8.15 属性的代理访问¶

问题¶

你想将某个实例的属性访问代理到内部另一个实例中去，目的可能是作为继承的一个替代方法或者实现代理模式。

解决方案¶

简单来说，代理是一种编程模式，它将某个操作转移给另外一个对象来实现。 最简单的形式可能是像下面这样：

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass


class B1:
    """简单的代理"""

    def __init__(self):
        self._a = A()

    def spam(self, x):
        # Delegate to the internal self._a instance
        return self._a.spam(x)

    def foo(self):
        # Delegate to the internal self._a instance
        return self._a.foo()

    def bar(self):
        pass
```

如果仅仅就两个方法需要代理，那么像这样写就足够了。但是，如果有大量的方法需要代理， 那么使用 `__getattr__()` 方法或许更好些：

```
class B2:
    """使用__getattr__的代理，代理方法比较多时候"""

    def __init__(self):
        self._a = A()

    def bar(self):
        pass

    # Expose all of the methods defined on class A
    def __getattr__(self, name):
        """这个方法在访问的attribute不存在的时候被调用
        the __getattr__() method is actually a fallback method
        that only gets called when an attribute is not found"""
        return getattr(self._a, name)
```

`__getattr__` 方法是在访问attribute不存在的时候被调用，使用演示：

```
b = B()
b.bar() # Calls B.bar() (exists on B)
b.spam(42) # Calls B.__getattr__('spam') and delegates to A.spam
```

另外一个代理例子是实现代理模式，例如：

```
# A proxy class that wraps around another object, but
# exposes its public attributes
class Proxy:
    def __init__(self, obj):
```

```
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

使用这个代理类时，你只需要用它来包装下其他类即可：

```
class Spam:
    def __init__(self, x):
        self.x = x

    def bar(self, y):
        print('Spam.bar:', self.x, y)

# Create an instance
s = Spam(2)
# Create a proxy around it
p = Proxy(s)
# Access the proxy
print(p.x)  # Outputs 2
p.bar(3)  # Outputs "Spam.bar: 2 3"
p.x = 37  # Changes s.x to 37
```

通过自定义属性访问方法，你可以用不同方式自定义代理类行为(比如加入日志功能、只读访问等)。

讨论¶

代理类有时候可以作为继承的替代方案。例如，一个简单的继承如下：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B(A):
    def spam(self, x):
        print('B.spam')
        super().spam(x)
    def bar(self):
        print('B.bar')
```

使用代理的话，就是下面这样：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B:
```

```
    def __init__(self):
        self._a = A()
    def spam(self, x):
        print('B.spam', x)
        self._a.spam(x)
    def bar(self):
        print('B.bar')
    def __getattr__(self, name):
        return getattr(self._a, name)
```

当实现代理模式时，还有些细节需要注意。 首先，`__getattr__()` 实际是一个后备方法，只有在属性不存在时才会调用。 因此，如果代理类实例本身有这个属性的话，那么不会触发这个方法的。 另外，`__setattr__()` 和 `__delattr__()` 需要额外的魔法来区分代理实例和被代理实例 `_obj` 的属性。 一个通常的约定是只代理那些不以下划线 _ 开头的属性(代理类只暴露被代理类的公共属性)。

还有一点需要注意的是，`__getattr__()` 对于大部分以双下划线(__)开始和结尾的属性并不适用。 比如，考虑如下的类：

```
class ListLike:
    """__getattr__对于双下划线开始和结尾的方法是不能用的，需要一个个去重定义"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)
```

如果是创建一个ListLike对象，会发现它支持普通的列表方法，如append()和insert()， 但是却不支持len()、元素查找等。例如：

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

为了让它支持这些方法，你必须手动的实现这些方法代理：

```
class ListLike:
    """__getattr__对于双下划线开始和结尾的方法是不能用的，需要一个个去重定义"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)

    # Added special methods to support certain list operations
    def __len__(self):
        return len(self._items)

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value

    def __delitem__(self, index):
        del self._items[index]
```

11.8小节还有一个在远程方法调用环境中使用代理的例子。