老男孩教育
oldboyedu.com

类之间的关系

讲师：尚玉杰

本章目录

- 依赖(关联)关系
- 组合(聚合)关系
- 继承(实现)关系

依赖(关联)关系

- 执行某个动作的时候，需要其他类的对象来帮助你完成这个操作
  - 将一个类的对象或者类名传到另一个类的方法使用
  - 此时的关系是最轻的，随时可以更换其他对象

```
class Person:
    def play(self,tools):
        tools.run()
        print("我要打游戏了")
class Compture:
    def run(self):
        print("电脑已经打开，DNF已登录")
class Phone:
    def run(self):
        print("王者荣耀已经登陆")
xiaoMing = Person()
xmPhone = Phone()
hwCom = Compture()
xiaoMing.play(xmPhone)
xiaoMing.play(hwCom)
```

练习

- 定义一个英雄类和一个反派类
  - 两个类都包含名字、血量，攻击力（ATK），和一个技能（skill）
  - 两派对象互殴
  - 血量为0死亡，删除对象（del 对象名）

删除对象的属性

- delattr(对象名, "属性名")
- del 对象名.属性名
- del和delattr功能有限，只针对实例对象，类属性删除不了

关联关系（组合）

- 在对象里面包含对象
  - 将一个类的对象封装到另一个类的对象的属性中，就叫组合
    - 一对一关系
    - 一对多关系

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

    def eat(self):
        if self.girlFriend:
            print(f"{self.name}带着他的女朋友{self.girlFriend.name}去吃饭")
        else:
            print("单身狗, 吃狗粮 ")

    def movie(self):
        if self.girlFriend:
            print(f"{self.name}带着他的女朋友{self.girlFriend.name}去看电影")
        else:
            print("单身狗不配看电影")
```

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

bao = Baooy("宝哥")
friend = Girl("唐艺昕")
bao.eat()
bao.movie()
bao.girlFriend = friend
bao.eat()
bao.movie()
```

关联关系

- 一对多关系

```
class Boy:
    def __init__(self):
        self.girl_list = []

    def baMei(self,girl):
        self.girl_list.append(girl)

    def happy(self):
        for i in self.girl_list:
            i.play()

class Girl:
    def __init__(self, name):
        self.name = name
    def play(self):
        print(f"{self.name}和你一起玩")
```

```
bao = Boy()
friend1 = Girl("唐艺昕")
friend2 = Girl("迪丽热巴")
friend3 = Girl("杨颖")
bao.baMei(friend1)
bao.baMei(friend2)
bao.baMei(friend3)
bao.happy()
```

练习

- 老师和学生模型(老师对学生是一对多，学生对老师是一对一)
  - 创建教师类和学生类
  - 教师类有姓名和学生列表两个属性
  - 教师类有添加学生的方法（添加的学生是具体对象）
  - 教师类有显示对应学生姓名和学号的方法
  - 学生类有学号/姓名/教师姓名三个属性
  - 创建多个学生，并添加到某位教师的学生列表中
  - 打印该教师的学生

练习

- 在前面英雄反派互殴的练习中加入一个武器类
  - 武器类有名称和伤害加成两个属性
  - 双方都可以选择拿取武器（添加一个拿武器的方法）
  - 拿取武器后伤害会变高

继承关系

- 面向对象编程（OOP）语言的一个主要功能就是“继承”
  - 它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展
  - 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”
  - 在Python中，同时支持单继承与多继承

继承关系

- 实现继承之后，子类将继承父类的属性和方法
  - 增加了类的耦合性（耦合性不宜多，宜精）、
  - 减少了重复代码
  - 使得代码更加规范化，合理化

组合VS继承

- 组合是指在新类里面创建原有类的对象，重复利用已有类的功能
  “has-a”关系

- 而继承允许设计人员根据其它类的实现来定义一个类的实现
  “is-a”关系

继承关系

- 不要轻易地使用继承，除非两个类之间是“is-a”的关系
  - 不要单纯地为了实现代码的重用而使用继承，因为过多地使用继承会破坏代码的可维护性，当父类被修改的时候，会影响到所有继承自它的子类，从而增加程序的维护难度与成本

- 总结：组装的时候用组合，扩展的是时候用继承

- python3中使用的都是新式类，如果一个类谁都不继承，那这个类会默认继承object类

继承关系

- 单继承：
  - 子类可以继承父类的属性和方法（猫狗都是动物），修改父类，所有子类都会受影响

- isinstance()及issubclass()
  - Python与其他语言不同，当我们定义一个 class 的时候，我们实际上就定义了一种数据类型
  - 我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样
  - Python 有两个判断继承的函数
    - isinstance() 用于检查实例类型:isinstance(对象,类型)
    - issubclass() 用于检查类继承：issubclass(子类,父类)

继承关系

- 方法重写
  - 子类可以重写父类中的方法
  - super()关键字在当前类中调用父类方法

- super()关键字：
  - 子类如果编写了自己的构造方法，但没有显式调用父类的构造方法，而父类构造函数初始化了一些属性，就会出现问题
  - 如果子类和父类都有构造函数，子类其实是重写了父类的构造函数，如果不显式调用父类构造函数，父类的构造函数就不会被执行
  - 解决方式：调用超类构造方法，或者使用super函数 `super(当前类名, self).__init__()`

多重继承和多继承

- 多重继承：包含多个间接父类
- 多继承
  - 有多个直接父类
  - 大部分面向对象的编程语言（除了 C++）都只支持单继承，而不支持多继承
    - 多继承不仅增加了编程的复杂度，而且很容易导致一些莫名的错误
  - Python 虽然在语法上明确支持多继承，但通常推荐如果不是很有必要，则尽量不要使用多继承，而是使用单继承
    - 这样可以保证编程思路更清晰，而且可以避免很多麻烦
  - 如果多个直接父类中包含了同名的方法
    - 此时排在前面的父类中的方法会“遮蔽”排在后面的父类中的同名方法

super()关键字详解

- super(type[, object-or-type])
- Python3可以使用直接使用 super().xxx 代替 super(Class, self).xxx

- 使用多继承，会涉及到查找顺序（MRO）、钻石继承等问题
  - 钻石继承
    - 单继承时 类名.__init__()的方式和super().__init__()方式没啥区别
    - 但是使用 类名.__init__()的方式在钻石继承时会出现问题（参见下页例子）

```
class YeYe:
    def __init__(self):
        print("初始化爷爷")

class Qinba(YeYe):
    def __init__(self):
        print("进入亲爸类")
        YeYe.__init__(self)
        print("初始化亲爸")

class GanDie(YeYe):
    def __init__(self):
        print("进入干爹类")
        YeYe.__init__(self)
        print("初始化干爹")

class ErZi(Qinba,GanDie):
    def __init__(self):
        Qinba.__init__(self)
        GanDie.__init__(self)
        print("初始化儿子")
bigB = ErZi()
```

```
class YeYe:
    def __init__(self):
        print("初始化爷爷")

class Qinba(YeYe):
    def __init__(self):
        super().__init__()
        print("初始化亲爸")

class GanDie(YeYe):
    def __init__(self):
        super().__init__()
        print("初始化干爹")

class ErZi(Qinba,GanDie):
    def __init__(self):
        super().__init__()
        print("初始化儿子")
bigB = ErZi()
```

super()关键字详解

- super的内核 mro方法：返回的是一个类的方法解析顺序表（顺序结构）
  - 我们定义的每一个类，Python 会计算出一个方法解析顺序（Method Resolution Order，MRO）列表，这也是super在父类中查找成员的顺序，它是通过一个C3线性化算法来实现的
  - 每个祖先都在其中出现一次
  - 我们可以使用下面的方式获得某个类的 MRO 列表
  - 类名.mro() 或者 对象名.__class__.mro()
  - 当你使用 super(cls，obj) 时，Python 会在 obj 的 MRO 列表上搜索 cls 的下一个类

迷惑的执行顺序

```
class YeYe:
    def __init__(self):
        print("初始化爷爷")

class Qinba(YeYe):
    def __init__(self):
        print("进入亲爸类")
        super(Qinba, self).__init__()
        print("初始化亲爸")

class GanDie(YeYe):
    def __init__(self):
        print("进入干爹类")
        super(GanDie, self).__init__()
        print("初始化干爹")

class ErZi(Qinba,GanDie):
    def __init__(self):
        super(ErZi, self).__init__()
        print("初始化儿子")
bigB = ErZi()
print(ErZi.mro())
```

- 进入亲爸类
- 进入干爹类
- 初始化爷爷
- 初始化干爹
- 初始化亲爸
- 初始化儿子

首先进入儿子的 __init__ 方法：
这里的 self 是当前儿子的实例，self.__class__.mro() 结果是：
[儿子，亲爸，干爹，爷爷，object]
可以看到，儿子的下一个类是 亲爸，于是跳到了亲爸的 __init__
这时会打印出进入亲爸类，并执行下面一行代码：
`super(Qinba, self).__init__()`
注意，这里的 self 也是当前儿子的实例，MRO 列表跟上面是一样的
搜索儿子在 MRO 中的下一个类，发现是干爹，于是跳到了干爹的 __init__
这时会打印出进入干爹类，而不是进入爷爷类
关键是要理解 super 的工作方式，而不是想当然地认为 super 调用了父类的方法

super()的具体用法总结

- 事实上，super 和父类没有实质性的关联。
  - super(cls, obj) 获得的是 cls 在 obj 的 MRO 列表中的下一个类

```
class ErZi(Qinba,GanDie):
    def __init__(self):
        super(ErZi, self).__init__()
        print("初始化儿子")
```

  - 从这个执行流程可以看到，如果我们不想调用亲爸的__init__，而想要调用干爹的__init__，那么super应该写成：super(亲爸, self)__init__()