

《源码探秘 CPython》79. 模块是如何导入的

原创 古明地觉 古明地觉的编程教室 2022-04-29 08:30 发表于北京



微信扫一扫
关注该公众号

收录于合集
#CPython

97个 >



下面来聊一聊模块的导入机制，我们之前考察的所有内容都具有一个相同的特征，那就是它们都局限在一个 .py 文件中。然而现实中不可能只有一个 .py 文件，而是存在多个，而多个 .py 文件之间存在引用和交互，这些也是程序的一个重要组成部分。那么这里我们就来分析，Python 中模块的导入机制。

首先在这里我们必须强调一点，一个单独的 .py 文件、或者 .pyc 文件、.pyd 文件，我们称之为一个**模块**；而多个模块组合起来放在一个目录中，这个目录我们称之为**包**。

但不管是模块，还是包，在虚拟机的眼中，它们都是PyModuleObject结构体实例，类型为PyModule_Type，而在Python中则都是一个<class 'module'>对象。

```
1 //Objects/moduleobject.c
2 PyTypeObject PyModule_Type = {
3     PyVarObject_HEAD_INIT(&PyType_Type, 0)
4     "module", /* tp_name */
5     sizeof(PyModuleObject), /* tp_basicsize */
6     //...
7 };
8 //Python的<class 'module'>对应底层的PyModule_Type
9 //而导入进来的模块对象则对应底层的 PyModuleObject
```

所以模块和包导入进来之后也是一个对象，下面我们通过Python来演示一下。

```
1 import os
2 import pandas
3
4 print(os)
5 print(pandas)
6 """
7 <module 'os' from 'C:\\python38\\lib\\os.py'>
8 <module 'pandas' from 'C:\\python38\\lib\\site-packages\\pandas\\__init_
9 __.py'>
10 """
11
12 print(type(os)) # <class 'module'>
    print(type(pandas)) # <class 'module'>
```

因此不管是模块还是包，在Python中都是一样的，我们后面会详细说。总之它们都是一个PyModuleObject，只不过为了区分，我们把单独的文件叫做模块，把包含文件的目录叫做包。但是在Python的底层则并没有区分那么明显，它们都是一样的。

所以为了后续不产生歧义，我们这里做一个约定，从现在开始本系列中出现的**模块**，指的就是单独的可导入文件；出现的**包**，指的就是目录。而模块和包，我们都可以称之为module对象，因为这两者本来就是<class 'module'>的实例对象。





我们以一个简单的import为序幕，看看相应的字节码。

```
1 s = "import sys"
2
3 if __name__ == '__main__':
4     import dis
5     dis.dis(compile(s, "<file>", "exec"))
6 """
7 0 LOAD_CONST          0 (0)
8 2 LOAD_CONST          1 (None)
9 4 IMPORT_NAME         0 (sys)
10 6 STORE_NAME          0 (sys)
11 8 LOAD_CONST          1 (None)
12 10 RETURN_VALUE
13 """
```

字节码非常简单，import sys 这行代码对应指令 IMPORT_NAME，可以类比之前的 LOAD_NAME，表示将名为 sys 的 module 对象加载进来，然后用变量 sys 保存。

当我们调用sys.path的时候，虚拟机就能很轻松地通过sys来获取path这个属性所对应的值了。因此就像我们之前说的那样，创建函数、类、导入模块等等，它们本质上和通过赋值语句创建一个变量是没有什么区别的。

关键就是这个IMPORT_NAME，我们看看它的实现，还记得从哪里看吗？我们说所有指令集的实现都在 ceval.c 的那个无限 for 循环的巨型 switch 中。

```
1 case TARGET(IMPORT_NAME): {
2     //PyUnicodeObject对象
3     //比如import sys, 那么这个name就是字符串"sys"
4     PyObject *name = GETITEM(names, oparg);
5     //我们看到这里有一个 fromlist 和 level
6     //显然需要从运行时栈中获取对应的值，我们再看一下刚才的字节码
7     //我们发现在IMPORT_NAME之前有两个LOAD_CONST, 将0和None压入了运行时栈
8     //因此这里会从运行时栈中获取到None和0, 然后分别赋值给fromlist和level
9     //至于这两个是干啥的, 我们后面说
10    PyObject *fromlist = POP();
11    PyObject *level = TOP();
12    //一个PyModuleObject *, 指向模块对象
13    PyObject *res;
14    //调用import_name, 将该函数的返回值赋值给res
15    res = import_name(tstate, f, name, fromlist, level);
16    Py_DECREF(level);
17    Py_DECREF(fromlist);
18    //设置为栈顶元素, 后续通过 STORE_NAME 将其弹出
19    //然后交给变量 sys 保存
20    SET_TOP(res);
21    if (res == NULL)
22        goto error;
23    DISPATCH();
24 }
```

因此重点在import_name这个函数中，但是在此之前我们需要先关注一下fromlist和level，而这一点可以从Python的层面来介绍。我们知道在Python里面导入一个模块直接通过import关键字即可，但是除了import，我们还可以使用内置函数__import__来进行导入。这个__import__是解释器使用的一个函数，不推荐我们直接使用，但我想说的是i m p o r t o s在虚拟机看来就是os = __import__("os")。

```
1 os = __import__("os")
2 SYS = __import__("sys")
3
```

```
4 print(os) # <module 'os' from 'C:\\python38\\Lib\\os.py'>
5 print(SYS.prefix) # C:\\python38
```

但是问题来了：

```
1 m1 = __import__("os.path")
2 print(m1) # <module 'os' from 'C:\\python38\\Lib\\os.py'>
3 # 我们惊奇地发现, 居然还是os模块
4 # 按理说应该是os.path(windows系统对应ntpath)才对啊
5 m2 = __import__("os.path", fromlist=[""])
6 print(m2) # <module 'ntpath' from 'C:\\python38\\Lib\\ntpath.py'>
7 # 你看到了什么, fromlist
8 # 没错, 我们加上一个fromlist, 就能导入子模块
```

为什么会这样呢？我们来看看__import__这个函数的解释，这个是PyCharm给抽象出来的。

```
* def __import__(name, globals=None, locals=None, fromlist=(), level=0): # real signature
    """
    __import__(name, globals=None, locals=None, fromlist=(), level=0) -> module

    Import a module. Because this function is meant for use by the Python
    interpreter and not for general use, it is better to use
    importlib.import_module() to programmatically import a module.

    The globals argument is only used to determine the context;
    they are not modified. The locals argument is unused. The fromlist
    should be a list of names to emulate ``from name import ...'', or an
    empty list to emulate ``import name''.
    When importing a module from a package, note that __import__('A.B', ...)
    returns package A when fromlist is empty, but its submodule B when
    fromlist is not empty. The level argument is used to determine whether to
    perform absolute or relative imports: 0 is absolute, while a positive number
    is the number of parent directories to search relative to the current module.
    """
    pass
```

大意就是，此函数会由import语句调用，当我们import的时候，解释器底层就会调用__import__。比如import os表示将"os"这个字符串传入__import__函数中，从指定目录加载os.py，当然也可能是os.pyd、或者一个名为os的目录，得到一个module对象，并将返回值赋值给变量os，也就是os = __import__("os")。

虽然我们可以通过这种方式来导入模块，但是Python不建议我们这么做。而globals参数则是确定import语句包的上下文，一般直接传globals()即可，但是locals参数我们基本不用，不过一般情况下globals和locals我们都不用管。

总之 __import__("os.path") 导入的不是 os.path，而还是 os 这个外层模块。如果想导入os.path，那么只需要给 fromlist 传入一个非空列表即可。其实不仅仅是非空列表，只要是一个非空的可迭代对象就行。

而 level 如果是 0，那么表示仅执行绝对导入；如果是一个正整数，表示要搜索的父目录的数量，也就是相对导入。

因此当包名是一个字符串的时候，我们就没办法使用 import 关键字了，这时就可以手动使用__import__。但是官方不推荐这么做，因为这是给解释器用的，官方推荐我们用 importlib。

```
1 import importlib
2
3 a = "pandas"
4 pd = importlib.import_module(a)
5 # 很方便的就导入了
6 # 直接通过字符串的方式导入一个 module对象
7 print(pd)
8 """
9 <module 'pandas' from 'C:\\python38\\lib\\site-packages\\pandas\\__init_
10 __.py'>
11 """
12
```

```

13 如果想导入 "模块中导入的模块"
14 # 比如: 模块a中导入了模块b, 我们希望导入a.b
15 # 或者导入一个包下面的子模块等等, 比如: pandas.core.frame
16 sub_mod = importlib.import_module("pandas.core.frame")
17 # 我们看到可以自动导入pandas.core.frame
18 print(sub_mod)
19 """
20 <module 'pandas.core.frame' from 'C:\\python38\\lib\\site-packages\\pand
21 as\\core\\frame.py'>
22 """
23
24 # 但如果是__import__, 默认的话是不行的, 导入的依旧是最外层pandas
25 print(__import__("pandas.core.frame"))
26 """
27 <module 'pandas' from 'C:\\python38\\lib\\site-packages\\pandas\\__init_
28 __.py'>
29 """
30
31 # 可以通过给fromList指定一个非空列表来实现
32 print(__import__("pandas.core.frame", fromlist=[""]))
    """
    <module 'pandas.core.frame' from 'C:\\python38\\lib\\site-packages\\pand
    as\\core\\frame.py'>
    """

```

上面的导入方式虽然很方便, 但有一个要求, 就是导入的模块必须位于搜索路径之下。举个栗子, 假设我们的项目在 D 盘, 但是有一个 test.py 模块位于 F 盘, 这时候该怎么做呢?

```

1  # 有一个文件 F:\mashiro\test.py
2  # 我们如何才能将它导入进来呢?
3
4  from importlib.machinery import SourceFileLoader
5
6  # 第一个参数是模块名, 第二个参数是模块的路径
7  # 这样就可以实现导入了
8  test = SourceFileLoader("test", r"F:\mashiro\test.py").load_module()
9  # 所以这是基于文件路径进行加载的
10 # 因此这个做法能够保证无论文件在什么地方, 都可以进行导入
11
12 # 上面这个类只能加载 py 文件
13 # 如果想加载 pyc/pyd 文件, 需要用下面两个类
14 from importlib.machinery import SourcelessFileLoader # pyc
15 from importlib.machinery import ExtensionFileLoader  # pyd

```

或者我们还可以通过 exec 的方式创建。

```

1  from types import ModuleType
2
3  # 此时的 os 里面啥也没有
4  os = ModuleType("我是 os 模块")
5
6  with open(r"C:\python38\Lib\os.py", encoding="utf-8") as f:
7      source = f.read()
8
9  # 通过 exec 执行读取出来的字符串
10 # 然后将名字空间换成 os 的属性字典
11 exec(source, os.__dict__)
12
13 print(os.__name__) # 我是 os 模块
14 print(os.path.join("x", "y", "z")) # x\y\z
15
16
17 print(hasattr(os, "嘿")) # False
18 exec("嘿 = '蛤'", os.__dict__)

```

```
19 print(os.嘿) # 蛤
```

当然啦，也可以把一个自定义的类变成模块，举个栗子：

```
1 from types import ModuleType
2
3 class A(ModuleType):
4
5     def __init__(self, name):
6         super(A, self).__init__(name)
7
8     def __getattr__(self, item):
9         return f"获取属性: {item}"
10
11    def __setattr__(self, key, value):
12        self.__dict__[key] = value
13
14    def __str__(self):
15        return f"<module '{self.__name__}' from '我来自于虚无'>"
16
17 a = A("我是 A")
18 print(a) # <module '我是 A' from '我来自于虚无'>
19 print(a.__name__) # 我是 A
20 print(a.xxx) # 获取属性: xxx
21
22 a.xxx = "yyy"
23 print(a.xxx) # yyy
24
25 # 如果我们将其加入到 sys.modules 里面的话
26 import sys
27 sys.modules["A1"] = A
28
29 import A1
30 print(A1 is A) # True
```

是不是很好玩呢？关于里面的一些细节，比如 sys.modules 是什么，后续会详细说。好了，扯了这么多，我们回到 IMPORT_NAME 这个指令，它是加载模块时对应的指令。在里面确定完参数之后，会调用 import_name，我们看看这个函数长什么样子。

```
1 //ceval.c
2 case TARGET(IMPORT_NAME): {
3     // 这个函数接收了五个参数
4     // tstate:线程状态对象, f:栈帧, name:模块名
5     // fromlist:一个 None, level:0
6     res = import_name(tstate, f, name, fromlist, level);
7 }
8
9 static PyObject *
10 import_name(PyThreadState *tstate, PyFrameObject *f,
11             PyObject *name, PyObject *fromlist, PyObject *level)
12 {
13     _Py_IDENTIFIER(__import__);
14     PyObject *import_func, *res;
15     PyObject* stack[5];
16
17     //获取内置函数 __import__
18     import_func = _PyDict_GetItemIdWithError(f->f_builtins, &PyId___impo
19 rt__);
20
21     //为NULL表示获取失败, 显然这些都是Python底层做的检测
22     //我们使用时不会出现, 如果出现, 只能说明解释器出问题了
23     if (import_func == NULL) {
24         if (!PyErr_Occurred(tstate)) {
25             _PyErr_SetString(tstate, PyExc_ImportError, "__import__ not
```

```

26     found");
27     }
28     return NULL;
29 }
30
31 //判断__import__是否被重载了
32 if (import_func == tstate->interp->import_func) {
33     int ilevel = _PyLong_AsInt(level);
34     if (ilevel == -1 && _PyErr_Occurred(tstate)) {
35         return NULL;
36     }
37     //未重载的话, 调用PyImport_ImportModuleLevelObject
38     res = PyImport_ImportModuleLevelObject(
39         name,
40         f->f_globals,
41         f->f_locals == NULL ? Py_None : f->f_locals,
42         fromlist,
43         ilevel);
44     return res;
45 }
46
47 //否则调用重载后的 __import__
48 Py_INCREF(import_func);
49
50 stack[0] = name;
51 stack[1] = f->f_globals;
52 stack[2] = f->f_locals == NULL ? Py_None : f->f_locals;
53 stack[3] = fromlist;
54 stack[4] = level;
55 res = _PyObject_FastCall(import_func, stack, 5);
56 Py_DECREF(import_func);
57 return res;
58 }

```

然后我们看到底层又调用了 `PyImport_ImportModuleLevelObject` , 显然核心隐藏在这里面, 我们来看一下它的实现。

```

1 //Python/import.c
2 PyObject *
3 PyImport_ImportModuleLevelObject(PyObject *name, PyObject *globals,
4     PyObject *locals, PyObject *fromlist,
5     int level)
6 {
7     _Py_IDENTIFIER(_handle_fromlist);
8     PyObject *abs_name = NULL;
9     PyObject *final_mod = NULL;
10    PyObject *mod = NULL;
11    PyObject *package = NULL;
12    PyInterpreterState *interp = _PyInterpreterState_GET_UNSAFE();
13    int has_from;
14
15    //名字不可以为空
16    if (name == NULL) {
17        PyErr_SetString(PyExc_ValueError, "Empty module name");
18        goto error;
19    }
20
21    //名字必须是PyUnicodeObject
22    if (!PyUnicode_Check(name)) {
23        PyErr_SetString(PyExc_TypeError, "module name must be a string");
24        goto error;
25    }
26
27    //level不可以小于0

```

```

28     if (level < 0) {
29         PyErr_SetString(PyExc_ValueError, "level must be >= 0");
30         goto error;
31     }
32
33     //level大于0
34     if (level > 0) {
35         //在相应的父目录寻找, 得到 abs_name
36         abs_name = resolve_name(name, globals, level);
37         if (abs_name == NULL)
38             goto error;
39     }
40     else {
41         //否则的话, 说明level==0
42         //因为level要求是一个大于等于0的整数
43         if (PyUnicode_GET_LENGTH(name) == 0) {
44             PyErr_SetString(PyExc_ValueError, "Empty module name");
45             goto error;
46         }
47         //此时直接将name赋值给abs_name
48         //因为此时是绝对导入
49         abs_name = name;
50         Py_INCREF(abs_name);
51     }
52
53     //调用 PyImport_GetModule 获取 module对象
54     mod = PyImport_GetModule(abs_name);
55     //...
56     if (mod == NULL && PyErr_Occurred()) {
57         goto error;
58     }
59     //...
60     //...
61     else {
62         //调用函数, 导入模块
63         final_mod = _PyObject_CallMethodIdObjArgs(interp->importlib,
64                                                     &PyId__handle_fromlist, m
65 od,
66                                                     fromlist, interp->import_
67 func,
68                                                     NULL);
69     }
70
71     error:
72     Py_XDECREF(abs_name);
73     Py_XDECREF(mod);
74     Py_XDECREF(package);
75     if (final_mod == NULL)
76         remove_importlib_frames();
77     return final_mod;
78 }

```

还是很好理解的, 关于 module 对象的导入, Python 也提供了非常丰富的写法。

```

1 import numpy
2 import numpy as np
3 import numpy.random as _random
4
5 from numpy import random
6 from numpy import random as _random
7 from numpy import *

```

从import的目标来说, 可以是包, 也可以是模块。而模块可以通过 .py 文件作为载体, 也可以通过 .pyc 或者 .pyd 等二进制文件作为载体。

下一篇文章我们就来详细分析。

收录于合集 [#CPython](#) 97

[< 上一篇](#)

《源码探秘 CPython》80. import机制的黑盒探测

[下一篇 >](#)

《源码探秘 CPython》78. 魔法方法都有哪些？作用是什么？

喜欢此内容的人还喜欢

[python] Python数据序列化模块pickle使用笔记
彭彭加油鸭



从零开始学 Python 之 math 模块
豆豆的杂货铺



Typescript - 模块与命名空间（2）
老李物语

