《源码探秘 CPython》45. pyc文件是怎么创建的?

原创 古明地觉 古明地觉的编程教室 2022-03-09 08:30

pyc文件的触发

前面我们提到，每一个代码块(code block)都会对应一个PyCodeObject对象，Python会将该对象存储在pyc文件中。但不幸的是，事实并不总是这样。有时，当我们运行一个简单的程序时并没有产生pyc文件，因此我们猜测：有些Python程序只是临时完成一些琐碎的工作，这样的程序仅仅只会运行一次，然后就不会再使用了，因此也就没有保存至pyc文件的必要。

如果我们在代码中加上了一个**import abc**这样的语句，再执行你就会发现Python为其生成了pyc文件，这就说明import会触发pyc的生成。

实际上，在运行过程中，如果碰到**import abc**这样的语句，那么Python会在设定好的path中寻找abc.pyc或者abc.pyd文件。如果没有这些文件，而是只发现了abc.py，那么Python会先将abc.py编译成PyCodeObject，然后创建pyc文件，并将PyCodeObject写到pyc文件里面去。

接下来，再对abc.pyc进行import动作，对，并不是编译成PyCodeObject对象之后就直接使用。而是先写到pyc文件里面去，然后再将pyc文件里面的PyCodeObject对象重新在内存中复制出来。

关于Python的import机制，我们后面会剖析，这里只是用来完成pyc文件的触发。当然得到pyc文件还有其它方法，比如使用py_compile模块。

```
# a.py
class A:
    a = 1

# b.py
import a
```

执行b.py的时候，会发现创建了**a.cpython-38.pyc**。另外关于pyc文件的创建位置，会在当前文件的同级目录下的**__pycache__**目录中创建，名字就叫做：**py文件名.cpython-版本号.pyc**。

pyc文件里面包含哪些内容

上面我们提到，Python通过**import module**进行加载时，如果没有找到相应的pyc或者pyd文件，就会在py文件的基础上自动创建pyc文件。而创建之后，会往里面写入三个内容：

1. magic number

这是Python定义的一个整数值，不同版本的Python会定义不同的magic number，这个值是为了保证Python能够加载正确的pyc。

比如Python3.7不会加载3.6版本的pyc，因为Python在加载pyc文件的时候会首先检测该pyc的magic number，如果和自身的magic number不一致，则拒绝加载。

2. pyc的创建时间

这个很好理解，判断源代码的最后修改时间和pyc文件的创建时间。如果pyc文件的创建时间比源代码的修改时间要早，说明在生成pyc之后，源代码被修改了，那么会重新编译并生成新的pyc，而反之则会直接加载已存在的pyc。

3. PyCodeObject对象

这个不用说了，肯定是要存储的。

pyc文件的写入

下面就来看看pyc文件是如何写入上面三个内容的。

既然要写入，那么肯定要有文件句柄，我们来看看：

```
//位置:Python/marshal.c

//FILE是 C 自带的文件句柄
//可以把WFILE看成是FILE的包装
typedef struct {
    FILE *fp;  //文件句柄
    //下面的字段在写入信息的时候会看到
    int error;
    int depth;
    PyObject *str;
    char *ptr;
    char *end;
    char *buf;
    _Py_hashtable_t *hashtable;
    int version;
} WFILE;
```

首先是写入magic number和创建时间，它们会调用`PyMarshal_WriteLongToFile`函数进行写入：

```
void
```

```
PyMarshal_WriteLongToFile(long x, FILE *fp, int version)
{
    //magic number和创建时间，只是一个整数
    //在写入的时候，使用char [4]来保存
    char buf[4];
    //声明一个WFILE类型变量wf
    WFILE wf;
    //内存初始化
    memset(&wf, 0, sizeof(wf));
    //初始化内部成员
    wf.fp = fp;
    wf.ptr = wf.buf = buf;
    wf.end = wf.ptr + sizeof(buf);
    wf.error = WFERR_OK;
    wf.version = version;
    //调用w_long将x、也就是版本信息或者时间写到wf里面去
    w_long(x, &wf);
    //刷到磁盘上
    w_flush(&wf);
}
```

所以该函数只是初始化了一个WFILE对象，真正写入则是调用的w_long。

```
static void
w_long(long x, WFILE *p)
{
    w_byte((char)( x      & 0xff), p);
    w_byte((char)((x>> 8) & 0xff), p);
    w_byte((char)((x>>16) & 0xff), p);
    w_byte((char)((x>>24) & 0xff), p);
}
```

w_long则是调用 w_byte 将 x 逐个字节地写到文件里面去。

而写入PyCodeObject对象则是调用了PyMarshal_WriteObjectToFile，我们也来看看长什么样子。

```
void
PyMarshal_WriteObjectToFile(PyObject *x, FILE *fp, int version)
{
    char buf[BUFSIZ];
    WFILE wf;
    memset(&wf, 0, sizeof(wf));
    wf.fp = fp;
    wf.ptr = wf.buf = buf;
    wf.end = wf.ptr + sizeof(buf);
    wf.error = WFERR_OK;
    wf.version = version;
    if (w_init_refs(&wf, version))
        return; /* caller mush check PyErr_Occurred() */
    w_object(x, &wf);
    w_clear_refs(&wf);
    w_flush(&wf);
}
```

可以看到和PyMarshal_WriteLongToFile基本是类似的，只不过在实际写入的时候，PyMarshal_WriteLongToFile调用的是w_long，而PyMarshal_WriteObjectToFile调用的是w_object。

```
static void
w_object(PyObject *v, WFILE *p)
{
    char flag = '\0';

```

```
    p->depth++;

    if (p->depth > MAX_MARSHAL_STACK_DEPTH) {
        p->error = WFERR_NESTEDTOODEEP;
    }
    else if (v == NULL) {
        w_byte(TYPE_NULL, p);
    }
    else if (v == Py_None) {
        w_byte(TYPE_NONE, p);
    }
    else if (v == PyExc_StopIteration) {
        w_byte(TYPE_STOPITER, p);
    }
    else if (v == Py_Ellipsis) {
        w_byte(TYPE_ELLIPSIS, p);
    }
    else if (v == Py_False) {
        w_byte(TYPE_FALSE, p);
    }
    else if (v == Py_True) {
        w_byte(TYPE_TRUE, p);
    }
    else if (!w_ref(v, &flag, p))
        w_complex_object(v, flag, p);

    p->depth--;
}
```

可以看到本质上还是调用了w_byte，但这仅仅是一些特殊的对象。如果是列表、字典之类的数据，那么会调用w_complex_object，也就是代码中的最后一个**else if**分支。

w_complex_object这个函数的源代码很长，我们看一下整体结构，具体逻辑就不贴了，我们后面会单独截取一部分进行分析。

```
static void
w_complex_object(PyObject *v, char flag, WFILE *p)
{
    Py_ssize_t i, n;
    //如果是整数的话，执行整数的写入逻辑
    if (PyLong_CheckExact(v)) {
        //......
    }
    //如果是浮点数的话，执行浮点数的写入逻辑
    else if (PyFloat_CheckExact(v)) {
        if (p->version > 1) {
            //......
        }
        else {
            //......
        }
    }
    //如果是复数的话，执行复数的写入逻辑
    else if (PyComplex_CheckExact(v)) {
        if (p->version > 1) {
            //......
        }
        else {
            //......
        }
    }
    //如果是字节序列的话，执行字节序列的写入逻辑
    else if (PyBytes_CheckExact(v)) {
```

```
        //......
    }
    //如果是字符串的话，执行字符串的写入逻辑
    else if (PyUnicode_CheckExact(v)) {
        if (p->version >= 4 && PyUnicode_IS_ASCII(v)) {
              //......
            }
            else {
                //......
            }
        }
        else {
            //......
        }
    }
    //如果是元组的话，执行元组的写入逻辑
    else if (PyTuple_CheckExact(v)) {
       //......
    }
    //如果是列表的话，执行列表的写入逻辑
    else if (PyList_CheckExact(v)) {
        //......
    }
    //如果是字典的话，执行字典的写入逻辑
    else if (PyDict_CheckExact(v)) {
        //......
    }
    //如果是集合的话，执行集合的写入逻辑
    else if (PyAnySet_CheckExact(v)) {
        //......
    }
    //如果是PyCodeObject对象的话
    //执行PyCodeObject对象的写入逻辑
    else if (PyCode_Check(v)) {
        //......
    }
    //如果是Buffer的话，执行Buffer的写入逻辑
    else if (PyObject_CheckBuffer(v)) {
        //......
    }
    else {
        W_TYPE(TYPE_UNKNOWN, p);
        p->error = WFERR_UNMARSHALLABLE;
    }
}
```

源代码虽然长，但是逻辑非常单纯，就是对不同的对象、执行不同的写动作，然而其最终目的都是通过w_byte写到pyc文件中。了解完函数的整体结构之后，我们再看一下具体细节，看看它在写入对象的时候到底写入了哪些内容？

```
static void
w_complex_object(PyObject *v, char flag, WFILE *p)
{
    //......
    else if (PyList_CheckExact(v)) {
        W_TYPE(TYPE_LIST, p);
        n = PyList_GET_SIZE(v);
        W_SIZE(n, p);
        for (i = 0; i < n; i++) {
            w_object(PyList_GET_ITEM(v, i), p);
        }
    }
    else if (PyDict_CheckExact(v)) {
        Py_ssize_t pos;
```

```
        PyObject *key, *value;
        W_TYPE(TYPE_DICT, p);
        /* This one is NULL object terminated! */
        pos = 0;
        while (PyDict_Next(v, &pos, &key, &value)) {
            w_object(key, p);
            w_object(value, p);
        }
        w_object((PyObject *)NULL, p);
    }
    //......
}
```

以列表和字典为例，它们在写入的时候实际上写的是内部的元素，其它对象也是类似的。

```
def foo():
    lst = [1, 2, 3]

# 把列表内的元素写进去了
print(
    foo.__code__.co_consts
)  # (None, 1, 2, 3)
```

但问题来了，如果只是写入元素的话，那么Python在加载的时候怎么知道它是一个列表呢？所以在写入的时候不能光写数据，类型信息也要写进去。我们再看一下上面列表和字典的写入逻辑，里面都调用了W_TYPE，它负责将类型信息写进去。

因此无论对于哪种对象，在写入具体数据之前，都会先调用W_TYPE将类型信息写进去。如果没有类型信息，那么当Python加载pyc文件的时候，只会得到一坨字节流，而无法解析字节流中隐藏的结构和蕴含的信息。

所以在往pyc文件里写入数据之前，必须先写入一个标识，诸如TYPE_LIST、TYPE_TUPLE、TYPE_DICT等等，这些标识正是对应的类型信息。

如果解释器在pyc文件中发现了这样的标识，则预示着上一个对象结束，新的对象开始，并且也知道新对象是什么样的对象，从而也知道该执行什么样的构建动作。当然，这些标识也是可以看到的，在底层已经定义好了。

```
//marshal.c
#define TYPE_NULL               '0'
#define TYPE_NONE               'N'
#define TYPE_FALSE              'F'
#define TYPE_TRUE               'T'
#define TYPE_STOPITER           'S'
#define TYPE_ELLIPSIS           '.'
#define TYPE_INT                'i'
/* TYPE_INT64 is not generated anymore.
   Supported for backward compatibility only. */
#define TYPE_INT64              'I'
#define TYPE_FLOAT              'f'
#define TYPE_BINARY_FLOAT       'g'
#define TYPE_COMPLEX            'x'
#define TYPE_BINARY_COMPLEX     'y'
#define TYPE_LONG               'l'
#define TYPE_STRING             's'
#define TYPE_INTERNED           't'
#define TYPE_REF                'r'
#define TYPE_TUPLE              '('
#define TYPE_LIST               '['
#define TYPE_DICT               '{'
#define TYPE_CODE               'c'
#define TYPE_UNICODE            'u'
```

```
#define TYPE_UNKNOWN            '?'
#define TYPE_SET                '<'
#define TYPE_FROZENSET          '>'
```

到了这里可以看到，其实Python对PyCodeObject对象的导出实际上是不复杂的。因为不管什么对象，最后都为归结为两种简单的形式，一种是数值写入，一种是字符串写入。

上面都是对数值的写入，比较简单，仅仅需要按照字节依次写入pyc即可。然而在写入字符串的时候，Python设计了一种比较复杂的机制，有兴趣可以自己阅读源码，这里不再介绍。

PyCodeObject的包含关系

有下面一个文件：

```
class A:
    pass

def foo():
    pass
```

显然编译之后会创建三个PyCodeObject对象，但是有两个PyCodeObject对象是位于另一个PyCodeObject对象当中的。

也就是**foo**和**A**对应的PyCodeObject对象，位于**模块**对应的PyCodeObject对象当中，准确的说是位于co_consts指向的常量池当中。举个栗子：

```
def f1():
    def f2():
        pass
    pass

print(
    f1.__code__.co_consts
)  # (None, <code object f2 ...>, 'f1.<locals>.f2')
```

我们看到**f2**对应的PyCodeObject确实位于**f1**的常量池当中，准确的说是**f1**的常量池中有一个指针指向**f2**对应的PyCodeObject。

不过这都不是重点，重点是PyCodeObject对象是可以嵌套的。当在一个作用域内部发现了一个新的作用域，那么新的作用域对应的PyCodeObject对象会位于外层作用域的PyCodeObject对象的常量池中，或者说被常量池中的一个指针指向。

而在写入pyc的时候会从最外层、也就是模块的PyCodeObject对象开始写入。如果碰到了包含的另一个PyCodeObject对象，那么就会递归地执行写入新的PyCodeObject对象。

如此下去，最终所有的PyCodeObject对象都会写入到pyc文件当中。因此pyc文件里的PyCodeObject对象也是以一种嵌套的关系联系在一起的，和代码块之间的关系是保持一致的。

```
def foo():
    pass

def bar():
    pass

```

```
class A:
    def foo(self):
        pass

    def bar(self):
        pass
```

这里问一下，上面那段代码中创建了几个PyCodeObject对象呢?

答案是6个，首先**模块**是一个，**foo**函数一个，**bar**函数一个，**类A**一个，类A里面的**foo函数**一个，类A里面的**bar函数**一个，所以一共是6个。

而且这里的PyCodeObject对象是层层嵌套的，一开始是对整个全局模块创建PyCodeObject对象，然后遇到了函数foo，那么再为函数foo创建PyCodeObject对象，依次往下。

所以，如果是常量值，则相当于是静态信息，直接存储起来便可。可如果是函数、类，那么会为其创建新的PyCodeObject对象，然后再收集起来。

小结

以上就是pyc文件相关的内容，源文件在编译之后会得到pyc文件。因此我们不光可以手动导入 pyc，用Python直接执行pyc文件也是可以的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》46. 虚拟机的执行环境：栈帧对象

下一篇 ›
《源码探秘 CPython》44. 解析PyCodeObject对象