20 原来虚拟机是一颗软件 CPU-慕课专栏

上一小节，我们研究了 *Python* 程序的编译过程，以及编译产物—— **代码对象** 。 **编译器** 依照语法规则对源码进行 **作用域** 划分，并以此为单位编译源码，最终为每个作用域生成一个代码对象。代码对象则保存了 **字节码** ，以及相关 **名字** 、 **常量** 等静态上下文信息。

编译器 将源码 **编译** 成代码对象后，便将接力棒传给 **虚拟机** ，由虚拟机负责 **执行** 。那么， *Python* 虚拟机如何解析并执行字节码指令呢？与语法作用域相对应的运行时 **名字空间** ，在虚拟机中又是如何动态维护的呢？带着这些疑问，我们开始本节关于 **虚拟机** 以及 **字节码** 执行过程的探索。

PyFrameObject

由于代码对象是静态的， *Python* 虚拟机在执行代码对象时，需要由一个辅助对象来维护执行上下文。那么，这个执行上下文需要包含什么信息呢？我们先根据已经掌握的知识大开脑洞猜测一下：

首先，我们需要一个动态容器，来存储代码对象作用域中的名字，这也就是 **局部名字空间** (*Locals*)。同理，上下文信息还需要记录 **全局名字空间** (*Globals*)以及 **内建名字空间** (*Builtins*) 的具体位置，确保相关名字查找顺畅。

其次，虚拟机需要保存当前执行字节码指令的编号，就像 *CPU* 需要一个寄存器(*IP*)保存当前执行指令位置一样。

因此，执行上下文理论上至少要包括以下这两方面信息：

- 名字空间
- 当前字节码位置

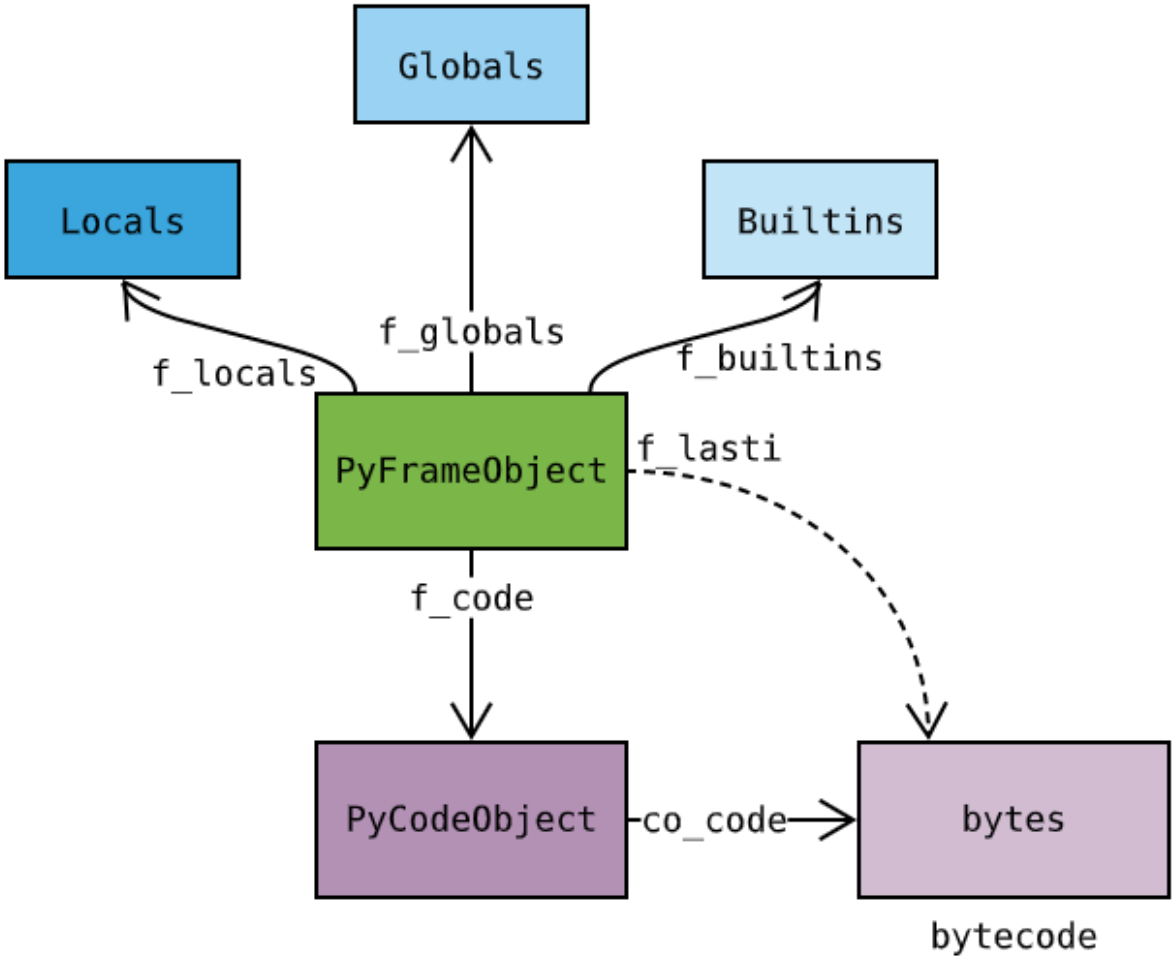
接下来，我们请出执行上下文的真身—— **栈帧对象** *PyFrameObject* ，看看我们有没有猜对。 *PyFrameObject* 在头文件 *Include/frameobject.h* 中定义：

```
typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;
    PyCodeObject *f_code;
    PyObject *f_builtins;
    PyObject *f_globals;
    PyObject *f_locals;
    PyObject **f_valuestack;

    PyObject **f_stacktop;
    PyObject *f_trace;
    char f_trace_lines;
    char f_trace_opcodes;


    PyObject *f_gen;

    int f_lasti;

    int f_lineno;
    int f_iblock;
    char f_executing;
    PyTryBlock f_blockstack[CO_MAXBLOCKS];
    PyObject *f_localsplus[1];
} PyFrameObject;
```

哇，这么多字段！虽然代码很多，但目前我们需要研究的只有以下这些：

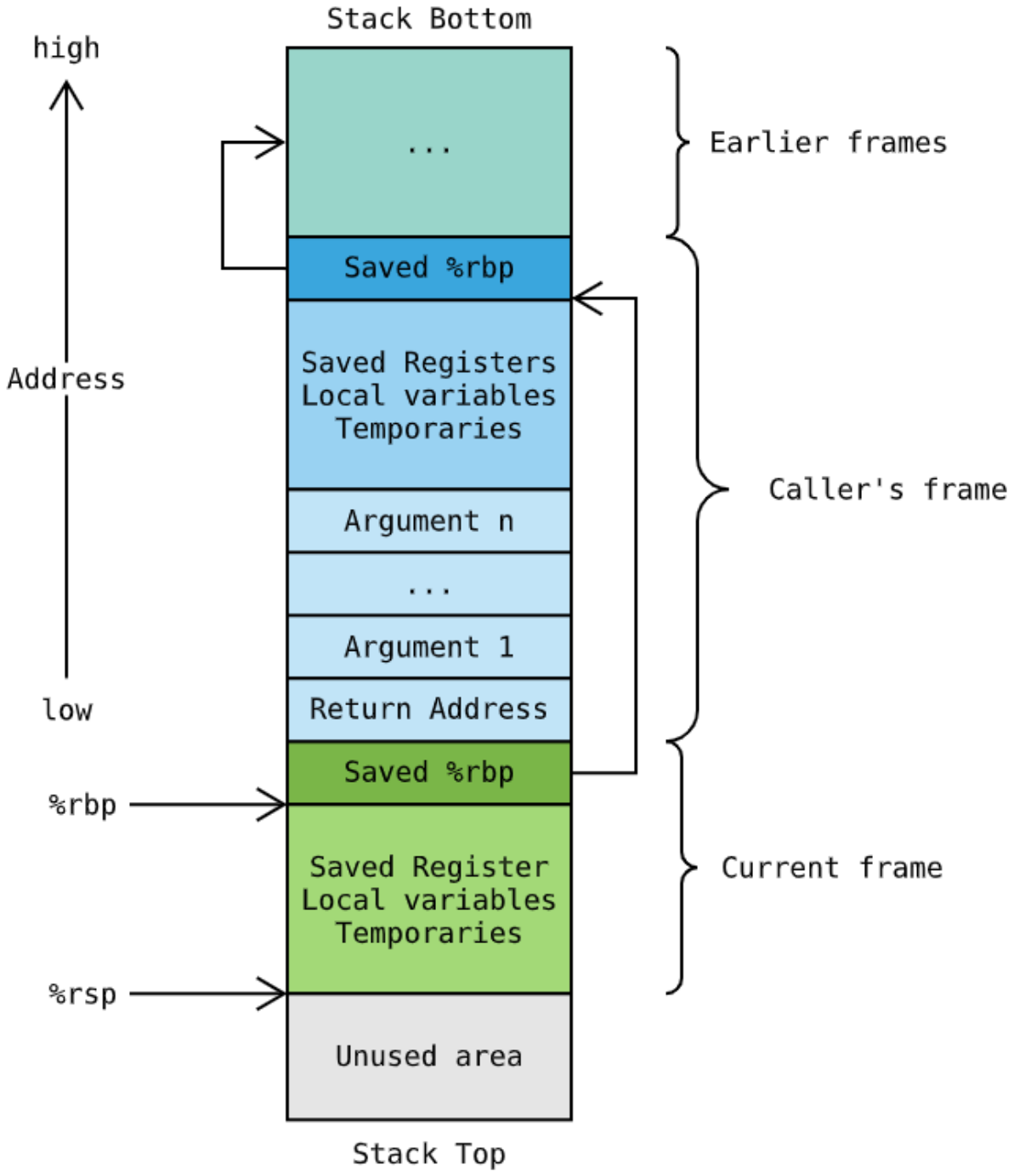
属性	描述
f_back	前一个栈帧对象，也就是调用者
f_builtins	内建名字空间
f_code	代码对象
f_globals	全局名字空间
f_lasti	上条已执行字节码指令编号
f_lineno	源码文件行数
f_locals	局部名字空间
f_localsplus	静态存储的局部名字空间和临时栈

看上去，关键字段我们基本都猜对了。至此，我们搞清楚了栈帧对象的结构以及在运行时所起的作用：



其中，*f_code* 字段保存了当前执行的代码对象，最核心的字节码就在代码对象中。而 *f_lasti* 字段则保存着上条已执行字节码的编号。虚拟机内部用一个 *C* 局部变量 *next_instr* 维护下条字节码的位置，并据此加载下一条待执行的字节码指令，原理跟 *CPU* 的 **指令指针** 寄存器（*%rip*）一样。

另外，注意到 *f_back* 字段指向前一个栈帧对象，也就是 **调用者** 的栈帧对象。这样一来，栈帧对象按照 **调用关系** 串成一个 **调用链** ！这不是跟 *x86 CPU* 栈帧布局如出一辙吗？我们先花点时间回顾一下 *x86 CPU* 栈帧布局与函数调用之间的关系：

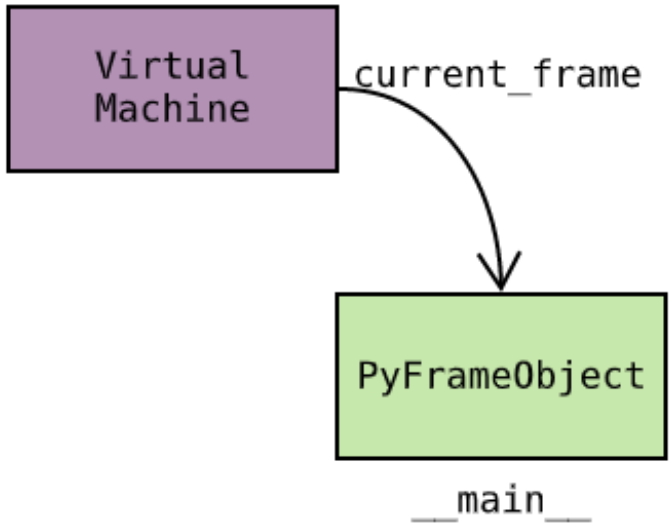


x86 体系处理器通过栈维护调用关系，每次函数调用时在栈上分配一个帧用于保存调用上下文以及临时存储。 *CPU* 中有两个关键寄存器， *%rsp* 指向当前栈顶，而 *%rbp* 则指向当前栈帧。每次调用函数时， **调用者** (*Caller*)负责准备参数、保存返回地址，并跳转到被调用函数代码；作为 **被调用者** (*Callee*)，函数先将当前 *%rbp* 寄存器压入栈（保存调用者栈帧位置），并将 *%rbp* 设置为当前栈顶（保存当前新栈帧位置）。由此， *%rbp* 寄存器与每个栈帧中保存的调用者栈帧地址一起完美地维护了函数调用关系链。

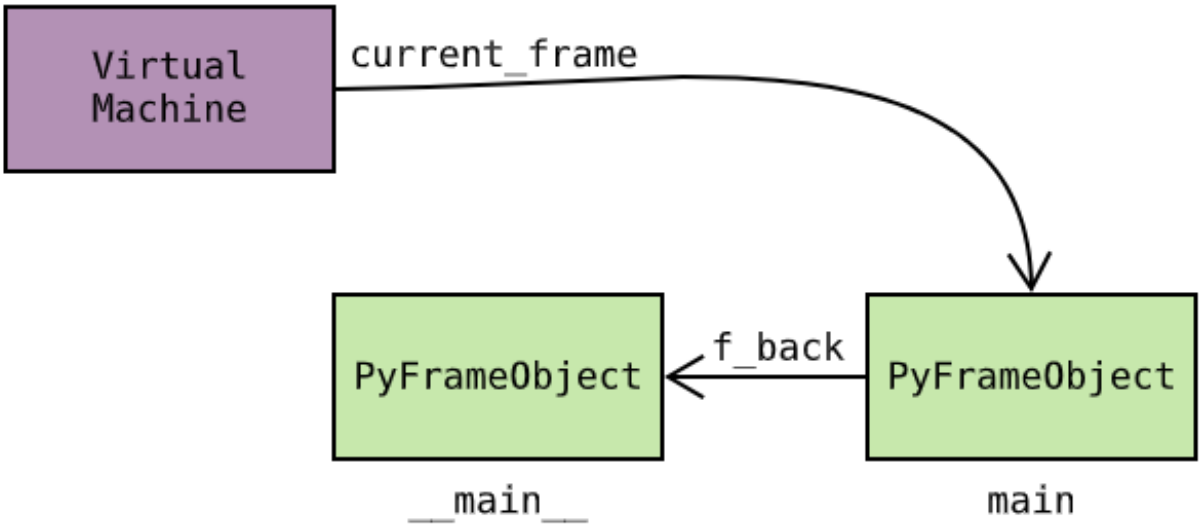
现在，我们回过头来继续考察 *Python* 栈帧对象链以及函数调用之前的关系。请看下面这个例子（ *demo.py* ）：

```
pi = 3.14

def square(r):
    return r ** 2

def circle_area(r):
    return pi * square(r)

def main():
    print(circle_area(5))

if __name__ == '__main__':
    main()
```

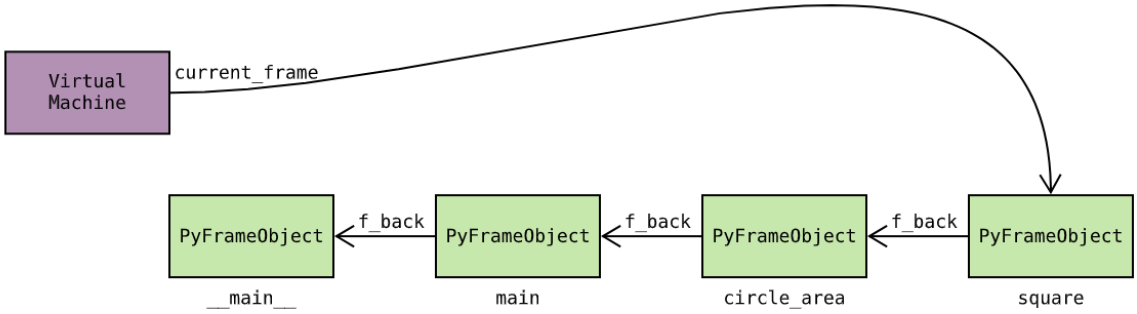
当 *Python* 开始执行这个程序时，虚拟机先创建一个栈帧对象，用于执行模块代码对象：



当虚拟机执行到模块代码第 *13* 行时，发生了函数调用。这时，虚拟机新建一个栈帧对，并开始执行函数 *main* 的代码对象：



随着函数调用逐层深入，当调用 *square* 函数时，调用链达到最长：



当函数调用完毕后，虚拟机通过 *f_back* 字段找到前一个栈帧对象并回到调用者代码中继续执行。

栈帧获取

栈帧对象 *PyFrameObject* 中保存着 *Python* 运行时信息，在底层执行流控制以及程序调试中非常有用。那么，在 *Python* 代码层面，有没有办法获得栈帧对象呢？答案是肯定的。调用标准库 *sys* 模块中的 *_getframe* 函数，即可获得当前栈帧对象：

```
>>> import sys
>>> frame = sys._getframe()
>>> frame
<frame at 0x10e3706a8, file '<stdin>', line 1, code <module>>
>>> dir(frame)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'f_back',
'f_builtins', 'f_code', 'f_globals', 'f_lasti', 'f_lineno', 'f_locals', 'f_trace', 'f_trace_lines', 'f_trace_opcodes']
```

获取栈帧对象后，有什么作用呢？举个例子，我们可以顺着 *f_back* 字段将调用关系和相关运行时信息打印出来：

```
import sys

pi = 3.14

def square(r):
    frame = sys._getframe()
    while frame:
        print('#', frame.f_code.co_name)
        print('Locals:', list(frame.f_locals.keys()))
        print('Globals:', list(frame.f_globals.keys()))
        print()

        frame = frame.f_back

    return r ** 2

def circle_area(r):
    return pi * square(r)

def main():
    print(circle_area(5))

if __name__ == '__main__':
    main()
```

例子程序在 *square* 函数中获取栈帧对象，然后逐层输出函数名以及对应的局部名字空间以及全局名字空间。程序执行后，你将看到这样的输出：

```
Locals: ['r', 'frame']
Globals: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__annotations__', '__builtins__',
'__file__', '__cached__', 'sys', 'pi', 'square', 'circle_area', 'main']


Locals: ['r']
Globals: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__annotations__', '__builtins__',
'__file__', '__cached__', 'sys', 'pi', 'square', 'circle_area', 'main']


Locals: []
Globals: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__annotations__', '__builtins__',
'__file__', '__cached__', 'sys', 'pi', 'square', 'circle_area', 'main']


Locals: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__annotations__', '__builtins__',
'__file__', '__cached__', 'sys', 'pi', 'square', 'circle_area', 'main']
Globals: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__annotations__', '__builtins__',
'__file__', '__cached__', 'sys', 'pi', 'square', 'circle_area', 'main']

78.5
```

栈帧获取面试题

如果面试官问你，能不能写个函数实现 *sys._getframe* 一样的功能，你能应付吗？

我们知道，*Python* 程序抛异常时，会将执行上下文带出来，保存在异常中：

```
>>> try:
...     1 / 0
... except Exception as e:
...     print(e.__traceback__.tb_frame)
...
<frame at 0x1079713f8, file '<stdin>', line 4, code <module>>
```

因此，我们自己的 *getframe* 函数可以这样来写：

```
def getframe():
    try:
        1 / 0
    except Exception as e:
        return e.__traceback__.tb_frame.f_back
```

请注意， *getframe* 中通过异常获得的是 *getframe* 自己的栈帧对象，必须通过 *f_back* 字段找到调用者的栈帧。

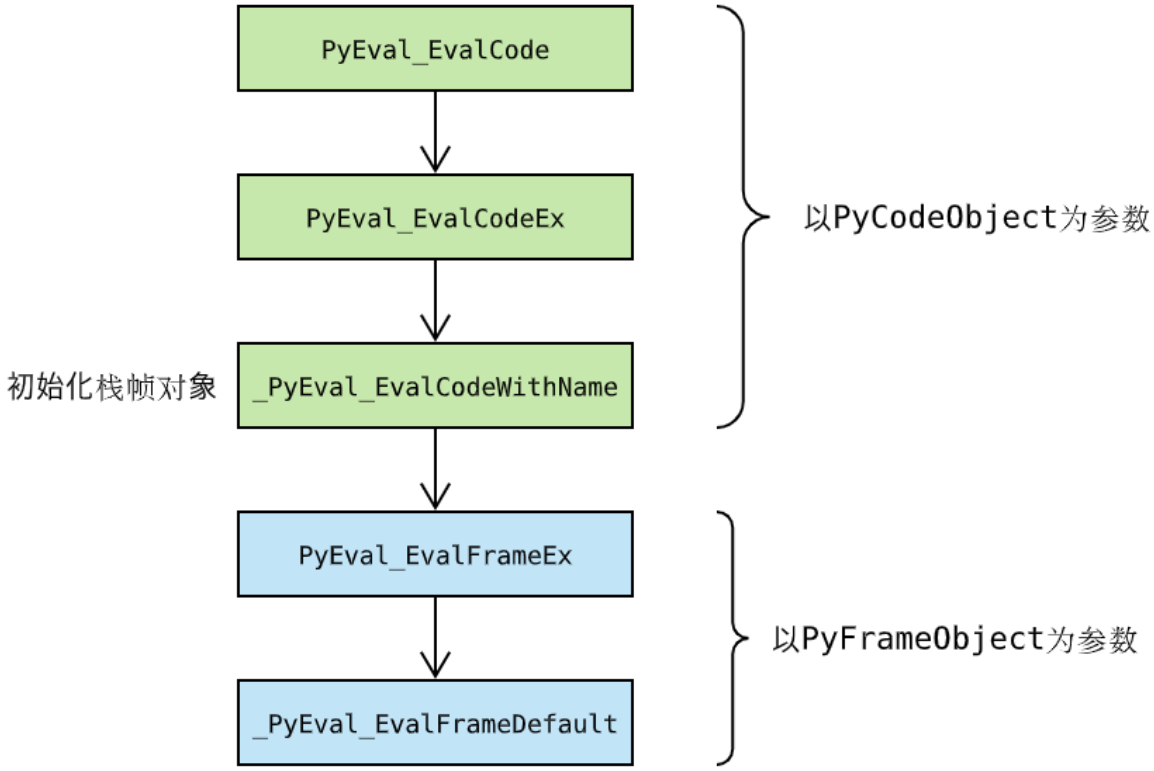
字节码执行

Python 虚拟机执行代码对象的代码位于 *Python/ceval.c* 中，主要函数有两个：*PyEval_EvalCodeEx* 是通用接口，一般用于函数这样带参数的执行场景； PyEval_EvalCode 是更高层封装，用于模块等无参数的执行场景。

```
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);
```

这两个函数最终调用 *_PyEval_EvalCodeWithName* 函数，初始化栈帧对象并调用 *PyEval_EvalFrame* 系列函数进行处理。栈帧对象将贯穿代码对象执行的始终，负责维护执行时所需的一切上下文信息。而 *PyEval_EvalFrame* 系列函数最终调用 *_PyEval_EvalFrameDefault* 函数，虚拟机执行的秘密就藏在这！

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

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



虽然 *Python/ceval.c* 代码量很大(超过 *5000* 行)，但主体结构并不复杂，很好理解。由于篇幅关系，我们没有办法对 *_PyEval_EvalFrameDefault* 函数展开深入介绍，只能用伪代码描述该函数的处理结构：

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

   for (;;) {


      opcode, oparg = read_next_byte_code(f);
      if (opcode == NULL) {
         break;
      }

      switch (opcode) {

         case LOAD_CONST:

            break;

         case LOAD_NAME:

            break;

      }
   }
}
```

- **函数主体是一个 *for* 循环，逐条读入字节码并执行；**

- *for* 循环体是一个很大的 *switch* ，判断字节码指令类型并分别处理；

最后，我们以几个典型的字节码例子，演示虚拟机执行的过程。

顺序执行

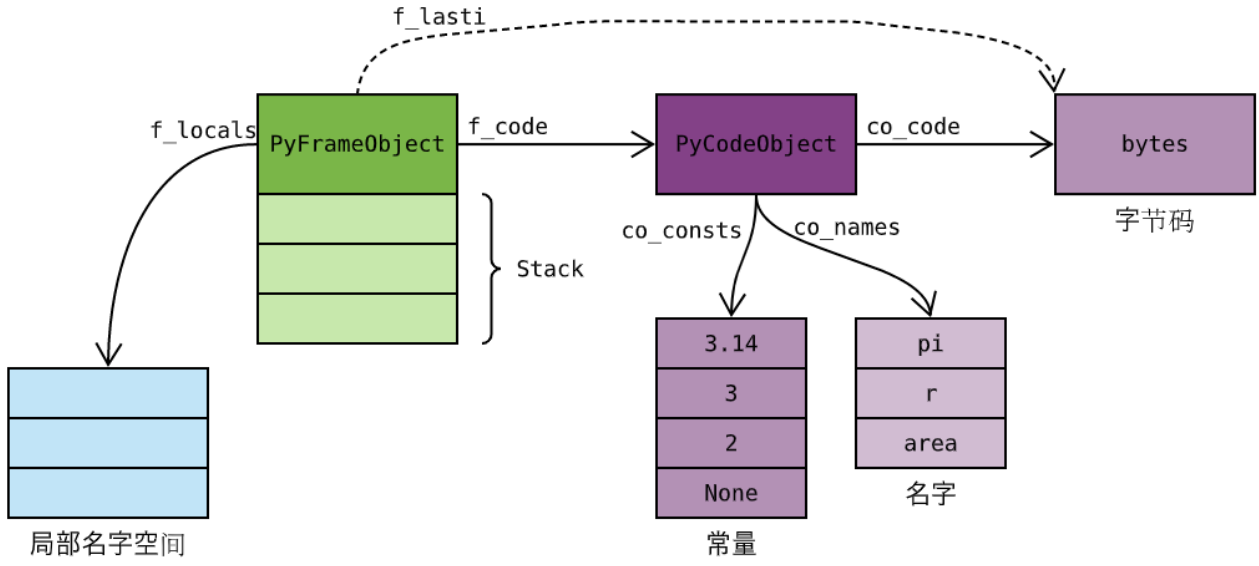
我们以顺序执行的字节码开始，这类字节码不包含任何跳转指令，逐条执行。下面是一个例子：

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

对这几行代码进行编译，我们可以得到这样的字节码：

```
  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
```

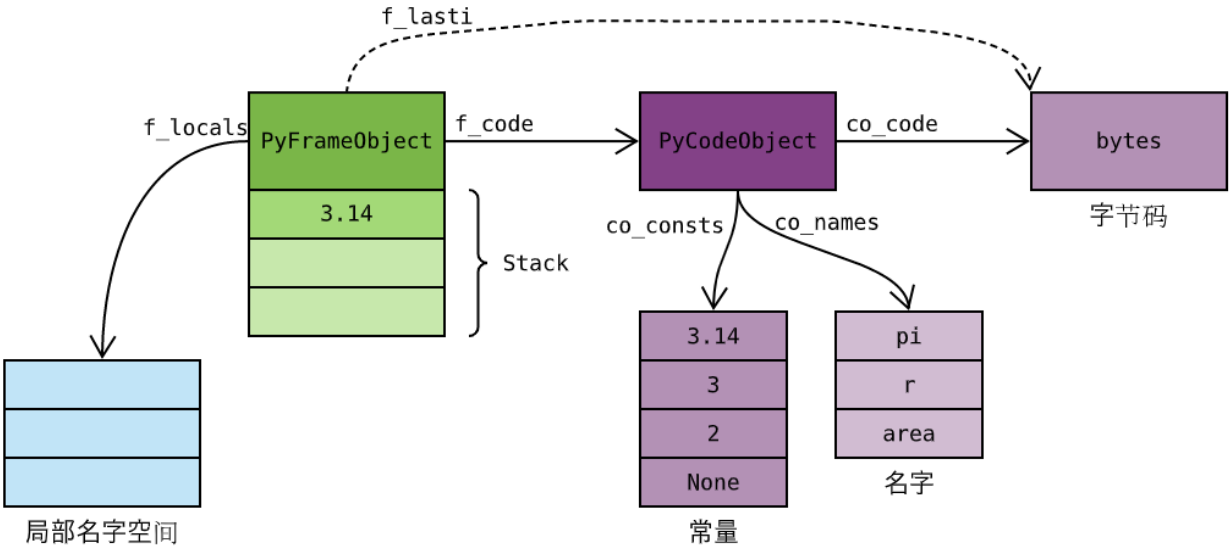
Python 虚拟机刚开始执行时，准备好栈帧对象用于保存执行上下文，关系如下(全局名字空间等信息省略)：



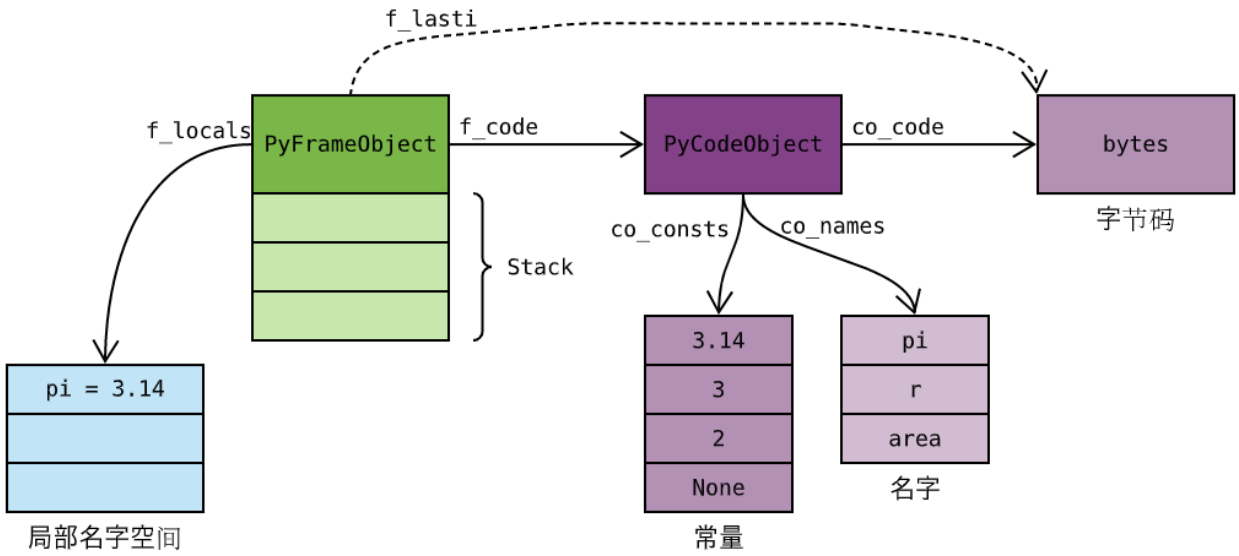
由于 *next_instr* 初始状态指向字节码开头，虚拟机开始加载第一条字节码指令：`LOAD_CONST 0`。字节码分为两部分，分别是 **操作码**（*opcode*）和 **操作数**（*oparg*）。`LOAD_CONST` 指令表示将常量加载进临时栈，常量下标由操作数给出。`LOAD_CONST` 指令在 *_PyEval_EvalFrameDefault* 函数 *switch* 结构的一个 *case* 分支中实现：

```
TARGET(LOAD_CONST) {
    PyObject *value = GETITEM(consts, oparg);
    Py_INCREF(value);
    PUSH(value);
    FAST_DISPATCH();
}
```

虚拟机执行这个指令时，先以操作数为下标从常量表中找到待加载常量并压入位于栈帧对象尾部的临时栈：

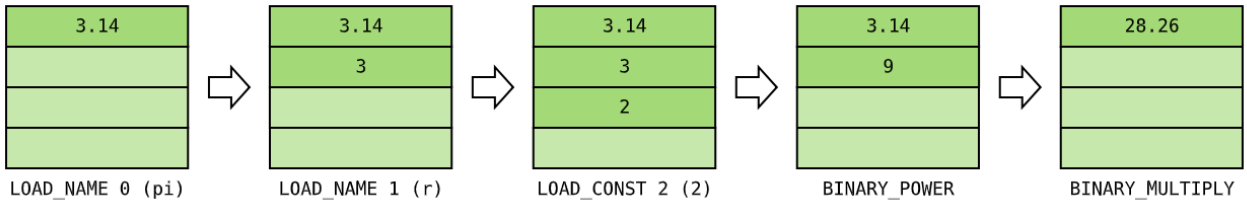


接着虚拟机接着执行 `STORE_NAME 0` 指令，将栈顶元素弹出并保存到局部名字空间，名字下标由操作数给出：



你可能会问，变量赋值为啥不直接存取名字空间，而是到临时栈绕一圈？主要原因在于：*Python* 字节码只有一个操作数，另一个操作数只能通过临时栈给出。 *Python* 字节码设计思想跟 *CPU* **精简指令集** 类似，指令尽量简化，复杂操作由多条指令组合完成。紧接着的两条字节码指令也是类似的，不再赘述。

接下的指令就不一条条展开介绍了，我们重点关注每条指令执行完毕后，临时栈的变化：



其中， `BINARY_POWER` 指令从栈上弹出两个操作数(底数 *3* 和 指数 *2*)进行 **幂运算** ，并将结果 *9* 压回栈中； `BINARY_MULTIPLY` 指令则进行 **乘积运算** ，步骤也是类似的。

if 判断

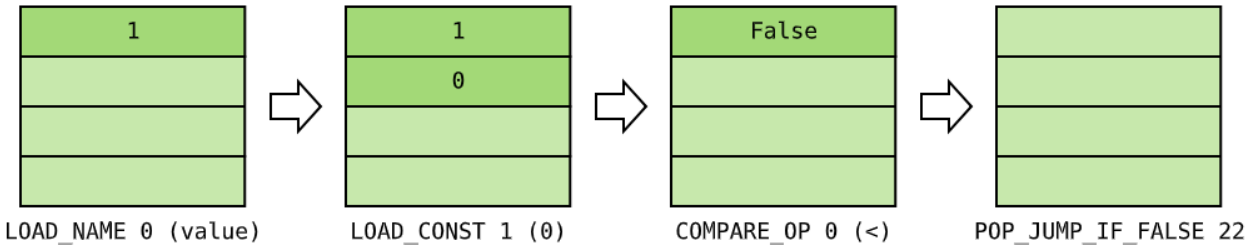
接下来，我们继续研究 *Python* 控制流字节码，先看看最简单的 *if* 判断语句：

```
value = 1
if value < 0:
    print('negative')
else:
    print('positive')
```

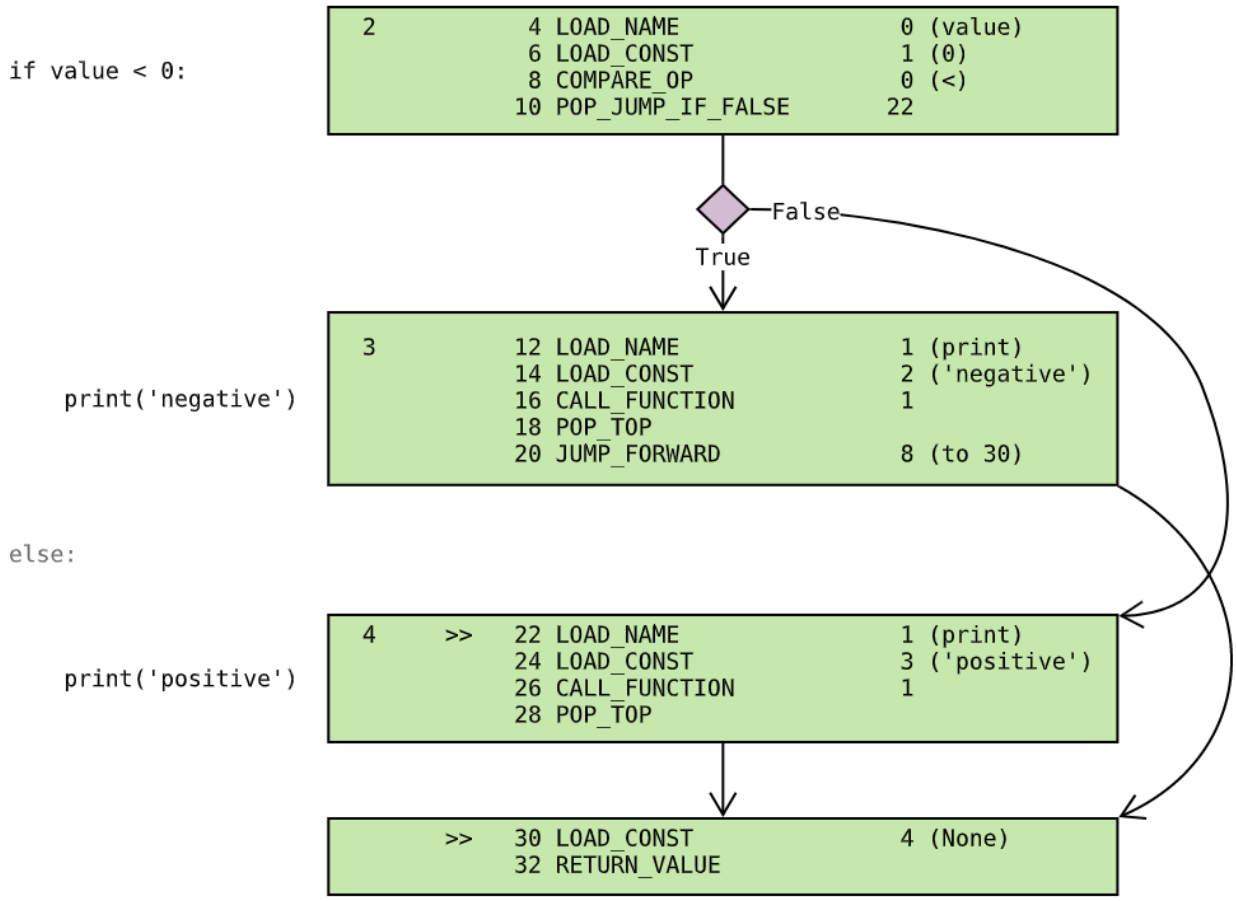
对这几行代码进行编译，我们可以得到这样的字节码：

```
 1           0 LOAD_CONST               0 (1)
             2 STORE_NAME               0 (value)

 2           4 LOAD_NAME                0 (value)
             6 LOAD_CONST               1 (0)
             8 COMPARE_OP               0 (<)
            10 POP_JUMP_IF_FALSE       22

 3          12 LOAD_NAME                1 (print)
            14 LOAD_CONST               2 ('negative')
            16 CALL_FUNCTION            1
            18 POP_TOP
            20 JUMP_FORWARD             8 (to 30)

 4     >>   22 LOAD_NAME                1 (print)
            24 LOAD_CONST               3 ('positive')
            26 CALL_FUNCTION            1
            28 POP_TOP
        >>  30 LOAD_CONST               4 (None)
            32 RETURN_VALUE
```

当执行到代码第 *2* 行 *if* 语句对应的字节码时，临时栈的变化如下：



`POP_JUMP_IF_FALSE` 指令将比较结果从栈顶弹出并判断真假，如果为假则跳到 *else* 分支对应的字节码执行；如果为真则继续执行，最终由 `JUMP_FORWARD` 指令完成跳转绕过 *else* 分支(跳过 *8* 个字节也就是 *4* 条字节码)。



while 循环

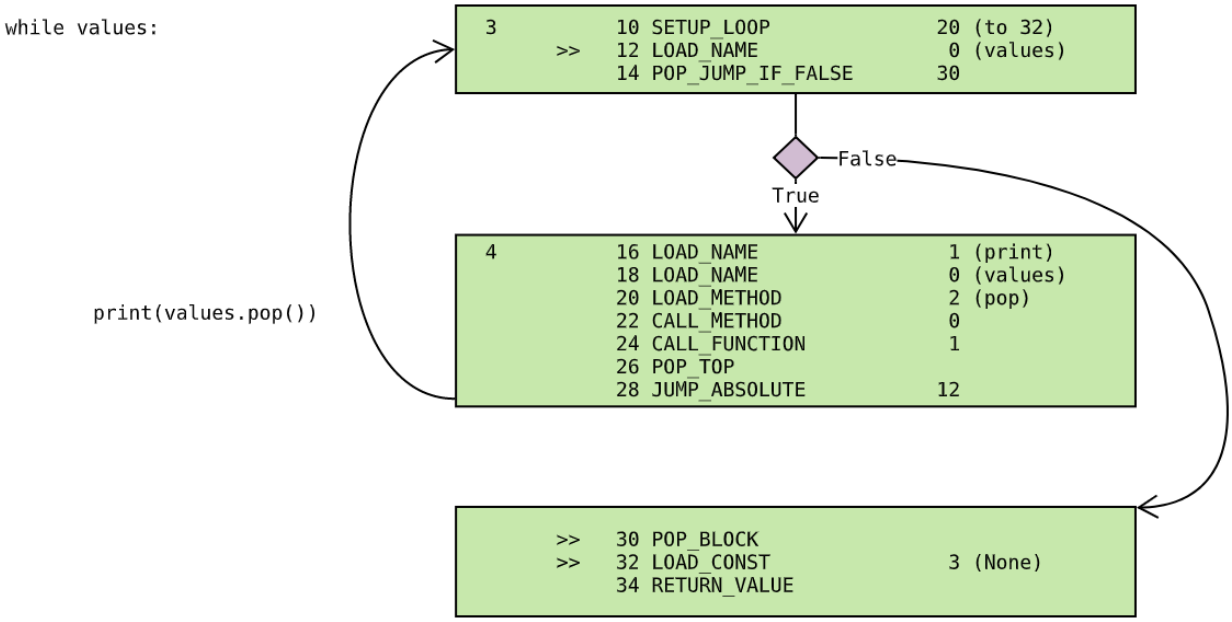
最后，我们快速过一遍循环控制结构，以 *while* 循环为例：

```
values = [1, 2, 3]

while values:
    print(values.pop())
```

对这几行代码进行编译，我们可以得到这样的字节码：

```
1           0 LOAD_CONST               0 (1)
            2 LOAD_CONST               1 (2)
            4 LOAD_CONST               2 (3)
            6 BUILD_LIST               3
            8 STORE_NAME               0 (values)

3           10 SETUP_LOOP              20 (to 32)
      >>    12 LOAD_NAME                0 (values)
            14 POP_JUMP_IF_FALSE       30

4           16 LOAD_NAME               1 (print)
            18 LOAD_NAME               0 (values)
            20 LOAD_METHOD             2 (pop)
            22 CALL_METHOD             0
            24 CALL_FUNCTION           1
            26 POP_TOP
            28 JUMP_ABSOLUTE           12
      >>    30 POP_BLOCK
      >>    32 LOAD_CONST               3 (None)
            34 RETURN_VALUE
```

经过前面学习，我们轻易便可以读懂这些字节码，并画出以下流程图：



1. `SETUP_LOOP` 指令拉开循环执行的序幕；
2. `POP_JUMP_IF_FALSE` 指令判断循环条件，成立则往下执行循环体，否则跳过循环体结束循环；
3. `JUMP_ABSOLUTE` 在循环体执行完毕后，跳回循环开头处，再次检查循环条件；
4. `POP_BLOCK` 指令在循环结束后，清理循环环境；

小结

本节我们深入 *Python* 虚拟机源码，研究虚拟机执行字节码的全过程。虚拟机在执行代码对象前，需要先创建 **栈帧对象** (*PyFrameObject*)，用于维护运行时的上下文信息。 *PyFrameObject* 关键信息包括：

- 局部名字空间(*f_locals*)；
- 全局名字空间(*f_globals*)；
- 内建名字空间(*f_builtins*)；
- 代码对象(*f_code*)；
- 上条已执行指令编号(*f_lasti*)；
- 调用者栈帧(*f_back*)；
- 静态局部名字空间与临时栈(*f_localsplus*)；

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

```
import sys
frame = sys._getframe()
```

最后，我们研究了典型字节码的执行过程，发现这个过程跟 *CPU* 执行机器指令的过程非常类似。**字节码就像是汇编语言，而虚拟机就像一颗软件 CPU** ！由此可见计算机知识间关系紧密，没有 **计算机组成原理** 、 **操作系统** 、 **计算机网络** 等基础知识加持，程序开发就像在沙子上盖大厦。

Python 虚拟机的代码量不小，而本节力求以最通俗的语言将虚拟机的原理讲清楚，因此没有深入源码的细枝末节。鼓励读者保持好奇心，深入到源码中，肯定会有所收获。修炼内功，从源码开始！