18 Python 程执行过程与字节码-慕课专栏

我们每天都要编写一些 *Python* 程序，或者用来处理一些文本，或者是做一些系统管理工作。程序写好后，只需敲下 *python* 命令，便可将程序启动起来并开始执行：

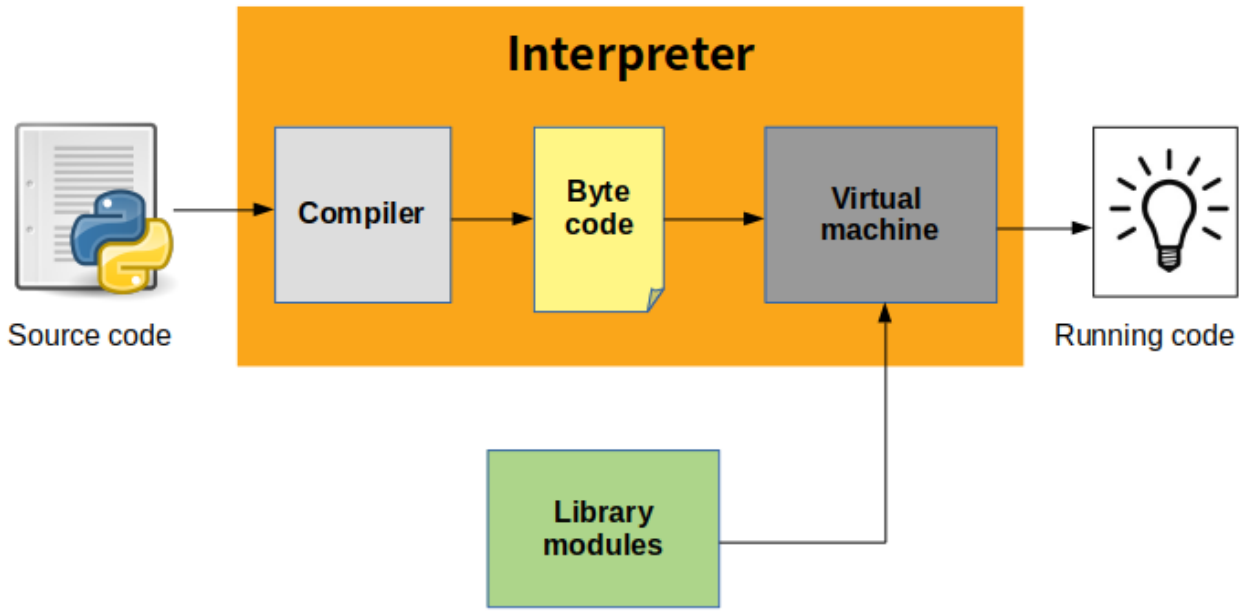
```
$ python some-program.py
```

那么，一个文本形式的 *.py* 文件，是如何一步步转换为能够被 *CPU* 执行的机器指令的呢？此外，程序执行过程中可能会有 *.pyc* 文件生成，这些文件又有什么作用呢？带着这些问题我们开始本节的探索。

Python程序执行过程

你也许听过这样的说法： *Python* 是一种解释性语言。这意味着 *Python* 程序不用编译，只需要用一个解释器来执行。事实真的是这样吗？

虽然从行为上看 *Python* 更像 *Shell* 脚本这样的解释性语言，但实际上 *Python* 程序执行原理本质上跟 *Java* 或者 *C#* 一样，都可以归纳为 **虚拟机** 和 **字节码** 。 *Python* 执行程序分为两步：先将程序代码编译成字节码，然后启动虚拟机执行字节码：



虽然 *python* 命令也叫做 *Python* **解释器** (*Interpreter*)，但跟其他脚本语言解释器有本质区别。实际上， *Python* 解释器包含 **编译器** 以及 **虚拟机** 两部分。当 *Python* 解释器启动后，主要执行以下两个步骤：

1. **编译器** 将 *.py* 文件中的 *Python* 源码编译成 **字节码** ；
2. **虚拟机** 逐行执行编译器生成的 **字节码** ；

因此， *.py* 文件中的 *Python* 语句并没有直接转换成机器指令，而是转换成 *Python* **字节码** 。

字节码

好了，我们知道 *Python* 程序的 **编译结果** 是字节码，里面应该藏着不少 *Python* 运行的秘密。因此，不管是为了更深入理解 *Python* 虚拟机运行机制，还是为了调优 *Python* 程序运行效率，字节码都是绕不过去的一关。那么， *Python* 字节码到底长啥样呢？我们如何才能获得一个 *Python* 程序的字节码呢？

为了回答以上问题，我们需要深入 *Python* 解释器源码，研究 *Python* **编译器** 。但出于几方面考虑，我不打算深入介绍 *Python* 编译器：

① *Python* 编译器工作原理与其他任何语言类似，市面上任何一本编译原理均有介绍；

② 编译原理是计算机基础学科，不是 *Python* 特有的，不在本专栏的篇幅内；

③ 能够影响 *Python* 编译过程的手段非常有限，研究 *Python* 编译器对开发工作帮助不大。因此，我们只需要知道 *Python* 解释器背后有一个编译器负责将源码编译成字节码即可， **字节码以及虚拟机才是我们重点研究的对象** 。

那我们还怎么研究字节码呀？别急， *Python* 提供了一个内置函数 *compile* 用于即时编译源码。我们只需将待编译源码作为参数调用 *compile* 函数，即可获得源码的编译结果。

源码编译

接下来，我们调用 *compile* 函数编译一个例子程序，以此演示该函数的用法：

```
PI = 3.14

def circle_area(r):
    return PI * r ** 2

class Dog(object):

    def __init__(self, name):
        self.name = name

    def yelp(self):
        print('woof, i am', self.name)
```

假设这段源码保存于 *demo.py* 文件，开始编译之前需要将源码从文件中读取出来：

```
>>> text = open('demo.py').read()
>>> print(text)
PI = 3.14

def circle_area(r):
    return PI * r ** 2

class Dog(object):

    def __init__(self, name):
        self.name = name

    def yelp(self):
        print('woof, i am', self.name)
```

接着，调用 *compile* 函数编译源码：

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

***compile* 函数必填的参数有 *3* 个：**

- ***source*，待编译 源码 ；**
- ***filename*，源码所在 文件名 ；**
- ***mode*， 编译模式， *exec* 表示将源码当做一个模块来编译；**

顺便提一下， *compile* 函数有 *3* 中不同的 编译模式 可供选择：

- ***exec*，用于编译模块源码；**
- ***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；**

***compile* 详细用法请参考 *Python* 文档，运行 *help* 内建函数可快速查看：**

```
>>> help(compile)
```

PyCodeObject

我们接着看源码编译结果到底是个什么东西：

```
>>> result
<code object <module> at 0x103d21150, file "demo.py", line 1>
>>> result.__class__
<class 'code'>
```

看上去我们得到了一个 代码对象，代码对象有什么特别的呢？接着顺势扒开 *Include/code.h* 看一看，我们找到了代表代码对象的 *C* 结构体 *PyCodeObject* 。 *PyCodeObject* 定义如下：

```
typedef struct {
    PyObject_HEAD
    int co_argcount;
    int co_kwonlyargcount;
    int co_nlocals;
    int co_stacksize;
    int co_flags;
    int co_firstlineno;
    PyObject *co_code;
    PyObject *co_consts;
    PyObject *co_names;
    PyObject *co_varnames;
    PyObject *co_freevars;
    PyObject *co_cellvars;

    Py_ssize_t *co_cell2arg;
    PyObject *co_filename;
    PyObject *co_name;
    PyObject *co_lnotab;
    void *co_zombieframe;
    PyObject *co_weakreflist;

    void *co_extra;
} PyCodeObject;
```

代码对象 *PyCodeObject* 用于存储编译结果，包括**字节码** 以及代码涉及的 **常量** **名字** 等等。关键字段包括：

字段	用途
co_argcount	参数个数
co_kwonlyargcount	关键字参数个数
co_nlocals	局部变量个数
co_stacksize	执行代码所需栈空间
co_flags	标识
co_firstlineno	代码块首行行号
co_code	指令操作码，也就是字节码
co_consts	常量列表
co_names	名字列表
co_varnames	局部变量名列表

字段	用途
co_freevars	
co_cellvars	

我们终于得到了字节码，尽管它现在看上去如同天书一般：

```
>>> result.co_code
b'd\x00Z\x00d\x01d\x02\x84\x00Z\x01G\x00d\x03d\x04\x84\x00d\x04e\x02\x83\x03Z\x03d\x05S\x00'
```

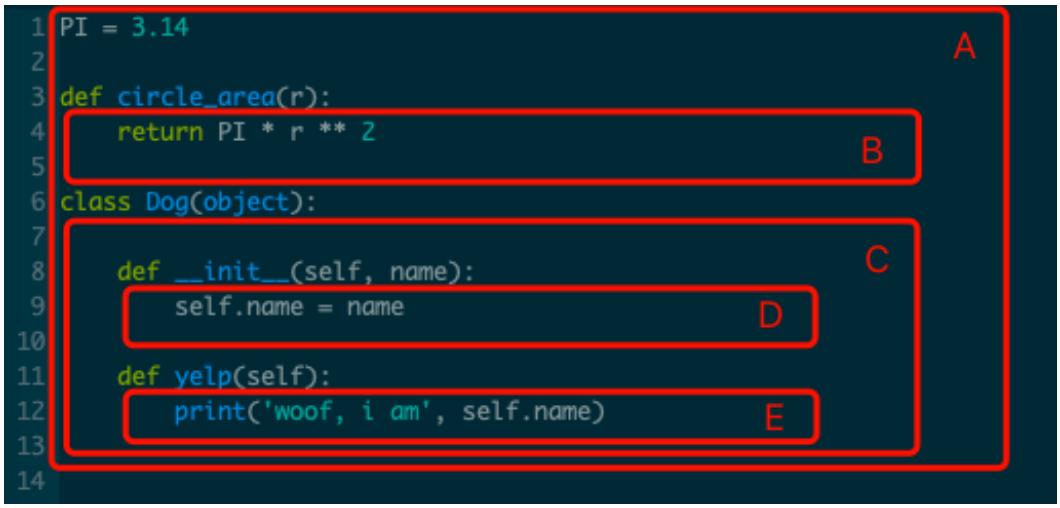
字节码我们现在还无法读懂，放一放。接着研究其他字段，看看名字列表，包含代码对象涉及的所有名字：

```
>>> result.co_names
('PI', 'circle_area', 'object', 'Dog')
```

常量列表则包括代码对象涉及的所有常量：

```
>>> result.co_consts
(3.14, <code object circle_area at 0x10356c5d0, file "demo.py", line 3>, 'circle_area', <code object
Dog at 0x10356cae0, file "demo.py", line 6>, 'Dog', None)
```

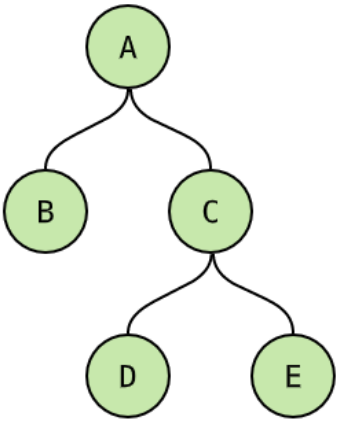
常量列表里还藏着两个代码对象！其中一个对应着 *circle_area* 函数体，另一个对应着 *Dog* 类定义体。回想起 *Python* **作用域** 的划分方式，很自然地联想到：**每个作用域对应着一个代码对象** ！如果这个假设成立， *Dog* 代码对象的常量列表应该还藏着两个代码对象，分别代表 ***init*** 方法和 *yelp* 方法的函数体：

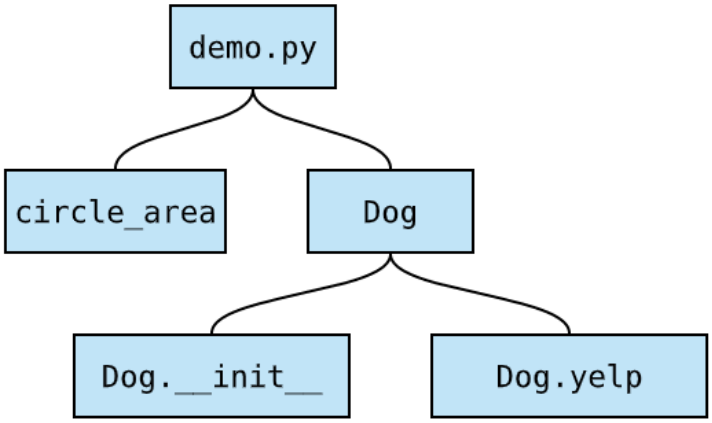


进一步研究代表类 *Dog* 的代码对象，我们发现事实确实如此：

```
>>> dog_code = result.co_consts[3]
>>> dog_code
<code object Dog at 0x10356cae0, file "demo.py", line 6>
>>> dog_code.co_consts
('Dog', <code object __init__ at 0x10356c420, file "demo.py", line 8>, 'Dog.__init__', <code object yelp
at 0x10356c930, file "demo.py", line 11>, 'Dog.yelp', None)
```

因此，我们得到以下结论： *Python* 源码编译后，每个作用域都对应着一个代码对象，子作用域代码对象位于父作用域代码对象的常量列表里，层级一一对应。





至此，我们对 *Python* 源码的编译结果—— **代码对象** 以及其中的 **字节码** 有了最基本的认识。虽然代码对象中的很多字段我们还没来得及研究，但不要紧，我们将在 **虚拟机** 、 **函数机制** 、 **类机制** 的学习中一一揭开这些秘密。

反编译

字节码是一堆不可读的字节序列，跟二进制机器码一样。我们想读懂机器码，可以将其反汇编。那么，字节码是不是也可以反编译呢？答案是肯定的—— *dis* 模块就是干这个事的：

```
>>> import dis
>>> dis.dis(result.co_code)
          0 LOAD_CONST               0 (0)
          2 STORE_NAME               0 (0)
          4 LOAD_CONST               1 (1)
          6 LOAD_CONST               2 (2)
          8 MAKE_FUNCTION            0
         10 STORE_NAME               1 (1)
         12 LOAD_BUILD_CLASS
         14 LOAD_CONST               3 (3)
         16 LOAD_CONST               4 (4)
         18 MAKE_FUNCTION            0
         20 LOAD_CONST               4 (4)
         22 LOAD_NAME                2 (2)
         24 CALL_FUNCTION            3
         26 STORE_NAME               3 (3)
         28 LOAD_CONST               5 (5)
         30 RETURN_VALUE
```

看到没，字节码反编译后的结果多么像汇编语言！其中，第一列是字节码 **偏移量** ，第二列是 **指令** ，第三列是 **操作数** 。以第一条字节码为例， *LOAD_CONST* 指令将常量加载进栈，常量下标由操作数给出。而下标为 *0* 的常量是：

```
>>> result.co_consts[0]
3.14
```

我们成功解开了第一条字节码：将常量 *3.14* 加载到栈！对其他字节码的解读也是类似的。

由于代码对象保存了常量、名字等上下文信息，因此直接对代码对象进行反编译可以得到更为清晰的结果：

```
>>> dis.dis(result)
  1           0 LOAD_CONST               0 (3.14)
              2 STORE_NAME               0 (PI)

  3           4 LOAD_CONST               1 (<code object circle_area at 0x10356c5d0, file "demo.py", line
3>)
              6 LOAD_CONST               2 ('circle_area')
              8 MAKE_FUNCTION            0
             10 STORE_NAME               1 (circle_area)

  6          12 LOAD_BUILD_CLASS
             14 LOAD_CONST               3 (<code object Dog at 0x10356cae0, file "demo.py", line 6>)
             16 LOAD_CONST               4 ('Dog')
             18 MAKE_FUNCTION            0
             20 LOAD_CONST               4 ('Dog')
             22 LOAD_NAME                2 (object)
             24 CALL_FUNCTION            3
             26 STORE_NAME               3 (Dog)
             28 LOAD_CONST               5 (None)
             30 RETURN_VALUE
```

注意到，操作数指定的常量或名字的实际值在旁边的括号内列出。另外，字节码以语句为单位进行分组，中间以空行隔开，语句行号在字节码前面给出。`PI = 3.14` 这个语句编译成以下两条字节码：

```
  1           0 LOAD_CONST               0 (3.14)
              2 STORE_NAME               0 (PI)
```

pyc

如果将 *demo* 作为模块导入， *Python* 将在 *demo.py* 文件所在目录下生成 *.pyc* 文件：

```
>>> import demo

$ ls __pycache__
demo.cpython-37.pyc
```

pyc 文件保存经过序列化处理的代码对象 *PyCodeObject* 。这样一来， *Python* 后续导入 *demo* 模块时，直接读取 *pyc* 文件并反序列化即可得到代码对象，避免了重复编译导致的开销。只有 *demo.py* 有新修改(时间戳比 *pyc* 文件新)， *Python* 才会重新编译。

因此， *Python* 中的 *.py* 文件可以类比 *Java* 中的 *.java* 文件，都是源码文件；而 *.pyc* 文件可以类比 *.class* 文件，都是编译结果(字节码)。只不过 *Java* 程序需要先用编译器 *javac* 命令来编译，再用虚拟机 *java* 命令来执行；而 *Python* 解释器把这个两个活都干了，更加智能。

小结

Python **程序** 由 **解释器** *python* 命令执行， *Python* 解释器中包含一个 **编译器** 和一个 **虚拟机** 。 *Python* 解释器执行 *Python* 程序时，分为以下两步：

1. **编译器** 将 *.py* 文件中的 *Python* 源码编译成 **字节码** ；
2. **虚拟机** 逐行执行编译器生成的 **字节码** ；

Python 源码的编译结果是代码对象 *PyCodeObject* ，对象中保存了 **字节码** 、 **常量** 以及 **名字** 等信息，代码对象与源码作用域一一对应。 *Python* 将编译生成的代码对象保存在 *.py* 文件中，以避免不必要的重复编译，提高效率。