**闭包和lambda表达式**

lambda和闭包最早可以追溯到lambda演算，lambda演算是上世纪30年代由Alonzo Church创造的，而我们就从这里开始说起。

lambda演算可以说是一种最简单的编程语言，你只可以用它来做的唯一的事情是：

* **应用**：将一个表达式应用到另一个表达式，表示f x。（把它当作是函数调用，其中f是函数，x是它的唯一参数）
* **抽象**：它可以绑定一个符号，改符号可以看作是一个“插槽”、空白的框、或者说一个“变量”。它是用希腊字母λ（lambda）加上符号名称（例如x）跟着一个点，最后加上一个表达式组成。然后将表达式转换为期望一个参数的函数。  
  例如：λx.x + 2表示包含一个x + 2的表达式，并且表示表达式中符号x是一个绑定变量（bound variable），它可以用你提供的值作为参数来替换。  
  注意，这种方式定义的函数是匿名的，所以你还不能引用它，但是你可以立即调用它(看应用的定义)，方式是提供它正在等待的参数，比如这样：（λx.x+ 2）7。然后，用表达式7（这种情况下是个字面值）被替换到x，作用在lambda子表达式x + 2上，所以你得到 7 + 2，最后你通过简单的算术规则得到9。

所以，我们解决了一个谜题：  
**lambda本质就是一个匿名函数，例如上面的λx.x + 2**

好的，现在是时候解决另一个谜题了：什么是闭包。为了做到这一点，我们来谈谈lambda表达式中的符号（变量）。

正如我所说，lambda 抽象的的做法是在它的子表达式中绑定一个符号，以便它称为一个可替代的参数。这样的符号称为约束的（bound）。但是如果表达式中还有其他符号呢？例如λx.x/y+2。这这个表达式中，符号x是被lambda抽象λx约束的。但是另一个符号y不受限制，他是自由的。我们不知道它是什么以及它来自哪里，所以我们不知道它代表什么，有什么价值，因此我们不能评估（evaluate）这个表达式直到我们找出y代表的意义。

事实上，与其他两个符号2和+一样。只是我们对这两个符号非常熟悉，通常会忘记计算机并不知道它们，我们需要通过在某处定义它们来告诉计算机它们的含义。例如，在库或在语言本身。

你可以想象自由符号是定义在表达式外面的地方，在它“周围的语境”（“surrounding context”）中，这称为环境（environment）。环境可能是一个更大的表达式，这是表达式的一部分。或者是在某个库，或者在语言本身（作为原生的）。

这让我们将lambda表达式分为两类：

* CLOSED expressions：这些表达式中出现的每一个符号都受到一些lambda抽象的约束。换句话说，它们是自己自足的，不需要评估任何周边语境。它们也被称为Combinators。
* OPEN expressions：这些表达式中的某些符号没有约束，也就是说，它们中的一些符号是自由的，它们需要一些外部信息，因此只有在提供这些符号的定义后才能对它们进行评估。

你可以通过提供一个环境来关闭一个开放的lambda表达式，该环境通过将所有的自由符号绑定到某些值（可能是数字，字符串，匿名函数或者说lambda等）来定义它们。

然后这里是闭包的部分：  
**lambda表达式的闭包是定义在外部上下文（环境）中特定的符号集，它们给这个表达式中的自由符号赋值。它将一个开放的、仍然包含一些未定义的符号lambda表达式变为一个关闭的lambda表达式，使这个lambda表达式不再具有任何自由符号。**

例如：你有一下这个lambda表达式：λx.x/y+2，符号x是受约束的，而y是自由的，因此这个表达式是开放的并且是不能被评估的，除非你说出y的意思（+和2也一样，是自由的）。假设你有一个环境，如下：

{y: 3, +: [built-in addition], 2: [built-in number], q: 42, w: 5}

这个环境为我们的lambda表达式提供了所有未定义（自由）的符号y, + , 2，还有一些额外的符号q, w。而我们需要定义的是这个环境的子集：

{ y: 3, +: [built-in addition], 2: [built-in number] }

然后这个子集正是我们lambda表达式的闭包。

换句话说，它关闭了一个开放的lambda表达式，这就是closure这个术语一开始出现的地方。还有这就是为什么关于这个问题很多人的回答都不是很正确的原因。

那么，他们为什么是错误的？为什么很多人说闭包是内存中的一些数据结构，或者是他们使用的语言的一些功能？或者为什么他们将闭包与lambdas混淆？

那么，Sun / Oracle，微软，Google等企业（ corporate marketoids）就应该收到指责，因为他们说这些是他们语言的结构（Java，C#，GO 等）。他们经常叫“闭包”，这应该只是lambda表达式（译者：没错，什么ruby，groovy就让我以为lambda表达式就是闭包，真的好气）。或者称“闭包”是他们用来实现词法作用域的一种特定技术，即一个函数可以访问它作用域外定义的变量。他们经常说函数“封闭（encloses）”这些变量，即捕获他们到某些数据结构去存储它们，防止它们在外部函数执行完后被销毁。但是这些只是他们民俗语言和营销，这只会让事情更加混乱，因为每一个语言供应商都使用它们自己的术语。

而且更糟糕的是，因为它们所说的话总是有一点是真实的，这让你不允许轻易的将其视为假的。让我解释一下：

如果你想实现一种使用lambdas作为一等公民（first-class citizens）的语言，你需要允许它们使用在其上下文中定义的符号（即在你的lambda中使用的那些自由变量）。即使周围的函数已经返回，这些符号也必须存在。问题是这些符号是绑定在函数的本地存储（通常是在调用栈上），当函数返回时将不再存在。因此，为了让lambda按照你期望的方式工作，你需要以某种方式从外部上下文中“捕获”所有这些自由变量并且保存它们，即使外部上下文将消失。也就是说，你需要找到该lambda表达式的闭包（所有使用到的变量）并且将它存储在其他地方（建立一个副本，或者提前准备一个除了栈以外的空间给它们）。你用来实现这个目标的实际方法就是你语言的“实现细节”。这里最重要的是闭包，它是一组从环境中获取的自由变量，你需要保存在某处。

人们开始调用他们语言中使用的实际的数据结构来实现“闭包”并没有花太长时间。该结构通常看起来是这样的：

Closure {

[pointer to the lambda function's machine code],

[pointer to the lambda function's environment]

}

并且这些数据结构可以作为参数传递给其他函数、作为函数返回、保存到变量、代表lambdas，并允许它们访问其封闭环境以及在该上下文中运行的机器码（machine code）。但这只是实现闭包的其中一种方式，而不是闭包本身。（译者：例如groovy就用一个Closure类来实现闭包）

正如我上面所解释的那样，**lambda表达式的闭包是其环境中定义的一个子集，它给包含在lambda表达式中的自由变量赋值，从而有效的关闭表达式。**（将一个开放的还不能评估的lambda表达式，转换成一个关闭的lambda表达式，然后可以进行评估，因为它包含的所有符号现在都已定义）。

其他任何东西都只是程序员和语言供应商们的“巫术”和“诅咒魔术”，并不知道这些概念的真正根源。（译者：原文"cargo cult" and "voo-doo magic"，科学家为了描述那些缺乏科学严谨的研究也创造了这些词汇：巫毒科学（voodoo science）和巫术科（cargocult science））

**3. 历史考据**

事实上，在翻译上面的StackOverflow的回答之前，我就已经几乎找全了资料，并且写了几天的文章，但是总是因为对编程语言设计的不熟悉导致卡壳，直到我找到这个回答，它抛弃了历史考据部分，直接开始讲述闭包的来源，而且我还知道这个回答才是最正确的。我将用我能找到的一些资料来概述一下闭包的来源，下面是我参考过的一些资料，部分资料需要梯子：

[Neal Gafter's blog——definition of closures](https://link.jianshu.com/?t=https%3A%2F%2Fgafter.blogspot.co.uk%2F2007%2F01%2Fdefinition-of-closures.html)   
[维基百科——闭包](https://link.jianshu.com/?t=https%3A%2F%2Fzh.wikipedia.org%2Fwiki%2F%25E9%2597%25AD%25E5%258C%2585_%28%25E8%25AE%25A1%25E7%25AE%2597%25E6%259C%25BA%25E7%25A7%2591%25E5%25AD%25A6%29)   
[维基百科——LISP](https://link.jianshu.com/?t=https%3A%2F%2Fzh.wikipedia.org%2Fwiki%2FLISP)   
[Joseph Weizenbaum——FUNARG Problem Explained](https://link.jianshu.com/?t=http%3A%2F%2Fwww.softwarepreservation.net%2Fprojects%2FLISP%2FMIT%2FWeizenbaum-FUNARG_Problem_Explained-1968.pdf)   
[Joel Moses——The Function of FUNCTION in LISP](https://link.jianshu.com/?t=https%3A%2F%2Fdspace.mit.edu%2Fbitstream%2Fhandle%2F1721.1%2F5854%2FAIM-199.pdf)   
[LISP 1.5 Programmer's Manual.pdf](https://link.jianshu.com/?t=http%3A%2F%2Fdoc.mbalib.com%2Fview%2Fa8d2b989c61867c32e93bb07d18d1f18.html)   
[维基百科——彼得·兰丁](https://link.jianshu.com/?t=https%3A%2F%2Fzh.wikipedia.org%2Fwiki%2F%25E5%25BD%25BC%25E5%25BE%2597%25C2%25B7%25E5%2585%25B0%25E4%25B8%2581)   
[维基百科——Scheme](https://link.jianshu.com/?t=https%3A%2F%2Fzh.wikipedia.org%2Fwiki%2FScheme)   
[Scheme:A Interpreter for Extended Lambda Calculus](https://link.jianshu.com/?t=https%3A%2F%2Fwww.researchgate.net%2Fpublication%2F227098423_Scheme_A_Interpreter_for_Extended_Lambda_Calculus)   
[李文浩——闭包的概念、形式与应用](https://link.jianshu.com/?t=https%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2Fl-cn-closure%2F%23note_2)

LISP是世界上第二款高阶编程语言（第一款是IBM的FORTRAN），也是第一款函数式编程语言，诞生于1958年，它是基于lambda演算设计的。

而LISP中有一个特性，而后将函数作为一等公民的编程语言都有这个特性——Functional Arguments，也就是说函数可以作为参数存在，它可以赋值给变量，也可以作为其它函数的入参，或者作为其它函数的返回值。

但是实现Functional Arguments有一个难点，简称为**FUNARG Problem**，这是当初编程语言设计者们的难题，长达十几年的时间里有没有很好的解决方案。

而FUNARG problem问题发生在一个函数返回另一个持有自由变量的函数时。  
例如下面这一段代码：

DEFINE F(X) TO BE

LET G(Y) BE

RETURN X^2 + Y^2

END

RETURN G

END

P ← F(3)

调用F(3)返回了一个函数G，将它赋值给P，以便后面可以通过P调用这个函数。这个函数引用了一个值为3的自由变量，所以系统需要将这个binding(标识符和值的映射)记录下来。这有什么难的吗？是的，非常难，因为早期以语言都使用动态作用域，为了能继续解释FUNARG问题，这里需要先解释下动态作用域和静态作用域。

我们先来说说词法作用域（lexical scoping），也可以称为（static scoping）静态作用域。  
现在的大部分语言都使用的是词法作用域，词法作用域的意思是：**变量的作用范围取决于它在源代码中定义的位置，在词法解析阶段即可确定**（例如static修饰符）  
例如在java中：

class Foo {

int a = 1;

void foo() {

int b = 2;

}

}

a的作用范围在整个Foo对象中，而b的作用范围仅仅只在foo函数中。

而动态作用域则相反，**变量的作用范围取决于函数的调用链**，例如：

DEFINE foo() TO BE

PRINT A

END

DEFINE bar(X) TO BE

A = X + 1

foo()

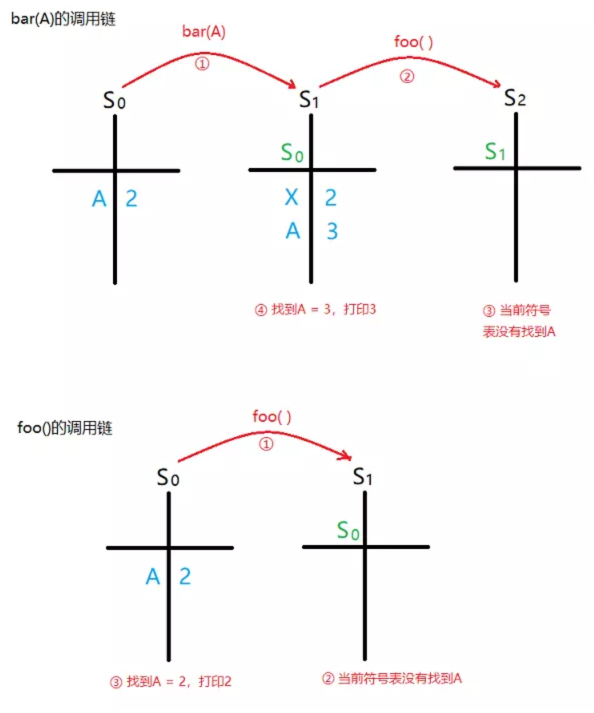
END

A = 2

bar(A) // 打印 3

foo() // 打印 2

同样是打印A，但是通过不同调用路径，最终打印出不同的值。  
为了更直观的描述动态作用域，这里画了一张图：在动态作用域的语言中，binding一般都使用符号表堆栈记录（symbol table stack），当函数调用时都会为它创建一张新的表记录去它当中定义的局部变量和形参，防止和函数外部的相同标识符混淆，例如S0和S1都拥有A这个符号，但是它们分别保存着不同的值，互不影响。另外，每张符号表都会和它的上一张表关联，形成一条调用链。而函数就是通过回溯这条调用链查找变量的值，如果所有表中都找不到该标识符，那么就会抛出一个error。



好了，说完动态作用域的问题，那么我们回到FUNARG问题中。之前说到，系统需要将P代表的函数中引用到的自由变量的binding记录下来，但是想保存这组binding是非常困难的，因为要防止命名冲突的情况，否则每次调用P的执行结果都可能不一致。这就是所谓的FUNARG问题，它是程序设计者们需要解决的。

应该是1964年左右,因为Peter J Landin(后面简称兰丁)在这时候提出了闭包概念

在当时，为了解决FUNARG问题，在LISP的某些版本和兰丁的SEGD用符号表树（symbol table tree）代替符号表栈（symbol table stack），为了讲述什么是闭包，我们来看下面这种情况。

LET G(X) BE

LET F(Z) BE

RETURN Z^2 + X

END

RETURN F

END

FA ← G(1)

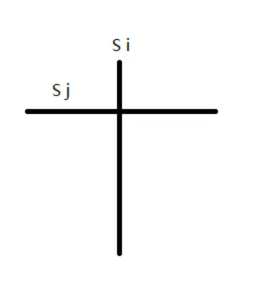
FB ← G(2)

FA(3) // 期望结果是10

FB(3) // 期望结果是11

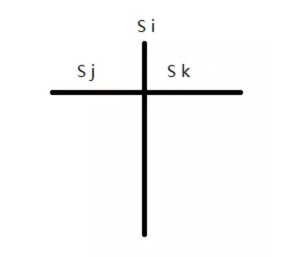
如上面的代码所示，FA和FB都持有一个自由变量X，而我们期望FA的值是10，而FB的值是11，虽然它们拥有相同的入参，但是它们持有的自由变量X的值不一样。而如何保存自由变量X的值呢？当时所有语言的都是使用符号表栈来保存binding，这意味着FUNARG问题无法解决，更何况这里需要保存三个不同的X，而解决这一方案的方法就是符号表树。

符号表栈在解释动态作用域的时候就已经画过图。



Si是当前符号表的名字，Sj则是上一张符号表的名字，下半部分则是binding的信息。

而符号表树的图就有点区别, 多了个Sk区域，表示可以从Sk中找需要的binding，而不是顺着Sj回去找。



好了，我们回来这段代码：

LET G(X) BE

LET F(Z) BE

RETURN Z^2 + X

END

RETURN F

END

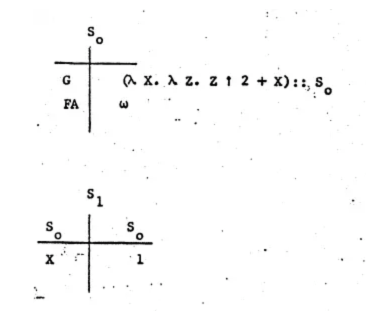
FA ← G(1)

FB ← G(2)

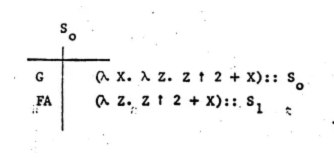
FA(3) // 期望结果是10

FB(3) // 期望结果是11

当G(1)调用时，符号表是这样的：

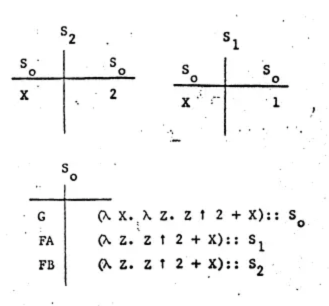


(λx. λz. z ^ 2 + x)::S0，函数可以用lambda的形式存储在栈中，S0则表示这个函数中的binding在S0中搜索。w则表示这个值还未初始化。当G(1)执行时，创建一个新的表S1。



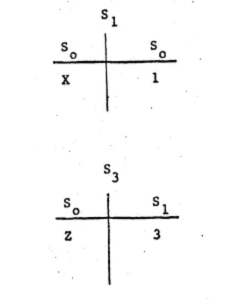
当G(1)执行完毕后FA也有了个值，它引用了S1这张表，而S1中保存着X = 1这个binding。

同理，当G(2)执行后，如下：



FB也引用了一张S2的表，其中保存着x = 2这个binding。

当FA(3)调用时，新建了一张S3的表：



从S1中查找到自由变量X的值，最终计算出结果为10。  
同理，FB的值为11。

S0引用着S1和S2这两张子表，这就是所谓的符号表树。而引用着子表的lambda表达式也被称为closure。

来做个简单的总结：  
**一个lambda表达式，没有绑定其它环境时，我们称它为open lambda，而绑定了其它环境的lambda表达式，我们称它们为闭包（closure），评估一个open lambda的结果就是一个闭包。**

实际上，上面图中 (λx. x + y)::Si格式表示的值，都属于闭包，包括G，FA，FB。

而关于动态作用域其实还有另一个知识我们需要知道：**deep binding（深约束）和shallow binding（浅约束）。**   
在一开始的有关动态作用域的说明中，按照函数调用顺序遍历找到最近的与之相关的符号，我们称之为浅约束。  
**而将函数作为参数时，需要将自由变量在单独的一个环境中保存，然后该函数引用这个单独的环境，我们称之为深约束。而这样捆绑起来的整体也就是我们所说的闭包。**  
**换句话说，动态作用域中解决FUNARG问题的技术是深约束，而深约束中使用到的环境和函数这个整体称为闭包。**

深约束的实现是非常困难的，另外程序员使用起来也是非常困难的，或者说非常容易出错，因为它需要使用到特殊的关键字标记函数或变量。

而将闭包发扬光大的是Scheme语言（1975年），Scheme是首个使用了词法作用域的LISP方言，据说它评估lambda表达式都会产生闭包，不需要程序员们考虑深约束浅约束的问题，不需要太关心自由变量引发的FUNARG问题。所以闭包也被称为词法闭包，解析器可以通过作用域自己判断产生闭包而不需要通过预定义的关键字去标记它。

**4. 总结**

总的来说，如果想解释闭包是什么，有两种意义，分别是形式上的和概念上的。

* **概念上的闭包**：在实现深约束（解决FUNARG问题）的过程中，函数需要引用到一个环境，而函数和这个环境形成的整体我们称为闭包。可以说闭包无处不在，例如对象。
* **形式上的闭包**：词法上下文中引用了自由变量的函数，在不同语言中有不同的表现形式，并且衍生了很多运用方式，比如隐藏数据，作为简易对象使用。

相关的一些概念：

* **lambda表达式**：可以用来表示函数的语法糖，本质是一个匿名函数。
* **动态/词法作用域**：动态作用域中变量的作用范围和函数的调用顺序和定义方式有关，运行时才能确定。而词法作用域中，变量的作用范围是在源代码中就可以确定的。
* **深约束**：动态作用域中为了解决FUNARG问题的技术，将引用环境和函数绑定在一起，函数会在绑定的环境中查找binding，实现起来非常困难。
* **浅约束**：动态作用域的查找binding的默认工作方式，函数通过遍历调用过程，找到最近的binding。