这篇文章准确说是『Python 源码剖析』的读书笔记，整理完之后才发现很长，那就将就看吧。

**1. 简单的例子**

先从一个简单的例子说起，包含了两个文件 foo.py 和 demo.py

[foo.py]

def add(a, b):

return a + b

[demo.py]

import foo

a = [1, 'python']

a = 'a string'

def func():

a = 1

b = 257

print(a + b)

print(a)

if \_\_name\_\_ == '\_\_main\_\_':

func()

foo.add(1, 2)

执行这个程序

python demo.py

输出结果

a string

258

同时，该文件目录多出一个 foo.pyc 文件

**2. 背后的魔法**

看完程序的执行结果，接下来开始一行行解释代码。

**2.1 模块**

Python 将 .py 文件视为一个 module，这些 module 中，有一个主 module，也就是程序运行的入口。在这个例子中，主 module 是 demo.py。

**2.2 编译**

执行 python demo.py 后，将会启动 Python 的解释器，然后将 demo.py 编译成一个字节码对象 PyCodeObject。

有的人可能会很好奇，编译的结果不应是 pyc 文件吗，就像 Java 的 class 文件，那为什么是一个对象呢，这里稍微解释一下。

在 Python 的世界中，一切都是对象，函数也是对象，类型也是对象，类也是对象（类属于自定义的类型，在 Python 2.2 之前，int, dict 这些内置类型与类是存在不同的，在之后才统一起来，全部继承自 object），甚至连编译出来的字节码也是对象，.pyc 文件是字节码对象（PyCodeObject）在硬盘上的表现形式。

在运行期间，编译结果也就是 PyCodeObject 对象，只会存在于内存中，而当这个模块的 Python 代码执行完后，就会将编译结果保存到了 pyc 文件中，这样下次就不用编译，直接加载到内存中。pyc 文件只是 PyCodeObject 对象在硬盘上的表现形式。

这个 PyCodeObject 对象包含了 Python 源代码中的字符串，常量值，以及通过语法解析后编译生成的字节码指令。PyCodeObject 对象还会存储这些字节码指令与原始代码行号的对应关系，这样当出现异常时，就能指明位于哪一行的代码。

### 2.3 pyc 文件

一个 pyc 文件包含了三部分信息：Python 的 magic number、pyc 文件创建的时间信息，以及 PyCodeObject 对象。

magic number 是 Python 定义的一个整数值。一般来说，不同版本的 Python 实现都会定义不同的 magic number，这个值是用来保证 Python 兼容性的。比如要限制由低版本编译的 pyc 文件不能让高版本的 Python 程序来执行，只需要检查 magic number 不同就可以了。由于不同版本的 Python 定义的字节码指令可能会不同，如果不做检查，执行的时候就可能出错。

下面所示的代码可以来创建 pyc 文件，使用方法

python generate\_pyc.py module\_name

例如

python generate\_pyc.py demo

[generate\_pyc.pyc]

import imp

import sys

def generate\_pyc(name):

fp, pathname, description = imp.find\_module(name)

try:

imp.load\_module(name, fp, pathname, description)

finally:

if fp:

fp.close()

if \_\_name\_\_ == '\_\_main\_\_':

generate\_pyc(sys.argv[1])

### 2.4 字节码指令

为什么 pyc 文件也称作字节码文件？因为这些文件存储的都是一些二进制的字节数据，而不是能让人直观查看的文本数据。

Python 标准库提供了用来生成代码对应字节码的工具 dis。dis 提供一个名为 dis 的方法，这个方法接收一个 code 对象，然后会输出 code 对象里的字节码指令信息。

s = open('demo.py').read()

co = compile(s, 'demo.py', 'exec')

import dis

dis.dis(co)

执行上面这段代码可以输出 demo.py 编译后的字节码指令

1 0 LOAD\_CONST 0 (-1)

3 LOAD\_CONST 1 (None)

6 IMPORT\_NAME 0 (foo)

9 STORE\_NAME 0 (foo)

3 12 LOAD\_CONST 2 (1)

15 LOAD\_CONST 3 (u'python')

18 BUILD\_LIST 2

21 STORE\_NAME 1 (a)

4 24 LOAD\_CONST 4 (u'a string')

27 STORE\_NAME 1 (a)

6 30 LOAD\_CONST 5 (<code object func at 00D97650, file "demo.py", line 6>)

33 MAKE\_FUNCTION 0

36 STORE\_NAME 2 (func)

11 39 LOAD\_NAME 1 (a)

42 PRINT\_ITEM

43 PRINT\_NEWLINE

13 44 LOAD\_NAME 3 (\_\_name\_\_)

47 LOAD\_CONST 6 (u'\_\_main\_\_')

50 COMPARE\_OP 2 (==)

53 POP\_JUMP\_IF\_FALSE 82

14 56 LOAD\_NAME 2 (func)

59 CALL\_FUNCTION 0

62 POP\_TOP

15 63 LOAD\_NAME 0 (foo)

66 LOAD\_ATTR 4 (add)

69 LOAD\_CONST 2 (1)

72 LOAD\_CONST 7 (2)

75 CALL\_FUNCTION 2

78 POP\_TOP

79 JUMP\_FORWARD 0 (to 82)

>> 82 LOAD\_CONST 1 (None)

85 RETURN\_VALUE

### 2.5 Python 虚拟机

demo.py 被编译后，接下来的工作就交由 Python 虚拟机来执行字节码指令了。Python 虚拟机会从编译得到的 PyCodeObject 对象中依次读入每一条字节码指令，并在当前的上下文环境中执行这条字节码指令。我们的程序就是通过这样循环往复的过程才得以执行。

### 2.6 import 指令

demo.py 的第一行代码是 import foo。import 指令用来载入一个模块，另外一个载入模块的方法是 from xx import yy。用 from 语句的好处是，可以只复制需要的符号变量到当前的命名空间中（关于命名空间将在后面介绍）。

前文提到，当已经存在 pyc 文件时，就可以直接载入而省去编译过程。但是代码文件的内容会更新，如何保证更新后能重新编译而不入旧的 pyc 文件呢。答案就在 pyc 文件中存储的创建时间信息。当执行 import 指令的时候，如果已存在 pyc 文件，Python 会检查创建时间是否晚于代码文件的修改时间，这样就能判断是否需要重新编译，还是直接载入了。如果不存在 pyc 文件，就会先将 py 文件编译。

### 2.7 绝对引入和相对引入

前文已经介绍了 import foo 这行代码。这里隐含了一个问题，就是 foo 是什么，如何找到 foo。这就属于 Python 的模块引入规则，这里不展开介绍，可以参考 [pep-0328](http://legacy.python.org/dev/peps/pep-0328/)。

### 2.8 赋值语句

接下来，执行到 a = [1, 'python']，这是一条赋值语句，定义了一个变量 a，它对应的值是 [1, 'python']。这里要解释一下，[变量](http://wilburding.github.io/blog/2013/05/05/what-is-a-variable/)是什么呢？

按照[维基百科]("<https://en.wikipedia.org/wiki/Variable_(computer_science>") 的解释

变量是一个存储位置和一个关联的符号名字，这个存储位置包含了一些已知或未知的量或者信息。

变量实际上是一个字符串的符号，用来关联一个存储在内存中的对象。在 Python 中，会使用 dict（就是 Python 的 dict 对象）来存储变量符号（字符串）与一个对象的映射。

那么赋值语句实际上就是用来建立这种关联，在这个例子中是将符号 a 与一个列表对象 [1, 'python'] 建立映射。

紧接着的代码执行了 a = 'a string'，这条指令则将符号 a 与另外一个字符串对象 a string 建立了映射。今后对变量 a 的操作，将反应到字符串对象 a string 上。

### 2.9 def 指令

我们的 Python 代码继续往下运行，这里执行到一条 def func()，从字节码指令中也可以看出端倪 MAKE\_FUNCTION。没错这条指令是用来创建函数的。Python 是动态语言，def 实际上是执行一条指令，用来创建函数（class 则是创建类的指令），而不仅仅是个语法关键字。函数并不是事先创建好的，而是执行到的时候才创建的。

def func() 将会创建一个名称为 func 的函数对象。实际上是先创建一个函数对象，然后将 func 这个名称符号绑定到这个函数上。

Python 中是无法实现 C 和 Java 中的重载的，因为重载要求函数名要相同，而参数的类型或数量不同，但是 Python 是通过变量符号（如这里的 func）来关联一个函数，当我们用 def 语句再次创建一个同名的函数时，这个变量名就绑定到新的函数对象上了。

### 2.10 动态类型

继续看函数 func 里面的代码，这时又有一条赋值语句 a = 1。变量 a 现在已经变成了第三种类型，它现在是一个整数了。那么 Python 是怎么实现动态类型的呢？答案就藏在具体存储的对象上。变量 a 仅仅只是一个符号（实际上是一个字符串对象），类型信息是存储在对象上的。在 Python 中，对象机制的核心是类型信息和引用计数（引用计数属于垃圾回收的部分）。

用 type(a)，可以输出 a 的类型，这里是 int

b = 257 跳过，我们直接来看看 print(a + b)，print 是输出函数，这里略过。这里想要探究的是 a + b。

因为 a 和 b 并不存储类型信息，因此当执行 a + b 的时候就必须先检查类型，比如 1 + 2 和 "1" + "2" 的结果是不一样的。

看到这里，我们就可以想象一下执行一句简单的 a + b，Python 虚拟机需要做多少繁琐的事情了。首先需要分别检查 a 和 b 所对应对象的类型，还要匹配类型是否一致（1 + "2" 将会出现异常），然后根据对象的类型调用正确的 + 函数（例如数值的 + 或字符串的 +），（还需要先将变量 MOV 到寄存器）。

### 2.11 命名空间 (namespace)

在介绍上面的这些代码时，还漏掉了一个关键的信息就是命名空间。在 Python 中，类、函数、module 都对应着一个独立的命名空间。而一个独立的命名空间会对应一个 PyCodeObject 对象，所以上面的 demo.py 文件编译后会生成两个 PyCodeObject，只是在 demo.py 这个 module 层的 PyCodeObject 中通过一个变量符号 func 嵌套了一个函数的 PyCodeObject。

命名空间的意义，就是用来确定一个变量符号到底对应什么对象。命名空间可以一个套一个地形成一条命名空间链，Python 虚拟机在执行的过程中，会有很大一部分时间消耗在从这条命名空间链中确定一个符号所对应的对象是什么。

在 Python中，命名空间是由一个 dict 对象实现的，它维护了（name，obj）这样的关联关系。

说到这里，再补充一下 import foo 这行代码会在 demo.py 这个模块的命名空间中，创建一个新的变量名 foo，foo 将绑定到一个 PyCodeObject 对象，也就是 foo.py 的编译结果。

#### 2.11.1 dir 函数

Python 的[内置函数 dir](https://docs.python.org/2/library/functions.html" \l "dir) 可以用来查看一个命名空间下的所有名字符号。一个用处是查看一个命名空间的所有属性和方法（这里的命名空间就是指类、函数、module）。

比如，查看当前的命名空间，可以使用 dir()，查看 sys 模块，可以使用 dir(sys)。

#### 2.11.2 LEGB 规则

Python 使用 LEGB 的顺序来查找一个符号对应的对象

locals -> enclosing function -> globals -> builtins

**locals**，当前所在命名空间（如函数、模块），函数的参数也属于命名空间内的变量

**enclosing**，外部嵌套函数的命名空间（闭包中常见）

def fun1(a):

def fun2():

# a 位于外部嵌套函数的命名空间

print(a)

**globals**，全局变量，函数定义所在模块的命名空间

a = 1

def fun():

# 需要通过 global 指令来声明全局变量

global a

# 修改全局变量，而不是创建一个新的 local 变量

a = 2

**builtins**，内置模块的命名空间。Python 在启动的时候会自动为我们载入很多内置的函数、类，比如 dict，list，type，print，这些都位于 \_\_builtins\_\_ 模块中，可以使用 dir(\_\_builtins\_\_)来查看。这也是为什么我们在没有 import 任何模块的情况下，就能使用这么多丰富的函数和功能了。

介绍完命名空间，就能理解 print(a) 这行代码输出的结果为什么是 a string 了。

### 2.12 内置属性 \_\_name\_\_

现在到了解释 if \_\_name\_\_ == '\_\_main\_\_' 这行代码的时候了。当 Python 程序启动后，Python 会自动为每个模块设置一个属性 \_\_name\_\_ 通常使用的是模块的名字，也就是文件名，但唯一的例外是主模块，主模块将会被设置为 \_\_main\_\_。利用这一特性，就可以做一些特别的事。比如当该模块以主模块来运行的时候，可以运行测试用例。而当被其他模块 import 时，则只是乖乖的，提供函数和功能就好。

### 2.13 函数调用

最后两行是函数调用，这里略去不讲。

## 3. 回顾

讲到最后，还有些内容需要再回顾和补充一下。

### 3.1 pyc 文件

Python 只会对那些以后可能继续被使用和载入的模块才会生成 pyc 文件，Python 认为使用了 import 指令的模块，属于这种类型，因此会生成 pyc 文件。而对于只是临时用一次的模块，并不会生成 pyc 文件，Python 将主模块当成了这种类型的文件。这就解释了为什么 python demo.py执行完后，只会生成一个 foo.pyc 文件。

如果要问 pyc 文件什么时候生成，答案就是在执行了 import 指令之后，from xx import yy 同样属于 import 指令。

### 3.2 小整数对象池

在 demo.py 这里例子中，所用的整数特意用了一个 257，这是为了介绍小整数对象池的。整数在程序中的使用非常广泛，Python 为了优化速度，使用了小整数对象池，避免为整数频繁申请和销毁内存空间。

Python 对小整数的定义是 [-5, 257)，这些整数对象是提前建立好的，不会被垃圾回收。在一个 Python 的程序中，所有位于这个范围内的整数使用的都是同一个对象，从下面这个例子就可以看出。

>>> a = 1

>>> id(a)

40059744

>>> b = 1

>>> id(b)

40059744

>>> c = 257

>>> id(c)

41069072

>>> d = 257

>>> id(257)

41069096

id 函数可以用来查看一个对象的唯一标志，可以认为是内存地址

对于大整数，Python 使用的是一个大整数对象池。这句话的意思是：

每当创建一个大整数的时候，都会新建一个对象，但是这个对象不再使用的时候，并不会销毁，后面再建立的对象会复用之前已经不再使用的对象的内存空间。（这里的不再使用指的是引用计数为0，可以被销毁）

### 3.3 字符串对象缓冲池

如果仔细思考一下，一定会猜到字符串也采用了这种类似的技术，我们来看一下

>>> a = 'a'

>>> b = 'a'

>>> id(a)

14660456

>>> id(b)

14660456

没错，Python 的设计者为一个字节的字符对应的字符串对象 (PyStringObject) 也设计了这样一个对象池。同时还有一个 intern 机制，可以将内容相同的字符串变量转换成指向同一个字符串对象。

intern 机制的关键，就是在系统中有一个（key，value）映射关系的集合，集合的名称叫做 interned。在这个集合中，记录着被 intern 机制处理过的 PyStringObject 对象。不过 Python 始终会为字符串创建 PyStringObject 对象，即便在interned 中已经有一个与之对应的 PyStringObject 对象了，而 intern 机制是在字符串被创建后才起作用。

>>> a = 'a string'

>>> b = 'a string'

>>> a is b

False

>>> a = intern('a string') # 手动调用 intern 方法

>>> b = intern('a string')

>>> a is b

True

关于 intern 函数 可以参考[官方文档](https://docs.python.org/2/library/functions.html#intern)，更多扩展阅读：

<http://stackoverflow.com/questions/15541404/python-string-interning>

值得说明的是，数值类型和字符串类型在 Python 中都是不可变的，这意味着你无法修改这个对象的值，每次对变量的修改，实际上是创建一个新的对象。得益于这样的设计，才能使用对象缓冲池这种优化。

Python 的实现上大量采用了这种内存对象池的技术，不仅仅对于这些特定的对象，还有专门的内存池用于小对象，使用这种技术可以避免频繁地申请和释放内存空间，目的就是让 Python 能稍微更快一点。更多内容可以参考[这里](http://blog.csdn.net/zhzhl202/article/details/7547445)。

如果想了解更快的 Python，可以看看 [PyPy](http://pypy.org/)

### 3.4 import 指令

前文提到 import 指令是用来载入 module 的，如果需要，也会顺道做编译的事。但 import 指令，还会做一件重要的事情就是把 import 的那个 module 的代码执行一遍，这件事情很重要。Python 是解释执行的，连函数都是执行的时候才创建的。如果不把那个 module 的代码执行一遍，那么 module 里面的函数都没法创建，更别提去调用这些函数了。

执行代码的另外一个重要作用，就是在这个 module 的命名空间中，创建模块内定义的函数和各种对象的符号名称（也就是变量名），并将其绑定到对象上，这样其他 module 才能通过变量名来引用这些对象。

Python 虚拟机还会将已经 import 过的 module 缓存起来，放到一个全局 module 集合 sys.modules 中。这样做有一个好处，即如果程序的在另一个地方再次 import 这个模块，Python 虚拟机只需要将全局 module 集合中缓存的那个 module 对象返回即可。

你现在一定想到了 sys.modules 是一个 dict 对象，可以通过 type(sys.modules) 来验证

### 3.5 多线程

demo.py 这个例子并没有用到多线程，但还是有必要提一下。

在提到多线程的时候，往往要关注线程如何同步，如何访问共享资源。Python 是通过一个全局解释器锁 GIL（Global Interpreter Lock）来实现线程同步的。当 Python 程序只有单线程时，并不会启用 GIL，而当用户创建了一个 thread 时，表示要使用多线程，Python 解释器就会自动激活 GIL，并创建所需要的上下文环境和数据结构。

Python 字节码解释器的工作原理是按照指令的顺序一条一条地顺序执行，Python 内部维护着一个数值，这个数值就是 Python 内部的时钟，如果这个数值为 N，则意味着 Python 在执行了 N 条指令以后应该立即启动线程调度机制，可以通过下面的代码获取这个数值。

import sys

sys.getcheckinterval() # 100

线程调度机制将会为线程分配 GIL，获取到 GIL 的线程就能开始执行，而其他线程则必须等待。由于 GIL 的存在，Python 的多线程性能十分低下，无法发挥多核 CPU 的优势，性能甚至不如单线程。因此如果你想用到多核 CPU，一个建议是使用多进程。

### 3.6 垃圾回收

在讲到垃圾回收的时候，通常会使用引用计数的模型，这是一种最直观，最简单的垃圾收集技术。Python 同样也使用了引用计数，但是引用计数存在这些缺点：

* 频繁更新引用计数会降低运行效率
* 引用计数无法解决循环引用问题

Python 在引用计数机制的基础上，使用了主流垃圾收集技术中的标记——清除和分代收集两种技术。