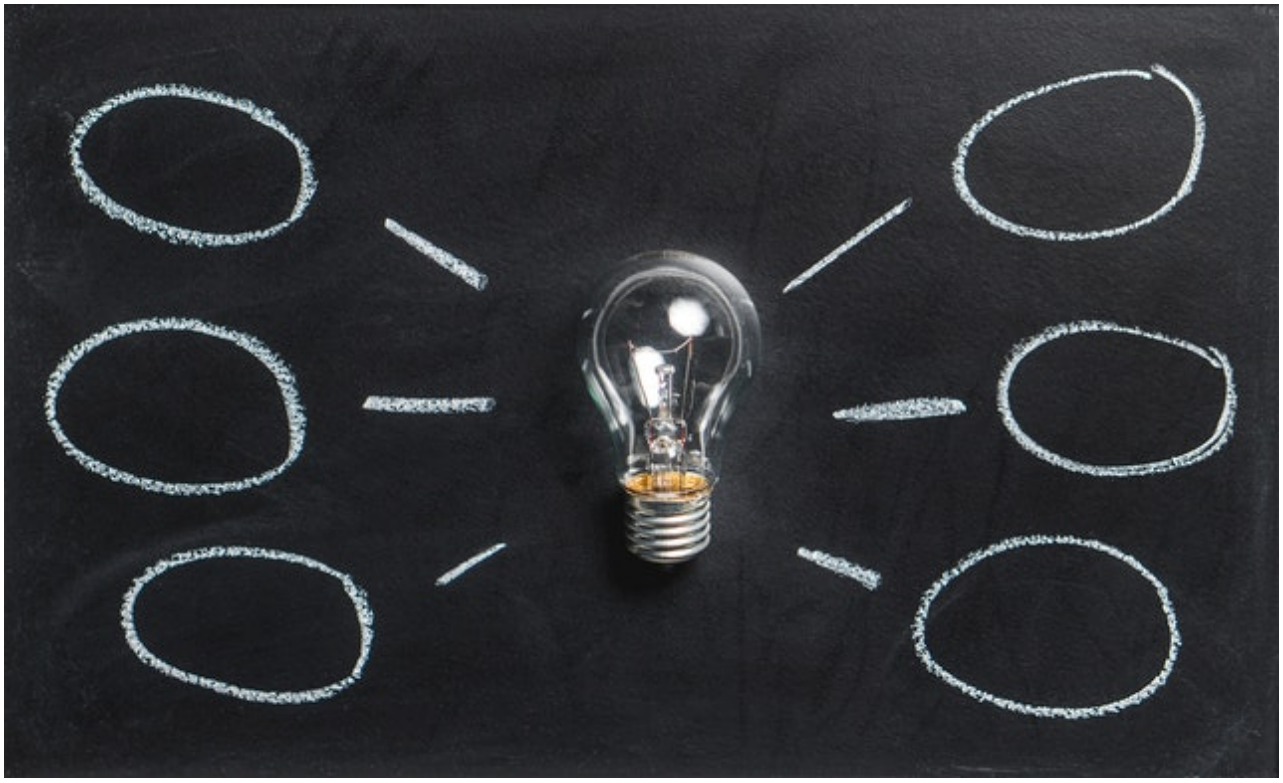
25 函数调用与虚拟机软件栈-慕课专栏

25 函数调用与虚拟机软件栈

更新时间：2020-07-23 15:51:57



与有肝胆人共事，从无字句处读书。
——周恩来

我们已经掌握了创建函数对象的秘密，并发挥自己的聪明才智以一种全新的方式创造了一个函数对象。虽然在实际项目中，我们不会这么做，但这种新尝试让我们可以更好地理解函数的行为。

现在，我们又对函数调用的秘密充满好奇。函数是怎么调用的？参数和返回值是如何调用的？递归又是如何实现的？带着这些问题，我们再次启程，研究 *circle_area* 这个我们既熟悉又陌生的函数。

我们将 circle_area 定义在 *geometry* 模块中，文件名为 *geometry.py* ：

```
pi = 3.14

def circle_area(r):
    return pi * r ** 2

def cylinder_volume(r, h):
    return circle_area(r) * h
```

注意到，模块中还有另一个函数 *cylinder_volume* 用于计算圆柱体体积，参数 *r* 是底面圆的半径，参数 *h* 是圆柱体高度。 *cylinder_volume* 先调用 *circle_area* 计算底面面积，再乘以高度得到圆柱体积。

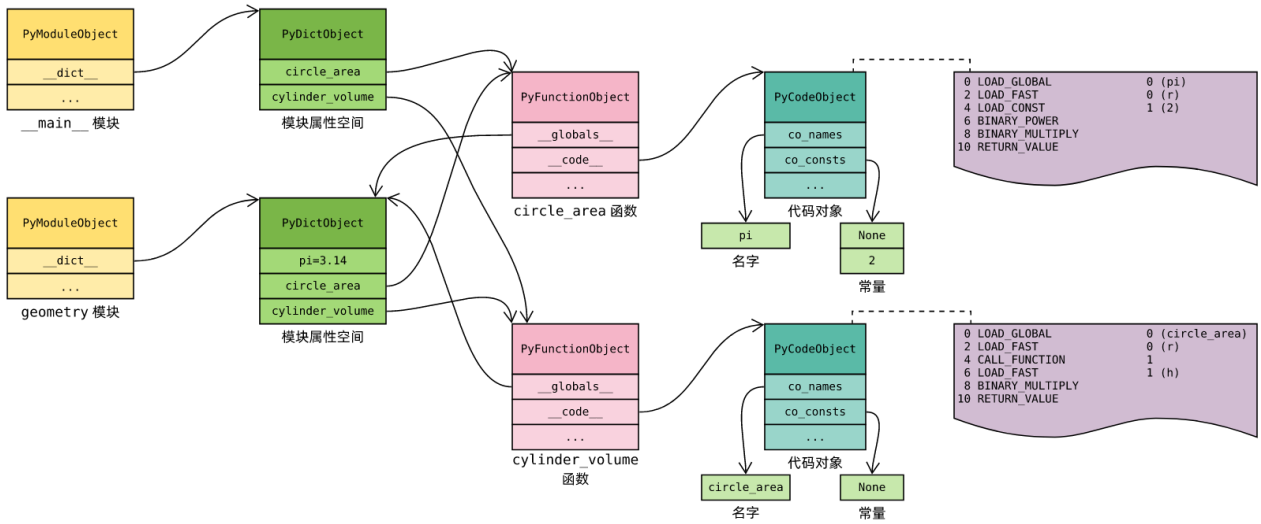
进入 *geometry.py* 所在目录，并启动 *Python* 终端，将 *geometry.py* 模块导入，即可调用相关函数：

```
>>> from geometry import circle_area, cylinder_volume
>>> circle_area(1.5)
7.065
```

如果你不想进入 *geometry.py* 所在目录，也可以将其路径加入到 *sys.path* ，这个方法我们在模块机制中介绍过：

```
>>> import sys
>>> sys.path.append('/some/path')
```

开始讨论函数调用流程之前，我们先来看看从 *geometry* 模块导入相关函数后虚拟机内部的状态：



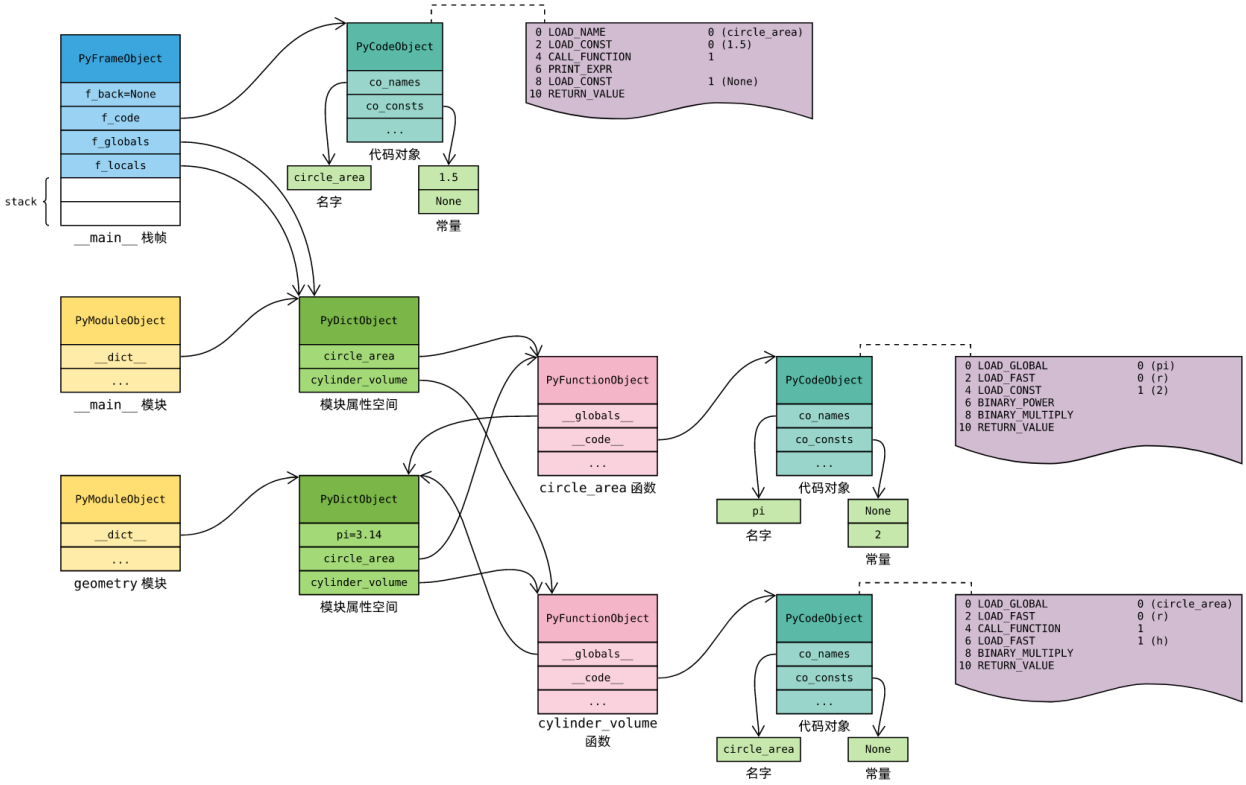
- *__main__* 模块是 *Python* 启动后的执行入口，每个 *Python* 程序均从 *__main__* 开始执行；
- *geometry* 是我们导入的模块，它有一个 *__dict__* 属性，指向模块属性空间；
- *geometry* 初始化后，属性空间里有一个浮点属性 *pi* 以及两个函数对象， *circle_area* 和 *cylinder_colume* ；
- 两个函数的 **全局名字空间** 与模块对象的 **属性空间** 是同一个 *dict* 对象；
- 两个函数都有一个 **代码对象** ，保存函数 **字节码** 以及 **名字** 、 **常量** 等静态上下文信息；
- 往下阅读前请务必理解该状态图，有疑问请复习虚拟机模块机制以及函数创建等章节，以加深理解；

每个 *Python* 程序都有一个 *__main__* 模块，以及与 *__main__* 模块对应的 **栈帧** 对象。*__main__* 模块是 *Python* 程序的入口，而与其对应的栈帧对象则是整个程序调用栈的起点。

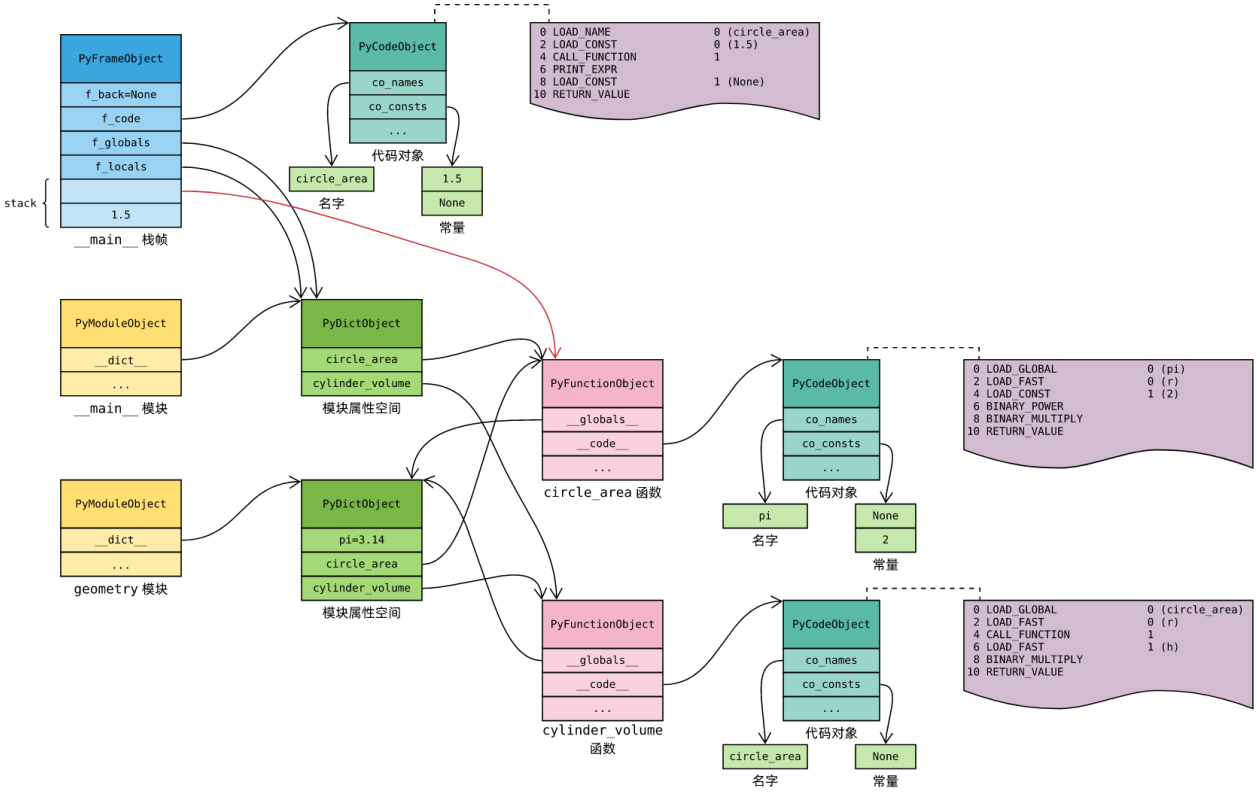
当我们在交互式终端输入语句时，也是类似的。 *Python* 先将代码编译成代码对象，再创建一个 **栈帧** 对象执行该代码对象。以 `circle_area(1.5)` 为例，编译可得到这样的字节码：

```
1           0 LOAD_NAME                0 (circle_area)
            2 LOAD_CONST               0 (1.5)
            4 CALL_FUNCTION            1
            6 PRINT_EXPR
            8 LOAD_CONST               1 (None)
           10 RETURN_VALUE
```

随后，*Python* 创建栈帧对象作为执行环境，准备执行编译后的代码对象：

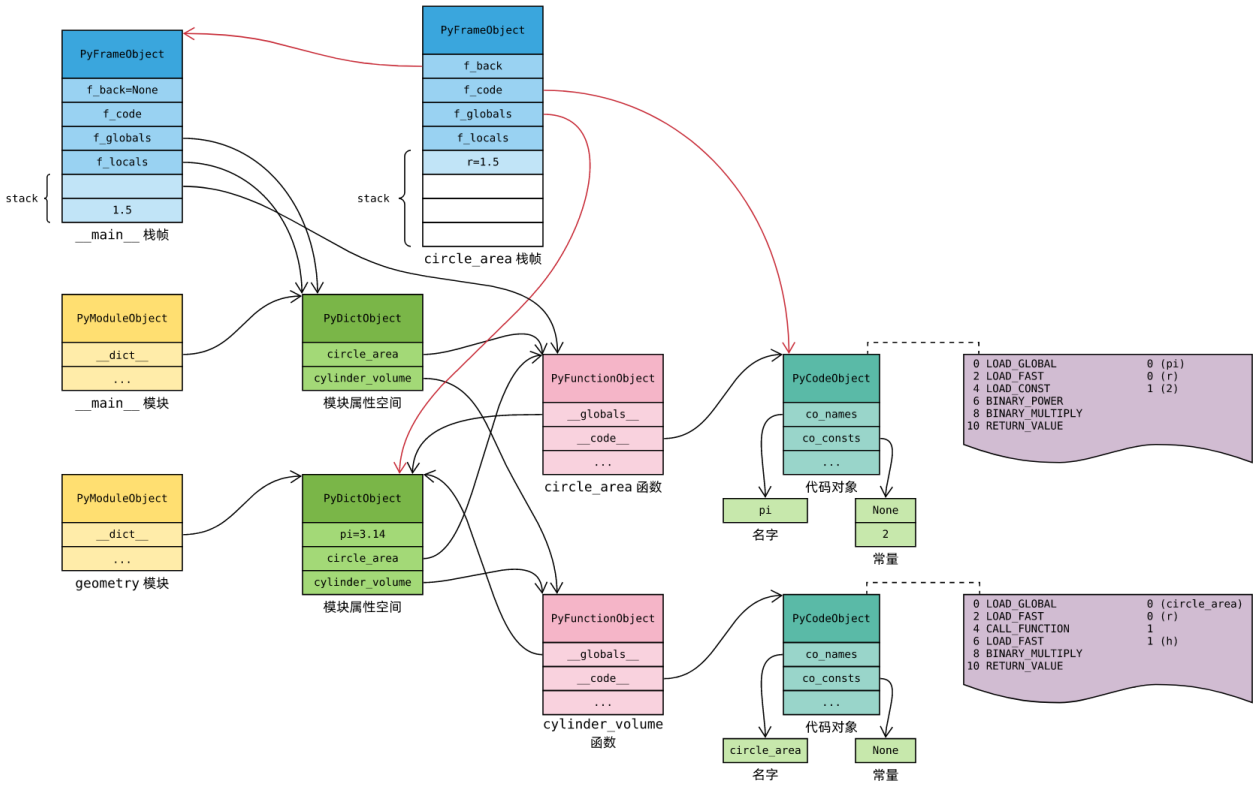


注意到，栈帧对象全局名字空间、局部名字空间均指向 *__main__* 模块的属性空间。`circle_area(1.5)` 的语句中，有些我们已经非常熟悉了。第一条字节码，将名为 *circle_area* 的对象，加载到栈顶，这是我们导入的函数。第二条字节码，将常量 *1.5* 加载到栈顶，这是准备传递给函数的变量。执行这两个字节码后，虚拟机状态变为：



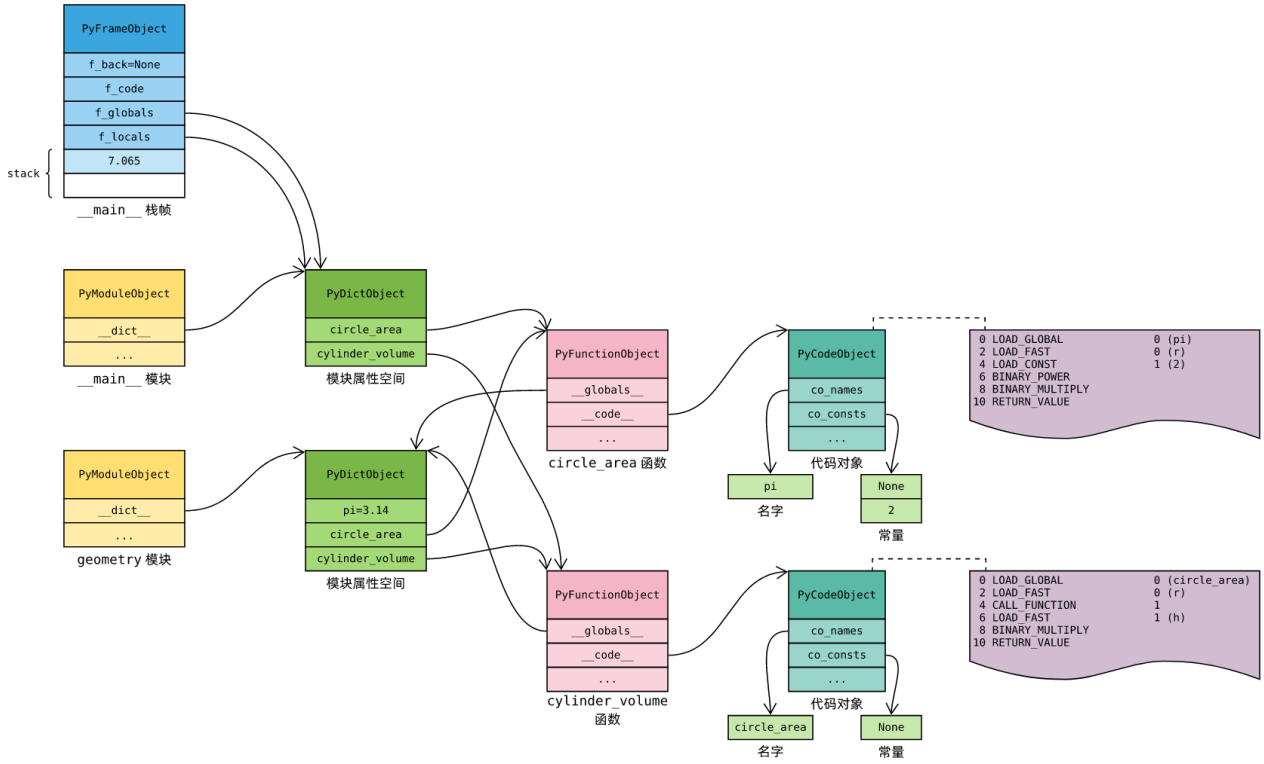
接着是 *CALL_FUNCTION* 字节码，顾名思义，我们知道正式它完成了调动函数的使命。*CALL_FUNCTION* 字节码的处理逻辑同样位于 Python/ceval.c 这个文件，有兴趣的童鞋可以阅读一下源码，这里用通俗的语言结合图示讲解这个字节码的作用。

CALL_FUNCTION 先创建一个新栈帧对象，作为 *circle_name* 函数的执行环境。新栈帧对象通过 *f_back* 指针，指向前一个栈帧对象，形成一个调用链。栈帧对象从函数对象取得 **代码** 对象，以及执行函数时的全局名字空间：



此外，注意到执行函数的栈帧对象 *f_locals* 字段为空，而不是跟 *f_globals* 一样执行一个 *dict* 对象。由于函数有多少局部变量是固定的，代码编译时就能确定。因此，没有必要用字典来实现局部名字空间，只需把局部变量依次编号，保存在栈底即可(*r=1.5* 处)。这样一来，通过编号即可快速存取局部变量，效率比字典更高。于此对应，有一个特殊的字节码 LOAD_FAST 用于加载局部变量，以操作数的编号为操作数。

circle_area 的字节码我们已经很熟悉了，便不再赘述了，请动手在栈帧上推演一番。最后，*RETURN_VALUE* 字节码将结算结果返回给调用者，执行权现在交回调用者的 *CALL_FUNCTION* 字节码。*CALL_FUNCTION* 先将结果保存到栈顶并着手回收 *circle_area* 函数的栈帧对象。



嵌套调用也是类似的，以 `cylinder_volume(1.5, 2)` 为例：

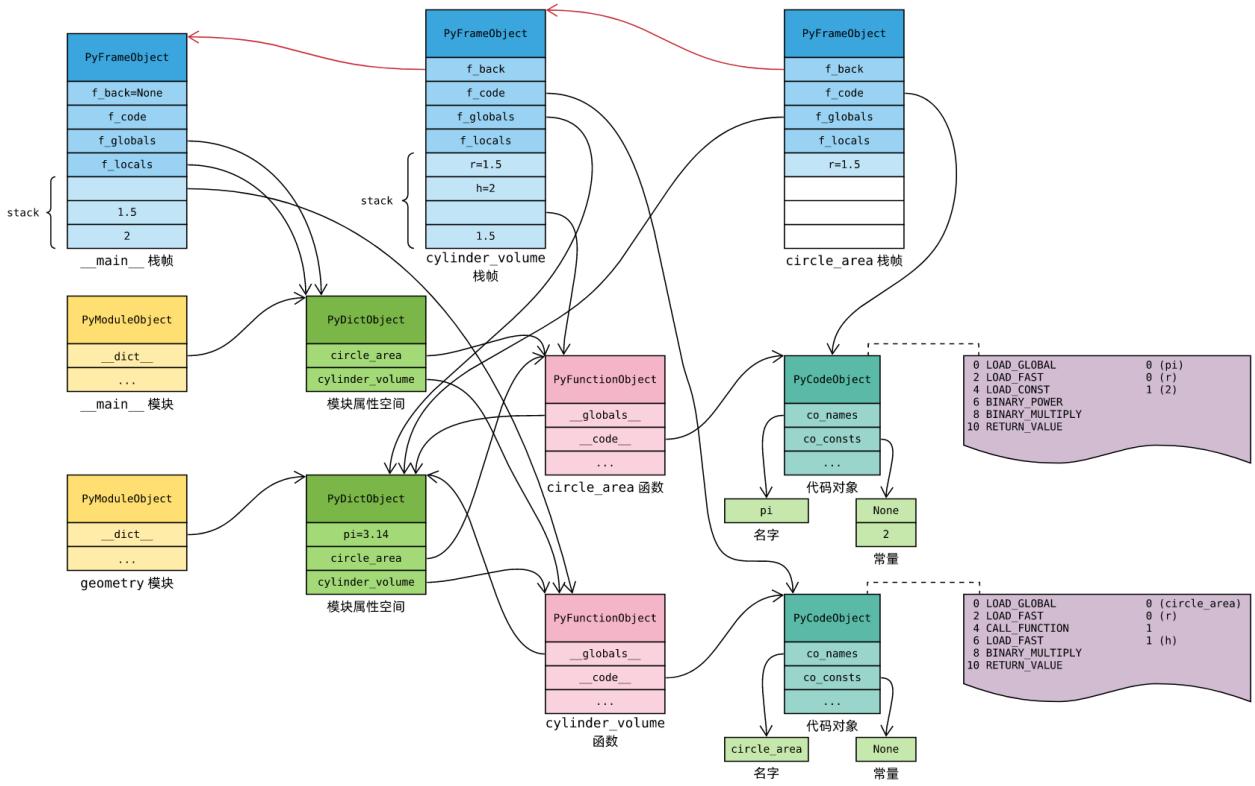
```
>>> cylinder_volume(1.5, 2)
14.13
```

Python 交互式终端同样先对这个语句进行编译，得到这样的字节码：

```
  1           0 LOAD_NAME                0 (cylinder_volume)
              2 LOAD_CONST               0 (1.5)
              4 LOAD_CONST               1 (2)
              6 CALL_FUNCTION            2
              8 PRINT_EXPR
             10 LOAD_CONST               2 (None)
             12 RETURN_VALUE
```

然后， *Python* 虚拟机以 *__main__* 栈帧对象为环境，执行这段字节码。当虚拟机执行到 *CALL_FUNCTION* 这个字节码时，创建新栈帧对象，准备执行函数调用。初始新栈帧对象时，函数参数来源于当前栈顶，而全局名字空间与代码对象来源于被调用函数对象。新栈帧对象初始

化完毕，虚拟机便跳到新栈帧，开始执行 *cylinder_volume* 的字节码。*cylinder_volume* 字节码中也有 *CALL_FUNCTION* 指令，调用 *circle_area* 函数。虚拟机依样画葫芦，为 *circle_area* 准备栈帧，并开始执行 *circle_area* 的字节码：



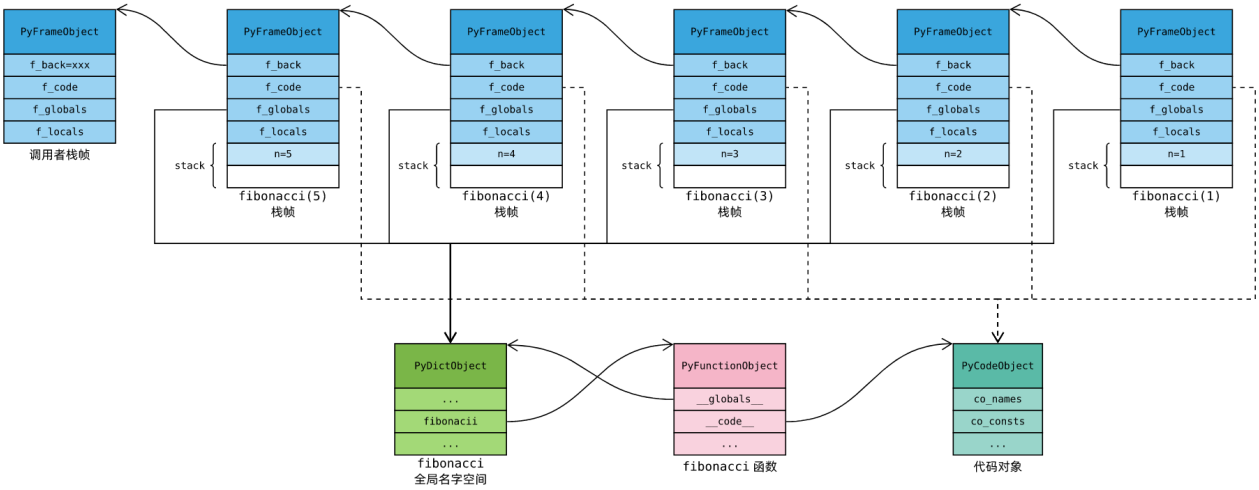
这样一来，随着函数调用的深入，栈帧链逐渐伸长；随着函数执行完毕并返回，栈帧链逐渐收缩。维护栈帧链条的关键是栈帧对象的 *f_back* 指针，它总是指向上个一栈帧对象，也就是调用者的栈帧，如上图红色箭头。我们在调试程序时，可以查看完整的堆栈信息，也是 *f_back* 指针的功劳。

正常情况下，函数调用层数不会太深，但递归调用就说不定了。我们来看一个典型的递归例子，斐波那契数列计算：

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fibonacci(n-1) + fibonacci(n-2
```

以 *fibonacci(5)* 为例，需要递归调用 *fibonacii(4)* ，而 *fibonacii(4)* 需要调用 *fibonacci(3)* ，以此类推。递归调用一直向下延伸，直到 *fibonacci(1)* 。因此，调用链最长时时这样子的：



由此可见，调用 *fibonacci(10000)* 时，栈帧链长度将达到 *10000* 。因此，实现递归算法需要特别小心，不免栈内存溢出。此外，每次函数调用都需要创建并销毁栈帧对象，是否意味着性能低下呢？

这是肯定的。*Python* 内部通过 **内存池** 技术对频繁分配回收内存的场景进行了优化，栈帧频繁创建回收导致的性能问题得到一定的缓解。内存池技术将在 *Python* **内存管理** 部分进行介绍，敬请期待。

24 函数对象诞生记

26 面试必问：嵌套函数、闭包与装饰器