属性访问引发的无限递归

weapon

闲来笑浮生悬笔一卷入毫端，朱绂临身可与言者不过二三。

起步

上一篇 中介绍了几种属性访问的区别，其中在 `__getattribute__` 中提到 `使用基类的方法来获取属性能避免在方法中出现无限递归的情况` 。 那么哪些情况会引发无限递归呢?

getattribute 引发的无限递归

```
class D(object):
    def __init__(self):
        self.test=20
        self.test2=21
    def __getattribute__(self,name):
        if name=='test':
            return 0
        else:
            return self.__dict__[name]

d = D()
print(d.test)
print(d.test2)
```

由于 `__getattribute__()` 方法是无条件触发，因此在访问 `self.__dict__` 时，会再次调用 `__getattribute__` ，从而引发无限递归。

所以常常使用 `object` 基类的 `__getattribute__` 来避免这种情况:

```
def __getattribute__(self,name):
    if name=='test':
        return 0.
    else:
        return object.__getattribute__(self, name)
```

getattr 引发的无限递归

```
import copy

class Tricky(object):
    def __init__(self):
        self.special = ["foo"]

    def __getattr__(self, name):
        if name in self.special:
            return "yes"
        raise AttributeError()

t1 = Tricky()
assert t1.foo == "yes"
t2 = copy.copy(t1)
assert t2.foo == "yes"
print("This runs, but isn't covered.")
```

这里存在了无限递归，你会发现最后两行代码没有执行。`__getattr__` 函数是当属性不存在时被调用的，而函数里使用的唯一属性是 `self.special` 。但它是在 `__init__` 中创建的，所以它应该始终存在，是吧?

答案在于 `copy.copy` 的工作方式上。复制对象时，它不会调用其 `__init__` 方法。它创建一个新的空对象，然后将属性从旧复制到新的。为了实现自定义复制，对象可以提供执行复制的功能，因此复制模块在对象上查找这些属性。这自然会调用 `__getattr__` 。

如果在 `__getattr__` 添加一些输出追踪:

```
def __getattr__(self, name):
    print(name)
    if name in self.special:
        return "yes"
    raise AttributeError()
```

得到的输出是:

```
foo
__getstate__
__setstate__
special
special
special
special
```

```
special
...
```

这里发生的过程是：复制模块查找 `__setstate__` 属性，该属性不存在，因此调用了 `__getattr__` 。新对象是空白的，它没有调用 `__init__` 来创建 `self.special` 。由于该属性不存在，因此调用 `__getattr__` ，并开始无限递归。

```
Traceback (most recent call last):
  File "E:/workspace/test3.py", line 15, in <module>
    t2 = copy.copy(t1)
  File "E:\workspace\.env\lib\copy.py", line 106, in copy
    return _reconstruct(x, None, *rv)
  File "E:\workspace\.env\lib\copy.py", line 281, in _reconstruct
    if hasattr(y, '__setstate__'):
```

从异常信息可以看出，引发这次雪崩的导火线是 `hasattr(y, '__setstate__')` 。 在这个案例中，函数可以这样修改：

```
def __getattr__(self, name):
    if name == "special":
        raise AttributeError()
    if name in self.special:
        return "yes"
    raise AttributeError()
```

总结

总之，在获取对象内部数据的钩子中用到了对象的属性一定要格外小心，不然容易引发递归问题。

发布于 2018-11-01

递归 Python Python 入门