
实现描述器

发布 2.7.18

Guido van Rossum
and the Python development team

五月 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 摘要	2
2 定义和简介	2
3 描述器协议	2
4 发起调用描述器	3
5 描述符示例	4
6 属性	4
7 函数和方法	6
8 静态方法和类方法	6

作者 Raymond Hettinger

联系方式 <python at rcn dot com>

目录

- 实现描述器
 - 摘要
 - 定义和简介
 - 描述器协议
 - 发起调用描述器
 - 描述符示例

- 属性
- 函数和方法
- 静态方法和类方法

1 摘要

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

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。

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined. Note that descriptors are only invoked for new style objects or classes (a class is new style if it inherits from `object` or `type`).

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。

The details of invocation depend on whether `obj` is an object or a class. Either way, descriptors only work for new style objects and classes. A class is new style if it is a subclass of `object`.

对于对象来说，`object.__getattr__()` 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.__getattr__()` 中将 `B.x` 转换为 `B.__dict__['x'].__get__(None, B)`。在纯 Python 中，它就像：

```
def __getattr__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattr__(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 `__getattr__()` 方法调用
- 重写 `__getattr__()` 会阻止描述器的自动调用
- `__getattr__()` is only available with new style classes and objects
- `object.__getattr__()` 和 `type.__getattr__()` 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 `__getattr__()` 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.__getattr__()` 进行搜索。

Note, in Python 2.2, `super(B, obj).m()` would only invoke `__get__()` if `m` was a data descriptor. In Python 2.3, non-data descriptors also get invoked unless an old-style class is involved. The implementation details are in `super_getattro()` in `Objects/typeobject.c`.

以上展示的关于描述器机制的细节嵌入在 `object`，`type`，和 `super()` 中的 `__getattr__()`。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 `__getattr__()` 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'):
        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
```

The protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound and unbound methods, static methods, and class methods are all based on the descriptor protocol.

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
```

(下页继续)

```
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

    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。

Class dictionaries store methods as functions. In a class definition, methods are written using `def` and `lambda`, the usual tools for creating functions. The only difference from regular functions is that the first argument is reserved for the object instance. By Python convention, the instance reference is called *self* but may be called *this* or any other variable name.

To support method calls, functions include the `__get__()` method for binding methods during attribute access. This means that all functions are non-data descriptors which return bound or unbound methods depending whether they are invoked from an object or a class. In pure python, it works like this:

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

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()
>>> D.__dict__['f'] # Stored internally as a function
<function f at 0x00C45070>
>>> D.f           # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f           # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

The output suggests that bound and unbound methods are two different types. While they could have been implemented that way, the actual C implementation of `PyMethod_Type` in `Objects/classobject.c` is a single object with two different representations depending on whether the `im_self` field is set or is `NULL` (the C equivalent of `None`).

Likewise, the effects of calling a method object depend on the `im_self` field. If set (meaning bound), the original function (stored in the `im_func` field) is called as expected with the first argument set to the instance. If unbound, all of the arguments are passed unchanged to the original function. The actual C implementation of `instancemethod_call()` is only slightly more complex in that it includes some type checking.

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	<code>f(obj, *args)</code>	<code>f(*args)</code>
静态方法	<code>f(*args)</code>	<code>f(*args)</code>
类方法	<code>f(type(obj), *args)</code>	<code>f(klass, *args)</code>

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.__getattr__(c, "f")` 或 `object.__getattr__(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)
...
>>> print E.f(3)
3
>>> print 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
```

(下页继续)

```
    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

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