22 模块动态加载， import 背后哪些事儿-慕课专栏

我们知道 *import* 关键字用于导入模块，例如：

```
import demo
```

此外，*import* 语句还有其他几种变体，例如：

```
import demo as d

from demo import value
from demo import value as v
```

那么，*Python* 模块加载的过程是怎样的呢？不同 *import* 语句都有哪些异同，背后具体执行了什么动作？*Python* 又是如何找到被导入模块的呢？带着这些问题，我们开始学习 *Python* 的 **模块加载** 机制。

字节码

透过字节码，我们可以洞悉 *Python* 执行语句的全部秘密。因此，我们从研究 *import* 语句字节码入手，逐步深入研究模块的加载过程。

import

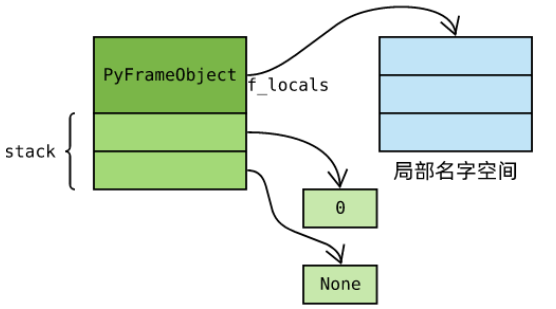
以最基本的 *import* 语句为例：

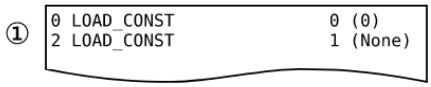
```
import demo
```

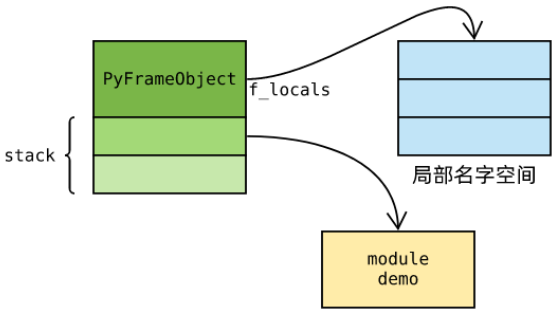
借助 *dis* 模块，我们将这个语句反编译，得到以下字节码：

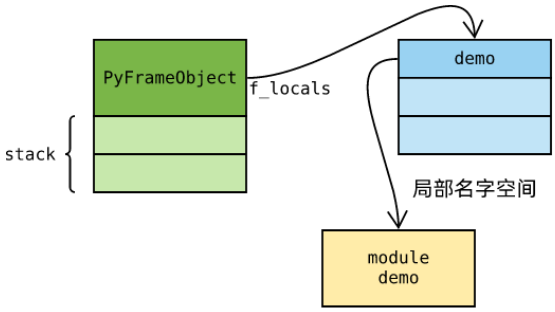
```
1           0 LOAD_CONST               0 (0)
            2 LOAD_CONST               1 (None)
            4 IMPORT_NAME              0 (demo)
            6 STORE_NAME               0 (demo)
            8 LOAD_CONST               1 (None)
           10 RETURN_VALUE
```

我们重点关注前 *4* 条字节码，看它们在 *Python* 虚拟机中是如何执行的：









1. 前 *2* 条字节码执行完毕后，*0* 以及 *None* 这两个常量被加载到栈中；
2. 顾名思义，*IMPORT_NAME* 指令负责加载模块，模块名由操作数指定，其他参数从栈上取；模块加载完毕后，模块对象便保存在栈顶；
3. 最后，*STORE_NAME* 指令从栈顶取出模块对象并保存到局部名字空间中；

至此，*Python* 模块动态加载的秘密已经浮出水面了。在字节码层面，*IMPORT_NAME* 负责加载模块，**模块名** 由操作数指定，其他参数来源于 **运行栈** 。虽然 *IMPORT_NAME* 指令还没来得及研究，成就感也是满满的呢！

import as

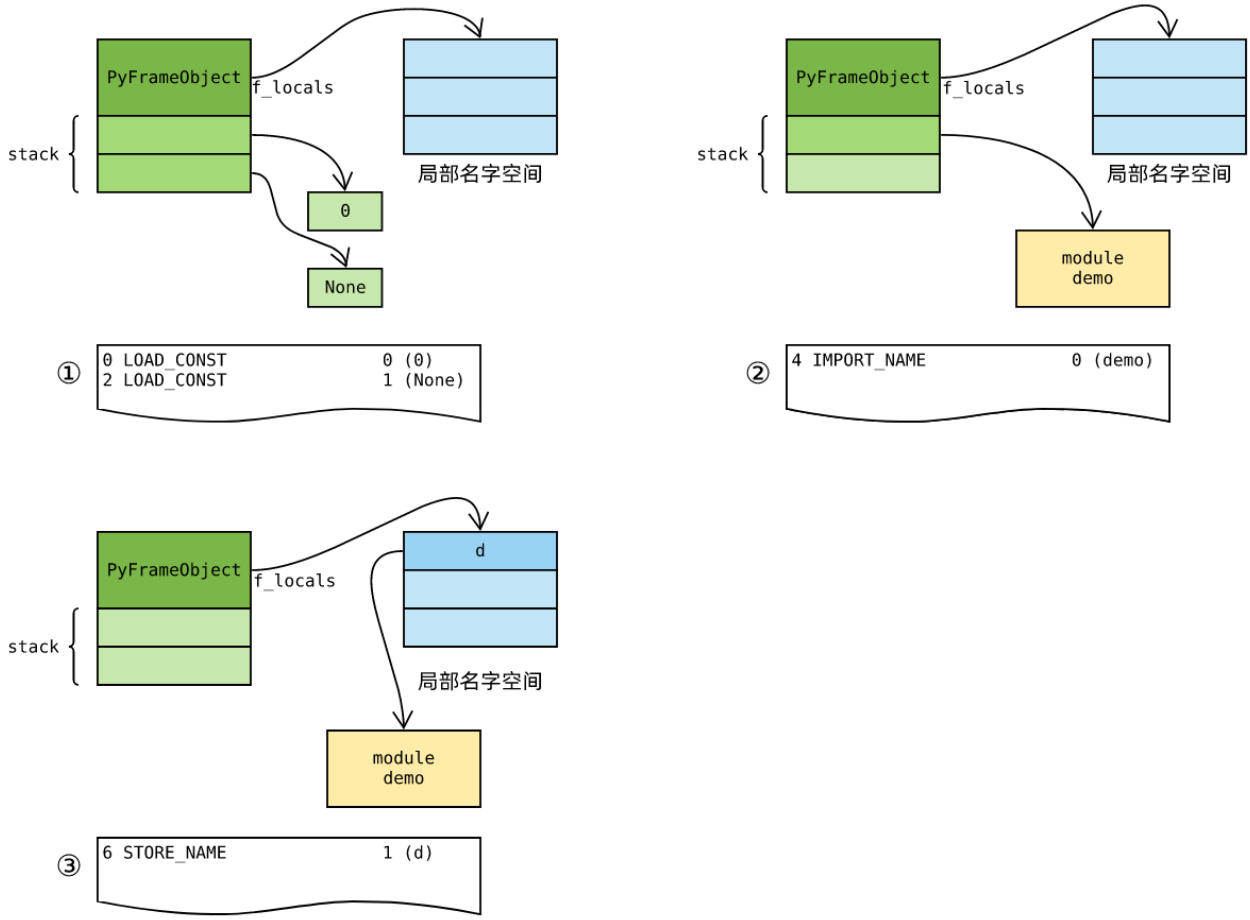
开始研究 *IMPORT_NAME* 指令实现细节前，一鼓作气将其他几种 import 语句变体拿下。先研究 *impot as* 语句：

```
import demo as d
```

同样用 dis 对语句进行反编译，我们得到以下字节码：

```
  1           0 LOAD_CONST               0 (0)
              2 LOAD_CONST               1 (None)
              4 IMPORT_NAME              0 (demo)
              6 STORE_NAME               1 (d)
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

这段字节码跟前一段几乎一模一样，区别只是 *STORE_NAME* 指令，它用换个名字来保存被加载模块：



因此，这个 *import* 语句变体其实等价于：

```
import demo
d = demo
del demo
```

from import

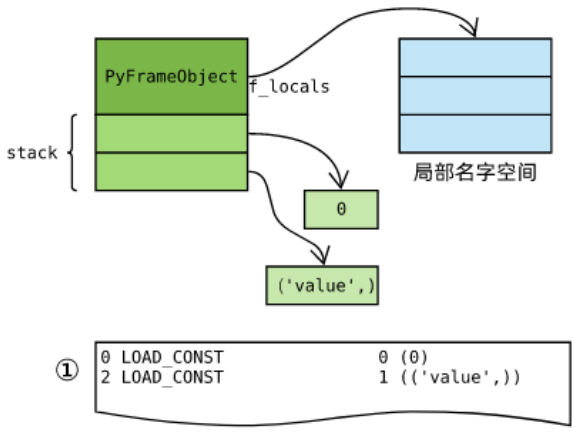
现在再接再厉，拿下 *from import* 语句：

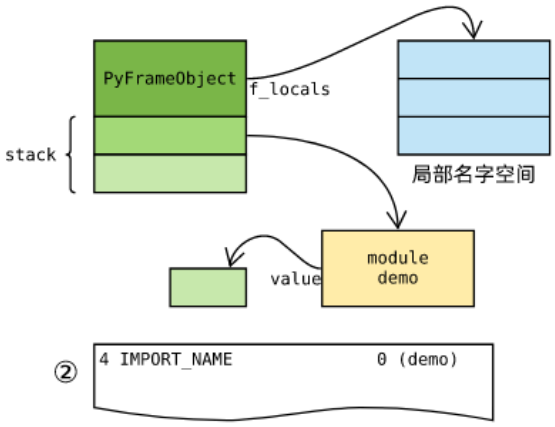
```
from demo import value
```

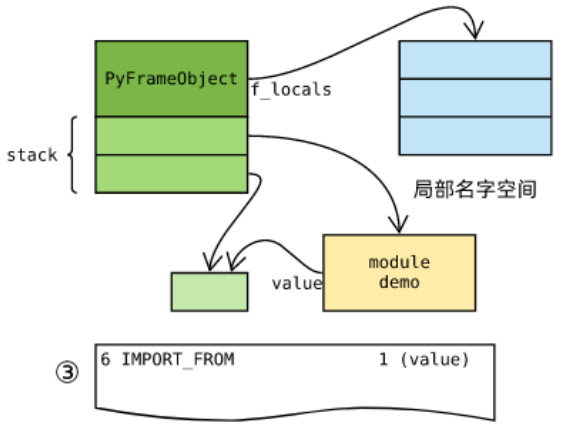
同样用 *dis* 对语句进行反编译，我们得到以下字节码：

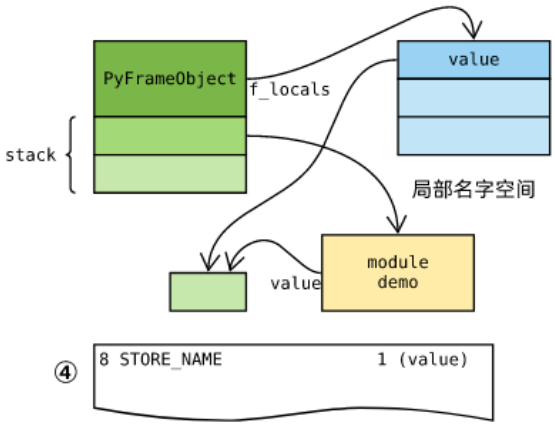
```
1           0 LOAD_CONST               0 (0)
            2 LOAD_CONST               1 (('value',))
            4 IMPORT_NAME              0 (demo)
            6 IMPORT_FROM              1 (value)
            8 STORE_NAME               1 (value)
           10 POP_TOP
           12 LOAD_CONST               2 (None)
           14 RETURN_VALUE
```

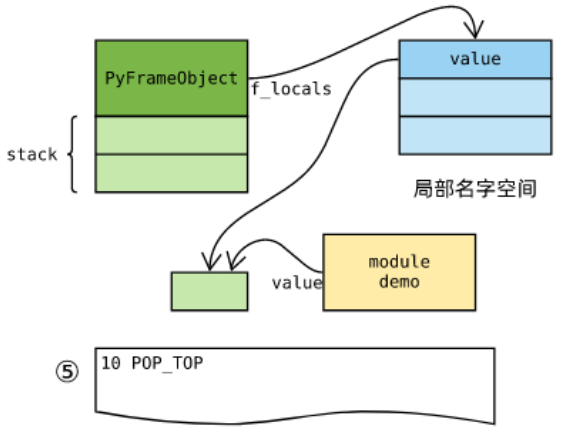
我们看到一个新面孔—— *IMPORT_FROM* 指令。该指令从栈顶模块中取出指定名字，并保存于栈顶。











注意到，*value* 以 **元组** 的形式保存于栈顶，*IMPORT_NAME* 指令如果发现 *value* 为 *demo* 模块的子模块，将同时加载 *value* 子模块。此外，*IMPORT_FROM* 与 *STORE_NAME* 这两个指令相互配合，从模块中取出给定名字并保存。如果 *from import* 语句一次性导入多个名字，字节码将包含多个由 *IMPORT_FROM* 和 *STORE_NAME* 组成的指令对：

```
from demo import value, data
```

```
  1           0 LOAD_CONST               0 (0)
              2 LOAD_CONST               1 (('value', 'data'))
              4 IMPORT_NAME              0 (demo)
              6 IMPORT_FROM              1 (value)
              8 STORE_NAME               1 (value)
             10 IMPORT_FROM              2 (data)
             12 STORE_NAME               2 (data)
             14 POP_TOP
             16 LOAD_CONST               2 (None)
             18 RETURN_VALUE
```

这也是 *IMPORT_FROM* 指令不将模块对象从栈顶弹出的原因——只有所有需要导入的名字均导入完毕，模块对象才可从栈顶弹出，这个使命正是由紧接着的 *POP_TOP* 指令完成！

综上所述，*from import* 语句实际上等价于：

```
import demo
value = demo.value
del demo
```

from import as

from import as 语句与 *from import* 语句的关系，跟 *import as* 语句与 *import* 语句的关系一样。这两种语句的差别只在 *STORE_NAME* 字节码的操作数上，而操作数直接决定了被导入对象以什么名字在局部名字空间中保存。*from import as* 语句示例以及对应的字节码分别如下，相信轻易即可看懂，就不再赘述了：

```
from demo import value as v
```

```
  1           0 LOAD_CONST               0 (0)
              2 LOAD_CONST               1 (('value',))
              4 IMPORT_NAME              0 (demo)
              6 IMPORT_FROM              1 (value)
              8 STORE_NAME               2 (v)
             10 POP_TOP
             12 LOAD_CONST               2 (None)
             14 RETURN_VALUE
```

模块加载流程

现在回过头来看 *Python* 虚拟机是如何执行 *IMPORT_NAME* 指令的，从中便可洞悉 *Python* **模块动态加载机制** 。

IMPORT_NAME 字节码指令在 *Include/opcode.h* 头文件中定义，其后紧挨着 *IMPORT_FROM* 指令：

```
#define IMPORT_NAME             108
#define IMPORT_FROM             109
```

Python 虚拟机实现位于 *Python/ceval.c* 源文件中，*IMPORT_NAME* 指令的处理逻辑也在其中(第 *2595* 行)：

```
TARGET(IMPORT_NAME) {
    PyObject *name = GETITEM(names, oparg);
    PyObject *fromlist = POP();
    PyObject *level = TOP();
    PyObject *res;
    res = import_name(f, name, fromlist, level);
    Py_DECREF(level);
    Py_DECREF(fromlist);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

1. 第 *2* 行，根据字节码 **操作数** 取出待加载 **模块名** ；
2. 第 *3* 行，从栈顶弹出 *fromlist* 参数，参数是一个元组，列举了需要加载的潜在 **子模块** ；
3. 第 *4* 行，从栈顶弹出 *level* 参数；
4. 第 *6* 行，调用 *import_name* 函数完成模块加载工作；
5. 第 *7-9* 行，释放参数并将加载到的 **模块对象** 保存到栈顶；

import_name 函数则调用位于 *Python/import.c* 的 *PyImport_ImportModuleLevelObject* 函数，接口如下：

```
PyObject *
PyImport_ImportModuleLevelObject(PyObject *name, PyObject *globals,
                                 PyObject *locals, PyObject *fromlist,
                                 int level);
```

函数大部分参数我们已经很熟悉了，但 *globals* 和 *locals* 这两个名字空间有什么作用呢？*Python* 导入模块时可以使用相对路径，这时需要根据 *import* 语句所在模块计算绝对路径，而模块名则保存在 *globals* 名字空间中：

```
from .a.b import c
```

我不打算事无巨细地介绍 *Python/import.c* 中的源码，毕竟超过 *2000* 行的代码量需要相当的篇幅才能讲解清楚。接下来，我力求以最简洁的 *Python* 语言，描述清楚模块的加载流程。鼓励学有余力的童鞋，到源码中探究一番。

解析得到绝对模块名后，*Python* 便开始查找并加载模块。那么，如果模块已经加载过了，*Python* 如何处理呢呢？说来也简单，*Python* 内部用一个 *dict* 对象记录所有已经加载的模块，这个 *dict* 位于 *sys* 模块中：

```
>>> import sys
>>> for name, module in sys.modules.items():
...     print(name)
...
sys
builtins
```

Python 加载模块前，先检查 *sys.modules* ；如果发现目标模块已经加载过，则直接将其返回。因此，一个模块不管被多少 *import* 语句导入，第一次加载后便不再重复加载了。

想要加载被导入模块，*Python* 需要找到模块代码的具体位置位置，这便是 *Python* **模块搜索** 过程。*Python* 在内部维护了一个模块搜索路径的列表，同样位于 *sys* 模块内：

```
>>> import sys
>>> for path in sys.path:
...     print(repr(path))
...
''
'/Users/fasion/opt/pythons/python3/lib/python37.zip'
'/Users/fasion/opt/pythons/python3/lib/python3.7'
'/Users/fasion/opt/pythons/python3/lib/python3.7/lib-dynload'
'/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7'
'/Users/fasion/opt/pythons/python3/lib/python3.7/site-packages'
```

Python 遍历每个路径，直到发现目标模块。如果遍历完所有路径还是没找到目标模块，*Python* 便只好抛异常了。目标模块代码找到后，*Python* 对代码进行编译，生成 *PyCodeObject* 。如果存在 *pyc* 文件，则可以省略编译过程，直接从 *pyc* 文件中加载 *PyCodeObject* 。以导入 *demo* 模块为例：

```
text = read('demo.py')


code = compile(text, 'demo.py', 'exec')
```

至此，*Python* 得到了代表模块逻辑的 **代码** 对象 *PyCodeObject* 。接着，*Python* 创建一个全新的 **模块** 对象。模块对象在 *Python* 内部由 *PyModuleObject* 结构体表示，位于 *Objects/moduleobject.c* 。 **模块** 对象也是内建对象中的一种，有兴趣的童鞋参照 **内建对象** 部分的思路研究源码，在此不再展开。

由于需要直接创建 **模块对象** 的场景极少，*Python* 没有将 **模块类型** 暴露出来，这跟其他内置对象类型略有差别：

```
>>> int
<class 'int'>
>>> module
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'module' is not defined
```

根据 **对象模型** 部分学到的知识，还有另一条路可以找到 **模块类型** 对象——通过已存在的 **模块** 对象顺藤摸瓜！由于每个 *Python* 程序都包含一个 *__main__* 模块，这就是最合适的媒介了：

```
>>> import __main__
>>> __main__
<module '__main__' (built-in)>
```

模块类型 对象便位于 **模块** 对象的 *ob_type* 字段，通过 *__class__* 属性即可获取：

```
>>> module = __main__.__class__
>>> module
<class 'module'>
```

Python 内部便是通过 **模块类型** 对象创建实例对象的，只需提供 **模块名** 以及 **模块文档信息** ：

```
>>> demo = module('demo', 'A test module')
>>> help(demo)
>>> dir(demo)
['__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> demo.__doc__
'A test module'
```

至此，我们得到一个全新的 **模块** 对象，尽管这个模块对象还是个空架子。根据前面章节的内容，我们知道 **模块** 的 **属性空间** 由一个 *dict* 对象实现，这个 *dict* 可通过 *__dict__* 属性找到：

```
>>> demo.__dict__
{'__name__': 'demo', '__doc__': 'A test module', '__package__': None, '__loader__': None, '__spec__':
None}
>>> demo.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'demo' has no attribute 'a'
>>> demo.__dict__['a'] = 1
>>> demo.a
1
```

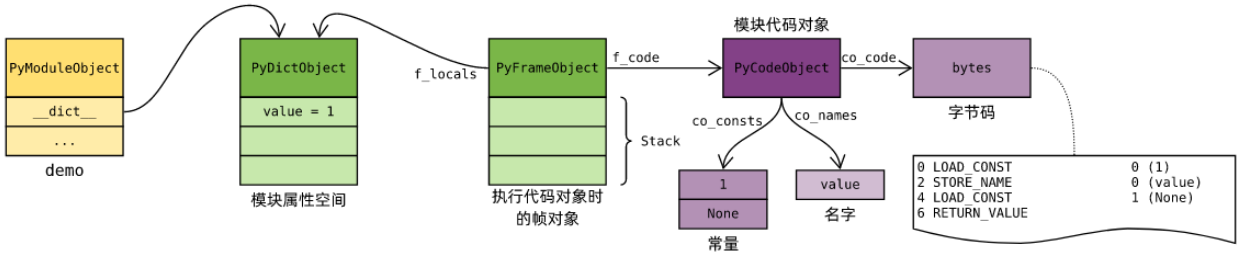
接着，*Python* 执行模块代码对象，完成模块初始化：

```
>>> exec(code, demo.__dict__, demo.__dict__)
```

注意到，模块 **属性空间** 作为 **全局名字空间** 以及 **局部名字空间** 传给了 *exec* 函数。

假设 *demo.py* 只包含一行代码，请大家自行脑补字节码在 *Python* 虚拟机执行的步骤以及结果：

```
value = 1
```



模块 **代码** 对象执行完毕后，*demo* **模块** 对象便具备血肉之躯了！

```
>>> demo.value
1
```

最后，别忘了将 *demo* 模块 对象保存到 *sys.modules* ，以避免不必要的重复加载：

```
>>> import sys
>>> sys.modules['demo'] = demo
```

这便是 *Python* **模块动态加载** 的全过程，其实也并不复杂，对吧？

模块搜索方式

曾几何时，我们对 *ModuleNotFoundError* 异常谈虎色变。明明已经准备好了模块代码，*Python* 为何不认呢？全面掌握 **模块加载** 机制，特别是 **模块搜索** 方式后，我们内心便不再畏惧。

Python 模块搜索路径保存于 *sys.path* 列表中，遇到 *ModuleNotFoundError* 异常首先要排查 *sys.path* 是否包含目标模块所在目录路径。如果目标模块所在路径不在 *sys.path* 中，则需要将其加入：

```
sys.path.append('/some/path')
```

```
sys.path.insert(0, '/some/path')
```

需要特别注意，同一路径不能重复多次加入，不然将影响 *Python* 的搜索效率，甚至导致 **内存泄露** 。我曾经帮一个初学者排查内存泄露问题，最后定位到这样的代码：

```
def some_function():
    sys.path.insert(0, '/some/path')
    import xxxx
```

这意味着函数每调用一次，*sys.path* 便增加一个元素！当函数被频繁调用，*sys.path* 列表最终将耗尽程序内存！

此外，在程序代码中写死模块搜索路径的方式并不可取。更优雅的方式是通过 *PYTHONPATH* 环境变量指定：

```
PYTHONPATH=/some/path python xxxx.py
```

这样一来，程序启动后给定路径 */some/path* 便在 *sys.path* 列表中了。