9.18 以编程方式定义类¶

问题¶

你在写一段代码，最终需要创建一个新的类对象。你考虑将类的定义源代码以字符串的形式发布出去。 并且使用函数比如 `exec()` 来执行它，但是你想寻找一个更加优雅的解决方案。

解决方案¶

你可以使用函数 `types.new_class()` 来初始化新的类对象。 你需要做的只是提供类的名字、父类元组、关键字参数，以及一个用成员变量填充类字典的回调函数。例如：

```
# stock.py
# Example of making a class manually from parts

# Methods
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price
def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}

# Make a class
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__
```

这种方式会构建一个普通的类对象，并且按照你的期望工作：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
4555.0
>>>
```

这种方法中，一个比较难理解的地方是在调用完 `types.new_class()` 对 `Stock.__module__` 的赋值。 每次当一个类被定义后，它的 `__module__` 属性包含定义它的模块名。 这个名字用于生成 `__repr__()` 方法的输出。它同样也被用于很多库，比如 `pickle` 。 因此，为了让你创建的类是“正确”的，你需要确保这个属性也设置正确了。

如果你想创建的类需要一个不同的元类，可以通过 `types.new_class()` 第三个参数传递给它。例如：

```
>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                         lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>
```

第三个参数还可以包含其他的关键字参数。比如，一个类的定义如下：

```
class Spam(Base, debug=True, typecheck=False):
    pass
```

那么可以将其翻译成如下的 `new_class()` 调用形式：

```
Spam = types.new_class('Spam', (Base,),
                       {'debug': True, 'typecheck': False},
                       lambda ns: ns.update(cls_dict))
```

`new_class()` 第四个参数最神秘，它是一个用来接受类命名空间的映射对象的函数。 通常这是一个普通的字典，但是它实际上是 `__prepare__()` 方法返回的任意对象，这个在9.14小节已经介绍过了。 这个函数需要使用上面演示的 `update()` 方法给命名空间增加内容。

讨论¶

很多时候如果能构造新的类对象是很有用的。 有个很熟悉的例子是调用 `collections.namedtuple()` 函数，例如：

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>
```

`namedtuple()` 使用 `exec()` 而不是上面介绍的技术。但是，下面通过一个简单的变化， 我们直接创建一个类：

```
import operator
import types
import sys

def named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))

    # Set the module to that of the caller
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

这段代码的最后部分使用了一个所谓的”框架魔法”，通过调用 `sys._getframe()` 来获取调用者的模块名。 另外一个框架魔法例子在2.15小节中有介绍过。

下面的例子演示了前面的代码是如何工作的：

```
>>> Point = named_tuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
```

```
>>>
```

这项技术一个很重要的方面是它对于元类的正确使用。 你可能像通过直接实例化一个元类来直接创建一个类：

```
Stock = type('Stock', (), cls_dict)
```

这种方法的问题在于它忽略了一些关键步骤，比如对于元类中 `__prepare__()` 方法的调用。 通过使用 `types.new_class()` ，你可以保证所有的必要初始化步骤都能得到执行。 比如，`types.new_class()` 第四个参数的回调函数接受 `__prepare__()` 方法返回的映射对象。

如果你仅仅只是想执行准备步骤，可以使用 `types.prepare_class()` 。例如：

```
import types
metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': type})
```

它会查找合适的元类并调用它的 `__prepare__()` 方法。 然后这个元类保存它的关键字参数，准备命名空间后被返回。

更多信息, 请参考 PEP 3115 , 以及 Python documentation .