魔术方法 ***

反射

概述

运行时, runtime, 区别于编译时, 指的是程序被加载到内存中执行的时候。

反射, reflection, 指的是运行时获取类型定义信息。

一个对象能够在运行时,像照镜子一样,反射出其类型信息。

简单说,在Python中,能够通过一个对象,找出其type、class、attribute或method的能力,称为反射或者自省。

具有反射能力的函数有 type()、isinstance()、callable()、dir()、getattr()等

内建函数	意义
getattr(object, name[, default])	通过name返回object的属性值。当属性不存在,将使用default返回,如果没有default,则抛出AttributeError。name必须为 字符串
setattr(object, name, value)	object的属性存在,则覆盖,不存在,新增
hasattr(object, name)	判断对象是否有这个名字的属性,name必须为 字符串

```
1  class Point:
2    def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6    p1 = Point(4, 5)
7    print(p1)
```

为上面Point类增加打印的方法

```
1 class Point:
2
      def __init__(self, x, y):
3
          self.x = x
4
           self.y = y
5
6 p1 = Point(4, 5)
7 print(p1)
8 print(p1.x, p1.y)
   print(getattr(p1, 'x'), getattr(p1, 'y'), getattr(p1, 'z', 100))
10 | setattr(p1, 'x', 10)
setattr(Point, '__str__', lambda self: "<Point {},{}>".format(self.x,
    self.y))
12 | print(p1)
```

反射相关的魔术方法

__getattr__() 、__setattr__() 、__delattr__() 这三个魔术方法,分别测试这三个方法 __getattr__()

```
class Point:
 1
 2
        def __init__(self, x, y):
 3
            self.x = x
            self.y = y
 4
       def __getattr__(self, item):
 6
 7
            print('getattr~~')
 8
            print(item)
 9
            return 100
10
    p1 = Point(4, 5)
11
12
   print(p1.x)
13 | print(p1.y)
14 print(p1.z)
```

实例属性查找顺序为:

```
instance.__dict__ --> instance.__class__.__dict__ --> 继承的祖先类(直到object)的 __dict__ ---找不到--> 调用__getattr__()
```

__setattr__()

```
class Point:
 2
        def __init__(self, x, y):
 3
            self.x = x
            self.y = y
 4
 5
 6
        def __getattr__(self, item):
 7
            print('getattr~~')
 8
            print(item)
 9
            return 100
10
        def __setattr__(self, key, value):
11
12
            print('setattr~~~, {}={}'.format(key, value))
13
14 p1 = Point(4, 5)
15 print(p1.x)
16 print(p1.y)
    print(p1.__dict__)
17
```

p1的实例字典里面什么都没有,而且访问x和y属性的时候竟然访问到了__getattr__(),为什么?

```
1 class Point:
2    def __init__(self, x, y):
3         self.x = x
4         self.y = y
```

```
def __getattr__(self, item):
 7
            print('getattr~~')
 8
            print(item)
9
            return 100
10
11
        def __setattr__(self, key, value):
12
            print('setattr~~~, {}={}'.format(key, value))
13
            self.__dict__[key] = value
14
            #setattr(self, key, value) # 对吗
15
16 p1 = Point(4, 5)
17 | print(p1.x)
18 print(p1.y)
19 print(p1.__dict__)
```

__setattr__()方法,可以拦截对实例属性的增加、修改操作,如果要设置生效,需要自己操作实例的__dict__。

__delattr__()

```
1 class Point:
2
       Z = 100
 3
       def __init__(self, x, y):
           self.x = x
4
5
            self.y = y
6
7
       def __delattr__(self, item):
            print('delattr, {}'.format(item))
8
9
10 p1 = Point(4, 5)
11 del p1.x
12 del p1.y
13
   del p1.z
14 | print(p1.__dict__)
print(Point.__dict__)
16 del Point.Z
17 print(Point.__dict__)
```

通过实例删除属性,就会尝试调用该魔术方法。

___getattribute___

```
1 class Point:
2
        z = 100
3
       def __init__(self, x, y):
4
           self.x = x
 5
            self.y = y
6
7
    p1 = Point(4, 5)
8
   print(p1.x, p1.y)
    print(Point.Z, p1.Z)
9
10 | print('-' * 30)
11
12  # 为Point类增加__getattribute__,观察变化
13 | class Point:
14
        z = 100
```

```
def __init__(self, x, y):
15
16
            self.x = x
17
            self.y = y
18
19
        def __getattribute__(self, item):
            print(item)
20
21
    p1 = Point(4, 5)
22
23 | print(p1.x, p1.y)
24 | print(Point.Z, p1.Z)
25 print(p1.__dict__)
```

实例的所有的属性访问,第一个都会调用__getattribute__方法,它阻止了属性的查找,该方法应该返回(计算后的)值或者抛出一个AttributeError异常。

- 它的return值将作为属性查找的结果。
- 如果抛出AttributeError异常,则会直接调用 __getattr__ 方法,因为表示属性没有找到。

```
1 class Point:
2
        z = 100
        def __init__(self, x, y):
 3
            self.x = x
 4
 5
            self.y = y
 6
 7
        def __getattr__(self, item):
 8
            return 'missing {}'.format(item)
 9
10
        def __getattribute__(self, item):
11
            print(item)
12
            #raise AttributeError('Not Found')
13
            #return self.__dict__[item]
14
15
            #return object.__getattribute__(self, item)
            return super().__getattribute__(item)
16
17
18 p1 = Point(4, 5)
19 | print(p1.x, p1.y)
20 print(Point.Z, p1.Z)
21 | print(p1.__dict__)
```

__getattribute__方法中为了避免在该方法中无限的递归,它的实现应该永远调用基类的同名方法以访问需要的任何属性,例如 object.__getattribute__(self, name)。

注意,除非你明确地知道 __getattribute__ 方法用来做什么,否则不要使用它。

总结

魔术方法	意义
getattr()	当通过搜索实例、实例的类及祖先类 查不到 属性,就会调用此方法
setattr()	通过。访问实例属性,进行增加、修改都要调用它
delattr()	当通过实例来删除属性时调用此方法
getattribute	实例所有的属性调用都从这个方法开始

实例属性查找顺序:

|实例调用__getattribute__() --> instance.__dict__ --> instance.__class__.__dict__ --> 继承的祖先类(直到object)的__dict__ --> 调用__getattr__()