实现描述器

发布 *3.7.8rc1*

Guido van Rossum
and the Python development team

六月 23, 2020

Python Software Foundation
Email: docs@python.org

Contents

作者 Raymond Hettinger

联系方式 <python at rcn dot com>

目录

1 摘要

定义描述器，总结描述器协议，展示描述器被如何使用。测试一个自定义的描述器和若干 Python 内置的描述器，包括函数、属性、静态方法和类方法。通过给出一个纯 Python 的等价实现和例程，展示每个描述器如何工作。

学习描述器不仅能提供接触到更多工具集的途径，还能更深地理解 Python 工作的原理并更加体会到其设计的优雅性。

2 定义和简介

一般地，一个描述器是一个包含“绑定行为”的对象，对其属性的访问被描述器协议中定义的方法覆盖。这些方法有：`__get__()`，`__set__()` 和 `__delete__()`。如果某个对象中定义了这些方法中的任意一个，那么这个对象就可以被称为一个描述器。

属性访问的默认行为是从一个对象的字典中获取、设置或删除属性。例如，`a.x` 的查找顺序会从 `a.__dict__['x']` 开始，然后是 `type(a).__dict__['x']`，接下来依次查找 `type(a)` 的基类，不包括元类。如果找到的值是定义了某个描述器方法的对象，则 Python 可能会重载默认行为并转而发起调用描述器方法。这具体发生在优先级链的哪个环节则要根据所定义的描述器方法及其被调用的方式来决定。

描述器是一个强大而通用的协议。它们是特征属性、方法静态方法、类方法和 `super()` 背后的实现机制。它们在 Python 内部被广泛使用来实现自 2.2 版中引入的新式类。描述器简化了底层的 C 代码并为 Python 的日常程序提供了一组灵活的新工具。

3 描述器协议

```
descr.__get__(self, obj, type=None) -> value

descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None
```

以上就是全部。定义这些方法中的任何一个的对象被视为描述器，并覆盖其默认行为。

如果一个对象同时定义了 `__get__()` 和 `__set__()`，则它会被视为数据描述器。仅定义了 `__get__()` 的称为非数据描述器（它们通常被用于方法，但也可以有其他用途）。

数据和非数据描述器的不同之处在于，如何计算实例字典中条目的替代值。如果实例的字典具有与数据描述器同名的条目，则数据描述器优先。如果实例的字典具有与非数据描述器同名的条目，则该字典条目优先。

为了使只读数据描述符，同时定义 `__get__()` 和 `__set__()`，并在 `__set__()` 中引发 `AttributeError`。用引发异常的占位符定义 `__set__()` 方法能够使其成为数据描述符。

4 发起调用描述器

描述符可以通过其方法名称直接调用。例如，`d.__get__(obj)` 。

或者，更常见的是在属性访问时自动调用描述符。例如，在中 `obj.d` 会在 `d` 的字典中查找 `obj` 。如果 `d` 定义了方法 `__get__()` ，则 `d.__get__(obj)` 根据下面列出的优先级规则进行调用。

调用的细节取决于 `obj` 是对象还是类。

对于对象来说，`object.__getattribute__()` 中的机制是将 `b.x` 转换为 `type(b).__dict__['x'].__get__(b, type(b))` 。这个实现通过一个优先级链完成，该优先级链赋予数据描述器优先于实例变量的优先级，实例变量优先于非数据描述符的优先级，并如果 `__getattr__()` 方法存在，为其分配最低的优先级。完整的 C 实现可在 Objects/object.c 中的 `PyObject_GenericGetAttr()` 找到。

对于类来说，机制是 `type.__getattribute__()` 中将 `B.x` 转换为 `B.__dict__['x'].__get__(None, B)` 。在纯 Python 中，它就像：

```
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

要记住的重要点是：

- 描述器由 `__getattribute__()` 方法调用
- 重写 `__getattribute__()` 会阻止描述器的自动调用
- `object.__getattribute__()` 和 `type.__getattribute__()` 会用不同的方式调用 `__get__()`.
- 数据描述符始终会覆盖实例字典。
- 非数据描述器会被实例字典覆盖。

`super()` 返回的对象还有一个自定义的 `__getattribute__()` 方法用来发起调用描述器。调用 `super(B, obj).m()` 会搜索 `obj.__class__.__mro__` 紧随 `B` 的基类 `A`，然后返回 `A.__dict__['m'].__get__(obj, B)`。如果其不是描述器，则原样返回 `m`。如果不在字典中，`m` 会转而使用 `object.__getattribute__()` 进行搜索。

这个实现的具体细节在 Objects/typeobject.c. 的 `super_getattro()` 中，并且你还可以在 Guido's Tutorial 中找到等价的纯 Python 实现。

以上展示的关于描述器机制的细节嵌入在 `object` ，`type` ，和 `super()` 中的 `__getattribute__()` 。当类派生自类 `object` 或有提供类似功能的元类时，它们将继承此机制。同样，类可以通过重写 `__getattribute__()` 阻止描述器调用。

5 描述符示例

以下代码创建一个类，其对象是数据描述器，该描述器为每个 get 或 set 打印一条消息。覆盖 `__getattribute__()` 是可以对每个属性执行此操作的替代方法。但是，此描述器对于跟踪仅几个选定的属性很有用：

```
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
```

(下页继续)

(续上页)

```
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

这个协议很简单，并提供了令人兴奋的可能性。有几种用例非常普遍，以至于它们被打包到单独的函数调用中。属性、绑定方法、静态方法和类方法均基于描述器协议。

6 属性

调用 `property()` 是构建数据描述器的简洁方式，该数据描述器在访问属性时触发函数调用。它的签名是：

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

该文档显示了定义托管属性 `x` 的典型用法：

```
class C(object):
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

要了解 `property()` 如何根据描述符协议实现，这里是一个纯 Python 的等价实现：

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc
```

(下页继续)

(续上页)

```
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

这个内置的 `property()` 每当用户访问属性时生效，随后的变化需要一个方法的参与。

例如，一个电子表格类可以通过 `Cell('b10').value` 授予对单元格值的访问权限。对程序的后续改进要求每次访问都要重新计算单元格；但是，程序员不希望影响直接访问该属性的现有客户端代码。解决方案是将对 value 属性的访问包装在属性数据描述器中:

```
class Cell(object):
    . . .
    def getvalue(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
    value = property(getvalue)
```

7 函数和方法

Python 的面向对象功能是在基于函数的环境构建的。使用非数据描述符，两者完成了无缝融合。

类字典将方法存储为函数。在类定义中，方法是用 `def` 或 `lambda` 这两个创建函数的常用工具编写的。方法与常规函数的不同之处仅在于第一个参数是为对象实例保留的。按照 Python 约定，实例引用称为 *self* ，但也可以称为 *this* 或任何其他变量名称。

为了支持方法调用，函数包含 `__get__()` 方法用于在访问属性时将其绑定成方法。这意味着所有函数都是非数据描述符，当从对象调用它们时，它们返回绑定方法。在纯 Python 中，它的工作方式如下:

```
class Function(object):
    . . .
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        if obj is None:
            return self
        return types.MethodType(self, obj)
```

运行解释器显示了函数描述器在实践中的工作方式:

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()

# Access through the class dictionary does not invoke __get__.
# It just returns the underlying function object.
>>> D.__dict__['f']
<function D.f at 0x00C45070>

# Dotted access from a class calls __get__() which just returns
# the underlying function unchanged.
>>> D.f
<function D.f at 0x00C45070>

# The function has a __qualname__ attribute to support introspection
>>> D.f.__qualname__
'D.f'

# Dotted access from an instance calls __get__() which returns the
# function wrapped in a bound method object
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>

# Internally, the bound method stores the underlying function,
# the bound instance, and the class of the bound instance.
>>> d.f.__func__
<function D.f at 0x1012e5ae8>
>>> d.f.__self__
<__main__.D object at 0x1012e1f98>
>>> d.f.__class__
<class 'method'>
```

8 静态方法和类方法

非数据描述器为把函数绑定到方法的通常模式提供了一种简单的机制。

概括地说，函数具有 `__get__()` 方法，以便在作为属性访问时可以将其转换为方法。非数据描述符将 `obj.f(*args)` 的调用转换为 `f(obj, *args)` 。调用 *klass.f(*args)*‘因而变成 `f(*args)` 。

下表总结了绑定及其两个最有用的变体：

转换形式	通过对象调用	通过类调用
function -- 函数	f(obj, *args)	f(*args)
静态方法	f(*args)	f(*args)
类方法	f(type(obj), *args)	f(klass, *args)

静态方法返回底层函数，不做任何更改。调用 `c.f` 或 `C.f` 等效于通过 `object.__getattribute__(c, "f")` 或 `object.__getattribute__(C, "f")` 查找。结果，该函数变得可以从对象或类中进行相同的访问。

适合于作为静态方法的是那些不引用 `self` 变量的方法。

例如，一个统计用的包可能包含一个实验数据的容器类。该容器类提供了用于计算数据的平均值，均值，中位数和其他描述性统计信息的常规方法。但是，可能有在概念上相关但不依赖于数据的函数。例如，`erf(x)` 是在统计中的便捷转换，但并不直接依赖于特定的数据集。可以从对象或类中调用它：`s.erf(1.5) --> .9332` 或 `Sample.erf(1.5) --> .9332`。

由于静态方法直接返回了底层的函数，因此示例调用是平淡的：

```
>>> class E(object):
...     def f(x):
...         print(x)
...     f = staticmethod(f)
...
>>> E.f(3)
3
>>> E().f(3)
3
```

使用非数据描述器，纯 Python 的版本 `staticmethod()` 如下所示：

```
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

与静态方法不同，类方法在调用函数之前将类引用放在参数列表的最前。无论调用方是对象还是类，此格式相同：

```
>>> class E(object):
...     def f(klass, x):
...         return klass.__name__, x
...     f = classmethod(f)
...
>>> print(E.f(3))
('E', 3)
>>> print(E().f(3))
('E', 3)
```

此行为适用于当函数仅需要使用类引用并且不关心任何底层数据时的情况。类方法的一种用途是创建替代的类构造器。在 Python 2.3 中，类方法 `dict.fromkeys()` 会从键列表中创建一个新的字典。纯 Python 的等价形式是:

```
class Dict(object):
    . . .
    def fromkeys(klass, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys = classmethod(fromkeys)
```

现在可以这样构造一个新的唯一键字典：

```
>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}
```

使用非数据描述符协议，纯 Python 版本的 `classmethod()` 如下：

```
class ClassMethod(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f
```

(下页继续)

(续上页)

```
    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```