第四章 面向对象

面向对象编程是目前比较流行的程序设计方法。在面向对象程序设计中，数据和对数据的操作可以封装在一个独立的数据结构中，这个数据结构就是对象。对象之间通过消息的传递来进行相互作用。

4.1 面向对象的基本概念

4.1.1 对象

对象是现实世界中的任何事物。对象不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

- 对象的状态或特征用数据值表示出来就是属性
- 对象的操作用于改变对象的状态，这些操作就是方法。

4.1.2 类

类是对象的模板，是对一组具有相同属性和相同操作的对象的抽象。类实际上是一种数据类型，一个类所包含的数据和方法用于描述一组对象的共同属性和行为。类的属性是对象状态的抽象，类的操作是对象行为的抽象。

类是对象的抽象化，而对象则是类的具体化，是类的实例。

4.1.3 消息

一个程序中通常包含多个消息，不同对象之间通过消息相互联系、相互作用。

消息由某个对象发出，用于请求另一个对象执行某项操作，或者回复某些消息。

在发送者将一个消息发送给某个对象时，消息包含接收对象去执行某种操作的信息。

发送一条消息至少需要：

- 接受消息的对象名（对象名）
- 接受对象要执行的操作的名称（方法名）
- 还可以包含参数。参数可以是认识该消息的对象所知道的变量名，或者是所有对象都知道的全局变量名

4.1.4 封装

封装是指将对象的数据和操作数据的过程结合起来所构成的单元，其内部消息对外部是隐藏的，外界不能直接访问对象的属性，而只能通过类对外提供的接口对类进行进行各种操作，这样可以保证程序中数据的安全性。

类是实施数据封装的工具，对象则是封装的具体实现，是封装的基本单位。

定义类时将其成员分为：

- 公有成员
- 私有成员

- 保护成员
  这样形成了类的访问机制，使得外界不能随意存取对象的内部数据。

4.1.5 继承

继承是指在一个类的基础上定义一个新的类。原有的类称为基类、超类或父类，新生成的类称为派生类或子类。

子类不仅可以通过继承从父类中得到所有的属性和方法，也可以对所得到的这些方法进行重写和覆盖，还可以添加一些新的属性和方法，从而扩展父类的功能。

父类体现出对象的共性和普遍性，子类则体现出对象的个性和特殊性。

继承反应了抽象程度不同的类之间的关系，即共性和个性的关系，普遍性和特殊性的关系。

4.1.6 多态

多态是指一个名称相同的方法产生了不同的动作行为，即不同对象收到相同的消息时产生了不同的行为方式。

多态可以通过覆盖和重载两种方式来实现：

- 覆盖：在子类中重新定义父类的成员方法
- 重载：允许存在多个同名函数，而这些函数的参数列表有所不同。

4.1.7 面向对象编程的基本步骤

1. 通过定义类来设置数据类型的的数据和行为
2. 基于类创建对象
3. 通过存取对象的属性或调用对象的方法来完成所需的操作

4.2 类与对象

类是一种自定义复合数据类型。

4.2.1 类的定义

类可以通过class语句来定义，其语法格式如下：

```
class 类名:
    类体
```

类体中定义类的变量成员和函数成员。

- 变量成员即类的属性，用于描述对象的状态和特征
- 函数成员即类的方法，用于实现对象的行为和操作

通过定义类可以实现数据和操作的封装。

在python中，一节皆对象。定义类时便创建了一个新的自定义类型对象，简称类对象。类名即指向类对象。此时可以通过类名和圆点运算符”.“来访问类的属性，其语法格式如下：

```
类名.属性名
```

【例题】 类定义示例

```
class Student:
    name="张三"
    gender ="男"

print(Student.name) # 访问类的属性
```

4.2.2 创建对象

类是对象的模板，对象是类的实例。定义类之后，可以通过赋值语句来创建类的实例对象，其语法如下：

```
对象名=类名(参数)
```

创建对象之后，该对象就拥有类中定义的所有属性和方法，此时可以通过对象名和圆点运算符来访问这些属性和方法，其语法格式如下：

```
对象名.属性名
对象名.方法名(参数)
```

【例题】利用类和对象计算圆的周长和面积

```
import math


class Circle:  # 定义类
    radius = 0  # 定义类的属性

    def getPerimeter(self):  # 定义类的方法。参数self代表类的实例对象
        return 2 * math.pi * self.radius

    def getArea(self):
        return math.pi * self.radius * self.radius


if __name__ == "__main__":
    c1 = Circle()  # 创建类的实例对象
    c1.radius = 10  # 设置对象的属性值
    print("圆的半径为：{}".format(c1.radius))
    print("周长={0},面积={1}".format(c1.getPerimeter(), c1.getArea()))  # 调
用类的方法
```

4.3 属性与方法

4.3.1 成员属性

在类中定义的变量成员就是属性。属性按所属的对象可以分为类属性和实例属性

- 类属性：类对象所拥有的属性，属于该类的所有实例对象
- 实例属性：类的实例对象所拥有的属性，属于该类的某个特定实例的属性

(1) 类属性

类属性按能否在外部访问可以分为公有属性和私有属性。它们都可以通过在类中定义成员变量来创建，创建类之后也可以在类定义的外部通过类名和圆点运算符来添加公有属性，定义属性时，如果属性名以双下画线"__"开头，则该属性就是私有属性，否则就是公有属性

在类的所有方法之外（即属性），无论是公有属性还是私有属性都可以通过变量名来访问，在类的成员方法内部则要通过"类名.属性名"形式来访问。 在类的外部，公有属性仍然可以通过"类名.属性名"形式来访问，私有属性则不能通过这种形式来访问，如果一定要类的外部访问类的私有属性，则必须使用一个新的属性名来访问该属性。这个新的属性名以一条下划线开头，后跟类名和私有属性名、 例如，在MyClass类的内部定义了一个名为**attr的私有成员属性。则在类成员方法之外可以直接通过**attr形式访问这个私有属性，在类成员方法中则通过以下相似来访问：

```
MyClass__attr
```

在类的外部，这个私有属性必须通过以下形式来访问：

```
MyClass_MyClass__attr
```

【类属性示例】

```
from inspect import isfunction


class DemoClass:
    pub_attr = 111  # 定义公有属性
    __priv_attr = 222  # 定义私有属性
    attr = pub_attr + __priv_attr  # 定义公有属性，引用了公有属性和私有属性

    def showAttrs(self):  # 定义类的实例方法，self表示当前实例
        for x in self.__class__.__dict__.items():  # 遍历类的属性和方法组成的字
典
            if isfunction(x[1]):  # 判断是否函数
                print("成员方法：{}".format(x[0]))
            elif type(x[1]) == int:
                if x[0].find("_DemoClass") == -1:  # 判断是否公有属性
                    print("公有属性：{}，值为{}".format(x[0], x[1]))
```

```
                else:
                    print("私有属性：{}，值为{}".format(x[0], x[1]))


if __name__ == "__main__":
    DemoClass().showAttrs()  # 调用实例方法
    print("-" * 30)
    DemoClass.pub_attr = 444  # 类外部对公有属性赋值
    DemoClass._DemoClass__priv_attr = 333  # 类外部对私有属性赋值
    DemoClass.attr3 = 555  # 添加新的公有属性
    DemoClass().showAttrs()
```

运行结果：

```
公有属性：pub_attr，值为111
私有属性：_DemoClass__priv_attr，值为222
公有属性：attr，值为333
成员方法：showAttrs
------------------------------
公有属性：pub_attr，值为444
私有属性：_DemoClass__priv_attr，值为333
公有属性：attr，值为333
成员方法：showAttrs
公有属性：attr3，值为555
```

(2) 实例属性

实例属性是某个类的实例所拥有的属性，属于该类的某个特定实例对象。实例属性可以在类的内部或外部通过赋值语句来创建

1. 在类的内部，定义类的构造方法**init**或其他实例方法时，通过在赋值语句中使用self关键字、圆点运算符和属性名类创建实例属性。语法：

```
self.属性名=值
```

- self是类的实例方法的第一个参数，代表类的当前实例。所谓实例方法就是类的实例能够使用的方法。定义实例方法时，必须设置一个用于接受类的实例的参数，而且这个参数必须是第一个参数。

2. 在类的外部，创建类的实例后，可以通过在赋值语句中使用实例对象名、圆点运算符和属性名来创建新的实例属性。语法：

```
对象名.属性名=值
```

```
class Student:
    def __init__(self, name, age):  # 定义构造函数
```

```
        self.name = name  # 定义实例属性
        self.age = age

    def showInfo(self):  # 定义实例方法
        for attr in self.__dict__.items():
            print("{}:{}".format(attr[0], attr[1]))


if __name__ == "__main__":
    stu = Student("zs", 18)  # 类的实例化
    stu.showInfo()  # 调用实例方法
    stu.gender = "male"  # 添加新的实例属性
    print("-" * 30)
    stu.showInfo()  # 调用实例方法
```

运行结果：

```
name:zs
age:18
------------------------------
name:zs
age:18
gender:male
```

(3) 类属性与实例属性的比较

区别：

1. 所属的对象不同，类属性属于类对象本身，可以由类的所有实例共享；实例属性则属性类的某个特定实例。如果存在同名的类属性和实例属性，则两者相互独立，互不影响
2. 定义的方法不同。类属性是在类中所有方法之外定义的，实例属性则是在构造方法或其他实例方法中定义的
3. 引用的方法不同。类属性是通过”类名.属性名“形式引用的，实例属性则是通过”对象名.属性名“形式引用的。

共同点和联系：

1. 类对象和实例对象都是对象。它们所属的类都可以通过**class**属性来获取，类对象属于type类，实例对象则属于创建实例时所调用的类。
2. 类对象和实例属性包含的属性及其值都可以通过**dict**属性来获取，该属性的值是一个字典，每个字典元素中包含一个属性及其值
3. 如果要读取的某个实例属性还不存在，但在类中定义了一个与其同名的类属性。则Python就会以这个类属性的值作为实例属性的值，同时还会创建一个新的实例属性，此后修改实例属性的值时，将不会对同名的类属性产生影响。

4.3.2 成员方法

定义类时，除了定义成员属性之外，还需要在类中定义一些函数，以便对类的成员属性进行操作。 类的成员方法分为：

- 内置方法
- 类方法
- 实例方法
- 静态方法

(1) 内置方法

在python中，每当定义一个类时，系统都会自动地为它添加一些默认的内置方法，这些方法通常由特定的操作触发，不需要显式调用，它们的命名有特殊的约定。 下面介绍两个常用的内置方法：

- 构造方法
- 析构方法

1) 构造方法

构造方法**init**(self,...)是在创建新对象时被自动调用的，可以用来对类的实例对象进行一些初始化的操作。

构造方法支持重载，定义类时可以根据需要重新编写构造方法，如果类中没有定义构造方法，则系统将执行默认的构造方法。

定义构造方法时，第一个参数用于接受类的当前实例，每当创建类的实例时，python会自动将当前实例传入构造方法，因此不必在类名后面的圆括号中写入这个参数。

```
class Car:
    def __init__(self, brand, color, length):
        self.brand = brand
        self.color = color
        self.length = length

    def run(self):
        print("{}的{}在行驶中".format(self.color, self.brand))


if __name__ == "__main__":
    car1 = Car("BMW", "红色", 4909)
    car1.run()  # 红色的BMW在行驶中
```

2) 析构方法

析构方法**del**(self)是在对象被删除之前自动调用的，不需要在程序中显式调用。

析构方法支持重载，通常可以通过该方法执行一些释放资源的操作。

```
class Demo:
    counter = 0

    def __init__(self, name):
```

```
        self.name = name
        self.__class__.counter += 1  # 修改类属性
        print("创建实例{},当前一共有{}个实例".format(self.name, self.counter))

    def __del__(self):
        self.__class__.counter -= 1
        print("删除实例{}当前一共有{}个实例".format(self.name, self.counter))


def func(name):
    print("函数调用开始")
    Demo(name)  # 在函数中创建对象
    print("函数调用结束")


if __name__ == "__main__":
    a = Demo("a")
    func("func")
    print("程序运行结束")
```

运行结果：

```
创建实例a,当前一共有1个实例
函数调用开始
创建实例func,当前一共有2个实例
删除实例func当前一共有1个实例
函数调用结束
程序运行结束
删除实例a当前一共有0个实例
```

(2) 类方法

类方法是类对象本身拥有的成员方法，通常可以用于对类属性进行修改。要将一个成员函数定义为类方法，必须将函数作为装饰器classmethod的目标函数，而且以类本身作为其第一个参数。语法如下：

```
@classmethod
def 函数名(cls,...):
    函数体
```

定义类方法后，可以通过类对象或实例对象来访问它，其语法格式如下：

```
类名.方法名([参数])
对象名.方法名([参数])
```

【例题】

```
class Math:
    __PI = 3.14

    @classmethod
    def getPI(cls):
        return cls.__PI


if __name__ == "__main__":
    print(Math.getPI())
    print(Math._Math__PI)
```

(3) 实例方法

实例方法是类的实例对象所用用的成员方法。定义类的实例方法时，必须以类的实例对象作为第一个参数。语法格式如下：

```
def 函数名(self,...):
    函数体
```

定义实例方法后，只能通过对象名、圆点运算符和方法名来调用它，而且不需要将对象实例作为参数传入方法中。语法格式如下：

```
对象名.函数名([参数])
```

【例题】利用实例方法计算三角形的面积

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def getArea(self):
        # ---海伦公式------
        p = (self.a + self.b + self.c) / 2  # 半周长
        s = (p * (p - self.a) * (p - self.b) * (p - self.c)) ** 0.5
        return s

    def isTrangle(self):
        return self.a + self.b > self.c and self.a + self.c > self.b and
self.b + self.c > self.a


if __name__ == "__main__":
    a, b, c = eval(input("输入三角形的三个边："))
```

```
    tri = Triangle(a, b, c)
    if tri.isTrangle():
        print("三角形的面积为：{:.2f}".format(tri.getArea()))
    else:
        print("不能构成三角形")
```

(4) 静态方法

类中的静态方法即不属于类对象，也不属于实例对象，它只是类中的一个普通的成员函数。静态方法主要是用来存放逻辑性的代码，但是和类本身没有交互，即在静态方法中，不会涉及到类中的方法和属性的操作。可以理解为将静态方法存在此类的名称空间中。

与类方法和实例方法不同，静态方法可以带任意数量的参数，也可以不带任何参数。此外，如果要将类中的一个成员函数定义为静态方法，还必须将其作为装饰器staticmethod的目标函数。语法格式如下：

```
@staticmethod
def 函数名([参数])
    函数体
```

当定义一个类时，可以在类的静态方法中通过类名来访问类属性，但是不能在静态方法中访问实例属性。 在类的外部，可以通过类对象成或实例对象来调用静态方法。语法格式如下：

```
类名.静态方法名([参数])
对象名.静态方法名([参数])
```

```
import time


class Utils:
    @staticmethod
    def getTime():
        return time.strftime("%H:%M:%S", time.localtime())


if __name__ == "__main__":
    print(Utils.getTime())
```

4.3.3 私有方法

默认情况下，在类中定义的各种方法都属于公有方法，可以在类的外部调用这些公有方法。根据需要，与也可在类中创建一些各种类型的私有方法。

在类中创建某种类型的私有方法的过程与创建相同类型的公有方法类似。所不同的是，在定义私有方法时，成员函数名必须以"__"开头。

私有方法只能在类的内部使用，其调用方法与公有方法类似。不允许也不提倡在类的外部使用私有方法，如果一定要在类的外部使用私有方法，则需要使用一个新的方法名，该方法名以"_"下划线开头，后跟类名和私有方法名。

```
class Demo:
    def __priv(self):
        print("this is private function")


if __name__ == "__main__":
    demo = Demo()
    demo._Demo__priv()  # 在类外部调用私有方法
```

4.4 继承

继承是指在一个父类的基础上定义一个新的子类。子类通过继承将从父类中得到所有的属性和方法，也可以对所得到的这些方法重写和覆盖，同时还可以添加一些新的属性和方法，从而扩展父类的功能。

继承关系按父类的多少分为：

- 单一继承：子类从单个父类中继承
- 多重继承：子类从多个父类中继承

4.4.1 单一继承

语法格式：

```
class 子类名(父类名):
    类体
```

基于父类创建新的子类之后，该子类将拥有父类中的所有公有属性和所有成员方法。

除了继承父类的素有公有成员外，还可以在子类中扩展父类的功能，这可以通过两种方式来实现：

- 在子类中增加新的成员属性和成员方法
- 对父类中已有的成员方法进行重定义，从而覆盖父类的同名方法

在某些情况下，可能希望在子类中继续保留父类的功能，此时就需要调用父类的方法。在子类中可以通过父类的名称或super()函数来调用父类的方法。语法:

```
super().要调用的父类的函数名([参数])

父类名称.要调用的父类的函数名([参数])
```

类对象拥有内置的属性：

- **name**：获取类对象的类名
- **bases**：获取类对象所属的若干个父类组成的元祖

实例对象拥有内置的属性：

- **ckass**：获取该对象所属的类

此外，还可以使用内置函数isinstance()函数来判断一个对象是否属于一个已知的类型，此函数类似于内置函数type()。区别在于：

- type()不考虑继承关系，不会认为子类是一种父类类型
- isinstance()会考虑继承关系，会认为子类是一种父类类型

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

    def showInfo(self):
        print("姓名:", self.name, sep="", end="; ")
        print("年龄:", self.age, sep="")

    @classmethod
    def showClass(cls):
        print("当前类名: ", cls.__name__, sep="")  # 获取类名
        print("所属父类: ", cls.__bases__[0].__name__, sep="")  # 获取父类的类
名


# 单一继承
class Student(Person):
    def __init__(self, sid, name, age):
        self.sid = sid
        super().__init__(name, age)  # 调用父类的构造方法

    # 子类中添加新的功能
    def setScore(self, grade):
        self.grade = grade

    # 覆盖父类的方法
    def showInfo(self):
        print("学号:", self.sid, sep="", end="; ")
        super().showInfo()


if __name__ == "__main__":
    person = Person("zs", 34)
    print("*" * 5, "个人信息", "*" * 5)
    person.showInfo()
    Person.showClass()
```

```
    print("=" * 30)
    stu = Student("20190101", "李明", 18)
    print("*" * 5, "个人信息", "*" * 5)
    stu.showInfo()
    stu.showClass()
```

结果：

```
***** 个人信息 *****
姓名:zs; 年龄:34
当前类名: Person
所属父类: object
==============================
***** 个人信息 *****
学号:20190101; 姓名:李明; 年龄:18
当前类名: Student
所属父类: Person
```

4.4.2 多重继承

语法格式：

```
class 子类名(父类名1,父类名2,...)
    类体
```

在多重继承中，子类将从指定的多个父类中继承所有公有成员。为了扩展父类的功能，通常需要在子类中使用super()函数来调用父类中的方法。如果多个父类拥有同名的成员方法，使用super()函数时将会调用哪个父类的方法呢?

要搞清楚这个问题，就需要对python中类的继承机制有所了解。对于继承链上定义的各个类，python将对所有父类进行排列并计算出一个方法解析顺序（Method Resolution Order,或MRO），通过类的**mro**属性可以返回一个元祖，其中包含方法解析顺序的各个类。当调用子类的某个方法时，python将从MRO最左边的子类开始，从左到右依次查找，直至找到所需要的方法为止。如果同一个方法在不同层次的类中都存在，则从前面的类中进行选择，以保证每个父类只继承一次，这样可以避免重复继承。

```
class A:
    def __init__(self):
        print("A__init__", self)

    def say(self):
        print("A say: Hello!", self)

    @classmethod
    def showMRO(cls):
```

```
        print(cls.__name__, cls.__mro__)


class B(A):
    def __init__(self):
        print("B__init__", self)

    def eat(self):
        print("B Eatting.", self)


class C(A):
    def __init__(self):
        print("C__init__", self)

    def eat(self):
        print("C Eatting.", self)


class D(B, C):
    def __init__(self):
        super().__init__()  # 父类B,C中均有构造方法，将调用MRO中B的构造方法
        print("D__init__", self)

    def say(self):
        super().say()  # B和C均无say()方法，从MRO中找到了A中的say()
        print("D say:Hello!", self)

    def dinner(self):
        self.say()  # 将调用自己类的say()
        super().say()  # 将调用A的say()
        self.eat()  # 从MRO中找到了B的say()
        super().eat()  # 从MRO中找到了B的say()
        C.eat(self)  # 忽略MRO 调用C的eat()


if __name__ == "__main__":
    A.showMRO()  # A (<class '__main__.A'>, <class 'object'>)
    B.showMRO()  # B (<class '__main__.B'>, <class '__main__.A'>, <class
'object'>)
    C.showMRO()  # C (<class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
    D.showMRO()  # D (<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>)
    print("*" * 30)
    d = D()
    print("*" * 30)
    d.say()
    print("*" * 30)
```

```
    d.dinner()
```