

《源码探秘 CPython》81. import 机制是怎么实现的？

原创 古明地觉 古明地觉的编程教室 2022-05-06 08:30 发表于北京

收录于合集

#CPython

97个 >



微信扫一扫
关注该公众号



通过上一篇的 import 黑盒探测，我们已经对 import 机制有了一个非常清晰的认识，Python 的 import 机制基本上可以切分为三个不同的功能。

- Python运行时的全局模块池的维护和搜索；
- 解析与搜索模块路径的树状结构；
- 对不同文件格式的模块执行动态加载机制；

尽管 import 的表现形式千变万化，但是都可以归结为：import x.y.z 的形式，当然 import sys 也可以看成是 x.y.z 的一种特殊形式。而诸如 from、as 与 import 的结合，实际上同样会进行 import x.y.z 的动作，只是最后在当前名字空间中引入的符号各有不同。

然后导入模块，虚拟机会调用 __import__，那么我们就来看看这个函数长什么样子。

```
1 static PyObject *
2 builtin___import__(PyObject *self, PyObject *args, PyObject *kwargs)
3 {
4     static char *kwlist[] = {"name", "globals", "locals", "fromlist",
5                               "level", 0};
6     //初始化globals、fromlist都为NULL
7     PyObject *name, *globals = NULL, *locals = NULL, *fromlist = NULL;
8     int level = 0; //表示默认绝对导入
9
10    //从PyTupleObject中解析出需要的信息
11    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "U|OOO:___import___",
12                                     kwlist, &name, &globals, &locals, &fromlist, &level))
13        return NULL;
14    //导入模块
15    return PyImport_ImportModuleLevelObject(name, globals, locals,
16                                             fromlist, level);
17 }
18
```

里面有一个PyArg_ParseTupleAndKeywords函数，我们需要提一下，它在虚拟机中是一个被广泛使用的函数，原型如下：

```
1 //Python/getargs.c
2 int PyArg_ParseTupleAndKeywords(PyObject *, PyObject *,
3                                 const char *, char **, ...);
```

这个函数的作用是参数解析，负责将 args 和 kwargs 中所包含的所有对象（指针）按指定的格式 format 解析成各种目标对象，可以是 Python 的对象，例如 PyListObject、PyLongObject，也可以是 C 的原生对象。

我们知道这个 builtin___import__ 里面的参数 args 指向一个 PyTupleObject，包含了 __import__ 函数运行所需要的参数和信息，它是虚拟机在执行 IMPORT_NAME 指令的时候打包产生的。

然而在这里，虚拟机进行了一个逆动作，将打包后的这个 PyTupleObject 拆开，重新获得当初的参数。Python 在自身的实现中大量使用了这样的打包、拆包策略，使得可变数量的对象能够很容易地在函数之间传递。

该系列完结后，会介绍如何用 C 给 Python 写扩展，到时候会剖析这个函数的用法。

在完成了对参数的拆包动作之后，会进入 `PyImport_ImportModuleLevelObject`，这个我们在 `import_name` 中已经看到了，当然它内部也是调用了 `__import__`。

另外每个包和模块都有一个 `__name__` 和 `__path__` 属性。

```
1 import numpy as np
2 import numpy.core
3 import six
4
5 print(np.__name__, np.__path__)
6 """
7 numpy ['C:\\python38\\lib\\site-packages\\numpy']
8 """
9
10 print(np.core.__name__, np.core.__path__)
11 """
12 numpy.core ['C:\\python38\\lib\\site-packages\\numpy\\core']
13 """
14
15 print(six.__name__, six.__path__)
16 """
17 six []
18 """
```

`__name__` 就是模块名或者包名，如果是包下面的包或者模块，那么就是 **包名.包名** 或者 **包名.模块名**；至于 `__path__` 则是包所在的路径，对于模块而言，`__path__` 为空列表。

此外还有一个 `__file__` 属性，对于模块而言就是其自身的完整路径；对于包而言则分两种情况，如果包内部存在 `__init__.py` 文件，那么得到的就是 `__init__.py` 文件的完整路径，没有则为 `None`。

下面来看一下不同的导入方式对应的字节码，然后在虚拟机的层面来理解这些导入方式。



以一个简单的模块导入为例：

```
1 import sys
2 """
3   0 LOAD_CONST           0 (0)
4   2 LOAD_CONST           1 (None)
5   4 IMPORT_NAME           0 (sys)
6   6 STORE_NAME            0 (sys)
7   8 LOAD_CONST           1 (None)
8  10 RETURN_VALUE
9 """
```

这是我们一开始考察的例子，现在我们已经很清楚地了解了 `IMPORT_NAME` 的行为。在 `IMPORT_NAME` 指令的最后，虚拟机会将 `PyModuleObject` 对象（指针）压入到运行时栈，随后会将 `<"sys", PyModuleObject *>` 存放到当前的 `local` 名字空间中。





```
1 import sklearn.linear_model.ridge
2 """
3   0 LOAD_CONST           0 (0)
4   2 LOAD_CONST           1 (None)
5   4 IMPORT_NAME           0 (sklearn.linear_model.ridge)
6   6 STORE_NAME           1 (sklearn)
7   8 LOAD_CONST           1 (None)
8  10 RETURN_VALUE
9 """
```

如果是级联导入，那么 `IMPORT_NAME` 的指令参数则是完整的路径信息，该指令的内部将解析这个路径，并为 `sklearn`, `sklearn.linear_model`, `sklearn.linear_model.ridge` 都创建一个 `PyModuleObject` 对象，这三者都存在于 `sys.modules` 里面。

但是我们看到 `STORE_NAME` 是 `sklearn`，表示只有 `sklearn` 这个符号暴露在了当前模块的 `local` 空间里面。可为什么是 `sklearn` 呢？难道不应该是 `sklearn.linear_model.ridge` 吗？

其实经过我们之前的分析这一点已经不再是问题了，因为 `import sklearn.linear_model.ridge` 并不是说导入一个模块或包叫做 `sklearn.linear_model.ridge`，而是先导入 `sklearn`，然后把 `linear_model` 放在 `sklearn` 的属性字典里面，再把 `ridge` 放在 `linear_model` 的属性字典里面。

同理 `sklearn.linear_model.ridge` 代表的是先从 `local` 空间里面找到 `sklearn`，再从 `sklearn` 的属性字典中找到 `linear_model`，然后在 `linear_model` 的属性字典里面找到 `ridge`。因为 `linear_model` 和 `ridge` 已经在相应的属性字典里面，我们通过 `sklearn` 一级一级往下找是可以找到的，因此只需要将符号 `sklearn` 暴露给 `local` 空间即可。

或者说暴露 `sklearn.linear_model.ridge` 本身就是不合理的，因为这表示导入一个名字就叫做 `sklearn.linear_model.ridge` 的模块或者包，但显然不存在。而即便我们创建了这样的一个模块或包，由于 Python 的语法解析规范依旧不会得到想要的结果。不然的话，假设 `import test_import.a`，那是导入名为 `test_import.a` 的模块或包呢？还是导入 `test_import` 下的 `a` 呢？

也正如我们之前分析的 `test_import.a`，我们在导入 `test_import.a` 的时候，会把 `test_import` 加载进来，然后把 `a` 加到 `test_import` 的属性字典里面，最后只需要把 `test_import` 返回即可。

因为通过 `test_import` 可以找到 `a`，或者说 `test_import.a` 代表的含义就是从 `test_import` 的属性字典里面获取 `a`，所以 `import test_import.a` 必须要返回 `test_import`，而且只需返回 `test_import`。

至于 `sys.modules` 里面虽然存在字符串名为 `"test_import.a"` 的 `key` 的，但这是为了避免重复加载所采取的策略，它依旧表示从 `test_import` 的属性字典里面获取 `a`。

```
1 import pandas.core
2
3 print(pandas.DataFrame({"a": [1, 2, 3]}))
4 """
5   a
6 0  1
7 1  2
8 2  3
9 """
10 # 所以通过 pandas.DataFrame 是可以调用的
```

导入 `pandas.core` 会先导入 `pandas`，也就是执行 `pandas` 内部的 `__init__` 文件。虽然 `sys.modules` 里面同时有 `"pandas"` 和 `"pandas.core"`，但是暴露在 `local` 空间的只有 `pandas`，所以调用 `pandas.DataFrame` 是完全合理的。至于 `pandas.core` 显然它无法暴露，因为这不符合 Python 的变量命名规范，变量的名称里面不能出现小数点，它只是单纯地表示从 `pandas` 的属性字典中加载 `core`。



```
1 from sklearn.linear_model import ridge
2 """
3 0 LOAD_CONST          0 (0)
4 2 LOAD_CONST          1 (('ridge',))
5 4 IMPORT_NAME         0 (sklearn.linear_model)
6 6 IMPORT_FROM         1 (ridge)
7 8 STORE_NAME          1 (ridge)
8 10 POP_TOP
9 12 LOAD_CONST          2 (None)
10 14 RETURN_VALUE
11 """
```

注意此时的 **2 LOAD_CONST** 不再是 None 了，而是一个元组，虚拟机将 ridge 放到了当前模块的 local 空间中。并且 sklearn.linear_model 和 sklearn 都被导入了，存在 sys.modules 里面。

但是 sklearn 却并不在当前 local 空间中，尽管它被创建了，但是又被隐藏了。IMPORT_NAME 是 sklearn.linear_model，也表示导入 sklearn，然后把 sklearn 下面的 linear_model 加入到 sklearn 的属性字典里面。

而之所以 sklearn 没在 local 空间里面，可以这样理解。当只出现 import 的时候，那么我们必须从头开始一级一级向下调用，所以顶层的包必须加入到 local 空间里面。但这里通过 **from ... import ...** 把 ridge 导出了，此时 ridge 已经指向了 sklearn 下面的 linear_model 下面的 ridge，那么就不需要 sklearn 了，或者说 sklearn 就没必要暴露在 local 空间里面了，但它确实被导入进来了。

并且 sys.modules 里面也不存在 "ridge" 这个 key，存在的是 "sklearn.linear_model.ridge"，暴露给 local 空间的符号是 ridge。

所以正如上面所说，不管什么导入，都可以归结为 **import x.y.z** 的形式，只是暴露出来的符号不同罢了。



```
1 import sklearn.linear_model.ridge as xxx
2 """
3 0 LOAD_CONST          0 (0)
4 2 LOAD_CONST          1 (None)
5 4 IMPORT_NAME         0 (sklearn.linear_model.ridge)
6 6 IMPORT_FROM         1 (linear_model)
7 8 ROT_TWO
8 10 POP_TOP
9 12 IMPORT_FROM         2 (ridge)
10 14 STORE_NAME         3 (xxx)
11 16 POP_TOP
12 18 LOAD_CONST          1 (None)
13 20 RETURN_VALUE
```

这个和上面的 `from & import` 类似, `"sklearn"`, `"sklearn.linear_model"`, `"sklearn.linear_model.ridge"` 都在 `sys.modules` 里面。但是我们加上了 `as xxx`, 那么这个 `xxx` 就直接指向了 `sklearn` 下面的 `linear_model` 下面的 `ridge`, 此时就不需要 `sklearn` 了。

因此只有 `xxx` 暴露在了当前模块的 `local` 空间里面, 而 `sklearn` 虽然也被导入了, 但它只在 `sys.modules` 里面, 没有暴露给当前模块的 `local` 空间。



```
1 from sklearn.linear_model import ridge as xxx
```

这个我想连字节码都不需要贴了, 和之前的 `from & import` 一样, 只是最后暴露给 `local` 空间的 `ridge` 变成了我们自己指定的 `xxx`。



与 module 对象有关的名字空间问题

同函数、类一样, 每个 `PyModuleObject` 也有自己的名字空间。一个模块不能直接访问另一个模块的内容, 尽管模块内部的作用域比较复杂, 比如: 遵循 `LEGB` 规则, 但是模块与模块之间的划分则是很明显的。

```
1 # test1.py
2 name = "古明地觉"
3
4 def print_name():
5     return name
```

```
1 # test2.py
2 from test1 import name, print_name
3 name = "古明地恋"
4 print(print_name()) # 古明地觉
```

执行 `test2.py` 之后, 发现打印的依旧是“古明地觉”。我们说 `Python` 是根据 `LEGB` 规则进行查找, 而 `print_name` 函数里面没有 `name`, 那么去外层找。`test2.py` 里面的 `name` 是“古明地恋”, 但是打印的依旧是 `test1.py` 里面的“古明地觉”。为什么?

还是那句话, 模块与模块之间的作用域划分的非常明显, `print_name` 是 `test1.py` 里面的函数, 所以在返回 `name` 的时候, 只会从 `test1.py` 中搜索, 无论如何都是不会跳过 `test1.py`、跑到 `test2.py` 里面的。

再来看个例子:

```
1 # test1.py
2 name = "古明地觉"
3 nicknames = ["小五", "少女觉"]
```

```
1 # test2.py
```

```

2 import test1
3 test1.name = "♥古明地觉♥"
4 test1.nicknames = ["觉大人"]
5
6 from test1 import name, nicknames
7 print(name) # ♥古明地觉♥
8 print(nicknames) # ['觉大人']

```

此时打印的结果变了，很简单，这里是直接把 test1 里面的变量修改了。因为这种方式，相当于直接修改 test1 的属性字典。那么后续再导入的时候，打印的就是修改之后的值。

```

1 # test1.py
2 name = "古明地觉"
3 nicknames = ["小五", "少女觉"]

```

```

1 # test2.py
2 from test1 import name, nicknames
3 name = "古明地恋"
4 nicknames.remove("小五")
5
6 from test1 import name, nicknames
7 print(name) # 古明地觉
8 print(nicknames) # ["少女觉"]

```

如果是 **from test1 import name, nicknames**，那么相当于在当前的 local 空间中新创建变量 name 和 nicknames，它们和 test1 中的 name 和 nicknames 指向相同的对象。

n a m e = " 古 明 地 觉 " 相当于重新赋值了，所以不会影响 test1 里的 name；而 nicknames.remove 则是在本地进行修改，所以会产生影响。



以上就是模块（包）相关的内容，虽然一个项目可以有很多个文件，但是每个文件的执行原理是一致的。无论一个文件是作为模块被导入，还是直接作为启动文件被执行，虚拟机的执行流程都没有变化。

通过模块和包，我们便可以对项目进行功能上的划分，从而更好地组织项目。

收录于合集 [#CPython 97](#)

[← 上一篇](#)

《源码探秘 CPython》82. Python运行时环境的初始化，解释器在启动时都做了什么？

[下一篇 →](#)

《源码探秘 CPython》80. import机制的黑盒探测

喜欢此内容的人还喜欢

python 7天进阶之路-对象和json转换
缪斯之子



[系列]微服务·深入理解 gRPC - Part2
走向架构师的每一天



Abaqus python脚本开发 第三章 各类指令的方法对象变量 (3)
山石结构



