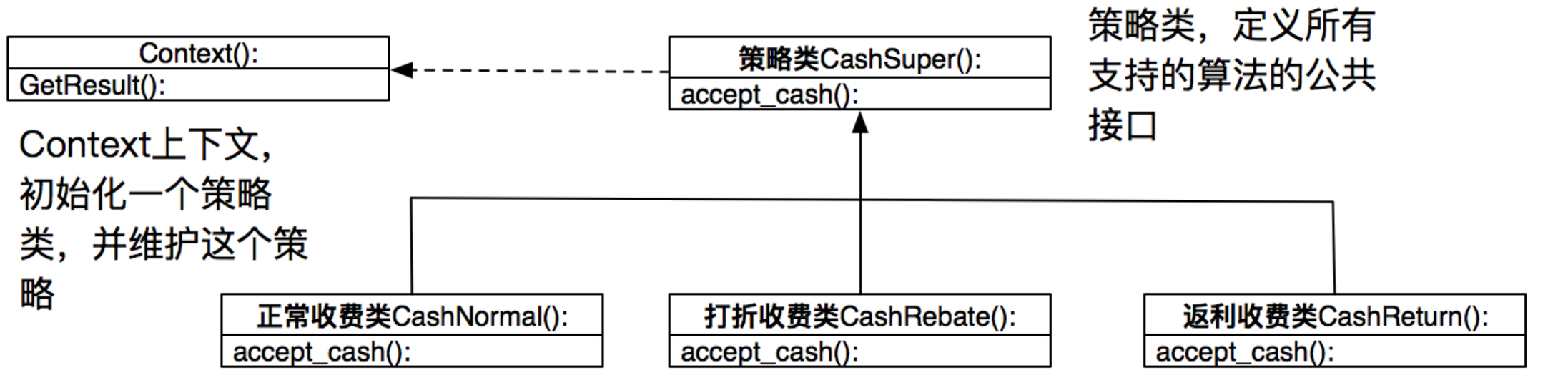
行为型模式-策略模式

概述：
策略模式使用多种算法来解决一个问题，最大的特性的是，能够在运行时透明地切换算法(客户端代码对变化无感知)．使用策略模式在运行时基于输入数据决定使用哪种算法．

使用一个策略类CashSuper定义需要的算法的公共接口，定义三个具体策略类 CashNormal CashRebate CashReturn 继承于CashSuper，定义一个上下文管理类，接收一个策略，并根据该策略得出结论，当需要更改策略时，只需要在实例的时候传入不同的策略就可以，免去了修改类的麻烦．类的设计如下图：



示例代码：

In [1]:
```
from abc import ABCMeta, abstractmethod


class cash_money:
    __metaclass__ = ABCMeta
    @abstractmethod
    def accept_money(self, *args, **kwargs):
        pass


class cash_normal(cash_money):
    def accept_money(self, money):
        return money

class cash_discount(cash_money):
    def __init__(self, discount):
        self.discount = discount

    def accept_money(self, money):
        return money * self.discount

class cash_cheap(cash_money):
    def __init__(self, a, b):
        self.a = a  #满x元
        self.b = b  #减x元

    def accept_money(self, money):
        if money >= self.a:
            s = money - (money / self.a) * self.b
            return s
        return money

class context:
    def __init__(self, cashObj):
        self.cashObj = cashObj

    def getResult(self, money):
        s = "You must Pay %s" %self.cashObj.accept_money(money)
        print(s)

if __name__ == '__main__':
    normal = cash_normal()
    discount = cash_discount(0.8)
    cheap = cash_cheap(100, 20)

    lee = context(normal)
    lee.getResult(100)
    print('#' * 10)
    lee = context(discount)
    lee.getResult(100)
    print('#' * 10)
    lee = context(cheap)
    lee.getResult(200)
```

```
You must Pay 100
##########
You must Pay 80.0
##########
You must Pay 160.0
```

优点：
1.各个策略可以自由切换，这也是依赖抽象类设计接口的好处之一
2.减少代码冗余
3.扩展性优秀 移植方便 使用灵活
缺点：
1.项目比较庞大时，策略可能比较多，不便于维护
2.策略的使用方必须知道有哪些策略，才能决定使用哪一个策略，这与迪米特法则是相违背的．

In [ ]: