《源码探秘 CPython》46. 虚拟机的执行环境：栈帧对象

原创 古明地觉 古明地觉的编程教室 2022-03-10 08:30

美食的终极意义在于获得幸福感

虚拟机的执行环境

接下来我们就来剖析虚拟机运行字节码的原理，我们知道Python虚拟机是Python的核心，在源代码被编译成PyCodeObject对象之后，就由虚拟机接手整个工作。Python虚拟机会从PyCodeObject中读取字节码，并在当前的上下文中执行，直到所有的字节码都被执行完毕。

那么问题来了，既然源代码在经过编译之后，所有字节码指令以及其他静态信息都存储在PyCodeObject当中，那么是不是意味着Python虚拟机就在PyCodeObject对象上进行所有的动作呢？

其实不能给出唯一的答案，因为尽管PyCodeObject包含了关键的字节码指令以及静态信息，但有一个东西是没有包含、也不可能包含的，就是程序在运行时的**执行环境**，这个执行环境在Python里面就是**栈帧**。

```
var = "satori"

def f():
    var = 666
    print(var)

f()
print(var)
```

上面的代码当中出现了两个**print(var)**，它们的字节码指令是相同的，但是执行的效果却显然是不同的，这样的结果正是执行环境的不同所产生的。因为环境的不同，var的值也不同。

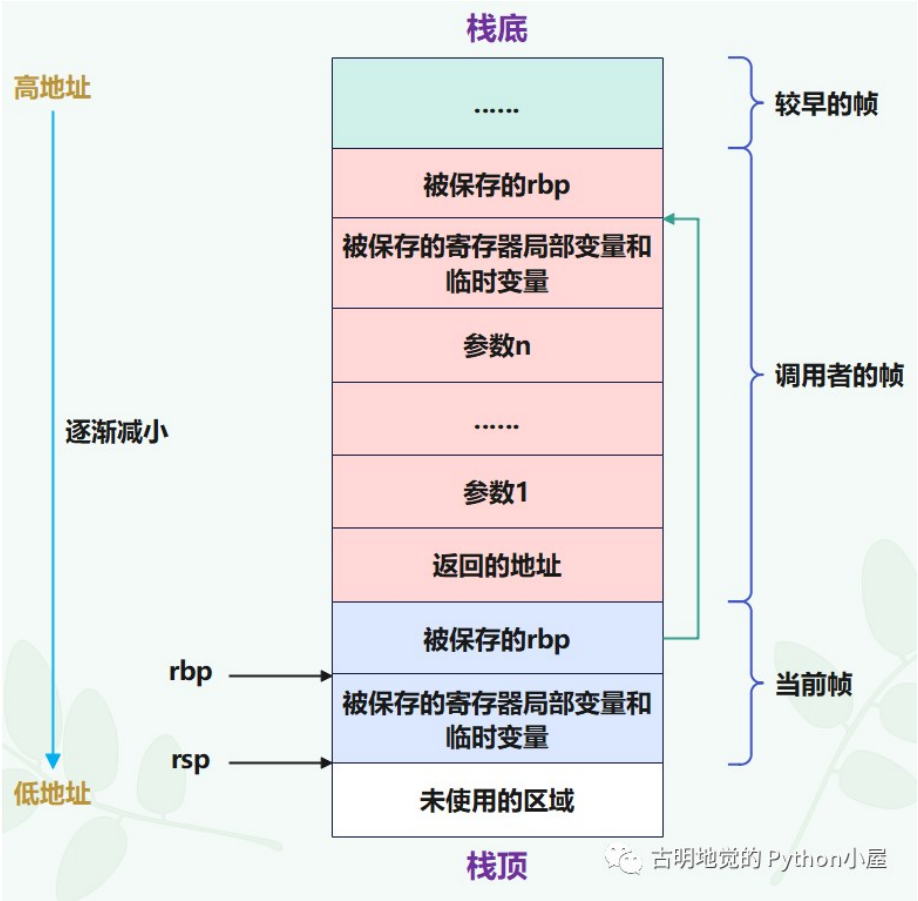
因此同一个符号在不同环境中可能指向不同的类型、不同的值，必须在运行时进行动态地捕捉和维护，这些信息不可能在PyCodeObject对象中被静态存储。

因此虚拟机并不是在PyCodeObject对象上执行操作的，而是栈帧对象。虚拟机在执行时，会根据PyCodeObject对象动态创建出栈帧对象，然后在栈帧里面执行字节码。

因此对于上面的代码，我们可以大致描述一下流程：

- 当虚拟机在执行第一条语句时，已经创建了一个栈帧，这个栈帧显然是模块对应的栈帧，假设叫做A；
- 所有的字节码都会在这个栈帧中执行，虚拟机可以从栈帧里面获取变量的值，也可以修改；
- 当发生函数调用的时候，这里是函数f，那么虚拟机会在栈帧A之上，为函数f创建一个新的栈帧，假设叫B，然后在栈帧B里面执行函数f的字节码指令；
- 在栈帧B里面也有一个名字为var的对象，但由于执行环境、或者说栈帧的不同，var也不同。比如两个人都叫小明，但一个是北京的、一个是上海的，所以这两者没什么关系；
- 一旦函数f的字节码指令全部执行完毕，那么会将当前的栈帧B销毁(也可以保留下来)，再回到调用者的栈帧中来。就像是递归一样，每当调用函数时，就会在当前栈帧之上创建一个新的栈帧，一层一层创建，一层一层返回；

而实际上Python虚拟机执行字节码这个过程，就是在模拟操作系统运行可执行文件。我们来看看在一台普通的x64的机器上，可执行文件是以什么方式运行的。在这里主要关注运行时栈的栈帧，如图所示：



x64体系处理器通过栈维护调用关系，每次函数调用时就在栈上分配一个帧用于保存**调用上下文**以及**临时存储**。CPU有两个关键寄存器，rsp指向当前栈顶，rbp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中执行代码；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rsp设为当前栈顶(保存新栈帧的位置)。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。

我们用一段Python代码解释一下：

```
def f(a, b):
    return a + b

```

```
def g():
    return f()

g()
```

当程序执行到**函数f**时，那么显然调用者的帧就是**函数g**的栈帧，而当前帧则是**函数f**的栈帧。

> 解释一下：栈是先入后出的数据结构，从栈底到栈顶地址是减小的。对于一个函数而言，所有对局部变量的操作都在自己的栈帧中完成，而调用函数的时候则会为其创建新的栈帧。

在上图中，我们看到运行时栈的地址是从高地址向低地址延伸的。当在**函数g**里面调用**函数f**的时候，系统就会在地址空间中，于g的栈帧之上创建f的栈帧。

当然在函数调用的时候，系统会保存上一个栈帧的**栈指针(rsp)**和**帧指针(rbp)**。当函数的调用完成时，系统又会把rsp和rbp的值恢复为创建**f栈帧**之前的值，这样程序的流程就又回到了**函数g**中。当然，程序的运行空间则也回到了函数g的栈帧中，这就是可执行文件在x64机器上的运行原理。

那么下面我们就来看看栈帧在底层长什么样，注意：栈帧也是一个对象。

栈帧的底层结构

栈帧在底层是由PyFrameObject结构体表示的，但Python的栈帧可不仅仅只是类似于x64机器上看到的那个简简单单的栈帧，Python的栈帧实际上包含了更多的信息。

```
typedef struct _frame {
    //可变对象的头部信息
    PyObject_VAR_HEAD
    //上一级栈帧，也就是调用者的栈帧
    struct _frame *f_back;
    //PyCodeObject对象
    //通过栈帧的f_code属性可以获取对应的PyCodeObject对象
    PyCodeObject *f_code;
    //builtin名字空间，一个PyDictObject对象
    PyObject *f_builtins;
    //global名字空间，一个PyDictObject对象
    PyObject *f_globals;
    //local名字空间，一个PyDictObject对象
    PyObject *f_locals;
    //运行时的栈底位置
    PyObject **f_valuestack;
    //运行时的栈顶位置
    PyObject **f_stacktop;
    //回溯函数，打印异常栈
    PyObject *f_trace;
    //是否触发每一行的回溯事件
    char f_trace_lines;
    //是否触发每一个操作码的回溯事件
    char f_trace_opcodes;
    //是否是基于生成器的PyCodeObject构建的栈帧
    PyObject *f_gen;
    //上一条已执行完毕的指令在f_code中的偏移量
    int f_lasti;
    //当前字节码对应的源代码行号
    int f_lineno;
    //当前指令在栈f_blockstack中的索引
    int f_iblock;
    //当前栈帧是否仍在执行
    char f_executing;
```

```
    //用于try和loop代码块
    PyTryBlock f_blockstack[CO_MAXBLOCKS];
    //动态内存
    //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
    PyObject *f_localsplus[1];
} PyFrameObject;
```

因此我们看到，虚拟机会根据PyCodeObject对象来创建一个栈帧，也就是PyFrameObject对象，虚拟机实际上是在PyFrameObject对象上执行操作的。

每一个PyFrameObject都会维护一个PyCodeObject，换句话说，每一个PyCodeObject都会隶属于一个PyFrameObject。并且从f_back可以看出，Python在实际执行时，会产生很多的PyFrameObject对象，而这些对象会被链接起来，形成一条执行环境链表，或者说栈帧链表。

而这正是x64机器上栈帧之间关系的模拟，在x64机器上，栈帧之间通过rsp和rbp指针建立了联系，使得新栈帧在结束之后能够顺利的返回到旧栈帧中，而Python则是利用f_back来完成这个动作。

图片

栈帧里面的f_code指向相应的PyCodeObject对象，而f_builtins、f_globals、f_locals则是指向三个独立的名字空间。在这里，我们看到了**名字空间**和**执行环境(即栈帧)**之间的关系，前者只是后者的一部分。

名字空间负责维护变量和对象，通过名字空间，我们能够找到一个符号被绑定在了哪个对象上。

另外在PyFrameObject的开头有一个PyObject_VAR_HEAD，表示栈帧是一个变长对象，即每一次创建的栈帧的大小可能是不一样的，那么这个变动在什么地方呢？

首先每一个PyFrameObject对象都维护了一个PyCodeObject对象，而每一个PyCodeObject对象都会对应一个代码块(code block)。在编译一段代码块的时候，会计算这段代码块执行时所需要的栈空间的大小，这个栈空间大小存储在PyCodeObject对象的co_stacksize中。

而不同的代码块所需要的栈空间是不同的，因此栈帧是一个变长对象。最后，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**。

> 注意：x64机器上的运行时栈不止包含执行所需要的内存空间，还有别的。但PyFrameObject对象的运行时栈则只包含执行所需要的内存空间，这一点务必注意。

在Python中访问PyFrameObject

在Python里面拿到栈帧对象，可以通过inspect模块。

```
import inspect

def f():
    # 返回当前所在的栈帧
    # 这个函数实际上是调用了 sys._getframe(1)
    return inspect.currentframe()

frame = f()
print(frame)  # <frame at ..., file 'D:/satori/main.py', line 6, code f>
print(type(frame))  # <class 'frame'>
```

我们看到栈帧的类型是**<class 'frame'>**，正如PyCodeObject对象的类型是**<class 'code'>**一样。这两个类没有暴露给我们，所以不可以直接使用。

同理，还有Python的函数，类型是**<class 'function'>**；模块，类型是**<class 'module'>**。这些解释器都没有给我们提供，如果直接使用的话，那么frame、code、function、module只是几个没有定义的变量罢了，这些类我们只能通过这种间接的方式获取。

下面我们就来获取一下栈帧的成员属性。

```
import inspect

def f():
    name = "古明地觉"
    age = 16
    return inspect.currentframe()

def g():
    name = "魔理沙"
    age = 333
    return f()

# 当我们调用函数g的时候，也会触发函数f的调用
# 而一旦f执行完毕，那么f对应的栈帧就被全局变量frame保存起来了
frame = g()

print(frame)  # <frame at ... 'D:/satori/main.py', line 6, code f>
# 获取上一级栈帧，即调用者的栈帧
# 显然是g的栈帧
print(frame.f_back)  # <frame at ... 'D:/satori/main.py', line 11, code
g>
# 模块也是有栈帧的，我们后面会单独说
print(frame.f_back.f_back)  # <frame at ... 'D:/satori/main.py', line 25
, code <module>>
# 显然最外层就是模块了
# 模块对应的上一级栈帧是None
print(frame.f_back.f_back.f_back)  # None

# 获取PyCodeObject对象
print(frame.f_code)  # <code object f ... "D:/satori/main.py", line 4>
print(frame.f_code.co_name)  # f

# 获取f_locals
# 即栈帧内部的local名字空间
print(frame.f_locals)  # {'name': '古明地觉', 'age': 16}
print(frame.f_back.f_locals)  # {'name': '魔理沙', 'age': 333}

# 获取栈帧对应的行号
print(frame.f_lineno)  # 6
```

```
print(frame.f_back.f_lineno)  # 11
"""
行号为6的位置是: return inspect.currentframe()
行号为11的位置是: return f()
"""
```

我们看到函数运行完毕之后，里面的局部变量居然还能获取，原因就是栈帧没被销毁，因为它被返回了，而且被外部变量接收了。同理：该栈帧的上一级栈帧也不能被销毁，因为当前栈帧的f_back指向它了，引用计数不为0，所以要保留。

通过栈帧我们可以获取很多的属性，我们后面还会慢慢说。此外，异常处理也可以获取到栈帧。

```
def foo():
    try:
        1 / 0
    except ZeroDivisionError:
        import sys
      # exc_info返回一个三元组
      # 分别是异常的类型、值、以及traceback
        exc_type, exc_value, exc_tb = sys.exc_info()
        print(exc_type)  # <class 'ZeroDivisionError'>
        print(exc_value)  # division by zero
        print(exc_tb)  # <traceback object at 0x00000135CEFDF6C0>

      # 调用exc_tb.tb_frame即可拿到异常对应的栈帧
      # 另外这个exc_tb也可以通过下面这种方式获取
      # except ZeroDivisionError as e; e.__traceback__
        print(exc_tb.tb_frame)  # <frame at ... 'D:/satori/main.py', lin
e 16, code foo>
        print(exc_tb.tb_frame.f_back)  # <frame at ... 'D:/satori/main.p
y', line 21, code <module>>
      # 显然tb_frame是当前函数foo的栈帧
      # 那么tb_frame.f_back就是整个模块对应的栈帧
      # 那么再上一级的话，栈帧就是None了
        print(exc_tb.tb_frame.f_back.f_back)  # None

foo()
```

通过以上方式就可以在Python中获取栈帧对象了。

小结

因为很多动态信息无法静态地存储在PyCodeObject对象中，所以PyCodeObject对象在交给虚拟机之后，虚拟机会在此之上动态地构建出PyFrameObject对象，也就是栈帧。

因此虚拟机是在栈帧里面执行的字节码，它包含了虚拟机在执行字节码时依赖的全部信息。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》47. 名字、作用域、名字空间（上）

下一篇 ›
《源码探秘 CPython》45. pyc文件是怎么创建的?

喜欢此内容的人还喜欢

Linux IO 相关的全面介绍

Linux码农

linux之nginx常见问题

日常的闲谈杂记

【职业】Linux下MySQL忘记密码时如何重置密码

哈罗沃编程