

微信扫一扫  
关注该公众号

收录于合集

#CPython

97个 &gt;



函数是任何一门编程语言都具备的基本元素，它可以将多个动作组合起来，一个函数代表了一系列的动作。而且在调用函数时会干什么来着，没错，要创建栈帧，用于函数的执行。

那么下面就来看看函数在 C 中是如何实现的，生得一副什么模样。



Python中一切皆对象，函数也不例外。函数这种抽象机制在底层是通过 PyFunctionObject 对象实现的，位于funcobject.h 中。

```
typedef struct {
    /* 头部信息，不用多说 */
    PyObject_HEAD
    /* 函数的 PyCodeObject 对象，函数就是根据该 PyCodeObject 对象创建的 */
    PyObject *func_code;
    /* 函数的 global 名字空间 */
    PyObject *func_globals;
    /* 函数参数的默认值，一个元组或者空 */
    PyObject *func_defaults;
    /* 只能通过关键字参数传递的 "参数" 和 "该参数的默认值" 组成的字典，或者空 */
    PyObject *func_kwdefaults;
    /* 闭包对象 */
    PyObject *func_closure;
    /* 函数的 doc */
    PyObject *func_doc;
    /* 函数名 */
    PyObject *func_name;
    /* 属性字典，一般为空 */
    PyObject *func_dict;
    /* 弱引用列表 */
    PyObject *func_weakreflist;
    /* 函数所在的模块 */
    PyObject *func_module;
    /* 函数参数的类型注解，一个字典或者空 */
    PyObject *func_annotations;
    /* 函数的全限定名，我们后面会说它和 func_name 之间的区别 */
    PyObject *func_qualname;
    // PyFunctionObject对象是一个 Python 函数，但在底层它也是由某个类实例化得到的
    // 所以调用 Python 函数的时候，会执行类型对象 (PyFunction_Type) 的 tp_call 方法
    // 但函数比较特殊，我们创建它就是为了调用的，所以为了优化调用效率，引入了 vectorcall
    vectorcallfunc vectorcall;
} PyFunctionObject;
```



古明地觉的 Python小屋

我们来实际获取一下这些成员，看看它们在 Python 中是如何表现的。

### 1) func\_code: 函数的字节码

```
1 def foo(a, b, c):
2     pass
3
4 code = foo.__code__
5 print(code) # <code object foo at .....>
6 print(code.co_varnames) # ('a', 'b', 'c')
```

## 2) func\_globals: global名字空间

```
1 def foo(a, b, c):
2     pass
3
4 name = "古明地觉"
5 print(foo.__globals__) # {..., 'name': '古明地觉'}
6 # 拿到的其实就是外部的 global 名字空间
7 print(foo.__globals__ == globals()) # True
```

## 3) func\_defaults: 函数参数的默认值

```
1 def foo(name="古明地觉", age=16):
2     pass
3
4 # 打印的是默认值
5 print(foo.__defaults__) # ('古明地觉', 16)
6
7
8 def bar():
9     pass
10
11 # 没有默认值的话, __defaults__ 为 None
12 print(bar.__defaults__) # None
```

## 4) func\_kwdefaults: 只能通过关键字参数传递的 "参数" 和 "该参数的默认值" 组成的字典

```
1 def foo(name="古明地觉", age=16):
2     pass
3
4 # 打印为 None, 这是因为虽然有默认值
5 # 但并不要求必须通过关键字参数的方式传递
6 print(foo.__kwdefaults__) # None
7
8
9 def bar(*, name="古明地觉", age=16):
10     pass
11
12 print(bar.__kwdefaults__) # {'name': '古明地觉', 'age': 16}
```

如果在前面加上一个 \*, 就表示后面的参数必须通过关键字的方式传递。因为如果不通过关键字参数传递的话, 那么无论多少个位置参数都会被 \* 接收, 无论如何也不可能传递给 name、age。

我们经常会看到 \*args, 这是因为我们需要函数调用时传递过来的值, 而通过 args 能以元组的形式来拿到这些值。但是这里我们不需要, 我们只是希望后面的参数必须通过关键字参数传递, 因此前面写一个 \* 即可, 当然写 \*args 也是可以的。

## 5) func\_closure: 闭包对象

```
1 def foo():
2     name = "古明地觉"
3     age = 16
4
5     def bar():
6         nonlocal name
7         nonlocal age
8
9     return bar
```

```

10
11 # 查看的是闭包里面 nonlocal 的值, 由于有两个 nonlocal
12 # 所以foo().__closure__ 是一个包含两个元素的元组
13 print(foo().__closure__)
14 """
15 (<cell at 0x000001FD1D3B02B0: int object at 0x00007FFDE559D660>,
16  <cell at 0x000001FD1D42E310: str object at 0x000001FD1D3DA090>)
17 """
18
19 print(foo().__closure__[0].cell_contents) # 16
20 print(foo().__closure__[1].cell_contents) # 古明地觉

```

注意：查看闭包属性我们使用的是内层函数，不是外层的 foo。

## 6) func\_doc: 函数的 doc

```

1 def foo():
2     """
3     hi, 欢迎来到我的小屋
4     遇见你真好
5     """
6     pass
7
8 print(foo.__doc__)
9 """
10
11     hi, 欢迎来到我的小屋
12     遇见你真好
13
14 """

```

## 6) func\_name: 函数的名字

```

1 def foo(name, age):
2     pass
3
4 print(foo.__name__) # foo

```

当然不光是函数，方法、类、模块都有自己的名字，

```

1 import numpy as np
2
3 print(np.__name__) # numpy
4 print(np.ndarray.__name__) # ndarray
5 print(np.array([1, 2, 3]).transpose.__name__) # transpose

```

## 7) func\_dict: 函数的属性字典

因为函数在底层也是由一个类实例化得到的，所以它可以有自己的属性字典，只不过这个字典一般为空。

```

1 def foo(name, age):
2     pass
3
4 print(foo.__dict__) # {}

```

当然啦，我们也可以整点骚操作：

```

def foo(name, age):
    pass

code = """

```

```

code = """
name, age = "古明地觉", 16
def foo():
    return "satori"
"""

exec(code, foo.__dict__)

print(foo.name, foo.age) # 古明地觉 16
print(foo.foo()) # satori

```

古明地觉的 Python 小屋

所以虽然叫函数，但它也是由某个类型对象实现的。

## 8) func\_weakreflist: 弱引用列表

Python 无法获取这个属性，底层没有提供相应的接口，关于弱引用此处就不深入讨论了。

## 9) func\_module: 函数所在的模块

```

1 def foo(name, age):
2     pass
3
4 print(foo.__module__) # __main__

```

类、方法、协程也有 `__module__` 属性。

## 10) func\_annotations: 类型注解

```

1 def foo(name: str, age: int):
2     pass
3
4 # Python3.5 新增的语法, 但只能用于函数参数
5 # 而在 3.6 的时候, 声明变量也可以使用这种方式
6 # 特别是当 IDE 无法得知返回值类型时, 便可通过类型注解的方式告知 IDE
7 # 这样就能使用 IDE 的智能提示了
8 print(foo.__annotations__)
9 # {'name': <class 'str'>, 'age': <class 'int'>}

```

## 11) func\_qualname: 全限定名

```

1 def foo():
2     pass
3 print(foo.__name__, foo.__qualname__) # foo foo
4
5
6 class A:
7
8     def foo(self):
9         pass
10 print(A.foo.__name__, A.foo.__qualname__) # foo A.foo

```



小结

以上就是函数的底层结构，在Python里面是由 `function` 实例化得到的。

```
1 def foo(name, age):
2     pass
3
4 # <class 'function'> 就是 C 里面的 PyFunction_Type
5 print(foo.__class__) # <class 'function'>
```

但是这个类底层没有暴露给我们，我们不能直接用，因为函数通过 `def` 创建即可，不需要通过类型对象来创建。

后续会介绍更多关于相关的知识。

收录于合集 #CPython 97

< 上一篇

《源码探秘 CPython》57. 函数是怎么创建的？

下一篇 >

《源码探秘 CPython》55. 虚拟机是如何捕获异常的？

喜欢此内容的人还喜欢

从零开始学 Python 之高阶函数

豆豆的杂货铺

×



从零开始学 Python 之递归函数

豆豆的杂货铺

×



【C语言】练习题 - 函数名重载的名字装饰

一起上编程课

×

