# 《源码探秘 CPython》63. 闭包是怎么实现的?

原创 古明地觉 古明地觉的编程教室 2022-04-07 09:00



收录于合集
#CPython
97个 >

## 楔子

我们之前一直反复提到四个字：**名字空间**。一段代码执行的结果不光取决于代码中的符号，更多的是取决于代码中符号的语义，而这个运行时的语义正是由名字空间决定的。

名字空间是由虚拟机在运行时动态维护的，但是有时我们希望将名字空间静态化。换句话说，我们希望有的代码不受名字空间变化带来的影响，始终保持一致的功能该怎么办呢？随便举个例子：

```
def login(user_name, password, user):
    if not (user_name == "satori" and password == "123"):
        return "用户名密码不正确"
    else:
        return f"欢迎: {user}"

print(login("satori", "123", "古明地觉"))  # 欢迎: 古明地觉
print(login("satori", "123", "古明地恋"))  # 欢迎: 古明地恋
```

我们注意到每次都需要输入username和password，于是我们可以通过使用嵌套函数来设置一个基准值：

```
def wrap(user_name, password):
    def login(user):
        if not (user_name == "satori" and password == "123"):
            return "用户名密码不正确"
        else:
            return f"欢迎: {user}"
    return login

login = wrap("satori", "123")
print(login("古明地觉"))  # 欢迎: 古明地觉
print(login("古明地恋"))  # 欢迎: 古明地恋
```

尽管函数login 里面没有user_name和password这两个局部变量，但是不妨碍我们使用它，因为外层函数 warp 里面有。

也就是说，函数 login作为函数 wrap的返回值被返回的时候，有一个名字空间(wrap的local名字空间)就已经和 login 紧紧地绑定在一起了。执行内层函数login的时候，在自己的local空间找不到，就会从和自己绑定的local空间里面去找，这就是一种将名字空间静态化的方法。这个名字空间和内层函数捆绑之后的结果我们称之为闭包(closure)。

> 为了描述方便，上面说的是 local 空间，但我们知道，局部变量不是从那里查找的，而是从 f_localsplus 里面。只是我们可以按照 LEGB 的规则去理解，这一点心理清楚就行。

也就是说：闭包=外部作用域+内层函数。并且在介绍函数的时候提到，PyFunctionObject 是虚拟机专门为字节码指令的传输而准备的大包袱，global名字空间、默认参数都和字节码指令捆绑在一起，同样的，也包括闭包。

## 实现闭包的基石

闭包的创建通常是利用嵌套函数来完成的，在PyCodeObject中，与嵌套函数相关的属性是co_cellvars和co_freevars，两者的具体含义如下：

- **co_cellvars：通常是一个tuple，保存了外层作用域中被内层作用域使用的变量的名字；**
- **co_freevars：通常是一个tuple，保存了内层作用域中使用的外层作用域的变量的名字；**

光看概念的话比较抽象，实际演示一下：

```python
def foo():
    name = "古明地觉"
    age = 16
    gender = "female"

    def bar():
        nonlocal name
        nonlocal age
        print(gender)
    return bar

print(foo.__code__.co_cellvars)  # ('age', 'gender', 'name')
print(foo().__code__.co_freevars)  # ('age', 'gender', 'name')
print(foo.__code__.co_freevars)  # ()
print(foo().__code__.co_cellvars)  # ()
```

无论是外层函数还是内层函数都有co_cellvars 和 co_freevars，这是肯定的，因为都是函数。但是无论是co_cellvars还是co_freevars，得到结果是一样的。

只不过外层函数需要使用 co_cellvars 获取，因为它包含的是外层函数中被内层函数使用的变量的名称；内层函数需要使用 co_freevars 获取，它包含的是内层函数中使用的外层函数的变量的名称。

如果使用外层函数获取co_freevars的话，那么得到的结果显然就是个空元组了。除非 foo 也作为某个函数的内层函数，并且内部使用外层函数的某个变量，同理内层函数也是一样的道理。

那么问题来了，闭包所需要的空间申请在哪个地方呢？没错，显然是 f_localsplus，这块内存被分成了四份，分别用于：局部变量、cell对象（指针）、free对象（指针）、运行时栈。

> 之前一直说的是 cell 对象、free 对象，但准确来说它们都是对象的指针，所以用 cell 变量、free 变量来描述或许更合适一些。

而在通过_PyFrame_New_NoTrack创建栈帧的时候，里面有一行代码泄漏了天机。

```
PyFrameObject* _Py_HOT_FUNCTION
_PyFrame_New_NoTrack(PyThreadState *tstate, PyCodeObject *code,
                     PyObject *globals, PyObject *locals)
{
    //......
        Py_ssize_t extras, ncells, nfrees;
        //获取 cell 对象的个数
        ncells = PyTuple_GET_SIZE(code->co_cellvars);
        //获取 free 对象的个数
        nfrees = PyTuple_GET_SIZE(code->co_freevars);
    //......
        f->f_code = code;
        //局部变量 + cell 对象 + free 对象所占的内存空间
        extras = code->co_nlocals + ncells + nfrees;
        //剩余那一份留给运行时栈，由于栈帧还未创建完毕，显然目前栈是空的
        //f->f_localsplus + extras 指向运行时栈的栈底（目前也是栈顶）
        f->f_valuestack = f->f_localsplus + extras;
        //内存初始化，由于都是 PyObject *，所以初始化为 NULL
        for (i=0; i<extras; i++)
```

```
        for (i=0; i<extras; i++)
            f->f_localsplus[i] = NULL;
        //local 名字空间也是 NULL
        f->f_locals = NULL;
        f->f_trace = NULL;
    //......
}
```



所以闭包同样是以静态的方式实现的。

## 闭包的实现过程

在介绍了实现闭包的基石之后，我们可以开始追踪闭包的具体实现过程了，当然还是要先看一下闭包对应的字节码。

```
s = f"""
def get_func():
    value = "inner"

    def func():
        print(value)
    return func

show_value = get_func()
show_value()
"""

if __name__ == '__main__':
    import dis
    dis.dis(compile(s, "<file>", "exec"))
```



以上是一个简单的闭包，字节码如下：

```
############ 模块的字节码 #############
# 加载函数 get_func 的 PyCodeObject、以及函数名 "get_func"
0 LOAD_CONST               0 (<code object get_func at 0x00......>)
2 LOAD_CONST               1 ('get_func')
# 构造函数
4 MAKE_FUNCTION            0
# 将得到的 PyFunctionObject 的指针使用变量 get_func 保存
6 STORE_NAME               0 (get_func)
# 加载 get_func
8 LOAD_NAME                0 (get_func)
# 函数调用
10 CALL_FUNCTION            0
# 从栈顶弹出返回值，使用变量 show_value 保存
# 如果我们没有用变量保存，那么这条指令就会变成 POP_TOP
12 STORE_NAME               1 (show_value)

# 加载 show_value
14 LOAD_NAME                1 (show_value)
# 函数调用
```

```
16 CALL_FUNCTION            0
# 从栈顶弹出返回值，然后丢弃
18 POP_TOP
# return None
20 LOAD_CONST               2 (None)
22 RETURN_VALUE

############ 外层函数 get_func 的字节码 ##############
Disassembly of <code object get_func at 0x00......>:
# 加载字符串常量 "inner"
0 LOAD_CONST                1 ('inner')
# 使用符号 value 保存，但这里使用的是 STORE_DEREF，一会分析
2 STORE_DEREF               0 (value)
# 又是新指令，一会分析
4 LOAD_CLOSURE              0 (value)
# 构造一个元组，显然是要存储什么东西，一会分析
6 BUILD_TUPLE               1
# 加载内层函数 func 的 PyCodeObject 对象
8 LOAD_CONST                2 (<code object func at 0x00......>)
# 加载内层函数的全限定名，注意是全限定名
# 因此构造函数的时候，使用的是 __qualname__、不是 __name__
# 由于函数 func 是一个内层函数，所以 __name__ 等于 "func"
# 但是 __qualname__ 等于 "get_func.<locals>.func"
10 LOAD_CONST               3 ('get_func.<locals>.func')
# 构造 PyFunctionObject
12 MAKE_FUNCTION            8 (closure)
# 使用符号 func 保存
14 STORE_FAST               0 (func)
# return None
16 LOAD_FAST                0 (func)
18 RETURN_VALUE

############ 内层函数 func 的字节码 ##############
Disassembly of <code object func at 0x00......>:
# 加载变量 print
0 LOAD_GLOBAL               0 (print)
# 加载变量 value，使用的是 LOAD_DEREF
2 LOAD_DEREF                0 (value)
# 函数调用
4 CALL_FUNCTION             1
# 从栈顶弹出返回值
6 POP_TOP
# return None
8 LOAD_CONST                0 (None)
10 RETURN_VALUE
```



里面的大部分指令都见过了，但是有三个例外，分别是 STORE_DEREF、LOAD_CLOSURE、LOAD_DEREF。

我们先看 STORE_DEREF 和 LOAD_DEREF，显然它们也是用来保存和加载一个变量，对于当前这个例子来说，变量就是 value。因此很容易得出结论，如果一个局部变量被内层函数所引用，那么指令将不再是 XXX_FAST，而是 XXX_DEREF。

而还有一个指令叫 LOAD_CLOSURE，它是做什么用的呢？我们一会说，总之此时已经在为闭包的构建添砖加瓦了。


我们知道虚拟机在执行CALL_FUNCTION指令时，会进入 _PyFunction_FastCallDict 中。

```
1  //frameobject.c
```

```
PyObject *
_PyFunction_FastCallDict(PyObject *func, PyObject *const *args, Py_ssize
_t nargs,
                         PyObject *kwargs)
{
    //......
    if (co->co_kwonlyargcount == 0 &&
        (kwargs == NULL || PyDict_GET_SIZE(kwargs) == 0) &&
        (co->co_flags & ~PyCF_MASK) == (CO_OPTIMIZED | CO_NEWLOCALS | CO
_NOFREE))
    //......
}
```

如果对于当前的 get_func 而言，由于存在内层函数，并且变量还被内层函数所引用，所以不会进入快速通道，而是会进入 _PyEval_EvalCodeWithName。

因此在_PyEval_EvalCodeWithName中，虚拟机会如同处理默认参数一样，将co_cellvars中的东西拷贝到新创建的PyFrameObject的f_localsplus里面。

```
PyObject *
_PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyObject *loc
als,
           PyObject *const *args, Py_ssize_t argcount,
           PyObject *const *kwnames, PyObject *const *kwargs,
           Py_ssize_t kwcount, int kwstep,
           PyObject *const *defs, Py_ssize_t defcount,
           PyObject *kwdefs, PyObject *closure,
           PyObject *name, PyObject *qualname)
{
    //......
    for (i = 0; i < PyTuple_GET_SIZE(co->co_cellvars); ++i) {
        //声明一个指针，指向 Cell 对象
        PyObject *c;
        Py_ssize_t arg;
        /* 处理被嵌套函数共享的外层函数的局部变量 */
        if (co->co_cell2arg != NULL &&
            (arg = co->co_cell2arg[i]) != CO_CELL_NOT_AN_ARG) {
            //创建 Cell 对象
            c = PyCell_New(GETLOCAL(arg));
            SETLOCAL(arg, NULL);
        }
        else {
            c = PyCell_New(NULL);
        }
        if (c == NULL)
            goto fail;
        //拷贝到 f_localsplus 的第二段内存中
        SETLOCAL(co->co_nlocals + i, c);
    }
    //......
    return retval;
}
```

嵌套函数有时候很复杂，如果嵌套的层数比较多的话：

```
def foo1():
    def foo2():
        x = 1
        def foo3():
            x = 2
            def foo4():
                print(x)
            return foo4
        return foo3
```

```
    return foo2


foo1()()()()
"""
2
"""
```

但是无论多少层，我们之前说的结论是不会变的。并且 Cell 对象在底层也是一个对象，那它必然也是一个 PyObject，我们看一下它的定义：

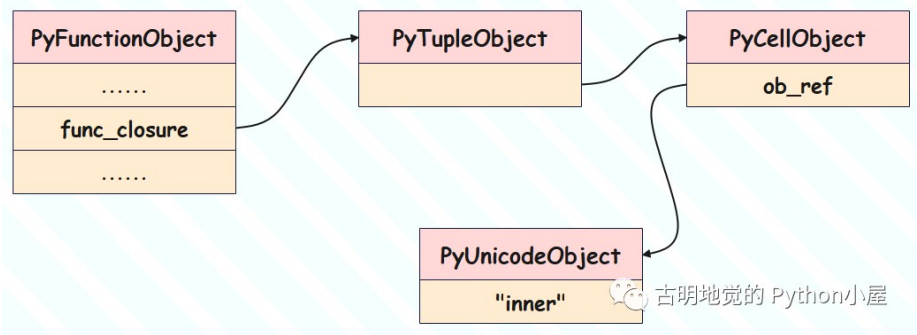
```
//cellobject.h
typedef struct {
    PyObject_HEAD
    PyObject *ob_ref;
} PyCellObject;
```

这个对象出乎意料的简单，仅仅维护了一个头部、和一个ob_ref(指向某个对象的指针)。

```
//cellobject.c
PyObject *
PyCell_New(PyObject *obj)
{
    //声明一个PyCellObject对象
    PyCellObject *op;

    //为这个PyCellObject申请空间，类型是PyCell_Type
    op = (PyCellObject *)PyObject_GC_New(PyCellObject, &PyCell_Type);
    if (op == NULL)
        return NULL;
    //这里的obj是什么呢？
    //显然是_PyEval_EvalCodeWithName里面的GETLOCAL(arg)或者NULL
    //说白了，就是我们之前说的那些被内层函数引用的外层函数的局部变量
    //如果没人引用的话就是NULL
    op->ob_ref = obj;
    Py_XINCREF(obj);
    //闭包也是可变对象，可能发生循环引用
    //因此要被 GC 跟踪
    _PyObject_GC_TRACK(op);
    return (PyObject *)op;
}
```

但实际上一开始并不知道这个ob_ref指向谁，什么时候才知道呢？是在我们一开始的闭包代码中，那句 value='inner' 执行的时候，才会真正知道ob_ref指向的是谁。

随后这个cell对象被拷贝到了新创建的PyFrameObject对象的f_localsplus中，并且位置是co->co_nlocals+i，说明在f_localsplus中，cell对象的位置是在局部变量之后的，这完全符合我们之前说的f_localsplus的内存布局。




我们看到闭包的变量是放在一个元组里面的，所以在一开始的指令里面出现了一个BUILD_TUPLE。

```
############ 外层函数 get_func 的字节码 #############
```

```
Disassembly of <code object get_func at 0x......>:
  0 LOAD_CONST               1 ('inner')
  2 STORE_DEREF              0 (value)
  4 LOAD_CLOSURE             0 (value)
  6 BUILD_TUPLE              1
```

因为内层函数使用了外层函数的一个局部变量，所以元组的长度是 1。

但是我们发现了一个奇怪的地方，那就是这个 cell 变量好像没有设置名字诶，它明明叫 value 的。实际上这和我们之前提到的虚拟机对局部变量的访问方式从基于字典的查找变成了基于数组的索引访问是一个道理。

在 get_func 这个函数执行的过程中，对 value 这个 cell 变量的查找是在f_localsplus中基于索引完成的，因此完全不需要知道 cell 变量的名字。

cell 变量的名字实际上是在处理被内层函数引用的外层函数的参数时产生的，我们说参数和内部的创建的变量都是局部变量，在处理参数的时候，就把value 这个 cell 变量一并处理了。

在处理了 cell变量之后，虚拟机将正式进入PyEval_EvalFrameEx，从而正式开始对函数 get_func 的调用过程。再看一下字节码：

```
  ############ 外层函数 get_func 的字节码 ##############
Disassembly of <code object get_func at 0x......>
  0 LOAD_CONST               1 ('inner')
  2 STORE_DEREF              0 (value)
  4 LOAD_CLOSURE             0 (value)
  6 BUILD_TUPLE              1
  8 LOAD_CONST               2 (<code object func at 0x......>)
 10 LOAD_CONST               3 ('get_func.<locals>.func')
 12 MAKE_FUNCTION            8 (closure)
 14 STORE_FAST               0 (func)
 16 LOAD_FAST                0 (func)
 18 RETURN_VALUE
```

执行LOAD_CONST之后，会将字符串'inner'压入运行时栈，紧接着便执行一条我们从未见过的全新字节码指令 STORE_DEREF：

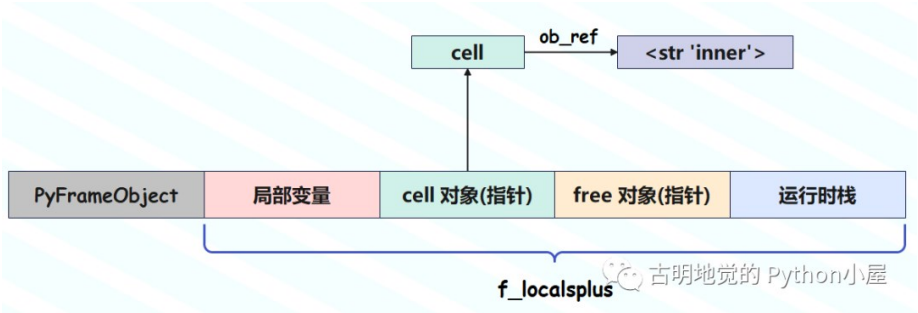
```
case TARGET(STORE_DEREF): {
    //这里pop弹出的显然是字符串'inner'
    PyObject *v = POP();
    //获取cell变量
    PyObject *cell = freevars[oparg];
    //获取老的cell对象
    PyObject *oldobj = PyCell_GET(cell);
    //我们看到了一个PyCell_SET，那么玄机肯定就在这里面了
    PyCell_SET(cell, v);
    Py_XDECREF(oldobj);
    DISPATCH();
}
```

ob_ref指向的对象似乎就是通过PyCell_SET设置的，没错，这家伙就是干这个勾当的。

```
//cellobject.h
#define PyCell_SET(op, v) (((PyCellObject *)(op))->ob_ref = v)

//cellobject.c
int
PyCell_Set(PyObject *op, PyObject *obj)
{
    PyObject* oldobj;
    if (!PyCell_Check(op)) {
        PyErr_BadInternalCall();
```

```
        return -1;
    }
    oldobj = PyCell_GET(op);
    Py_XINCREF(obj);
    PyCell_SET(op, obj);
    Py_XDECREF(oldobj);
    return 0;
}
```

如此一来，get_func 对应的栈帧的 f_localsplus 就发生了变化。




```
def get_func():
    value = "inner"

    def func():
        print(value)
    return func

show_value = get_func()
show_value()
```

此时在get_func的环境中，value 符号对应着一个PyUnicodeObject对象，但closure要将这个约束进行冻结，为了在嵌套函数func被调用的时候还可以使用这个约束。

因此工具人PyFunctionObject就又登场了，在执行接下来的 def func() 表达式对应的字节码时，虚拟机就会将<value, 'inner'>这个约束塞到PyFunctionObject中。而相应的指令就是 LOAD_CLOSURE：

```
case TARGET(LOAD_CLOSURE): {
    PyObject *cell = freevars[oparg];
    Py_INCREF(cell);
    PUSH(cell);
    DISPATCH();
}
```

LOAD_CLOSURE 会将刚刚放置好的PyCellObject *（cell 对象的指针）取出，并压入运行时栈，紧接着BUILD_TUPLE指令将PyCellObject *打包进一个PyTupleObject。显然这个元组可以存放多个PyCellObject *，只不过我们的例子中只有一个。

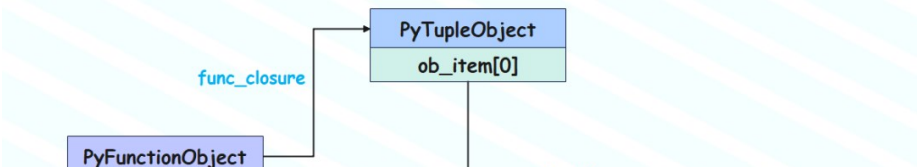
```
  ############ 外层函数 get_func 的字节码 ##############
Disassembly of <code object get_func at 0x......>:
  0 LOAD_CONST               1 ('inner')
  2 STORE_DEREF              0 (value)
  4 LOAD_CLOSURE             0 (value)
  #构造元组，压入运行时栈
  6 BUILD_TUPLE              1
  8 LOAD_CONST               2 (<code object func at 0x......>)
 10 LOAD_CONST               3 ('get_func.<locals>.func')
 12 MAKE_FUNCTION            8 (closure)
 14 STORE_FAST               0 (func)
 16 LOAD_FAST                0 (func)
 18 RETURN_VALUE
```

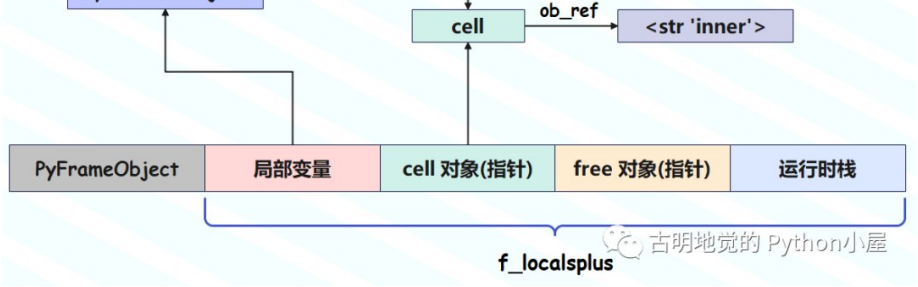
随后虚拟机又通过两个LOAD_CONST将内层函数func对应的PyCodeObject、和函数名LOAD进来，压入运行时栈，紧接着以一个MAKE_FUNCTION指令完成约束和PyCodeObject之间的绑定。

注意这里的指令依旧是MAKE_FUNCTION，但参数是8，我们再次看看MAKE_FUNCTION这个指令，还记得它在哪里吗？没错，之前说了只要是字节码指令，都在ceval.c中。

```
TARGET(MAKE_FUNCTION) {
    //弹出名字:get_func.<locals>.func
    //这里的名字是全限定名 __qualname__
    PyObject *qualname = POP();
    //弹出PyCodeObject
    PyObject *codeobj = POP();
    //以PyCodeObject对象、global命名空间、名字为参数
    //构造出PyFunctionObject
    PyFunctionObject *func = (PyFunctionObject *)
        PyFunction_NewWithQualName(codeobj, f->f_globals, qualname);

    Py_DECREF(codeobj);
    Py_DECREF(qualname);
    if (func == NULL) {
        goto error;
    }
    //此时运行时栈中还剩下一个元组
    //而我们看到参数是8，因此这个条件是成立的
    if (oparg & 0x08) {
        assert(PyTuple_CheckExact(TOP()));
        //弹出闭包需要使用的变量信息，也就是元组
        //并写入到func_closure中
        func ->func_closure = POP();
    }

    //这是处理注解的:只在python3.6+中存在
    if (oparg & 0x04) {
        assert(PyDict_CheckExact(TOP()));
        func->func_annotations = POP();
    }

    //处理关键字参数
    if (oparg & 0x02) {
        assert(PyDict_CheckExact(TOP()));
        func->func_kwdefaults = POP();
    }

    //处理默认参数
    if (oparg & 0x01) {
        assert(PyTuple_CheckExact(TOP()));
        func->func_defaults = POP();
    }

    //压入运行时栈
    PUSH((PyObject *)func);
    DISPATCH();
}
```

此时便将约束(内层函数需要使用的变量信息)和内层函数绑定在了一起，然后执行STORE_FAST将新创建的PyFunctionObject对象（函数 func）放置到了f_localsplus当中。这样的话，f_localsplus就又发生了变化。






从图上我们发现内层函数居然在get_func的局部变量里面，是的没有错。其实按照我们之前说的，函数即变量，所以函数和普通变量一样，都是在上一级栈帧的f_localsplus里面。

最后这个新建的PyFunctionObject对象被压入到了上一级栈帧的运行时栈中，并且被作为上一个栈帧的返回值返回了。显然有人就能猜到下一步要做什么了，既然拿到了闭包、或者说内层函数对应的PyFunctionObject，那么肯定要使用啊。

## 使用闭包

closure是在get_func函数中被创建的，而对closure的使用，则是在func中。

执行show_value时，因为func对应的PyCodeObject的co_flags域中包含了CO_NESTED，因此在不会进入快速通道function_code_fastcall。

不过问题是，虚拟机是怎么知道co_flags域中包含了CO_NESTED呢？

```
def get_func():
    value = "inner"

    def func():
        print(value)
    return func

show_value = get_func()
print(show_value.__code__.co_flags)  # 19
```

我们看到func函数的co_flags是19，那么这个值是什么计算出来的呢？我们在介绍PyCodeObject对象和pyc文件那一章中提到，co_flags这个域主要用于mask，用来判断参数和函数类型的。

```
//code.h
#define CO_OPTIMIZED    0x0001
#define CO_NEWLOCALS    0x0002
#define CO_VARARGS      0x0004
#define CO_VARKEYWORDS  0x0008
#define CO_NESTED       0x0010
#define CO_GENERATOR    0x0020
#define CO_NOFREE       0x0040
#define CO_COROUTINE            0x0080
#define CO_ITERABLE_COROUTINE   0x0100
#define CO_ASYNC_GENERATOR      0x0200
```

函数没有参数，显然CO_VARARGS和CO_VARKEYWORDS是不存在的：

```
print(0x0001 | 0x0002 | 0x0010)  # 19
# 因此闭包包含CO_NESTED
```

处理逻辑会进入通用通道，看一下里面和闭包相关的逻辑：

```
//ceval.c
PyObject *
_PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyObject *loc
als,
```

```
            PyObject *const *args, Py_ssize_t argcount,
            PyObject *const *kwnames, PyObject *const *kwargs,
            Py_ssize_t kwcount, int kwstep,
            PyObject *const *defs, Py_ssize_t defcount,
            PyObject *kwdefs, PyObject *closure,
            PyObject *name, PyObject *qualname)
{
    Py_ssize_t i, n;
    /* Copy closure variables to free variables */
    for (i = 0; i < PyTuple_GET_SIZE(co->co_freevars); ++i) {
        PyObject *o = PyTuple_GET_ITEM(closure, i);
        Py_INCREF(o);
        freevars[PyTuple_GET_SIZE(co->co_cellvars) + i] = o;
    }
    //...
    //...
    //...
}
```

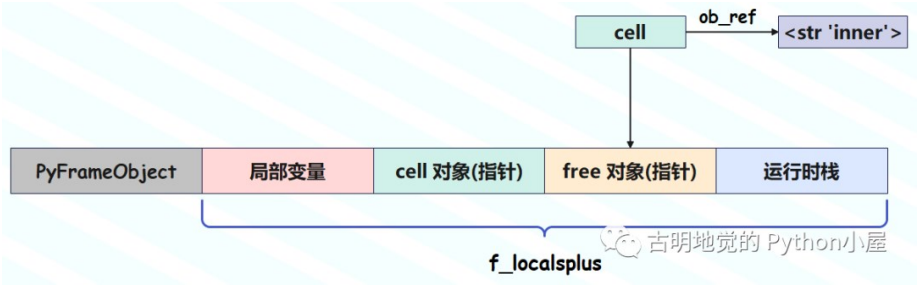
其中的closure变量是作为倒数第三个参数传递进来的，我们可以看看到底传递了什么？

```
//funcobject.h
#define PyFunction_GET_CLOSURE(func) \
        (((PyFunctionObject *)func) -> func_closure)


PyObject *
_PyFunction_FastCallDict(PyObject *func, PyObject *const *args, Py_ssize
_t nargs,
                         PyObject *kwargs)
{
    //......
    result = _PyEval_EvalCodeWithName((PyObject*)co, globals, (PyObject
*)NULL,
                                      args, nargs,
                                      k, k != NULL ? k + 1 : NULL, nk, 2,
                                      d, nd, kwdefs,
                                      closure, name, qualname);
    Py_XDECREF(kwtuple);
    return result;
}
```

我们看到是把PyFunctionObject对象的func_closure拿出来了，显然这个func_closure就是PyFunctionObject对象中的、装满了PyCellObject *的元组。

然后在_PyEval_EvalCodeWithName中，进行的动作就是将这个PyTupleObject里面的PyCellObject *一个一个地放到f_localsplus中相应的位置，注意：此时是内层函数func对应的栈帧的f_localsplus。

在处理完之后，func对应的栈帧的f_localsplus就变成了这样。




所以外层函数在构建内层函数时，会将 cell 变量打包成一个元组，交给内层函数的func_closure成员。然后执行内层函数创建栈帧的时候，再将func_closure中的 cell 变量拷贝到f_localsplus 的第三段内存中。当然对于内层函数而言，此时它应该叫做 free 变量。

> 外层函数获取 co_cellvars，内层函数获取 co_freevars

```
def get_func():
    value = "inner"

    def func():
        print(value)
    return func


show_value = get_func()
# func_closure指的是内层函数的func_closure，所以：
print(get_func.__closure__)  # None
print(show_value.__closure__)  # (<cell at 0x00000.....>,)

# 我们看到外层函数的__closure__为None
# 内层函数的__closure__则不是None
# 因此相当于将所有的cell对象(指针)拷贝了一份，存在了free区域
print(show_value.__closure__[0].cell_contents)  # inner
```

而在调用内层函数func的过程中，当引用外层作用域的符号时，一定是到f_localsplus里面的free区域（第三段内存）去获取对应PyCellObject *。然后通过内部的 ob_ref 进而获取符号对应的值。

这也正是func函数中print(value)表达式对应的第一条字节码指令LOAD_DEREF 0的意义，从 free 区域中获取索引为 0 的元素。

```
case TARGET(LOAD_DEREF): {
    //获取PyCellObject对象
    PyObject *cell = freevars[oparg];
    //获取PyCellObject对象的ob_ref指向的对象
    PyObject *value = PyCell_GET(cell);
    if (value == NULL) {
        format_exc_unbound(tstate, co, oparg);
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);//压入运行时栈
    DISPATCH();
}
```

此外通过闭包，我们还可以玩出一些新花样，但是工作中不要这么做。

```
def get_func():
    value = "inner"

    def func():
        print(value)
    return func

show_value = get_func()
show_value()  # inner

show_value.__closure__[0].cell_contents = "内层函数"
show_value()  # 内层函数
```

以上就是闭包相关的内容，总的来说不算太复杂。

> 内层函数访问外层函数中的变量，依旧是静态访问的，只不过是在 f_localsplus 的第三段内存（free 区域）里面访问；而普通的局部变量，是在 f_localsplus 的第一段内存（局部变量区域）里面访问。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》64. 装饰器是怎么实现的?

下一篇 ›
《源码探秘 CPython》62. 函数的 local 名字空间

喜欢此内容的人还喜欢

浅谈Kotlin协程及首页弹窗中的应用
洋钱罐技术团队


20行Python代码破解了网站登入
Red Teams


python 7天进阶之路-参数args,kwargs
缪斯之子