26 面试必问：嵌套函数、闭包与装饰器-慕课专栏

Python 函数可以嵌套定义，我们先考察一个典型的例子：

```
def adder(n):
    def handler(x):
        return n+x
    return handler
```

adder 函数负责创建处理函数 *handler* ，处理函数计算参数 x 与固定值 *n* 的和。因此，如果我们需要一个为参数加上 *1* 的函数，调用 *adder(1)* 即可轻松得到：

```
>>> add1 = adder(1)
>>> add1(10)
11
>>> add1(15)
16
```

同样，如果我们需要一个将参数加上 *5* 的函数，调用 *adder(5)* 即可：

```
>>> add5 = adder(5)
>>> add5(10)
15
>>> add5(15)
20
```

很显然，对于 *add1* 来说，*n* 的值是 *1* ；对于 *add5* 来说，*n* 的值是 *5* ；两者保存独立，互不干扰。

理论上，当函数 *adder* 返回，局部变量 *n* 应该就被回收了，为什么 *handler* 函数还能访问到它呢？

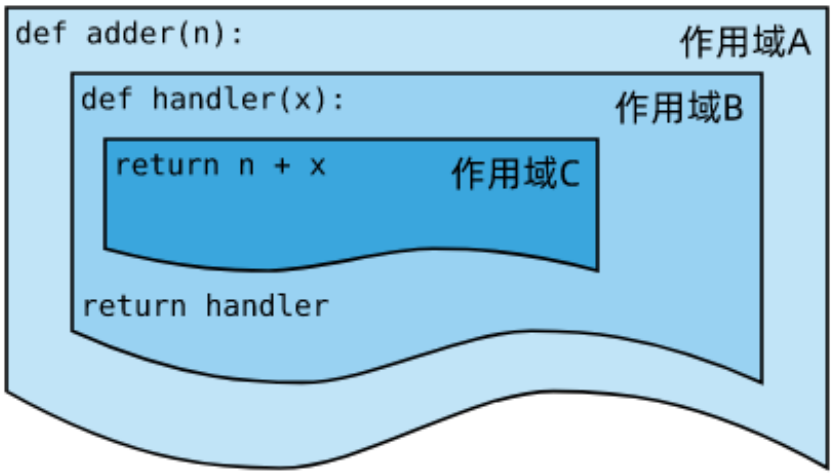
另外，像 *adder* 函数和 *handler* 函数这种嵌套写法，到底有什么作用？适用于什么开发场景？有什么需要特别注意的地方吗？

为了解答这诸多疑惑，我们需要深入学习 **嵌套函数** 与 **闭包** 变量的来龙去脉。

嵌套函数

像 *adder* 函数和 *handler* 这样，在一个函数的函数体内定义另一个函数，就构成了 **嵌套函数** 。我们在 **虚拟机** 部分讲解 **作用域** 时，已对嵌套函数有所了解。你还记得嵌套函数与作用域之间密切的联系吗？

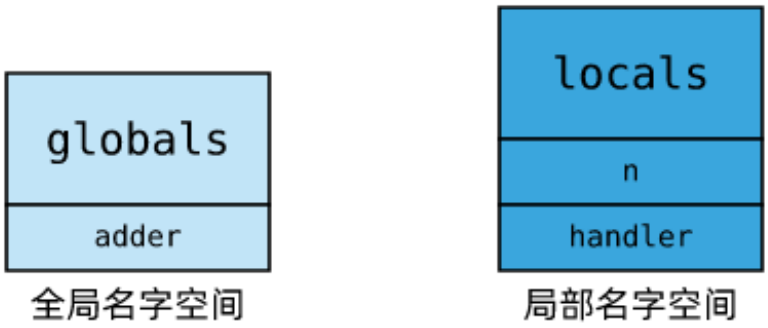
根据我们在虚拟机部分学到的知识，*adder-handler* 这段简单的代码却包含着 *3* 个不同的作用域：



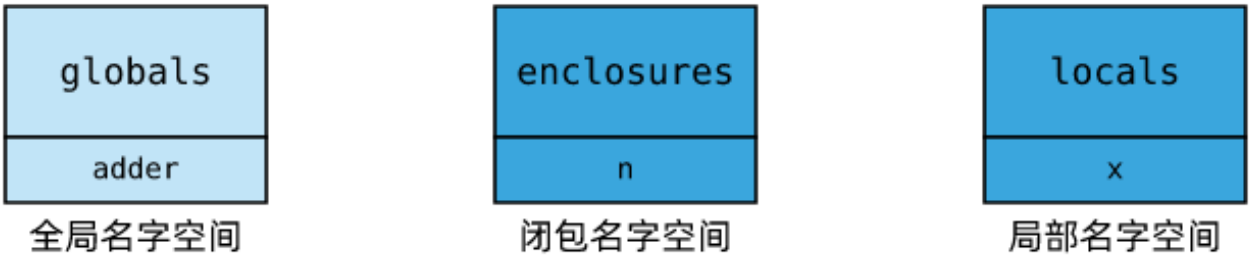
作用域是一个静态概念，由 *Python* 代码语法决定，与编译后产生的 **代码对象** 一一对应。作用域规定了能够被某个代码块访问的变量有哪些，但对变量具体的值则一概不关心。

一旦 *Python* 程序开始运行，虚拟机需要为作用域中的变量分配一定的存储空间，这就是 **名字空间** 。名字空间依照作用域规则实现，它决定了某个变量在运行时的取值，可以看做是作用域在运行时的动态表现方式。

当 *adder* 函数执行时，作用域 *A* 在虚拟机中表现为 **全局** 名字空间，作用域 *B* 表现为 **局部** 名字空间：



当 *handler* 函数执行时，例如调用 *adder(10)* 时，作用域 *A* 在虚拟机中表现为 **全局** 名字空间，作用域 *B* 表现为 **闭包** 名字空间：作用域 *C* 表现为 **局部** 名字空间：



全局与局部这两个名字空间我们已经非常熟悉了，那么 **闭包** 名字空间又该如何理解呢？

闭包

闭包 (*closure*) 是 **词法闭包** (*Lexical Closure*) 的简称，是指引用了自由变量的函数。这些被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。

以 *adder(10)* 为例，它是一个 *handler* 函数对象，闭包变量 *n* 值总是 *10* 。那么，内层函数是如何访问闭包作用域的呢？我们对函数代码对象进行反汇编，从中可以看出端倪：

```
>>> add10 = adder(10)
>>> add10
<function adder.<locals>.handler at 0x10dc2b6a8>
>>> add10.__code__
<code object handler at 0x10dbe5150, file "<stdin>", line 2>
>>> dis.dis(add10.__code__)
  3           0 LOAD_DEREF               0 (n)
              2 LOAD_FAST                0 (x)
              4 BINARY_ADD
              6 RETURN_VALUE
```

我们发现一条全新的字节码 *LOAD_DEREF* ，正是它执行了闭包变量查找工作！*LOAD_FAST* 指令则负责局部变量查找，而局部名字空间我们已经有所了解，它藏身于栈帧对象中。

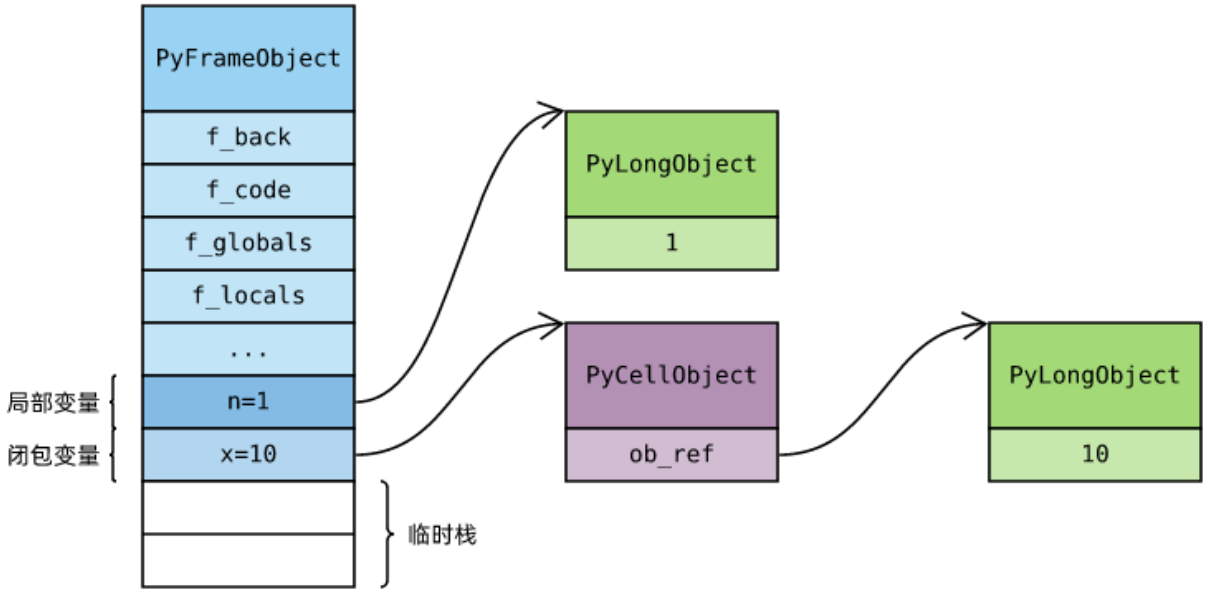
那么，闭包名字空间又藏在哪呢？

顺着虚拟机处理 *LOAD_DEREF* 字节码的代码，不难找到答案。与该字节码处理相关的源码位于 *Python/ceval.c* 文件，有兴趣的同学可以深入看一下。

因篇幅所限，这里直接给出答案：当闭包函数 *handler* 执行时，闭包变量藏身于 *PyFrameObject* 中。

还记得吗？每次函数调用虚拟机都会创建一个 *PyFrameObject* 对象，用于保存函数调用上下文。全局名字空间与局部名字空间都藏身其中，闭包名字空间也不例外。

前面章节提过，*PyFrameObject* 结构体最后部分是不固定的，依次存放着静态局部名字空间、闭包名字空间以及临时栈。以 *add10(1)* 为例，函数运行时 *PyFrameObject* 状态如下如下：



我们重点关注 *PyFrameObject* 末尾处，局部变量、闭包变量以及临时栈依次排列。

由于函数局部变量、闭包变量个数在编译阶段就能确定，运行时并不会增减，因此无须用 *dict* 对象来保存。相反，将这些变量依次排列保存在数组中，然后通过数组下标来访问即可。这就是所谓的静态名字空间。

对于局部变量 *n* ，数组对应的槽位保存着整数对象 *1* 的地址，表示 *n* 与 *1* 绑定。而闭包变量 x 则略有差别，槽位不直接保存整数对象 *10* ，而是通过一个 *PyCellObject* 间接与整数对象 *10* 绑定。

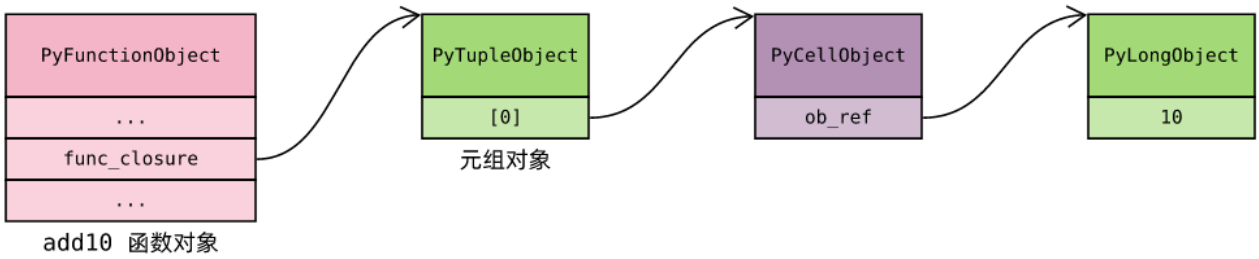
那么，闭包变量是如何完成初始化的呢？

我们知道全局名字空间 *f_globals* 来源于函数对象，那么闭包名字空间是否也是如此呢？再次研究 *CALL_FUNCTION* 字节码处理逻辑，印证了我们的猜测：

函数对象 *PyFunctionObject* 中有一个字段 *func_closure* ，保存着函数所有闭包变量。我们可以通过名字 *__closure__* 可以访问到这个底层结构体字段：

```
>>> add10.__closure__
(<cell at 0x10dc09e28: int object at 0x10da161a0>,)
```

这是一个由 *PyCellObject* 组成的元组，*PyCellObject* 则保存着闭包变量的值。当函数调用发生时，*Python* 虚拟机创建 *PyFrameObject* 对象，并从函数对象取出该元组，依次填充相关静态槽位。



接着我们来考察 *PyCellObject* 对象的行为，通过 *cell_contents* 属性可以访问闭包变量值：

```
>>> add10.__closure__[0]
<cell at 0x10dc09e28: int object at 0x10da161a0>
>>> add10.__closure__[0].cell_contents
10
```

我们尝试将函数闭包变量进行修改，发现函数的行为将受到影响：

```
>>> add10.__closure__[0].cell_contents = 20
>>> add10(1)
21
```

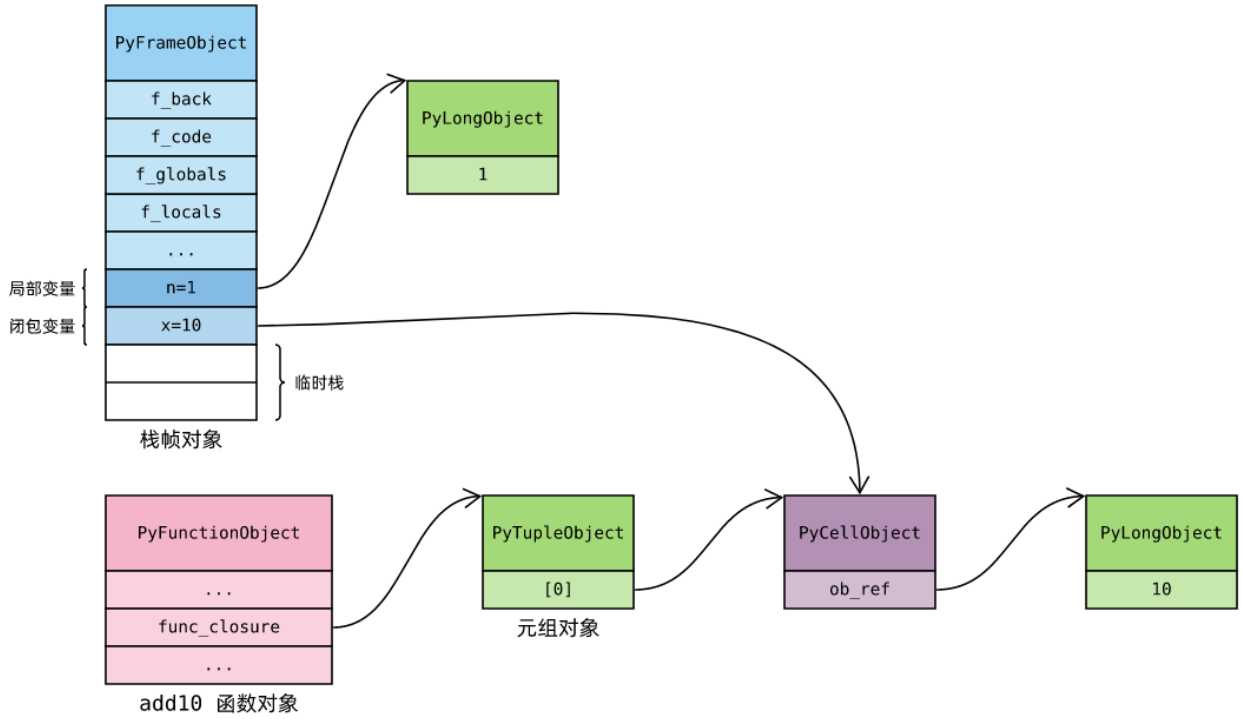
这种行为虽合乎常理，但千万不要在实际项目中应用，不然可能会踩坑！

至此，一切都明了了！从 *adder* 函数字节码告诉我们，它将 *handler* 函数所有闭包变量存为 *tuple* 对象，再执行 *MAKE_FUNCTION* 字节码完成函数对象创建。该元组最终将作为 *closure* 参数传给函数对象，并保存在 *func_closure* 字段中。

```
>>> dis.dis(adder)
  2           0 LOAD_CLOSURE          0 (n)
              2 BUILD_TUPLE           1
              4 LOAD_CONST            1 (<code object handler at 0x102864150, file "<stdin>", line 2>)
              6 LOAD_CONST            2 ('adder.<locals>.handler')
              8 MAKE_FUNCTION         8
             10 STORE_FAST            1 (handler)

  4          12 LOAD_FAST             1 (handler)
             14 RETURN_VALUE
```

那么，为什么闭包变量要通过 *PyCellObject* 间接引用呢？我们将函数对象与运行时的栈帧对象放在一起来讨论：



最新的 *Python* 提供了 *nonlocal* 关键字，支持修改闭包变量。如果没有 *PyCellObject* ，函数在运行时直接修改 *PyFrameObject* ，函数返回就被回收了。借助 *PyCellObject* ，函数在运行时修改的是 *ob_ref* 。这样一来，就算函数返回，修改还是随函数而存在。

装饰器

借助闭包，我们可以让函数具备搭积木的魔法，例如：函数调用前后做一些统一的处理操作。

事不宜迟，我们来实践一下，实现 *log_call* 函数，在指定函数调用前后各输出一句话：

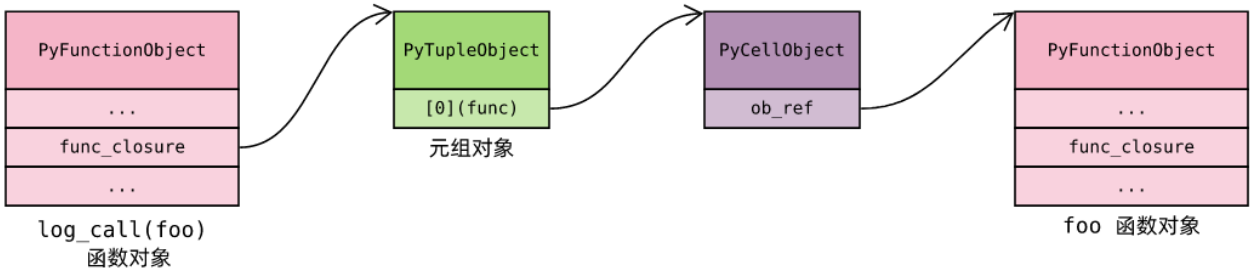
```
def log_call(func):
    def handler(*args, **kwargs):
        print('start to call')
        result = func(*args, **kwargs)
        print('end of call')
        return result
    return handler
```

log_call 接受一个函数对象 *func* 作为参数，这就是我们想为其注入魔法，以便在调用前后输出提示的函数，姑且称为 **原函数** 。它返回一个闭包函数 *handler* ，姑且称为 **代理函数** ，闭包变量 *func* 与 *handler* 紧紧绑定在一起。

当 *handler* 函数被调用时，它先输出提示，再调用原函数 *func* ，函数调用完毕再次输出提示。注意到，由于原函数参数是未知的，因此 *handler* 通过可变参数 *args 以及 ***kwargs* 进行传递。

现在，我们将 *log_call* 用起来！对于任意的函数 *foo* ，*log_call* 轻松即可为它注入魔法：

```
>>> def foo():
...     print('foo processing')
...
>>> log_call(foo)()
start to call
foo processing
end of call
```



如果我想想为其他函数，例如 bar 注入 *log_call* 魔法，可以这样实现：

```
def bar():
    print('bar processing')

bar = log_call(bar)

bar()
```

这代码未免太丑陋了！为此，*Python* 引入了语法糖 *@xxxx* ：

```
@log_call
def bar():
    print('bar processing')

bar()
```

这段代码与上面那段完全等价，却更加优雅、清晰！

像 *log_call* 这样，为其他函数注入新功能的函数，就是所谓的 **装饰器** (*decorator*)。因篇幅所限，我们仅吃了一味开胃菜，诸多细节来不及展开，后续章节将进一步深入讨论。