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

原创 古明地觉 古明地觉的编程教室 2022-03-07 08:30

收录于合集
#CPython
97个 >

楔子

当我们想要执行一个py文件的时候，只需要**python xxx.py**即可，但是你有没有想过这背后的流程是怎么样的呢？

从现在开始我们就进入Python虚拟机的环节了，之前都是在介绍Python的一些内置对象，但是虚拟机的执行流程、以及背后的原理也是值得我们关注的。

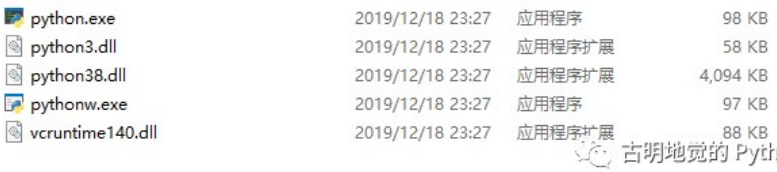
这里我们先来说一下解释器执行py文件的流程：

1. 首先将文件里面的内容读取出来，所以从这个角度上讲，文件名不一定非要是.py结尾，.txt也是可以的，只要文件里面的内容符合Python代码规范即可；
2. 读取文件里面的内容之后会对其进行分词，将源代码切分成一个一个的token；
3. 然后Python编译器会对token进行语法解析，建立抽象语法树(AST，abstract syntax tree)；
4. 编译器再将得到的AST编译成PyCodeObject对象；
5. 最终由Python虚拟机来执行字节码；

这里我们看到了Python编译器、Python虚拟机，而且我们平常还会说Python解释器，那么三者之间有什么区别呢？

实际上**Python解释器=Python编译器+Python虚拟机**，Python编译器负责将Python源代码编译成PyCodeObject对象，然后交给Python虚拟机来执行。

那么Python编译器和Python虚拟机都在什么地方呢？如果打开Python的安装目录，会发现有一个python.exe，点击的时候会通过它来启动一个终端。



但问题是这个文件大小还不到100K，不可能容纳一个编译器加一个虚拟机，所以下面还

有一个python38.dll，没错，编译器、虚拟机都藏身于python38.dll当中。

因此Python虽然是解释型语言，但也有编译的过程。Python源代码会被Python编译器编译成PyCodeObject对象，然后再交给虚拟机来执行。而之所以要存在编译，是为了提前分配好常量、能够让虚拟机更快速地执行，而且还可以尽早检测出语法上的错误。

那么下面我们就来看看PyCodeObject对象长什么样子。

PyCodeObject对象和pyc文件的关系

我们知道Python代码的编译结果是PyCodeObject，所以里面必然隐藏了Python运行的秘密，因此不管是深入理解虚拟机还是调优Python的运行效率，了解PyCodeObject都是绕不过去的一个坎。

但是需要注意的是，我们这里会研究PyCodeObject，但是不会研究Python是怎么编译得到的它。因为Python编译器的工作原理和其它语言基本类似，很多关于编译原理的书籍都有介绍，编译这个过程不是Python特有的。并且研究Python的编译过程，对于我们开发帮助不是很大。

所以我们只需要知道Python解释器的背后有一个编译器会通过**读取文件**、**对源代码分词**、**分词之后会语法解析并建立AST**、**对AST编译**得到PyCodeObject对象即可。至于这一列步骤是怎么做的，是怎么将源代码变成PyCodeObject对象不是我们所关心的，我们的重点是研究对象本身以及虚拟机。

在Python开发时，我们肯定都见过这个pyc文件，它一般位于**__pycache__**目录中，那么这个pyc文件和PyCodeObject之间有什么关系呢？

首先我们都知道字节码，虚拟机的执行实际上就是对字节码不断解析的一个过程。然而除了字节码之外，还应该包含一些其它的信息，这些信息也是Python运行的时候所必需的，比如常量、变量名等等。

我们常听到**py文件被编译成字节码**，这句话其实不太严谨，实际上字节码只是一个PyBytesObject对象、或者说一段字节序列。但很明显，光有字节码是不够的，还有很多的静态信息也需要被收集起来，它们整体被称为PyCodeObject。

而PyCodeObject对象中有一个成员co_code，它是一个指针，指向了这段字节序列。但是这个对象除了有co_code指向的字节码之外，还有很多其它成员，负责保存代码涉及到的常量、变量(名字、符号)等等。

但是问题来了，难道每一次执行都要将源文件编译一遍吗？如果没有对源文件进行修改的话，那么完全可以使用上一次的编译结果。相信此时你能猜到pyc文件是干什么的了，它就是负责保存编译之后的PyCodeObject对象。

所以我们知道了，pyc文件里面的内容是PyCodeObject对象。对于Python编译器来说，PyCodeObject对象是对源代码编译之后的结果，而pyc文件则是这个对象在硬盘上表现形式。

在程序运行期间，编译结果存在于内存的PyCodeObject对象当中，而Python结束运行之后，编译结果又被保存到了pyc文件当中。当下一次运行的时候，Python会根据pyc文件中记录的编译结果直接建立内存中的PyCodeObject对象，而不需要再度重新编译了，当然前提是没有对源文件进行修改。

PyCodeObject的底层结构

我们来看一下这个结构体长什么样子，它的定义位于Include/code.h中。

```
typedef struct {
    //头部信息，我们看到真的一切皆对象，字节码也是个对象
    PyObject_HEAD
    //可以通过位置参数传递的参数个数
    int co_argcount;
    //只能通过位置参数传递的参数个数，Python3.8新增
    int co_posonlyargcount;
    //只能通过关键字参数传递的参数个数
    int co_kwonlyargcount;
    //代码块中局部变量的个数，也包括参数
    int co_nlocals;
    //执行该段代码块需要的栈空间
    int co_stacksize;
    //参数类型标识
    int co_flags;
    //代码块在对应文件的行号
    int co_firstlineno;
    //指令集，也就是字节码，它是一个bytes对象
    PyObject *co_code;
    //常量池，一个元组，保存代码块中的所有常量
    PyObject *co_consts;
    //一个元组，保存代码块中引用的其它作用域的变量
    PyObject *co_names;
    //一个元组，保存当前作用域中的变量
    PyObject *co_varnames;
    //内层函数引用的外层函数的作用域中的变量
    PyObject *co_freevars;
    //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
    PyObject *co_cellvars;
    //无需关注
    Py_ssize_t *co_cell2arg;
    //代码块所在的文件名
    PyObject *co_filename;
    //代码块的名字，通常是函数名、类名，或者文件名
    PyObject *co_name;
    //字节码指令与python源代码的行号之间的对应关系
    //以PyByteObject的形式存在
    PyObject *co_lnotab;

    //剩下的无需关注了
    void *co_zombieframe;
    PyObject *co_weakreflist;
    void *co_extra;
    unsigned char *co_opcache_map;
    _PyOpcache *co_opcache;
    int co_opcache_flag;
    unsigned char co_opcache_size;
} PyCodeObject;
```

这里面的每一个成员，我们后面都会逐一演示进行说明。总之Python编译器在对Python源代码进行编译的时候，对于代码中的每一个block，都会创建一个PyCodeObject与之对应。

但是多少代码才算得上是一个block呢？事实上，Python有一个简单而清晰的规则：当进入一个新的名字空间，或者说作用域时，就算是进入了一个新的block了。这里又引出了名字空间，别急，我们后面会一点一点说，总之先举个栗子：

```
class A:
```

```
    a = 123

def foo():
    a = []
```

我们仔细观察一下上面这个文件，它在编译完之后会有三个PyCodeObject对象，一个是对应整个**py文件(模块)**的，一个是对应**class A**的，一个是对应**def foo**的。因为这是三个不同的作用域，所以会有三个PyCodeObject对象。

在这里，我们开始提及Python中一个至关重要的概念：**名字空间(name space)**、也叫**命名空间**、**名称空间**，都是一个东西。名字空间是符号的上下文环境，符号的含义取决于名字空间。更具体的说，一个变量名对应的变量值是什么，在Python里面是不确定的，需要由名字空间来确定。

Python的变量只是一个名字，或者说符号。比如说上面代码中的**变量a**，在某个名字空间中，它可能指向一个PyLongObject对象；而在另一个名字空间中，它可能指向一个PyListObject对象。但是在一个名字空间中，一个符号只能有一种含义。

而且名字空间可以一层套一层，形成一条名字空间链，Python虚拟机在执行的时候，会有很大一部分时间消耗在从名字空间链中确定一个名字指向的对象是谁。这也侧面说明了，Python为什么比较慢。

如果你现在对名字空间还不是很了解，不要紧，随着剖析的深入，你一定会对名字空间和Python在名字空间链上的行为有着越来越深刻的理解。

总之现在需要记住的是：一个**code block**对应一个**名字空间(或者说作用域)**、同时也对应一个**PyCodeObject对象**。Python的类、函数、模块都对应着一个独自的名字空间，因此也都会有一个PyCodeObject对象与之对应。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》44. 解析PyCodeObject对象

下一篇 ›
《源码探秘 CPython》42. 迭代器的实现原理