面向对象特性

封装

1. 任何对象都有明确的边界，把属性保护在边界之内，称之为封装------数据隐藏
2. 边界：保护属性不被外部访问，往往内部也不能访问外部
3. 封装需要通道：接口---用于数据传输

4. 封装的粒度：

    粒度过大：导致对象过于复杂，不利于各司其职

    粒度过小：导致对象过于简单，过程过于复杂

分门别类

5. 封装的必要性：

    1. 面向过程---非结构化编程
    2. 面向对象---结构化编程

    程序员：2w行

    5000个函数

    1000个类

    100个文件（模块）

    10个包

    3个板块

    3. 封装，简化了编程模型，更容易的记忆及调用内容

4. 封装 要体现可重用性

继承

1. 继承是类和类之间的关系
2. 大的类：父类　　小的类： 子类

父类---子类

抽象---特殊

动物（吃，睡）---狗（吃，睡）+猫（吃，睡）+鱼（吃，睡，游泳）

3. 语法：

class 子类类名（父类类名）：

属性

方法

4. 父类： 超类，基类　　子类：子类

super class　　sub class

object是一个类---根类

object类是所有类的父类

- 继承的特点

1. 子类可以继承父类的成员，但是不能继承私有成员

   本质上私有成员也继承下来了，只是访问方式改变

2. 如果子类没有初始化方法，则会调用父类的初始化方法
3. 如果子类和父类都没有初始化方法，则会调用object的初始化方法
4. 如果子类实现了和父类一样的方法，子类的方法遮蔽父类的方法

   子类的方法名和父类的方法名一致，子类的特殊实现遮蔽了父类的一般实现

   ---方法覆盖
5. Python是多继承的，可以继承多个父类

- 补充

1. 循环继承问题

   乱伦问题
2. 钻石继承问题

   多继承问题---菱形继承问题

   如果一个类，有多个子类，这些子类又有相同的子类，此时子类调用父类的初始化方法时，越高级的类，被调用的次数越多，从而造成资源浪费

   解决方案：

   super().__init__( )

   super()指代父类实例对象

   说明：super()的底层使用mro实现的

   mro：继承链式关系

原理：mro中包含所有的继承关系相关的类，并且只会出现一遍，super底层使用mro，保证所有的类，最多只被调用一次，避免了资源浪费

补充：

mro()是一个方法，必定是类的方法（继承是类和类的关系）

mro()的底层： __mro__ 魔法属性

循环继承问题

```
class GrandPa(Father):# 爷爷类
    pass

class Father(GrandPa):
    pass

# class Son(Father,GrandPa):
#     pass

class Son(GrandPa,Father):
    pass

s=Son()
```

```
class GrandPa(object):
    def __init__(self):
        super().__init__() # object的init
```

```
        print('我是爷爷')

class Father(GrandPa):
    def __init__(self):
        # GrandPa.__init__(self)
        super().__init__()
        print('我是爸爸')

class Uncle(GrandPa):
    def __init__(self):
        # GrandPa.__init__(self)
        super().__init__()
        print('我是叔叔')

class Son(Father,Uncle):
    def __init__(self):
        # Father.__init__(self)
        # Uncle.__init__(self)
        # super() 指代父类实例对象
        super().__init__()
        print('我是儿子')

s=Son()

# for i in Son.mro():
#     print(i)
print(Son.mro())
```

```
# print(object.__dict__)

print(GrandPa.mro())
```

多态

```
1. 同一种事物，具有多种形态
        依托于继承
        父类可以有多个子类的表现形式
        使用父类的声明，可以传入任何子类，依然可以正确运行
2. 没有继承就没有多态
        多态是继承的衍生品
```

- 多态的作用

```
1. 增加程序的灵活性
2. 增加了程序的可扩展性
        本质是继承给予的
        父类扩展了子类
```

```
class Animal:
    def eat(self):
        print('animal can eat')

class Dog(Animal):
    def eat(self):
        print('dog eat bone')
```

```
class Monkey(Animal):
    def eat(self):
        print('monkey eat banana')

class Machine:
    def work(self):
        print('machine can work')

class Robot(Machine,Animal):
    pass

def fun(animal): # 传入父类的多种形态（子类）
    animal.eat()


monkey=Monkey()
fun(monkey)
robot=Robot()
fun(robot)
```

补充：

```
1. issubclass(class,class_or_tuple)
        判断第一个参数是否第二个参数的子类
        第二个参数可以是一个类，也可以是一个元组（元组内的每一个元素，都是类）
2. isinstance(obj, class_or_tuple)
```

```
        判断第一个参数是否第二个参数的实例对象
        第二个参数可以是一个类，也可以是一个元组（元组内的每一个元素，都是类）
3. type(obj)
        返回参数的类型
4. hasattr(obj, name)
        判断第一个对象中是否存在第二个参数指定的属性
5. getattr(object, name[, default])
        object:目标对象
        name：属性名
        default：如果属性不存在时返回的内容
        返回object对象中的name属性
6. setattr(obj, name, value)
        给obj对象设置name属性值为value
7. delattr(obj, name)
        删除obj中的name指定的属性
8. property（fget=None,fset=None,fdel=None,doc=None）
        将属性和属性之间进行关联，同生共死
```

```
class A:
    def __init__(self):
        self.__age=18

    def get_age(self):
        return self.__age
```

```
    def set_age(self,newAge):
        self.__age=newAge

    def del_age(self):
        del self.__age


 x=property(fget=get_age,fset=set_age,fdel=de
l_age,doc='hehe') # property对象

a=A()
# print(a.get_age())
print(a.x)  # 访问---fget()  get_age()
a.x=100  # 赋值 ---fset(100)   set_age(100)
print(a.x)
print(a._A__age)
# del  a.x  # 删除 --- fdel()  del_age()
# print(a.get_age())
a.set_age(200)
print(a.x)
```