描述器 Descriptors

描述器的表现

用到3个魔术方法:`__get__()`、`__set__()`、`__delete__()`

方法签名如下
object.__get__(self, instance, owner)
object.__set__(self, instance, value)
object.__delete__(self, instance)

self 指代当前实例，调用者
instance 是owner的实例
owner 是属性的所属的类

请思考下面程序的执行流程是什么?

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        print('A.init')

class B:
    x = A()
    def __init__(self):
        print('B.init')


print('-'*20)
print(B.x.a1)

print('='*20)
b = B()
print(b.x.a1)

# 运行结果
A.init
--------------------
a1
====================
B.init
a1
```


可以看出执行的先后顺序吧?
类加载的时候，类变量需要先生成，而类B的x属性是类A的实例，所以类A先初始化，所以打印A.init。
然后执行到打印B.x.a1。
然后实例化并初始化B的实例b。
打印b.x.a1，会查找类属性b.x，指向A的实例，所以返回A实例的属性a1的值。

看懂执行流程了，再看下面的程序，对类A做一些改造。
如果在类A中实现`__get__`方法，看看变化

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        print('A.init')

    def __get__(self, instance, owner):
        print("A.__get__ {} {} {}".format(self, instance, owner))

class B:
    x = A()
    def __init__(self):
        print('B.init')


print('-'*20)
print(B.x)
#print(B.x.a1) # 抛异常AttributeError: 'NoneType' object has no attribute 'a1'

print('='*20)
b = B()
print(b.x)
#print(b.x.a1) # 抛异常AttributeError: 'NoneType' object has no attribute 'a1'

# 运行结果
A.init
--------------------
A.__get__ <__main__.A object at 0x0000000001084E48> None <class '__main__.B'>
None
====================
B.init
A.__get__ <__main__.A object at 0x0000000001084E48> <__main__.B object at 0x0000000001084F28>
<class '__main__.B'>
None
```

因为定义了`__get__`方法，类A就是一个**描述器**，对类B或者类B的实例的x属性读取，成为对类A的实例的访问，就会调用`__get__`方法

如何解决上例中访问报错的问题，问题应该来自`__get__`方法。

self, instance, owner这三个参数，是什么意思？来看输出结果

```
__get__(self, instance, owner)方法的签名，会传入3个参数
B.x调用返回 <__main__.A object at 0x0000000000B84E48> None <class '__main__.B'>
b.x调用放回 <__main__.A object at 0x0000000000B84E48> <__main__.B object at 0x0000000000B84F28>
<class '__main__.B'>
```

self 对应都是A的实例
owner 对应都是B类
instance 说明
- None表示不是B类的实例，对应调用B.x
- <__main__.B object at 0x0000000000B84F28>表示是B的实例，对应调用B().x

使用返回值解决。返回self，就是A的实例，该实例有a1属性，返回正常。

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        print('A.init')

    def __get__(self, instance, owner):
        print("A.__get__ {} {} {}".format(self, instance, owner))
        return self # 解决返回None的问题

class B:
    x = A()
    def __init__(self):
        print('B.init')


print('-'*20)
print(B.x)
print(B.x.a1)

print('='*20)
b = B()
print(b.x)
print(b.x.a1)
```

那么类B的实例属性也可以这样吗?

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        print('A.init')

    def __get__(self, instance, owner):
        print("A.__get__ {} {} {}".format(self, instance, owner))
        return self # 解决返回None的问题

class B:
    x = A()
    def __init__(self):
        print('B.init')
        self.b = A() # 实例属性也指向一个A的实例


print('-'*20)
print(B.x)
```

```
print(B.x.a1)

print('='*20)
b = B()
print(b.x)
print(b.x.a1)

print(b.b) # 并没有触发__get__
```

从运行结果可以看出，只有类属性是类的实例才行。

描述器定义

Python中，一个类实现了`__get__`、`__set__`、`__delete__`三个方法中的任何一个方法，就是描述器。实现这三个中的某些方法，就支持了描述器协议。

如果仅实现了`__get__`，就是**非数据描述符 non-data descriptor**；
同时实现了`__get__`、`__set__`就是**数据描述符 data descriptor**。

如果一个类的**类属性**设置为描述器实例，那么它被称为owner属主。
当该类的类属性被查找、设置、删除时，就会调用描述器相应的方法。

属性的访问顺序

为上例中的类B增加实例属性x

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        print('A.init')

    def __get__(self, instance, owner):
        print("A.__get__ {} {} {}".format(self, instance, owner))
        return self

class B:
    x = A()
    def __init__(self):
        print('B.init')
        self.x = 'b.x' # 增加实例属性x


print('-'*20)
print(B.x)
print(B.x.a1)

print('='*20)
b = B()
print(b.x)
print(b.x.a1) # AttributeError: 'str' object has no attribute 'a1'
```

类A只实现了__get__()方法，b.x访问到了实例的属性，而不是描述器。

继续修改代码，为类A增加`__set__`方法。

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        print('A.init')

    def __get__(self, instance, owner):
        print("A.__get__ {} {} {}".format(self, instance, owner))
        return self

    def __set__(self, instance, value):
        print('A.__set__ {} {} {}'.format(self, instance, value))
        self.data = value

class B:
    x = A()
    def __init__(self):
        print('B.init')
        self.x = 'b.x' # 增加实例属性x


print('-'*20)
print(B.x)
print(B.x.a1)

print('='*20)
b = B()
print(b.x) # 返回什么
print(b.x.a1) # 返回什么
print(b.x.data) # 返回什么?
```


所有的b.x就会访问描述器的__get__()方法，代码中返回的self就是描述器实例，它的实例字典中就保存着a1和data属性，可以打印b.x.__dict__就可以看到这些属性。

属性查找顺序
实例的`__dict__` 优先于 非数据描述器
数据描述器 优先于 实例的`__dict__`

`__delete__`方法有同样的效果，有了这个方法，也是数据描述器。

尝试着增加下面的2行代码，看看字典的变化
b.x = 500
B.x = 600

b.x = 500，这是调用数据描述器的`__set__`方法，或调用非数据描述器的实例覆盖。
B.x = 600，赋值即定义，这是覆盖类属性。把描述器给替换了。

Python中的描述器

描述器在Python中应用非常广泛。

Python的方法（包括staticmethod()和classmethod()）都实现为非数据描述器。因此，实例可以重新定义和覆盖方法。这允许单个实例获取与同一类的其他实例不同的行为。

property()函数实现为一个数据描述器。因此，实例不能覆盖属性的行为。

```
class A:
    @classmethod
    def foo(cls): # 非数据描述器
        pass

    @staticmethod # 非数据描述器
    def bar():
        pass

    @property # 数据描述器
    def z(self):
        return 5

    def getfoo(self): # 非数据描述器
        return self.foo

    def __init__(self): # 非数据描述器
        self.foo = 100
        self.bar = 200
        #self.z = 300

a = A()
print(a.__dict__)
print(A.__dict__)
```


foo、bar都可以在实例中覆盖，但是z不可以。

练习

1、 实现StaticMethod装饰器

实现StaticMethod装饰器，完成staticmethod装饰器的功能

2、 实现ClassMethod装饰器

实现ClassMethod装饰器，完成classmethod装饰器的功能

```
# 类staticmethod装饰器

class StaticMethod: # 怕冲突改名
    def __init__(self, fn):
        self._fn = fn
```

```
    def __get__(self, instance, owner):
        return self._fn

class A:
    @StaticMethod
    # stmtd = StaticMethod(stmtd)
    def stmtd():
        print('static method')

A.stmtd()
A().stmtd()
```

```
# 类classmethod装饰器
class ClassMethod: # 怕冲突改名
    def __init__(self, fn):
        self._fn = fn

    def __get__(self, instance, owner):
        ret = self._fn(owner)
        return ret

class A:
    @ClassMethod
    # clsmtd = ClassMethod(clsmtd)
    # 调用方式为 A.clsmtd() 或者 A().clsmtd()
    def clsmtd(cls):
        print(cls.__name__)

print(A.__dict__)
A.clsmtd
A.clsmtd()
```

A.clsmtd() 的意思就是None()，一定报错。怎么修改?

应该用partial函数

```
from functools import partial

# 类classmethod装饰器
class ClassMethod: # 怕冲突改名
    def __init__(self, fn):
        self._fn = fn

    def __get__(self, instance, cls):
        ret = partial(self._fn, cls)
        return ret

class A:
    @ClassMethod
    # clsmtd = ClassMethod(clsmtd)
```

```
    # 调用A.clsmtd() 或者 A().clsmtd()
    def clsmtd(cls):
        print(cls.__name__)

print(A.__dict__)
print(A.clsmtd)
A.clsmtd()
A().clsmtd()
```

3、对实例的数据进行校验

```
class Person:
    def __init__(self, name:str, age:int):
        self.name = name
        self.age = age
```

对上面的类的实例的属性name、age进行数据校验

思路

1. 写函数，在`__init__`中先检查，如果不合格，直接抛异常
2. 装饰器，使用inspect模块完成
3. 描述器

```
# 写函数检查
class Person:
    def __init__(self, name:str, age:int):
        params = ((name, str),(age, int))
        if not self.checkdata(params):
            raise TypeError()
        self.name = name
        self.age = age

    def checkdata(self, params):
        for name, tap in params:
            if not isinstance(name, tap):
                return False
        return True

p = Person('tom', '20')
```

这种方法耦合度太高。
装饰器的方式，前面写过类似的，这里不再赘述。

描述器方式
属性写入时，要做类型检查，需要使用数据描述器，写入实例属性的时候做检查。
每一个属性都应该是这个属性描述器。

```
class TypeCheck:
    def __init__(self, name, typ):
```

```
        self.name = name
        self.type = typ

    def __get__(self, instance, owner):
        print('get~~~~')

    def __set__(self, instance, value):
        print('set~~~~')

class Person:
    name = TypeCheck()
    age = TypeCheck()
    def __init__(self, name:str, age:int):
        self.name = name
        self.age = age

p = Person('tom', 20)
print(p.age)
```

继续完成代码

```
class TypeCheck:
    def __init__(self, name, typ):
        self.name = name
        self.type = typ

    def __get__(self, instance, owner):
        print('get~~~~')
        if instance:
            return instance.__dict__[self.name]
        return self

    def __set__(self, instance, value):
        print('set~~~~')
        if not isinstance(value, self.type):
            raise TypeError(value)
        instance.__dict__[self.name] = value

class Person:
    name = TypeCheck('name', str) # 硬编码
    age = TypeCheck('age', int) # 不优雅
    def __init__(self, name:str, age:int):
        self.name = name
        self.age = age

p = Person('tom', 20)
print(p.__dict__)
print(p.age)
```

代码看似不错，但是有硬编码，能否Person中只要写`__init__()`方法就行了？如下

```
class Person:
    def __init__(self, name:str, age:int):
        self.name = name
        self.age = age
```

上面代码，需要

- 注入name、age类属性，且使用描述器
- 提取`__init__()`方法的形参名称和类型注解的类型

需要写段程序完成上述功能

对类使用inspect会有什么效果?
params = inspect.signature(Person).parameters
看看返回什么结果

完整代码如下

```
import inspect

class TypeCheck:
    def __init__(self, name, typ):
        self.name = name
        self.type = typ

    def __get__(self, instance, owner):
        print('get~~~~')
        if instance:
            return instance.__dict__[self.name]
        return self

    def __set__(self, instance, value):
        print('set~~~~')
        if not isinstance(value, self.type):
            raise TypeError(value)
        instance.__dict__[self.name] = value


def typeinject(cls):
    sig = inspect.signature(cls)
    params = sig.parameters
    for name, param in params.items():
        print(name, param)
        if param.annotation != param.empty: # 注入类属性
            setattr(cls, name, TypeCheck(name, param.annotation))
    return cls

@typeinject
class Person:
    # 类属性，由装饰器注入
    # name = TypeCheck('name', str) # 硬编码
    # age = TypeCheck('age', int) # 不优雅
    def __init__(self, name:str, age:int):
```

```
        self.name = name
        self.age = age

print(Person.__dict__)

p1 = Person('tom', 20)
p2 = Person('jerry', '18')
```

可以把上面的函数装饰器改为类装饰器，如何写?

```
import inspect

class TypeCheck:
    def __init__(self, name, typ):
        self.name = name
        self.type = typ

    def __get__(self, instance, owner):
        print('get~~~~')
        if instance:
            return instance.__dict__[self.name]
        return self

    def __set__(self, instance, value):
        print('set~~~~')
        if not isinstance(value, self.type):
            raise TypeError(value)
        instance.__dict__[self.name] = value

class TypeInject:
    def __init__(self, cls):
        self.cls = cls
        sig = inspect.signature(cls)
        params = sig.parameters
        for name, param in params.items():
            print(name, param)
            if param.annotation != param.empty:  # 注入类属性
                setattr(cls, name, TypeCheck(name, param.annotation))

    def __call__(self, *args, **kwargs):
        return self.cls(*args, **kwargs) # 新构建一个新的Person对象

@TypeInject # Person = TypeInject(Person)
class Person:
    # 类属性，由装饰器注入
    # name = TypeCheck('name', str) # 硬编码
    # age = TypeCheck('age', int) # 不优雅
    def __init__(self, name:str, age:int):
        self.name = name
        self.age = age

print(Person.__dict__)
```

```
p1 = Person('tom', 18)
p2 = Person('tom', '20')
```