17. 核能来袭-类与类之间的关系

本节主要内容:

1. 依赖关系
2. 关联关系, 组合关系, 聚合关系
3. 继承关系, self到底是什么鬼?
4. 类中的特殊成员

一. 类与类之间的依赖关系

大千世界, 万物之间皆有规则和规律. 我们的类和对象是对大千世界中的所有事物进行归类. 那事物之间存在着相对应的关系. 类与类之间也同样如此. 在面向对象的世界中. 类与类中存在以下关系:

1. 依赖关系
2. 关联关系
3. 组合关系
4. 聚合关系
5. 继承关系
6. 实现关系

由于python是一门弱类型编程语言. 并且所有的对象之间其实都是多态的关系. 也就是说, 所有的东西都可以当做对象来使用. 所以我们在写代码的时候很容易形成以上关系. 首先. 我们先看第一种, 也是这些关系中紧密程度最低的一个, 依赖关系.

首先, 我们设计一个场景. 还是最初的那个例子. 要把大象装冰箱. 注意. 在这个场景中, 其实是存在了两种事物的. 一个是大象, 大象负责整个事件的掌控者, 还有一个是冰箱, 冰箱负责被大象操纵.

首先, 写出两个类, 一个是大象类, 一个是冰箱类.

```
class Elphant:

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

    def open(self):
        '''
        开门
        :return:
        '''
        pass

    def close(self):
```

```
        '''
        关门
        :return:
        '''
        pass


class Refrigerator:

    def open_door(self):
        print("冰箱门被打开了")

    def close_door(self):
        print("冰箱门被关上了")
```

冰箱的功能非常简单, 只要会开门, 关门就行了. 但是大象就没那么简单了. 想想. 大象开门和关门的时候是不是要先找个冰箱啊. 然后呢? 打开冰箱门. 是不是打开刚才找到的那个冰箱门. 然后装自己. 最后呢? 关冰箱门, 注意, 关的是刚才那个冰箱吧. 也就是说. 开门和关门用的是同一个冰箱. 并且. 大象有更换冰箱的权利, 想进那个冰箱就进那个冰箱. 这时, 大象类和冰箱类的关系并没有那么的紧密. 因为大象可以指定任何一个冰箱. 接下来. 我们把代码完善一下.

```
class Elphant:

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

    def open(self, ref):
        print("大象要开门了. 默念三声. 开!")
        # 由外界传递进来一个冰箱, 让冰箱开门. 这时. 大象不用背着冰箱到处跑.
        # 类与类之间的关系也就不那么的紧密了. 换句话说. 只要是有open_door()方法的对象. 都可以接收运行
        ref.open_door()


    def close(self, ref):
        print("大象要关门了. 默念三声. 关!")
        pass

    def take(self):
        print("钻进去")


class Refrigerator:

    def open_door(self):
        print("冰箱门被打开了")

    def close_door(self):
        print("冰箱门被关上了")
```

```
# 造冰箱
r = Refrigerator()

# 造大象
el = Elphant("神奇的大象")
el.open(r)  # 注意. 此时是把一个冰箱作为参数传递进去了. 也就是说. 大象可以指定任何一个冰箱.
el.take()
el.close(r)
```

此时, 我们说, 大象和冰箱之间就是依赖关系. 我用着你. 但是你不属于我. 这种关系是最弱的. 比如. 公司和雇员之间. 对于正式员工, 肯定要签订劳动合同. 还得小心伺候着. 但是如果是兼职. 那无所谓. 需要了你就来. 不需要你就可以拜拜了. 这里的兼职(临时工) 就属于依赖关系. 我用你. 但是你不属于我.

二. 关联关系.组合关系, 聚合关系

其实这三个在代码上写法是一样的. 但是, 从含义上是不一样的.

1. 关联关系. 两种事物必须是互相关联的. 但是在某些特殊情况下是可以更改和更换的.
2. 聚合关系. 属于关联关系中的一种特例. 侧重点是xxx和xxx聚合成xxx. 各自有各自的声明周期. 比如电脑. 电脑里有CPU, 硬盘, 内存等等. 电脑挂了. CPU还是好的. 还是完整的个体
3. 组合关系. 属于关联关系中的一种特例. 写法上差不多. 组合关系比聚合还要紧密. 比如人的大脑, 心脏, 各个器官. 这些器官组合成一个人. 这时. 人如果挂了. 其他的东西也跟着挂了.

首先我们看关联关系: 这个最简单. 也是最常用的一种关系. 比如. 大家都有男女朋友. 男人关联着女朋友. 女人关联着男朋友. 这种关系可以是互相的, 也可以是单方面的.

```
class Boy:

    def __init__(self, name, girlFriend=None):
        self.name = name
        self.girlFriend = girlFriend

    def have_a_dinner(self):
        if self.girlFriend:
            print("%s 和 %s一起去吃晚餐" % (self.name, self.girlFriend.name))
        else:
            print("单身狗. 吃什么饭")


class Girl:
```

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



b = Boy("alex")
b.have_a_dinner()

# 突然牛B了. 找到女朋友了
g = Girl("如花")
b.girlFriend = g # 有女朋友了. 6666
b.have_a_dinner()


gg = Girl("李小花")
bb = Boy("wusir", gg) # 娃娃亲. 出生就有女朋友. 服不服

bb.have_a_dinner()  # 多么幸福的一家

# 突然.bb失恋了. 娃娃亲不跟他好了
bb.girlFriend = None

bb.have_a_dinner() # 又单身了.
```

注意. 此时Boy和Girl两个类之间就是关联关系. 两个类的对象紧密练习着. 其中一个没有了. 另一个就孤单的不得了. 关联关系, 其实就是 我需要你. 你也属于我. 这就是关联关系. 像这样的关系有很多很多. 比如. 学校和老师之间的关系.

School --- 学校

Teacher--- 老师

老师必然属于一个学校. 换句话说. 每个老师肯定有一个指定的工作机构. 就是学校. 那老师的属性中必然关联着学校.

```
class School:
    def __init__(self, name, address):
        self.name = name
        self.address = address


class Teacher:
    def __init__(self, name, school=None):
        self.name = name
        self.school = school


s1 = School("老男孩北京校区", "美丽的沙河")
s2 = School("老男孩上海校区", "迪士尼旁边")
s3 = School("老男孩深圳校区", "南山区法院欢迎你")
```

```
t1 = Teacher("金王", s1)
t2 = Teacher("银王", s1)
t3 = Teacher("海峰", s2)
t4 = Teacher("高鑫", s3)

# 找到高鑫所在的校区地址
print(t4.school.address)
```

想想, 这样的关系如果反过来. 一个老师可以选一个学校任职, 那反过来. 一个学校有多少老师呢? 一堆吧? 这样的关系如何来描述呢?

```
class School:
    def __init__(self, name, address):
        self.name = name
        self.address = address
        self.t_list = [] # 每个学校都应该有一个装一堆老师的列表

    def add_teacher(self, teacher):
        self.t_list.append(teacher)

class Teacher:
    def __init__(self, name, school=None):
        self.name = name
        self.school = school


s1 = School("老男孩北京校区", "美丽的沙河")
s2 = School("老男孩上海校区", "迪士尼旁边")
s3 = School("老男孩深圳校区", "南山区法院欢迎你")


t1 = Teacher("金王", s1)
t2 = Teacher("银王", s1)
t3 = Teacher("海峰", s2)
t4 = Teacher("高鑫", s3)


s1.add_teacher(t1)
s1.add_teacher(t2)
s1.add_teacher(t3)

# 查看沙河校区有哪些老师
for t in s1.t_list:
    print(t.name)
```

好了. 这就是关联关系. 当我们在逻辑上出现了. 我需要你. 你还得属于我. 这种逻辑 就是关联关系. 那注意. 这种关系的紧密程度比上面的依赖关系要紧密的多. 为什么呢? 想想吧

至于组合关系和聚合关系. 其实代码上的差别不大. 都是把另一个类的对象作为这个类的

属性来传递和保存. 只是在含义上会有些许的不同而已.

三. 继承关系.

在面向对象的世界中存在着继承关系. 我们现实中也存在着这样的关系. 我们说过. x是一种y, 那x就可以继承y. 这时理解层面上的. 如果上升到代码层面. 我们可以这样认为. 子类在不影响父类的程序运行的基础上对父类进行的扩充和扩展. 这里.我们可以把父类被称为超类或者基类. 子类被称为派生类.

首先, 类名和对象默认是可以作为字典的key的

```
class Foo:
    def __init__(self):
        pass

    def method(self):
        pass

    # __hash__ = None

print(hash(Foo))
print(hash(Foo()))
```

既然可以hash. 那就是说字典的key可以是对象或者类

```
dic = {}
dic[Foo] = 123
dic[Foo()] = 456
print(dic) # {<class '__main__.Foo'>: 123, <__main__.Foo object at
0x103491550>: 456}
```

虽然显示的有点儿诡异. 但是是可以用的.

接下来. 我们来继续研究继承上的相关内容. 在本节中主要研究一下self. 记住. 不管方法之间如何进行调用. 类与类之间是何关系. 默认的self都是访问这个方法的对象.

```
class Base:

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

    def func1(self):
        print(self.num)


class Foo(Base):
    pass

obj = Foo(123)
obj.func1() # 123  运行的是Base中的func1
```

继续:

```
class Base:

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

    def func1(self):
        print(self.num)


class Foo(Base):

    def func1(self):
        print("Foo. func1", self.num)

obj = Foo(123)
obj.func1() # Foo. func1 123  运行的是Foo中的func1
```

再来:

```
class Base:

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

    def func1(self):
        print(self.num)
        self.func2()

    def func2(self):
        print("Base.func2")


class Foo(Base):

    def func2(self):
        print("Foo.func2")

obj = Foo(123)
obj.func1() # 123  Foo.func2  func1是Base中的  func2是子类中的
```

总结. self在访问方法的顺序: 永远先找自己的. 自己的找不到再找父类的.

接下来. 来难得:

```
class Base:

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

```
    def func1(self):
        print(self.num)
        self.func2()

    def func2(self):
        print(111, self.num)


class Foo(Base):

    def func2(self):
        print(222, self.num)


lst = [Base(1), Base(2), Foo(3)]
for obj in lst:
    obj.func2() # 111 1  |   111 2   | 222 3
```

再来. 还不够绕.

```
class Base:

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

    def func1(self):
        print(self.num)
        self.func2()

    def func2(self):
        print(111, self.num)


class Foo(Base):

    def func2(self):
        print(222, self.num)


lst = [Base(1), Base(2), Foo(3)]
for obj in lst:
    obj.func1() # 那笔来吧. 好好算.
```

结论: self就是你访问方法的那个对象. 先找自己, 然后在找父类的.

四. 类中的特殊成员

什么是特殊成员呢? __init_()就是一个特殊的成员. 说白了. 带双下划线的那一坨. 这些方法在特殊的场景的时候会被自动的执行. 比如,

1. 类名() 会自动执行__init__()
2. 对象() 会自动执行__call__()
3. 对象[key] 会自动执行__getitem__()
4. 对象[key] = value 会自动执行__setitem__()
5. del 对象[key] 会自动执行 __delitem__()
6. 对象+对象 会自动执行 __add__()
7. with 对象 as 变量 会自动执行__enter__ 和__exit__
8. 打印对象的时候 会自动执行 __str__
9. 干掉可哈希 __hash__ == None 对象就不可哈希了.

创建对象的真正步骤:

首先, 在执行类名()的时候. 系统会自动先执行__new__()来开辟内存. 此时新开辟出来的内存区域是空的. 紧随其后, 系统自动调用__init__()来完成对象的初始化工作. 按照时间轴来算.

1. 加载类
2. 开辟内存(__new__)
3. 初始化(__init__)
4. 使用对象干xxxxxxxxx

类似的操作还有很多很多. 我们不需要完全刻意的去把所有的特殊成员全都记住. 实战中也用不到那么多. 用到了查就是了.