《源码探秘 CPython》44. 解析PyCodeObject对象

原创 古明地觉 古明地觉的编程教室 2022-03-08 08:30

心痛

获取PyCodeObject对象

在上一篇文章中，我们知道了py文件编译之后会生成PyCodeObject对象，并且还会保存在pyc文件里。那么我们在Python里面如何才能访问到这个对象呢？

首先PyCodeObject对象在Python里面的类型是**<class 'code'>**，但是这个类Python没有暴露给我们，因此**code**这个名字在Python里面只是一个没有定义的变量罢了。

但是我们可以通过其它的方式进行获取，比如函数。

```
def func():
    pass

print(func.__code__)  # <code object ......
print(type(func.__code__))  # <class 'code'>
```

我们可以通过函数的**__code__**属性拿到底层对应的PyCodeObject对象，当然也可以获取里面的成员，我们来演示一下。

co_argcount：可以通过位置参数传递的参数个数

```
def foo(a, b, c=3):
    pass
print(foo.__code__.co_argcount)  # 3

def bar(a, b, *args):
    pass
print(bar.__code__.co_argcount)  # 2

def func(a, b, *args, c):
    pass
print(func.__code__.co_argcount)  # 2
```

foo中的参数a、b、c都可以通过位置参数传递，所以结果是3；对于bar，则是两个，这里不包括***args**；而函数func，显然也是两个，因为**参数c**也只能通过关键字参数传递。

co_posonlyargcount：只能通过位置参数传递的参数个数，Python3.8新增

```python
def foo(a, b, c):
    pass

print(foo.__code__.co_posonlyargcount)  # 0

def bar(a, b, /, c):
    pass

print(bar.__code__.co_posonlyargcount)  # 2
```

注意：这里是只能通过位置参数传递的参数个数。对于 foo 而言，里面的三个参数既可以通过位置参数、也可以通过关键字参数传递；而函数 bar，里面的a、b只能通过位置参数传递。

co_kwonlyargcount：只能通过关键字参数传递的参数个数

```python
def foo(a, b=1, c=2, *, d, e):
    pass
print(foo.__code__.co_kwonlyargcount)  # 2
```

这里是d和e，它们必须通过关键字参数传递。

co_nlocals：代码块中局部变量的个数，也包括参数

```python
def foo(a, b, *, c):
    name = "xxx"
    age = 16
    gender = "f"
    c = 33

print(foo.__code__.co_nlocals)  # 6
```

局部变量有 a、b、c、name、age、gender，所以我们看到在编译之后，函数的局部变量就已经确定了，因为它们是静态存储的。

co_stacksize：执行该段代码块需要的栈空间

```python
def foo(a, b, *, c):
    name = "xxx"
    age = 16
    gender = "f"
    c = 33

print(foo.__code__.co_stacksize)  # 1
```

这个暂时不需要太关注。

co_flags：参数类型标识

如果一个函数的参数出现了 *args，那么co_flags&0x04为真；如果一个函数的参数出现了 **kwargs，那么co_flags&0x08为真；

```python
def foo1():
    pass
# 结果全部为假
print(foo1.__code__.co_flags & 0x04)  # 0
print(foo1.__code__.co_flags & 0x08)  # 0

def foo2(*args):
    pass
# co_flags & 0x04 为真，因为出现了 *args
print(foo2.__code__.co_flags & 0x04)  # 4
print(foo2.__code__.co_flags & 0x08)  # 0
```

```
12
13 def foo3(*args, **kwargs):
14     pass
15 # 显然 co_flags & 0x04 和 co_flags & 0x08 均为真
16 print(foo3.__code__.co_flags & 0x04)  # 4
17 print(foo3.__code__.co_flags & 0x08)  # 8
```

当然啦，co_flags 可以做的事情并不止这么简单，它还能检测一个函数的类型。比如函数内部出现了 yield，那么它就是一个生成器函数，调用之后可以得到一个生成器；使用 async def 定义，那么它就是一个协程函数，调用之后可以得到一个协程。

这些在词法分析的时候就可以检测出来，编译之后会体现在 co_flags 这个成员中，我们举个栗子：

```
1  # 如果是生成器函数
2  # 那么 co_flags & 0x20 为真
3  def foo1():
4      yield
5  print(foo1.__code__.co_flags & 0x20)  # 32
6
7  # 如果是协程函数
8  # 那么 co_flags & 0x80 为真
9  async def foo2():
10     pass
11 print(foo2.__code__.co_flags & 0x80)  # 128
12 # 显然 foo2 不是生成器函数
13 # 所以 co_flags & 0x20 为假
14 print(foo2.__code__.co_flags & 0x20)  # 0
15
16 # 如果是异步生成器函数
17 # 那么 co_flags & 0x200 为真
18 async def foo3():
19     yield
20 print(foo3.__code__.co_flags & 0x200)  # 512
21 # 显然它不是生成器函数、也不是协程函数
22 # 因此和 0x20、0x80 按位与之后，结果都为假
23 print(foo3.__code__.co_flags & 0x20)  # 0
24 print(foo3.__code__.co_flags & 0x80)  # 0
```

以上就是 co_flags 的作用。

co_firstlineno：代码块在对应文件的起始行

```
1 def foo(a, b, *, c):
2     pass
3
4 # 显然是文件的第一行
5 # 或者理解为 def 所在的行
6 print(foo.__code__.co_firstlineno)  # 1
```

如果函数出现了调用呢？

```
1 def foo():
2     return bar
3
4 def bar():
5     pass
6
7 print(foo().__code__.co_firstlineno)  # 4
```

如果执行foo，那么会返回函数bar，最终得到的就是bar的字节码，因此返回**def bar():**所在的行数。所以每个函数都有自己的作用域，以及PyCodeObject对象。

co_names：符号表，一个元组，保存代码块中引用的其它作用域的变量

```
c = 1

def foo(a, b):
    print(a, b, c)
    d = (list, int, str)

print(
    foo.__code__.co_names
)  # ('print', 'c', 'list', 'int', 'str')
```

一切皆对象，但看到的都是指向对象的变量，所以**print、c、list、int、str**都是变量，它们都不在当前foo函数的作用域中。

co_varnames：符号表，一个元组，保存在当前作用域中的变量

```
c = 1

def foo(a, b):
    print(a, b, c)
    d = (list, int, str)
print(foo.__code__.co_varnames)  # ('a', 'b', 'd')
```

a、b、d是位于当前foo函数的作用域当中的，所以编译阶段便确定了局部变量是什么。

co_consts：常量池，一个元组，保存代码块中的所有常量

```
x = 123

def foo(a, b):
    c = "abc"
    print(x)
    print(True, False, list, [1, 2, 3], {"a": 1})
    return ">>>"

print(
    foo.__code__.co_consts
)  # (None, 'abc', True, False, 1, 2, 3, 'a', '>>>')
```

co_consts里面出现的都是常量，但**[1，2，3]**和**{"a"：1}**却没有出现，由此我们可以得出，列表和字典绝不是在编译阶段构建的。编译时，只是收集了里面的元素，然后等到运行时再去动态构建。

不过问题来了，在构建的时候解释器怎么知道是要构建列表、还是字典、亦或是其它的什么对象呢？所以这就依赖于字节码了，解释字节码的时候，会判断到底要构建什么样的对象。

因此解释器执行的是字节码，核心逻辑都体现在字节码中。但是光有字节码还不够，它包含的只是程序的主干逻辑，至于变量、常量，则从符号表和常量池里面获取。

co_freevars：内层函数引用的外层函数的作用域中的变量

```
def f1():
    a = 1
    b = 2
    def f2():
        print(a)
    return f2

# 这里拿到的是f2的字节码
print(f1().__code__.co_freevars)  # ('a',)
```

函数**f2**引用了函数**f1**中的**变量a**。

co_cellvars：外层函数的作用域中被内层函数引用的变量，本质上和co_freevars是一样的

```
def f1():
    a = 1
    b = 2
    def f2():
        print(a)
    return f2

# 但这里调用的是f1的字节码
print(f1.__code__.co_cellvars)  # ('a',)
```

函数**f1**中的**变量a**被内层函数**f2**引用了。

co_filename：代码块所在的文件名

```
def foo():
    pass

print(foo.__code__.co_filename)  # D:/satori/main.py
```

co_name：代码块的名字

```
def foo():
    pass
# 这里就是函数名
print(foo.__code__.co_name)  # foo
```

co_code：字节码

```
def foo(a, b, /, c, *, d, e):
    f = 123
    g = list()
    g.extend([tuple, getattr, print])

print(foo.__code__.co_code)
#b'd\x01}\x05t\x00\x83\x00}\x06|\x06\xa0\x01t\x02t\x03t\x04g\x03\xa1\x01\
x01\x00d\x00S\x00'
```

这便是字节码，它只保存了要操作的指令，因此光有字节码是肯定不够的，还需要其它的静态信息。显然这些信息连同字节码一样，都位于PyCodeObject中。

co_lnotab：字节码指令与源代码行号之间的对应关系，以PyByteObject的形式存在

```
def foo(a, b, /, c, *, d, e):
    f = 123
    g = list()
    g.extend([tuple, getattr, print])

print(foo.__code__.co_lnotab)  # b'\x00\x01\x04\x01\x06\x01'
```

我们知道一行py代码会对应多条字节码指令，但事实上，co_lnotab没有直接记录这些信息，记录的是增量值。比如说：

图片

那么co_lnotab就应该是: 0 1 6 1 44 5，其中0和1很好理解，就是co_code和.py文件的起始位置。
而6和1表示字节码的偏移量增加了6，.py文件的行号增加了1；
而44和5表示字节码的偏移量增加了44，而.py文件的行号增加了5。

以上我们就分析了PyCodeObject里面的成员都代表什么含义。

compile

我们上面通过函数的__code__属性获取了该函数的PyCodeObject对象，但是还有没有其他的方法呢？显然是有的，答案是通过内置函数compile，不过在介绍compile之前，先介绍一下eval和exec。

eval：传入一个字符串，然后把字符串里面的内容拿出来。

```
a = 1
# 所以eval("a")就等价于a
print(eval("a"))  # 1

print(eval("1 + 1 + 1"))  # 3
```

注意：eval是有返回值的，返回值就是字符串里面内容。或者说eval是可以作为右值的，比如**a=eval("xxx")**。

所以eval里面一定是一个表达式，表达式计算之后是一个具体的值。绝不可以是语句，比如**a=eval("b=3")**，这样等价于**a=(b=3)**，显然这会出现语法错误。

因此eval里面把字符串剥掉之后就是一个普通的值，不可以出现诸如**if**、**def**等语句。

```
try:
    eval("xxx")
except NameError as e:
    print(e)  # name 'xxx' is not defined
```

此时等价于**xxx**，但是**xxx**没有定义，所以报错。

```
# 此时是合法的，等价于 print('xxx')
print(eval("'xxx'"))  # xxx
```

exec：传入一个字符串，把字符串里面的内容当成语句来执行，这个是没有返回值的，或者说返回值是None。

```
# 相当于 a = 1
exec("a = 1")
print(a)  # 1

statement = """
a = 123
if a == 123:
    print("a等于123")
else:
```

```
    print("a不等于123")
"""
exec(statement)  # a等于123
```

注意：**a等于123**并不是exec返回的，而是把上面那坨字符串当成普通代码执行的时候print出来的。这便是exec的作用，将字符串当成语句来执行。

那么它和eval的区别就显而易见了，eval是要求字符串里面的内容能够当成一个值，返回值就是里面的值。而exec则是直接执行里面的内容，返回值是None。

```
print(eval("1 + 1"))  # 2
print(exec("1 + 1"))  # None

# 相当于 a = 2
exec("a = 1 + 1")
print(a)  # 2

try:
    # 相当于a=2，但很明显，a=2是一个语句
    # 它无法作为一个值，因此放到eval里面就报错了
    eval("a = 1 + 1")
except SyntaxError as e:
    print(e)  # invalid syntax (<string>, line 1)
```

还是很好区分的，但是eval和exec在生产中尽量要少用。另外，eval 和 exec 还可以接收第二个参数和第三个参数，我们在介绍名字空间的时候再说。

compile：关键来了，它执行后返回的就是一个PyCodeObject对象。

这个函数接收哪些参数呢？参数一：当成代码执行的字符串；参数二：可以为这些代码起一个文件名；参数三：执行方式，支持三种，分别是exec、single、eval。

- exec：将源代码当做一个模块来编译；
- single：用于编译一个单独的Python语句(交互式下)；
- eval：用于编译一个eval表达式；

```
statement = "a, b = 1, 2"
# 这里我们选择 exec，当成一个模块来编译
co = compile(statement, "古明地觉的 Python小屋", "exec")
print(co.co_firstlineno)  # 1
print(co.co_filename)  # 古明地觉的 Python小屋
print(co.co_argcount)  # 0
# 我们是a, b = 1, 2这种方式赋值
# 所以(1, 2)会被当成一个元组加载进来
# 从这里我们看到，元组是在编译阶段就已经确定好了
print(co.co_consts)  # ((1, 2), None)

statement = """
a = 1
b = 2
"""
co = compile(statement, "<file>", "exec")
print(co.co_consts)  # (1, 2, None)
print(co.co_names)  # ('a', 'b')
```

我们后面在分析PyCodeObject的时候，会经常使用compile的方式。

然后 compile 还可以接收一个 flags 参数，也就是第四个参数，如果指定为 1024，那么得到的就不再是PyCodeObject对象了，而是一个_ast.Module 对象。

```
print(
    compile("a = 1", "<file>", "exec").__class__
)  # <class 'code'>
```

```
4
5 print(
6     compile("a = 1", "<file>", "exec", flags=1024).__class__
7 )  # <class '_ast.Module'>
```

_ast 是用 C 实现的模块，内嵌在解释器里面，用于帮助我们更好地理解Python的抽象语法树。当然，构建抽象语法树的话，我们更习惯使用标准库中的 ast 模块，它里面了导入了 _ast。

那么问题来了，这个 _ast.Module 对象能够干什么呢？别着急，我们后续在介绍栈帧的时候说。不过由于抽象语法树比较底层，对我们理解Python没有什么实质性的帮助，因此知道 compile 的前三个参数的用法即可。

字节码与反编译

关于Python的字节码，是后面剖析虚拟机的重点，现在先来看一下。我们知道Python执行源代码之前会先编译得到PyCodeObject对象，里面的co_code指向了字节码序列。

Python虚拟机会根据这些字节码序列来进行一系列的操作(当然也依赖其它的静态信息)，从而完成对程序的执行。

每个操作都对应一个**操作指令**、也叫**操作码**，总共有120多种，定义在**Include/opcode.h**中。

```
1  #define POP_TOP                   1
2  #define ROT_TWO                   2
3  #define ROT_THREE                 3
4  #define DUP_TOP                   4
5  #define DUP_TOP_TWO               5
6  #define NOP                       9
7  #define UNARY_POSITIVE           10
8  #define UNARY_NEGATIVE           11
9  #define UNARY_NOT                12
10 #define UNARY_INVERT             15
11 #define BINARY_MATRIX_MULTIPLY   16
12 #define INPLACE_MATRIX_MULTIPLY  17
13 #define BINARY_POWER             19
14 #define BINARY_MULTIPLY          20
15 #define BINARY_MODULO            22
16 #define BINARY_ADD               23
17 #define BINARY_SUBTRACT          24
18 #define BINARY_SUBSCR            25
19 #define BINARY_FLOOR_DIVIDE      26
20 #define BINARY_TRUE_DIVIDE       27
21 #define INPLACE_FLOOR_DIVIDE     28
22 ...
23 ...
```

操作指令只是一个整数，然后我们可以通过反编译的方式查看每行Python代码都对应哪些操作指令：

```
1 # Python中的dis模块专门负责干这件事情
2 import dis
3
4 def foo(a, b):
5     c = a + b
6     return c
7
8 # 里面接收一个字节码
```

```
# 当然函数也是可以的，会自动获取co_code
dis.dis(foo)
"""
  5           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 STORE_FAST               2 (c)

  6           8 LOAD_FAST                2 (c)
             10 RETURN_VALUE
"""
```

字节码反编译后的结果多么像汇编语言，其中第一列是源代码行号，第二列是字节码偏移量，第三列是操作指令。关于反编译的内容，我们会在剖析函数的时候，深入介绍。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》45. pyc文件是怎么创建的?

下一篇 ›
《源码探秘 CPython》43. PyCodeObject对象与pyc文件