6.结构型模式--装饰器模式

什么是装饰器模式?
    我们有的时候总是需要给一个类或者一个对象增加一些行为, 一般情况下使用继承和关联两种方式来实现; 其中使用关联这种方式来实现并符合一定的设计规范的我们称之为装饰器模式.

继承方式:
    一般情况下我们都是通过继承来给一个或者多个类来添加方法, 通过继承使子类获得了父类的行为. 虽然继承是一种适用广泛的方法, 但是继承是一种静态行为, 即在代码编写阶段就已经固定, 无法动态的控制一个类增加行为的种类和时间.

关联方式(装饰器模式):
    我们通过将一个A对象嵌入另一个B对象里面, 即将一个B对象里面的属性的值设置为A对象。通过在调用A对象的动作前后添加行为来给A对象增加功能, 这种模式我们称之为装饰器模式.

    装饰模式可以在不需要创造更多子类的情况下, 将对象的功能加以扩展, 就像搭积木, 可以通过简单积木的组合形成复杂的、功能更为强大的结构这就是使用装饰模式的意义.

生活中的实例:

奶茶店 有饮品 茶叶
       有配料 金桔 柠檬 冰

饮品与配料可以做出的产品有 茶 金桔茶 金桔柠檬茶 金桔柠檬冰茶 柠檬茶 柠檬冰茶 冰茶 7种.

代码示例: 通过继承方式得到每种饮料的名字与价格

In [3]:
```
#coding:utf8

class Tea:
    def __init__(self):
        self.name = "茶"
        self.price = 2

    def getName(self):
        return self.name

    def getPrice(self):
        return self.price


class Ice:
    def add_ice_price(self):
        return 1
    def add_ice_name(self):
        return '冰'

class Orange:
    def add_orange_price(self):
        return 2
    def add_orange_name(self):
        return "金桔"

class Lemon:
    def add_lemon_price(self):
        return 3

    def add_lemon_name(self):
        return "柠檬"


class Ice_tea(Tea, Ice):
    def getName(self):
        return Ice.add_ice_name(self) + Tea.getName(self)

    def getPrice(self):
        return + Ice.add_ice_price(self) + Tea.getPrice(self)


class Orange_tea(Orange, Tea):
    def getName(self):
        return Orange.add_orange_name(self)  + Tea.getName(self)
    def getPrice(self):
        return Orange.add_orange_price(self) + Tea.getPrice(self)


class Orange_ice_tea(Orange, Ice, Tea):
    def getName(self):
        return Orange.add_orange_name(self) + Ice.add_ice_name(self) + Tea.getName(self)

    def getPrice(self):
        return Orange.add_orange_price(self) + Ice.add_ice_price(self) + Tea.getPrice(self)


if __name__ == '__main__':
    x = Ice_tea()
    print("产品: %s, 价格: %s元" %(x.getName(), x.getPrice()))

    y = Orange_tea()
    print("产品: %s, 价格: %s元" % (y.getName(), y.getPrice()))

    z = Orange_ice_tea()
    print("产品: %s, 价格: %s元" % (z.getName(), z.getPrice()))
```

```
产品: 冰茶, 价格: 3元
产品: 金桔茶, 价格: 4元
产品: 金桔冰茶, 价格: 5元
```

    从上面代码中看首先我们定义了饮品类Tea和配料类(Ice Orange Lemon). 当我们要产生一个产品的时候, 通过继承不同的具体类来实现, 比如加冰加金桔加柠檬的茶, 我们通过继承茶, 金桔和柠檬 冰四个类来实现, 可以看出如果要实现所有的类那么我们需要7个子类来完成, 支持多继承的语言才能这样实现如果是单继承的语言则需要通过多级继承来完成, 不仅冗余度增加而且维护这种复杂的多级继承关系的代码将全是眼泪.

下面我们使用装饰器模式, 代码将会精简.

In [2]:

```
#coding:utf8

from abc import ABCMeta, abstractmethod

class Tea:
    def __init__(self):
        self.name = "茶"
        self.price = 2

    def getName(self):
        return self.name

    def getPrice(self):
        return self.price


class Ingredients:
    __metaclass__ = ABCMeta
    def __init__(self, drink):
        self.drink = drink
    @abstractmethod
    def getName(self):
        pass
    @abstractmethod
    def getPrice(self):
        pass



class Ice(Ingredients):
    def add_ice_price(self):
        return 1
    def add_ice_name(self):
        return '冰'

    def getName(self):
        return self.add_ice_name() + self.drink.getName()

    def getPrice(self):
        return self.add_ice_price() + self.drink.getPrice()


class Orange(Ingredients):
    def add_orange_price(self):
        return 2
    def add_orange_name(self):
        return "金桔"

    def getName(self):
        return self.add_orange_name() + self.drink.getName()

    def getPrice(self):
        return self.add_orange_price() + self.drink.getPrice()

class Lemon(Ingredients):
    def add_lemon_price(self):
        return 3

    def add_lemon_name(self):
        return "柠檬"

    def getName(self):
        return self.add_lemon_name() + self.drink.getName()

    def getPrice(self):
        return self.add_lemon_price() + self.drink.getPrice()




if __name__ == '__main__':
    x = Ice(Tea())
    print('产品：%s，价格：%s元' %(x.getName(), x.getPrice()))
    y = Lemon(Ice(Tea()))
    print('产品：%s，价格：%s元' %(y.getName(), y.getPrice()))
    z = Orange(Lemon(Ice(Tea())))
    print('产品：%s，价格：%s元' %(z.getName(), z.getPrice()))
    w = Orange(Lemon(Tea()))
    print('产品：%s，价格：%s元' %(w.getName(), w.getPrice()))
```

```
产品：冰茶，价格：3元
产品：柠檬冰茶，价格：6元
产品：金桔柠檬冰茶，价格：8元
产品：金桔柠檬茶，价格：7元
```

通过装饰模式来扩展对象的功能比继承模式更灵活，构建和装饰器可以独立扩展，新增功能不需要添加大量的子类，但是装饰模式也产生了许多小对象，增加了排错的难度.

以下情况适合使用装饰器模式：
1)在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责.
2)需要动态地给一个对象增加功能，这些功能也可以动态地被撤销.
3)当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时. 不能采用继承的情况有，系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长.