《源码探秘 CPython》72. 自定义类对象的底层实现与metaclass（上）

原创 古明地觉 古明地觉的编程教室 2022-04-20 08:30

收录于合集
#CPython
97个 >

楔子

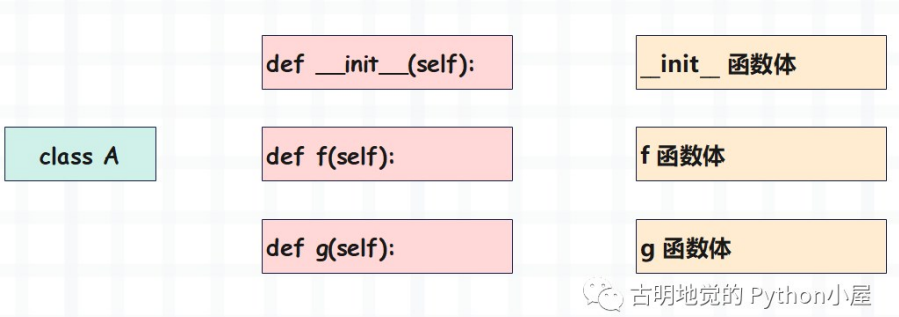
Python 除了提供很多内置的类之外，还支持我们定义属于自己的类，那么底层是如何做的呢？下面就来看看。

老规矩，如果想知道底层是怎么做的，那么就必须要通过观察字节码来实现。这里我们随便定义一个类，然后反编译一下：

```
class Girl:
    name = "古明地觉"

    def __init__(self):
        print(f"__init__: {self.name}")

    def f(self):
        print("f")

    def g(self, name):
        self.name = name
        print(self.name)

girl = Girl()
girl.f()
girl.g("古明地恋")
"""
__init__: 古明地觉
f
古明地恋
"""
```

通过之前对函数机制的分析，我们知道对于一个包含函数定义的 Python 源文件，在编译之后会得到一个和源文件对应的PyCodeObject对象，其内部的常量池中存储了和函数对应的PyCodeObject对象。那么对于包含类的Python源文件，编译之后的结果又是怎么样的呢？

显然我们可以照葫芦画瓢，根据以前的经验我们可以猜测模块对应的PyCodeObject对象的常量池中肯定存储了类对应的PyCodeObject对象，类对应的PyCodeObject对象的常量池中则存储了__init__、f、g 三个函数对应的PyCodeObject对象。然而事实也确实如此。



在介绍函数的时候，我们看到函数的声明(def语句)和函数的实现虽然在逻辑上是一个整体，但它们的字节码指令却是分离在两个PyCodeObject对象中的。

在类中，同样存在这样的分离现象。声明类的 class 语句，编译后的字节码指令存储在模块对应的

PyCodeObject中；而类的实现、也就是类里面的逻辑，编译后的字节码指令则存储在类对应的PyCodeObject中。所以我们在模块级别中只能找到类，无法直接找到类里面的成员。

另外还可以看到，类的成员函数和一般的函数相同，也会有这种声明和实现分离的现象。正所谓函数即变量，类也是如此，def、class 本质上都是定义一个变量，该变量指向具体的 PyFunctionObject 或者 PyTypeObject。

```
s = """
class Girl:
    name = "古明地觉"

    def __init__(self):
        print(f"__init__: {self.name}")

    def f(self):
        print("f")

    def g(self, name):
        self.name = name
        print(self.name)
"""

# 此时的code显然是模块对应的PyCodeObject对象
code = compile(s, "<file>", "exec")
print(code)  # <code object <module> at 0x00...>

# 常量池里面存储了 Girl 对应的 PyCodeObject 对象
print(code.co_consts[0])  # <code object Girl at 0x00...>

# Girl的PyCodeObject对象的常量池里面存储了几个函数的PyCodeObject对象
# 比如 Girl.g 的 PyCodeObject
print(code.co_consts[0].co_consts[6])  # <code object g at 0x00...>
print(
    code.co_consts[0].co_consts[6].co_varnames
)  # ('self', 'name')
```


相信这些内容已经没有什么难度了，总之函数、类在编译之后都会对应一个 PyCodeObject。由于函数、类可以嵌套，那么 PyCodeObject 也是可以嵌套的，并且也会作为一个常量被收集起来，存储在外层的 PyCodeObject 的常量池当中。

自定义类对象的动态元信息

自定义类对象的元信息指的就是关于这个类的信息描述，比如名称、所拥有的的属性、方法，该类实例化时要为实例对象申请的内存空间大小等。有了这些元信息，才能创建自定义类对象，否则我们是没办法创建的。

注意：元信息是一个非常重要的概念，在很多框架中都会出现。比如说 Hive，数据的元信息就是存储在 MySQL 里面。而在编程语言中，也正是通过元信息才实现了反射等动态特性，尤其是 Python，将元信息的概念发挥地淋漓尽致，因此 Python 也提供了其它编程语言所不具备的高度灵活的动态特征。

我们将类简化一下，看看它的字节码长什么样子。

```
s = """
class Girl:

    def f(self):
        print("我是 f")

    def g(self):
```

```
        print("我是 g")
"""

code = compile(s, "<file>", "exec")

# 在查看字节码之前，先看看常量池
for const in code.co_consts:
    print(const)
"""
<code object Girl at 0x00000......>
Girl
None
"""

for const in code.co_consts[0].co_consts:
    print(const)
"""
Girl
<code object f at 0x00000......>
Girl.f
<code object g at 0x00000......>
Girl.g
None
"""
```

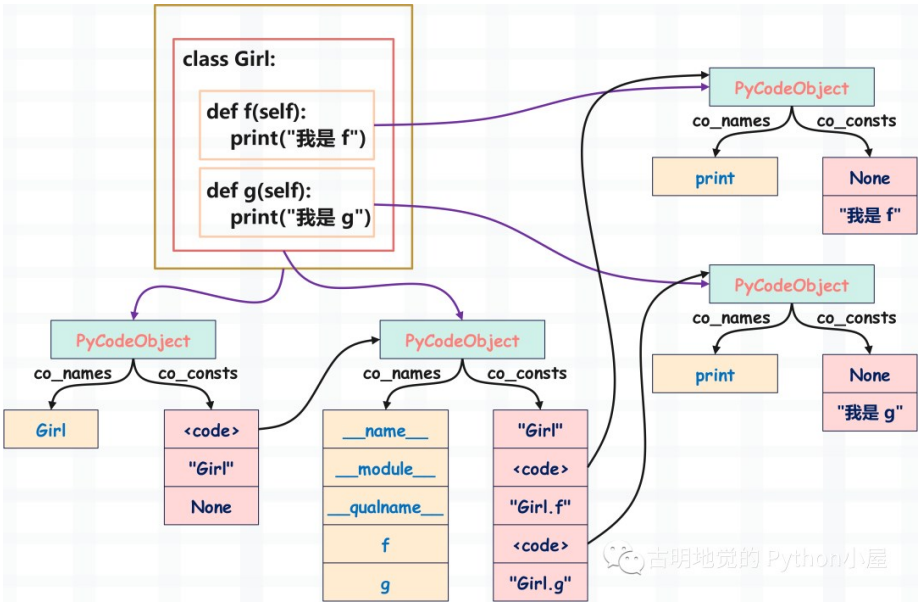

观察一下类的常量池，第一个元素显然是类名，一个字符串；第二和第三个元素则是函数 f 对应的 PyCodeObject 以及全限定名；第四和第五个元素则是函数 g 对应的 PyCodeObject 以及全限定名；最后一个是 None，当然这个 None 是一定会有的。

而字节码如下：

```
  # 模块对应的字节码
  0 LOAD_BUILD_CLASS
  2 LOAD_CONST               0 (<code object Girl at 0x0000......>)
  4 LOAD_CONST               1 ('Girl')
  6 MAKE_FUNCTION            0
  8 LOAD_CONST               1 ('Girl')
 10 CALL_FUNCTION            2
 12 STORE_NAME               0 (Girl)
 14 LOAD_CONST               2 (None)
 16 RETURN_VALUE

  # class Girl 对应的字节码
Disassembly of <code object Girl at 0x0000......>:
  0 LOAD_NAME                0 (__name__)
  2 STORE_NAME               1 (__module__)
  4 LOAD_CONST               0 ('Girl')
  6 STORE_NAME               2 (__qualname__)

  8 LOAD_CONST               1 (<code object f at 0x0000......>)
 10 LOAD_CONST               2 ('Girl.f')
 12 MAKE_FUNCTION            0
 14 STORE_NAME               3 (f)

 16 LOAD_CONST               3 (<code object g at 0x0000......>)
 18 LOAD_CONST               4 ('Girl.g')
 20 MAKE_FUNCTION            0
 22 STORE_NAME               4 (g)
 24 LOAD_CONST               5 (None)
 26 RETURN_VALUE

  # Girl.f 对应的字节码
Disassembly of <code object f at 0x0000......>:
  0 LOAD_GLOBAL              0 (print)
  2 LOAD_CONST               1 ('我是 f')
```

```
  4 CALL_FUNCTION            1
  6 POP_TOP
  8 LOAD_CONST               0 (None)
 10 RETURN_VALUE

  # Girl.g 对应的字节码
Disassembly of <code object g at 0x0000......>:
  0 LOAD_GLOBAL              0 (print)
  2 LOAD_CONST               1 ('我是 g')
  4 CALL_FUNCTION            1
  6 POP_TOP
  8 LOAD_CONST               0 (None)
 10 RETURN_VALUE
```


结构很清晰，总共 4 个 PyCodeObject，分别对应模块、类 Girl、函数 Girl.f、函数 Girl.g。



下面我们来对字节码逐一分析，首先是模块的字节码：

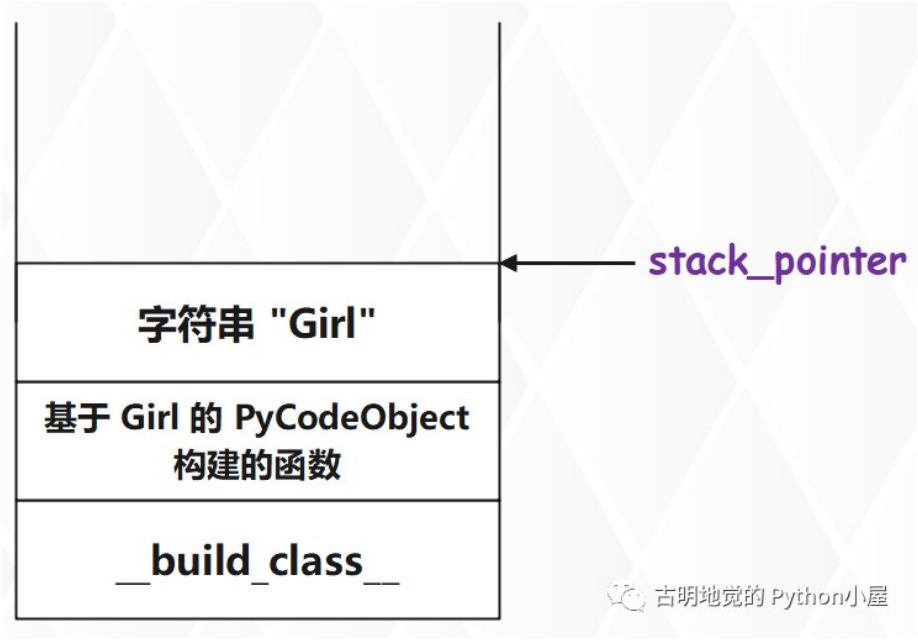
```
#一条新指令，会将内置函数 __build_class__ 压入栈中
#至于这个 __build_class__ 是干啥的，一会说
0 LOAD_BUILD_CLASS
#加载Girl对应的PyCodeObject对象；
2 LOAD_CONST               0 (<code object Girl at 0x0000......>)
#加载字符串"Girl"
4 LOAD_CONST               1 ('Girl')
#问题来了，我们看到是MAKE_FUNCTION
#不是说要构建类吗？为什么是MAKE_FUNCTION呢？
#别急，往下看
6 MAKE_FUNCTION            0
#再次加载字符串"Girl"
8 LOAD_CONST               1 ('Girl')
#以构建的PyFunctionObject和字符串 "Girl" 为参数
#调用 __build_class__，创建一个类
10 CALL_FUNCTION           2
#将创建的类使用变量 Girl 进行保存
12 STORE_NAME              0 (Girl)
 # return None
14 LOAD_CONST              2 (None)
16 RETURN_VALUE
```

关键指令是 LOAD_BUILD_CLASS，它的逻辑很简单，就是将内置函数 __build_class__ 压入运行时栈。

紧接着通过两个 LOAD_CONST 将 Girl 的 PyCodeObject 和字符串 "Girl" 压入栈中，再用 MAKE_FUNCTION 将其弹出，构造出一个 PyFunctionObject，将其指针压入栈中。此时栈里面还剩下两个元素，也就是**刚入栈的函数**和**内置函数 __build_class__**。而这个刚入栈的函数，就是基于 Girl 的 PyCodeObject 构建的。

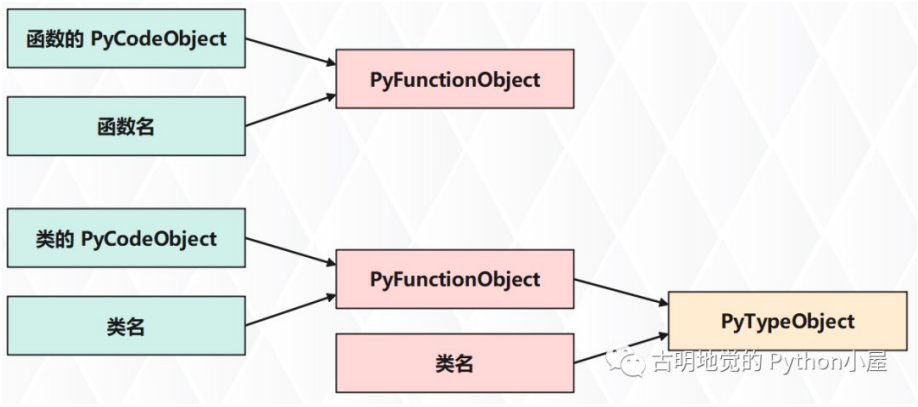
不过还是那个问题，Girl 明明是个类，为啥要 MAKE_FUNCTION 呢？接下来的两条指令会告诉你答案。

构建完函数之后又通过 LOAD_CONST 将字符串 "Girl" 压入栈中，显然它代表类名。而此时栈里面有三个元素：



然后就是一个 CALL_FUNCTION，指令参数是 2。不用想，肯定是以构建的函数和字符串"Girl" 为参数，调用 __build_class__。而 __build_class__ 会创建一个类并返回，然后压入运行时栈，最后再通过 STORE_NAME 将创建的类对象使用变量 Girl 保存。

所以类不是上来就构建的，根据 PyCodeObject 和名称构造出来的实际上是一个 PyFunctionObject，尽管使用的是类的 PyCodeObject。当 PyFunctionObject 构造完毕时，再在其之上构造 PyTypeObject，而这一步由 __build_class__ 负责。



所以，可以得出如下结论：

```
class A(object):
    pass
# 在底层将会被翻译成
A = __build_class__(<PyFunctionObject A>, "A")


# 如果是
class A(int):
    pass
# 在底层将会被翻译成
A = __build_class__(<PyFunctionObject A>, "A", int)
```

我们实际操作一下：

```
MyInt = __build_class__(lambda: None, "MyInt", int)
```

```
2
3 print(MyInt)  # <class '__main__.MyInt'>
4 print(MyInt.__base__)  # <class 'int'>
5 print(MyInt(3) + 5)  # 8
```

如果参数类型不正确的话，就会报出如下错误：

```
1  try:
2      __build_class__()
3  except TypeError as e:
4      print(e)
5  """
6  __build_class__: not enough arguments
7  """
8
9  try:
10     # 第一个参数 func 必须是函数
11     __build_class__("", "")
12 except TypeError as e:
13     print(e)
14 """
15 __build_class__: func must be a function
16 """
17
18 try:
19     # 第二个参数 name 必须是字符串
20     __build_class__(lambda: None, 123)
21 except TypeError as e:
22     print(e)
23 """
24 __build_class__: name is not a string
25 """
```

记住这几个报错信息，后面会看到。此外我们也能看出，__build_class__ 的一个参数叫 func、第二个参数叫 name。

所以__build_class__就是用来将一个函数对象变成一个类对象。

再来看看类对象的字节码：

```
1 Disassembly of <code object Girl at 0x0000......>:
2   #将模块的名字压入栈中
3   0 LOAD_NAME                0 (__name__)
4   #使用类的 __module__ 进行保存
5   #所以通过类的 __module__，能找到该类属于哪一个模块
6   2 STORE_NAME               1 (__module__)
```

```
  #加载字符串 "Girl"
  4 LOAD_CONST               0 ('Girl')
  #作为类的全限定名
  6 STORE_NAME               2 (__qualname__)

  #加载函数 f 的 PyCodeObject 和字符串 "Girl.f"
  8 LOAD_CONST               1 (<code object f at 0x0000......>)
 10 LOAD_CONST               2 ('Girl.f')
  #构造函数
 12 MAKE_FUNCTION            0
  #使用变量 f 保存
 14 STORE_NAME               3 (f)

  # 和上面构造 f 类似
 16 LOAD_CONST               3 (<code object g at 0x0000......>)
 18 LOAD_CONST               4 ('Girl.g')
 20 MAKE_FUNCTION            0
 22 STORE_NAME               4 (g)
 24 LOAD_CONST               5 (None)
 26 RETURN_VALUE
```

我们在介绍函数的时候提过："函数的局部变量是不可变的，在编译的时候就已经确定了，是以一种静态的方式存放在f_localsplus中，f_locals初始是一个NULL，函数里面的局部变量是通过静态的方式来访问的"。

但是类则不一样，类是可以动态修改的，可以随时增加属性、方法，这就意味着类是不可能通过静态方式来查找属性的。事实上也确实如此，类也有一个 f_locals，而对于类来说，变量是从 f_locals 中查找的。

```python
class Girl:

    def f(self):
        print("我是 f")

    def g(self):
        print("我是 g")

print(__name__)  # __main__
print(Girl.__module__)  # __main__
print(Girl.__qualname__)  # Girl
print(Girl.f is Girl.__dict__["f"])  # True
```

所以整体过程就是：先将PyCodeObject构建成函数，再通过__build_class__将函数变成一个类，当__build_class__结束之后我们的自定义类就破茧而出了。

因此剩下的问题就是__build_class__是如何将一个函数变成类的，想要知道答案，那么只能去源码中一探究竟了。不过在看源码之前，我们还需要了解一样东西：metaclass。

元类，被誉为是深度的魔法，但是个人觉得有点夸张了。首先元类是做什么的，它是用来控制我们自定义类的生成过程的，默认情况下，我们自定义的类都是由 type 创建的。但是我们可以手动指定某个类的元类，但是在介绍元类之前，我们还需要看一下Python的两个特殊的魔法方法：__new__和__init__。

类在实例化的时候会自动调用__init__，但其实在调用__init__之前会先调用__new__。

- __new__：为实例对象申请一片内存；
- __init__：为实例对象设置属性；

```
class A:

    def __new__(cls, *args, **kwargs):
        print("__new__")

    def __init__(self):
        print("__init__")
A()
"""
__new__
"""
```

然而我们看到只有__new__被调用了，__init__则没有。原因就在于__new__里面必须将A的实例对象返回，才会执行__init__，并且执行的时候会自动将__new__的返回值作为参数传给__init__当中的self。

```
class A:

    def __new__(cls, *args, **kwargs):
        print("__new__")
        # 这里的参数cls就表示A这个类本身
        # object.__new__(cls) 便是根据cls创建cls的实例对象
        return object.__new__(cls)

    def __init__(self):
        # 然后执行__init__，里面的self指的就是实例对象
        # 执行__init__时，__new__的返回值会自动作为参数传递给self
        print("__init__")

A()
"""
__new__
__init__
"""
```

所以一个对象是什么，取决于其类型对象的__new__返回了什么。

```
class A:

    def __new__(cls, *args, **kwargs):
        print("__new__")
        # 这里必须返回A的实例对象，否则__init__函数是不会执行的
        return 123

    def __init__(self):
        print("__init__")


a = A()
print(a + 1)
"""
__new__
124
"""
```

我们看到A在实例化之后得到的是一个整数，原因就是__new__返回了123。

创建类的另一种方式

创建类的时候除了通过class关键字之外，我们还可以使用type这个古老却又强大的类来创建。

```
#type这个类里面可以接收一个参数或者三个参数
#如果接收一个参数，那么表示查看类型;
#如果接收三个参数，那么表示创建一个类

try:
    A = type("A", "")
except Exception as e:
    print(e)  # type() takes 1 or 3 arguments
```

查看类型就不说了，下面看看如何用 type 创建一个类：

```
# type接收的三个参数：类名、继承的基类、属性
class A(list):
    name = "古明地觉"

# 上面这个类翻译过来就是
A = type("A", (list, ), {"name": "古明地觉"})
print(A)  # <class '__main__.A'>
print(A.__name__)  # A
print(A.__base__)  # <class 'list'>
print(A.name)  # 古明地觉
```

所以还是很简单的，我们还可以自定义一个类继承自 type。

```
class MyType(type):

    def __new__(mcs, name, bases, attr):
        print(name)
        print(bases)
        print(attr)

#指定 metaclass，表示A这个类由MyType创建
#我们说__new__是为实例对象开辟内存的
#那么MyType的实例对象是谁呢？显然就是这里的A
#因为A指定了metaclass为MyType，所以A的类型就是MyType
class A(int, object, metaclass=MyType):
    name = "古明地觉"
"""
A
(<class 'int'>, <class 'object'>)
{'__module__': '__main__', '__qualname__': 'A', 'name': '古明地觉'}
"""

# 我们看到一个类在创建的时候会向元类的__new__中传递三个值
# 分别是类名、继承的基类、类的属性
# 但此时A并没有被创建出来
print(A)  # None
```

我们说__new__一定要将创建的实例对象返回才可以，这里的MyType是元类。所以类对象A就是MyType的实例对象，MyType的__new__就负责为类对象A分配空间。但是显然我们这里并没有分配，而且返回的还是一个None，如果我们返回的是 123，那么print(A)就是123。

```
class MyType(type):

    def __new__(mcs, name, bases, attr):
        return []

class A(metaclass=MyType):
    pass
```

```
8
9  # A 是由 MyType 生成的, MyType 返回的是 []
10 # 因此 A 就是 []
11 print(A)  # []
```

所以元类和类之间的关系 与 类和实例对象的关系，之间是很相似的，因为完全可以把类对象看成是元类的实例对象。因此A既然指定了metaclass为MyType，就表示A这个类由MyType创建，那么MyType的__new__函数返回了什么，A就是什么。

```
1  class MyType(type):
2
3      def __new__(mcs, name, bases, attr):
4          return "嘿嘿嘿"
5
6  class A(metaclass=MyType):
7      pass
8
9  print(A + "哟哟哟")  # 嘿嘿嘿哟哟哟
```

这便是Python语言具备的高度动态特性，那么问题来了，如果我想把A创建出来、像普通的类一样使用的话，该咋办呢？因为默认情况下是由type创建，底层帮你做好了，但现在是我们手动指定元类，那么一切就需要我们来手动指定了。

显然，这里创建还是要依赖于type，只不过需要我们手动指定，而且在手动指定的同时还可以增加一些我们自己的操作。

```
class MyType(type):

    def __new__(mcs, name, bases, attr):
        name = name * 2
        bases = (list,)
        attr.update({"name": "古明地觉", "nickname": "小五萝莉"})

        # 这里直接交给type即可，然后type来负责创建
        # 所以super().__new__实际上会调用type.__new__
        # type(name, bases, attr) 等价于 type.__new__(type, name, bases, attr)
        return super().__new__(mcs, name, bases, attr)
        # 但是这里我们将__new__的第一个参数换成了mcs，也就是这里的MyType
        # 等价于type.__new__(mcs, name, bases, attr)，表示将元类设置成MyType
        # 注意：不能写type(name, bases, attr)，因为这样的话类还是由type创建的

class Girl(metaclass=MyType):
    pass

# 我们看到类的名字变了，默认情况下是Girl
# 但是我们在创建的时候将name乘了个2
print(Girl.__name__)  # GirlGirl

# 那么显然Girl这里也继承自list
print(Girl("你好呀"))  # ['你', '好', '呀']

# 同理Girl还有两个属性
print(Girl.name, Girl.nickname)  # 古明地觉 小五萝莉
```


我们之前还说过，一个类在没有指定的metaclass的时候，如果它的父类指定了，那么这个类的metaclass等于父类的metaclass。

```
1  class MyType(type):
2
3      def __new__(mcs, name, bases, attr):
4          name = name * 2
5          bases = (list,)
6
7          return super().__new__(mcs, name, bases, attr)
8
9  class Girl(metaclass=MyType):
10     pass
11
12 class A(Girl):
13     pass
14
```

```
print(A.__class__)  # <class '__main__.MyType'>
print(A.__name__)  # AA
```

并且当时还举了个flask的例子，有一种更加优雅的写法。

```
class MyType(type):

    def __new__(mcs, name, bases, attr):
        return super().__new__(mcs, name, bases, attr)


def with_metaclass(meta, bases):
    return meta("tmp", bases, {"gender": "female"})


#with_metaclass(MyType, (list,)) 会返回一个类
#这个类由MyType创建，并且继承自list
#那么Girl再继承这个类
#等价于Girl也是由MyType创建，并且也会继承自list
class Girl(with_metaclass(MyType, (list,))):
    pass

print(Girl.__class__)  # <class '__main__.MyType'>

# 所以with_metaclass(meta, bases) 本身没有太大意义
# 只是为了帮助我们找到元类和继承的类
# 但我们毕竟继承它了，就意味着我们也可以找到它的属性
print(Girl.gender)  # female
```

注意：我们说负责创建类对象的是元类，而元类要么是type、要么是继承自type的子类。

```
class MyType(type):

    def __new__(mcs, name, bases, attr):
        return super().__new__(mcs, name, bases, attr)


# type直接加括号表示由type创建，所以需要通过__new__手动指定
# 并且将 __new__ 的第一个参数换成 MyType
Girl = type.__new__(MyType,
                    "GirlGirlGirl",
                    (list,),
                    {"add": lambda self, value: value + 123})
print(Girl.__name__)  # GirlGirlGirl

g = Girl()
print(g.add(123))  # 246

try:
    type.__new__(int, "A", (object,), {})
except TypeError as e:
    # 指定为int则报错，告诉我们int不是type的子类
    # 因为只有两种情况：要么是type、要么是type的子类
    print(e)
    # type.__new__(int): int is not a subtype of type
```

怎么样，是不是觉得元类很简单呢？其实元类没有什么复杂的，只需要把元类和类对象之间的关系，想象成类对象和实例对象即可。类对象的 __new__ 里面返回了啥，实例就是啥。那么同理，元类的 __new__ 里面返回了啥，类对象就是啥。

为了更好地理解这一点，我们再举个栗子：

```
class MyType(type):

```

```
    def __new__(mcs, name, bases, attr):
        if "f" in attr:
            attr.pop("f")
        return super().__new__(mcs, name, bases, attr)

class Girl(metaclass=MyType):

    def f(self):
        return "f"

    def g(self):
        return "g"

print(Girl().g())  # g
try:
    print(Girl().f())
except AttributeError as e:
    print(e)  # 'Girl' object has no attribute 'f'
```

惊了，我们看到居然没有 f 这个属性，我们明显定义了啊，显然原因就是我们在创建类的时候将其 pop 掉了。

首先创建一个类需要三个元素: 类名、继承的基类、类的一些属性(以字典的形式)，然后会将这三个元素交给元类进行创建。但是我们在创建的时候偷偷地将 f 从 attr 里面给 pop 掉了，因此创建出来的类是没有 f 这个函数的。

元类确实蛮有趣的，而且也没有想象中的那么难，可以多了解一下。基于元类，我们可以实现很多高级操作，可以让代码逻辑变得更加优雅。

此外我们再来看两个和元类有关的魔法函数，分别是 __prepared__ 和 __init_sublcass__。

__prepared__

```
class MyType(type):

    @classmethod
    def __prepare__(mcs, name, bases):
        print("__prepared__")
        # 必须返回一个mapping
        # 至于它是干什么的我们后面说
        return {}

    def __new__(mcs, name, bases, attr):
        print("__new__")
        return super().__new__(mcs, name, bases, attr)

class Girl(metaclass=MyType):
    pass
"""
__prepared__
__new__
"""
```

我们看到__prepare__会在__new__之前被调用，那么它是做什么的呢？答案是添加属性，我们解释一下。

```
class MyType(type):

    @classmethod
    def __prepare__(mcs, name, bases):
```

```
        return {"name": "古明地觉"}

    def __new__(mcs, name, bases, attr):
        return super().__new__(mcs, name, bases, attr)


class Girl(metaclass=MyType):
    pass

print(Girl.name)  # 古明地觉
```

现在应该知道__prepare__是干什么的了，它接收一个name、一个bases，返回一个mapping。我们知道 name、bases、attr 会传递给 __new__，但是在 __new__ 之前会先经过__prepared__。

而 __prepared__返回一个映射(mapping)，假设叫 m 吧，那么会将attr和m合并，相当于执行了attr.update(m)，然后再将 name、bases、attr 交给 __new__。

此外__prepared__这个方法是被classmethod装饰的，并且里面一定要返回一个mapping，否则报错：**TypeError: MyType.__prepare__() must return a mapping, not xxx**

__init_subclass__

它类似于一个钩子函数，在一些简单地场景下可以代替元类。

```
class Base:

    def __init_subclass__(cls, **kwargs):
        print(cls)
        print(kwargs)
# 当类被创建的时候
# 会触发其父类的__init_subclass__
class A(Base):
    pass
"""
<class '__main__.A'>
{}
"""

class B(Base, name="古明地觉", age=16):
    pass
"""
<class '__main__.B'>
{'name': '古明地觉', 'age': 16}
"""
```

所以父类的__init_subclass__里面的 cls 并不是父类本身，而是继承它的类。kwargs，就是额外设置的一些属性，因此我们可以实现一个属性添加器。

```
class Base:

    def __init_subclass__(cls, **kwargs):
        for k, v in kwargs.items():
            setattr(cls, k, v)


class A(Base, name="古明地觉", age=16,
        __str__=lambda self: "hello world" ):
    pass


print(A.name, A.age)  # 古明地觉 16
print(A())  # hello world
```

除了属性添加器，我们还可以实现一个属性拦截器。

```
class Base:

    def __init_subclass__(cls, **kwargs):
        if hasattr(cls, "shit") and hasattr(cls.shit, "__code__"):
            raise Exception(f"{cls.__name__}不允许定义 'shit' 函数")

class A(Base):
    def shit(self):
        pass
"""
Traceback (most recent call last):
  File "......", line 7, in <module>
    class A(Base):
  File "......", line 5, in __init_subclass__
    raise Exception(f"{cls.__name__}不允许定义 'shit' 函数")
Exception: A不允许定义 'shit' 函数
"""
```

以上就是元类相关的知识，记得在前面的文章中已经说过了，这里再啰嗦一遍，这样我们后续分析源码的时候就会轻松一些。

那么下一篇文章，我们就来从源代码的角度分析自定义类对象和 metaclass 的底层实现。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》73. 自定义类对象的底层实现与 metaclass（下）

下一篇 ›
《源码探秘 CPython》71. 类对象 MRO 的设置，与基类的继承