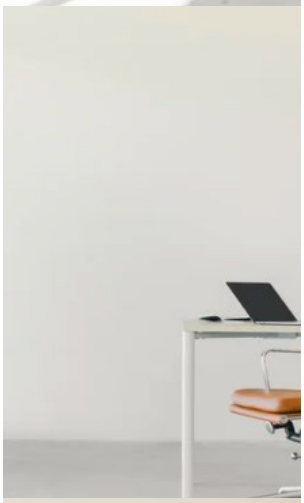
《源码探秘 CPython》58. 函数在底层是如何调用的?

原创 古明地觉 古明地觉的编程教室 2022-03-30 09:00



加强自身学习，提高自身素质。积累工作经验，改进工作方法，向周围同志学习，注重别人优点，学习他们处理问题的方法，查找不足，提高自己。

楔子

上一篇文章我们说了Python函数的底层实现，并且还演示了如何通过函数的类型对象自定义一个函数，以及如何获取函数的参数。虽然这在工作中没有太大意义，但是可以让我们深刻理解函数的行为。

本次就来看看函数如何调用的。

PyCFunctionObject

在介绍调用之前，我们需要补充一个知识点。

```
1  def foo():
2      pass
3
4  print(type(foo))  # <class 'function'>
5  print(type(sum))  # <class 'builtin_function_or_method'>
```

函数实际上分为两种：

- **如果是Python函数，底层会对应PyFunctionObject。其类型在Python里面是 <class 'function'>，在底层是 PyFunction_Type；**
- **如果是C函数，底层会对应PyCFunctionObject。其类型在Python里面是 <class 'builtin_function_or_method'>，在底层类型是 PyCFunction_Type；**

像内置函数、使用 C 扩展编写的函数，它们都是 PyCFunctionObject。

```
PyTypeObject PyCFunction_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "builtin_function_or_method",
    sizeof(PyCFunctionObject),
    0,
    (destructor)meth_dealloc,
    offsetof(PyCFunctionObject, vectorcall),
    0,
```


另外从名字上可以看出PyCFunctionObject不仅用于C函数，还用于方法。关于方法，我们后续在介绍类的时候细说，这里暂时不做深入讨论。

总之对于 Python 函数和 C 函数，底层在实现的时候将两者分开了，因为 C 函数可以有更快的执行方式。

函数的调用

```
1  s = """
2  def foo():
3      a, b = 1, 2
4      return a + b
5
6  foo()
7  """
8
9  if __name__ == '__main__':
10     import dis
11     dis.dis(compile(s, "call_function", "exec"))
```

还是以一个简单的函数为例，看看它的字节码：

```
# 遇到 def，表示构建函数
```

```
 # 于是加载PyCodeObject、函数名 "foo"
 0 LOAD_CONST               0 (<code object foo at 0x......>)
 2 LOAD_CONST               1 ('foo')
 # 构建函数
 4 MAKE_FUNCTION            0
 # 将构建的函数使用 foo 保存
 6 STORE_NAME               0 (foo)

 # 加载变量 foo
 8 LOAD_NAME                0 (foo)
 # 函数调用，该指令是一会要剖析的重点
10 CALL_FUNCTION            0
 # 从栈顶弹出返回值
12 POP_TOP
 # return None
14 LOAD_CONST               2 (None)
16 RETURN_VALUE

Disassembly of <code object foo at 0x0.......>:
 # 函数的字节码，因为模块和函数都会对应 PyCodeObject
 # 只不过后者在前者的常量池中

 # 加载元组常量 (1, 2)
 0 LOAD_CONST               1 ((1, 2))
 # 解包，将元组里的元素压入运行时栈
 2 UNPACK_SEQUENCE          2
 # 分别赋值给 a、b
 4 STORE_FAST               0 (a)
 6 STORE_FAST               1 (b)

 # 加载变量 a、b
 8 LOAD_FAST                0 (a)
10 LOAD_FAST                1 (b)
 # 进行加法运算
12 BINARY_ADD
 # 将相加之后的值返回
14 RETURN_VALUE
```


我们看到调用函数用的是CALL_FUNCTION指令，那么这个指令都做了哪些事情呢？

```
case TARGET(CALL_FUNCTION): {
    PREDICTED(CALL_FUNCTION);
    PyObject **sp, *res;
    //指向运行时栈的栈顶
    sp = stack_pointer;
    //调用函数，将返回值赋值给res
    //tstate表示线程状态对象
    //&sp是一个三级指针，oparg表示指令的操作数
    res = call_function(tstate, &sp, oparg, NULL);
    //函数执行完毕之后，sp会指向运行时栈的栈顶
    //所以再将修改之后的 sp赋值给stack_pointer
    stack_pointer = sp;
    //将 res 压入栈中：*stack_pointer++ = res
    PUSH(res);
    if (res == NULL) {
        goto error;
    }
    DISPATCH();
}
```

记得CALL_FUNCTION这个指令之前是说过的，但是函数的核心执行流程是在call_function里面，它位于 ceval.c 中，我们来看一下。

```
Py_LOCAL_INLINE(PyObject *) _Py_HOT_FUNCTION
call_function(PyThreadState *tstate, PyObject ***pp_stack,
              Py_ssize_t oparg, PyObject *kwnames)
{
```

```
/* pp_stack是在CALL_FUNCTION指令中传入的栈顶指针的指针
|     | 由于栈里面的元素都是 PyObject *
|     | 所以栈顶指针 stack_pointer 是二级指针 PyObject**
|_____| 因为 stack_pointer 永远指向运行时栈的栈顶
|_____|
|_____| 而 pp_stack是 PyObject ***，它是三级指针，也就是栈顶指针的指针
|_____| 如果解引用，那么 *pp_stack等价于stack_pointer
|_____| 两者都指向运行时栈的栈顶，也就是我们说的栈顶指针
 * 栈底到栈顶的元素依次是：函数、参数1、参数2、...、参数n
 * 所以 *((*pp_stack) - oparg - 1) 即可拿到函数指针
 */
PyObject **pfunc = (*pp_stack) - oparg - 1;
//这里的 func就等价于我们在Python中定义的foo
//可能有人好奇，为什么非要搞一个三级指针出来
//为什么不能直接传 stack_pointer 呢？
//答案是函数执行完毕之后，运行时栈的元素会发生变化
//而这也意味着stack_pointer会发生变化，因此必须把它的指针传进去
PyObject *func = *pfunc;
//两个 PyObject *
PyObject *x, *w;
//通过关键字参数传递的参数个数，对于当前函数来说是 0
Py_ssize_t nkwargs = (kwnames == NULL) ? 0 : PyTuple_GET_SIZE(kwnames);
//通过位置参数传递的参数个数，对于当前函数来说也是 0
Py_ssize_t nargs = oparg - nkwargs;
//移动栈指针，这里相当于 (*pp_stack) - oparg
//所以在移动之后，stack 会指向第一个参数
PyObject **stack = (*pp_stack) - nargs - nkwargs;

//上面相当于获取函数、以及相关的参数，而获取完就开始调用了
//调用有两种方式，分别是：trace_call_function和_PyObject_Vectorcall
//所以我们需要调用 C 函数，来实现 Python 函数的调用
//那么这两个 C 函数之间有什么区别呢？
//首先虚拟机支持我们给线程绑定一个追踪函数，可以通过 threading.settrace实现
//然后在执行字节码的时候，会触发绑定的追踪函数，该操作是线程绑定的
//此外还有sys.settrace，它会作用于整个解释器，也就是所有线程
//如果绑定了追踪函数，那么tstate->use_tracing为真，会执行trace_call_function
if (tstate->use_tracing) {
    x = trace_call_function(tstate, func, stack, nargs, kwnames);
}
//如果没有绑定追踪函数，那么执行 _PyObject_Vectorcall
//事实上在trace_call_function里面，还是调用了_PyObject_Vectorcall
//因此就Python函数本身而言，它的执行始终是通过 _PyObject_Vectorcall 实现的
//这个_PyObject_Vectorcall应该还有印象吧，上一篇文章提到过
//介绍函数的时候，我们看到 PyFunctionObject有一个vectorcall成员
//而在通过PyFunction_NewWithQualName创建函数时
//会将_PyObject_Vectorcall赋值给vectorcall
else {
    x = _PyObject_Vectorcall(func, stack,
            nargs | PY_VECTORCALL_ARGUMENTS_OFFSET, kwnames);
}
//执行完毕之后，将返回值赋值给 x，而在CALL_FUNCTION指令中，有下面一行代码：
//res = call_function(tstate, &sp, oparg, NULL);
//在 CALL_FUNCTION 里面拿到的 res，就是这里的 x，后续会将 res 压入运行时栈
//如果没有接收返回值，那么再用 POP_TOP 将其从栈顶弹出、丢弃
//如果接收了返回值，那么就用 STROE_FAST 将其保存起来
assert((x != NULL) ^ (_PyErr_Occurred(tstate) != NULL));

//当然啦，在后续将 res 压入运行时栈之前
//这里要先将当前栈中已有的元素全部清除
while ((*pp_stack) > pfunc) {
    // 将元素从栈中弹出，并减少引用计数
    w = EXT_POP(*pp_stack);
    Py_DECREF(w);
}
//while循环结束之后，函数以及相关参数就被清空了，此时栈顶也发生了改变
//因此在 CALL_FUNCTION 中，还要将 *pp_stack 重新赋值给 stack_pointer
//所以会有一行 stack_pointer = sp，而这个 sp 就是 *pp_stack
//现在应该明白调用 call_function 时，为什么传递三级指针(&sp)了
//因为需要在 call_function 执行完毕之后，外部的 sp 能够被影响
//所以必须传一个 &sp 过去，也就是三级指针，当*pp_stack变化时，外部的sp也在变化
//而 call_function执行完毕后，sp 会指向新的栈顶，再将它赋值给 stack_pointer
//然后执行 PUSH(res)，也就是 *stack_pointer++ = res
```

```
    //然后执行 PUSH(res); 也就是 *stack_pointer++ = res
    return x;
    //通过以上的分析，是不是将内容又都串联起来了呢？
}
```

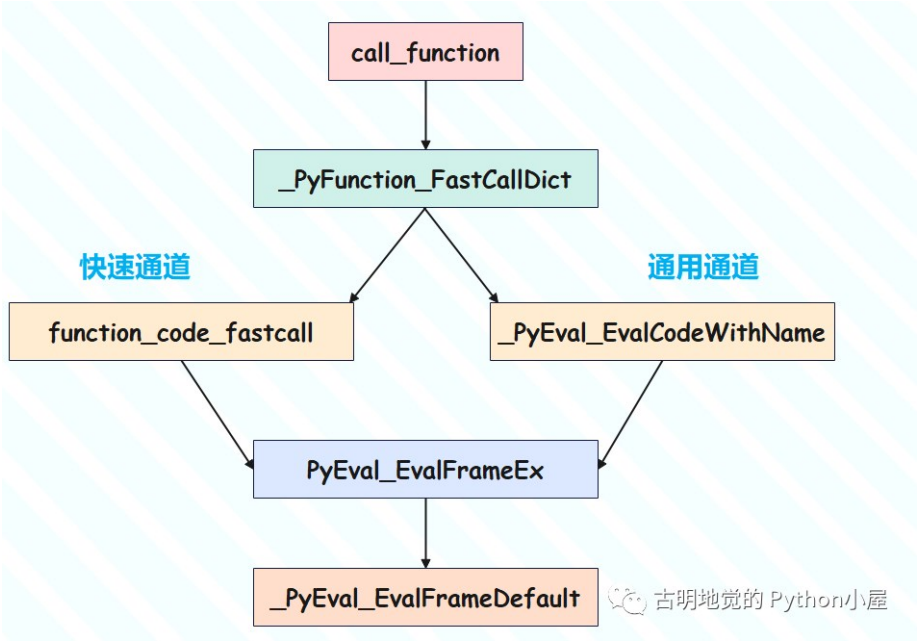

因此接下来重点就在_PyObject_Vectorcall函数上面，在该函数内部又会调用其它函数，最终会走到_PyFunction_FastCallDict 这里。

```c
//Objects/call.c
PyObject *
_PyFunction_FastCallDict(PyObject *func, PyObject *const *args, Py_ssize
_t nargs,
                         PyObject *kwargs)
{
    //获取PyCodeObject对象
    PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
    //获取global名字空间
    PyObject *globals = PyFunction_GET_GLOBALS(func);
    //获取默认值
    PyObject *argdefs = PyFunction_GET_DEFAULTS(func);
    //....

    //我们观察一下下面的return
    //一个是function_code_fastcall，一个是最后的_PyEval_EvalCodeWithName
    //从名字上能看出来function_code_fastcall是一个快分支
    //但是这个快分支要求函数调用时不能传递关键字参数
    if (co->co_kwonlyargcount == 0 &&
        (kwargs == NULL || PyDict_GET_SIZE(kwargs) == 0) &&
        (co->co_flags & ~PyCF_MASK) == (CO_OPTIMIZED | CO_NEWLOCALS | CO
_NOFREE))
    {
        /* Fast paths */
        if (argdefs == NULL && co->co_argcount == nargs) {
            //function_code_fastcall里面逻辑很简单
            //直接抽走当前PyFunctionObject里面PyCodeObject和global名字空间
            //根据PyCodeObject对象直接为其创建一个PyFrameObject对象
            //然后PyEval_EvalFrameEx执行栈帧
            //也就是真正的进入了函数调用，执行函数里面的代码
            return function_code_fastcall(co, args, nargs, globals);
        }
        else if (nargs == 0 && argdefs != NULL
                 && co->co_argcount == PyTuple_GET_SIZE(argdefs)) {
            /* function called with no arguments, but all parameters have
               a default value: use default values as arguments .*/
            args = _PyTuple_ITEMS(argdefs);
            return function_code_fastcall(co, args, PyTuple_GET_SIZE(argd
efs),
                                          globals);
        }
    }

    //适用于有关键字参数的情况
    nk = (kwargs != NULL) ? PyDict_GET_SIZE(kwargs) : 0;
    //.....
    //调用_PyEval_EvalCodeWithName
    result = _PyEval_EvalCodeWithName((PyObject*)co, globals, (PyObject
*)NULL,
                                      args, nargs,
                                      k, k != NULL ? k + 1 : NULL, nk, 2,
                                      d, nd, kwdefs,
                                      closure, name, qualname);
    Py_XDECREF(kwtuple);
    return result;
}
```

所以函数调用时会有两种方式：



因此我们看到，总共有两条途径，分别针对有无关键字参数。但是最终殊途同归，都会走到PyEval_EvalFrameEx那里，然后虚拟机在新的栈帧中执行新的PyCodeObject。

不过可能有人会问，我们之前说过PyFrameObject是根据PyCodeObject创建的，而PyFunctionObject也是根据PyCodeObject创建的，那么PyFrameObject和PyFunctionObject之间有啥关系呢？

如果把PyCodeObject比喻成**妹子**的话，那么PyFunctionObject就是妹子的**备胎**，PyFrameObject就是妹子的**心上人**。其实在栈帧中执行的指令时候，PyFunctionObject的影响就已经消失了，真正对栈帧产生影响的是PyFunctionObject里面的PyCodeObject对象和global名字空间。

也就是说，最终是PyFrameObject对象和PyCodeObject对象两者如胶似漆，跟PyFunctionObject对象之间没有关系，所以PyFunctionObject辛苦一场，实际上是为别人做了嫁衣。PyFunctionObject主要是对PyCodeObject和global名字空间的一种打包和运输方式。

关于这两条执行途径的具体细节，以及参数是如何解析的，我们下一篇文章来说。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》59. 函数是如何解析位置参数的?

下一篇 ›
《源码探秘 CPython》57. 函数是怎么创建的?