

24 函数对象诞生记-慕课专栏

 imooc.com/read/76/article/1920

函数，作为计算机程序中**抽象执行流程**的基本单位，在**功能分解**、**代码复用**等方面发挥着至关重要的作用。*Python* 中的函数，相信你一定不会陌生：

```
>>> pi = 3.14
>>> def circle_area(r):
...     return pi * r ** 2
...
>>> circle_area(2)
12.56
```

这段代码将圆面积计算功能组织成一个函数 *circle_area*，圆半径 *r* 以参数形式作为输入，函数负责计算面积，并将结果作为返回值输出。这样一来，任何需要计算圆面积的地方，只需要调用 *circle_area* 即可，达到了功能分解以及代码复用的目的。

我们知道 *Python* 中一切都是对象，函数也是一种对象。那么，作为一等对象的函数，到底长什么模样，有什么特殊行为呢？*Python* 代码又是如何一步步变身为函数对象的呢？洞悉函数秘密后，可以实现哪些有趣的功能呢？带着这些疑问，我们开始探索函数对象。

函数对象长啥样

首先，借助内建函数 *dir* 观察函数对象，发现了不少新属性：

```
>>> dir(circle_area)
['_annotations_', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__',
 '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__']
```

通过名字，我们可以猜测出某些字段的功能。*__code__* 应该是一个**代码对象**，里面保存着函数的**字节码**。从字节码中，我们可以清晰地读懂函数的执行逻辑，跟 *Python* 语句表达的意思一模一样：

```
>>> circle_area.__code__
<code object circle_area at 0x10d52d270, file "<stdin>", line 1>
>>> import dis
>>> dis.dis(circle_area.__code__)
2       0 LOAD_GLOBAL           0 (pi)
        2 LOAD_FAST              0 (r)
        4 LOAD_CONST             1 (2)
        6 BINARY_POWER
        8 BINARY_MULTIPLY
       10 RETURN_VALUE
```

又如，*__globals__* 顾名思义应该是函数的全局名字空间，全局变量 *pi* 的藏身之地。

```
>>> circle_area.__globals__
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, 'pi': 3.14, 'circle_area': <function circle_area at 0x10d573950>, 'dis': <module
'dis' from
'/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/lib/python3.7/dis.py'>}
```

确实如此，*pi* 的出现证实了我们的猜测。由于 *pi* 是在 `__main__` 模块中定义的，保存在模块的属性空间内。那么，`__globals__` 到底是模块属性空间本身，还是它的一个拷贝呢？我们接着观察：

```
>>> circle_area.__globals__ is sys.modules['__main__'].__dict__
True
```

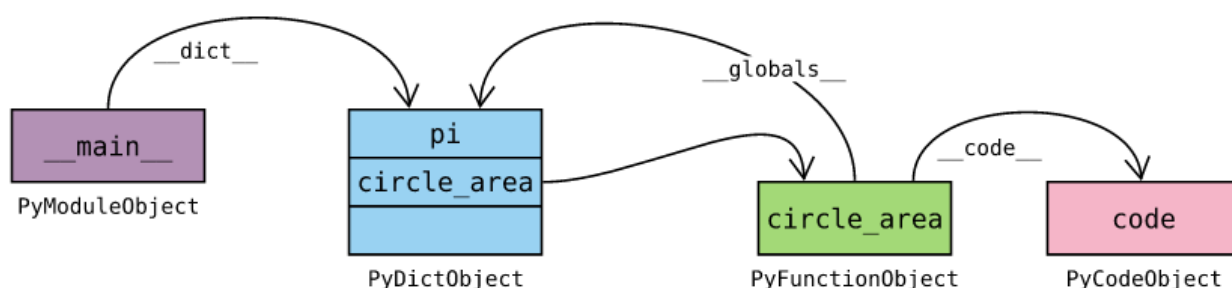
注意到，`sys.modules` 保存着 Python 当前所有已导入的模块对象，包括 `__main__` 模块。我们取出 `__main__` 模块的属性空间 `__dict__` 与函数全局名字空间对比，发现他们是同一个 `dict` 对象。原来函数全局名字空间和模块属性空间就是这样紧密绑定在一起的！

此外，我们还可以找到函数名以及所属模块名这两个字段：

```
>>> circle_area.__name__
'circle_area'
>>> circle_area.__module__
'__main__'
```

函数对象关键属性整理如下：

属性	描述
<code>code</code>	代码对象，包含函数字节码
<code>globals</code>	函数全局名字空间
<code>module</code>	函数所属模块名
<code>name</code>	函数名字



函数对象如何创建

我们已经初步看清 **函数** 对象的模样，它和 **代码** 对象关系密切，**全局名字空间** 就是它所在 **模块** 对象的 **属性空间**。那么，*Python* 又是如何完成从代码到函数对象的转变的呢？想了解这其中的秘密，还是得从字节码入手。

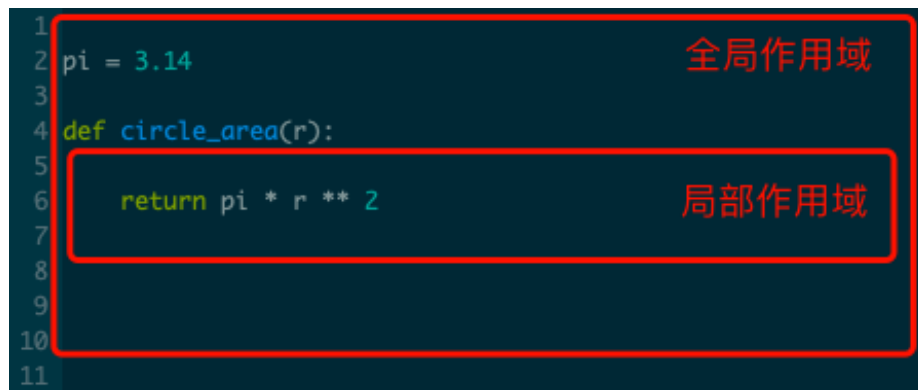
现在我们要想方设法搞到定义函数的字节码，先将函数代码作为文本保存起来：

```
>>> text = '''
... pi = 3.14
... def circle_area(r):
...     return pi * r ** 2
... '''
```

然后，调用 `compile` 函数编译函数代码，得到一个 **代码** 对象：

```
>>> code = compile(text, 'test', 'exec')
```

根据 **虚拟机** 部分的学习，我们知道 **作用域** 与 **代码** 对象之间的一一对应的关系。定义函数的这段代码虽然简短，里面却包含了两个不同的 **作用域**：一个是模块级别的 **全局作用域**，一个函数内部的 **局部作用域**：

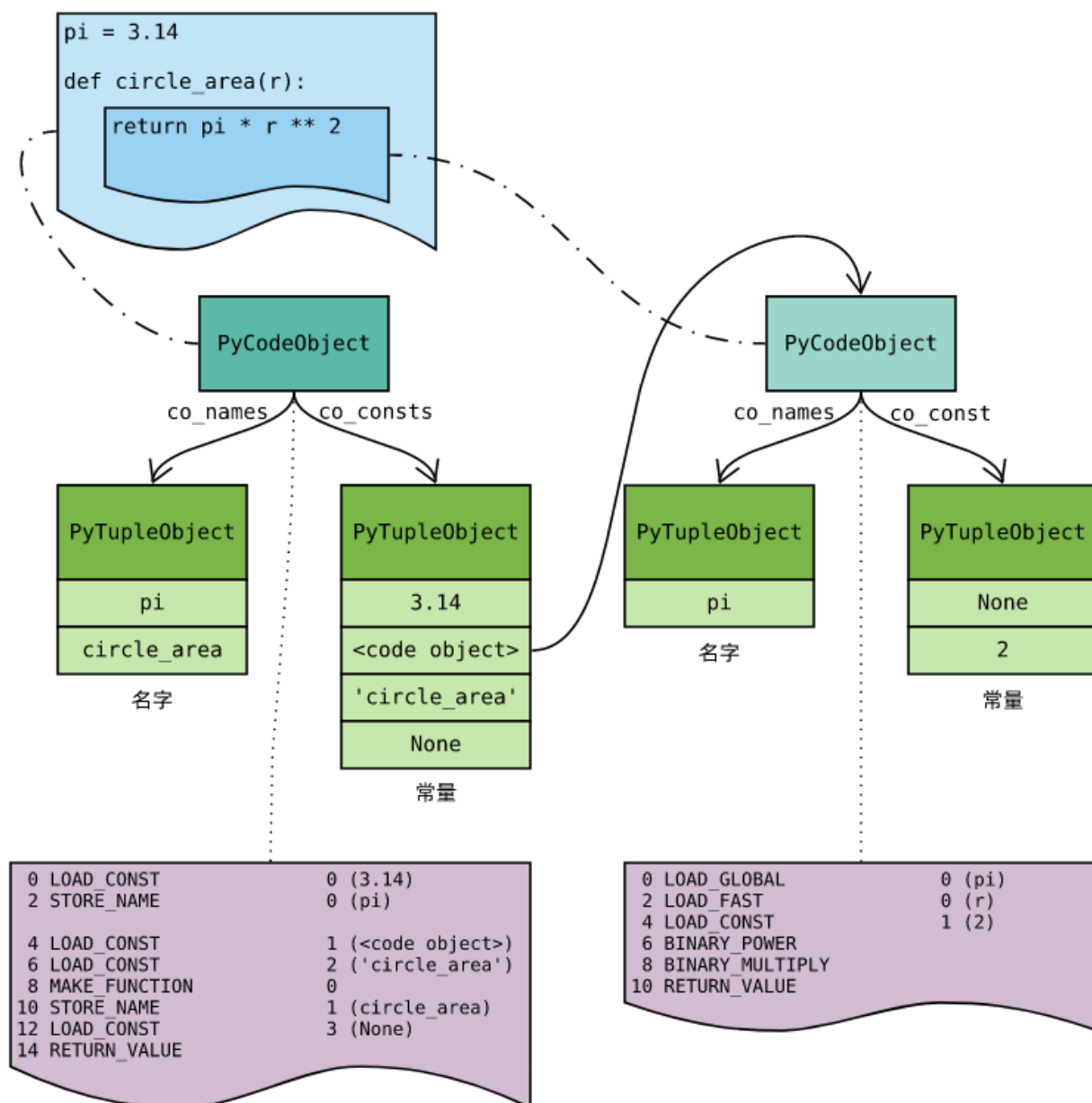


```
1
2 pi = 3.14                                全局作用域
3
4 def circle_area(r):
5     return pi * r ** 2                  局部作用域
6
7
8
9
10
11
```

那么，为啥 `compile` 函数只返回一个代码对象呢？因为局部代码对象作为一个 **常量**，藏身于全局代码对象中。而 `compile` 函数则只需返回全局代码对象：

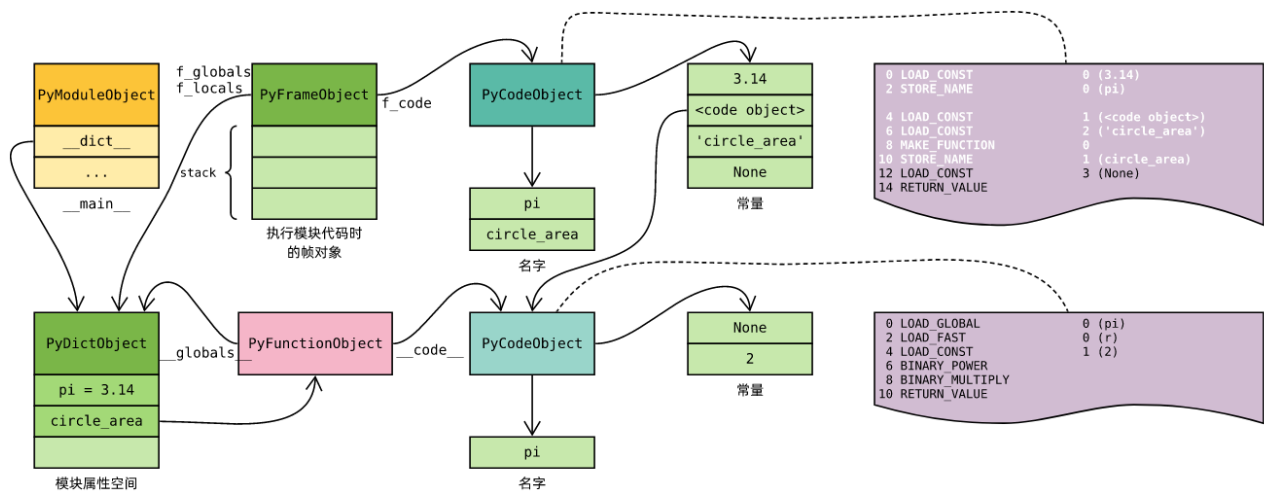
```
>>> code.co_names
('pi', 'circle_area')
>>> code.co_consts
(3.14, <code object circle_area at 0x10e179420, file "test", line 3>, 'circle_area', None)

>>> code.co_consts[1]
<code object circle_area at 0x10e179420, file "test", line 3>
>>> code.co_consts[1].co_names
('pi',)
>>> code.co_consts[1].co_consts
(None, 2)
```



看来，函数对象诞生的秘密就藏在 `MAKE_FUNCTION` 指令中。

开始深入源码研究 `MAKE_FUNCTION` 指令前，我们先推演一遍虚拟机执行这段字节码的全过程。假设 `circle_area` 在 `__main__` 模块中定义，全局代码对象则作为模块代码执行，以模块 **属性空间** 为 **全局名字空间** 和 **局部名字空间**。前两行字节码与函数创建无关，在将 `3.14` 作为 `pi` 值保存到 **局部名字空间**，它也是模块的 **属性空间**：



注意到，这个 **局部名字空间** 正好是模块对象的 **属性空间**！如果函数是在模块 `demo` 中定义，则可以这样引用：

```
>>> import demo
>>> demo.circle_area(2)
```

至此，函数诞生的整个历程我们已经尽在掌握，可以腾出手来研究 `MAKE_FUNCTION` 这个字节码了。

MAKE_FUNCTION

经过 **虚拟机** 部分学习，我们对研究字节码的套路早已了然于胸。虚拟机处理字节码的逻辑位于 `Python/ceval.c`：

```

TARGET(MAKE_FUNCTION) {
    PyObject *qualname = POP();
    PyObject *codeobj = POP();
    PyFunctionObject *func = (PyFunctionObject *)
        PyFunction_NewWithQualName(codeobj, f->f_globals, qualname);

    Py_DECREF(codeobj);
    Py_DECREF(qualname);
    if (func == NULL) {
        goto error;
    }

    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();
    }

    PUSH((PyObject *)func);
    DISPATCH();
}

```

1. 第 2-3 行，从栈顶弹出关键参数；
2. 第 4-5 行，调用 `PyFunction_NewWithQualName` 创建 **函数** 对象，**全局名字空间** 来源于当前 **帧** 对象；
3. 第 13-16 行，如果函数为 **闭包函数**，从栈顶取 **闭包变量**；
4. 第 17-20 行，如果函数包含注解，从栈顶取注解；
5. 第 21-28 行，如果函数参数由默认值，从栈顶取默认值，分为普通默认值以及非关键字默认值两种；

`PyFunction_NewWithQualName` 函数在 `Objects/funcobject.c` 源文件中实现，主要参数有 3 个：

- `code`，**代码对象**；
- `globals`，**全局名字空间**；
- `qualname`，**函数名**；

`PyFunction_NewWithQualName` 函数则实例化 **函数** 对象 (`PyFunctionObject`)，并根据参数初始化相关字段。

当然了，我们也可以用 `Python` 语言模拟这个过程。根据 **对象模型** 中规则，调用 **类型** 对象即

可创建 **实例** 对象。只是 Python 并没有暴露 **函数类型** 对象，好在它不难找：

```
>>> def a():
...     pass
...
>>> function = a.__class__
>>> function
<class 'function'>
```

我们随便定义了一个函数，然后通过 `__class__` 找到它的 **类型** 对象，即 **函数类型** 对象。

然后，我们准备函数的 **代码** 对象：

```
>>> text = '''
... def circle_area(r):
...     return pi * r ** 2
... '''
>>> code = compile(text, 'test', 'exec')
>>> func_code = code.co_consts[0]
>>> func_code
<code object circle_area at 0x10e029150, file "test", line 2>
```

由此一来，函数三要素便已俱备，调用 **函数类型** 对象即可完成临门一脚：

```
>>> circle_area = function(func_code, globals(), 'circle_area')
>>> circle_area
<function circle_area at 0x10e070620>
```

至此，我们得到了梦寐以求的 **函数** 对象，而且是以一种全新的方式！

但是，我们把全局变量 `pi` 忘在脑后了，没有它函数跑不起来：

```
>>> circle_area(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "test", line 3, in circle_area
NameError: name 'pi' is not defined
```

这难不倒我们，加上便是：

```
>>> pi = 3.14
>>> circle_area(2)
12.56
```

成功了！不仅如此，我们还可以为函数加上参数默认值：

```
>>> circle_area.__defaults__ = (1,)
>>> circle_area()
3.14
>>> circle_area(3)
28.26
>>> circle_area(1)
3.14
```


由此一来，如果调用 `circle_area` 函数时未指定参数，则默认以 1 为参数。

