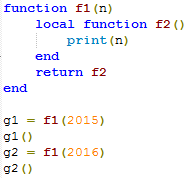
<http://www.2cto.com/kf/201503/382691.html>

<http://simohayha.iteye.com/blog/540546>源码解析

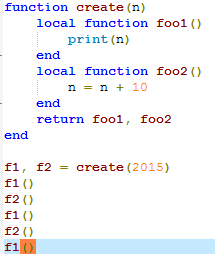
在lua中，闭包（closure）是由一个函数和该函数会访问到的非局部变量（或者是upvalue）组成的，其中非局部变量（non-local variable）是指不是在局部作用范围内定义的一个变量，但同时又不是一个全局变量，主要应用在前台函数和匿名函数里，因此，若一个闭包没有会访问的非局部变量，那么它就是通常说的函数。也就是说，在lua中，函数是闭包的一种特殊情况。另外，在lua的C API中，所有关于lua中的函数的核心API都是与closure（而非function）来命名的，也可视为这一观点的延续，在lua中，函数是一种第一类型值，他们具有特定的词法域

词法域是指一个函数可以嵌套在另一个函数中，内部的函数可以访问外部函数的变量





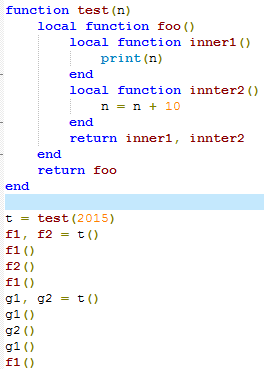
注意这里的g1和g2的函数体相同（都是f1的内嵌函数f2的函数体），但打印值不同。这是因为创建这个两个闭包时，他们都拥有局部变量n的独立实例。事实上，lua编译一个函数时，会为它生成一个原型（prototype），其中包含了函数体对应的虚拟机指令、函数用到的常量值（数、文本字符串等）和一些调试信息。在运行时，每当lua执行一个形如function…end这样的表达式时，它就会创建一个新的数据对象，其中包含了响应函数原型的引用及一个由于所有upvalue引用组成的数组，而这个数据对象就称为闭包。由此可见，函数是编译器的概念，是静态的，而闭包时运行期概念，是动态的，g1和g2的值严格来说不是函数而是闭包，并且是两个不相同的闭包，而每个闭包能保有自己的upvalue值，所以g1和g2打印出来的值当然就不相同了。这些upvalue变量可以在内部函数之间共享，即upvalue是提供一种闭包之间共享数据的方法





注意上面的例子中，闭包f1和f2共享同一个upvalue了，这是因为当lua发现两个闭包的upvalue指向的是当前堆栈上的想通变量时，会聪明地只生成一个拷贝，然后让这两个闭包共享该拷贝，这样，任意一个闭包对该upvalue进行修改都会被另一个探知

闭包在创建之时期upvalue就已不在堆栈上的情况上也有可能发生，这是因为内嵌函数能引用更外层外包函数的局部变量：



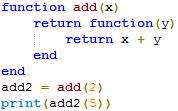


注意上面的执行结果表明了闭包f1、f2、g1和g2都共有同一个upvalue，这是因为在创建inner1、inner2这两个闭包被创建时堆栈上根本没有找到n的踪影，而是直接使用闭包foo的upvalue。

t=test(2015)之后，t这个闭包一定已经吧n妥善保存好了，之后f1、f2如果在当前对战上没有找到n就会自动到他们的外包闭包的upvalue引用数组中去找，并把找到的应用值拷贝到自己的upvalue引用数组中，所以f1、f2、g1和g2引用的upvalue实际也是同一个变量，而刚才描述的搜索机制则确保了最后他们的upvalue应用都会指向同一个地方。

**闭包的实现原理**

当lua编译一个函数时，它会生成一个原型（prototype），原型中包含函数的虚拟机指令、函数中的常量（数值和字符串等）和一些调试信息。在任何时候只要lua执行一个funcion…end表达时，他都会创建一个新的闭包。每个闭包都有一个相应函数原型的引用以及一个数组，数组中每个元素都是一个对upvalue的引用，可以通过该数组来访问外部的局部变量。值得注意的是，在lua5.2之前，闭包中海包括一个对环境（environment）的引用，环境实质上就是一个table，函数可以在该表中索引全局变量，从lua5.2开始，取消了闭包中的环境，而引入一个变量\_ENV来设置闭包环境，由此可见，函数是编译器概念，是静态的，而闭包时运行期概念，是动态的。

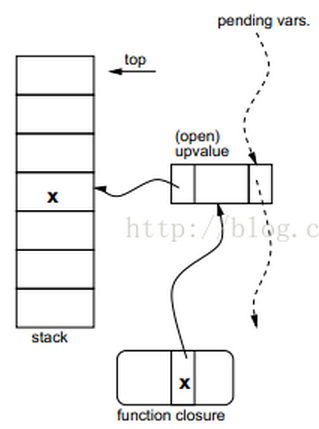


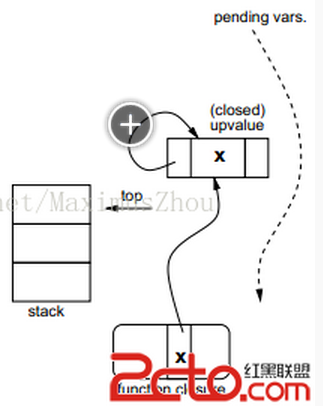


当add2被调用时，其函数体访问了外部的局部变量x（在lua中，函数参数也是局部变量）。然而，当调用add2函数时，创建add2的add函数已经返回了，如果x在栈中创建，则当add返回时，x已经不存在了（即x的存储空间被回收了）

为了解决上面的问题，在lua中，使用一种称为upvalue结构来实现闭包，任何外部的局部变量都是通过upvalue来间接访问。Upvalue实际指的是变量而不是值，upvalue最初存放的是指向堆栈的指针，如下图1所示，当局部变量需要超出函数范围的时候，变量本身将被移动到upvalue内部（注意也只是在此刻才执行这个操作），如图2所示。由于访问外部局部变量都是通过upvalue间接来完成的，所以这种移动对于程序来说是透明的，对于声明这个局部变量的函数来说，它访问的是它自己的局部变量，可以直接从堆栈中获取

如果一个函数产生了多个内部闭包，那么这些闭包需要能够正确的访问相同的局部变量。为了实现这个需求，lua对于每一个堆栈的所有upvalue维护了一个链表，。当一个新的闭包被创建出来，它将遍历所有的外部局部变量。对于每一个局部变量，它都试图在链表中寻找一个匹配的upvalue，如果能够找到，则直接应用这个upvalue，否则就创建一个新的upvalue并且接入链表。注意：链表的搜索过程相当快捷，因为对于每一个局部变量只会创建一个upvalue节点，一旦upvalue不再被引用，将自动由垃圾收集器负责回收。

图1

图2

通过为每个变量最多创建一个upvalue并按需要重复利用这个upvalue，保证了未决状态（未超过声明周期）的局部变量（pending vars）能够在闭包之间正确的共享。为了保证这种唯一性，lua维护这条链表，该链表中每个节点对应一个打开的upvalue（opend upvalue）结构，打开的upvalue是指当前正指向栈局部变量的upvalue，如上图的未巨状态的局部变量链表（the pendingvars list）。当lua创建一个新的闭包时，lua会遍历当前函数所有的外部局部变量，对于每一个外部的局部变量，若在上面的链表中能找到该变量，则重复使用该打开的upvalue，否则，lua会创建一个新的打开的upvalue，并把它插入链表中。

当局部变量离开作用域（即超过变量生命周期），这个打开的upvalue就会变成关闭的upvalue（closed upvalue），并把它从链表中删除，如上面的图2所示，一旦某个关闭的upvalue不再被任何闭包所引用，那么它的存储空间就会被回收。

一个函数有可能存取其更外层函数而非直接外层函数的局部变量。在这种情况下，当创建闭包时，这个局部变量可能不再栈中。Lua使用flat闭包（clat closures）来处理这种情况。使用flat闭包，无论何时一个函数访问一个外部的局部变量并且该变量不再直接外部函数中，该变量也会进入直接外部函数的闭包中。当一个函数被实例化时，其对应闭包的所有变量要么在直接外部函数的栈中，要么在直接外部函数的闭包中。