《源码探秘 CPython》57. 函数是怎么创建的?

原创 古明地觉 古明地觉的编程教室 2022-03-29 09:00

上一篇介绍了**PyFunctionObject**结构体，它是函数的底层实现，在里面我们看到了func_code成员，它指向一个PyCodeObject对象，函数就是根据它创建的。

因为一个PyCodeObject是对一段代码的静态表示，Python编译器在将源代码进行编译之后，对里面的每一个代码块（code block）都会生成一个、并且是唯一一个PyCodeObject对象，该对象包含了这个代码块的一些静态信息，也就是可以从源代码中看到的信息。

比如：某个函数对应的代码块里面有一个 a=1 这样的表达式，那么符号a和整数1、以及它们之间的联系就是静态信息。这些信息会被静态存储起来，符号a被存在符号表co_varnames中、整数1被存在常量池co_consts中。这两者之间是一个赋值语句，因此会有两条指令**LOAD_CONST**和**STORE_FAST**存在字节码指令序列co_code中。

这些信息是编译的时候就可以得到的，因此PyCodeObject对象是编译之后的结果。

但是PyFunctionObject对象是何时产生的呢？显然它是Python代码在运行时动态产生的，更准确的说，是在执行一个def语句的时候创建的。

当虚拟机在当前栈帧中执行字节码时发现了def语句，那么就代表发现了新的PyCodeObject对象，因为它们是可以层层嵌套的。所以虚拟机会根据这个PyCodeObject对象创建对应的PyFunctionObject对象，然后将函数名和PyFunctionObject对象（函数体）组成键值对放在当前的local空间中。

而在PyFunctionObject对象中，也需要拿到相关的静态信息，因此会有一个func_code成员指向PyCodeObject。

除此之外，PyFunctionObject对象中还包含了一些函数在执行时所必需的动态信息，即上下文信息。比如func_globals，就是函数在执行时关联的global空间，说白了就是在局部变量找不到的时候能够找全局变量，可如果连global空间都没有的话，那即便想找也无从下手呀。

而global作用域中的符号和值必须在运行时才能确定，所以这部分必须在运行时动态创建，无法静态存储在PyCodeObject中，因此要根据PyCodeObject对象创建PyFunctionObject对象，相当于一个封装。总之一切的目的，都是为了更好地执行字节码。

我们举个例子：

```
# 首先虚拟机从上到下执行字节码
name = "古明地觉"
age = 16


# 啪，很快啊，出现了一个def
def foo():
    pass

"""
出现 def，虚拟机就知道源代码进入一个新的作用域了
也就是遇到一个新的PyCodeObject对象了
而通过def知道这是一个函数，所以会进行封装
把PyCodeObject对象封装成PyFunctionObject对象
所以当执行完def语句之后，一个函数就被创建了
```

```
创建完之后会放在当前的local空间中，当然对于模块来说：local空间也是global空间
"""
print(locals())  # {......, 'foo': <function foo at 0x0000018299FAF3A0>}
```


调用的时候，会从local空间中取出符号foo对应的PyFunctionObject对象。然后根据这个PyFunctionObject对象创建PyFrameObject对象，也就是为函数创建一个栈帧，随后将执行权交给新创建的栈帧，并在新创建的栈帧中执行字节码。

> 函数的类型是<class 'function'>，但这个类解释器没有暴露给我们。

函数是怎么创建的？

经过分析我们知道，当执行到def语句时会创建函数，并保存在local空间中。而通过**函数名()**进行调用时，会从local空间取出和函数名绑定的函数对象，然后执行。

那么问题来了，函数（对象）是怎么创建的呢？或者说虚拟机是如何完成PyCodeObject对象到PyFunctionObject对象之间的转变呢？显然想了解这其中的奥秘，就必须从字节码入手。

```
s = """
name = "古明地觉"
def foo(a, b):
    print(a, b)

foo(1, 2)
"""

import dis
dis.dis(compile(s, "func", "exec"))
```

源代码很简单，定义一个变量name和函数foo，然后调用函数。显然这里面会产生两个PyCodeObject，我们来看一下。

```
    # 加载字符串常量 "古明地觉"
2   0 LOAD_CONST               0 ('古明地觉')
    # 使用变量 name 保存起来
    2 STORE_NAME               0 (name)

    # 注意这一步，也是 LOAD_CONST，但它加载的一个 PyCodeObject 对象
    # 所以 PyCodeObject 对象也是一个常量，会通过 LOAD_CONST 进行加载
3   4 LOAD_CONST               1 (<code object foo at 0x00000......>)
    # 加载字符串常量 "foo"，也就是函数名
    6 LOAD_CONST               2 ('foo')
    # 从名字上也能看出是创建函数，基于加载的 PyCodeObject 对象进行创建
    8 MAKE_FUNCTION            0
    # 使用变量 foo 保存起来
    # 也就是将 "foo" 和 PyFunctionObject对象绑定起来放到 local 空间中
    # 后续通过 foo() 即可对函数发起调用
   10 STORE_NAME               1 (foo)

    # 函数创建完了，我们执行函数
    # 通过变量 foo 找到指向的函数对象（PyFunctionObject）
6  12 LOAD_NAME                1 (foo)
    # 加载整数常量 1 和 2
   14 LOAD_CONST               3 (1)
   16 LOAD_CONST               4 (2)
    # 调用
   18 CALL_FUNCTION            2
    # 从栈顶弹出返回值
   20 POP_TOP
    # return None
   22 LOAD_CONST               5 (None)
   24 RETURN_VALUE
```

```
  # 以上是模块对应的字节码指令，下面是函数 foo 的字节码指令
Disassembly of <code object foo at 0x00000......>:
  # 从局部作用域中加载内置变量 print
4      0 LOAD_GLOBAL              0 (print)
  # 从局部作用域中加载局部变量 a 和 b
       2 LOAD_FAST                0 (a)
       4 LOAD_FAST                1 (b)
  # 调用
       6 CALL_FUNCTION            2
  # 从栈顶弹出返回值
       8 POP_TOP
  # return None
      10 LOAD_CONST               0 (None)
      12 RETURN_VALUE
```


上面有一个有趣的现象，就是源代码的行号。之前看到源代码的行号都是从上往下、依次增大的，这很好理解，毕竟一条一条解释嘛。但是这里却发生了变化，先执行了第 6 行，之后再执行第 4 行。

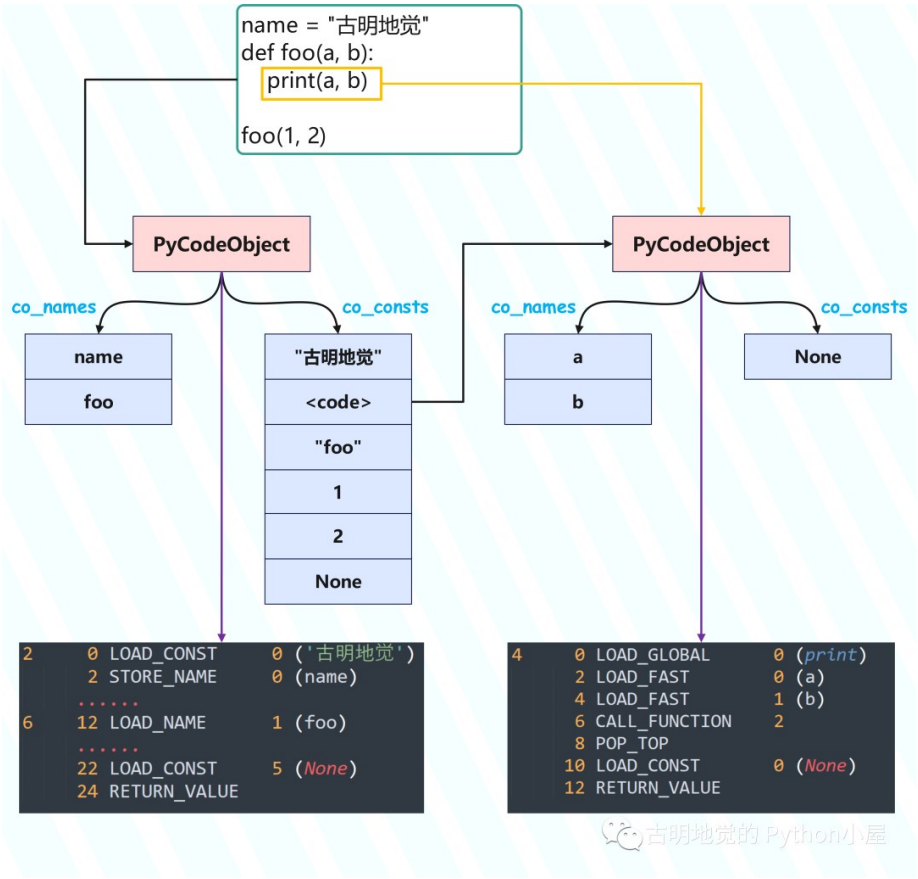
如果是从Python层面的函数调用来理解的话，很容易一句话就解释了，因为函数只有在调用的时候才会执行，而调用肯定发生在创建之后。但是从字节码的角度来理解的话，我们发现函数的声明和实现是分离的，是在不同的PyCodeObject对象中。

确实如此，虽然函数名和函数体是一个整体，但是虚拟机在实现的时候，却在物理上将它们分离开了。

正所谓函数即变量，我们可以把函数当成普通的变量来处理。函数名就是变量名，它位于模块对应的PyCodeObject的符号表中；函数体就是变量指向的值，它是基于一个独立的 PyCodeObject构建的。

> 换句话说，在编译时，函数体里面的代码会位于一个新的 PyCodeObject 对象当中，所以函数的声明和实现是分离的。

至此，函数的结构就已经非常清晰了。



所以函数名和函数体是分离的，它们存储在不同的PyCodeObject对象当中。

分析完结构之后，重点就要落在MAKE_FUNCTION指令上了，我们说当遇到def foo(a, b)的时候，就知道要创建函数了。在语法上这是函数的声明语句，但是从虚拟机的角度来看这其实是函数对象的创建语句。

所以下面我们就要分析一下这个指令，看看它到底是怎么将一个PyCodeObject对象变成一个PyFunctionObject对象的。

```
case TARGET(MAKE_FUNCTION): {
    //弹出符号表中的函数名
    PyObject *qualname = POP();
    //弹出对应的PyCodeObject对象
    PyObject *codeobj = POP();
    //创建PyFunctionObject对象， 接收三个参数
    //分别是PyCodeObject对象、global名字空间、函数的全限定名
    //我们看到创建函数的时候将global名字空间传递了进去
    //所以我们现在明白为什么函数可以调用__globals__了
    //当然也明白为什么函数在局部变量找不到的时候可以去找全局变量了
    PyFunctionObject *func = (PyFunctionObject *)
        PyFunction_NewWithQualName(codeobj, f->f_globals, qualname);

    //减少引用计数
    //如果函数创建失败会返回 NULL，跳转至 error
    Py_DECREF(codeobj);
    Py_DECREF(qualname);
    if (func == NULL) {
        goto error;
    }

    //编译时，解释器能够静态检测出函数有没有设置闭包、类型注解等属性
    //比如设置了闭包，那么 oparg & 0x08 为真
    //设置了类型注解，那么 oparg & 0x04 为真
    //如果条件为真，那么进行相关属性设置
    if (oparg & 0x08) {
        assert(PyTuple_CheckExact(TOP()));
        func ->func_closure = POP();
    }
    if (oparg & 0x04) {
        assert(PyDict_CheckExact(TOP()));
        func->func_annotations = POP();
    }
    if (oparg & 0x02) {
        assert(PyDict_CheckExact(TOP()));
        func->func_kwdefaults = POP();
    }
    if (oparg & 0x01) {
        assert(PyTuple_CheckExact(TOP()));
        func->func_defaults = POP();
    }

    //将函数对象的指针压入运行时栈
    PUSH((PyObject *)func);
    DISPATCH();
}
```

整个步骤很好理解，先通过LOAD_CONST将**PyCodeObject对象**和**符号foo**压入栈中。然后执行MAKE_FUNCTION的时候，将两者从栈中弹出，再加上当前栈帧对象中维护的global名字空间，三者作为参数传入PyFunction_NewWithQualName函数中，从而构建出相应的PyFunctionObject对象。

下面来看看PyFunction_NewWithQualName是如何构造出一个函数的，它位于funcobject.c 中。

```
// Objects/funcobject.c
PyObject *
PyFunction_NewWithQualName(PyObject *code, PyObject *globals, PyObject *
qualname)
{
```

```
    //要返回的PyFunctionObject对象的指针
    PyFunctionObject *op;
    //函数的doc、常量池、函数所在的模块
    PyObject *doc, *consts, *module;
    if (__name__ == NULL) {
        __name__ = PyUnicode_InternFromString("__name__");
        if (__name__ == NULL)
            return NULL;
    }
    //通过PyObject_GC_New为函数对象申请空间，这里我们看到了 gc
    //因为函数是可以发生循环引用的，因此需要被 GC 跟踪
    //而想被 GC 跟踪，则需要有一个PyGC_Head
    //所以此处使用PyObject_GC_New，同时也会为PyGC_Head申请内存
    op = PyObject_GC_New(PyFunctionObject, &PyFunction_Type);
    //申请失败返回 NULL
    if (op == NULL)
        return NULL;

    //下面就是设置PyFunctionObject对象的成员属性了
    op->func_weakreflist = NULL;
    Py_INCREF(code);
    op->func_code = code;
    Py_INCREF(globals);
    op->func_globals = globals;
    op->func_name = ((PyCodeObject *)code)->co_name;
    Py_INCREF(op->func_name);
    op->func_defaults = NULL; /* No default arguments */
    op->func_kwdefaults = NULL; /* No keyword only defaults */
    op->func_closure = NULL;
    //以后会通过_PyFunction_Vectorcall来实现函数的调用
    op->vectorcall = _PyFunction_Vectorcall;
    //通过PyCodeObject对象获取常量池
    consts = ((PyCodeObject *)code)->co_consts;

    //我们知道函数的 doc 其实就是一个字符串
    //显然它也是常量池的一个常量，并且是常量池的第一个常量
    //如果函数没有 doc，那么它的常量池里的第一个元素一定不是字符串
    if (PyTuple_Size(consts) >= 1) {
        //所以如果consts的长度 >=1，并且第一个元素是字符串
        //那么它就是函数的 doc
        doc = PyTuple_GetItem(consts, 0);
        if (!PyUnicode_Check(doc))
            doc = Py_None;
    }
    else //否则doc就是None
        doc = Py_None;
    Py_INCREF(doc);
    //下面也是设置PyFunctionObject对象的成员
    op->func_doc = doc;
    op->func_dict = NULL;
    op->func_module = NULL;
    op->func_annotations = NULL;

    /* __module__: If module name is in globals, use it.
       Otherwise, use None. */
    module = PyDict_GetItemWithError(globals, __name__);
    //从global名字空间中获取__name__，作为函数的 __module__
    if (module) {
        Py_INCREF(module);
        op->func_module = module;
    }
    else if (PyErr_Occurred()) {
        Py_DECREF(op);
        return NULL;
```

```
    }
    //op->func_qualname = qualname if qualname else op->func_name
    if (qualname)
        op->func_qualname = qualname;
    else
        op->func_qualname = op->func_name;
    Py_INCREF(op->func_qualname);
    //让函数对象被 GC 跟踪
    _PyObject_GC_TRACK(op);
    //返回其指针
    return (PyObject *)op;
}
```

以上就是函数的创建过程，说白了就是对PyCodeObject进行了一个封装。在创建完毕之后会回到MAKE_FUNCTION，然后和**函数名**组成 entry 存在当前栈帧的 local 名字空间中，整体还是比较简单的。

但如果再加上类型注解、以及默认值，会有什么效果呢？

```
s = """
name = "古明地觉"
def foo(a: int = 1, b: int = 2):
    print(a, b)

foo(1, 2)
"""

import dis
dis.dis(compile(s, "func", "exec"))
```

这里我们加上了类型注解和默认值，看看它的字节码指令会有什么变化？

```
    0 LOAD_CONST               0 ('古明地觉')
    2 STORE_NAME               0 (name)

    4 LOAD_CONST               7 ((1, 2))
    6 LOAD_NAME                1 (int)
    8 LOAD_NAME                1 (int)
   10 LOAD_CONST               3 (('a', 'b'))
   12 BUILD_CONST_KEY_MAP      2
   14 LOAD_CONST               4 (<code object foo at 0x0......>)
   16 LOAD_CONST               5 ('foo')
   18 MAKE_FUNCTION            5 (defaults, annotations)
   ......
   ......
```

不难发现，在构建函数时会先将默认值以元组的形式压入运行时栈；然后再根据使用了类型注解的**参数**和**类型**构建一个字典，并将这个字典压入运行时栈。

后续创建函数的时候，会将默认值保存在 func_defaults 成员中，类型注解对应的字典会保存在 func_annotations 成员中。

```
def foo(a: int = 1, b: int = 2):
    print(a, b)

print(foo.__defaults__)
print(foo.__annotations__)
# (1, 2)
# {'a': <class 'int'>, 'b': <class 'int'>}
```

基于类型注解和描述符，我们便可以像静态语言一样，实现函数参数的类型约束。介绍完描述符之后，我们会举例说明。

函数的一些骚操作

我们通过一些骚操作，来更好地理解一下函数。

之前说<class 'function'>是函数的类型对象，而这个类底层没有暴露给我们，但是我们依旧可以通过曲线救国的方式进行获取。

```
def f():
    pass

print(type(f))  # <class 'function'>
# lambda匿名函数的类型也是 function
print(type(lambda: None))  # <class 'function'>
```

那么下面就来创建函数：

```
gender = "female"

def f(name, age):
    return f"name: {name}, age: {age}, gender: {gender}"
# 得到PyCodeObject对象
code = f.__code__
# 根据类function创建函数对象
# 接收三个参数：PyCodeObject对象、名字空间、函数名
new_f = type(f)(code, globals(), "根据f创建的new_f")

# 打印函数名
print(new_f.__name__)  # 根据f创建的new_f

# 调用函数
print(new_f("古明地觉", 16))  # name: 古明地觉, age: 16, gender: female
```

是不是很神奇呢？另外我们说函数在访问变量时，显然先从自身的符号表中查找，如果没有再去找全局变量。这是因为，我们在创建函数的时候将global名字空间传进去了，如果我们不传递呢？

```
gender = "female"

def f(name, age):
    return f"name: {name}, age: {age}, gender: {gender}"

code = f.__code__
try:
    new_f = type(f)(code, None, "根据f创建的new_f")
except TypeError as e:
    print(e)  # function() argument 'globals' must be dict, not None
# 这里告诉我们function的第二个参数globals必须是一个字典
# 我们传递一个空字典
new_f1 = type(f)(code, {}, "根据f创建的new_f1")

# 打印函数名
print(new_f1.__name__)  # 根据f创建的new_f1

# 调用函数
try:
    print(new_f1("古明地觉", 16))
except NameError as e:
```

```
    print(e)  # name 'gender' is not defined

# 我们看到提示 gender 没有定义
```

因此现在我们又从Python的角度理解了一遍，为什么函数能够在局部变量找不到的时候，去找全局变量。原因就在于构建函数的时候，将global名字空间交给了函数，使得函数可以在global空间进行变量查找，所以它才能够找到全局变量。而我们这里给了一个空字典，那么显然就找不到gender这个变量了。

```
gender = "female"

def f(name, age):
    return f"name: {name}, age: {age}, gender: {gender}"

code = f.__code__
new_f = type(f)(code, {"gender": "萌妹子"}, "根据f创建的new_f")

# 我们可以手动传递一个字典进去
# 此时我们传递的字典对于函数来说就是global名字空间
# 所以在函数内部找不到某个变量的时候，就会去我们指定的名字空间中查找
print(new_f("古明地觉", 16))  # name: 古明地觉, age: 16, gender: 萌妹子
# 所以此时的gender不再是外部的 "female"，而是我们指定的 "萌妹子"
```

此外我们还可以为函数指定默认值：

```
def f(name, age, gender):
    return f"name: {name}, age: {age}, gender: {gender}"

# 必须接收一个PyTupleObject对象
f.__defaults__ = ("古明地觉", 16, "female")
print(f())
"""
name: 古明地觉, age: 16, gender: female
"""
```

我们看到函数 f 明明接收三个参数，但是调用时不传递居然也不会报错，原因就在于我们指定了默认值。而默认值可以在定义函数的时候指定，也可以通过**__defaults__**指定，但很明显我们应该通过前者来指定。

> 如果你用的是pycharm，那么会在 f() 这个位置给你飘黄，提示你参数没有传递。但我们知道，由于使用 __defaults__ 已经设置了默认值，所以这里是不会报错的。只不过pycharm没有检测到，当然基本上所有的 IDE 都无法做到这一点，毕竟动态语言。

另外**__defaults__**接收的元组里面的元素个数和参数个数不匹配怎么办？

```
def f(name, age, gender):
    return f"name: {name}, age: {age}, gender: {gender}"

f.__defaults__ = (15, "female")
print(f("古明地恋"))
"""
name: 古明地恋, age: 15, gender: female
"""
```

由于元组里面只有两个元素，意味着我们在调用时需要至少传递一个参数，而这个参数会赋值给 name。原因就是在设置默认值的时候是从后往前设置的，也就是 **"female"** 会给赋值给 gender，**15** 会赋值给 age。而 name 没有得到默认值，那么它就需要调用者显式传递了。

如果返回值从前往后设置的话，会出现什么后果？此时 **15** 会赋值给 name，**"female"** 会赋值给 age，此时就等价于如下：

```
def f(name=15, age="female", gender):
    return f"name: {name}, age: {age}, gender: {gender}"
```

这样的函数能够通过编译吗？显然是不行的，因为默认参数必须在非默认参数的后面。所以Python的这个做法是完全正确的，必须要从后往前进行设置。

另外我们知道默认值的个数是小于等于参数个数的，如果大于会怎么样呢？

```
def f(name, age, gender):
    return f"name: {name}, age: {age}, gender: {gender}"

f.__defaults__ = ("古明地觉", "古明地恋", 15, "female")
print(f())
"""
name: 古明地恋, age: 15, gender: female
"""
```

依旧从后往前进行设置，当所有参数都有默认值了，那么就结束了。当然，如果不使用__defaults__，是不可能出现默认值个数大于参数个数的。

可要是__defaults__指向的元组先结束，那么没有得到默认值的参数就必须由我们来传递了。

最后，再来说一下如何深拷贝一个函数。首先如果是你的话，你会怎么拷贝一个函数呢？不出意外的话，你应该会使用 copy 模块。

```
import copy

def f(a, b):
    return [a, b]

# 但是问题来了，这样能否实现深度拷贝呢？
new_f = copy.deepcopy(f)
f.__defaults__ = (2, 3)
print(new_f())  # [2, 3]
```

修改 f 的 __defaults__，会对 new_f 产生影响，因此我们并没有实现函数的深度拷贝。

> copy 模块无法对函数、方法、回溯栈、栈帧、模块、文件、套接字等类型实现深度拷贝。

那我们应该怎么做呢？

```
from types import FunctionType

def f(a, b):
    return "result"

# FunctionType 就是函数的类型对象
# 它也是通过 type 得到的
new_f = FunctionType(f.__code__,
                     f.__globals__,
                     f.__name__,
                     f.__defaults__,
                     f.__closure__)
# 显然 function 还可以接收第四个参数和第五个参数
# 分别是函数的默认值和闭包

# 然后别忘记将属性字典也拷贝一份
# 由于函数的属性字典几乎用不上，这里就浅拷贝了
new_f.__dict__.update(f.__dict__)

f.__defaults__ = (2, 3)
print(f.__defaults__)  # (2, 3)
print(new_f.__defaults__)  # None
```

此时修改 f 不会影响 new_f，当然在拷贝的时候也可以自定义属性。

其实上面实现的深拷贝，本质上就是定义了一个新的函数。由于是两个不同的函数，那么自然就没有联系了。

判断函数有哪些参数

最后再来看看如何检测一个函数有哪些参数，首先函数的局部变量(包括参数)在编译时就已经确定，会存在符号表co_varnames中。

```
def f(a, b, /, c, d, *args, e, f, **kwargs):
    g = 1
    h = 2

varnames = f.__code__.co_varnames
print(varnames)
# ('a', 'b', 'c', 'd', 'e', 'f', 'args', 'kwargs', 'g', 'h')
```

注意: 在定义函数的时候，*和**最多只能出现一次。

显然 a 和 b 必须通过位置参数传递，c 和 d 可以通过位置参数或者关键字参数传递，e 和 f 必须通过关键字参数传递。

而从打印的符号表来看，里面的符号是有顺序的。参数永远在函数内部定义的局部变量的前面，比如 g 和 h 就是函数内部定义的局部变量，所以它在所有参数的后面。

而对于参数，*和**会位于最后面，其它参数位置不变。所以除了 g 和 h，最后面的就是 args 和 kwargs。

那么接下来我们就可以进行检测了。

```
def f(a, b, /, c, d, *args, e, f, **kwargs):
    g = 1
    h = 2

varnames = f.__code__.co_varnames

# 1. 寻找必须通过位置参数传递的参数
posonlyargcount = f.__code__.co_posonlyargcount
print(posonlyargcount)  # 2
print(varnames[: posonlyargcount])  # ('a', 'b')

# 2. 寻找可以通过位置参数传递或者关键字参数传递的参数
argcount = f.__code__.co_argcount
print(argcount)  # 4
print(varnames[: 4])  # ('a', 'b', 'c', 'd')
print(varnames[posonlyargcount: 4])  # ('c', 'd')

# 3. 寻找必须通过关键字参数传递的参数
kwonlyargcount = f.__code__.co_kwonlyargcount
print(kwonlyargcount)  # 2
print(varnames[argcount: argcount + kwonlyargcount])  # ('e', 'f')

# 4. 寻找 *args 和 **kwargs
flags = f.__code__.co_flags
# 在介绍 PyCodeObject 的时候，我们说里面有一个 co_flags 成员
# 它是函数的标识，可以对函数类型和参数进行检测
```

```
# 如果co_flags和 4 进行按位与之后为真，那么就代表有*args，否则没有
# 如果co_flags和 8 进行按位与之后为真，那么就代表有**kwargs，否则没有
step = argcount + kwonlyargcount
if flags & 0x04:
    print(varnames[step])  # args
    step += 1

if flags & 0x08:
    print(varnames[step])  # kwargs
```

以上我们检测出了函数都有哪些参数，你也可以将其封装成一个函数，实现代码的复用。

然后需要注意一下 args 和 kwargs，打印的内容主要取决定义时使用的名字。如果定义的时候是 *ARGS 和 **KWARGS，那么这里就会打印 ARGS 和 KWARGS，只不过一般我们都叫做 *args 和 **kwargs。

如果我们定义的时候不是 *args，只是一个 *，那么它就不是参数了。

```
def f(a, b, *, c):
    pass

# 我们看到此时只有a、b、c
print(f.__code__.co_varnames)  # ('a', 'b', 'c')

print(f.__code__.co_flags & 0x04)  # 0
print(f.__code__.co_flags & 0x08)  # 0
# 显然此时也都为假
```

单独的一个 * 只是为了强制要求后面的参数必须通过关键字参数的方式传递。

这一次我们简单地分析了一下函数是如何创建的，并且还在Python的层面上做了一些小trick。最后我们也分析了如何通过PyCodeObject对象来检索函数的参数，以及相关种类，标准库中的inspect模块也是这么做的。准确的说，是我们模仿人家的思路做的。

现在你是不是对函数有了一个更深刻的认识了呢？当然目前介绍的只是函数的一部分内容，还有函数如何调用、位置参数和关键字参数如何解析、对于有默认值的参数如何在我们不传递的时候使用默认值以及在我们传递的时候使用我们传递的值、*args 和 **kwargs又如何解析、闭包怎么实现、还有装饰器等等等等，这些我们接下来慢慢说。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》58. 函数在底层是如何调用的?

下一篇 ›
《源码探秘 CPython》56. 函数的底层结构

【C语言】练习题 - 函数名重载的名字装饰

一起上编程课