2.lua:3,5> (3 instructions at 0046e728)
2 params, 3 slots, 0 upvalues, 2 locals, 0 constants, 0 functions
        序号    代码行    指令
        1       [4]     ADD             2 0 1    ; //ADD A B C  //将寄存器
        2       [4]     RETURN          2 2        ; //RETURN A B //B:2表
        3       [5]     RETURN          0 1        ; //RETURN A B //B:1表
constants (0) for 0046e728:
locals (2) for 0046e728:
    寄存器索引    起始指令序号   终止指令序号   -1得到实际指令序号
        0       a       1       4        ; a变量的指令范围为[0, 3]，起始为0
        1       b       1       4        ; b变量的指令范围为[0, 3]
upvalues (0) for 0046e728:
```

> luac.exe -l - // 从标准设备读入脚本，输完后按回车，然后按 Ctrl+Z 并回车，会打印出输入内容对应的二进制 chunk 内容 注：进入输入模式后可按 Ctrl+C 强制退出
>
> luac.exe -l – // 使用上次输入，打印出二进制 chunk 内容
>
> luac.exe -l -l – // 使用上次输入，详细模式下打印出二进制 chunk 内容（参数为 2 个-l）

Stack Based VM vs Rigister Based VM

高级编程语言的虚拟机是利用软件技术对硬件进行的模拟和抽象。按照实现方式，可分为两类：基于栈（Stack Based）和基于寄存器（Rigister Based）。Java、.NET CLR、Python、Ruby、Lua5.0 之前的版本的虚拟机都是基于栈的虚拟机；从 5.0 版本开始，Lua 的虚拟机改成了基于寄存器的虚拟机。

一个简单的加法赋值运算：a=b+c

基于栈的虚拟机，会转化成如下指令:

```
push b; // 将变量b的值压入stack
push c; // 将变量c的值压入stack
add; // 将stack顶部的两个值弹出后相加，然后将结果压入stack顶
mov a; // 将stack顶部结果放到a中
```

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。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。

基于寄存器的虚拟机，会转化成如下指令：

```
1   add a b c; // 将b与c对应的寄存器的值相加，将结果保存在a对应的寄存器中
```

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；

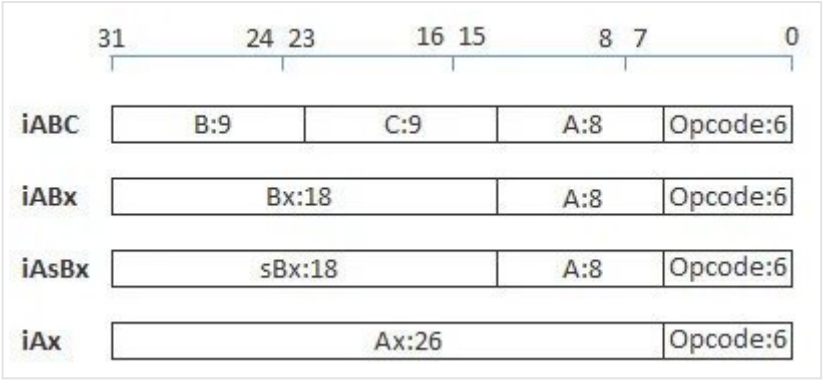
(2)实际寄存器，有点类似操作数栈，也是一个全局的运行时栈，只不过这个栈是跟函数走的，一个函数对应一个栈帧，栈帧里每个 slot 就是一个寄存器，第 1 步中通过别名映射后的地址就是每个 slot 的地址。

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 指令集

Lua 虚拟机的指令集为定长（Fixed-width）指令集，每条指令占 4 个字节（32bits），其中操作码（OpCode）占 6bits，操作数（Operand）使用剩余的 26bits。Lua5.3 版本共有 47 条指令，按功能可分为 6 大类：常量加载指令、运算符相关指令、循环和跳转指令、函数调用相关指令、表操作指令和 Upvalue 操作指令。

按编码模式分为 4 类：iABC（39）、iABx（3）、iAsBx（4）、iAx（1）



4 种模式中，只有 iAsBx 下的 sBx 操作数会被解释成有符号整数，其他情况下操作数均被解释为无符号整数。操作数 A 主要用来表示目标寄存器索引，其他操作数按表示信息可分为 4 种类型：OpArgN、OpArgU、OpArgR、OpArgK：

类型	示例	说明
OpArgN	B:1 C A:3 MOVE	不表示任何信息，不会被使用 MOVE指令的C操作数为OpArgN类型
OpArgR	B:1 C A:3 MOVE B:2 C:4 A:1 CONCAT	iABC指令中的B、C操作数，表示寄存器索引
	sBx:-3 A:0 FORLOOP	iAsBx指令中的sBx操作数，表示跳转偏移
OpArgK	Bx:2 A:4 LOADK	LOADK指令中的Bx操作数，表示常量表索引
	B:0x001 C:0x100 A:4 ADD	iABC指令中的B、C操作数 B、C操作数只能使用9bits中的低8位 最高位为1，表示常量表索引 最高位为0，表示寄存器索引
OpArgU	B:0 C:1 A:2 LOADBOOL	表示布尔值、整数值、Upvalue索引、子函数索引等

Lua 栈索引

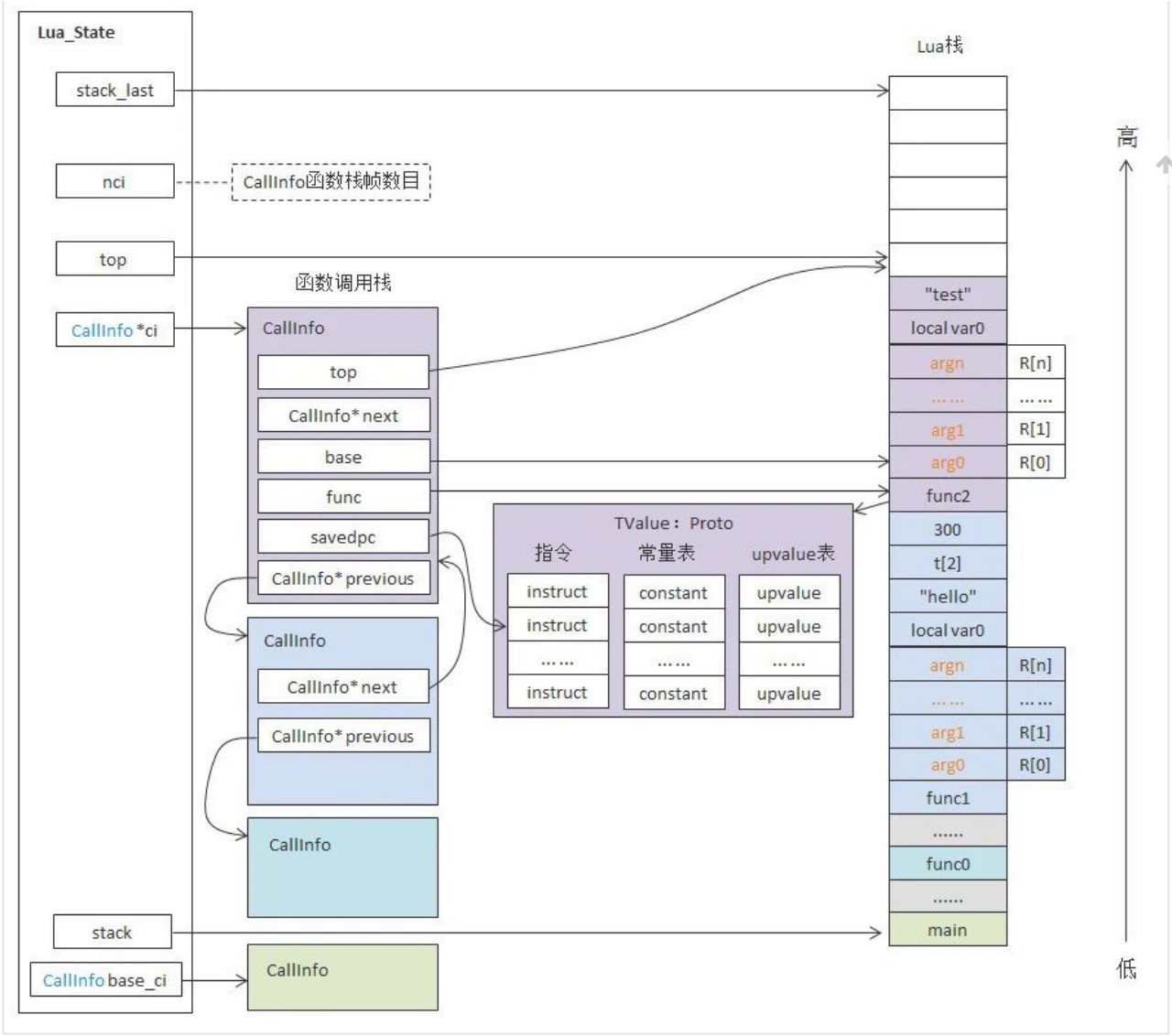
注 1：绝对索引是从 1 开始由栈底到栈顶依次增长的；

注 2：相对索引是从-1 开始由栈顶到栈底依次递减的（在 lua API 函数内部会将相对索引转换为绝对索引）；

注 3：上图栈的容量为 7，栈顶绝对索引为 5，有效索引范围为：[1,5]，可接受索引范围为：[1, 7]；

注 4：Lua 虚拟机指令里寄存器索引是从 0 开始的，而 Lua API 里的栈索引是从 1 开始的，因此当需要把寄存器索引当成栈索引使用时，要进行+1。

Lua State



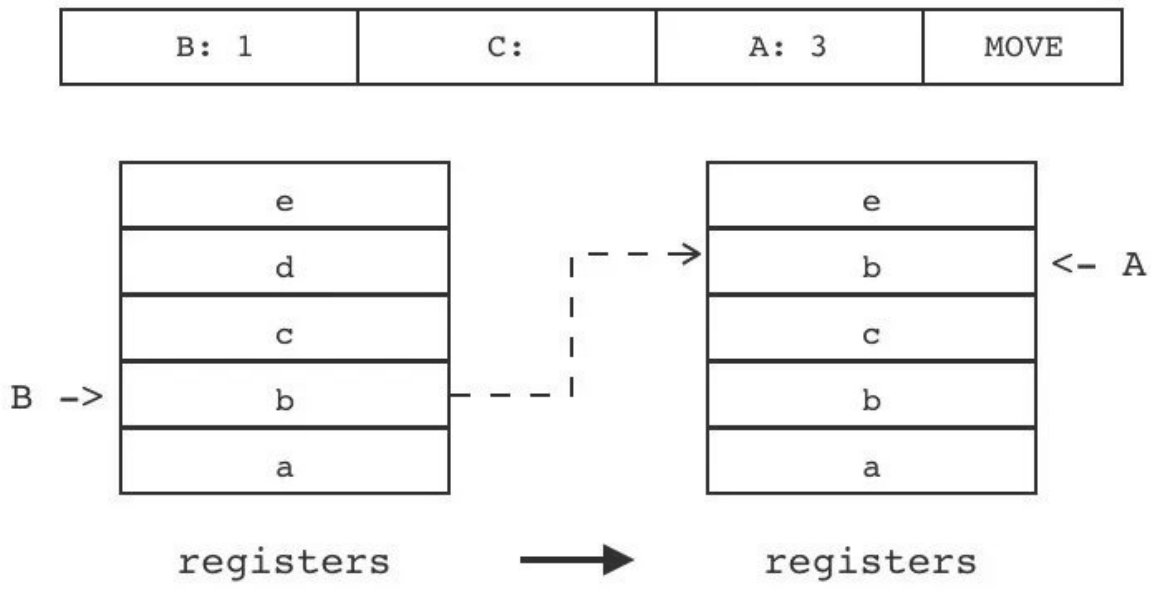
指令表

下面是 Lua 的 47 条指令详细说明：

指令名称	类型	操作码	B	C	A
MOVE	iABC	oxoo	OpArgR	OpArgN	目标寄存器idx

B:1 C A:3 MOVE

把源寄存器（索引由 B 指定）里的值移动到目标寄存器（索引有 A 指定），常用于局部变量赋值和参数传递。



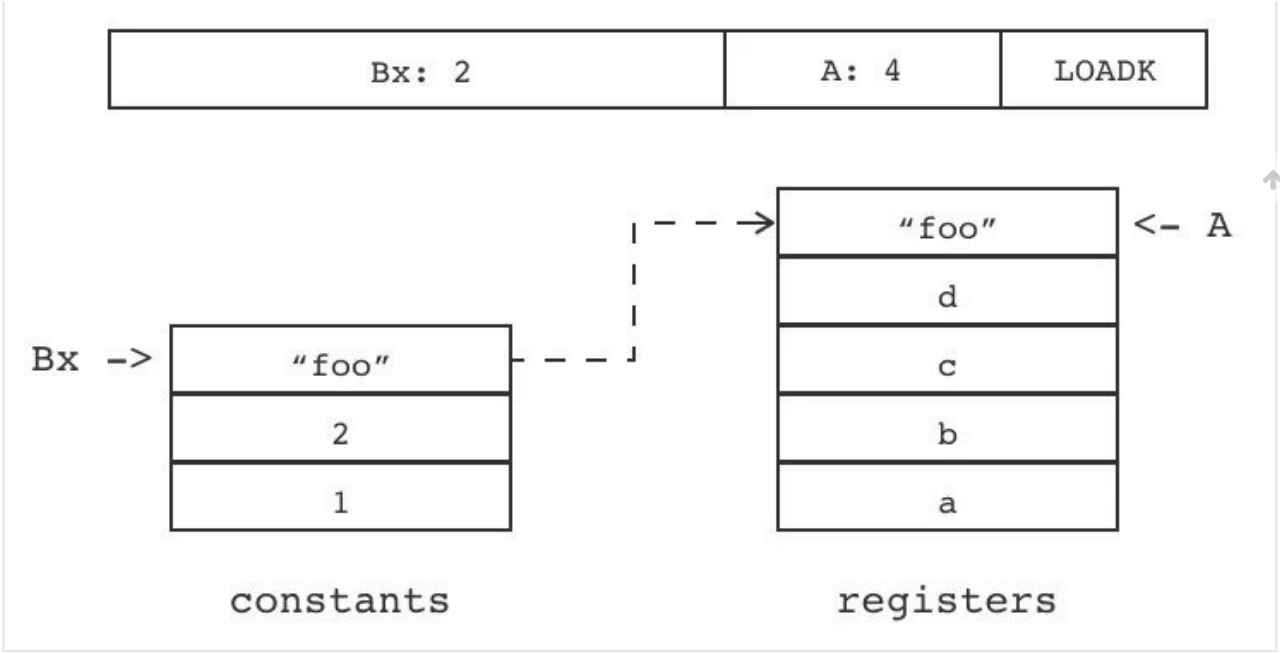
公式：R(A) := R(B)

指令名称	类型	操作码	Bx	A
LOADK	iABx	oxo1	OpArgK	目标寄存器idx

Bx:2 A:4 LOADK

给单个寄存器（索引由 A 指定）设置成常量（其在常量表的索引由 Bx 指定），将常量表里的某个常量加载到指定寄存器。

在 lua 中，数值型、字符串型等局部变量赋初始值 （数字和字符串会放到常量表中）：



公式：R(A) := Kst(Bx)

指令名称	类型	操作码	Bx	A
LOADKX	iABx	oxo2	OpArgN	目标寄存器idx

Bx A:4 LOADKX

Ax:585028 EXTRAARG

LOADK 使用 Bx（18bits，最大无符号整数为 262143）表示常量表索引。当将 lua 作数据描述语言使用时，常量表可能会超过这个限制，为了应对这种情况，lua 提供了 LOADKX 指令。LOADKX 指令需要和 EXTRAAG 指令搭配使用，用后者的 Ax（26bits）操作数来指定常量索引。

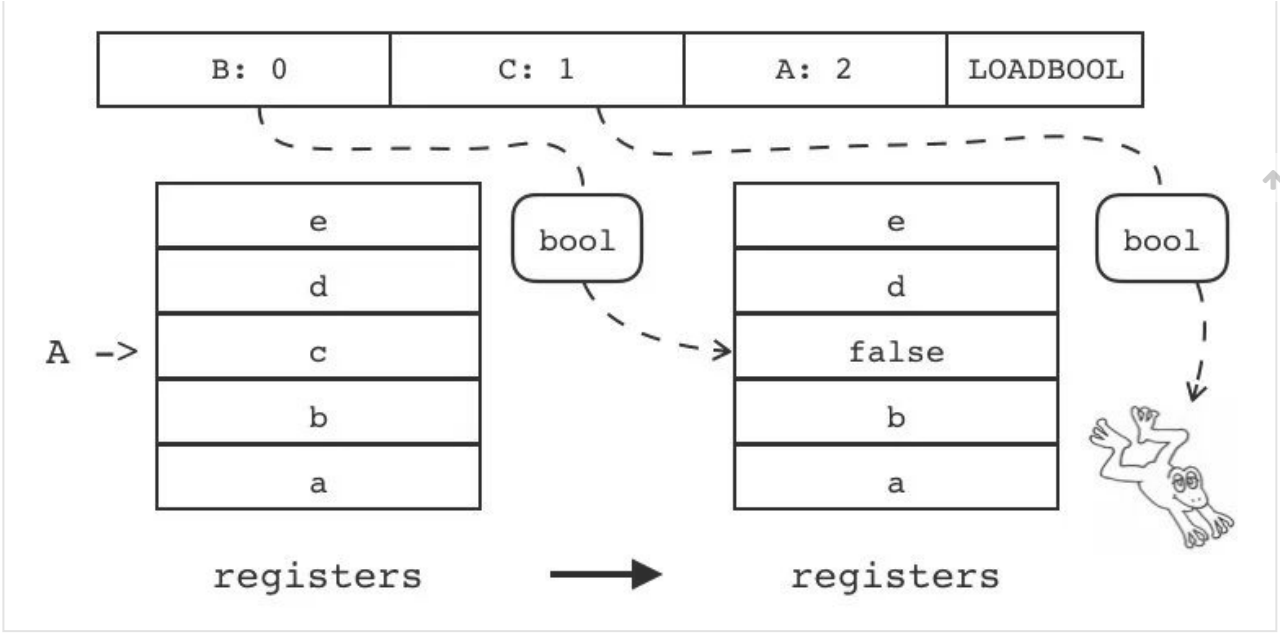
公式：R(A) := Kst(Ax)

指令名称类型操作码BCA

LOADBOOLiABC0x03OpArgUOpArgU目标寄存器 idx

B:0 C:1 A:2 LOADBOOL

给单个寄存器（索引由 A 指定）设置布尔值（布尔值由 B 指定），如果寄存器 C 为非 0 则跳过下一条指令。



公式:

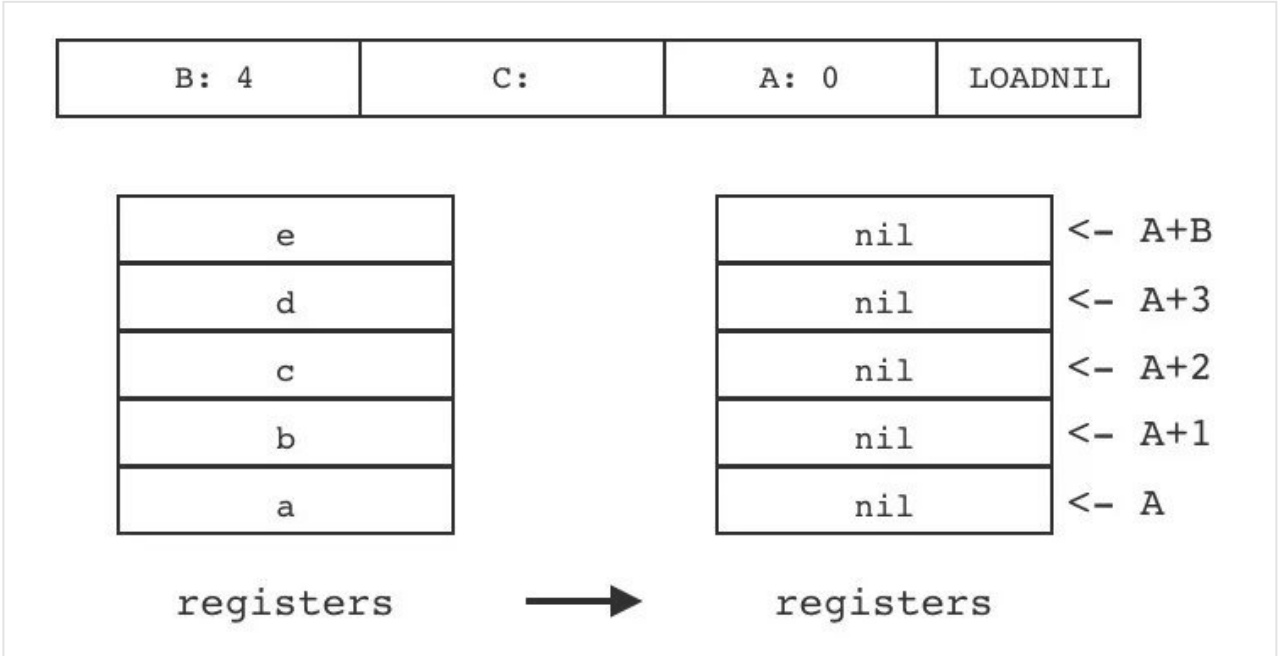
R(A) := (bool)B

if(C) pc++

指令名称类型操作码BCA
LOADNILiABC0x04OpArgUOpArgN目标寄存器 idx

B:4 C A:0 LOADNIL

将序号[A,A+B]连续 B+1 个寄存器设置成 nil 值，用于给连续 n 个寄存器放置 nil 值。在 lua 中，局部变量的默认初始值为 nil，LOADNIL 指令常用于给连续 n 个局部变量设置初始值。

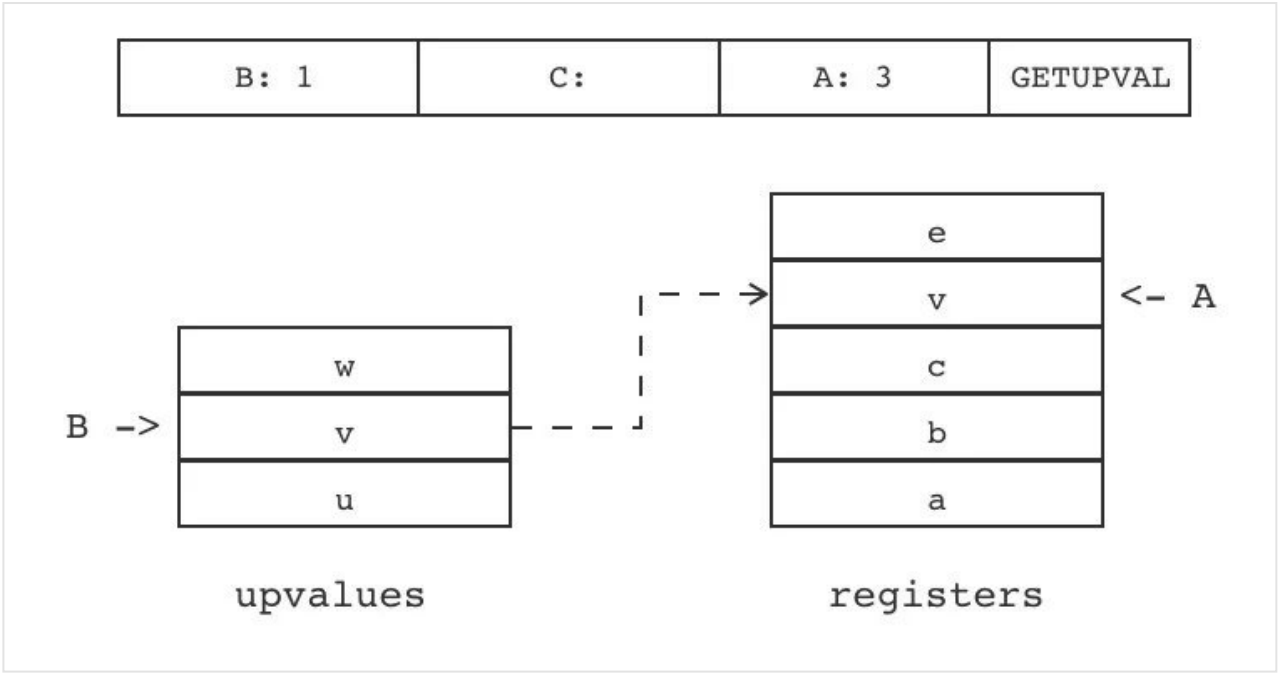


公式：R(A), R(A+1), ... ,R(A+B) := nil

指令名称类型操作码BCA
GETUPVALiABC0x05OpArgUOpArgN目标寄存器 idx

B:1 C A:3 GETUPVAL

把当前闭包的某个 Upvalue 值（索引由 B 指定）拷贝到目标寄存器（索引由 A 指定）中 。

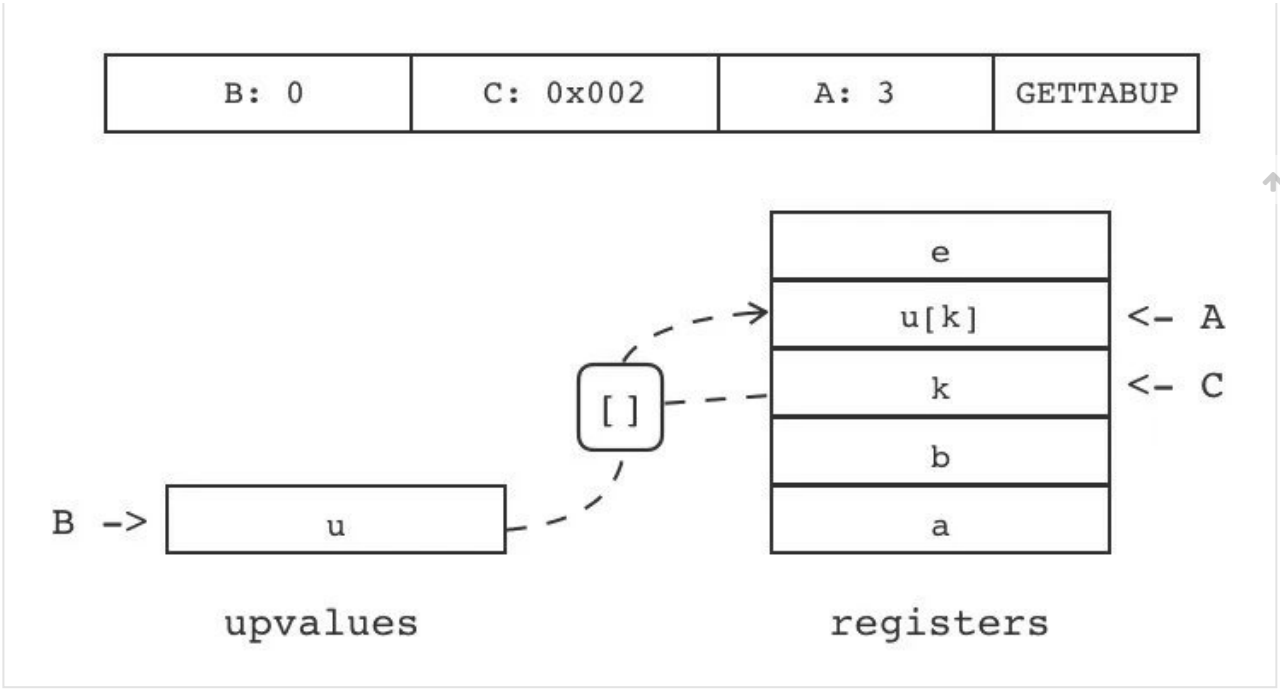


公式：R(A) := Upvalue[B]

指令名称类型操作码BCA
GETTABUPiABC0x06OpArgUOpArgK目标寄存器 idx

B:0 C:0x002 A:3 GETTABUP

把当前闭包的某个 Upvalue 值（索引由 B 指定）拷贝到目标寄存器（索引由 A 指定）中，与 GETUPVAL 不同的是，Upvalue 从表里取值（键由 C 指定，为寄存器或常量表索引）。

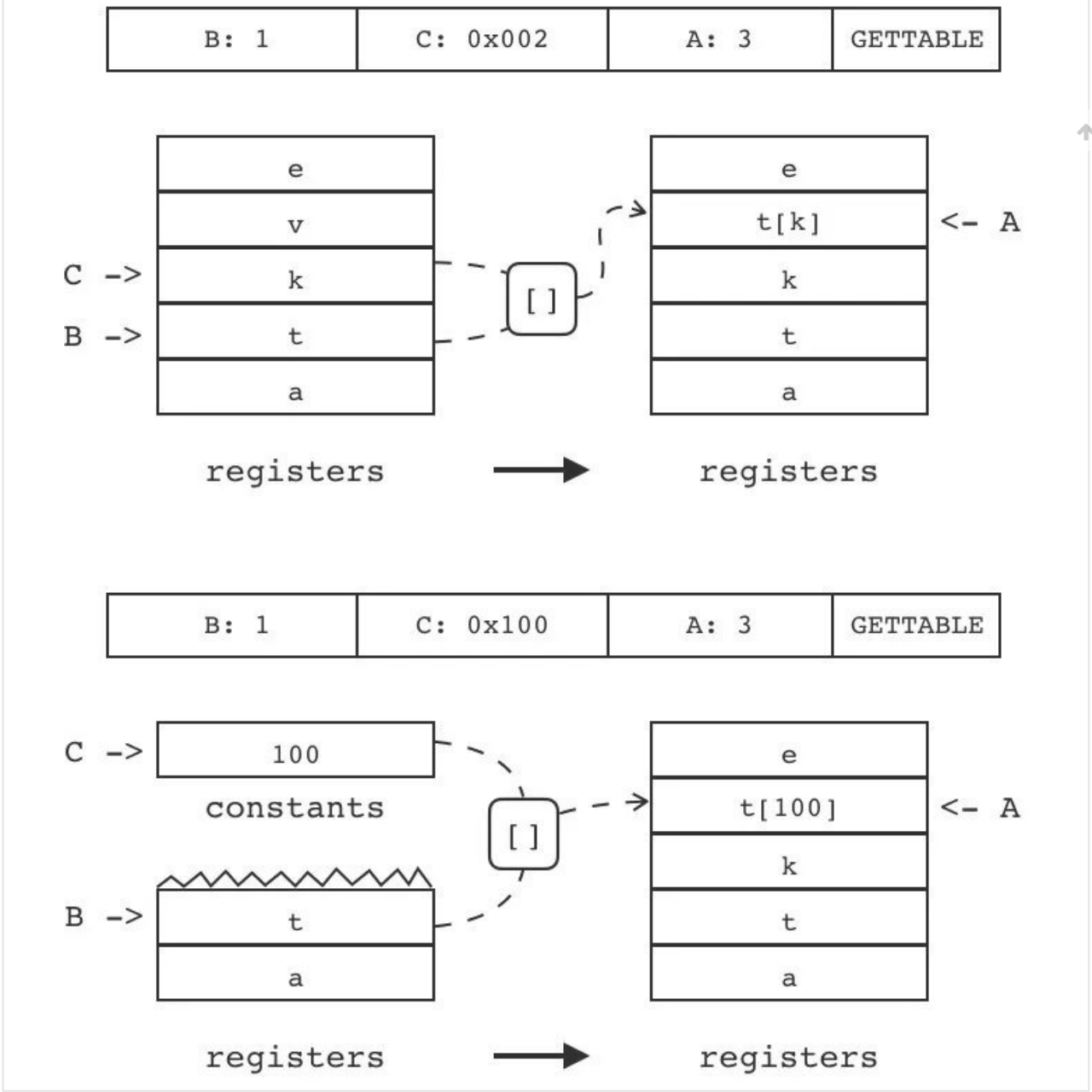


R(A) := Upvalue[B][rk(c)]

指令名称类型操作码BCA
GETTABLEiABC0x07OpArgROpArgK目标寄存器 idx

B:0 C:0x002 A:3 GETTABLE

把表中某个值拷贝到目标寄存器（索引由 A 指定）中，表所在寄存器索引由 B 指定，键由 C（为寄存器或常量表索引）指定。



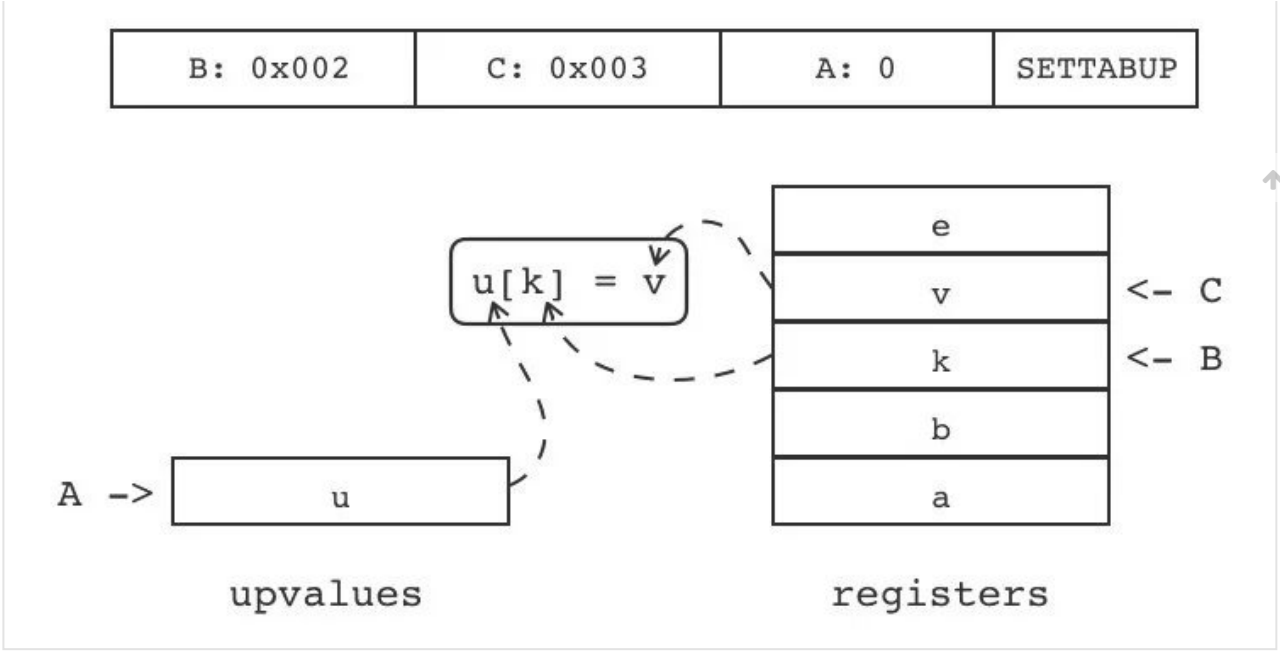
公式：R(A) := R[B][rk(c)]

指令名称类型操作码BCA

SETTABUPiABC0x08OpArgKOpArgK目标寄存器 idx

B:0x002 C:0x003 A:0 SETTABUP

设置当前闭包的某个 Upvalue 值（索引由 A 指定）为寄存器或常量表的某个值（索引由 C 指定），与 SETUPVAL 不同的是，Upvalue 从表里取值（键由 B 指定，为寄存器或常量表索引）。



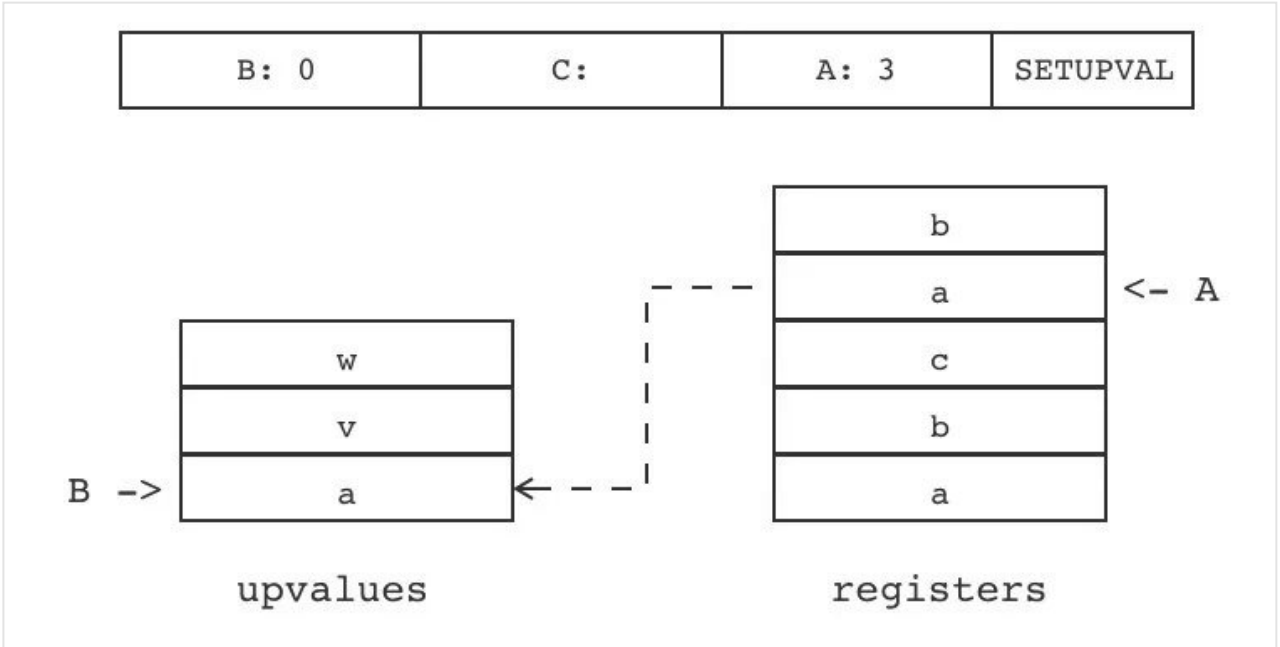
Upvalue[A][rk(b)] := RK(C)

指令名称类型操作码BCA

SETUPVALiABC0x09OpArgUOpArgN目标寄存器 idx

B:0 C A:3 SETUPVAL

设置当前闭包的某个 Upvalue 值（索引由 B 指定）为寄存器的某个值（索引由 A 指定）。



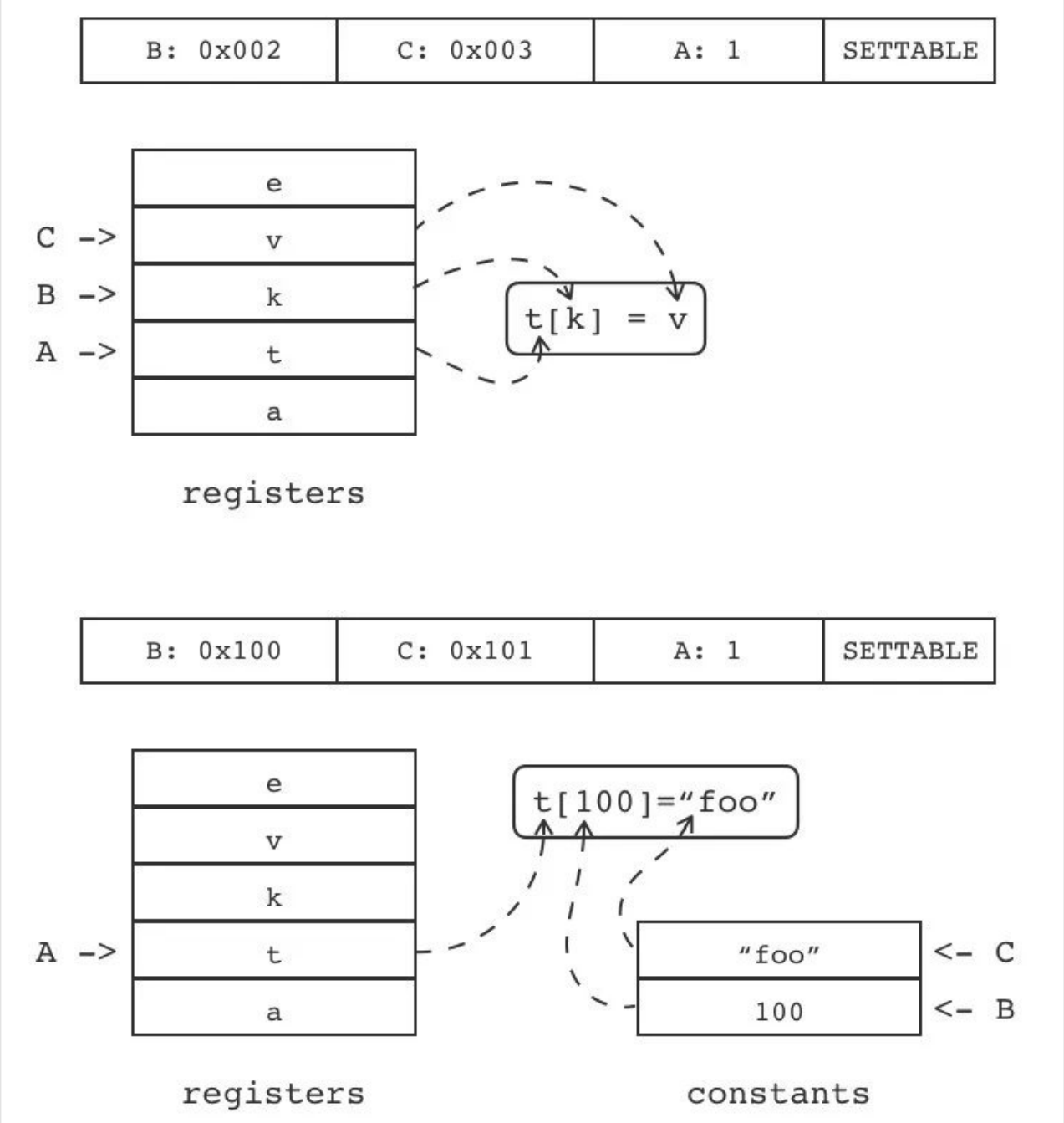
公式：Upvalue[B] := R(A)

指令名称类型操作码BCA

SETTABLEiABC0x0AOpArgKOpArgK目标寄存器 idx

B:0x002 C:0x003 A:1 SETTABLE

给寄存器中的表（索引由 A 指定）的某个键进行赋值，键和值分别由 B 和 C 指定（为寄存器或常量表索引）。



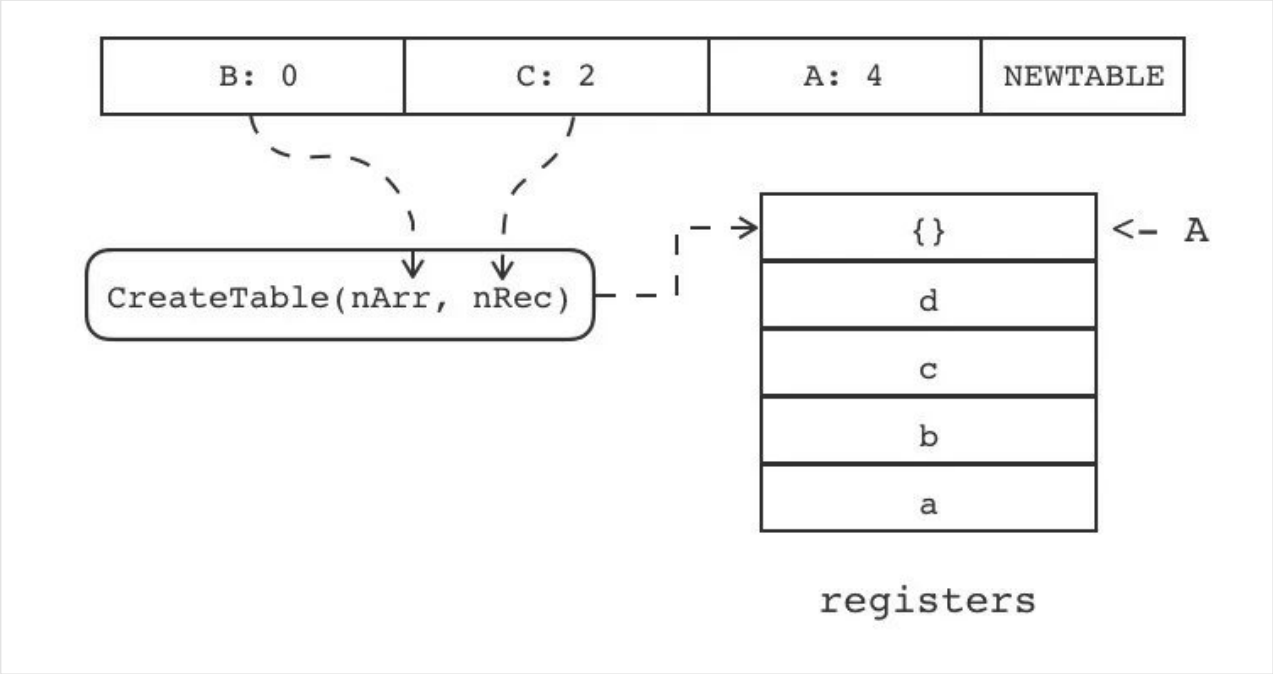
公式：R(A)[RK(B)] := RK(C)

指令名称类型操作码BCA

NEWTABLEiABC0x0BOpArgUOpArgU目标寄存器 idx

B:0 C:2 A:4 NEWTABLE

创建空表，并将其放入指定寄存器（索引有 A 指定），表的初始数组容量和哈希表容量分别有 B 和 C 指定。

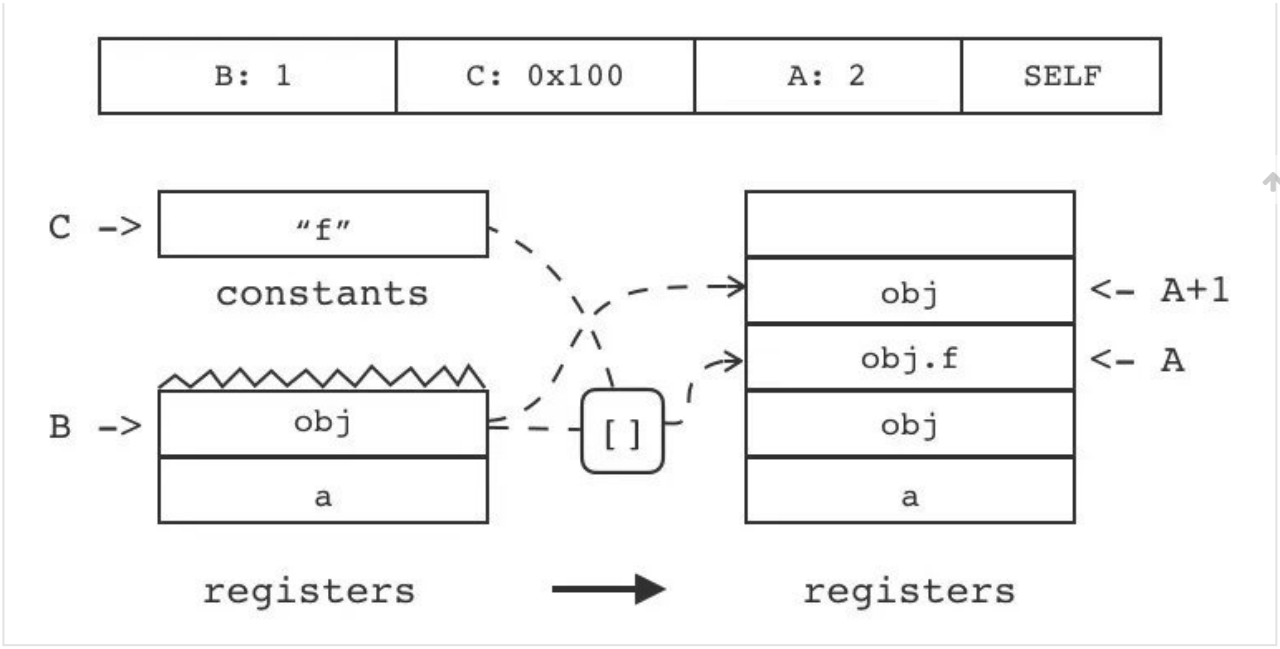


公式：R(A) := {} (size = B, C)

指令名称类型操作码BCASELFiABC0x0COpArgROpArgK目标寄存器 idx

B:1 C:0x100 A:2 SELF

把寄存器中对象（索引由 B 指定）和常量表中方法（索引由 C 指定）拷贝到相邻的两个目标寄存器中，起始目标寄存器的索引由 A 指定。



公式:

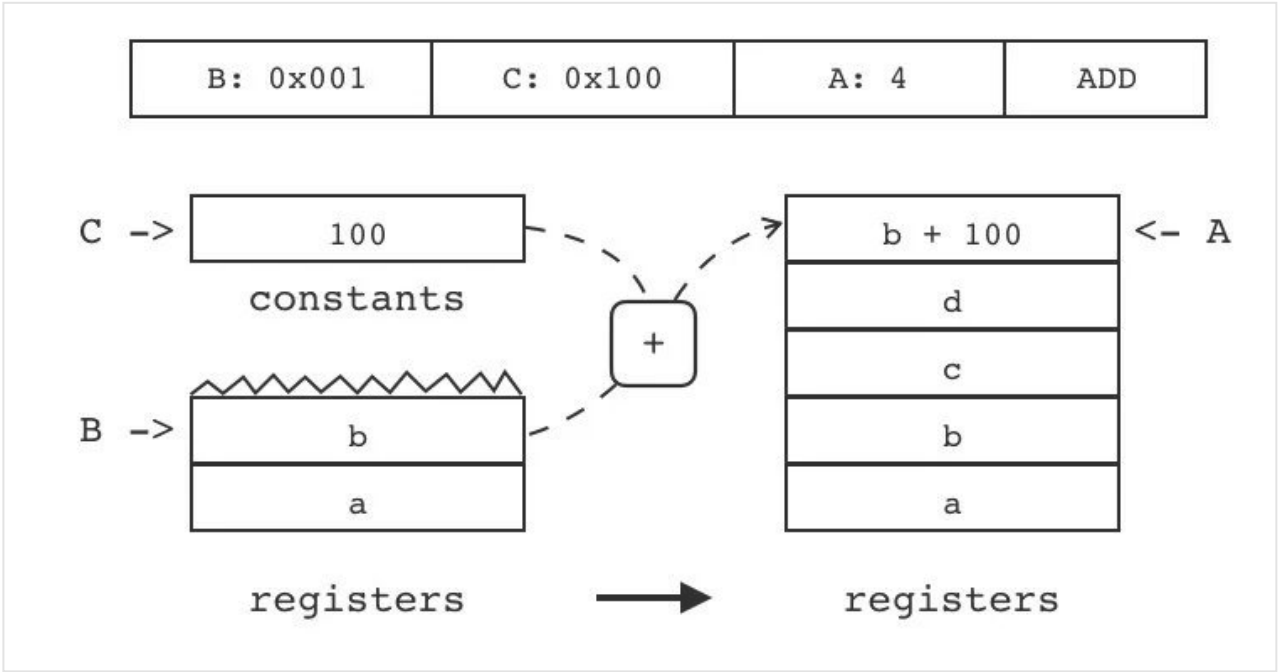
R(A+1) := R(B)

R(A) := R(B)[RK(C)]

指令名称类型操作码BCAADDiABC0x0DOpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 ADD

对两个寄存器或常量值（索引由 B 和 C 指定）进行相加，并将结果放入另一个寄存器中（索引由 A 指定）。



公式：R(A) := RK(B) + RK(C)

指令名称类型操作码BCASUBiABC0x0EOpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 SUB

对两个寄存器或常量值（索引由 B 和 C 指定）进行相减，并将结果放入另一个寄存器中（索引由 A 指定）

公式：

R(A) := RK(B) - RK(C)

指令名称类型操作码BCAMULiABC0x0FOpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 MUL

对两个寄存器或常量值（索引由 B 和 C 指定）进行相乘，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) * RK(C)

指令名称类型操作码BCAMODiABC0x10OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 MOD

对两个寄存器或常量值（索引由 B 和 C 指定）进行求摸运算，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) % RK(C)

指令名称类型操作码BCAPOWiABC0x11OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 POW

对两个寄存器或常量值（索引由 B 和 C 指定）进行求幂运算，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) ^ RK(C)

指令名称类型操作码BCADIViABC0x12OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 DIV

对两个寄存器或常量值（索引由 B 和 C 指定）进行相除，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) / RK(C)

指令名称类型操作码BCAIDIViABC0x13OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 IDIV

对两个寄存器或常量值（索引由 B 和 C 指定）进行相整除，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) // RK(C)

指令名称类型操作码BCABANDiABC0x14OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 BAND

对两个寄存器或常量值（索引由 B 和 C 指定）进行求与操作，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) & RK(C)

指令名称类型操作码BCABORiABC0x15OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 BOR

对两个寄存器或常量值（索引由 B 和 C 指定）进行求或操作，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) | RK(C)

指令名称类型操作码BCABXORiABC0x16OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 BXOR

对两个寄存器或常量值（索引由 B 和 C 指定）进行求异或操作，并将结果放入另一个寄存器中（索引由 A 指定）

公式：R(A) := RK(B) ~ RK(C)

指令名称类型操作码BCASHLiABC0x17OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 SHL

索引由 B 指定的寄存器或常量值进行左移位操作（移动位数的索引由 C 指定的寄存器或常量值），并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) << RK(C)

指令名称类型操作码BCASHRiABC0x18OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:4 SHR

索引由 B 指定的寄存器或常量值进行右移位操作（移动位数的索引由 C 指定的寄存器或常量值），并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := RK(B) >> RK(C)

指令名称类型操作码BCAUNMiABC0x19OpArgROpArgN目标寄存器 idx

B:1 C A:3 UNM

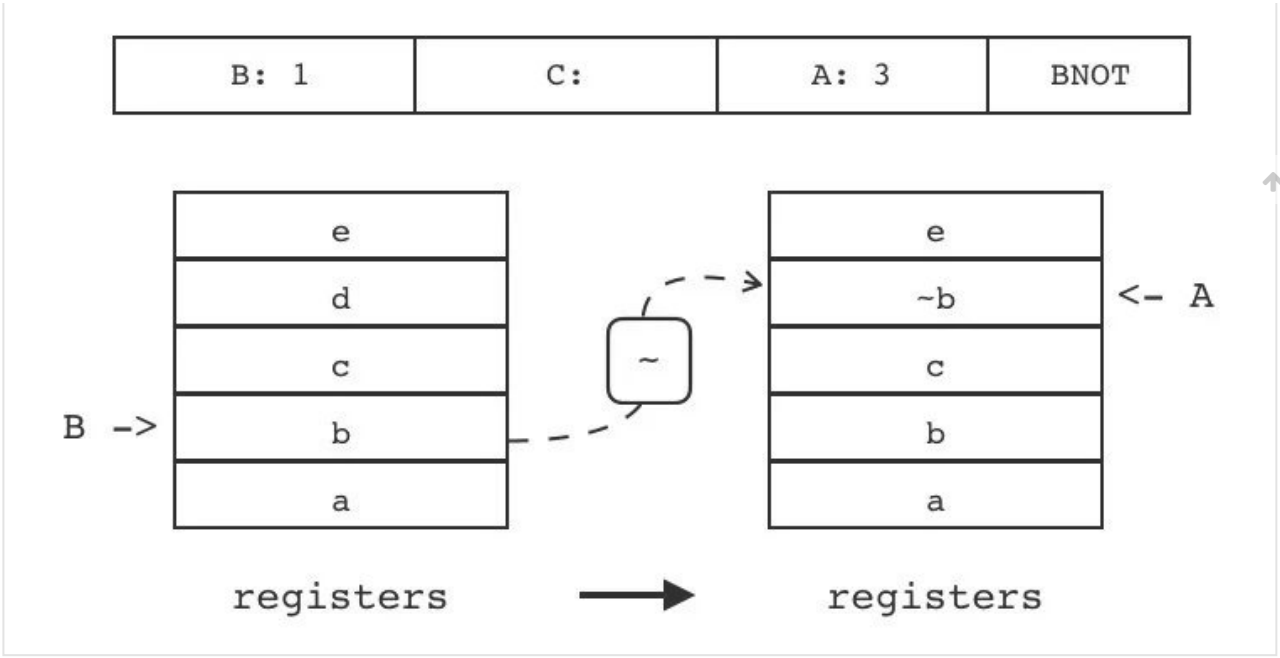
对寄存器（索引由 B 指定）进行取负数操作，并将结果放入另一个寄存器中（索引由 A 指定）。

公式：R(A) := - R(B)

指令名称类型操作码BCABNOTiABC0x1AOpArgROpArgN目标寄存器 idx

B:1 C A:3 BNOT

对寄存器（索引由 B 指定）进行取反操作，并将结果放入另一个寄存器中（索引由 A 指定）。

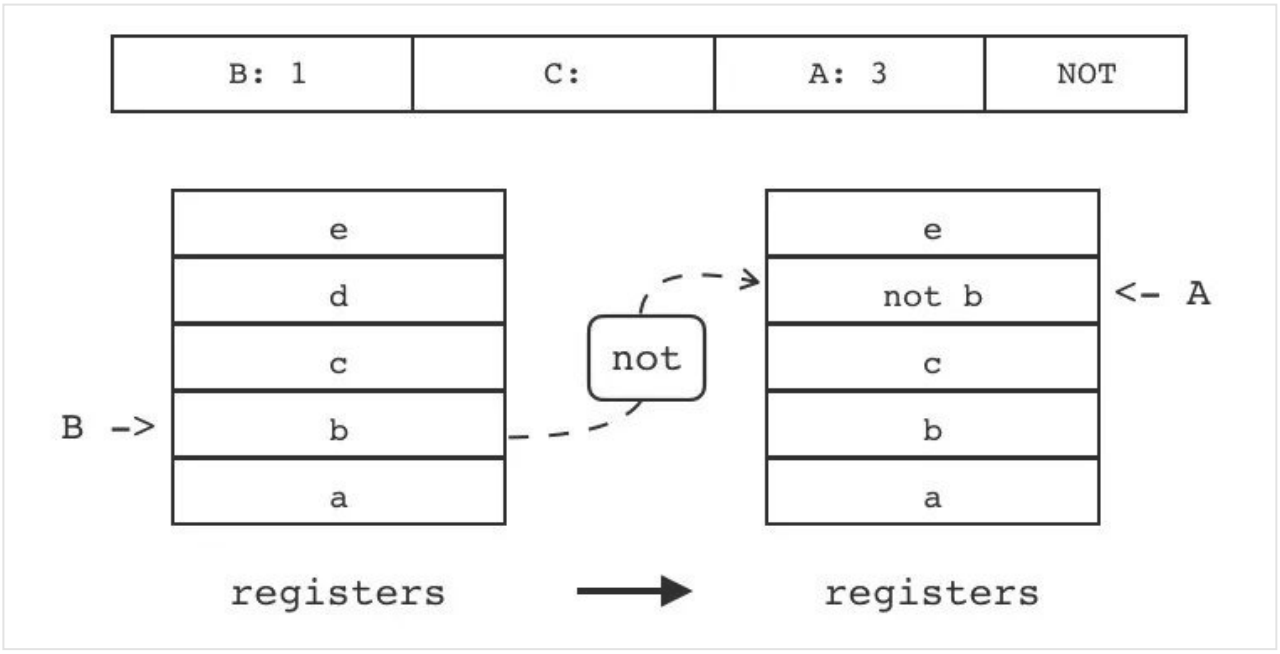


公式：R(A) := ~ R(B)

指令名称类型操作码BCANOTiABC0x1BOpArgROpArgN目标寄存器 idx

B:1 C A:3 NOT

对寄存器（索引由 B 指定）进行求非操作，并将结果放入另一个寄存器中（索引由 A 指定）。

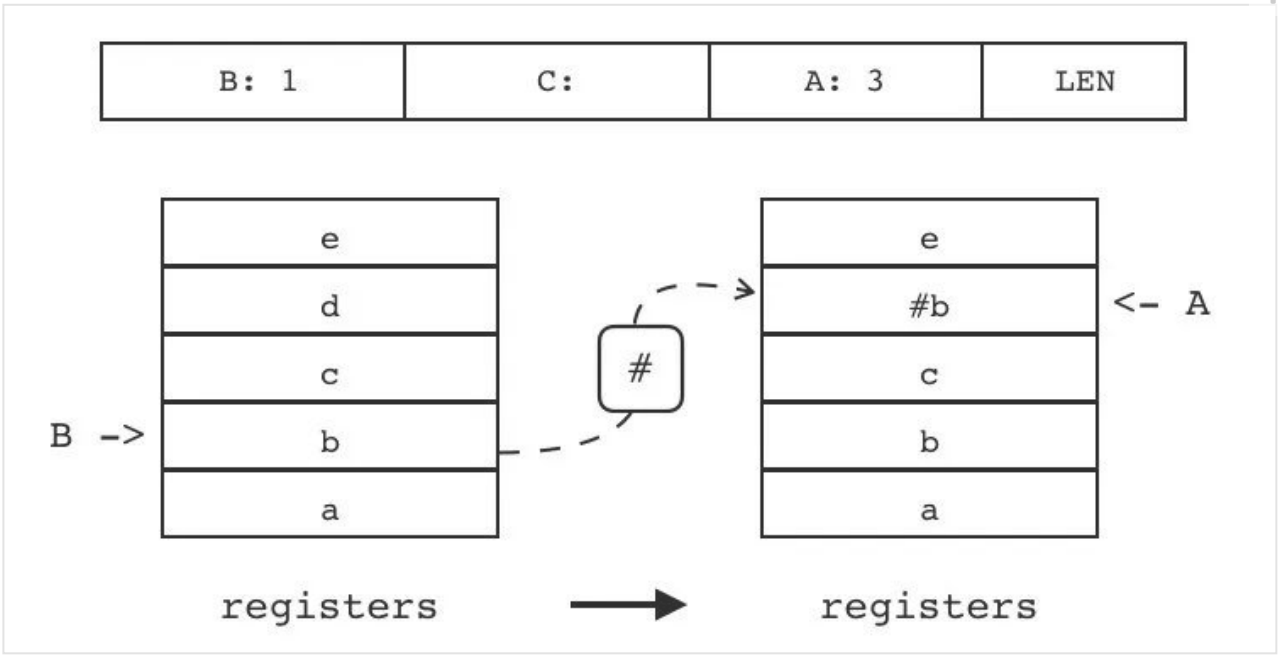


公式：R(A) := not R(B)

指令名称类型操作码BCALENiABC0x1COpArgROpArgN目标寄存器 idx

B:1 C A:3 LEN

对寄存器（索引由 B 指定）进行求长度操作，并将结果放入另一个寄存器中（索引由 A 指定）。



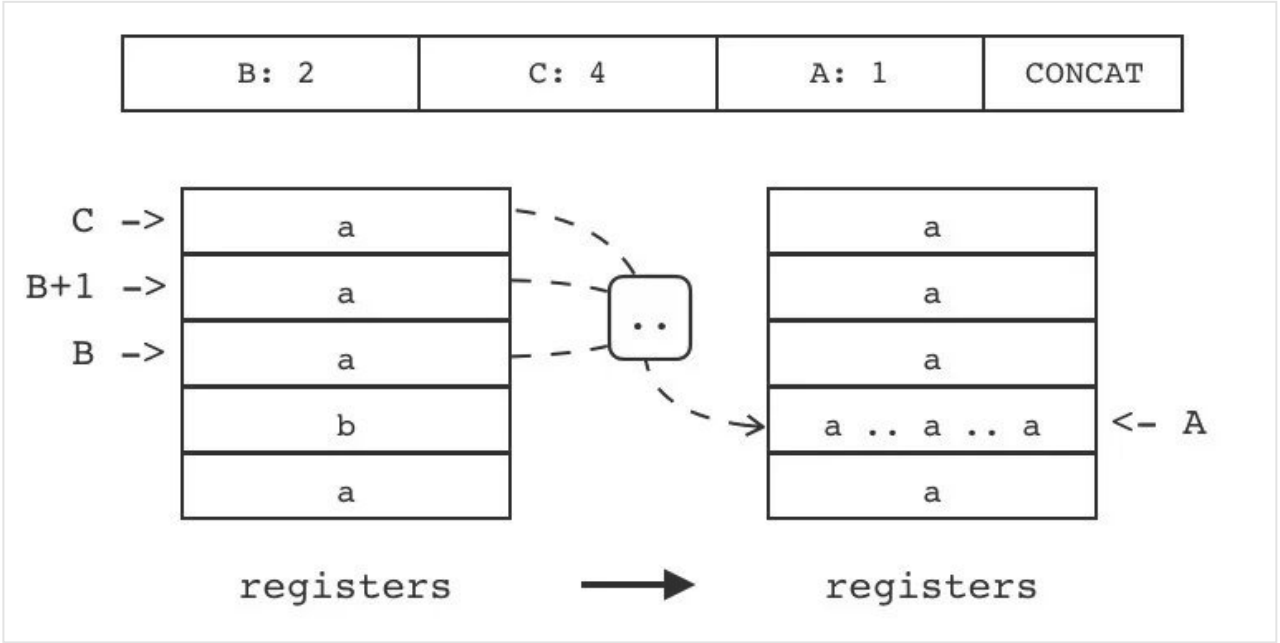
公式：R(A) := length of R(B)

指令名称类型操作码BCA

CONCATiABC0x1DOpArgROpArgR目标寄存器 idx

B:2 C:4 A:1 CONCAT

将连续 n 个寄存器（起始索引和终止索引由 B 和 C 指定）里的值进行拼接，并将结果放入另一个寄存器中（索引由 A 指定）。



公式：R(A) := R(B) .. .. R(C)

指令名称类型操作码sBxAJMPiAsBx0x1EOpArgR目标寄存器 idx

sBx:-1 A JMP

当 sBx 不为 0 时，进行无条件跳转，执行 pc = pc + sBx（sBx 为-1，表示将当前指令再执行一次 注：这将是一个死循环）

sBx:0 A:0x001 JMP;

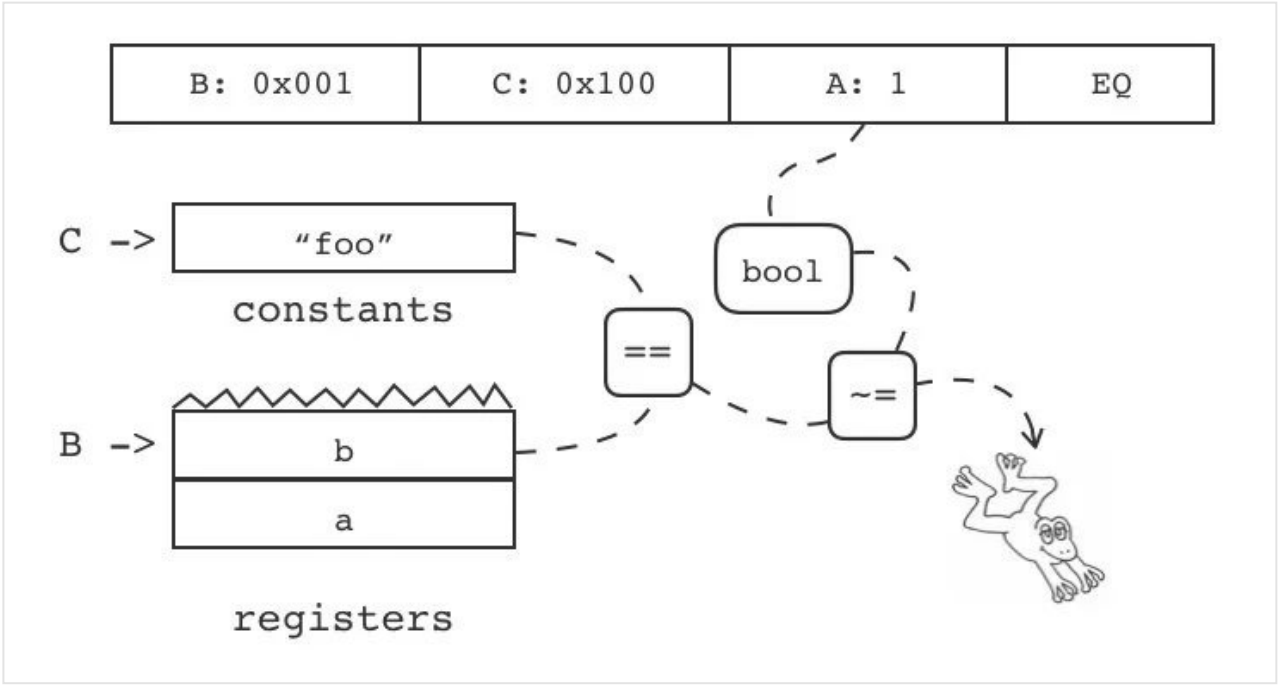
当 sBx 为 0 时（继续执行后面指令，不跳转），用于闭合处于开启状态的 Upvalue（即：把即将销毁的局部变量的值复制出来，并更新到某个 Upvalue 中）。

当前闭包的某个 Upvalue 值的索引由 A 指定:

指令名称类型操作码BCAEQiABC0x1FOpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:1 EQ

寄存器或常量表（索引由 B 指定）是否等于寄存器或常量表（索引由 C 指定），若结果等于操作数 A，则跳过下一条指令。



公式：if ((RK(B) == RK(C)) pc++

指令名称类型操作码BCALTiABC0x20OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:1 LT

寄存器或常量表（索引由 B 指定）是否小于寄存器或常量表（索引由 C 指定），若结果等于操作数 A，则跳过下一条指令。

公式：if ((RK(B) < RK(C)) pc++

指令名称类型操作码BCALEiABC0x21OpArgKOpArgK目标寄存器 idx

B:0x001 C:0x100 A:1 LE

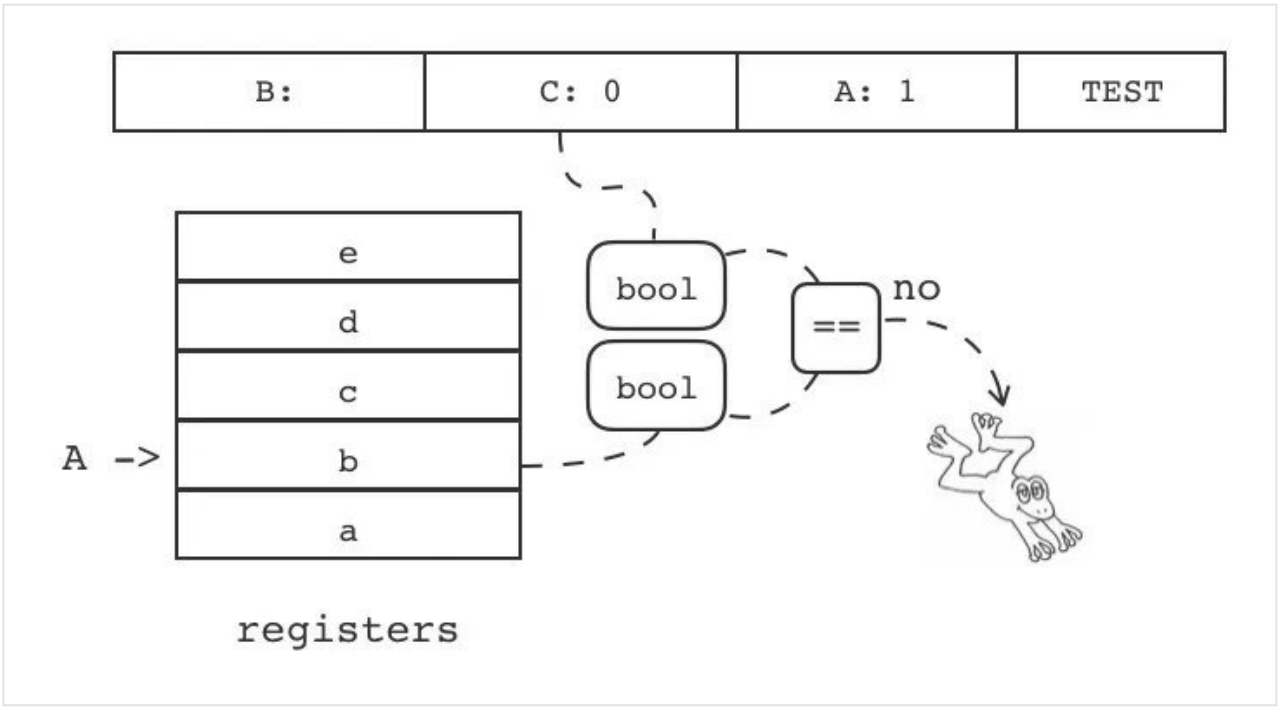
寄存器或常量表（索引由 B 指定）是否小于等于寄存器或常量表（索引由 C 指定），若结果等于操作数 A，则跳过下一条指令。

公式：if ((RK(B) <= RK(C)) pc++

指令名称类型操作码BCATESTiABC0x22OpArgNOpArgU目标寄存器 idx

B C:0 A:1 TEST

判断寄存器（索引由 A 指定）中的值转换为 bool 值后，是否和操作数 C 表示的 bool 值一致，若结果不一致，则跳过下一条指令。



公式：

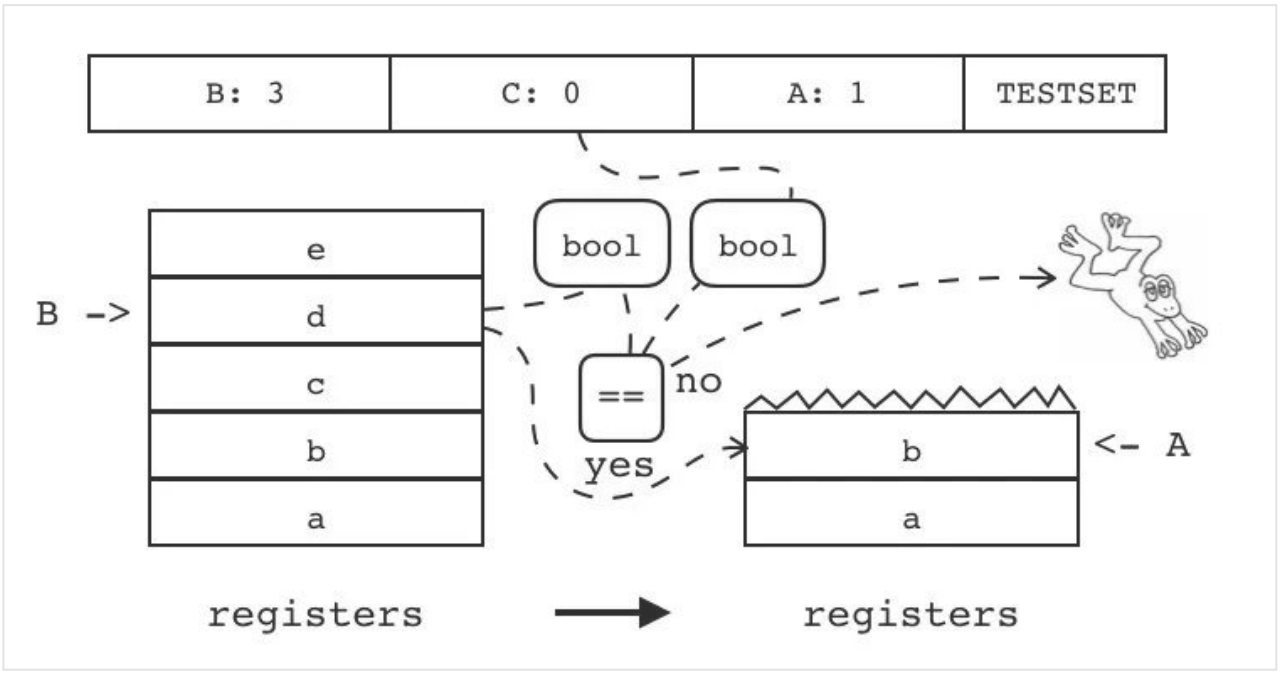
if not (R(A) <=> C) pc++

注：<=>表示按 bool 值比较

指令名称类型操作码BCA
TESTSETiABC0x23OpArgROpArgU目标寄存器 idx

B:3 C:0 A:1 TESTSET

判断寄存器（索引由 B 指定）中的值转换为 bool 值后，是否和操作数 C 表示的 bool 值一致，若结果一致，将寄存器（索引由 B 指定）中的值复制到寄存器中（索引由 A 指定），否则跳过下一条指令。



公式：

```
1  if (R(B) \<=\> C)
2     R(A) := R(B)
3  else
4     pc++
```

注：<=>表示按 bool 值比较

指令名称类型操作码BCACALLiABC0x24OpArgUOpArgU目标寄存器 idx

B:5 C:4 A:0 CALL

被调用函数位于寄存器中（索引由 A 指定），传递给被调用函数的参数值也在寄存器中，紧挨着被调用函数，参数个数为操作数 B 指定。

① B==0，接受其他函数全部返回来的参数

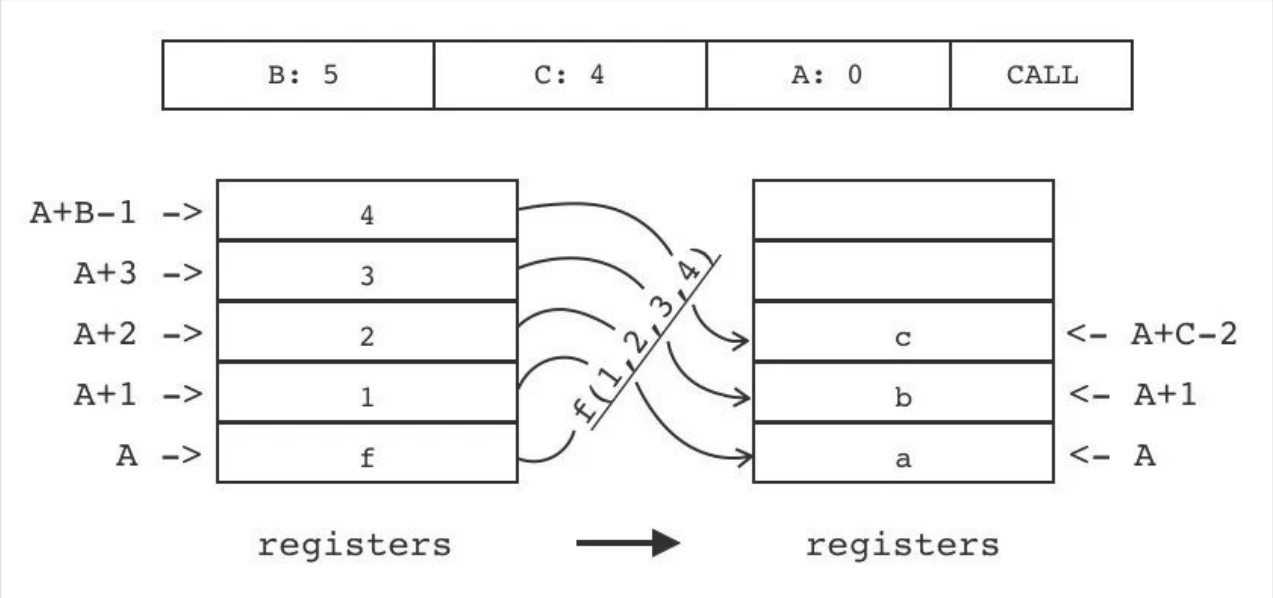
② B>0，参数个数为 B-1

函数调用结束后，原先存放函数和参数值的寄存器会被返回值占据，具体多少个返回值由操作数 C 指定。

① C==0，将返回值全部返回给接收者

② C==1，无返回值

③ C>1，返回值的数量为 C-1



公式：R(A), ... ,

指令名称类型操作码BCA
TAILCALLiABC0x25OpArgUOpArgU目标寄存器 idx

函数调用一般通过调用栈来实现。用这种方法，每调用一个函数都会产生一个调用帧。

如果调用层次太深（如递归），容易导致栈溢出。尾递归优化则可以让我们发挥递归函数调用威力的同时，避免调用栈溢出。利用这种优化，被调函数可以重用主调函数的调用帧，因此可有效缓解调用栈溢出症状。不过该优化只适合某些特定情况。

如：return f(args) 会被编译器优化成 TAILCALL 指令，公式：return R(A)(R(A+1), … , R(A+B-1))

指令名称类型操作码BCA
RETURNiABC0x26OpArgUOpArgN目标寄存器 idx

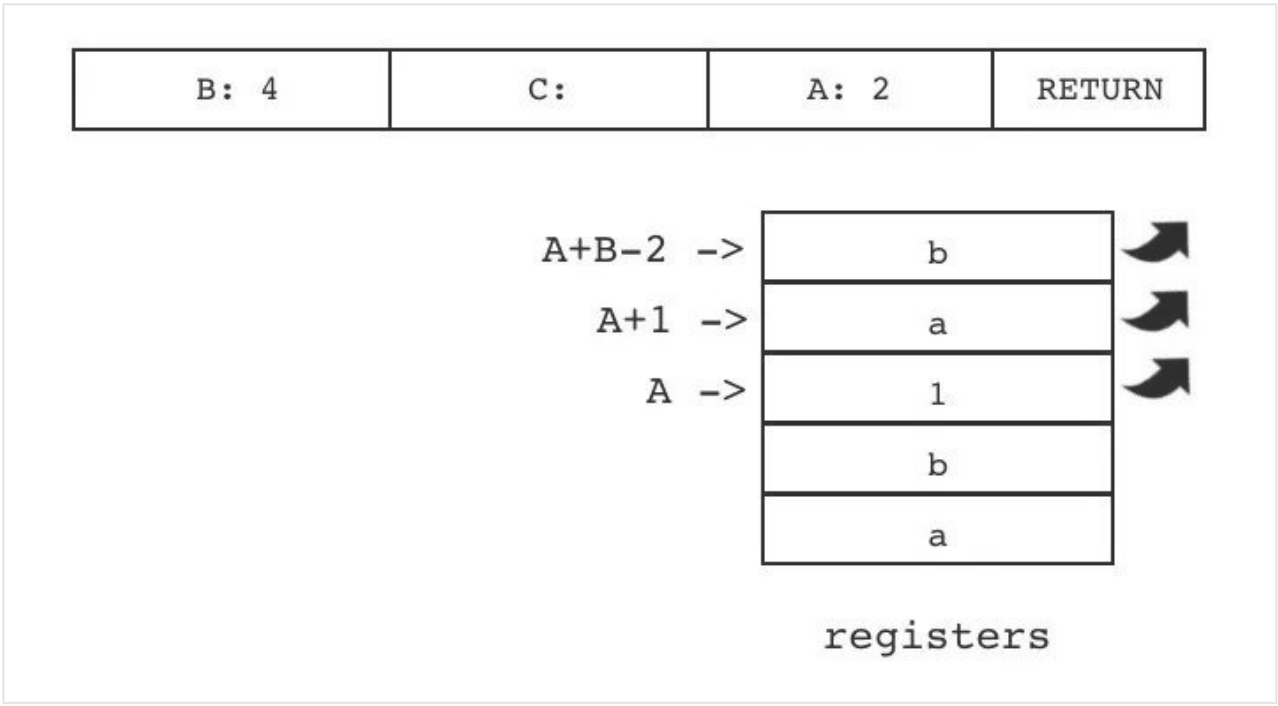
B:4 C A:2 RETURN

把存放在连续多个寄存器里的值返回给父函数，其中第一个寄存器的索引由操作数 A 指定，寄存器数量由操作数 B 指定，操作数 C 没有使用，需要将返回值推入栈顶：

① B==1，不需要返回任何值

② B > 1，需要返回 B-1 个值；这些值已经在寄存器中了，只用再将它们复制到栈顶即可

③ B==0，一部分返回值已经在栈顶了，只需将另一部分也推入栈顶即可

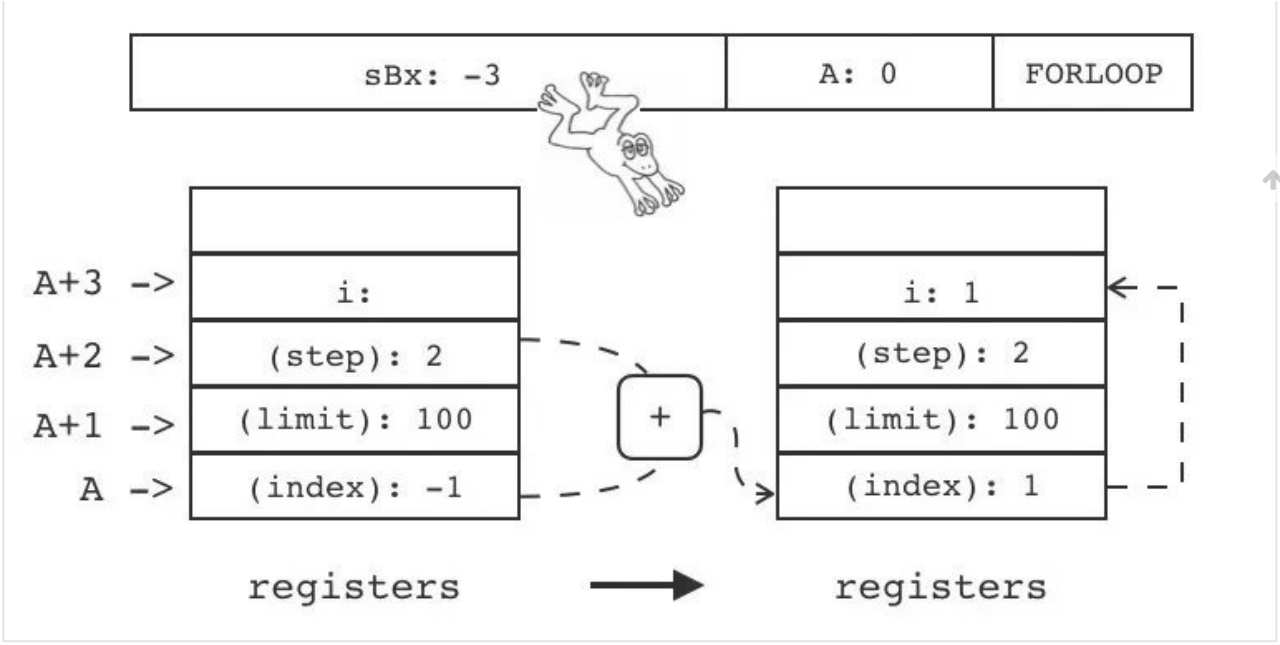


公式：return R(A),…,R(A+B-2)

指令名称类型操作码sBxAFORLOOPiAsBx0x27OpArgR目标寄存器 idx

数值 for 循环：用于按一定步长遍历某个范围内的数值 如：for i=1,100,2 do f() end // 初始值为 1，步长为 2，上限为 100

该指令先给 i 加上步长，然后判断 i 是否在范围之内。若已经超出范围，则循环结束；若为超出范围，则将数值拷贝给用户定义的局部变量，然后跳转到循环体内部开始执行具体的代码块。



公式：

R(A) += R(A+2)

if R(A) <?= R(A+1)

pc+=sBx

R(A+3)=R(A)

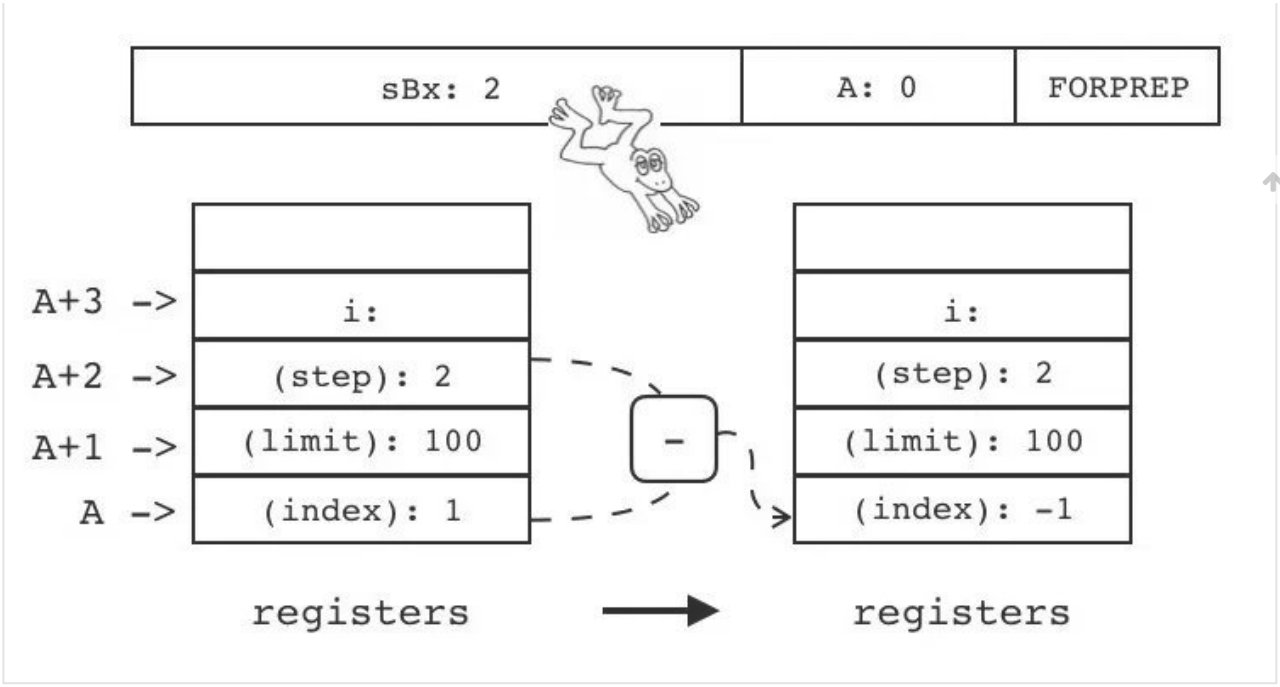
注：当步长为正数时<?=为<=

当步长为负数时<?=为>=

指令名称类型操作码sBxAFORPREPiAsBx0x28OpArgR目标寄存器 idx

数值 for 循环：用于按一定步长遍历某个范围内的数值 如：for i=1,100,2 do f() end // 初始值为 1，步长为 2，上限为 100。

该指令的目的是在循环之前预先将 i 减去步长（得到-1），然后跳转到 FORLOOP 指令正式开始循环：



公式：

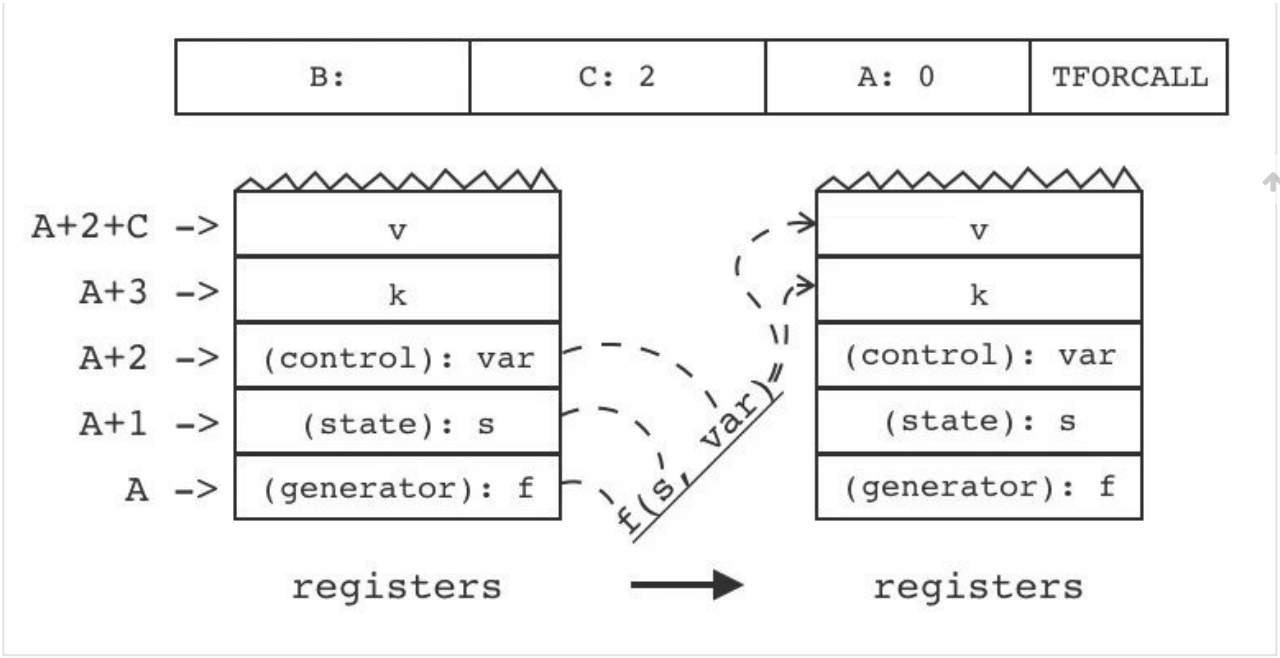
R(A)-=R(A+2)

pc+=sBx

指令名称类型操作码BCA
TFORCALLiABC0x29OpArgNOpArgU目标寄存器 idx

通用 for 循环：for k,v in pairs(t) do print(k,v) end

编译器使用的第一个特殊变量(generator):f 存放的是迭代器，其他两个特殊变量(state):s、(control):var 来调用迭代器，把结果保存在用户定义的变量 k、v 中。

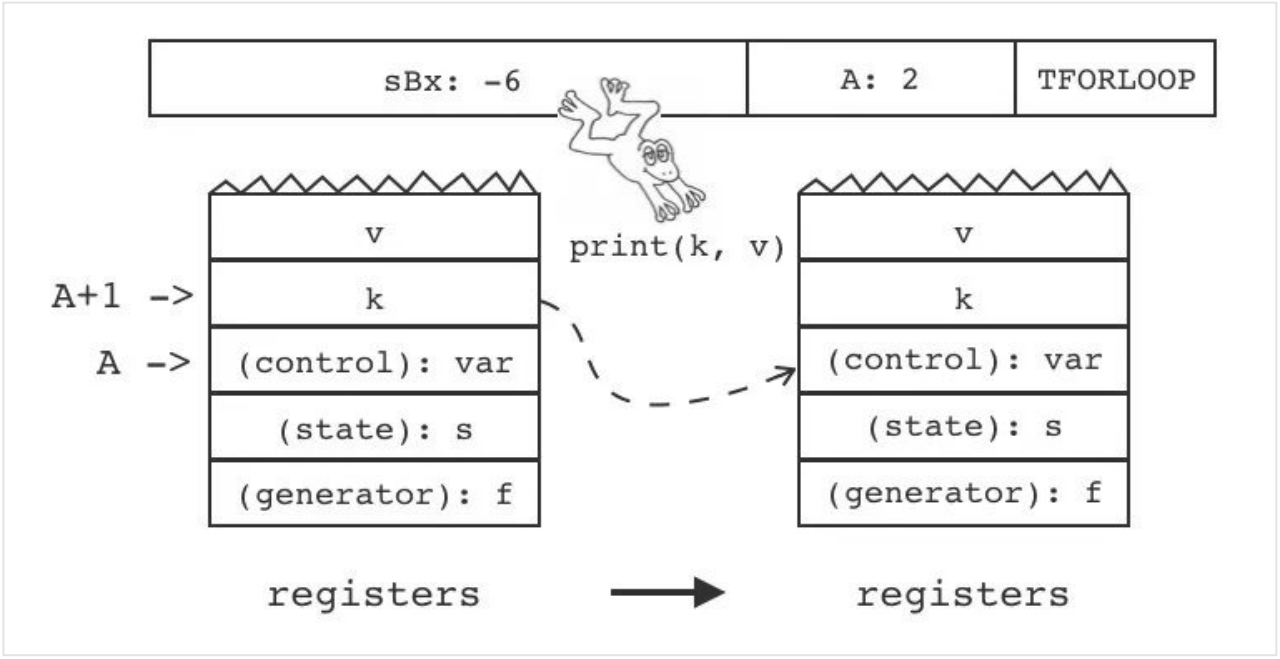


公式：R(A+3),...,R(A+2+C) := R(A)(R(A+1),R(A+2))

指令名称类型操作码sBxATFORLOOPiAsBx0x2AOpArgR目标寄存器 idx

通用 for 循环：for k,v in pairs(t) do print(k,v) end

若迭代器返回的第一个值（变量 k）不是 nil，则把该值拷贝到(control):var，然后跳转到循环体；若为 nil，则循环结束。



公式：

if R(A+1) ~= nil

R(A)=R(A+1)

pc+=sBx

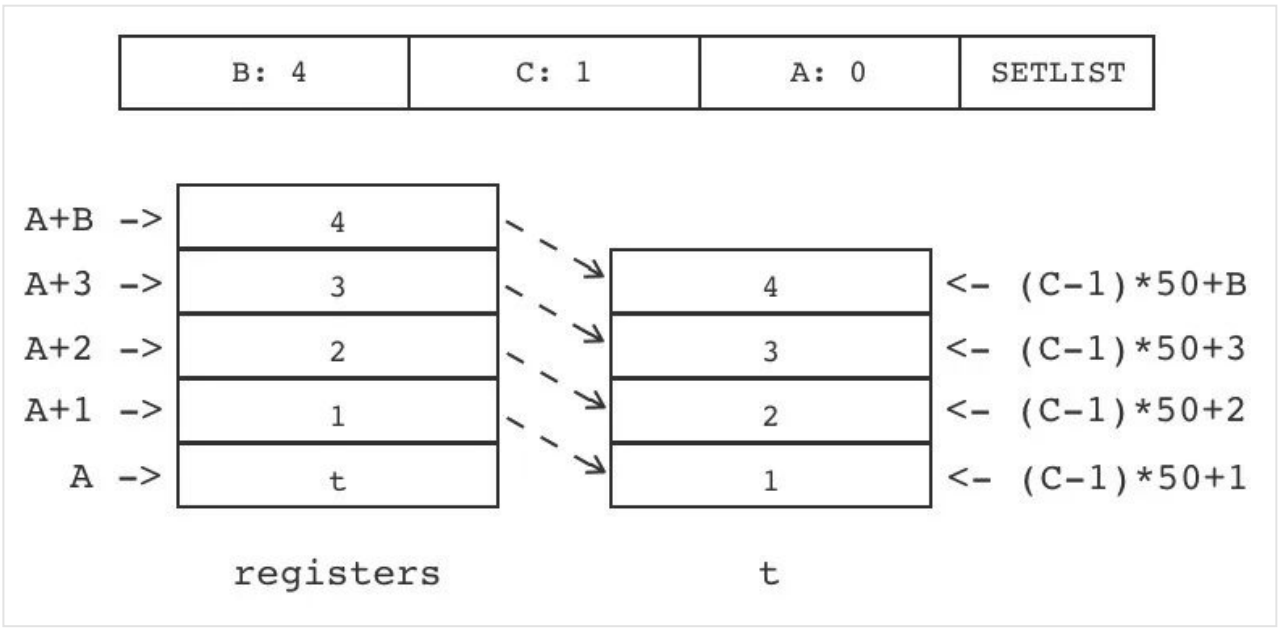
指令名称类型操作码BCA
SETLISTiABC0x2BOpArgUOpArgU目标寄存器 idx

SETTABLE 是通用指令，每次只处理一个键值对，具体操作交给表去处理，并不关心实际写入的是表的 hash 部分还是数组部分。SETLIST 则是专门给数组准备的，用于按索引批量设置数组元素。其中数组位于寄存器中，索引由操作数 A 指定；需要写入数组的一系列值也在寄存器中，紧挨着数组，数量由操作数 B 指定；数组起始索引则由操作数 C 指定。

因为 C 操作数只有 9bits，所以直接用它表示数组索引显然不够用。这里解决办法是让 C 操作数保存批次数，然后用批次数乘上批大小（FPF，默认为 50）就可以算出数组的起始索引。因此，C 操作数能表示的最大索引为 25600（50*512），当数组长度大于 25600 时，SETLIST 指令后会跟一条 EXTRAARG 指令，用其 Ax 操作数来保存批次数。

综上，C>0，表示的是批次数+1，否则，真正批次数存放在后续的 EXTRAARG 指令中。

操作数 B 为 0 时，当表构造器的最后一个元素是函数调用或者 vararg 表达式时，Lua 会把它们产生的所有值都收集起来供 SETLIST 使用。



公式：

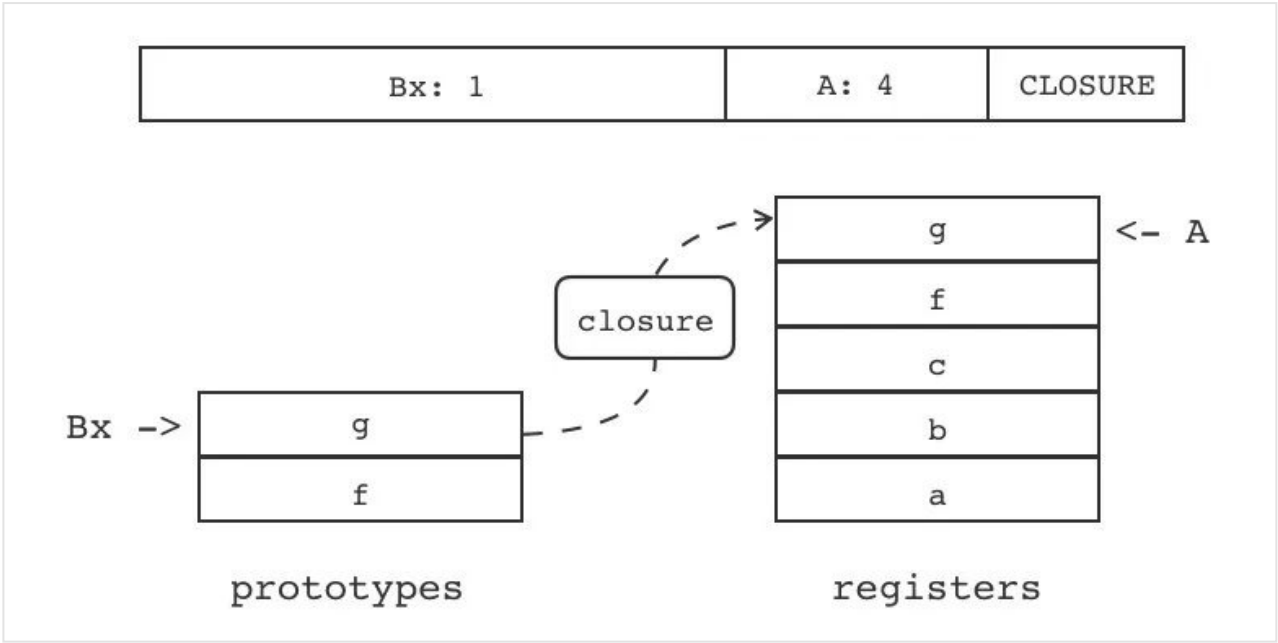
R(A)[(C-1)*FPF+i] := R(A+i)

1 <= i <= B

指令名称类型操作码BxACLOSUREiABx0x2COpArgU目标寄存器 idx

把当前 Lua 函数的子函数原型实例化为闭包，放入由操作数 A 指定的寄存器中子函数原型来自于当前函数原型的子函数原型表，索引由操作数 Bx 指定。

下图为将 prototypes 表中索引为 1 的 g 子函数，放入索引为 4 的寄存器中：

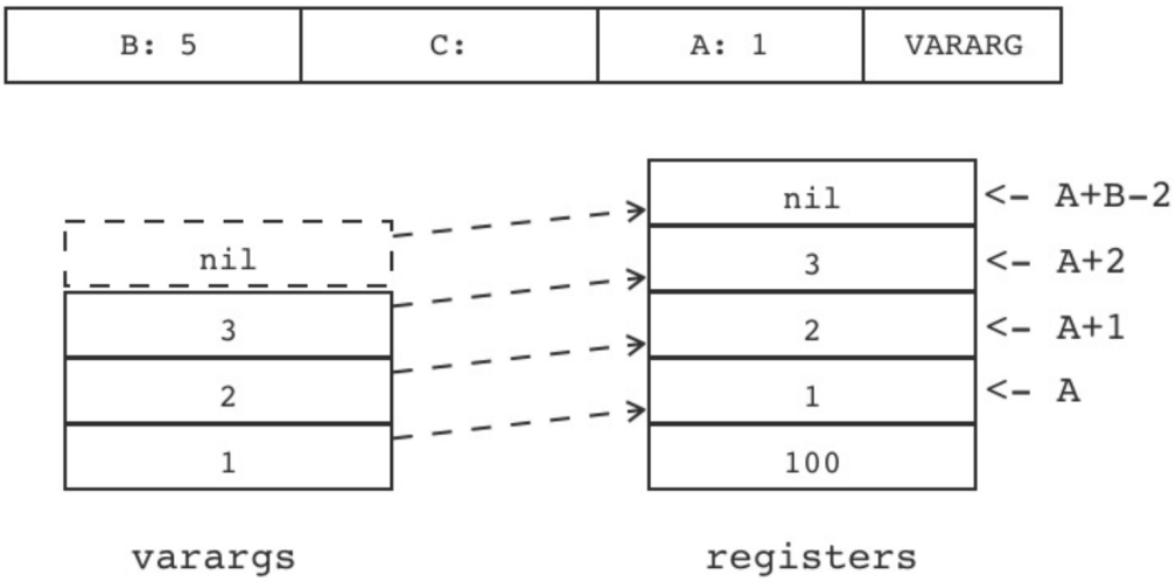


公式：R(A) := closure(KPROTO[Bx])

指令名称类型操作码BCA
VARARGiABC0x2DOpArgUOpArgN目标寄存器 idx

把传递给当前函数的变长参数加载到连续多个寄存器中。

其中第一个寄存器的索引由操作数 A 指定，寄存器数量由操作数 B 指定，操作数 C 没有使用，操作数 B 若大于 1，表示把 B-1 个 vararg 参数复制到寄存器中，否则只能等于 0。



公式：R(A),R(A+1),...R(A+B-2)=vararg

指令名称类型操作码AxEXTRAARGiAx0x2EOpArgU

Ax:67108864 EXTRAARG

Ax 有 26bits，用来指定常量索引，可存放最大无符号整数为 67108864，可满足大部分情况的需要了。

参考

- 《自己动手实现 Lua》源代码
- Lua 设计与实现–虚拟机篇
- Lua 5.3 Bytecode Reference
- Lua 源码解析
  作者：nicochen，腾讯 IEG 游戏开发工程师

----------本文结束 ♥ 感谢阅读----------

lua lua知识点

探索Lua52内部实现:GC3

C4：4个函数，528行代码实现可自举的 C语言编译器