第 5 章

面向对象程序设计方法

面向对象程序设计是一种以对象作为基本结构的设计方法，所有编程均以对象为核心，其本质是建立模型体现抽象思维的过程，因为模型是用来反映现实世界中事物特征变化规律的一种抽象，即面向对象程序设计方法使用类、封装、继承、多态性模拟现实世界中不同实体间的相互关系，把复杂的行为分解为简单对象模块再归纳到类中。

5.1 面向对象程序设计技术

面向对象程序设计技术将程序处理的数据和方法封装为一个整体，称为对象。程序中用对象模型模拟现实中的事物，使得空间模型和问题模型的结构相一致。使用面向对象的方法解决问题的思路更加符合人类一贯的思维方法，因为面向对象可从现实存在的事物出发，通过适当的抽象，使工程问题更加模块化，以实现更低的耦合和更高的内聚。在设计模式上，面向对象可以更好地实现开-闭原则，不仅能提高代码阅读性，编程也更加容易。

5.1.1 面向对象描述及应用案例

Python 是完全面向对象的语言，它把一切都视为对象，通过类定义的数据结构实例，对本身已有的静态特征（状态）和动态特征（操作）进行描述。也就是说，它把状态和操作封装于对象体之中，并提供一种访问机制，使对象的“私有数据”仅能由该对象操作执行，用户只能通过允许公开的操作提出要求，查询和修改对象的状态。

Python 变量是对象的一种引用，对象变量包括类变量和实例变量。类变量归类所有，使用时通过类名引用；实例变量归实例所有，引用时加 self. 前缀即可。如果在实例中对类变量赋值，会复制一份为实例变量覆盖类变量，各实例自行修改类变量时，引用到的值都会改变。

【例 5-1】 建立一个 Employee 类，内有 name（姓名）、salary（工资）变量，根据工资计算 tax（纳税额）并输出，在类变量修改后再输出。

```
class Employee:                          #定义类
    name = "张三峰"                      #定义了类变量 name
    salary = 12000                       #定义类变量 salary
    #下面定义了一个 Tax()实例方法
```

```
    def Tax(self):                          #定义类方法实现计算
        if self.salary < 10000:
            tax = self.salary * 0.05
        else:
            tax = self.salary * 0.1
        print(tax)
e1 = Employee()                             #建立类对象
print(Employee.name)                        #调用未修改的类变量
print(Employee.salary)                      #调用未修改的类变量
print("应该纳税:",end = '')                  #输出未修改的纳税额
e1.Tax()                                    #调用类方法实现计算
Employee.name = "李四广"                     #修改类变量的值
Employee.salary = 8000                      #修改类变量的值
print(Employee.name)                        #调用修改后的类变量
print(Employee.salary)                      #调用修改后的类变量
print("应该纳税:",end = '')                  #输出修改后的纳税额
e1.Tax()
```

运行结果为

```
张三峰
12000
应该纳税: 1200.0
李四广
8000
应该纳税: 400.0
```

说明：类变量为所有实例化对象共有，通过类名修改类变量的值，会影响所有实例化对象。

5.1.2 面向对象特征

面向对象具有抽象特征，它把现实世界中的某一类东西提取出来，用程序代码表示，抽象出来的一般叫作类或接口，使用时按照类/接口名加入参数即可，无须关心内部的细节。此外，面向对象编程还具有三大特征：封装、继承和多态。

1. 封装

封装是将属性和方法放到类内部，通过对象访问属性或方法，隐藏功能的实现细节，也可以设置访问权限。封装的意义是：

(1) 将属性和方法放到一起作为一个整体，然后通过实例化对象来处理；

(2) 隐藏内部实现细节，只需要与实现对象及其属性和方法交互；

(3) 对类的属性和方法增加访问权限控制。

2. 继承

子类需要复用父类的属性或方法，且子类也可以提供自己的属性和方法，称为继承。继承的意义是：

(1) 实现代码的重用，相同的代码不需要重复编写；

(2) 子类可以在父类功能上进行重写,扩展类的功能。

当已经创建了一个类,需要创建一个与之相似的类时,通过添加/删除/修改其中的几个方法,可以从原来的类中派生出一个新类,把原来的类称为父类或基类,而派生出的类称为子类或派生类,子类继承了父类的所有数据和方法。

3. 多态

多态指在父类中定义的属性和方法被子类继承之后,可以具有不同的数据类型或表现出不同的行为,这使得同一个属性或方法在父类及其各子类中具有不同的含义。使用时根据参数列表的不同,区分不同的方法调用。

多态的实现方法如下。

(1) 定义一个父类。

(2) 定义多个子类,并重写父类的方法。

(3) 传递子类对象给调用者,不同子类对象能产生不同执行效果。

5.2　类的概念及基本使用

具有相同特征的对象称为类(Class),每个类包含多个对象,它是对现实世界的抽象。

5.2.1　类的描述

类是用来描述具有相同属性和方法对象的集合,它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。类以共同特性和操作定义实体,类也是用于组合各对象所共有操作和属性的一种机制。

例如,人类,从性别可分为男人、女人;从职业可分为工人、农民、军人、医务人员、教师、学生等;从年龄可分为少年、青年、中年和老年等;从外观看,均有一个鼻子和两只眼睛;从行为上描述,都有呼吸,都能摄取食物以补充能量,都能排出废物,都能对外界刺激作出反应,都能产生繁殖后代,等等。

再如,交通类,可分为汽车、火车、飞机、轮船、自行车等,外观共性是都有轮子,行为共性是均可承载重量,都是交通工具。

5.2.2　类和对象的区别

类和对象(Object)可简单理解为:类是对具有共同属性和行为对象的抽象描述,对象则是具体存在的事物。类是对象的模板(Template),对象是类的一个实例(Instance),或对象是具有具体属性和行为的实体,它们的关系如图 5-1 所示。

对动物类的实例化是对每个对象(动物)信息的实例化,包括名称、大小、颜色、几条腿等。对交通类对象(汽车)信息的实例化包括品牌、颜色、型号、重量、体积及油耗等各种参数。

类和对象的描述如图 5-2 所示。

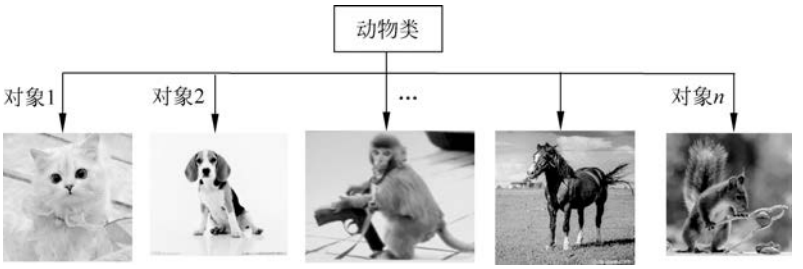

图 5-1　类和对象的关系

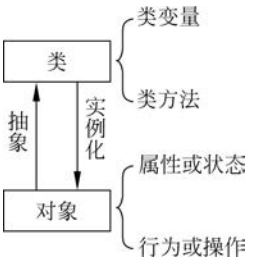

图 5-2　类和对象的描述

类包括类变量和类方法(方法也称为函数),它们用于处理类及实例对象的相关数据,其中,以双下画线"__"开头和结尾的类变量和方法都称为类的特殊成员。例如,类的__init__()方法是对象的构造函数,负责在对象初始化时进行一系列的构建操作,包括对象创建、连接数据库、连接 FTP 服务器、进行 API 验证等操作。

对象具有属性和方法,属性是对象的特征,是包含对象变量的信息(数字、字符串等);方法是完成某个特定任务的代码块,也称为对象函数,包括传递参数和返回值。在类的声明中,属性是用变量表示的,该变量称为实例变量,它在类的内部进行声明。动物类的不同对象,用不同名称(变量)、特征(属性)和行为(方法)进行描述。

5.2.3　对象属性、类方法、类变量及应用案例

Python 的对象属性有一套统一的管理方案,property 是一种特殊的属性,访问它时会执行一段功能(函数)进行一系列的逻辑计算,最终将计算结果返回。类方法和类变量是类中的函数和变量。

1. Python 对象属性

(1) 使用@property 装饰器操作类属性。装饰器可以实现获取、设置、删除隐藏的属性,通过装饰器可以对属性的取值和赋值加以控制,提高代码的稳定性。

(2) 使用类或实例直接操作类属性(如 obj. name、obj. age=18、del obj. age)。

(3) 使用 Python 内置函数操作属性。

2. 类的方法(函数)

在类中定义的方法可以分为公有方法、私有方法两大类。公有方法和私有方法一般指属于对象的实例方法,私有方法的名字以双下画线开始,它不能在类外部被使用或直接访问。

在类的内部,使用 def 关键字定义一个方法,与一般函数定义不同,类方法必须包含参数 self,且为第 1 个参数,self 代表的是类的实例。self 的名字也可以使用 this 替代,但最好还是按照约定用 self 命名,类的私有方法一般以大写字母开头,模块一般用小写加下画线的方式命名。一般在类中定义的变量为全局变量,在方法中定义的变量为局部变量,只作用于

当前实例的类。

3. 类中的私有变量

双下画线私有变量只能在类的内部访问，在类中调用时加上类方法：@classmethod。

【例 5-2】 私有变量的使用。

```
class Object1:                          #定义类
    def __init__(self,name):            #定义类的初始化方法
        self.name = name
    def prin(self):                     #定义类方法
        print(self.name)
    __age = 20                          #私有变量只能在类的内部访问
    @classmethod                        #调用类方法
    def pri(cls):                       #定义类方法实现输出
        print(cls.__age)
        #然后再使用
a = Object1('张三')                     #建立类对象
a.prin()                                #调用类方法
Object1.pri()                           #通过这样直接调用类中的私有变量
```

运行结果为

```
张三
20
```

5.2.4 类的定义及应用案例

在使用类时，需要先定义，然后再创建类的实例对象，常用方法是通过类的实例对象访问类中的属性和方法。

1. 类定义的结构

类定义的结构如下。

```
class 类名 (基类):
    使用构造函数初始化类变量
    def 函数名 1(self, 参数类别):
        初始化
    def 函数名 2(self,参数类别):
        初始化
```

例如：

```
import time
class Person:                           #定义一个类,名为 Person
  def __init__(self,name):              #构造函数初始化类变量 name
      self.name = name                  #初始化变量,方便继承
  def runx(self):                       #定义运行函数,从上面继承变量
      print(self.name)
      #打印 name 的值,或者做一些别的处理
      time.sleep(1)
a = Person('张三')                      #实例化变量
a.runx()                                #调用
```

2. 标准输出流 stdout 的使用

sys 模块下的 sys.stdout 对象称为标准输出流，它可以将一个文本信息输出到屏幕上。使用该方法输出时，需要导入 sys 模块，例如：

```
import sys
a = "Apple"
sys.stdout.write(a)
```

运行结果为

```
Apple
```

说明：

(1) stdout 与 print()函数不同，它不会自动追加换行符，也不会自动插入分隔符。sys.stdout.write('hello'+'\n')等价于 print('hello')。

(2) 只接收字符串类型的参数，如果要输出数字，需要使用 str()函数转换，如 sys.stdout.write(str(123.456)+'\n')。

一个 stdout 的示例如下。

```
import sys
a = "Apple"
sys.stdout.write(a + '\n')
sys.stdout.write('文本居中输出'.center(30, '='))
sys.stdout.write('\n' + str(123.456) + '\n')
sys.stdout.write(str(123.456) + '\n')
```

运行结果为

```
Apple
============ 文本居中输出 ============
123.456
123.456
```

3. __init__()函数的使用

类中的__init__()函数是类的构造函数，为系统进行初始化。

【例 5-3】 __init__()函数的使用。

```
class Person:                                  #定义 Person 类
    def __init__(self,name):                   #定义初始化方法
        self.name = name
        self.sex = '男'
        self.age = 18
obj = Person('张三')                           #通过类对象 obj 自动执行类中的__init__()方法
print('姓名:',obj.name, '性别:',obj.sex,'年龄:',obj.age)        #输出
```

运行结果为

```
姓名: 张三  性别: 男  年龄: 18
```

也可写成

```
class Person:
    def __init__(self, name, sex, age):
        self.name = name
        self.sex = sex
        self.age = age
    def g1(self):                                    #定义类方法
        return "姓名:{}, 性别:{},年龄: {}".format(self.name, self.sex, self.age)
p1 = Person("张三", "男", 18)
print(p1.g1())
```

运行结果为

```
姓名:张三, 性别:男,年龄: 18
```

说明：两种方法均实现了构造函数的初始化。

4. 嵌套类的格式及应用案例

1）嵌套类格式

```
class 样例类名(object):                              #开始
      pass
      class 外部类名(object):
      class 内部类名(object):
            pass
            class 子类名(父类名):
            """已显式从父类继承"""
class 样例类名                                       #结束
      pass
      class 外部类名
      class 内部类名
            pass
```

2）嵌套类应用案例

【例 5-4】 嵌套类的使用。

```
class C:                                             #定义类
    class A:                                         #A 类嵌套在 C 类中
        def __init__(self):
            self.a = '我是 A 类变量'                  #初始化 A 类变量
            print(self.a)

    class B(A):                                      #B 类嵌套在 C 类中且继承 A 类
        def __init__(self):
            self.b = '我是 B 类变量'                  #初始化 B 类变量
            self.a = '在 B 类中给 A 类变量赋值'
            print(self.b,self.a)
    c1 = A()                                         #调用 A 类
    c2 = B()                                         #调用 B 类
    print(c1.a,c2.a,c2.b)                            #输出 A 类和 B 类中的变量
```

```
c3 = C()                                    # 调用 C 类
c4 = c3.A();c5 = c3.B()                     # 使用 C 类调用 A 类和 B 类的输出方法
print(c5.a,c5.b)                            # 输出嵌套类方法
```

运行结果为

```
我是 A 类变量
我是 B 类变量 在 B 类中给 A 类变量赋值
我是 A 类变量 在 B 类中给 A 类变量赋值 我是 B 类变量
我是 A 类变量
我是 B 类变量 在 B 类中给 A 类变量赋值
在 B 类中给 A 类变量赋值 我是 B 类变量
```

5.3 类的方法及调用方式

Python 类中有实例化方法、静态方法和类方法 3 种，其中类方法、静态方法都可以被类和类实例调用，实例化方法仅可以被类实例调用。类方法的隐含调用参数是类，而类实例方法的隐含调用参数是类的实例，静态方法没有隐含调用参数。

5.3.1 实例化方法调用及应用案例

实例化方法是通过实例对象进行调用，方法是先将类赋值给实例对象，通过实例对象调用类中的方法。

【例 5-5】 实例化方法的调用。

```
class dd:
    def __init__(self,name,url):                                    # 建立 dd 类
        self.name = name                                            # 初始化类变量
        self.url = url
    def runx(self):                                                 # 建立类方法实现输出
        print(self.name)
        print(self.url)
a = dd('深圳北理莫斯科大学:','http://www.smbu.edu.cn')               # 实例化对象
a.runx()                                                            # 实例调用
```

运行结果为

```
深圳北理莫斯科大学:
http://www.smbu.edu.cn
```

5.3.2 静态方法调用及应用案例

静态方法是类中的函数，不需要实例，即静态方法是不需要实例化就可以由类执行的方法。静态方法和类本身没有交互，即在静态方法中，不会涉及类中的方法和属性的操作，可以理解为将静态方法保存在此类的名称空间中。因此，若有一个方法实现的功能比较独立，就可以使用静态方法实现。Python 引入静态方法的步骤如下。

（1）通过@staticmethod 装饰器定义静态方法。

（2）@staticmethod 必须写在方法上。

（3）静态方法的调用格式为“类名.静态方法名(参数列表)”,在静态方法中访问实例属性和实例方法会导致错误。

【例 5-6】 使用静态方法计算年龄。

```
class Person:                                    #定义类
  country = "中国"                               #类变量赋初始值
  city = "深圳"
  count = 0
  def __init__(self,name,prof):                  #类初始化方法
    self.name = name
    self.prof = prof
    Person.count = Person.count + 1
  @staticmethod
  def add(a,b):                                  #类方法实现计算
    print("这是第 {} 人".format(Person.count))
    print("{0} + {1} = {2}".format(a,b,a + b))
    return a + b
  def prt(self):                                 #类方法实现格式输出
    print("姓名:{0},职业:{1}".format(self.name,self.prof))
print("我的国籍是{},目前在{}".format(Person.country,Person.city))
p1 = Person("刘威达","教师")
p1.prt()
print("即:30 岁的我,5 年后年龄是:",Person.add(30,5))
p2 = Person("何丽丽","学生")
p2.prt()
print("即:21 岁的我,5 年后年龄是:",Person.add(21,5))
```

运行结果为

```
我的国籍是中国,目前在深圳
姓名:刘威达,职业:教师
这是第 1 人
30 + 5 = 35
即:30 岁的我,5 年后年龄是: 35
姓名:何丽丽,职业:学生
这是第 2 人
21 + 5 = 26
即:21 岁的我,5 年后年龄是: 26
```

5.3.3　类方法调用及应用案例

类方法是将类本身作为对象进行操作的方法,它和静态方法的区别在于：不管是从实例调用还是从类调用,类方法都用第 1 个参数把类传递过来。类方法需要一个关键词@classmethod 来装饰,它可以通过实例对象和类对象进行调用,其调用在实例对象初始化之前进行,若需要在初始化前做一些工作,可以考虑使用类方法。

【例 5-7】 使用类方法计算年龄。

```
from datetime import date
class Student(object):                                    #定义 Student 类
  def __init__(self, name, age):                          #初始化
    self.name = name
    self.age = age
  @classmethod                                            #定义类方法
  def birth(cls, name, birth_year):                       #类方法的形参为姓名和生日年份
    return cls(name, date.today().year - birth_year)      #返回年龄值
  def prt(self):                                          #输出方法
      print("姓名:{0},年龄:{1}".format(self.name, self.age))
student1 = Student("王五常", 20)                          #实例化 1
student2 = Student.birth("李三朵", 2001)                  #实例化 2
student1.prt()
student2.prt()
```

运行结果为

```
姓名: 王五常,年龄: 20
姓名: 李三朵,年龄: 21
```

5.3.4 类变量及应用案例

在类中可以使用类名或类对象名重新赋值,其输出结果及顺序如下。

```
class Test:                                   #创建类
    action = '我在类中'                       #在类中直接赋值
print ("1",Test.action)                       #输出类初始值
t = Test()                                    #建立类对象调用
print("2",t.action)                           #使用类对象输出类对象值
Test.action = '类值变化了'                    #引用类名重新赋值覆盖了原类对象值
print("3",Test.action)                        #使用类名输出类对象值
print("4",t.action)                           #使用类对象输出类对象值
t.action = '又重新赋值啦'                     #引用类对象重新赋值
print("5",Test.action)                        #使用类名输出类对象值
print("6",t.action)                           #使用类对象输出类对象值
```

运行结果为

```
1 我在类中
2 我在类中
3 类值变化了
4 类值变化了
5 类值变化了
6 又重新赋值啦
```

5.3.5 使用 self 参数维护对象状态及应用案例

【例 5-8】 使用类方法编写简单计算器。

```
class calculate(object):                      #定义类
```

```
        def __init__(self,x,y):                    #初始化变量
            self.x = x
            self.y = y
            self.result = 0                        #保存结果的变量
        def add(self,x,y):                         #加法运算方法
            self.result = x + y
        def sub(self,x,y):                         #减法运算方法
            self.result = x - y
        def mul(self,x,y):                         #乘法运算方法
            self.result = x * y
        def div(self,x,y):                         #除法运算方法
            self.result = x/y
    s = calculate()                                #类实例化
    s.add(5,8)                                     #实例调用
    print(s.result)
    s.sub(5,8)
    print(s.result)
    s.mul(5,8)
    print(s.result)
    s.div(5,8)
    print(s.result)
```

运行结果为

```
13
-3
40
0.625
```

5.3.6　__del__(self)与__str__(self)结构应用案例

Python 中__str__用于 class 类中，在主函数中使用 print()函数打印一个实例时会运行该函数；__del__也用于 class 类中，在该实例被删除时运行。

【例 5-9】　__del__(self)与__str__(self)的使用。

```
class Cat:                                         #定义类
    def __init__(self, new_name):
        self.name = new_name
        print('%s 来了' % self.name)
    def __del__(self):
        print('%s 走了' % self.name)
    def __str__(self):
        return '我是 %s' % self.name
if __name__ == '__main__':
    hua = Cat('小花花')
    print(hua)                                     #自动执行__str__()方法
    meng = Cat('小萌萌')
    print(meng)
    del(hua)                                       #自动执行__del__()方法
    print('测试程序完成啦')
```

运行结果为

```
小花花 来了
我是 小花花
小萌萌 来了
我是 小萌萌
小花花 走了
测试程序完成啦
小萌萌 走了
```

5.4 类调用案例

【例 5-10】 类的常规使用。

```
class House:
    purpose = '存储地址:'
    region = '属于中国西部地区'
w1 = House()
print(w1.purpose, w1.region)
w2 = House()
w2.region = '属于中国东部地区'
print(w2.purpose, w2.region)
```

运行结果为

```
存储地址: 属于中国西部地区
存储地址: 属于中国东部地区
```

【例 5-11】 定义一个动物类,添加两个类方法,输出各自的特征。

```
class Animal:                                          #定义类
   def __init__(self,color,weight,legs,mouth):         #初始化类参数
       self.color = color
       self.weight = weight
       self.legs = legs
       self.mouth = mouth
   def cat(self,name):                                 #类方法 1
       self.name = name      #实例变量:定义在方法中的变量,只作用于当前实例的类
       print('我是',name)
       print("我的任务是讨主人喜欢!")
   def hen(self,name):                                 #类方法 2
       self.name = name     #实例变量:定义在方法中的变量,只作用于当前实例的类
       print('我是',name)
       print("我的任务是给主人产蛋!")
   def __str__(self):                                  #print()函数自动调用格式
       return '我的颜色是:%s,重量是:%.2f 千克,有:%d 只脚,嘴巴会 %s' %(self.color,
self.weight,self.legs, self.mouth)
t1 = Animal('黄色',2.2,4,'叫')                         #类的实例化 1
t1.cat("波斯猫")
```

```
print(t1)
t2 = Animal('黑色',1.5,2,'叫')        #类的实例化 2
t2.hen("小黑")
print(t2)
```

运行结果为

```
我是 波斯猫
我的任务是讨主人喜欢!
我的颜色是:黄色,重量是:2.20 千克,有:4 只脚,嘴巴会 叫
我是 小黑
我的任务是给主人产蛋!
我的颜色是:黑色,重量是:1.50 千克,有:2 只脚,嘴巴会 叫
```

【例 5-12】 self 的使用(1)。

```
class People(object):
    def __init__(self,name,age,gender, money):              #初始化方法
        self.name = name
        self.age = age
        self.gender = gender
        self.money = money
    def play(self):                                         #创建类方法
        print("使用 Python 编写代码中")
p1 = People('用户 1', 30, '女', 10000)                      #实例对象
print("用户名:%s,年龄:%d,性别:%s,工资:%d" % (p1.name, p1.age, p1.gender,p1.money))
                                                            #输出
p1.play()
```

运行结果为

```
用户名: 用户 1,年龄: 30,性别: 女,工资: 10000
使用 Python 编写代码中
```

【例 5-13】 已知：张三,女,体重 55 千克,喜欢跑步,每次跑步体重减少 1 千克；李四,男,体重 75 千克,喜欢打羽毛球,每次打球体重减少 0.5 千克。编写类方法实现。

```
class Person:
    def __init__(self, name,sex,weight):
        self.name = name
        self.sex = sex
        self.weight = weight
    def __str__(self):
        return '我的名字叫%s,性别是 %s,体重现在是%.1f'%(self.name, self.sex,self.weight)
    def run(self):
        print('%s 喜欢跑步' % self.name)
        self.weight -= 1
    def ball(self):
        print('%s 喜欢打羽毛球'%self.name)
        self.weight -= 0.5
xx = Person('张三','女',55)
```

```
xm = Person('李四','男',75.5)
xx.run()
xm.ball()
print(xx)
print(xm)
```

运行结果为

```
张三 喜欢跑步
李四 喜欢打羽毛球
我的名字叫张三,性别是 女,体重现在是 54.0
我的名字叫李四,性别是 男,体重现在是 75.0
```

【例 5-14】 self 的使用(2)。

```
class People(object):                              #创建一个 People 类
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def gohome(self):                              #类方法 1
        print("{},{},{}放学回家".format(self.name,self.age,self.gender))
    def travel(self):                              #类方法 2
        print("{},{},{}和家人开车去旅游" .format(self.name,self.age,self.gender))
    def work(self):                                #类方法 3
        print("{}.{},{}大学毕业准备去工作".format(self.name,self.age,self.gender))
Liuming = People("刘明",18,"男")                   #实例对象
zhangfan = People("张帆",22,"男")
Lisisi = People("李思思",10, "女")
Liuming.travel()                                   #调用
zhangfan.work()
Lisisi.gohome()
```

运行结果为

```
刘明,18,男,和家人开车去旅游
张帆,22,男,大学毕业准备去工作
李思思,10,女,放学回家
```

【例 5-15】 使用 append()函数添加属性。

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []                           #为 Dog 类创建一个空列表
    def add_trick(self, trick):
        self.tricks.append(trick)
d = Dog('泰迪')
e = Dog('松狮')
d.add_trick('在翻跟头')
e.add_trick('在玩球')
print(d.name,d.tricks)
print(e.name,e.tricks)
```

运行结果为

```
泰迪 ['在翻跟头']
松狮 ['在玩球']
```

【例 5-16】 类变量与实例变量的使用。

```
class A:                                    #定义 A 类
    object_a = 1                            #定义类变量
    def __init__(self, x, y):               #x 和 y 是实例变量
        self.x = x
        self.y = y
if __name__ == '__main__':
    print(A.object_a)                       #输出类变量
    a = A(2, 3)
    print(a.x, a.y, a.object_a)             #输出实例变量和类变量
    A.object_a = 10                         #在实例中对类变量重新赋值
    print(a.object_a, A.object_a)           #输出修改后的值
```

运行结果为

```
1
2 3 1
10 10
```

【例 5-17】 定义一个 Home 类，其中摆放的家具有床（占 $6m^2$）、衣柜（占 $4m^2$）、写字台（占 $3m^2$），将 3 件家具添加到两居室房子中，要求输出户型、总面积、剩余面积、家具名称列表。

```
class Home:
    def __init__(self, name, area):
        self.name = name
        self.area = area
    def __str__(self):
        return '[%s] 占地 %.2f' % (self.name, self.area)
class House:
    def __init__(self, house_type, area):
        self.house_type = house_type
        self.area = area
        self.free_area = area
        self.item_list = []
    def __str__(self):
        return '户型:%s\n总面积:%.2f[剩余:%.2f]\n家具:%s' % (self.house_type, self.
area, self.free_area, self.item_list)
    def add_item(self, item):
        print('要添加 %s' % item)
        if item.area > self.free_area:
            print('%s 的面积太大了,无法添加' % item.name)
            return
        self.item_list.append(item.name)
```

```
            self.free_area -= item.area
bed = Home('床', 6)
print(bed)
chest = Home('衣柜', 4)
print(chest)
table = Home('写字台', 3)
print(table)
my_home = House('两居室', 60)
my_home.add_item(bed)
my_home.add_item(chest)
my_home.add_item(table)
print(my_home)
```

运行结果为

```
[床] 占地 6.00
[衣柜] 占地 4.00
[写字台] 占地 3.00
要添加 [床] 占地 6.00
要添加 [衣柜] 占地 4.00
要添加 [写字台] 占地 3.00
户型:两居室
总面积:60.00[剩余:47.00]
家具:['床', '衣柜', '写字台']
```

【例 5-18】 建立一个学生成绩管理系统，内有 4 个菜单，通过选择菜单完成相应的操作。

```
class Student(object):
    def __init__(self, name, num, score, cname):
        self.name = name
        self.num = num
        self.score = score
        self.cname = cname
    def __str__(self):
        return "姓名:%s, 学号: %s 成绩: %d 课程名: %s " % (self.name, self.num, self.
score, self.cname)
class StudentManage(object):
    Stu = []
    def start(self):
        self.Stu.append(Student('张三', '2022101', 100, "Python 程序设计"))
        self.Stu.append(Student('李四', '2022102', 82, "Python 程序设计"))
        self.Stu.append(Student('王五', '2022103', 91, "Python 程序设计"))
        self.Stu.append(Student('赵六', '2022103', 81, "Python 程序设计"))
    def menu(self):
        self.start()
        while True:
            print("""\t\t\t\t\t**********************
                  学生成绩管理系统
                   1. 成绩查询
```

```
                    2. 增加记录
                    3. 删除记录
                    4. 退出
                    ************************* """)
            choice = input("请选择:")
            if choice == '1':
                num = input("学号:")
                self.checkStu(num)
            elif choice == '2':
                self.addStu()
            elif choice == '3':
                self.delstu()
            elif choice == '4':
                return
            else:
                print("请输入正确的选择!")
    def addStu(self):
        num = input("学号:")
        self.Stu.append(Student(input("姓名:"),num,eval(input("成绩:")), input("课程名称")))
        print("添加学生学号 %s 成功!" % (num))
        for i in range(len(self.Stu)):
           print(self.Stu[i].name,self.Stu[i].num,self.Stu[i].score,self.Stu[i].cname)
    def delstu(self):
        num = input("输入删除的学号:")
        ret = self.checkStu(num)
        if ret != None:
          #self.Stu.pop()
            self.Stu.remove(num)
            print("删除 %s 成功" % (num))
        else:
            print("该学号 %s 不存在!" % (num))
    def checkStu(self,num):
        for Student in self.Stu:
            if Student.num == num:
                print(Student.num,Student.name,Student.score,Student.cname)
                return Student
        else:
            return None
StudentManage = StudentManage()
StudentManage.menu()
```

运行结果为

```
                    ************************
                    学生成绩管理系统
                    1. 成绩查询
                    2. 增加记录
                    3. 删除记录
                    4. 退出
                    ************************
```

```
请选择:1
学号: 2022101
2022101 张三 100 Python 程序设计
                    ************************
                    学生成绩管理系统
                     1. 成绩查询
                     2. 增加记录
                     3. 删除记录
                     4. 退出
                    ************************
请选择:2
学号: 2022108
姓名:合适
成绩:99
课程名称 Python 程序设计
添加学生学号 2022108 成功!
张三 2022101 100 Python 程序设计
李四 2022102 82 Python 程序设计
王五 2022103 91 Python 程序设计
赵六 2022103 81 Python 程序设计
合适 2022108 99 Python 程序设计
                    ************************
                    学生成绩管理系统
                     1. 成绩查询
                     2. 增加记录
                     3. 删除记录
                     4. 退出
                    ************************
请选择:3
输入删除的学号:2022103
删除 2022103 成功
```

5.5 类的封装、继承和多态

封装机制保证了类内部数据结构的完整性,使用户无法看到类中的数据结构,避免了外部对内部数据的影响,提高了程序的可维护性。继承是一种层次模型,对象的新类可从现有类派生,这个过程称为类的继承。新类继承原类的属性,新类称为原类的派生类(子类),原类称为新类的基类(父类),它是允许类重用的一种方法。多态允许不同类的对象响应相同的消息。多态使程序更具灵活性、抽象性,它较好地解决了应用程序中函数、变量的同名问题。

5.5.1 封装及应用案例

封装使用户只能借助类方法访问数据,避免用户对类中属性或方法的不合理操作。对类进行封装还可提高代码的复用性。

【例 5-19】 计算三角形面积，对外界来说不需要知道中间的计算过程，把这些对外界无关紧要的内容封装起来，只需要输入名称，如"三角形"，以及 3 条边长度，即可得到面积。

```
import math
class triangle:
    def __init__(self,name,a,b,c):
        self.name = name                          #名称
        self.__a = a                              #a,b,c 为三角形边长
        self.__b = b
        self.__c = c
    @property
    def area (self):                              #对外提供的接口,封装了内部实现
        s = (self.__a + self.__b + self.__c)/2
        return math.sqrt(s * (s - self.__a) * (s - self.__b) * (s - self.__c))
S1 = triangle("三角形",3,4,5)                     #加入三角形 3 条边
print(S1.name,S1.area)                            #通过接口调用 area()函数,得到三角形面积
```

运行结果为

```
三角形 6.0
```

说明：@property 是 Python 的一种装饰器，用来修饰方法。使用@property 装饰器创建只读属性，@property 装饰器会将方法转换为相同名称的只读属性，可以与所定义的属性配合使用，这样可以防止属性被修改。

【例 5-20】 在 School 类中，对学校名称长度小于 4 和地址开头不包含 http://的输入进行检验，这些判断封装到类方法中。

```
class School:
    def webset(self, name):
        if len(name) < 4:
            raise ValueError('名称长度必须大于 4!')
        self.__name = name
    def getname(self):
        return self.__name                        #定义私有属性,封装属性 name
    name = property(getname, webset)
    def setadd(self, add):
        if add.startswith("http://"):
            self.__add = add
        else:
            raise ValueError('地址必须以 http:// 开头')
    def getadd(self):
        return self.__add                         #定义私有属性,封装 add
    add = property(getadd, setadd)
    def __display(self):                          #定义私有方法
        print(self.__name, self.__add)
sch = School()
sch.name = input("输入学校的名称:")
sch.add = input("输入学校的网址:")
```

```
print(sch.name)
print(sch.add)
```

运行结果如下。

(1) 输入学校名称长度小于 4。

```
输入学校的名称:北京
Traceback (most recent call last):
  File "F:\tools\PythonProject\test1.py", line 22, in <module>
    sch.name = input("输入学校的名称:")
  File "F:\tools\PythonProject\test1.py", line 4, in university
    raise ValueError('名称长度必须大于 4!')
ValueError: 名称长度必须大于 4!
```

(2) 输入网址未以“http://”开头。

```
输入学校的名称:北京大学
输入学校的网址:beijing
Traceback (most recent call last):
  File "F:\tools\PythonProject\test1.py", line 23, in <module>
    sch.add = input("输入学校的网址:")
  File "F:\tools\PythonProject\test1.py", line 14, in website
    raise ValueError('地址必须以 http:// 开头')
ValueError: 地址必须以 http:// 开头
```

(3) 输入正确显示结果。

```
输入学校的名称: 深圳北理莫斯科大学
输入学校的网址: http://www.smbu.edu.cn
深圳北理莫斯科大学
http://www.smbu.edu.cn
```

说明:

(1) 将对学校名称长度小于 4 或地址未以“http://”开头的输入判断封装到了类方法中。

(2) School 类中将 name 和 add 属性都隐藏起来,通过 webset()方法判断 name 的长度,调用 startswith()方法(见 3.2.1 节),控制用户输入的地址必须以“http://”开头,否则程序将会抛出异常。

(3) 通过对 webset()和 setadd()方法进行适当的设计,可以避免用户对类中属性不合理操作,从而提高了类的可维护性和安全性。

5.5.2 继承及应用案例

继承的作用是实现代码重用,相同的代码不用重复编写。

1. 继承的描述

子类(派生类)拥有父类(基类)的所有方法和属性,即一个派生类(Derived Class)继承基类(Base Class)的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。

例如，Dog 类的对象派生自 Animal 类，Student 类的对象派生自 Person 类，它们都继承 Object 类。继承的表示如图 5-3 所示。

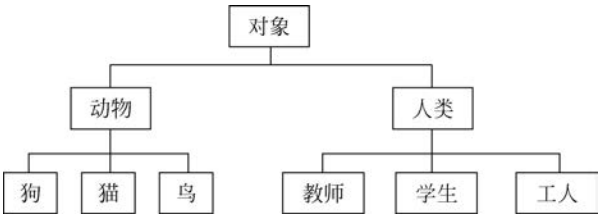

图 5-3　继承的表示

Python 完全支持继承、重载、派生、多重继承，有益于增强源代码的复用性。

2. 继承的特点

(1) 在继承中基类的构造函数(__init__()方法)不会被自动调用，它需要在其派生类的构造函数中专门调用。

(2) 在调用基类的方法时，需要加上基类的类名前缀，且需要带上 self 参数变量。在类中调用普通函数时并不需要带上 self 参数。

(3) Python 总是首先查找对应类型的方法，如果系统不能在派生类中找到对应的方法，才开始到基类中逐个查找。也就是说，系统顺序是先在本类中查找调用的方法，如果找不到才去基类中查找。

(4) 如果一个类不继承其他类，就显式地从 Object 类(基类)继承。嵌套类也一样。

3. 继承类定义

继承类的语法格式为

```
class 继承类名(基类名):
    <语句 1>
    <语句 2>
    ...
    <语句 N>
```

其中，基类名必须定义于包含派生类定义的作用域中，也允许用其他任意表达式代替基类名称所在的位置。当基类定义在另一个模块中时，方法是

```
class 派生类名称(方法名. 基类名):
```

派生类定义的执行过程与基类相同。当构造类对象时，基类会被记住。此信息将被用来解析属性引用：如果请求的属性在类中找不到，搜索将转往基类中进行查找。如果基类本身也派生自其他某个类，则此规则将被递归地应用。

派生类名是创建该类的一个新实例。派生类可能会重写其基类的方法，所以调用同一基类中定义的另一个基类方法，最终可能会调用覆盖它的派生类方法。在派生类中的重载方法是使用父类的同名方法扩展内容。

直接调用基类方法，语法格式为

```
基类名.方法名(self, 数值序列)
```

仅当此基类在全局作用域中以基类的名称被访问时方可使用此方式。

4. 父类与子类的关系

子类与父类对象是继承关系，判断对象之间的关系，可以通过 isinstance(变量，类型)函数进行判断。例如：

```
class Person(object):
  pass
class Student(Person):
  pass
per1 = Person()
per2 = Student()
```

其中，子类(Student)与父类(Person)的关系是 is 的关系，即 Student 继承于 Person 类，则 per1 是 Person 类的实例，但它不是 Student 类的实例；per2 是 Person 类的实例，per2 也是 Student 类的实例。

判断方法为

```
print("per1 is Person",isinstance(per1,Person))
print("per1 is Student",isinstance(per1,Student))
print("per2 is Person",isinstance(per2,Person))
print("per2 is Student",isinstance(per2,Student))
```

运行结果为

```
per1 is Person True
per1 is Student False
per2 is Person True
per2 is Student True
```

5. 继承调用规则

【例 5-21】 父类和子类的调用顺序。

```
class Parent:                                  #定义父类
   parentAttr = 100                            #父类变量
   def __init__(self):                         #父类初始化
      print("调用父类构造函数")
   def parentMethod(self):                     #父类方法
      print('调用父类方法')
   def setAttr(self, attr):
      Parent.parentAttr = attr
   def getAttr(self):
      print("父类属性 :", Parent.parentAttr)
class Child(Parent):                           #定义子类
   def __init__(self):                         #子类初始化
      print("调用子类构造方法")
   def childMethod(self):                      #子类方法
      print('调用子类方法')
c = Child()                                    #实例化子类
```

```
c.childMethod()                    # 调用子类的方法
c.parentMethod()                   # 调用父类方法
c.setAttr(200)                     # 再次调用父类的方法,设置属性值
c.getAttr()                        # 再次调用父类的方法,获取属性值
```

运行结果为

```
调用子类构造方法
调用子类方法
调用父类方法
父类属性 : 200
```

6. 继承的应用案例

Python 中常使用 super()函数继承父类的函数,它是一个内置函数,用于调用父类中的方法,语法格式为

```
super(类型[, self])
```

self 会首先调用自己的方法或属性,当自身没有目标属性或方法时,再去父类中寻找;super()函数会直接去父类中寻找目标属性或方法,多用来处理多重继承问题中直接用类名调用父类方法时的查找顺序问题。

【例 5-22】 继承的简单应用。

```
class Father(object):
    def __init__(self,profession):
        profession = '教师'
        print(profession)
class Son(Father):
    def __init__(self):
        print('儿子职业是:')
        super().__init__(self)
a = Son()
```

运行结果为

```
儿子职业是:
教师
```

【例 5-23】 继承函数的使用。

```
class Animal(object):
  def run(self):
     print('动物正在跑呢...')
class Dog(Animal):
    print('这只狗正在跑呢...')
class Cat(Animal):
    print('这只猫正在跑呢...')
dog = Dog()
dog.run()
cat = Cat()
cat.run()
```

运行结果为

```
这只狗正在跑呢...
这只猫正在跑呢...
动物正在跑呢...
动物正在跑呢...
```

说明：继承的好处是子类获得了父类的全部功能。由于 Animal 类实现了 run()方法，因此，Dog 和 Cat 类自动拥有了 run()方法。

【例 5-24】 对例 5-23 进行修改，当子类和父类都存在相同的 run()方法时，子类的 run()方法覆盖了父类的 run()方法，代码运行时，总是会调用子类的 run()方法。

```
class Animal(object):
  def run(self):
    print('动物正在跑呢...')
class Dog(Animal):
  def run(self):
    print('狗在快速追赶呢...')
  print('这只狗正在跑呢...')
class Cat(Animal):
  def run(self):
    print('猫在快速逃跑呢...')
  print('这只猫正在跑呢...')
dog = Dog()
dog.run()
cat = Cat()
cat.run()
```

运行结果为

```
这只狗正在跑呢...
这只猫正在跑呢...
狗在快速追赶呢...
猫在快速逃跑呢...
```

【例 5-25】 使用父类与子类的继承。

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
        print(self.name, self.gender)
class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
        print(self.name,self.gender,self.score)
stu = Student('张三,','男,',85)
```

运行结果为

```
张三, 男,
张三, 男, 85
```

Student 类需要有 name 和 gender 属性，若 Person 类中有该属性，直接继承即可。另外，需要新增 score 属性，继承类时一定要使用 super(子类名，self).__init__(子类需要继承父类的参数)初始化父类，否则继承父类的子类将没有 name 和 gender 属性。当调用父类时，即

```
super(子类名,self)
```

将返回当前类继承的父类。然后再调用__init__()方法。此时__init__()方法已经不需再传入 self 参数，在 super()函数中已经传入。

7. 父类与子类的类型及应用案例

子类类型可以向上转型看作父类类型(子类是父类)；父类类型不可以向下转型看作子类类型，因为子类类型多了一些自己的属性和方法。

【例 5-26】 父类和子类的使用。

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course
t = Teacher('姜增如', '女', 'Python 语言')
s = Student('玛丽', '女', 98)
p = Person('陶木', '男')
print('姓名:{},性别:{},教授课程:{},正在上课'.format(t.name, t.gender,t.course))
print('姓名:{},性别:{},成绩:{},放学回家了'.format (s.name, s.gender,s.score))
print('姓名:{},性别:{} '.format(p.name, p.gender))
print(isinstance(t, Person))        #True
print(isinstance(t, Student))       #False
print(isinstance(t, Teacher))       #True
print(isinstance(t, object))        #True
print(isinstance(p, Student))       #False
print(isinstance(p, Teacher))       #False
```

运行结果为

```
姓名:姜增如,性别:女,教授课程:Python 语言,正在上课
姓名:玛丽,性别:女,成绩:98,放学回家了
姓名:陶木,性别:男
```

```
True
False
True
True
False
False
```

8. 方法重写及应用案例

方法的重写一般用于在继承父类的基础上，添加了新的属性，其特点如下。

（1）如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（Override），也称为方法的重写。

（2）如果在子类定义的方法中，其名称、返回类型及参数列表正好与父类中的相匹配，则子类的方法重写了父类的方法。

（3）方法重写在不同类中是实现多态的必要条件，即子类重写父类的方法，使子类具有不同的方法实现。Python 可从一个父类派生出多个子类，可以使子类之间有不同的行为，这种行为称为多态。子类与父类拥有同一个方法，子类的方法优先级高于父类，即子类覆盖父类。

【例 5-27】 方法重写。

```
class Person(object):
    def __init__(self, name, sex):
        self.name = name
        self.sex = sex
    def whoAmI(self):
        return '我是外国人, 我的名字是 %s ,我是%s生' % (self.name,self.sex)
class Student(Person):
    def __init__(self, name, sex, score):
        super(Student, self).__init__(name, sex)
        self.score = score
    def whoAmI(self):
        return'我是一个学生,我的名字是%s,我是%s生,Python成绩是%d' % (self.name,self.
sex,self.score)
p = Person('杰克', '男')
s = Student('玛利亚', '女',88)
print(p.whoAmI())
print(s.whoAmI())
```

运行结果为

```
我是外国人, 我的名字是 杰克 ,我是男生
我是一个学生,我的名字是 玛利亚,我是女生,Python 成绩是 88
```

说明：方法调用将作用在对象的实际类型上，对于 Student 类，它实际上拥有自己的 whoAmI()方法以及从 Person 类继承的 whoAmI()方法，但调用 s.whoAmI()方法时总是先查找自身的定义，如果没有定义，则顺着继承链向上查找，直到在某个父类中找到为止。这种方法调用就称为多态。

【例 5-28】 继承与方法的重写。

```
class A():
    def __init__(self):
      print('这是 A 类的初始化属性')
    def fun1(self):
      print('这是 A 类的方法')
class B(A):
    def __init__(self):
      print('这是 B 类的初始化属性')
      super().__init__()            #super()函数引入 A 类的初始化属性
    def fun2(self):
      print('这是 B 类的方法')
b = B()
b.fun1()                            #B 继承 A 的 fun1 方法
b.fun2()
print("B 继承 A 类吗?",issubclass(B, A))
```

运行结果为

```
这是 B 类的初始化属性
这是 A 类的初始化属性
这是 A 类的方法
这是 B 类的方法
B 继承 A 类吗? True
```

5.5.3 多重继承及应用案例

多重继承是一个子类同时拥有多个父类，使用时在类名的括号中添加多个类，实现多重继承。

1. 多重继承的规则

Python 支持多重继承，一个类的方法和属性可以定义在当前类，也可以定义在基类，当调用类方法或类属性时，就需要对当前类和基类进行搜索，以确定方法或属性的位置，而搜索的顺序就称为方法解析顺序(Method Resolution Order，MRO)。单继承 MRO 很简单，就是从当前类开始，逐个搜索它的父类；而对于多重继承，MRO 相对会复杂一些，它的规则顺序如下。

(1) 子类永远在父类的前面。

(2) 如果有多个父类，会根据它们在列表中的顺序检查。

(3) 如果对下一个类存在两种不同的合法选择，那么选择第 1 个父类。

2. 多重继承语法及使用

多重继承语法格式为

```
class 父类名
    def __init__(self):
        ...
class 子类名(父类名 1, 父类名 2, 父类名 3):
    def __init__(self):
      super(B,self).__init__()
```

（1）使用 super()函数可以逐一调用所有父类方法，并且只执行一次。调用顺序遵循 MRO 类属性的顺序，其写法包括：

```
super(子类名, self).父类方法(参数列表)
super(子类名, self).__init__()                    #执行父类的 __init__()方法
super().父类方法(参数列表)                          #执行父类的实例方法
super().__init__()                                #执行父类的 __init__()方法
```

（2）如果多个父类中有同名的属性和方法，则默认使用第 1 个父类的属性和方法。

（3）指定执行父类的方法，无论何时何地，self 都表示子类的对象。在调用父类方法时，通过传递 self 参数控制方法和属性的访问修改。

（4）当父类方法不能满足子类的需求时，可以对方法进行重写或覆盖父类的方法。

【例 5-29】 多重继承的使用顺序。

```
class A(object):
    def __init__(self, a):
        print('这里初始化 A 类');
        self.a = a
class B(A):
    def __init__(self, a):
        super(B, self).__init__(a);
        print('这里初始化 B 类 ')
class C(A):
    def __init__(self, a):
        super(C, self).__init__(a);
        print('这里初始化 C 类')
class D(B, C):
    def __init__(self, a):
        super(D, self).__init__(a);
        print('这里初始化 D 类')
x = A("输出 A 类的内容");print(x.a)
y = B("输出 B 类的内容");print(y.a)
z = C("输出 C 类的内容");print(z.a)
k = D("输出 D 类的内容");print(k.a)
```

运行结果为

```
这里初始化 A 类
输出 A 类的内容
这里初始化 A 类
这里初始化 B 类
输出 B 类的内容
这里初始化 A 类
这里初始化 C 类
输出 C 类的内容
这里初始化 A 类
这里初始化 C 类
这里初始化 B 类
这里初始化 D 类
输出 D 类的内容
```

说明：D 类同时继承自 B 类和 C 类，B 类和 C 类又继承自 A 类，因此 D 类拥有了 A、B 和 C 类的全部功能。如果没有多重继承，就需要在 D 类中写入 A、B、C 类的所有功能。

【例 5-30】 多重继承的使用。

```
class Father(object):                          #父类 1
  def __init__(self,name):
    self.name = name
  def play(self):
    print("喜欢踢足球")
  def prof(self):
    print("职业是教师")
class Mother(object):                          #父类 2
  def __init__(self,name):
    self.name = name
  def eat(self):
    print("喜欢吃甜食")
  def frof(self):
    print("职业是会计")
class Children(Father,Mother):                 #继承父类 1 和父类 2
  def __init__(self,name):
    Father.__init__(self,name)                 #多重继承时调用父类的属性
    Mother.__init__(self,name)                 #多重继承时调用父类的属性
sp = Children('李子')
print(sp.name)
sp.play()
sp.eat()
sp.prof()                                      #继承第 1 个父类的值
```

运行结果为

```
李子
喜欢踢足球
喜欢吃甜食
职业是教师
```

【例 5-31】 若有 Person、Child 和 Baby 3 个类，使用 Baby 类继承 Child 类，再将 Child 类继承 Person 类的递推方式实现多重继承。

```
class Person(object):                          #父类 Person
    def __init__(self, name, sex):
        self.name = name
        self.sex = sex
class Child(Person):                           #继承 Person 类
  def prt(self):
    if self.sex == 'male':
      print("这个孩子是个男孩")
    else:
      print("这个孩子是个女孩")
class Baby(Child):                             #继承 Child 类
    pass
```

```
may = Baby('马丽亚', '女孩')
print(may.name, may.sex)
may.prt()
```

运行结果为

```
马丽亚 女孩
这个孩子是个女孩
```

【例 5-32】 多重继承中 MRO 的使用。

```
class A:
    def __init__(self):
         print("在 A 类中输出!")
class B(A):
    def __init__(self):
        super(B,self).__init__()
        print("在 B 类中输出!")
class C(A):
    def __init__(self):
        super(C,self).__init__()
        print("在 C 类中输出!")
class D(B,C):
    def __init__(self):
         super(D,self).__init__()
         print("在 D 类中输出!")
print(D())
```

运行结果为

```
在 A 类中输出!
在 C 类中输出!
在 B 类中输出!
在 D 类中输出!
<__main__.D object at 0x000001F02BA9BDC0>
```

(1) 实例化 D 类以后,运行__init__()方法,然后运行 super()方法,由于 super()方法的特性,传入参数 D 时,super()方法会通过 MRO 寻找到下一个索引值作为 D 类的父类,即 B 类。

(2) D 类中的 super()方法会执行 B 类中的__init__()方法,可以看到 B 类中也有一个 super()方法,通过 D 类的 MRO 列表,可以看到 B 类的下一个索引值是 C 类。

(3) B 类中的 super()方法会执行 C 类中的__init__()方法,C 类中也有 super()方法,C 类的下一个索引值是 A 类。

(4) C 类中的 super()方法会执行 A 类中的__init__()方法, A 类中没有 super()方法,程序执行 A 类中的 print()函数,然后执行 C 类中的 print()函数,再执行 B 类中的 print()函数,最后执行 D 类中的 print()函数。因此,程序输出结果为 A、C、B、D 类的输出顺序。

(5) 子类永远在父类的前面,若有多个父类,会根据它们在列表中的顺序检查,例 5-32

中的D类实例化，D类继承了B类和C类，根据上述规则，D类的父类就是B，B类的父类是A，但由于子类永远在父类前面，因为C类的父类也是A。因此，正确的MRO顺序是D-B-C-A-对象（对象是所有类的父类）。

3. 重写与覆盖

多个父类有同名属性和方法，子类的方法属性MRO决定了属性和方法的查找顺序。当子类重写父类的属性和方法时，若子类和父类的方法名和属性名相同，则默认使用子类重写父类，称为子类重写父类的同名方法和属性。使用重写的目的是当子类发现父类的大部分功能都能满足需求，但是有一些功能不满足，则子类可以重写父类方法。若重写之后发现仍然需要父类方法，则可以强制调用父类方法。

【例5-33】 覆盖与重写的使用。

```
class Animal():
    def eat(self):
        print('吃')
    def drink(self):
        print('喝')
    def run(self):
        print('跑')
    def sleep(self):
        print('睡')
class Cat(Animal):
    def shout(self):
        print('喵')
class Hellokitty(Cat):
    def speak(self):
        print('可以说外语')
    def shout(self):
        print('喊叫出多种声音')
kt = Hellokitty()
kt.shout()
```

运行结果为

```
喊叫出多种声音
```

（1）如果子类中重写了父类的方法，在运行时只会调用在子类中重写的方法。

（2）多重继承的目的是从两种继承树中分别选择并继承出子类，以便组合功能使用。如果继承了多个父类，且父类都有同名方法，则默认只执行第1个父类的同名方法且只执行一次。

5.5.4　多态及应用案例

多态是一种机制，它在类的继承中得以实现，在类的方法调用中得以体现。

1. 多态的概念

多态是一种使用对象的方式，子类重写父类方法，调用不同子类对象的相同父类方法，

可以产生不同的执行结果，即不同对象调用同一个方法，实现的功能不同。多态的好处是调用灵活，有了多态，更容易编写出通用的代码程序，以适应不断变化的需求，增强程序的复用性。

例如，Python 中的＋运算符，作用在数字上时表示数值相加，如 1＋2＝3；作用在字符串上时表示字符串的拼接，如'a' ＋ 'b' ＝ 'ab'；作用在列表上时表示列表的拼接，如[1]＋[2]＝[1,2]。

说明：多态如同加号一样，同样的方法用在不同对象上，实现的功能完全不一样。

2. 多态应用案例

【例 5-34】 多态的简单应用，输出不同对象的行进速度。

```
class Car(object):
  def speed(self):
    print("汽车行进速度可达到每小时 120 千米")
class Person(object):
  def speed(self):
    print("步行行进速度可达到每小时 6 千米")
class Bike(object):
  def speed(self):
    print("自行车行进速度可达到每小时 30 千米")
def speed(obj):
  obj.speed()
car = Car()
speed(car)
bike = Bike()
speed(bike)
per = Person()
speed(per)
```

运行结果为

```
汽车行进速度可达到每小时 120 千米
自行车行进速度可达到每小时 30 千米
步行行进速度可达到每小时 6 千米
```

【例 5-35】 针对同一方法 say()，根据人群的不同职业，输出其职责。

```
class Person(object):
  def __init__(self,name):
    self.name = name
  def say(self):
    print(f"{self.name}需要做好自己的本职工作!")
class people(Person):
    pass
class Teacher(Person):
  def say(self):
    print(f"{self.name}要认真备课,组织好课堂教学!")
class Student(Person):
  def say(self):
```

```
    print(f"{self.name}需要认真听课,做到课前预习和课后复习!")
p = people("任何人")
p1 = Teacher("李老师")
p2 = Student("华仔")
p.say()
p1.say()
p2.say()
```

运行结果为

```
任何人需要做好自己的本职工作!
李老师要认真备课,组织好课堂教学!
华仔需要认真听课,做到课前预习和课后复习!
```

【例 5-36】 针对同一方法 printest(),输出不同人的爱好。

```
class base(object):
  def __init__(self, name):
    self.name = name
  def printest(self):
    print("业余时间喜欢跳舞,我的名字是: ", self.name)
class subclass1(base):
  def printest(self):
    print("业余时间喜欢画画,我的名字是: ", self.name)
class subclass2(base):
  def printest(self):
    print("业余时间喜欢旅游,我的名字是: ", self.name)
class subclass3(base):
    pass
def testFunc(test):
  test.printest()
testFunc(subclass1("王明敏"))
testFunc(subclass2("马出生"))
testFunc(subclass3("张熙睿"))
```

运行结果为

```
业余时间喜欢画画,我的名字是: 王明敏
业余时间喜欢旅游,我的名字是: 马出生
业余时间喜欢跳舞,我的名字是: 张熙睿
```

【例 5-37】 多态的使用,不同对象产生不同的运行结果。

```
class Dog(object):                          #定义 Dog 类
    def work(self):                         #父类提供统一的方法,哪怕是空方法
        pass
class ArmyDog(Dog):                         #继承 Dog 类
    def work(self):                         #子类重写方法,并且处理自己的行为
        print('追击敌人')
class DrugDog(Dog):
    def work(self):
        print('追查毒品')
```

```
class Person(object):                    #建立 Person 类
    def work_with_dog(self, dog):
        dog.work()                       #对象不同,产生不同的运行结果
dog = Dog()                              #子类对象可以当作父类来使用
print(isinstance(dog, Dog))              #输出
ad = ArmyDog()                           #建立子类对象
print(isinstance(ad, Dog))               #输出
dd = DrugDog()                           #建立子类对象
print(isinstance(dd, Dog))               #输出
p = Person()                             #建立 Person 类对象
p.work_with_dog(dog)
p.work_with_dog(ad)                      #同一个方法,只要是 Dog 类的子类就可以传递
p.work_with_dog(dd)          #传递不同对象,最终 work_with_dog()方法产生了不同的运行结果
```

运行结果为

```
True
True
True
追击敌人
追查毒品
```

最终运行结果中,Person 类中只需要调用 Dog 对象的 work()方法,而不关心具体是什么狗。work()方法是在 Dog 父类中定义的,子类重写并处理不同方式的实现,在程序执行时,传入不同的 Dog 对象作为实参,就会产生不同的运行结果。

【例 5-38】 继承和多态的联合使用。

```
class Car(object):                                  #定义一个汽车类
  def __init__(self,type,No):                      #初始化属性,汽车品牌、车牌号
      self.type = type
      self.No = No
  def start(self):
      print('汽车准备出发了!')
  def stop(self):
     print('乘客到站请下车!')
class Taxi(Car):                                    #定义出租车类调用父类 Car
  def __init__(self,type,No,company,name):          #继承父类方法,并且补充自己的类属性
     super().__init__(type,No)                      #继承调用父类属性
     self.company = company
     self.name = name
  def start(self):                                  #重写类方法
     print('乘客您好,欢迎!')
     print(f'我是{self.name},欢迎乘坐出租车')
     print(f'我是{self.company}出租车公司的,我的车牌号是{self.No},请问您要去哪里?')
  def stop(self):
     print('目的地到了,请您付款下车,欢迎再次乘坐!')
class MyCar(Car):                                   #定义一个私家车子类
  def __init__(self,type,No,name):
     super().__init__(type,No)
     self.name = name
```

```
    def stop(self):                    #重写类方法
        print('我们已经到了深圳湾公园,这里很不错')
    def start(self):                   #重写类方法
        print(f'我是{self.name},我的私家车是{self.type},车牌号是{self.No}')
TaxiCar = Taxi('特斯拉 A5','粤 B6828','深圳神马传奇','出租司机')
TaxiCar.start()
TaxiCar.stop()
print("------"*10)
MyCar = MyCar('比亚迪 S8','粤 B6862','公司职员')
MyCar.start()
MyCar.stop()
```

运行结果为

```
乘客您好,欢迎!
我是出租司机,欢迎乘坐出租车
我是深圳神马传奇出租车公司的,我的车牌号是粤 B6828,请问您要去哪里?
目的地到了,请您付款下车,欢迎再次乘坐!
----------------------------------
我是公司职员,我的私家车是比亚迪 S8,车牌号是粤 B6862
我们已经到了深圳湾公园,这里很不错
```

结论：面向对象中所建立的对象，注重描述一个事物在整个问题要实现的功能。而面向过程注重的是过程的逐步实现，为完成某个步骤对函数逐步分析解决功能问题。