《源码探秘 CPython》62. 函数的 local 名字空间

原创 古明地觉 古明地觉的编程教室 2022-04-06 09:00

在看完函数的参数解析之后，我们来聊一聊函数的 local 名字空间。

我们知道函数的参数和函数内部定义的变量都属于局部变量，均是通过静态的方式访问的。

```
x = 123

def foo1():
    global x
    a = 1
    b = 2

# a 和 b 是局部变量, x 是全局变量, 因此是 2
print(foo1.__code__.co_nlocals)  # 2


def foo2(a, b):
    pass

print(foo2.__code__.co_nlocals)  # 2


def foo3(a, b):
    a = 1
    b = 2
    c = 3

print(foo3.__code__.co_nlocals)  # 3
```

无论是参数还是内部新创建的变量，本质上都是局部变量。并且我们发现如果函数内部定义的变量和参数名称一致，那么参数就没用了。

这很好理解，因为本质上就相当于重新赋值了，此时外面无论给函数foo2的参数a、b传什么值，最终都会变成 1 和 2。所以其实局部变量的实现机制和函数参数的实现机制是一致的。

按照之前的理解，当访问一个全局变量时，会去访问 global 名字空间，而这也确实如此。但是当访问函数的局部变量时，是不是访问其内部的 local 名字空间呢？

之前我们说过 Python 变量的访问是有规则的，按照**本地**、**闭包**、**全局**、**内置**的顺序去查找，所以当然会首当其冲去 local 名字空间里面查找啊。

但不幸的是，在调用函数期间，虚拟机通过**_PyFrame_New_NoTrack**创建栈帧对象时，这个至关重要的 local 名字空间并没有被创建。

```
//frameobject.c
PyFrameObject* _Py_HOT_FUNCTION
_PyFrame_New_NoTrack(PyThreadState *tstate, PyCodeObject *code,
                     PyObject *globals, PyObject *locals)
{
    //...
        f->f_locals = NULL;
        f->f_trace = NULL;
    //...
}
```

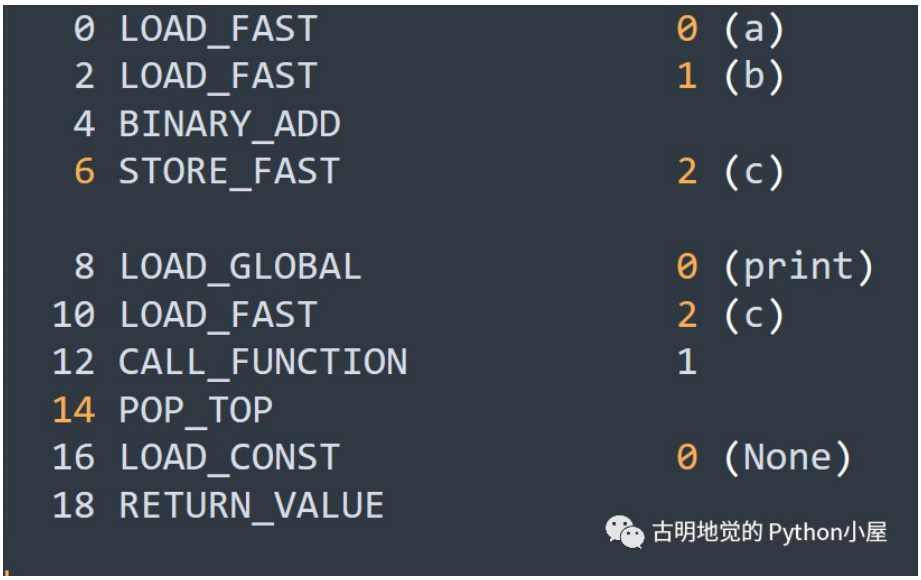
对于模块而言，它的 f_locals和 f_globals指向是同一个PyDictObject；但对于函数而言，f_locals 却是 NULL。那么问题来了，这些重要的符号到底存储在什么地方呢？

> 显然我们知道是存储在co_varnames中，但你们就装作不知道配合我一下好吧(#^.^#)

我们先来举个栗子：

```
def foo(a, b):
    c = a + b
    print(c)
```

它的字节码如下：



栈帧的 f_localsplus 这段连续内存（数组）是给四个老铁使用的，分别是：局部变量、cell对象、free对象、运行时栈。而我们看到字节码偏移量为 6 和 10 的两条指令分别是：STORE_FAST 和 LOAD_FAST，所以它和我们之前分析参数的时候是一样的，都是存储在 f_localsplus 的第一段内存中。

此时我们对局部变量 c 的藏身之处已经了然于心，但是为什么函数的实现没有使用 local 名字空间呢？答案很简单，因为函数内部的局部变量有多少，在编译的时候已经确定了，个数是不会变的。因此编译时就能确定局部变量占用的内存大小，也能确定访问局部变量的字节码指令应该如何访问内存。

```
def foo(a, b):
    c = a + b
    print(c)

print(foo.__code__.co_varnames)  # ('a', 'b', 'c')
```

我们看到符号 c 位于符号表中索引为 2 的位置（编译时就已确定），那么通过 f_localsplus[2] 即可拿到变量 c 对应的值。

这个过程是基于数组索引实现的静态查找，它的效率非常高。而 local 空间是一个字典，虽然字典也是经过高度优化的，但肯定没有静态查找快。

因此，尽管虚拟机为函数实现了 local 空间（初始为 NULL，后续访问的时候会进行填充），但是在变量查找时却没有使用它，原因就是为了更高的效率，而且函数是一等公民，使用频率很高。

> 结论：虽然查找的时候是按照 LEGB 规则，但其实局部变量是静态访问的，不过完全可以按照 LEGB 的方式来理解。

我们从 Python 的层面来演示一下：

```python
x = 1

def foo():
    globals()["x"] = 2

foo()
print(x)  # 2
```

我们在函数内部访问了 global 名字空间，而 global 空间全局唯一，在 Python 层面上就是一个字典。

- **查找变量 x，等价于 globals()["x"];**
- **给变量 x 赋值为 123，等价于 globals()["x"] = 123;**

因此在执行完 foo() 之后，全局变量 x 就被修改了。但 local 名字空间也是如此吗？我们来看看：

```python
def foo():
    x = 1
    locals()["x"] = 2
    print(x)


foo()  # 1
```

我们按照相同的套路，却并没有成功，这是为什么？原因就是我们刚才解释的那样，函数内部有哪些局部变量在编译时就已经确定好了，存储在符号表 co_varnames 中，查询的时候是从 f_localsplus 中静态查找的，而不是从 locals() 中查找。

locals() 不像 globals()，虽然它们都是字典，但 globals() 全局唯一。我们调用 globals() 就直接访问到了存放全局变量的字典，一旦做了更改，肯定会影响外面的全局变量。

而 locals() 则不会，因为局部变量压根就不是从它这里访问的，尽管它和 globals() 类似，在函数中也唯一，也会随着当前的上下文动态改变。

```python
def foo(a, b):
    x = 1
    print(locals())
    print(id(locals()))
    y = 2
    print(locals())
    print(id(locals()))

foo(1, 2)
"""
{'a': 1, 'b': 2, 'x': 1}
2459571657088
{'a': 1, 'b': 2, 'x': 1, 'y': 2}
2459571657088
"""
```

我们看到真的就类似于全局名字空间一样，前后地址没有变化，但是键值对的个数在增加。因为 locals() 底层会执行 PyEval_GetLocals，实际上拿到就是当前栈帧对象的 f_locals 属性。

```c
PyObject *
PyEval_GetLocals(void)
{
    // 获取当前的线程状态对象，然后拿到里面的栈帧对象
    PyThreadState *tstate = _PyThreadState_GET();
    PyFrameObject *current_frame = _PyEval_GetFrame(tstate);
    if (current_frame == NULL) {
        _PyErr_SetString(tstate, PyExc_SystemError,
                         "frame does not exist");
```

```
        return NULL;
    }

    if (PyFrame_FastToLocalsWithError(current_frame) < 0) {
        return NULL;
    }

    assert(current_frame->f_locals != NULL);
    // 获取 local 名字空间
    return current_frame->f_locals;
}
```


所以 local 名字空间的表现和 global 名字空间是类似的，都会随着上下文动态改变。只是我们知道，局部变量不是从 local 名字空间里面访问的，不管怎么操作 locals()，都不会影响局部变量。

因此我们可以看到一个比较奇特的现象：

```
def foo(a, b):
    # 当前 local 空间只有 a 和 b
    d = locals()
    print(d)
    # 此时多了一个 d
    print(locals())
    print(d["d"] is d["d"]["d"] is d["d"]["d"]["d"])

foo(1, 2)
"""
{'a': 1, 'b': 2}
{'a': 1, 'b': 2, 'd': {...}}
True
"""
```

仔细思考一下肯定很好理解，它就有点类似 globals() 与 __builtins__ 之间的关系：

```
# __builtins__ 等价于 import builtins as __builtins__
x = 123
print(
    globals()["__builtins__"].globals()["__builtins__"].globals()["x"]
)  # 123
```

再看一个例子：

```
def foo():
    locals()["x"] = 1
    print(x)

foo()
```

此时会得到什么结果估计不用我说了，因为本地、全局、builtin 里面都没有变量 x，所以报错。尽管在locals()里面我们设置了，但局部变量的值不是从它这里获取的，而是从 f_localsplus 里面。而且查看符号表的话，会发现里面也没有 'x' 这个符号。

如果我们设置一个全局变量呢？

```
x = 123

def foo():
    locals()["x"] = 1
    print(x)

foo()  # 123
```

显然此时会访问全局变量。

我们再来搭配 exec 关键字，区别会更加明显。

```
def foo():
    print(locals())  # {}
    exec("x = 1")
    print(locals())  # {'x': 1}
    try:
        print(x)
    except NameError as e:
        print(e)  # name 'x' is not defined
foo()
```

尽管 locals() 变了，但是依旧访问不到 x，因为虚拟机并不知道 exec("x = 1") 是创建一个局部变量，它只知道这是一个函数调用。

而 exec("x = 1") 默认影响的是当前所在的作用域，所以效果就是改变了局部名字空间，里面多了一个 **"x": 1** 键值对。但关键的是，局部变量 x 不是从局部名字空间中查找的，exec 终究还是错付了人。由于函数 foo 对应的 PyCodeObject 对象的符号表中并没有 x 这个符号，所以报错了。

```
exec("x = 1")
print(x)  # 1
```

这么做是可以的，因为 exec 默认影响的是当前作用域，而这里的当前作用域就是全局作用域，所以 global 名字空间会多一个 key 为 "x" 的键值对。而全局变量是从global 名字空间中查找的，所以这里没有问题。

```
def foo():
    # 此时 exec 影响的是全局名字空间
    exec("x = 123", globals())
    # 这里不会报错，但是此时的 x 不是局部变量，而是全局变量
    print(x)

foo()
print(x)
"""
123
123
"""
```

再来看一个奇怪的问题：

```
def foo():
    exec("x = 1")
    print(locals()["x"])

foo()
"""
1
"""

def bar():
    exec("x = 1")
    print(locals()["x"])
    x = 123
```

```
14
15 bar()
16 """
17 Traceback (most recent call last):
18   File .....
19     bar()
20   File .....
21     print(locals()["x"])
22 KeyError: 'x'
23 """
```

这是什么情况？函数 bar只是多了一行赋值语句，为啥就报错了呢？要想搞懂这个问题，首先要明确两点：

- **1. 函数的局部变量在编译的时候已经确定，并存储在对应的 PyCodeObject 对象的符号表(co_varnames) 中，这是由语法规则所决定的；**
- **2. 函数内的局部变量在其整个作用域范围内都是可见的；**

为了更好地解释上面这个例子，这里再举一个常见的错误：

```
1  x = 1
2
3  def foo():
4      print(x)
5
6  def bar():
7      print(x)
8      x = 2
9
10 print(foo.__code__.co_varnames)  # ()
11 print(bar.__code__.co_varnames)  # ('x',)
```

调用函数foo没有问题，但调用 bar 的时候会报出如下错误：UnboundLocalError: local variable 'x' referenced before assignment。

原因就在于上面说的两个点，函数内的局部变量在编译的时候已经确定，当进行语法解析的时候，看到了 x=2 这样的字眼，就知道内部会存在一个名为 x 的局部变量。所以对于 bar 函数而言，符号表中是存在 "x" 这个符号的。

而函数内的局部变量在整个作用域内又都是可见的，因此对于函数bar而言，在 print(x) 的时候知道符号表中存在 "x" 这个符号。那么它也就认为局部作用域中存在 x 这个局部变量，因此就不会去找全局变量了，而是去找局部变量。

但是显然 print(x) 是在 x=2 之前发生的，所以此时 print(x) 的时候就报错了。

> UnboundLocalError: 局部变量 'x' 在赋值（x=2）之前被引用（print）了

因为 print(x) 的时候，f_localsplus中还没有对应的值与之绑定，或者说 x 此时还是一个NULL(空指针)，并没有指向一块合法的内存（已存在的 PyObject）。

当虚拟机执行到 x=2 之后，x 才会和 2 这个 PyLongObject 对象进行绑定，只可惜我们在绑定之前就使用 x 这个变量了，显然这是不合法的。可以看一下字节码：

```
def bar():
    print(x)
    x = 2

if __name__ == '__main__':
    import dis
    dis.dis(bar)
"""
  5           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (x)
```

```
              4 CALL_FUNCTION            1
              6 POP_TOP

  6           8 LOAD_CONST               1 (2)
             10 STORE_FAST               0 (x)
             12 LOAD_CONST               0 (None)
             14 RETURN_VALUE
"""
```


我们看到指令是LOAD_FAST，说明加载的是一个局部变量，但这个局部变量的赋值是发生在LOAD_FAST之后。

那么一开始的那个问题就很好解释了：

```python
def foo():
    exec("x = 1")
    print(locals())

def bar():
    exec("x = 1")
    print(locals())
    x = 123


foo()  # {'x': 1}
bar()  # {}
```

对于 foo 而言，结果符合我们的预期；但对于 bar 而言，只是多了一个赋值语句，结果局部空间就变成空字典了。

原因和 UnboundLocalError 类似，因为 'x' 已经在符号表当中了，所以 exec("x = 1") 不会再往局部空间中加入这个键值对。但如果将 bar 里面的 x=123 改成 y=123，那么显然输出的结果就是一样的了。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》63. 闭包是怎么实现的?

下一篇 ›
《源码探秘 CPython》61. *args 和 **kwargs 是如何解析的?