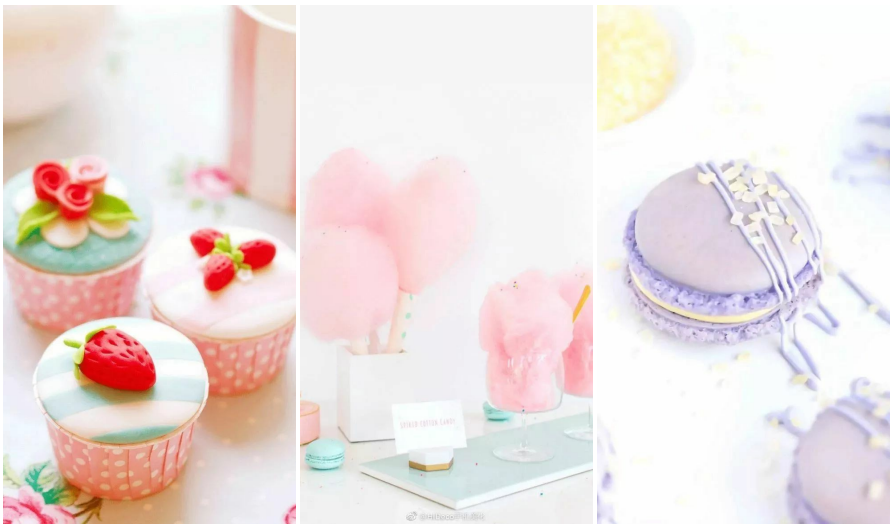
《源码探秘 CPython》47. 名字、作用域、名字空间（上）

原创 古明地觉 古明地觉的编程教室 2022-03-11 14:46



楔子

我们在PyFrameObject里面看到了3个独立的名字空间：f_locals、f_globals、f_builtins。名字空间对于Python来说是一个非常重要的概念，Python虚拟机的运行机制和名字空间有着非常紧密的联系。并且在Python中，与名字空间这个概念紧密联系在一起的还有**名字**、**作用域**这些概念，下面就来剖析这些概念是如何体现的。

变量只是一个名字

很早的时候我们就说过，从解释器的角度来看，变量只是一个泛型指针**PyObject ***，而从Python的角度来看，变量只是一个名字、或者说符号，用于和对象进行绑定的。

变量的定义本质上就是建立名字和对象之间的约束关系，所以**a = 1**这个赋值语句本质上就是将**a**和**1**绑定起来，让我们通过a这个符号可以找到对应的PyLongObject。

除了变量赋值，创建函数、类也相当于定义变量，或者说完成名字和对象之间的绑定。

```
1 def foo(): pass
2
3 class A(): pass
```

创建一个函数也相当于定义一个变量，会先根据函数体创建一个函数对象，然后将**名字foo**和**函数对象**绑定起来。所以函数名和函数体之间是分离的，同理类也是如此。

```
1 import os
```

导入一个模块，也是在定义一个变量。**import os**相当于将**名字os**和**模块对象**绑定起来，通过**os**可以找到指定的模块对象。

import numpy as np当中的as语句同样是在定义变量，将名字np和对应的模块对象绑定起来，以后就可以通过np这个名字去获取指定的模块了。

> 另外，当我们导入一个模块的时候，解释器是这么做的。比如：import os等价于os=__import__("os")，可以看到本质上还是一个赋值语句。

作用域和名字空间

我们说赋值语句、函数定义、类定义、模块导入，本质上只是完成了名字和对象之间的绑定。而从概念上讲，我们实际上得到了一个name和obj之间的映射关系，通过name可以获取对应的obj，而它们的容身之所就是名字空间。

所以名字空间是通过PyDictObject对象实现的，这对于映射来说简直再适合不过了。而在前面介绍字典的时候，我们说过字典是被高度优化的，原因就是虚拟机本身也重度依赖字典，从这里的名字空间即可得到体现。

但是一个模块内部，名字还存在可见性的问题，比如：

```
a = 1

def foo():
    a = 2
    print(a)  # 2

foo()
print(a)  # 1
```

我们看到同一个变量名，打印的确是不同的值，说明指向了不同的对象，换句话说这两个变量是在不同的名字空间中被创建的。

然后我们知道名字空间本质上是一个字典，如果两者是在同一个名字空间，那么由于字典的key的不重复性，当执行a=2的时候，会把字典里面key为**a**的value给更新成**2**。但是在外面还是打印1，这说明两者所在的不是同一个名字空间，打印的也就自然不是同一个a。

因此对于一个模块而言，内部是可能存在多个名字空间的，每一个名字空间都与一个作用域相对应。作用域就可以理解为一段程序的正文区域，在这个区域里面定义的变量是有意义的，然而一旦出了这个区域，就无效了。

对于作用域这个概念，至关重要的是要记住：它仅仅是由源程序的文本所决定的。在Python中，一个变量在某个位置是否起作用，是由它的文本位置决定的。

因此Python具有静态作用域(词法作用域)，而名字空间则是作用域的动态体现，一个由程序文本定义的作用域在Python运行时会转化为一个名字空间、即一个PyDictObject对象。而进入一个函数，显然进入了一个新的作用域，因此函数在执行时，会创建一个名字空间。

> 我们之前说，在对Python源代码进行编译的时候，对于代码中的每一个block，都会创建一个PyCodeObject与之对应。而当进入一个新的名字空间、或者说作用域时，我们就算是进入一个新的block了。
>
> 而根据我们使用Python的经验，显然函数、类都是一个新的block，当Python运行的时候会为它们创建各自的名字空间。

所以名字空间是名字、或者变量的上下文环境，名字的含义取决于名字空间。更具体的说，一个变量绑定的对象是不确定的，需要由名字空间来决定。

位于同一个作用域的代码可以直接访问作用域中出现的名字，即所谓的**直接访问**；但不同作用域，则需要通过**访问修饰符 .** 进行属性访问。

```python
class A:
    a = 1


class B:
    b = 2
    print(A.a)  # 1
    print(b)  # 2
```

如果想在B里面访问A里面的内容，要通过**A.属性**的方式，表示通过A来获取A里面的属性。但是访问B的内容就不需要了，因为都是在同一个作用域，所以直接访问即可。

访问名字这样的行为被称为**名字引用**，名字引用的规则决定了Python程序的行为。

```python
a = 1

def foo():
    a = 2
    print(a)  # 2

foo()
print(a)  # 1
```

还是上面的代码，如果我们把函数里面的**a=2**给删掉，意味着函数的作用域里面已经没有a这个变量了，那么再执行程序会有什么后果呢？从Python层面来看，显然是会寻找外部的a。因此我们可以得到如下结论：

- 作用域是层层嵌套的；
- 内层作用域可以访问外层作用域；
- 外层作用域无法访问内层作用域，尽管我们没有试，但是想都不用想。如果是把外层的a=1给去掉，那么最后面的print(a)铁定报错；
- 查找元素会依次从当前作用域向外查找，也就是查找元素时，对应的作用域是按照从小往大、从里往外的方向前进的；

LGB规则

我们说函数、类有自己的作用域，但是模块对应的源文件本身也有相应的作用域。比如：

```python
name = "古明地觉"
age = 16

def foo():
    return 123

class A:
    pass
```

由于这个文件本身也有自己的作用域，显然是global作用域，所以解释器在运行这个文件的时候，也会为其创建一个名字空间，而这个名字空间就是global名字空间。它里面的变量是全局的，或者说是模块级别的，在当前文件的任意位置都可以直接访问。

而函数也有作用域，这个作用域称为local作用域，对应local名字空间；同时Python自身还定义了一个最顶层的作用域，也就是builtin作用域，像内置函数、内建对象都在builtin里面。

这三个作用域在Python2.2之前就存在了，所以那时候Python的作用域规则被称之为LGB规则：名字引用动作沿着**local作用域(local名字空间)**、**global作用域(global名字空间)**、**builtin作用域(builtin名字空间)**来查找对应的变量。

而获取名字空间，Python也提供了相应的内置函数：

- locals函数：获取当前作用域的local名字空间，local名字空间也称为局部名字空间；
- globals函数：获取当前作用域的global名字空间，global名字空间也称为全局名字空间；
- __builtins__函数：或者import builtins，获取当前作用域的builtin名字空间，builtint名字空间也称为内置名字空间；

每个函数都有自己local名字空间，因为不同的函数对应不同的作用域，但是global名字空间则是全局唯一。

```
name = "古明地觉"

def foo():
    pass

print(globals())
# {..., 'name': '古明地觉', 'foo': <function foo at 0x000002977EDF61F0>}
```

里面的...表示省略了一部分输出，我们看到创建的全局变量就在里面。而且foo也是一个变量，它指向一个函数对象。

但是注意，我们说foo也是一个独立的block，因此它会对应一个PyCodeObject。但是在解释到**def foo**的时候，会根据这个PyCodeObject对象创建一个PyFunctionObject对象，然后将foo和这个函数对象绑定起来。

当我们调用foo的时候，再根据PyFunctionObject对象创建PyFrameObject对象、然后执行，这些留在介绍函数的时候再细说。总之，我们看到foo也是一个全局变量，全局变量都在global名字空间中。

总之，**global名字空间**全局唯一，它是程序运行时的**全局变量和与之绑定的对象**的容身之所，你在任何一个地方都可以访问到global名字空间。正如，你在任何一个地方都可以访问相应的全局变量一样。

此外，我们说名字空间是一个字典，变量和对象会以键值对的形式存在里面。那么换句话说，如果我手动地往这个global名字空间里面添加一个键值对，是不是也等价于定义一个全局变量呢？

```
globals()["name"] = "古明地觉"
print(name)  # 古明地觉


def f1():
    def f2():
        def f3():
            globals()["age"] = 16
        return f3
    return f2


f1()()()
print(age)  # 16
```

我们看到确实如此，通过往global名字空间里面插入一个键值对完全等价于定义一个全局变量。并且global名字空间是唯一的，你在任何地方调用globals()得到的都是global名字空间，正如你在任何地方都可以访问到全局变量一样。

所以即使是在函数中向global名字空间中插入一个键值对，也等价于定义一个全局变量、并和对象绑定起来。

- name="xxx" 等价于 globals["name"]="xxx";
- print(name) 等价于 print(globals["name"]);

对于**local名字空间**来说，它也对应一个字典，显然这个字典就不是全局唯一的了，每一个局部作用域都会对应自身的local名字空间。

```python
def f():
    name = "夏色祭"
    age = 16
    return locals()


def g():
    name = "神乐mea"
    age = 38
    return locals()


print(locals() == globals())  # True
print(f())  # {'name': '夏色祭', 'age': 16}
print(g())  # {'name': '神乐mea', 'age': 38}
```

显然对于模块来讲，它的**local名字空间**和**global名字空间**是一样的，也就是说，模块对应的PyFrameObject对象里面的f_locals和f_globals指向的是同一个PyDictObject对象。

但是对于函数而言，局部名字空间和全局名字空间就不一样了。调用locals是获取自身的局部名字空间，而不同函数的local名字空间是不同的。但是globals函数的调用结果是一样的，获取的都是global名字空间，这也符合**函数内找不到某个变量的时候会去找全局变量**这一结论。

所以我们说在函数里面查找一个变量，查找不到的话会找全局变量，全局变量再没有会查找内置变量。本质上就是按照自身的local空间、外层的global空间、内置的builtin空间的顺序进行查找。

因此local空间会有很多个，因为每一个函数或者类都有自己的局部作用域，这个局部作用域就可以称之为该函数的local空间；但是global空间则全局唯一，因为该字典存储的是全局变量。无论你在什么地方，通过调用globals函数拿到的永远是全局名字空间，向该空间中添加键值对，等价于创建全局变量。

对于builtin名字空间，它也是一个字典。当local空间、global空间都没有的时候，会去builtin空间查找。问题来了，builtin名字空间如何获取呢？答案是使用builtins模块，通过builtins.__dict__即可拿到builtin名字空间。

```python
# 等价于__builtins__
import builtins

#我们调用list显然是从内置作用域、也就是builtin名字空间中查找的
#但我们只写list也是可以的
#因为local空间、global空间没有的话，最终会从builtin空间中查找
#但如果是builtins.list，那么就不兜圈子了
#表示："builtin空间，就从你这获取了"
print(builtins.list is list)  # True

builtins.dict = 123
#将builtin空间的dict改成123
#那么此时获取的dict就是123
#因为是从内置作用域中获取的
print(dict + 456)  # 579

str = 123
#如果是str = 123，等价于创建全局变量str = 123
#显然影响的是global空间
print(str)  # 123
# 但是此时不影响builtin空间
print(builtins.str)  # <class 'str'>
```

这里提一下Python2当中，while 1比while True要快，为什么？

因为True在Python2中不是关键字，所以它是可以作为变量名的。那么Python在执行的时候就要先看local空间和global空间里有没有True这个变量，有的话使用我们定义的，没有的话再使用内置的True。

而1是一个常量，直接加载就可以，所以while True多了符号查找这一过程。但是在Python3中两者就等价了，因为True在Python3中是一个关键字，也会直接作为一个常量来加载。

记得之前介绍 eval 和 exec 的时候，我们说这两个函数里面还可以接收第二个参数和第三个参数，它们分别表示global名字空间、local名字空间。

```
# 如果不指定，默认当前所在的名字空间
# 显然此时是全局名字空间
exec("name = '古明地觉'")
print(name)  # 古明地觉

# 但是我们也可以指定某个名字空间
dct = {}
# 将 dct 作为全局名字空间
# 这里我们没有指定第三个参数，也就是局部名字空间
# 如果指定了全局名字空间、但没有指定局部名字空间
# 那么局部名字空间默认和全局名字空间保持一致
exec("name = 'satori'", dct)
print(dct["name"])  # satori
```

至于 eval 也是同理：

```
dct = {"seq": [1, 2, 3, 4, 5]}
try:
    print(eval("sum(seq)"))
except NameError as e:
    print(e)  # name 'seq' is not defined

# 告诉我们 seq 没有被定义
# 因为我们需要将 dct 作为名字空间
print(eval("sum(seq)", dct))  # 15
```

所以名字空间本质上就是一个字典，所谓的变量不过是字典里面的一个 key。为了进一步加深印象，再举个模块的例子：

```
# 我们自定义一个模块吧
# 首先模块也是一个对象，类型为 <class 'module'>
# 但是底层没有将这个类暴露给我们，所以需要换一种方式获取
import sys
module = type(sys)
# 以上就拿到了模块的类型对象，调用即可得到模块对象
my_module = module("自己定义的")
print(sys)  # <module 'sys' (built-in)>
print(my_module)  # <module '自己定义的'>

# 此时的 my_module 啥也没有，我们为其添砖加瓦
my_module.__dict__["name"] = "古明地觉"
print(my_module.name)  # 古明地觉
# 给模块设置属性，本质上也是操作相应的属性字典
# 当然获取属性也是如此。如果再和exec结合的话
code = """
age = 16
def foo():
```

```
    return "我是函数foo"

from functools import reduce
"""
# 此时属性就设置在了模块的属性字典里面
exec(code, my_module.__dict__)
print(my_module.age)  # 16
print(my_module.foo())  # 我是函数foo
print(my_module.reduce(int.__add__, [1, 2, 3, 4, 5]))  # 15
```

怎么样，是不是很有趣呢？当然啦，这只是名字空间的上半部分内容，下一篇文章来介绍下半部分。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》48. 名字、作用域、名字空间（下）

下一篇 ›
《源码探秘 CPython》46. 虚拟机的执行环境：栈帧对象