《源码探秘 CPython》49. 虚拟机是怎么执行字节码的?

原创 古明地觉 古明地觉的编程教室 2022-05-25 08:30 发表于北京

收录于合集
#CPython
97个 >

人生总是充满着苦痛，但是仍然要笑脸相迎。

Python虚拟机的运行框架

> 感谢读者 "川味小炒肉"，指出了这篇文章出现的一处错误，故重写。

当Python启动后，首先会进行运行时环境的初始化。注意这里的运行时环境，它和前面说的执行环境是不同的概念。运行时环境是一个全局的概念，而执行环境是一个栈帧，是一个与某个code block相对应的概念。现在不清楚两者的区别不要紧，后面会详细介绍。

关于运行时环境的初始化是一个很复杂的过程，涉及到Python进程、线程的创建，类型对象的完善等非常多的内容，我们后面会单独剖析。这里就假设初始化动作已经完成，我们已经站在了Python虚拟机的门槛外面，只需要轻轻推动第一张骨牌，整个执行过程就像多米诺骨牌一样，一环扣一环地展开。

之前说过，虚拟机执行的不是PyCodeObject对象，而是会在其之上动态构建PyFrameObject对象。构建的时候，会使用以下两个函数：

```
PyObject *
PyEval_EvalCode(PyObject *co, PyObject *globals, PyObject *locals);

PyObject *
PyEval_EvalCodeEx(PyObject *_co, PyObject *globals, PyObject *locals,
                  PyObject *const *args, int argcount,
                  PyObject *const *kws, int kwcount,
                  PyObject *const *defs, int defcount,
                  PyObject *kwdefs, PyObject *closure);
```

PyEval_EvalCodeEx是通用接口，一般用于函数这种带参数的执行场景；PyEval_EvalCode是更高层封装，用于模块等无参数的执行场景。

但这两个函数，最终都会调用_PyEval_EvalCodeWithName函数，创建并初始化栈帧对象。

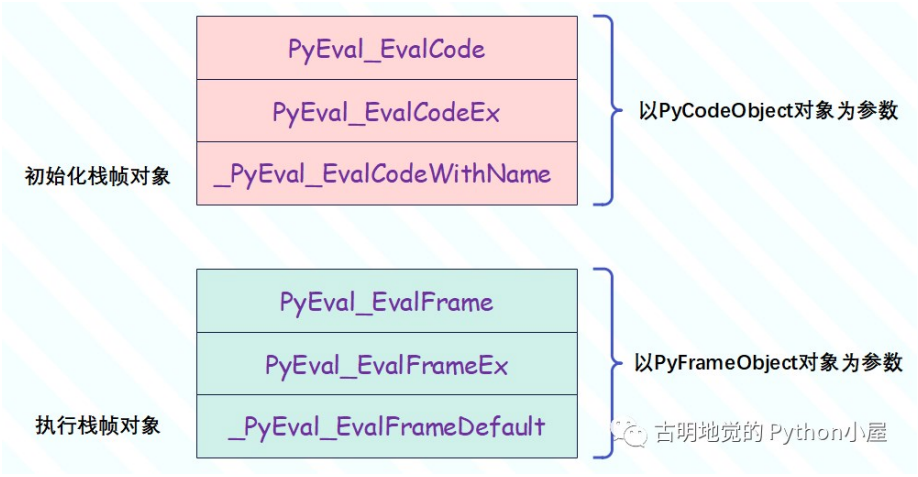
> 栈帧对象将贯穿代码执行的整个生命周期，负责维护执行时所需要的一切上下文信息。

一旦栈帧对象初始化完毕，那么就要进行处理了，处理的时候会调用PyEval_EvalFrame和PyEval_EvalFrameEx函数。

```
PyObject *
PyEval_EvalFrame(PyFrameObject *f);

PyObject *
PyEval_EvalFrameEx(PyFrameObject *f, int throwflag);
```

当然啦，上面这两个函数最终会调用_PyEval_EvalFrameDefault函数，虚拟机执行的秘密就藏在这里。

```
PyObject* _Py_HOT_FUNCTION
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag);
```



_PyEval_EvalFrameDefault函数是虚拟机运行的核心，这一个函数大概在3100行左右。

可以说代码量非常大，但是逻辑并不难理解。

```
// 源代码位于 Python/ceval.c 中
PyObject* _Py_HOT_FUNCTION
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
    //......
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;
    next_instr = first_instr;
    if (f->f_lasti >= 0) {
        assert(f->f_lasti % sizeof(_Py_CODEUNIT) == 0);
        next_instr += f->f_lasti / sizeof(_Py_CODEUNIT) + 1;
    }
    // 栈顶指针
    stack_pointer = f->f_stacktop;
    assert(stack_pointer != NULL);
    f->f_stacktop = NULL;
    //......
}
```

该函数首先会初始化一些变量，PyCodeObject对象包含的信息不用多说，还有一个重要的动作就是初始化堆栈的栈顶指针stack_pointer，使其等于f->f_stacktop，关于stack_pointer后续会细说。

然后栈帧中的f_code就是PyCodeObject对象，PyCodeObject对象里面的co_code域则保存着字节码指令序列。而虚拟机执行字节码就是从头到尾遍历整个co_code、对指令逐条执行的过程。

至于字节码指令序列本身则是一个PyBytesObject对象，对于C而言就是一个普普通通

的字符数组，一条指令就是一个字符、或者说一个整数。而在遍历的时候会使用以下两个变量：

- first_instr：永远指向字节码指令序列的第一条字节码指令；
- next_instr：永远指向下一条待执行的字节码指令；

当然别忘记 f_lasti，它记录了上一条已经执行过的字节码指令的偏移量。

那么这个动作是如何一步步完成的呢？其实就是一个for循环加上一个巨大的switch case结构。

```
PyObject* _Py_HOT_FUNCTION
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
    //......
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;
    //......

    // 死循环，不断遍历字节码指令
    for (;;) {
        if (_Py_atomic_load_relaxed(eval_breaker)) {
        // 读取下一条字节码指令
        // 字节码指令位于:f->f_code->co_code
        // 偏移量由 f->f_lasti 决定
            opcode = _Py_OPCODE(*next_instr);
        //opcode就是字节码指令序列中的每一条指令
        //在Include/opcode.h中定义了大量的指令
            if (opcode == SETUP_FINALLY ||
                opcode == SETUP_WITH ||
                opcode == BEFORE_ASYNC_WITH ||
                opcode == YIELD_FROM) {
                goto fast_next_opcode;
            }

        fast_next_opcode:
            //......
            //判断该指令属于什么操作，然后执行相应的逻辑
            switch (opcode) {
                // 加载常量
                case TARGET(LOAD_CONST):
                    // ....
                    break;
                // 加载变量
                case TARGET(LOAD_NAME):
                    // ...
                    break;
                // ...
        }
    }
}
```

在这个执行架构中，对字节码的遍历是通过宏来实现的：

```
#define INSTR_OFFSET()  \
    (sizeof(_Py_CODEUNIT) * (int)(next_instr - first_instr))

#define NEXTOPARG()  do { \
        _Py_CODEUNIT word = *next_instr; \
        opcode = _Py_OPCODE(word); \
        oparg = _Py_OPARG(word); \
```

```
8         next_instr++; \
9     } while (0)
```

首先每条字节码指令都会带有唯一一个参数，co_code中索引为0 2 4 6 8...的整数便是指令，索引为1 3 5 7 9...的整数便是参数。所以 co_code 里面并不全是字节码指令，每条指令后面都还跟着一个参数。因此next_instr每次向后移动两个字节，便可跳到下一条指令。

> next_instr和first_instr都是 _Py_CODEUNIT * 类型的变量，这个 _Py_CODEUNIT 是一个 uint16_t。所以只要执行 next_instr++，便可向后移动两字节，跳到下一条指令。

然后我们看一下上面的宏，`INSTR_OFFSET`计算的显然就是下一条待执行的指令和第一条指令之间的偏移量；然后是 `NEXTOPARG`，里面的变量 word 显然就是待执行的指令，当然，由于 word 占两字节，所以也包括了参数。其中 word 的前 8 位是指令 opcode，后 8 位是参数 oparg。然后在解析出来指令以及参数之后，再将 next_instr++，继续跳到下一条指令。

而接下来就要执行上面刚解析出来的字节码指令了，会利用switch语句对指令进行判断，根据判断的结果选择不同的case分支。

```
1346        case TARGET(LOAD_CONST): {
1347            PREDICTED(LOAD_CONST);
1348            PyObject *value = GETITEM(consts, oparg);
1349            Py_INCREF(value);
1350            PUSH(value);
1351            FAST_DISPATCH();
1352        }
1353
1354        case TARGET(STORE_FAST): {
1355            PREDICTED(STORE_FAST);
1356            PyObject *value = POP();
1357            SETLOCAL(oparg, value);
1358            FAST_DISPATCH();
1359        }
1360
1361        case TARGET(POP_TOP): {
1362            PyObject *value = POP();
1363            Py_DECREF(value);
1364            FAST_DISPATCH();
1365        }
1366
1367        case TARGET(ROT_TWO): {
1368            PyObject *top = TOP();
1369            PyObject *second = SECOND();
1370            SET_TOP(second);
1371            SET_SECOND(top);
1372            FAST_DISPATCH();
1373        }
1374
1375        case TARGET(ROT_THREE): {
1376            PyObject *top = TOP();
```


每一个case分支，对应一个字节码指令的实现，不同的指令执行不同的case分支。所以这个switch case语句非常的长，函数总共3000多行，这个switch就占了2400行。因为指令非常多，比如：LOAD_CONST、LOAD_NAME、YIELD_FROM等等，而每一个指令都要对应一个case分支。

然后当某个case分支执行完毕时，说明当前的这一条字节码指令执行完毕了，那么虚拟机的执行流程会跳转到标签fast_next_opcode所在位置，或者for循环所在位置。但不管如何，虚拟机接下来的动作就是获取下一条字节码指令和指令参数，完成对下一条指令的执行。

所以，通过for循环一条一条遍历co_code中包含的所有字节码指令，然后交给内部的switch语句、选择不同的case分支进行执行，如此周而复始，最终完成了对Python程序的执行。

尽管目前只是简单的分析，但相信你也能大体地了解Python执行引擎的整体结构。在虚拟机的执行流程进入了那个巨大的for循环，并取出第一条字节码指令交给里面的switch语句之后，第一张多米诺骨牌就已经被推倒，命运不可阻挡的降临了。一条接一条的字

节码如同潮水般涌来，浩浩荡荡，横无际涯。

通过反编译的方式进行演示

指令分为很多种，我们这里就以简单的顺序执行为例，不涉及任何的跳转指令，看看Python是如何执行字节码的。

```
code = """pi = 3.14
r = 3
area = pi * r ** 2
"""
# 将上面的代码以模块的方式进行编译
co = compile(code, "<file>", "exec")
print(co.co_consts)  # (3.14, 3, 2, None)
print(co.co_names)  # ('pi', 'r', 'area')
print(co.co_varnames)  # ()
```

这里需要提一下里面的符号表，在介绍PyCodeObject的时候，我们说过co_names和co_varnames都表示符号表。但co_names指的是当前代码块引用的其它作用域的变量；co_varnames表示当前作用域的变量。

对于函数而言，内部有哪些变量在编译的时候就能确定。如果变量是在自身内部创建的，那么会静态存储在符号表co_varnames当中；如果是引用的其它作用域的变量，那么会静态存储在符号表co_names当中。

但我们上面的代码是以模块的方式编译的，而模块的局部作用域和全局作用域相同，所以符号要通过co_names获取，而co_varnames是一个空元组。

然后我们使用dis.dis(co)进行反编译，看看得到的字节码指令长什么样子：

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

  2           4 LOAD_CONST               1 (3)
              6 STORE_NAME               1 (r)

  3           8 LOAD_NAME                0 (pi)
             10 LOAD_NAME                1 (r)
             12 LOAD_CONST               2 (2)
             14 BINARY_POWER
             16 BINARY_MULTIPLY
             18 STORE_NAME               2 (area)
             20 LOAD_CONST               3 (None)
             22 RETURN_VALUE
```

- 第一列是源代码的行号；
- 第二列是指令的偏移量，或者说该指令在整个字节码指令序列中的索引。因为每条指令后面都跟着一个参数，所以偏移量是 0 2 4 6 8...；
- 第三列是字节码指令，简称指令。指令也叫操作码，它们在宏定义中代表整数；
- 第四列是字节码指令参数，简称指令参数、或者参数，指令参数也叫操作数；
- 第五列是dis模块给我们额外提供的信息，一会说；

我们从上到下依次解释每条指令都干了什么？

- 0 LOAD_CONST：表示加载一个常量（指针），并压入"运行时栈"（关于运行时栈一会儿还会解释）。后面的 0 表示从常量池中加载索引为0的常量、或者说对象，至于 3.14 则表示加载的对象是3.14。所以最后面的括号里面的内容实际上起到的是一个提示作用，告诉你加载的对象是什么。
- 2 STORE_NAME：表示将LOAD_CONST加载的对象用一个名字绑定起来。0 (pi) 则表示使用符号表中索引为0的名字(符号)，且名字为"pi"。
- 4 LOAD_CONST 和 6 STORE_NAME 的作用显然和上面是一样的，只不过后面的索引变成了

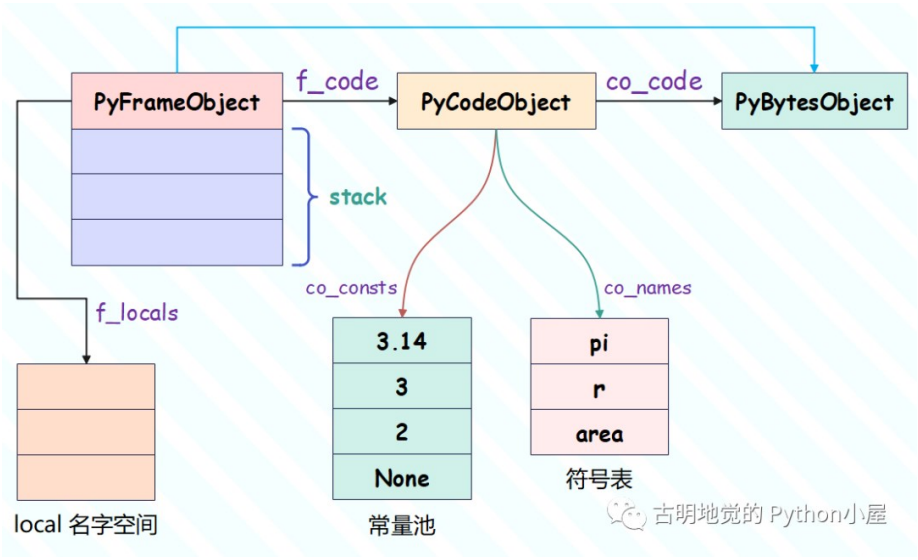
1，表示加载常量池中索引为1的对象、符号表中索引为1的符号(名字)。另外从这里我们也能看出，一行赋值语句实际上对应两条字节码(加载常量、与名字绑定)。

- 8 LOAD_NAME：表示加载符号表中 pi 对应的值。
- 10 LOAD_NAME：表示加载符号表中 r 对应的值。
- 12 LOAD_CONST：表示加载2这个常量，后面的 2 (2) 代表常量池中索引为2的对象是2。
- 14 BINARY_POWER 表示进行幂运算；16 BINARY_MULTIPLY 表示进行乘法运算；18 STORE_NAME 表示用符号表中索引为2的符号(area)和上一步计算的结果进行绑定。
- 20 LOAD_CONST：表示将None加载进来；22 RETURN_VALUE 表示将None返回，虽然它不是在函数里面，但也是有这一步的。

我们通过几张图展示一下上面的过程，为了阅读方便，这里将相应的源代码再贴一份：

```
1  pi = 3.14
2  r = 3
3  area = pi * r ** 2
```

首先虚拟机刚开始执行时，会准备好栈帧对象用于保存执行上下文，关系如下(省略部分信息)：

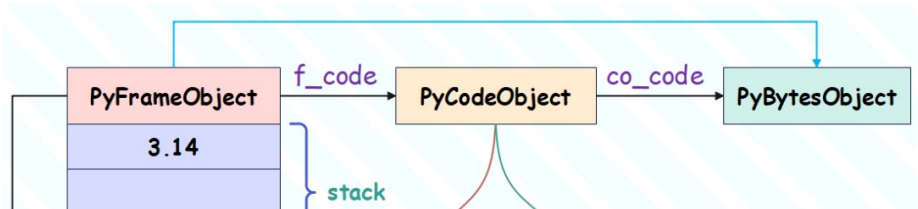


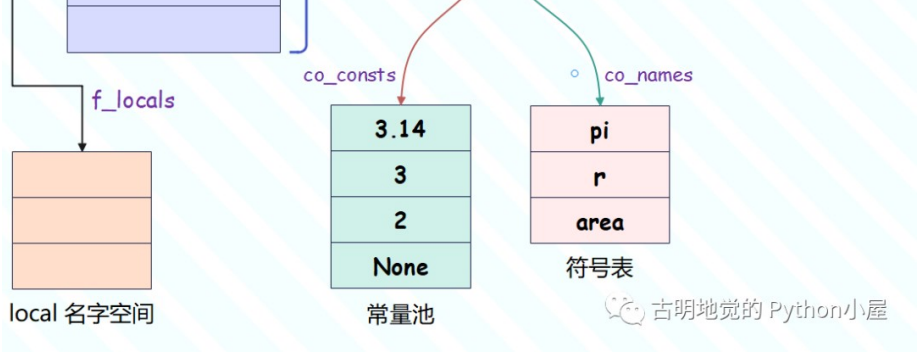
接下来就开始执行字节码了，由于next_instr初始状态指向字节码开头，所以虚拟机开始加载第一条字节码指令：LOAD_CONST 。LOAD_CONST指令表示将常量加载进运行时栈，常量下标由指令参数给出。

> 在源码中，指令（操作码）对应的变量是 opcode，指令参数（操作数）对应的变量是 oparg

```
1  // 代码位于 Python/ceval.c 中
2  case TARGET(LOAD_CONST): {
3      //调用元组的GETITEM方法
4      //从常量池中加载索引为oparg的对象(常量)
5      //当然啦，这里为了方便称其为对象，但其实是指向对象的指针
6      PREDICTED(LOAD_CONST);
7      PyObject *value = GETITEM(consts, oparg);
8      //增加引用计数
9      Py_INCREF(value);
10     //压入运行时栈
11     //这个运行时栈位于栈帧对象的尾部，我们一会儿会说
12     PUSH(value);
13     FAST_DISPATCH();
14 }
```

执行完之后，上面的关系图就变成了下面这样：

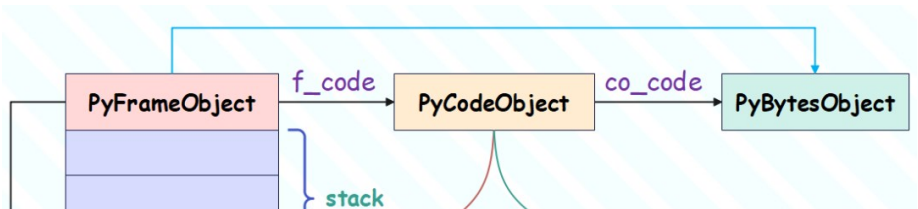


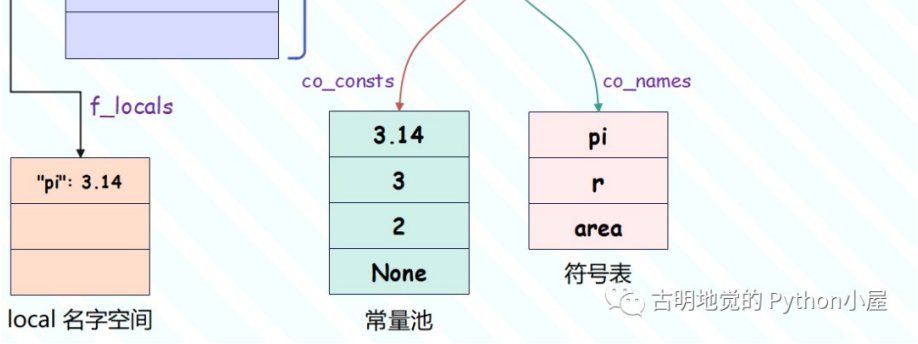


接着虚拟机执行STORE_NAME指令，从符号表中获取索引为0的符号、即pi。然后将栈顶元素3.14弹出，再把**符号pi**和**整数对象3.14**绑定起来保存到local名字空间中。

```
case TARGET(STORE_NAME): {
    //从符号表中加载索引为oparg的符号
    //符号本质上就是一个PyUnicodeObject对象
    PyObject *name = GETITEM(names, oparg);
    //从运行时栈的栈顶弹出元素
    //显然是上一步压入的3.14(指针)
    PyObject *v = POP();
    //获取名字空间namespace
    PyObject *ns = f->f_locals;
    int err;
    if (ns == NULL) {
    //如果没有名字空间则报错
    //这个tstate是和线程密切相关的, 我们后面会说
        _PyErr_Format(tstate, PyExc_SystemError,
                      "no locals found when storing %R", name);
        Py_DECREF(v);
        goto error;
    }
    //将符号和对象绑定起来放在ns中
    //名字空间是一个字典, PyDict_CheckExact则检测ns是否为字典
    //如果不是字典, 那么其类对象一定要继承字典
    if (PyDict_CheckExact(ns))
        //PyDict_CheckExact(ns)类似于type(ns) is dict
        //除此之外, 还有PyDict_Check(ns)
        //它类似于isinstance(ns, dict), 检测标准相对要宽松一些
        //另外, 底层的所有对象都有类似的检测逻辑
        err = PyDict_SetItem(ns, name, v);
    else
       //走到这里说明type(ns)不是dict, 那么它应该继承dict
       //如果不继承, err 会返回非 0
       //此时调用的是PyObject_SetItem, 也就是自己实现的__setitem__
       //如果没有实现, 并继承了字典, 则最终调用的还是字典的__setitem__
        err = PyObject_SetItem(ns, name, v);

    //对象的引用计数减1, 因为从运行时栈中弹出了
    Py_DECREF(v);
    //err!=0, 证明设置元素出错了, 跳转至error标签
    if (err != 0)
        goto error;
    DISPATCH();
}
```

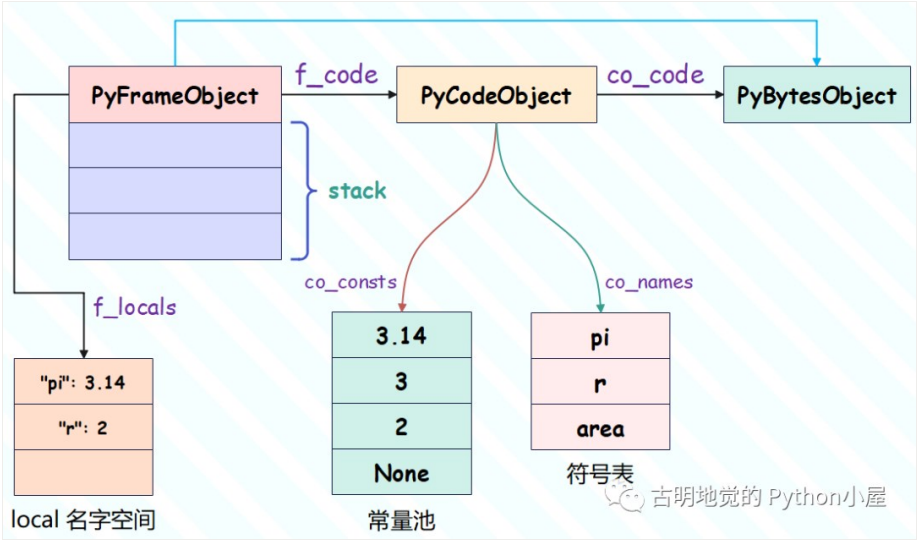
执行完之后，关系图进一步变化，变成下面这样：



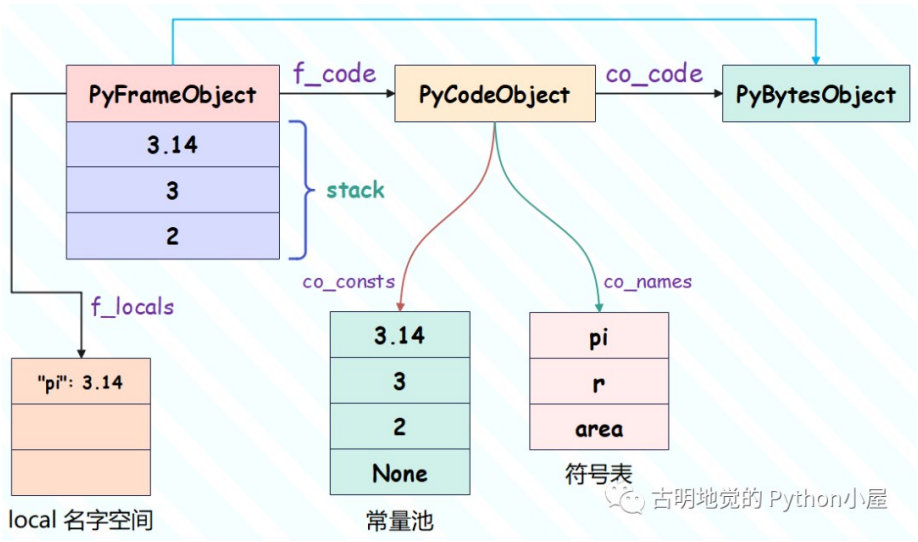


到这里你可能会好奇，变量赋值为啥不直接通过名字空间，而是要跑到临时栈绕一圈？主要原因在于： 每条操作码最多只有一个操作数，因此另一个操作数就只能通过临时栈给出。所以 Python 的字节码设计思想跟CPU精简指令集类似，指令尽量简化，复杂指令由多条简单指令组合完成。

同理，r = 2 对应的两条指令和 pi = 3.14 是类似的，只不过操作数不同。



然后8 LOAD_NAME、10 LOAD_NAME、12 LOAD_CONST，表示将**符号pi**对应的值、**符号r**对应的值，以及**常量2**压入运行时栈。

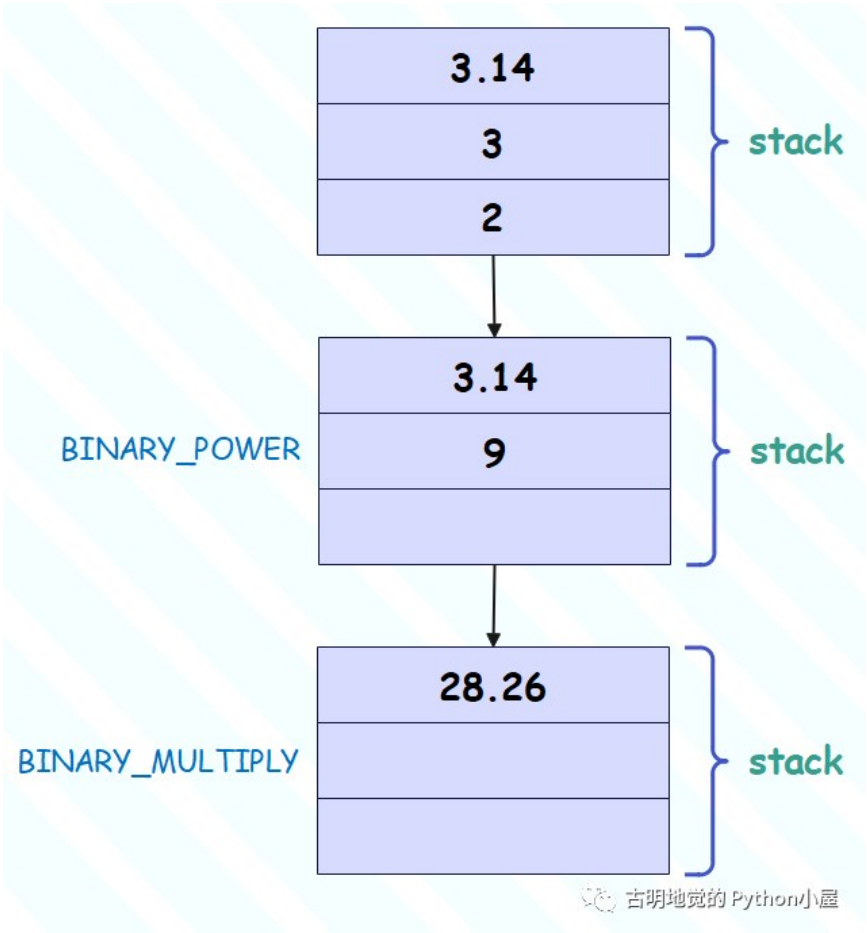


然后14 BINARY_POWER表示幂运算，16 BINARY_MULTIPLY表示乘法运算。其中，BINARY_POWER指令会从栈上弹出两个操作数（指数2 和 底数3）进行幂运算，并将结果 9 压回栈中；BINARY_MULTIPLY 指令则是从栈上弹出 9 和 3.14 进行乘积运算 ，步骤也是类似的。

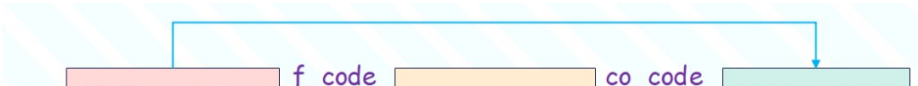
```
case TARGET(BINARY_POWER): {
    //从栈顶弹出元素， 这里是指数2
    PyObject *exp = POP();
    //我们看到这个是TOP
    //所以它不是弹出底数3，而是获取底数3
```

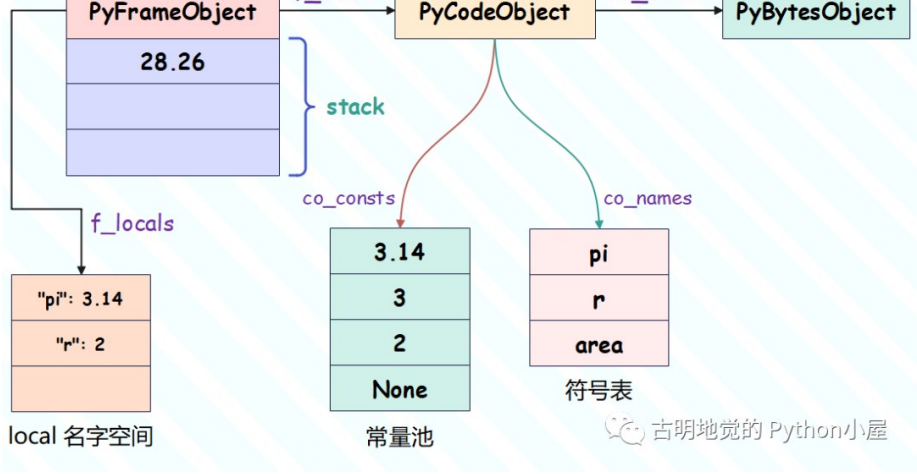
```
    //至于3这个元素，它依旧在栈里面，并且此时位于栈顶
    PyObject *base = TOP();
    //进行幂运算
    PyObject *res = PyNumber_Power(base, exp, Py_None);
    Py_DECREF(base);
    Py_DECREF(exp);
    //将幂运算的结果再设置为栈顶
    //所以原来的3被计算之后的9给替换掉了
    SET_TOP(res);
    if (res == NULL)
        goto error;
        DISPATCH();
}

case TARGET(BINARY_MULTIPLY): {
    //同理这里也是弹出元素9
    PyObject *right = POP();
    //获取元素3.14
    PyObject *left = TOP();
    //乘法运算
    PyObject *res = PyNumber_Multiply(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    //将运算的结果28.26将原来的3.14给替换掉
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

整个运行时栈的变化如下：

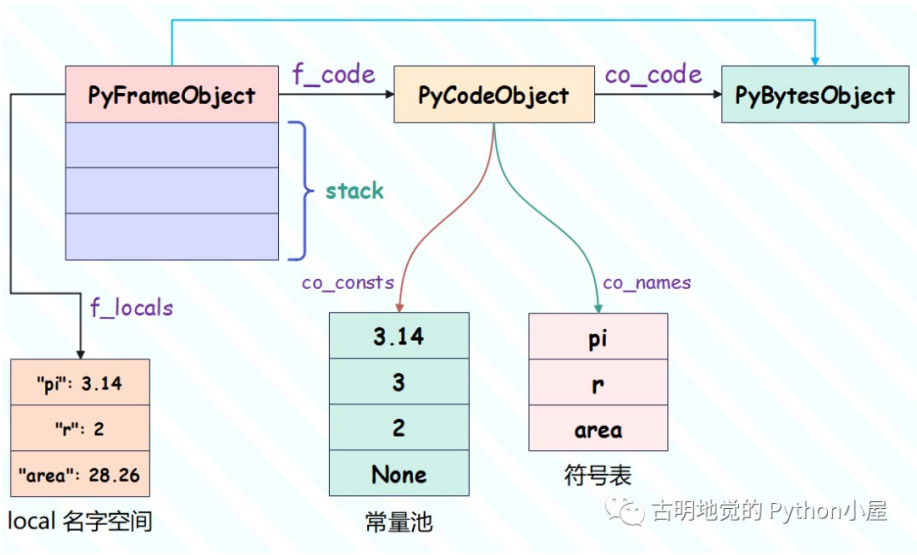


最终栈帧关系图如下：





然后，虚拟机再执行指令18 STORE_NAME，会从符号表中加载索引为2的符号area，并弹出栈顶元素（此时是浮点数28.26），将两者绑定起来放到名字空间中。



整体的执行流程便如上面几张图所示，当然字节码指令有很多，比如除了LOAD_CONST、STORE_NAME之外，还有LOAD_FAST、LOAD_GLOBAL、STORE_FAST，以及if语句、循环语句所使用的跳转指令，运算使用的指令等等等等，这些在后续会慢慢遇到。

> 指令都定义在Include/opcode.h中

栈帧的动态内存空间

在上面反复提到了运行时栈，我们说加载常量的时候会将常量(对象)从常量池中取出、并压入运行时栈；当进行计算或者使用变量绑定的时候，再将它从栈里面弹出来。那么这个运行时栈所需要的空间都保存在什么地方呢？

PyFrameObject中有这么一个属性f_localsplus(可以回头看一下栈帧的定义)，我们说它是动态内存，用于维护局部变量+cell对象集合+free对象集合+运行时栈所需要的空间。因此可以看出这段内存不仅仅是用来给运行时栈使用的，还有别的对象使用。

```
PyFrameObject*
PyFrame_New(PyThreadState *tstate, PyCodeObject *code,
            PyObject *globals, PyObject *locals)
{
    //本质上调用了_PyFrame_New_NoTrack
    PyFrameObject *f = _PyFrame_New_NoTrack(tstate, code, globals, local
s);
    if (f)
        _PyObject_GC_TRACK(f);
```

```
    return f;
}

PyFrameObject* _Py_HOT_FUNCTION
_PyFrame_New_NoTrack(PyThreadState *tstate, PyCodeObject *code,
                     PyObject *globals, PyObject *locals)
{
    //上一级的栈帧，PyThreadState指的是线程对象
    PyFrameObject *back = tstate->frame;
    //当前的栈帧
    PyFrameObject *f;
    //builtin名字空间
    PyObject *builtins;
    /*
    ...
    */
    else {
        Py_ssize_t extras, ncells, nfrees;
        ncells = PyTuple_GET_SIZE(code->co_cellvars);
        nfrees = PyTuple_GET_SIZE(code->co_freevars);
        /*
        ...
        */
        f->f_code = code;
        //extras:局部变量+cell对象集合+free对象集合
        //剩下的那部分空间就是给运行时栈用的
        extras = code->co_nlocals + ncells + nfrees;
        for (i=0; i<extras; i++)
            f->f_localsplus[i] = NULL;
        f->f_locals = NULL;
        f->f_trace = NULL;
    }
    //...
    return f;
}
```

在介绍栈帧的时候，我们说过这样一段话：

> 其实栈帧里面的内存空间分为两部分，一部分是编译代码块需要的空间，另一部分是执行代码块所需要的空间，我们也称之为运行时栈。

实际上这段描述不太严谨，因为在创建栈帧对象时，额外申请的运行时栈对应的空间并不完全是给运行时栈使用的。

> co_freevars、co_cellvars与闭包相关，后续系列介绍

f_localsplus这段连续的内存空间被分成了四份，前三份分别给局部变量、co_freevars、co_cellvars使用，而剩下的那一份才是给真正的运行时栈使用的。

这次我们深入 Python 虚拟机，研究了虚拟机是如何执行字节码的。虚拟机在执行PyCodeObject对象里面的字节码之前，需要先根据PyCodeObject对象创建栈帧对象(PyFrameObject)，用于维护运行时的上下文信息。然后在PyFrameObject的基础上，执行字节码。

而 PyFrameObject 的关键信息包括：

- f_locals: 局部名字空间；
- f_globals：全局名字空间；
- f_builtins：内置名字空间；
- f_code：PyCodeObject对象；
- f_lasti：上一条已执行完毕的字节码指令的偏移量，或者说索引也可以。另外这个f_lasti和生成

器的实现有着密不可分的关系，生成器之所以能够从中断的位置恢复执行，正是因为f_lasti。当然，这些内容就留到介绍生成器的时候再说吧；

- f_back：该栈帧的上一级栈帧、即调用者栈帧；
- f_localsplus：局部变量 + co_freevars + co_cellvars + 运行时栈，这四部分需要的空间；

栈帧对象通过**f_back**串成一个**栈帧调用链**，这与CPU栈帧调用链有异曲同工之妙。此外我们还借助 inspect 模块成功取得栈帧对象(底层是通过sys模块)，并在此基础上输出整个函数调用链。

总的来说Python虚拟机的代码量不小，但是核心并不难理解，主要是_PyEval_EvalFrameDefault里面的一个巨大for循环，更准确的说for循环里面的那个巨型switch语句。该switch语句case了每一个操作指令，当出现什么指令就执行什么操作。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》50. 剖析字节码指令，观测Python程序的执行过程

下一篇 ›
《源码探秘 CPython》48. 名字、作用域、名字空间（下）