Pyhton实现常用的23种设计模式【详解】

一、概念

软件工程中，**设计模式**是指软件设计问题的**推荐方案**。

设计模式一般是描述如何组织代码和使用最佳实践来解决常见的设计问题。

设计模式是高层次的方案，与具体实现细节无关（如算法，数据结构，网页等）。

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

设计模式可以提高代码的可重用性和可读性，增强系统的可靠性和可维护性，解决一系列的复杂问题，提高协作效率。

二、设计模式分类

经典的《设计模式》一书归纳出**23种**设计模式。

这23种模式又可归为，**创建型**、**结构型**和**行为型**3大类。

1.创建型模式

提供**实例化**的方法，为适合的状况提供相应的对象创建方法。

```
社会化的分工越来越细，自然在软件设计方面也是如此，因此对象的创建和对象的使用分开也就成为了必然趋势。
因为对象的创建会消耗掉系统的很多资源，所以单独对对象的创建进行研究，从而能够高效地创建对象就是创建型模式要探讨的问题。
```

这里有5个具体的创建型模式，它们分别是：

```
1、工厂方法模式【Factory Method】
2、抽象工厂模式【Abstract Factory】
3、创建者模式【Builder】
4、原型模式【Prototype】
5、单例模式【Singleton】
```

2、结构型模式

通常用来处理实体之间的关系，使得这些实体能够更好地协同工作。

```
在解决了对象的创建问题之后，对象的组成以及对象之间的依赖关系就成了开发人员关注的焦点，
因为如何设计对象的结构、继承和依赖关系会影响到后续程序的维护性、代码的健壮性、耦合性等。
```

这里有7个具体的结构型模式可供研究，它们分别是：

```
1、外观模式【Facade】
2、适配器模式【Adapter】
3、代理模式【Proxy】
4、装饰模式【Decorator】
5、桥接模式【Bridge】
6、组合模式【Composite】
7、享元模式【Flyweight】
```

3、行为型模式

用于在不同的实体间进行通信，为实体之间的通信提供更容易，更灵活的通信方法。

```
在对象的创建和对象的结构问题都解决了之后，就剩下对象的行为问题了。
如果对象的行为设计的好，那么对象的行为就会更清晰，
它们之间的协作效率就会提高。
```

这里有11个具体的行为型模式，它们分别是：

```
1、模板方法模式【Template Method】
2、观察者模式【Observer】
3、状态模式【State】
4、策略模式【Strategy】
5、职责链模式【Chain of Responsibility】
6、命令模式【Command】
7、访问者模式【Visitor】
8、调停者模式【Mediator】
9、备忘录模式【Memento】
10、迭代器模式【Iterator】
11、解释器模式【Interpreter】
```

三、设计模式六大原则

1.单一原则（Single Responsibility Principle）

一个类只负责一项职责，尽量做到类只有一个行为原因引起变化；

业务对象（BO business object）、业务逻辑（BL business logic）拆分

2.里氏替换原则（LSP liskov substitution principle）

子类可以扩展父类的功能，但不能改变原有父类的功能；

（目的：增强程序的健壮性）实际项目中，每个子类对应不同的业务含义，使父类作为参数，传递不同的子类完成不同的业务逻辑。

3.依赖倒置原则（dependence inversion principle）

面向接口编程；（通过接口作为参数实现应用场景）

依赖于抽象而不依赖于具体。

抽象就是接口或者抽象类，细节就是实现类

依赖倒置原则定义：

- 上层模块不应该依赖下层模块，两者应依赖其抽象；
- 抽象不应该依赖细节，
- 细节应该依赖抽象；

【接口负责定义public属性和方法，并且申明与其他对象依赖关系，抽象类负责公共构造部分的实现，实现类准确的实现业务逻辑】

4.接口隔离（interface segregation principle）

建立单一接口；（扩展为类也是一种接口，一切皆接口）

使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度。

降低依赖，降低耦合。

定义：

- 客户端不应该依赖它不需要的接口；
- 类之间依赖关系应该建立在最小的接口上；

5.迪米特原则（law of demeter LOD）

最少知道原则，尽量降低类与类之间的耦合；

一个对象应该对其他对象有最少的了解

即一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6.开闭原则（open closed principle）

用抽象构建架构，用实现扩展原则；

开闭原则就是说**对扩展开放，对修改关闭。**

一个软件实体通过扩展来实现变化，而不是通过修改原来的代码来实现变化。实现一个热插拔的效果。

开闭原则是对软件实体的未来事件而制定的对现行开发设计进行约束的一个原则。

四、创建型模式实现

提供**实例化**的方法，为适合的状况提供相应的对象创建方法。

1、工厂模式【Factory Method】

意图:

工厂模式包涵一个超类。这个超类提供一个抽象化的接口来创建一个特定类型的对象，而不是决定哪个对象可以被创建。

为了实现此方法，需要创建一个工厂类创建并返回。

当程序运行输入一个“类型”的时候，需要创建于此相应的对象。这就用到了工厂模式。在如此情形中，实现代码基于工厂模式，可以达到可扩展，可维护的代码。

当增加一个新的类型，不再需要修改已存在的类，只增加能够产生新类型的子类。

适用性:

当一个类不知道它所必须创建的对象的类的时候。

当一个类希望由它的子类来指定它所创建的对象的时候。

当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

当需要创建的对象不多且不会频繁增加时【可枚举的对象】

比如:

a.多种数据库(MySQL/MongoDB)的实例

b.多种格式文件的解析器(XML/JSON)

c.根据不同环境加载配置文件【当输入“development”,则加载开发环境的配置文件，而输入“production”，则加载生产环境下的配置文件。】

代码

```
# encoding: utf-8
'''
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 1、工厂模式.py
@time: 2020/3/10 16:46
@author:LDC
'''
"""
工厂模式
工厂模式是一个在软件开发中用来创建对象的设计模式。
工厂模式包涵一个超类。这个超类提供一个抽象化的接口来创建一个特定类型的对象，而不是决定哪个对象可以被创建。
为了实现此方法，需要创建一个工厂类创建并返回。
当程序运行输入一个“类型”的时候，需要创建于此相应的对象。
这就用到了工厂模式。在如此情形中，实现代码基于工厂模式，
可以达到可扩展，可维护的代码。
当增加一个新的类型，不在需要修改已存在的类，只增加能够产生新类型的子类。
举例：
多种品牌的汽车4S店
```

```
当买车时，有很多种品牌可以选择，比如北京现代、别克、凯迪拉克、特斯拉等，
那么此时该怎样进行设计呢？
这时就可以用到工厂类
"""


class CarStore(object):
    # 定义一个4S店基类

    def create_car(self, type_name):
        """:type_name 汽车名称
        """
        # 定义一个创建汽车的方法，但没有实现具体功能，而具体功能需要在子类中实现
        pass

    def order(self, type_name):
        # 让工厂根据类型，生产一辆汽车
        self.car = self.create_car(type_name)
        self.car.move()
        self.car.stop()


# 定义车类型
class YilanteCar(object):
    # 定义伊兰特车类
    def move(self):
        # 定义车的方法，移动
        print("---伊兰特车在移动---")

    def stop(self):
        # 定义车的方法，停车
        print("---伊兰特停车---")


# 定义车类型
class BiekeCar(object):
    # 定义别克车类
    def move(self):
        # 定义车的方法，移动
        print("---别克车在移动---")

    def stop(self):
        # 定义车的方法，停车
        print("---别克停车---")


# 定义一个生产汽车的工厂，让其根据具体的订单生产车
class CarFactory(object):
    # 创建汽车
    def create_car(self, type_name):
        self.type_name = type_name
        if self.type_name == '伊兰特':
            self.car = YilanteCar()
        elif self.type_name == '别克':
            self.car = BiekeCar()
        else:
            self.car = None
        return self.car
```

```
81
82
83  # 定义一个广州现代4S店类
84  class XiandaiCarStore(CarStore):
85
86      def create_car(self, type_name):
87          # 在具体子类中实现父类的创建汽车的方法
88          # 子类调用工厂创建汽车
89          self.car_factory = CarFactory()
90          return self.car_factory.create_car(type_name)
91
92
93  if __name__ == '__main__':
94      yilante = XiandaiCarStore()
95      yilante.order("伊兰特")
96
97  """
98  总结：其实这个方法，就是无限的罗列需要考虑的情况并给出对应的处理。
99  缺点：如果我们要新增一个“产品”，
100         例如宝马BMW的汽车，除了新增一个iBMW类外还要修改CarFactory内的create_car方法。
101         这样就违背了软件设计中的开闭原则[1]，即在扩展新的类时，尽量不要修改原有代码。
102 改进：1、使用多个工厂，增加一个产品，同时增加一个工厂
103        2、使用抽象工厂模式
104 """
105
106
```

2、抽象工厂模式【Abstract Factory】

意图：

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。 **适用性：**

```
1  一个系统要独立于它的产品的创建、组合和表示时。
2  一个系统要由多个产品系列中的一个来配置时。
3  当你要强调一系列相关的产品对象的设计以便进行联合使用时。
4  当你提供一个产品类库，而只想显示它们的接口而不是实现时。
```

每一个模式都是针对一定问题的解决方案。 抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。

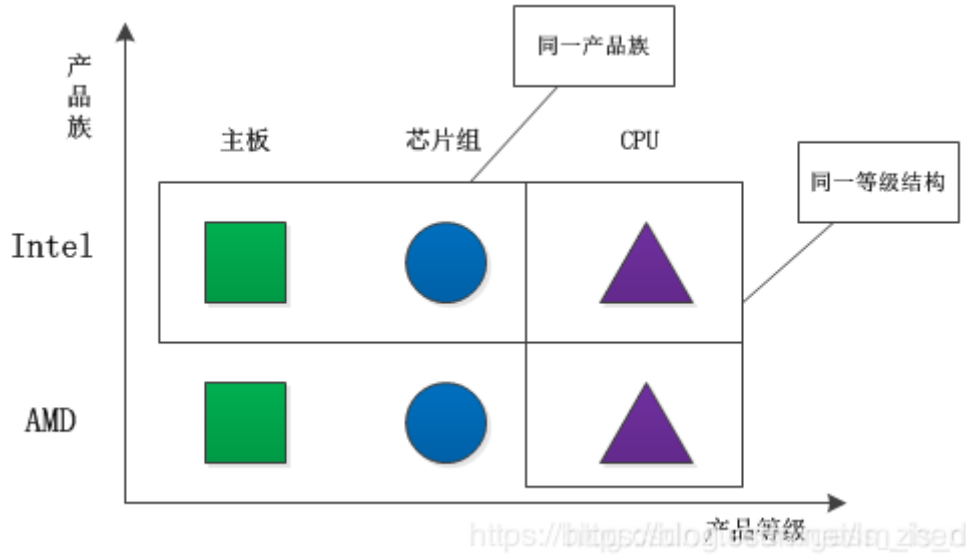
两个重要的概念：**产品族**和**产品等级**。

2.1 产品族和产品等级

所谓产品族，是指位于不同产品等级结构中，功能相关联的产品组成的家族。

比如AMD的主板、芯片组、CPU组成一个家族，Intel的主板、芯片组、CPU组成一个家族。而这两个家族都来自于三个产品等级：主板、芯片组、CPU。

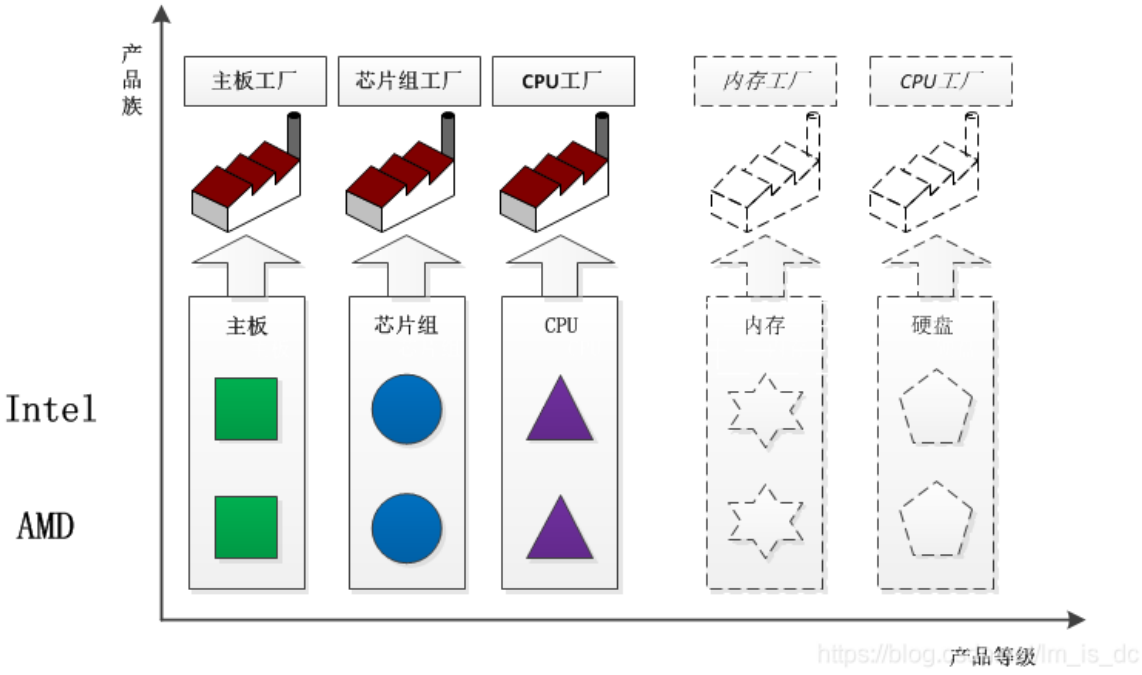
一个等级结构是由相同的结构的产品组成，示意图如下：



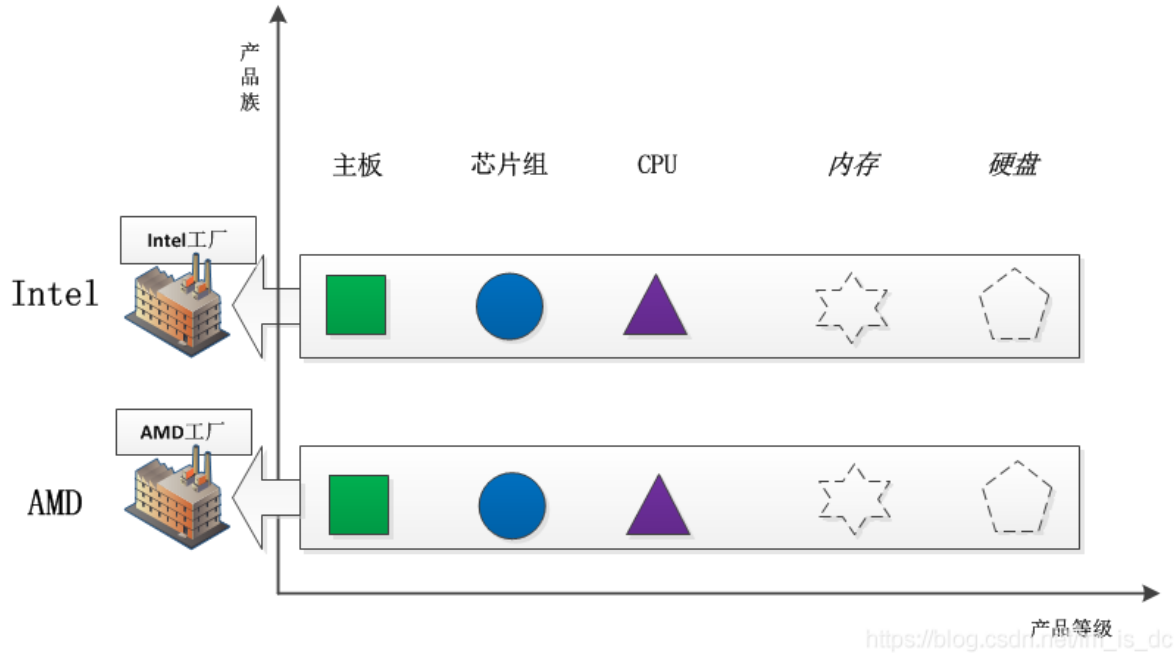
显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。

产品的等级结构与产品族将产品按照不同方向划分，形成一个二维的坐标系。横轴表示产品的等级结构，纵轴表示产品族，上图共有两个产品族，分布于三个不同的产品等级结构中。**只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一的确定这个产品。**

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目的增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。如下图：



那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然可以的，而且这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建。



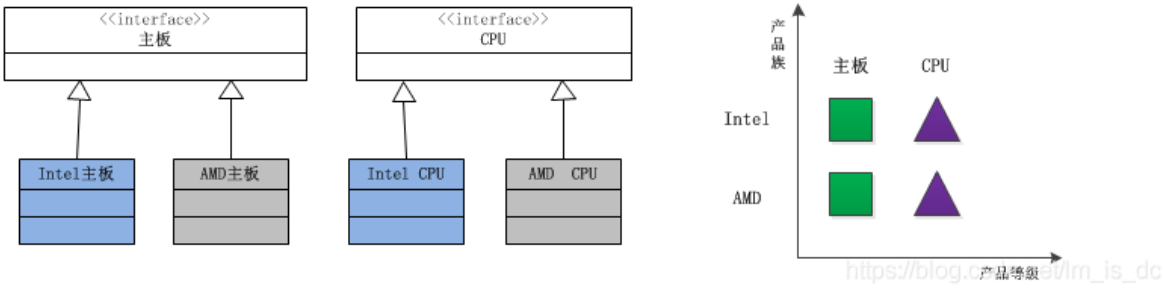

```
可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。
显然，这时候抽象工厂模式比简单工厂模式、工厂方法模式更有效率。
对应于每一个产品族都有一个具体工厂。
而每一个具体工厂负责创建属于同一个产品族，但是分属于不同等级结构的产品。
```

2.2 抽象工厂模式结构

抽象工厂模式是对象的创建模式，它是工厂方法模式的进一步推广。

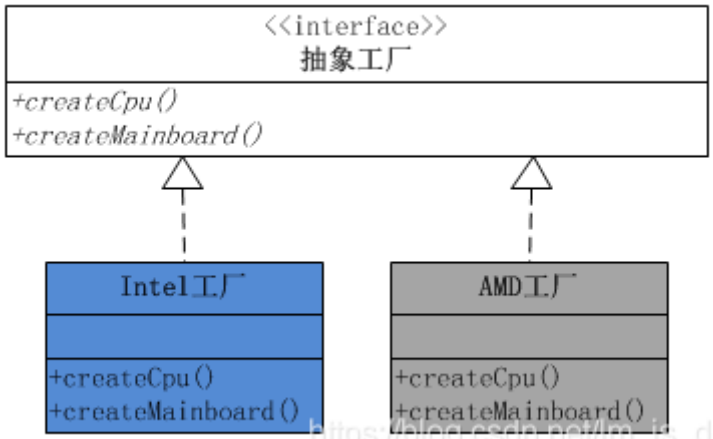
假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品。

通过使用抽象工厂模式，可以处理具有相同（或者相似）等级结构中的多个产品族中的产品对象的创建问题。如下图所示：



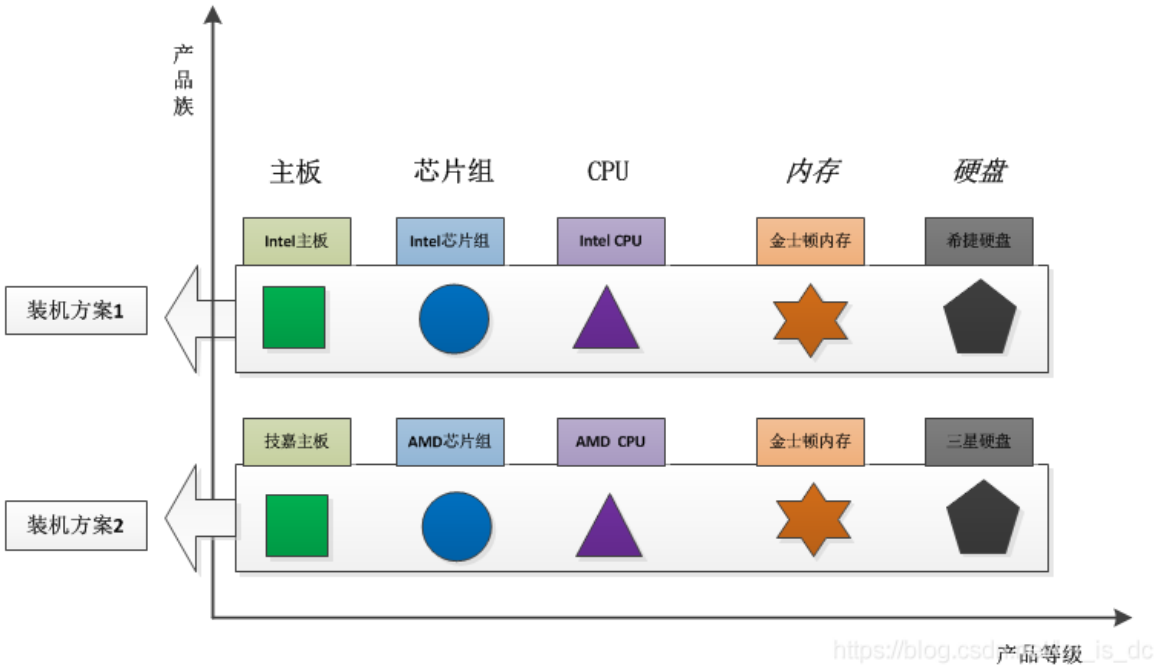
由于这两个产品族的等级结构相同，因此使用同一个工厂族也可以处理这两个产品族的创建问题，这就是抽象工厂模式。

根据产品角色的结构图，就不难给出工厂角色的结构设计图。



可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品等级结构的产品对象。

抽象工厂的功能是为一系列相关对象或相互依赖的对象创建一个接口。一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法。比如上面例子中的主板和CPU，都是为了组装一台电脑的相关对象。不同的装机方案，代表一种具体的电脑系列。



由于抽象工厂定义的一系列对象通常是相关或相互依赖的，这些产品对象就构成了一个产品族，也就是**抽象工厂定义了一个产品族。**

这就带来非常大的灵活性，切换产品族的时候，只要提供不同的抽象工厂实现就可以了，也就是说现在是以一个产品族作为一个整体被切换。

在什么情况下应当使用抽象工厂模式

1.一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。

2.这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

3.同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。（比如：Intel主板必须使用Intel CPU、Intel芯片组）

4.系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

2.3 抽象工厂模式的起源

抽象工厂模式的起源或者最早的应用，是用于创建分属于不同操作系统的视窗构建。比如：命令按键（Button）与文字框（Text)都是视窗构建，在UNIX操作系统的视窗环境和Windows操作系统的视窗环境中，这两个构建有不同的本地实现，它们的细节有所不同。

在每一个操作系统中，都有一个视窗构建组成的构建家族。在这里就是Button和Text组成的产品族。而每一个视窗构件都构成自己的等级结构，由一个抽象角色给出抽象的功能描述，而由具体子类给出不同操作系统下的具体实现。

2.4 抽象工厂模式的优点

分离接口和实现　　客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已。也就是说，客户端从具体的产品实现中解耦。

使切换产品族变得容易　　因为一个具体的工厂实现代表的是一个产品族，比如上面例子的从Intel系列到AMD系列只需要切换一下具体工厂。

```
总的来说，抽象工厂模式隔离了具体类的生成，使得客户端并不需要知道什么被创建 。
当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象 。
增加新的产品族很方便，无须修改已有系统，符合开闭原则
```

2.5 抽象工厂模式的缺点

不太容易扩展新的产品　　如果需要给整个产品族添加一个新的产品，那么就需要修改抽象工厂，这样就会导致修改所有的工厂实现类。

2.6 使用环境

一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节 系统中有多于一个的产品族，但每次只使用其中某一产品族 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来 产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 2、抽象工厂模式.py
@time: 2020/3/12 17:26
@author:LDC
'''
"""
抽象工厂模式
提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；
而抽象工厂模式则需要面对多个产品等级结构。
"""


class AbstractFactory(object):
    # 创建一个抽象工厂
    computer_name = ''  # 电脑名称

```

```python
    def create_cpu(self):
        # 定义一个创建cpu方法具体实现由子类完成
        pass

    def create_mainboard(self):
        # 定义一个创建主板方法具体实现由子类完成
        pass


class AbstractCpu(object):
    # 定义一个cpu产品抽象类
    series_name = ''
    instructions = ''
    arch = ''


class IntelCpu(AbstractCpu):
    # 定义一个Intel公司的cpu产品类，继承【cpu产品抽象类】
    def __init__(self, series):
        self.series_name = series  # 序列号名称


class AmdCpu(AbstractCpu):
    # 定义一个Amd公司的cpu产品类，继承【cpu产品抽象类】
    def __init__(self, series):
        self.series_name = series  # 序列号名称


class AbstractMainboard(object):
    # 定义一个mainboard(主板)产品抽象类
    series_name = ''


class IntelMainboard(AbstractMainboard):
    # 定义一个Intel公司的mainboard(主板)产品类，继承【mainboard(主板)抽象类】
    def __init__(self, series):
        self.series_name = series  # 序列号名称


class AmdMainboard(AbstractMainboard):
    # 定义一个Amd公司的mainboard(主板)产品类，继承【mainboard(主板)抽象类】
    def __init__(self, series):
        self.series_name = series  # 序列号名称


class IntelFactory(AbstractFactory):
    # 创建一个生产Intel公司产品的工厂，继承抽象工厂类
    computer_name = 'Intel I7-series computer '

    def create_cpu(self):
        # 在工厂里定义一个创建cpu产品方法
        return IntelCpu('I7-6500')

    def create_mainboard(self):
        # 在工厂里定义一个创建mainboard(主板)产品方法
        return IntelCpu('Intel-6000')


```

```python
class AmdFactory(AbstractFactory):
    # 创建一个生产Amd公司产品的工厂，继承抽象工厂类
    computer_name = 'Amd 4 computer '

    def create_cpu(self):
        # 在工厂里定义一个创建cpu产品方法
        return AmdCpu('amd444')

    def create_mainboard(self):
        # 在工厂里定义一个创建mainboard(主板)产品方法
        return AmdMainboard('AMD-4000')


class ComputerEnginee(object):
    # 定义一个装机工程师
    def make_computer(self, factory_obj):
        self.prepare_hardwares(factory_obj)

    def prepare_hardwares(self, factory_obj):
        # 定义一个硬件装机方法
        self.cpu = factory_obj.create_cpu()
        self.mainboard = factory_obj.create_mainboard()
        info = ''' -----------电脑【{}】信息-----------
            cpu: 【{}】
            mainboaed: 【{}】
        '''.format(factory_obj.computer_name, self.cpu.series_name, self.mainboard.series_name)
        print(info)


if __name__ == '__main__':
    engineer = ComputerEnginee()  # 装机工程师
    intel_factory = IntelFactory()  # Intel工厂
    engineer.make_computer(intel_factory)  # 工程师装Intel的电脑

    amd_factory = AmdFactory()  # Intel工厂
    engineer.make_computer(amd_factory)  # 工程师装Amd的电脑

"""
总结:
抽象工厂和工厂模式的对比区别:
抽象工厂：规定死了，依赖限制，像上面实验，你用intel的机器只能配置intel的CPU不能配置AMD的CPU（由各自的工厂指定自己的产品生产品牌）
工厂模式：不是固定死的，举例：你可使用intel的机器配置AMD的CPU
抽象工厂模式在工厂方法基础上扩展了工厂对多个产品创建的支持，
更适合一些大型系统，
比如系统中有多于一个的产品族，且这些产品族类的产品需实现同样的接口，
像很多软件系统界面中不同主题下不同的按钮、文本框、字体等等。
"""

```

输出：

```
----------电脑【Intel I7-series computer 】信息----------
            cpu: 【I7-6500】
            mainboaed: 【Intel-6000】

 ----------电脑【Amd 4 computer 】信息----------
            cpu: 【amd444】
            mainboaed: 【AMD-4000】
```

3、创建者模式【Builder】

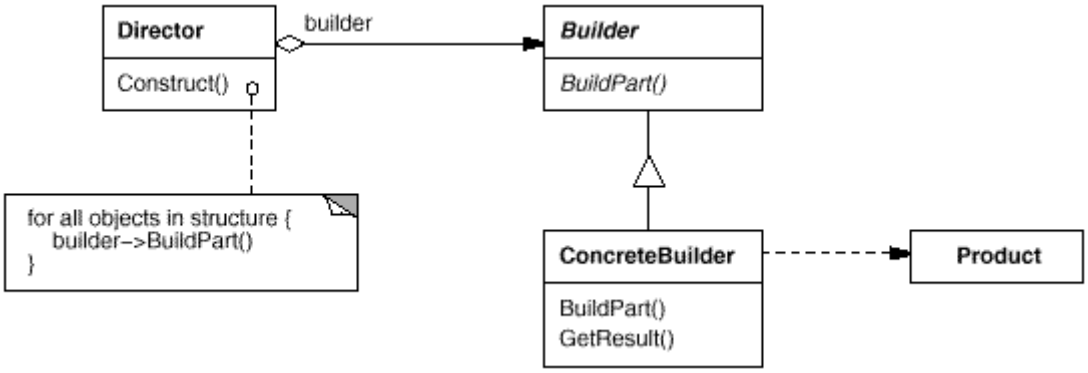
意图:

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

适用性:

当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

当构造过程必须允许被构造的对象有不同的表示时。



代码:

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 3、创建者模式.py
@time: 2020/3/12 20:24
@author:LDC
'''
"""
创建者模式
相关模式：思路和模板方法模式很像，模板方法是封装算法流程，对某些细节，提供接口由子类修改，
创建者模式更为高层一点，将所有细节都交由子类实现。
创建者模式：将一个复杂对象的构建与他的表示分离，使得同样的构建过程可以创建不同的表示。
基本思想
某类产品的构建由很多复杂组件组成；
这些组件中的某些细节不同，构建出的产品表象会略有不同；
通过一个指挥者按照产品的创建步骤来一步步执行产品的创建；
当需要创建不同的产品时，只需要派生一个具体的创建者，重写相应的组件构建方法即可。
"""


# 定义一个创建者基类
```

```python
class PersonBuilder(object):

    def build_head(self):
        pass

    def build_body(self):
        pass

    def build_arm(self):
        pass

    def build_leg(self):
        pass


# 胖子
class PersonFatBuilder(PersonBuilder):
    type = '胖子'

    def build_head(self):
        info = "构建{}的大...头".format(self.type)
        print(info)

    def build_body(self):
        info = "构建{}的身体".format(self.type)
        print(info)

    def build_arm(self):
        info = "构建{}的手".format(self.type)
        print(info)

    def build_leg(self):
        info = "构建{}的脚".format(self.type)
        print(info)


# 瘦子
class PersonThinBuilder(PersonBuilder):
    type = '瘦子'

    def build_head(self):
        info = "构建{}的头".format(self.type)
        print(info)

    def build_body(self):
        # 注意与别的产品细节不同
        info = "构建{}的瘦小身体".format(self.type)
        print(info)

    def build_arm(self):
        info = "构建{}的手".format(self.type)
        print(info)

    def build_leg(self):
        info = "构建{}的脚".format(self.type)
        print(info)


```

```python
# 指挥者
class PersonDirector(object):
    pd = None

    def __init__(self, pd):
        self.pd = pd

    def create_person(self):
        # 指挥者按照产品的创建步骤来一步步执行产品的创建
        self.pd.build_head()
        self.pd.build_body()
        self.pd.build_arm()
        self.pd.build_leg()


class ClientUI(object):
    pb_fat = PersonFatBuilder()  # 创建胖子组件
    pd = PersonDirector(pb_fat)  # 创建指挥者
    pd.create_person()  # 指挥者组建胖子这一个产品

    pb_thin = PersonThinBuilder()  # 创建瘦子组件
    pd.pd = pb_thin  # 指挥者要创建的产品改为瘦子
    pd.create_person()  # 创建产品


if __name__ == '__main__':
    ClientUI()

```

输出：

```
构建胖子的大...头
构建胖子的身体
构建胖子的手
构建胖子的脚
构建瘦子的头
构建瘦子的瘦小身体
构建瘦子的手
构建瘦子的脚

```

4、原型模式【Prototype】

意图：

用原型实例指定创建对象的种类，并且通过**拷贝这些原型创建新的对象。**

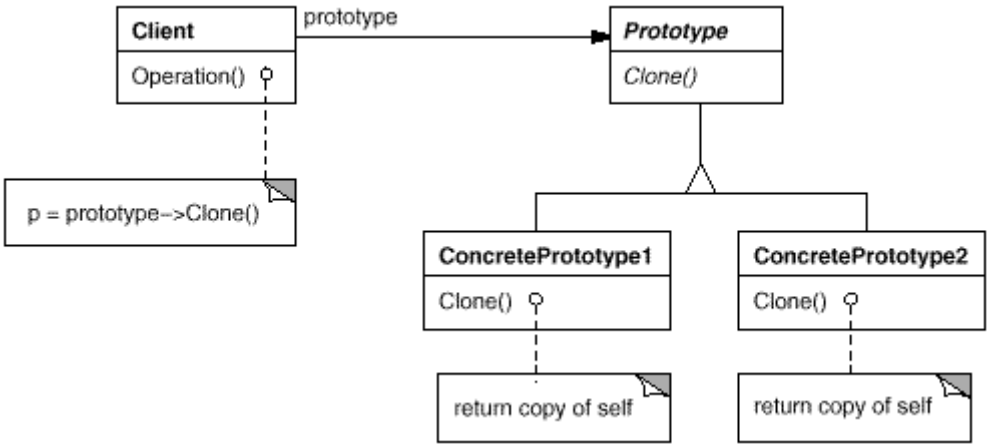
当我们已有一个对象，并希望创建该对象的一个完整副本时，原型模式就派上用场了。在我们知道对象的某些部分会被变更但又希望保持原有对象不变之时，通常需要对象的一个副本。在这样的案例中，重新创建原有对象是没有意义的。 另一个案例是，当我们想复制一个复杂对象时，使用原型模式会很方便。对于复制复杂对象，我们可以将对象当作是从数据库中获取的，并引用其他一些也是从数据库中获取的对象。若通过多次重复查询数据来创建一个对象，则要做很多工作。在这种场景下使用原型模式要方便得多。

适用性：

当要实例化的类是在运行时刻指定时，

比如： 通过动态装载； 或者为了避免创建一个与产品类层次平行的工厂类层次时； 或者当一个类的实例只能有几个不同状态组合中的一种时。

建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。



代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 4、原型模式.py
@time: 2020/3/12 21:24
@author:LDC
'''
"""
原型模式
用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。
当我们想复制一个复杂对象时，使用原型模式会很方便。
对于复制复杂对象，我们可以将对象当作是从数据库中获取的，
并引用其他一些也是从数据库中获取的对象。
若通过多次重复查询数据来创建一个对象，则要做很多工作。
在这种场景下使用原型模式要方便得多。
"""
import copy
from collections import OrderedDict


class Book(object):
    # 创建一个书籍对象
    def __init__(self, name, authors, price, **rest):
        '''rest的例子有：出版商、长度、标签、出版日期'''
        self.name = name
        self.authors = authors
        self.price = price  # 单位为美元
        self.__dict__.update(rest)  # self.__dict__是包含
name,authors,price,**rest的元素的字典

    def __str__(self):
        mylist = []
        # 对对象中的内置字典进行升序排序，然后再固定字典元素位置
        ordered = OrderedDict(sorted(self.__dict__.items()))
```

```
        for i in ordered.keys():
            mylist.append('{}: {}'.format(i, ordered[i]))
            if i == 'price':
                mylist.append('$')
            mylist.append('\n')
        return ''.join(mylist)


class ProtoType(object):
    # 创建一个原型
    def __init__(self):
        self._object = {}

    def register_object(self, name, obj):
        """注册一个对象"""
        self._object[name] = obj

    def unregister_object(self, name, obj):
        """删除一个对象"""
        del self._object[name]

    def clone(self, identifier, **attr):
        """根据 identifier 在原型列表中查找原型对象并克隆"""
        obj = copy.deepcopy(self._object.get(identifier))
        if not obj:
            raise ValueError('Incorrect object identifier:
{}'.format(identifier))
        obj.__dict__.update(attr)  # 用新的属性值替换原型对象中的对应属性
        return obj


def main():
    b1 = Book('The C Programming Language', ('Brian W. Kernighan', 'Dennis
M.Ritchie'),
              price=118, publisher='Prentice Hall', length=228,
publication_date='1978-02-22',
              tags=('C', 'programming', 'algorithms', 'data structures'))
    prototype = ProtoType()  # 实例化原型
    cid = 'k&r-first'
    prototype.register_object(cid, b1)  # 注册一个书籍原型
    # 对书籍原型进行克隆
    b2 = prototype.clone(cid, name='The C Programming Language(ANSI)',
price=48.99,
                         length=274, publication_date='1988-04-01',
edition=2)
    for i in (b1, b2):
        print(i)
    print("ID b1 : {} != ID b2 : {}".format(id(b1), id(b2)))


if __name__ == '__main__':
    main()

"""
总结:
用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。
原型模式本质就是克隆对象，
所以在对象初始化操作比较复杂的情况下，很实用，能大大降低耗时，提高性能，
```

```
因为“不用重新初始化对象，而是动态地获得对象运行时的状态”。

浅拷贝（Shallow Copy）:指对象的字段被拷贝，而字段引用的对象不会被拷贝，
拷贝的对象和源对象只是名称相同，但是他们共用一个实体。
深拷贝（deep copy）:对对象实例中字段引用的对象也进行拷贝。

比如：当我们出版了一本书《Python 设计模式 1.0版》，若10 年后我们觉得这本书跟不上时代了，
这时候需要去重写一本《Python 设计模式 2.0版》，
那么我们是完全重写一本书呢？还是在原有《Python 设计模式 1.0版》的基础上进行修改呢？
当然是后者，这样会省去很多排版、添加原有知识等已经做过的工作。
"""

```

输出：

```
authors: ('Brian W. Kernighan', 'Dennis M.Ritchie')
length: 228
name: The C Programming Language
price: 118$
publication_date: 1978-02-22
publisher: Prentice Hall
tags: ('C', 'programming', 'algorithms', 'data structures')

authors: ('Brian W. Kernighan', 'Dennis M.Ritchie')
edition: 2
length: 274
name: The C Programming Language(ANSI)
price: 48.99$
publication_date: 1988-04-01
publisher: Prentice Hall
tags: ('C', 'programming', 'algorithms', 'data structures')

ID b1 : 45655344 != ID b2 : 47259216

```

5、单例模式【Singleton】

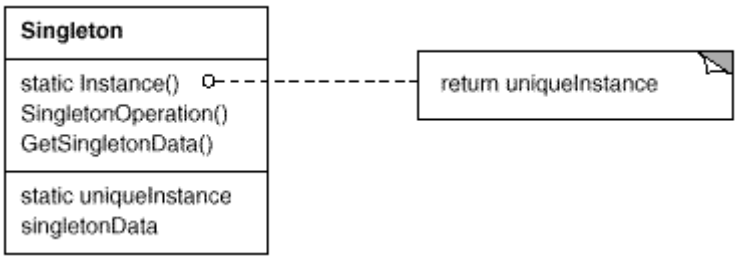
意图：

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

适用性：

当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。



比如： 某个服务器程序的配置信息存放在一个文件中，客户端通过一个AppConfig的类来读取配置文件的信息。

如果在程序运行期间，有很多地方都需要使用配置文件的内容，也就是说，很多地方都需要创建AppConfig对象的实例。

这就导致系统中存在多个APPConfig的实例对象，而这样会严重浪费内存资源，尤其是在配置文件内容很多的情况下。

事实上，类似APPConfig这样的类，我们希望在程序运行期间只存在一个实例对象。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 5、单例模式.py
@time: 2020/3/13 21:45
@author:LDC
'''
"""
单例模式
实现__new__方法
并在将一个类的实例绑定到类变量_instance上，
如果cls._instance为None说明该类还没有实例化过，实例化该类，并返回
如果cls._instance不为None，直接返回cls._instance
"""


class Singleton(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_instance'):
            # 判断是否有该实例存在，前面是否已经有人实例过，如果内存没有该实例...往下执行
            # 需要注明该父类的内存空间内最多允许相同名字子类的实例对象存在1个（不可多个）
            orig = super(Singleton, cls)  # farther class
            cls._instance = orig.__new__(cls)
        return cls._instance


class MyClass(Singleton):
    def __init__(self, name):
        self.name = name


class ldc(Singleton):
    def __init__(self, name):
        self.name = name
```

```
37
38
39  if __name__ == '__main__':
40      # 实例化一个类MyClass
41      a = MyClass("first class")
42      print(a.name)
43      # 对类MyClass进行第二次实例化
44      b = MyClass("second class")
45      print(a.name, b.name)
46      # 实例化一个类ldc
47      c = ldc('third')
48      print(a.name, b.name, c.name)
49      print(id(a), id(b), id(c))
50
51
52  """
53  总结：
54  通过执行结果我们可以看出：一个类永远只允许一个实例化对象，不管多少个进行实例化，都返回第一
    个实例化的对象
55  """
```

输出：

```
1  first class
2  second class second class
3  second class second class third
4  54008208 54008208 54008304
```

五、结构型模式实现

其主要**用来处理一个系统中不同实体（比如类和对象）之间关系**，关注的是提供一种简单的对象组合方式来创造新的功能。

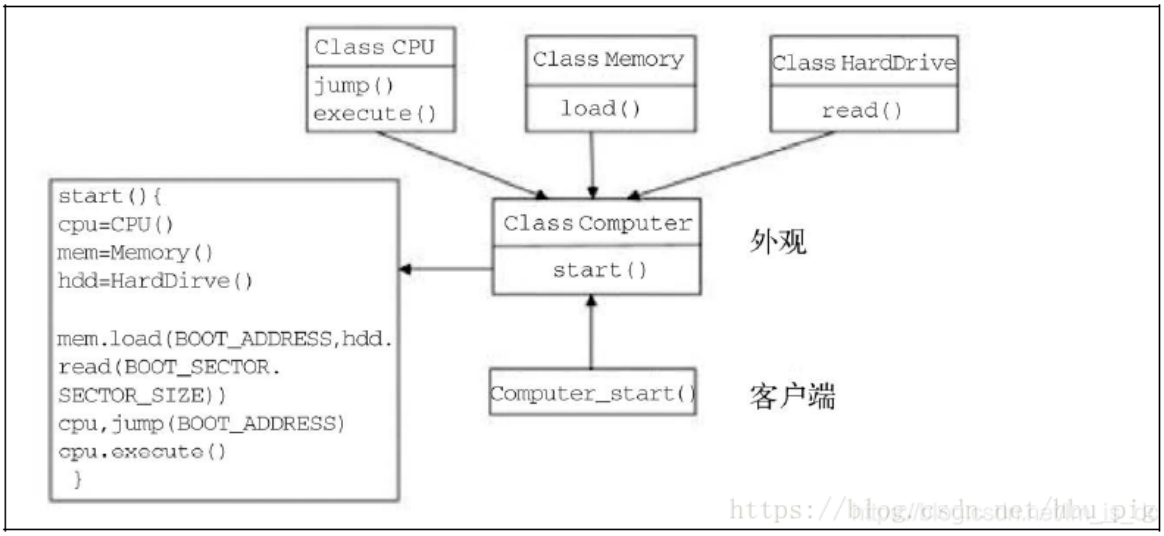
1、外观模式【Facade】

意图：

为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

本质上，外观（Facade）是在已有的复杂系统之上实现的一个抽象层。

下图演示了外观的角色。从图中展示的类可知，仅Computer类需要暴露给客户端代码。客户端仅执行Computer的start()方法。所有其他复杂部件都由外观类Computer来维护。



适用性：

使用外观模式的最常见理由是为一个复杂系统提供单个简单的入口点。引入外观之后，客户端代码通过简单地调用一个方法/函数就能使用一个系统。同时内部系统并不会丢失任何功能，外观只是封装了内部系统。

比如：

当你致电一个银行或公司，通常是先被连线到客服部门，客服职员在你和业务部门及帮你解决具体问题的职员之间充当一个外观的角色。

也可以将汽车或摩托车的启动钥匙视为一个外观。外观是激活一个系统的便捷方式，系统的内部则非常复杂。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 6、外观模式.py
@time: 2020/3/13 22:02
@author:LDC
'''
"""
外观模式
假设有一组火警报警系统，由三个子元件构成：一个警报器，一个喷水器，一个自动拨打电话的装置。
#当火警发生时，先警报器响起警报，喷水器开始喷水，最后开始拨打火警电话
"""


class AlarmSensor(object):
    # 警报器
    def run(self):
        print('Alarm Ring...')


class WaterSprinkler(object):
    # 喷水器
    def run(self):
```

```
        print("Spray Water...")


class EmergencyDialer(object):
    # 拨打火警电话
    def run(self):
        print("Dial 119...")


class EmergencyFacade(object):
    # 定义一个外观类,其中封装对子系统的操作
    def __init__(self):
        self.alarm_sensor = AlarmSensor()
        self.water_sprinkler = WaterSprinkler()
        self.emergency_dialer = EmergencyDialer()

    def run_all(self):
        self.alarm_sensor.run()
        self.water_sprinkler.run()
        self.emergency_dialer.run()


if __name__ == '__main__':
    emergency_facade = EmergencyFacade()
    emergency_facade.run_all()


"""
总结：
根据“单一职责原则”，在软件中将一个系统划分为若干个子系统有利于降低整个系统的复杂性，
一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小，而达到该目标的途径之一就是引入一个外观对象，
它为子系统的访问提供了一个简单而单一的入口。
外观模式也是“迪米特法则”的体现，通过引入一个新的外观类可以降低原有系统的复杂度，
同时降低客户类与子系统类的耦合度。

外观模式要求一个子系统的外部与其内部的通信通过一个统一的外观对象进行，
外观类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与外观对象打交道，
而不需要与子系统内部的很多对象打交道。
外观模式的目的在于降低系统的复杂程度。
外观模式从很大程度上提高了客户端使用的便捷性，使得客户端无须关心子系统的工作细节，通过外观角色即可调用相关功能。

优点：
主要优点在于对客户屏蔽子系统组件，减少了客户处理的对象数目并使得子系统使用起来更加容易，
它实现了子系统与客户之间的松耦合关系，并降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程；

缺点：
其缺点在于不能很好地限制客户使用子系统类，而且在不引入抽象外观类的情况下，
增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

使用情况：
适用情况包括：
要为一个复杂子系统提供一个简单接口；
客户程序与多个子系统之间存在很大的依赖性；
在层次化结构中，需要定义系统中每一层的入口，使得层与层之间不直接产生联系。

```

```
"""
```

输出：

```
Alarm Ring...
Spray Water...
Dial 119...
```

2、适配器模式【Adapter】

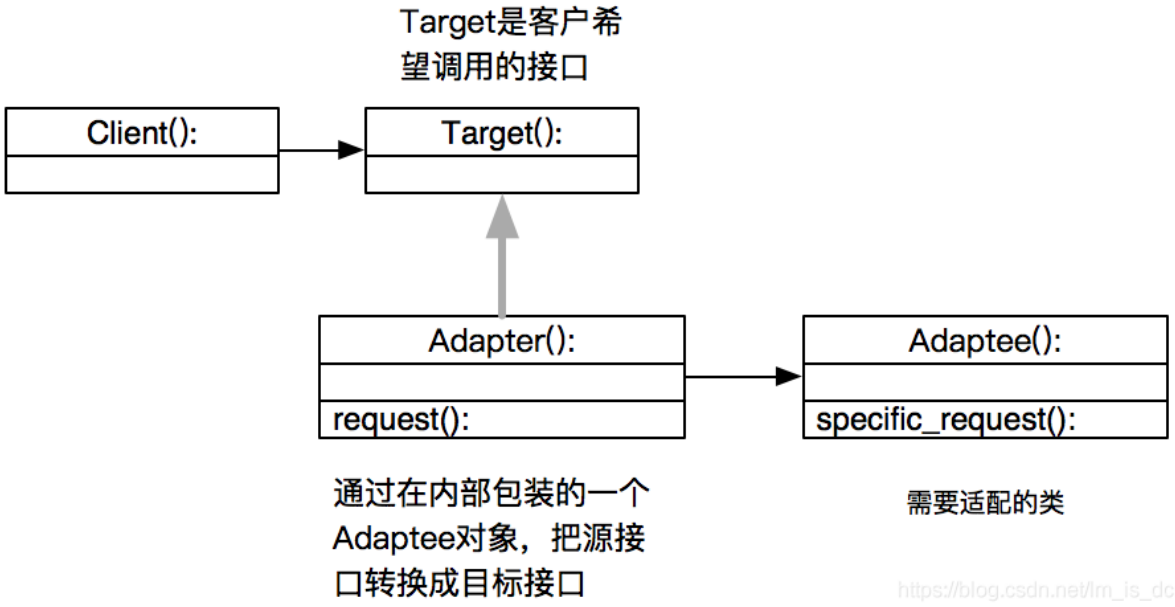
意图：

所谓适配器模式是指一种接口适配技术，它可通过某个类来使用另一个接口与之不兼容的类，运用此模式，两个类的接口都无需改动。

适配器模式主要应用于希望复用一些现存的类，但是接口又与复用环境要求不一致的情况，比如在需要对早期代码复用一些功能等应用上很有实际价值。

适用性：

适配器模式(Adapter Pattern):将一个类的接口转换成为客户希望的另外一个接口.Adapter Pattern使得原本由于接口不兼容而不能一起工作的那些类可以一起工作. 应用场景:系统数据和行为都正确,但接口不符合时,目的是使控制范围之外的一个原有对象与某个接口匹配,适配器模式主要应用于希望复用一些现存的类,但接口又与复用环境不一致的情况



比如： 如果你有一部智能手机或者一台平板电脑，在想把它（比如，iPhone手机的闪电接口）连接到你的电脑时，就需要使用一个USB适配器。如果你从大多数欧洲国家到英国旅行，在为你的笔记本电脑充电时，就需要使用一个插头适配器。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 7、适配器模式.py
@time: 2020/3/13 23:42
```

```
@author:LDC
'''
"""
适配器一：
NBA球星中分为前锋，中锋，后卫，它们交流都是英语。
假如中国球星姚明刚开始加入NBA，他肯定不能和队员，教练正确交流，因为他不会英文。
这个时候，我们就需要给姚明请一个翻译人员，既能和姚明交流，又能和教练交流，
翻译人员在这个过程中起的就是一个适配器的作用。
"""
import abc


class Player(metaclass=abc.ABCMeta):
    # 定义一个抽象类
    def __init__(self, name):
        self.name = name

    @abc.abstractmethod
    def attack(self):
        pass

    @abc.abstractmethod
    def defense(self):
        pass


class Forwards(Player):
    # 定义前锋类
    def __init__(self, name):
        Player.__init__(self, name)  # 父类初始化

    def attack(self):
        print("前锋{}进攻".format(self.name))

    def defense(self):
        print("前锋{}防守".format(self.name))


class Center(Player):
    # 定义中锋类
    def __init__(self, name):
        Player.__init__(self, name)  # 父类初始化

    def attack(self):
        print("中锋{}进攻".format(self.name))

    def defense(self):
        print("中锋{}防守".format(self.name))


class Guards(Player):
    # 定义后卫类
    def __init__(self, name):
        Player.__init__(self, name)  # 父类初始化

    def attack(self):
        print("后卫{}进攻".format(self.name))

```

```
    def defense(self):
        print("后卫{}防守".format(self.name))


# 当前中锋是姚明，他不认识Attack和Denfense所以需要一个翻译作为适配器，为了适配姚明和英语

class ForeignCenter(object):
    # 定义一个外籍中锋类
    def __init__(self, name):
        self.name = name

    def 攻击(self):
        print("外籍中锋{}在进攻".format(self.name))

    def 防守(self):
        print("外籍中锋{}在防守".format(self.name))


class Transtator(Player):
    # 定义一个翻译人员的类，作为适配器
    def __init__(self, name):
        Player.__init__(self, name)
        self.wjzf = ForeignCenter(name)  # 实例化外籍中锋

    def attack(self):
        self.wjzf.攻击()

    def defense(self):
        self.wjzf.防守()


"""
适配器二：
使用类中的内部字典做适配器
有三个类（一个叫做Computer，另外两个叫做Synthesizer，Human），
我们现在想做的就是将Computer类和Synthesizer，Human做适配。
假设这三个类都不能改。
用户只知道Computer中的execute()方法，
怎样调用Synthesizer 的play()方法和Human中的speak()方法?
此时我们就需要考虑做个适配器Adapter。

"""


class Computer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} computer'.format(self.name)

    def execute(self):
        return 'executes a program'


class Synthesizer:
    def __init__(self, name):
```

```
        self.name = name

    def __str__(self):
        return 'the {} synthesizer'.format(self.name)

    def play(self):
        return 'is playing an electronic song'


class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return '{} the human'.format(self.name)

    def speak(self):
        return 'says hello'


class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)  # adapterd_method是一个字典，键是客户调用的方法，值是被调用的方法。

    def __str__(self):
        return str(self.obj)


if __name__ == '__main__':
    print('-------------使用抽象类实现适配器---------------')
    # 前锋实例化
    f_Batir = Forwards("巴蒂尔")
    f_Batir.attack()
    f_Batir.defense()

    # 后卫实例化
    g_Maddie = Guards("麦迪")
    g_Maddie.attack()
    g_Maddie.defense()

    # 中锋实例化
    ym = Transtator("姚明")
    ym.attack()
    ym.defense()

    print("-------------使用类中的内置字典实现适配器---------------")
    objects = [Computer('Asus')]
    synth = Synthesizer('moog')
    objects.append(Adapter(synth, dict(execute=synth.play)))
    human = Human('Bob')
    objects.append(Adapter(human, dict(execute=human.speak)))
    for i in objects:
        print('{} {}'.format(str(i), i.execute()))
```

```
"""
总结:
什么时候使用Adapter模式
很多时候，我们并非从0开始编程，特别是当现有的类已经被充分测试过了，Bug很少，
而且已经被用于其他软件之中时，我们更愿意将这些类作为组件重复利用。

Adapter模式会对现有的类进行适配，生成新的类。通过该模式可以很方便地创建我们需要的方法群。
当出现Bug时，由于我们明确知道Bug不在现有的类（Adaptee角色）中，所以只需调查Adapter角色的类即可。

如果没有现成的代码让现有的类适配新的接口（API）时，使用Adapter模式似乎是理所应当的。
在Adapter模式中，并非一定需要现成的代码。只要知道现有类的功能，就可以。

版本升级与兼容性
软件的生命周期总是伴随着版本的升级，而很多时候需要与旧版本兼容。
这个时候可以让新版本扮演Adaptee角色，旧版本扮演Target角色。
接着编写一个Adapter角色的类，让它使用新版本的类来实现旧版本的类中的功能。

功能完全不同的类
当然，当Adaptee角色与Target角色的功能完全不同时，Adapter模式是无法使用的。
就如同我们无法用交流100伏特电压让自来水管出水一样。

"""
```

输出:

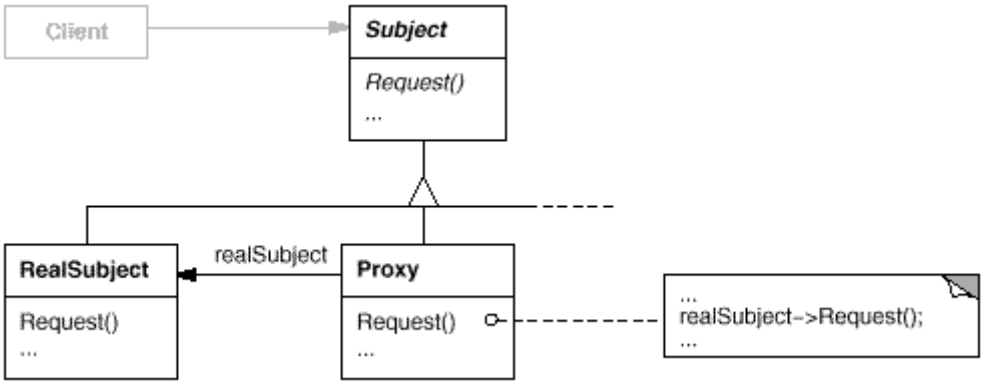
```
-------------使用抽象类实现适配器---------------
前锋巴蒂尔进攻
前锋巴蒂尔防守
后卫麦迪进攻
后卫麦迪防守
外籍中锋姚明在进攻
外籍中锋姚明在防守
-------------使用类中的内置字典实现适配器---------------
the Asus computer executes a program
the moog synthesizer is playing an electronic song
Bob the human says hello

```

3、代理模式【Proxy】

意图:

为其他对象提供一种代理以控制对这个对象的访问。

```
主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。
在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），
直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。
```



适用性：

想在访问一个类时做一些控制。

即增加中间层实现与被代理类组合。

比如：

1、Windows 里面的快捷方式。 2、买火车票不一定在火车站买，也可以去代售点。 3、一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。

优点： 1、职责清晰。 2、高扩展性。 3、智能化。

缺点： 1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。 2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

使用场景：按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access）代理。 5、Cache代理。 6、防火墙（Firewall）代理。 7、同步化（Synchronization）代理。 8、智能引用（Smart Reference）代理。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 8、代理模式.py
@time: 2020/3/14 13:13
@author:LDC
'''

"""
代理模式
应用特性：在通信双方中间需要一些特殊的中间操作时引用，多加一个中间控制层。
结构特性：建立一个中间类，创建一个对象，接收一个对象，然后把两者联通起来
"""


class SenderBase(object):
    # 定义一个发信息的基类
    def send_something(self, something):
        pass


class SendClass(SenderBase):
    # 定义一个发信息的类
```

```
    def __init__(self, receiver):
        self.receiver = receiver

    def send_something(self, something):
        print('send {} to {}'.format(something, self.receiver))


class Proxy(SenderBase):
    # 定义一个代理类
    def __init__(self, receiver):
        self.send_obj = SendClass(receiver)

    def send_something(self, something):
        self.send_obj.send_something(something)


class ReceiveClass(object):
    # 定义一个接收类
    def __init__(self, someone):
        self.name = someone

    def __str__(self):
        return self.name


if __name__ == '__main__':
    receiver = ReceiveClass('ldc')
    proxy = Proxy(receiver)
    proxy.send_something('成功使用了代理')
    print(receiver.__class__)
    print(proxy.__class__)


"""
总结：
代理模式为其他对象提供一种代理以控制对这个对象的访问。
代理模式就如同一个"过滤器"，它不实现具体功能，具体功能由被调用的实体来实现，
代理实现的是对调用的控制功能，它能够允许或者拒绝调用实体对被调用实体的访问。
"""
```

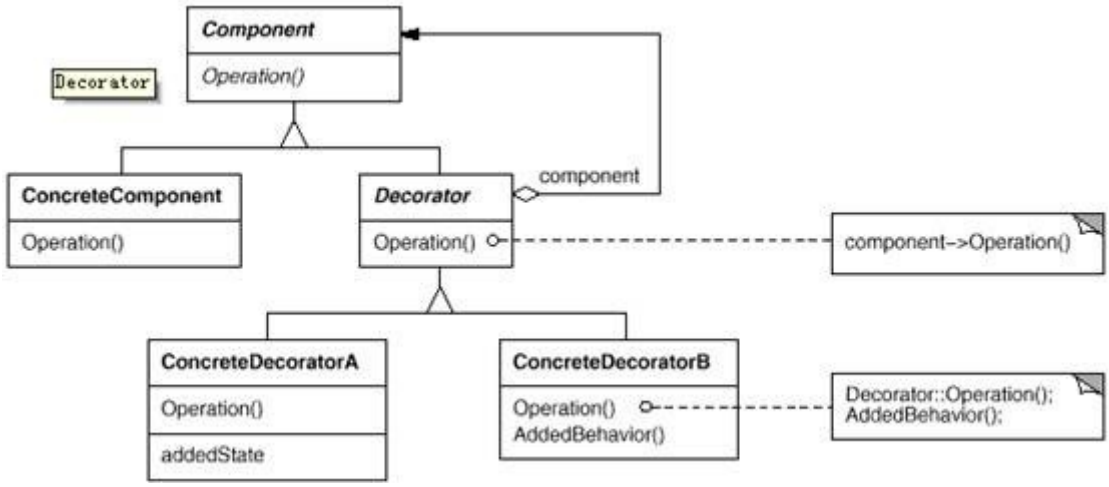
输出：

```
send 成功使用了代理 to ldc
<class '__main__.ReceiveClass'>
<class '__main__.Proxy'>
```

4、装饰模式【Decorator】

意图：

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。



适用性：

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。

处理那些可以撤消的职责。

当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，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。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

比如：

把每个要装饰的功能放在单独的类中，用这个类去包装所要装饰的对象，因此客户端可以有选择地、按顺序的使用装饰功能包装对象。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 9、装饰模式.py
@time: 2020/3/14 14:09
@author:LDC
'''


class Person(object):
    # 定义一个人类
    def __init__(self, name):
        self.name = name

    def show(self):
        print('{}穿着衣服'.format(self.name))


class Decorate(Person):
    # 定义一个装饰类
    component = None

    def __init__(self):
        pass

    def decorate(self, component):
```

```
        self.component = component

    def show(self):
        if self.component is not None:
            self.component.show()


class TShirts(Decorate):
    # 定义一个T恤类
    def __init__(self):
        pass

    def show(self):
        print('Big Tshirts')
        super(TShirts, self).show()


class BigTrouser(Decorate):
    # 定义一个裤子类
    def __init__(self):
        pass

    def show(self):
        print('Big Trouser')
        super(BigTrouser, self).show()


if __name__ == '__main__':
    ldc = Person('ldc')
    ts = TShirts()
    bt = BigTrouser()
    ts.decorate(ldc)
    bt.decorate(ldc)
    ts.show()
    bt.show()

"""
总结:
1.一般来说，通过继承可以获得父类的属性，还可以通过重载修改其方法。
2.装饰模式可以不以继承的方式而动态地修改类的方法。
3.装饰模式可以不以继承的方式而返回一个被修改的类。
"""

```

输出:

```
Big Tshirts
ldc穿着衣服
Big Trouser
ldc穿着衣服
```

5、桥接模式【Bridge】

意图：

将抽象部分与实现部分分离，使它们都可以独立的变化。

桥接模式的核心意图就是把类的实现独立出来，让他们各自变化。这样使每种实现的变化不会影响其他实现，从而达到应对变化的目的

适用性：

1．如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。 2．设计要求实例化角色的任何改变不应当影响客户端，或者说实例化角色的改变对客户端是完全透明的。 3．一个构件有多于一个的抽象化角色和实例化角色，系统需要它们之间进行动态耦合。 4．虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

比如：

就拿汽车在路上行驶的来说。即有小汽车又有公共汽车，它们都能在市区中的公路上行驶，也能在高速公路上行驶。这你会发现，对于交通工具（汽车）有不同的类型，然而它们所行驶的环境（路）也在变化，在软件系统中就要适应两个方面的变化？怎样实现才能应对这种变化呢？ 概述: 在软件系统中，某些类型由于自身的逻辑，它具有两个或多个维度的变化，那么如何应对这种“多维度的变化”？如何利用面向对象的技术来使得该类型能够轻松的沿着多个方向进行变化，而又不引入额外的复杂度？这就要使用Bridge模式。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 10_1、桥接模式.py
@time: 2020/3/14 14:42
@author:LDC
'''
"""
桥接模式
Bridge效果及实现要点：
1. Bridge模式使用“对象间的组合关系”解耦了抽象和实现之间固有的绑定关系，
使得抽象和实现可以沿着各自的维度来变化。

2. 所谓抽象和实现沿着各自维度的变化，即“子类化”它们，得到各个子类之后，
便可以任意它们发展，从而获得不同路上的不同汽车。

3. Bridge模式有时候类似于多继承方案，
但是多继承方案往往违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差。
Bridge模式是比多继承方案更好的解决方法。

4. Bridge模式的应用一般在“两个非常强的变化维度”，有时候即使有两个变化的维度，
但是某个方向的变化维度并不剧烈——换言之两个变化不会导致纵横交错的结果，并不一定要使用Bridge模式。
"""


class AbstractRoad(object):
    # 路基类
```

```
    car = None


class AbstractCar(object):
    # 车辆基类

    def run(self):
        raise NotImplementedError


class Street(AbstractRoad):
    # 市区街道
    def run(self):
        # 执行车辆对象的方法
        self.car.run()
        print("在市区街道上行驶")


class SpeedWay(AbstractRoad):
    # 高速公路
    def run(self):
        # 执行车辆对象的方法
        self.car.run()
        print("在高速公路上行驶")


class Car(AbstractCar):
    # 小汽车
    def run(self):
        # 被其它对象调用执行
        print("小汽车在")


class Bus(AbstractCar):
    # 公共汽车
    def run(self):
        # 被其它对象调用执行
        print("公共汽车在")


if __name__ == '__main__':
    # 小汽车在高速公路上行驶
    road1 = SpeedWay()
    road1.car = Car()
    road1.run()
    # 公共汽车在高速公路上行驶
    road2 = SpeedWay()
    road2.car = Bus()
    road2.run()
    # 公共汽车在市区上行驶
    road3 = Street()
    road3.car = Bus()
    road3.run()

"""
总结:
Bridge模式是一个非常有用的模式，也非常复杂，它很好的符合了开放-封闭原则和优先使用对象，而不是继承这两个面向对象原则
```

```
"""


```

输出：

```
小汽车在
在高速公路上行驶
公共汽车在
在高速公路上行驶
公共汽车在
在市区街道上行驶
公共汽车在

```

桥接模式（Bridge）来做(多维度变化); 结合上面的例子,增加一个维度"人",不同的人开着不同的汽车在不同的路上行驶(三个维度); 结合上面增加一个类"人",并重新调用.

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 10_2、桥接模式之多维度.py
@time: 2020/3/14 15:40
@author:LDC
'''

"""
结合上面的例子,增加一个维度"人",不同的人开着不同的汽车在不同的路上行驶(三个维度);
结合上面增加一个类"人",并重新调用.
"""


class AbstractRoad(object):
    # 路基类
    car = None


class AbstractCar(object):
    # 车辆基类

    def run(self):
        raise NotImplementedError


class People(object):
    # 定义一个人类
    road = None


class Street(AbstractRoad):
    # 市区街道
    def run(self):

```

```
        # 执行车辆对象的方法
        self.car.run()
        print("在市区街道上行驶")


class SpeedWay(AbstractRoad):
    # 高速公路
    def run(self):
        # 执行车辆对象的方法
        self.car.run()
        print("在高速公路上行驶")


class Car(AbstractCar):
    # 小汽车
    def run(self):
        # 被其它对象调用执行
        print("小汽车在")


class Bus(AbstractCar):
    # 公共汽车
    def run(self):
        # 被其它对象调用执行
        print("公共汽车在")


# 加上人
class Man(People):
    def drive(self):
        print("男人开着")
        self.road.run()  # 调用其它对象的执行方法


class Woman(People):
    def drive(self):
        print("女人开着")
        self.road.run()  # 调用其它对象的执行方法


if __name__ == '__main__':
    # 小汽车在高速公路上行驶
    road1 = SpeedWay()
    road1.car = Car()
    road1.run()
    # 公共汽车在高速公路上行驶
    road2 = SpeedWay()
    road2.car = Bus()
    road2.run()
    # 人开车
    road3 = Street()
    road3.car = Bus()

    p1 = Man()
    p1.road = road3
    p1.drive()

    p2 = Woman()
```

```
    p2.road = road2
    p2.drive()


```

输出：

```
小汽车在
在高速公路上行驶
公共汽车在
在高速公路上行驶
男人开着
公共汽车在
在市区街道上行驶
女人开着
公共汽车在
在高速公路上行驶
```

6、组合模式【Composite】

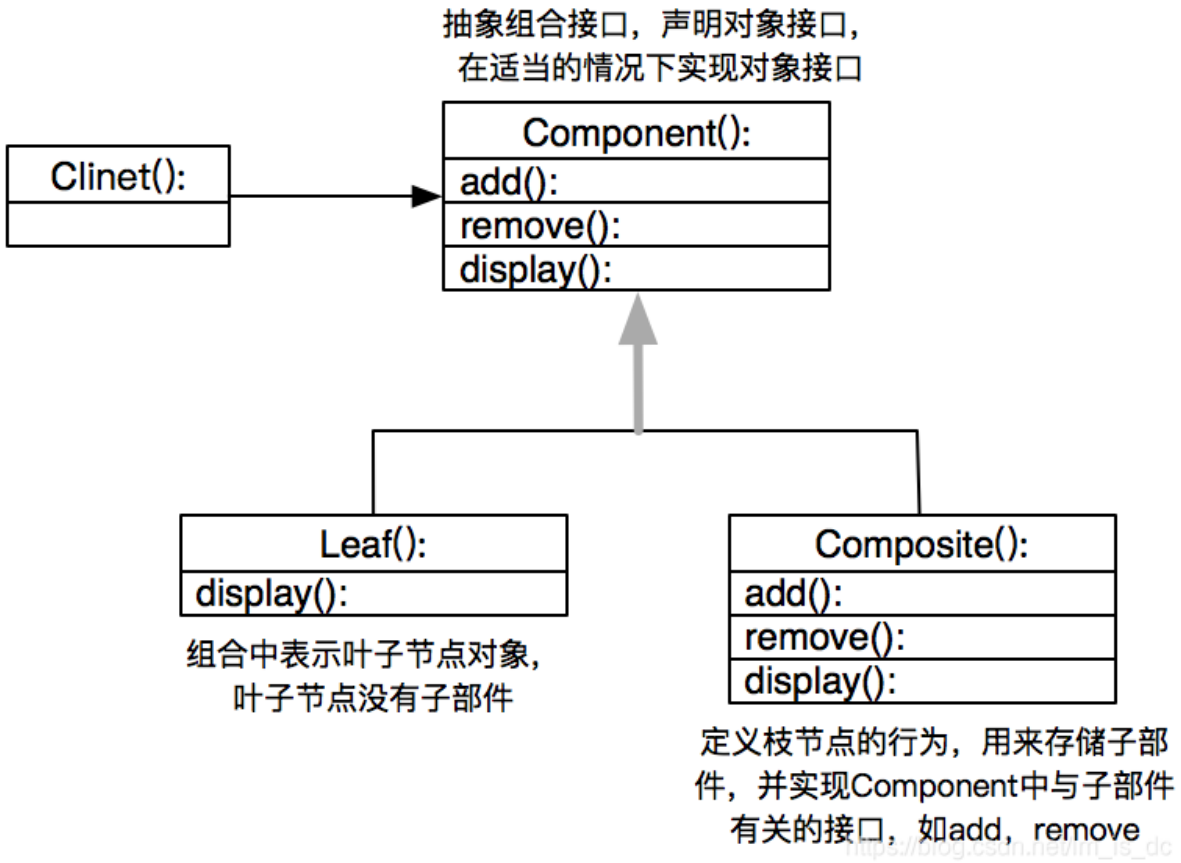
意图：

将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

Composite模式采用 **树性结构** 来实现普遍存在的对象容器，从而将**一对多**的关系转化为**一对一**的关系，使得客户代码可以一致地(复用)处理对象和对象容器， 无需关心处理的是单个的对象，还是组合的对象容器。

客户代码与纯粹的抽象接口——而非对象容器的内部实现结构——发生依赖，从而更能“应对变化“。

Composite模式在具体实现中，可以让父对象中的子对象反向追溯；如果父对象有频繁的遍历需求，可使用缓存技术来改善效率。



适用性：

在需要体现部分与整体层次的结构时；

希望用户忽略组合对象与单个对象的不同，统一的使用组合结构中的所有对象时。

比如：

会员卡消费，首先

1.我们的部件有，总店，分店，加盟店!

2.我们的部件共有的行为是：刷会员卡

3.部件之间的层次关系，也就是店面的层次关系是，总店下有分店、分店下可以拥有加盟店。

有了我们这几个必要条件后，要求就是目前店面搞活动当我在总店刷卡后，就可以累积相当于在所有下级店面刷卡的积分总额

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 11、组合模式.py
@time: 2020/3/14 15:49
@author:LDC
'''

"""
组合模式
```

```
Composite模式采用 树性结构 来实现普遍存在的对象容器，
从而将一对多的关系转化为一对一的关系，使得客户代码可以一致地(复用)处理对象和对象容器，
无需关心处理的是单个的对象，还是组合的对象容器。
客户代码与纯粹的抽象接口——而非对象容器的内部实现结构——发生依赖，
从而更能”应对变化“。
Composite模式在具体实现中，可以让父对象中的子对象反向追溯；
如果父对象有频繁的遍历需求，可使用缓存技术来改善效率。

会员卡消费，首先

1.我们的部件有，总店，分店，加盟店！
2.我们的部件共有的行为是：刷会员卡
3.部件之间的层次关系，也就是店面的层次关系是，总店下有分店、分店下可以拥有加盟店。
有了我们这几个必要条件后，要求就是目前店面搞活动当我在总店刷卡后，就可以累积相当于在所有下级店面刷卡的积分总额

"""


class Sotre(object):
    # 定义一个组织类，店面基类

    # 添加店面
    def add(self, store):
        pass

    # 删除店面
    def remove(self, store):
        pass

    # 显示店面
    def display(self, depth):
        pass

    # 刷消费卡
    def pay_by_card(self):
        pass


class BranceStore(Sotre):
    # 定义分店类（总店是第一个店，与其它分店功能类似）
    def __init__(self, name):
        self.name = name  # 店名
        self.my_store_list = []  # 开的分店

    def add(self, store):
        # 添加店面
        self.my_store_list.append(store)

    def remove(self, store):
        # 删除店面
        self.my_store_list.remove(store)

    def display(self, depth):
        # 显示店面
        # print(self.name, depth, self.my_store_list)
        print('{}-{}'.format(' ' * depth, self.name))
        for store in self.my_store_list:
```

```
            store.display(depth + 2)

    def pay_by_card(self):
        print("店面[%s]的积分已累加进该会员卡" % self.name)
        for s in self.my_store_list:
            s.pay_by_card()


class JoinStore(Sotre):
    # 定义加盟店

    def __init__(self, name):
        self.name = name

    def pay_by_card(self):
        print("店面[%s]的积分已累加进该会员卡" % self.name)

    def add(self, store):
        print("无添加子店权限")

    def remove(self, store):
        print("无删除子店权限")

    def display(self, depth):
        print('{}-{}'.format(' ' * depth, self.name))


if __name__ == '__main__':
    store = BranceStore('广州总店')
    brance = BranceStore('天河分店')
    store.add(brance)  # 总店开分店
    tx_join_brance = JoinStore('棠下加盟店')
    yg_join_brance = JoinStore('元岗加盟店')
    brance.add(tx_join_brance)  # 分店开加盟店
    brance.add(yg_join_brance)  # 分店开加盟店

    store.display(1)
    store.pay_by_card()

"""
总结:
这样在累积所有子店面积分的时候，就不需要去关心子店面的个数了，也不用关心是否是叶子节点还是组合节点了，
也就是说不管是总店刷卡，还是加盟店刷卡，都可以正确有效的计算出活动积分。
应用场景:
在需要体现部分与整体层次的结构时
希望用户忽略组合对象与单个对象的不同，统一的使用组合结构中的所有对象时
"""

```

输出:

```
-广州总店
    -天河分店
        -棠下加盟店
        -元岗加盟店
店面[广州总店]的积分已累加进该会员卡
店面[天河分店]的积分已累加进该会员卡
店面[棠下加盟店]的积分已累加进该会员卡
店面[元岗加盟店]的积分已累加进该会员卡

```

7、享元模式【Flyweight】

意图：

运用共享技术有效地支持大量细粒度的对象。

享元旨在优化性能和内存使用。 内部状态：享元对象中不会随环境改变而改变的共享部分。比如围棋棋子的颜色。 外部状态：随环境改变而改变、不可以共享的状态就是外部状态。比如围棋棋子的位置。

适用性：

程序中使用了大量的对象，造成很大的存储开销。

如果删除对象的外部状态，可以用相对较少的共享对象取代很多组对象，就可以考虑使用享元模式。

比如：

假设我们正在设计一个性能关键的游戏，例如第一人称射击（First-Person Shooter，FPS）游戏。在FPS游戏中，玩家（士兵）共享一些状态，如外在表现和行为。例如，在《反恐精英》游戏中，同一团队（反恐精英或恐怖分子）的所有士兵看起来都是一样的（外在表现）。同一个游戏中，（两个团队的）所有士兵都有一些共同的动作，比如，跳起、低头等（行为）。这意味着我们可以创建一个享元来包含所有共同的数据。当然，士兵也有许多因人而异的可变数据，这些数据不是享元的一部分，比如，枪支、健康状况和地理位置等。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 12、享元模式.py
@time: 2020/3/14 19:03
@author:LDC
'''
"""
享元模式
享元对象中不会随环境改变而改变的共享部分。比如围棋棋子的颜色。

构造一小片水果树的森林，小到能确保在单个终端页面中阅读整个输出。
然而，无论你构造的森林有多大，内存分配都保持相同。
"""

import random
```

```python
from enum import Enum

# 定义只有3种树的枚举类型
TreeType = Enum('TreeType', ('apple_tree', 'cherry_tree', 'peach_tree'))


class Tree:
    """
    把Tree类变换成一个元类，元类支持自引用。这意味着cls引用的是Tree类。
    当客户端要创建Tree的一个实例时，会以tree_type参数传递树的种类。
    树的种类用于检查是否创建过相同种类的树。
    如果是，则返回之前创建的对象；
    否则，将这个新的树种添加到池中，并返回相应的新对象，

    """
    pool = dict()  # 创建一个字典缓存。类属性（类的所有实例共享的一个变量）

    def __new__(cls, tree_type):
        obj = cls.pool.get(tree_type, None)  # 从缓存中获取树种类型
        if not obj:
            # 如果获取不到树类型就创建一个新的，并保存到缓存中
            obj = object.__new__(cls)
            cls.pool[tree_type] = obj
            obj.tree_type = tree_type
        return obj

    def render(self, age, x, y):
        """
        用于在屏幕上渲染一棵树。注意，享元不知道的所有可变（外部的）信息都需要客户端代码显示地传递。
        在当前案例中，每棵树都用到一个随机的年龄和一个x,y形式的位置。
        :param age: 树龄
        :param x: x轴
        :param y: y轴
        :return:
        """
        print('render a tree of type {} and age {} at ({}, {})'.format(self.tree_type, age, x, y))


def main():
    rnd = random.Random()
    age_min, age_max = 1, 30  # 树龄单位为年
    min_point, max_point = 0, 100  # 树的坐标0到100
    tree_counter = 0  # 树的数量
    for _ in range(10):
        # 随机生成10棵苹果树
        t1 = Tree(TreeType.apple_tree)
        t1.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point)
                  )
        tree_counter += 1

    for _ in range(3):
        # 随机生成10棵樱桃树
        t2 = Tree(TreeType.cherry_tree)
        t2.render(rnd.randint(age_min, age_max),
```

```
                    rnd.randint(min_point, max_point),
                    rnd.randint(min_point, max_point)
                    )
        tree_counter += 1

    for _ in range(5):
        # 随机生成10棵桃子树
        t3 = Tree(TreeType.peach_tree)
        t3.render(rnd.randint(age_min, age_max),
                    rnd.randint(min_point, max_point),
                    rnd.randint(min_point, max_point)
                    )
        tree_counter += 1

    print('渲染的树数目：{}'.format(tree_counter))
    print('实际上创建的树数目:{}'.format(len(Tree.pool)))
    t4 = Tree(TreeType.cherry_tree)
    t5 = Tree(TreeType.cherry_tree)
    t6 = Tree(TreeType.apple_tree)
    print(id(t4), id(t5), id(t6), )


if __name__ == '__main__':
    main()


"""
总结：
在单例的基础上做了改动，也就是当你实例化一个对象，
就判断你实例化的该对象（包含形参）是否存在父类的指定的字典，
存在就把之前实例化对象返回给你（等于没创建新的实例，而是赋值多一个变量而已，指向同一个内存地址），
如果不存在，就创建新的实例化对象返回，并且存放在指定字典
"""
```

输出：

```
render a tree of type TreeType.apple_tree and age 17 at (31, 39)
render a tree of type TreeType.apple_tree and age 24 at (18, 3)
render a tree of type TreeType.apple_tree and age 29 at (54, 15)
render a tree of type TreeType.apple_tree and age 7 at (85, 88)
render a tree of type TreeType.apple_tree and age 30 at (76, 90)
render a tree of type TreeType.apple_tree and age 10 at (34, 77)
render a tree of type TreeType.apple_tree and age 20 at (65, 74)
render a tree of type TreeType.apple_tree and age 29 at (49, 63)
render a tree of type TreeType.apple_tree and age 6 at (74, 34)
render a tree of type TreeType.apple_tree and age 20 at (39, 95)
render a tree of type TreeType.cherry_tree and age 8 at (26, 81)
render a tree of type TreeType.cherry_tree and age 15 at (96, 58)
render a tree of type TreeType.cherry_tree and age 9 at (94, 93)
render a tree of type TreeType.peach_tree and age 25 at (23, 1)
render a tree of type TreeType.peach_tree and age 8 at (8, 89)
render a tree of type TreeType.peach_tree and age 3 at (9, 82)
render a tree of type TreeType.peach_tree and age 19 at (52, 50)
render a tree of type TreeType.peach_tree and age 15 at (40, 40)
渲染的树数目：18
实际上创建的树数目:3
```

```
46488880 46488880 46406224

```

六、行为型模式实现

用于在不同的实体间进行通信，为实体之间的通信提供更容易，更灵活的通信方法。

1、模板方法模式【Template Method】

意图：

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。

在多个算法或框架具有类似或相同的逻辑的时候，可以使用模板方法模式，以实现代码重用。

适用性：

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。

首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。

最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook ”操作，这样就只允许在这些点进行扩展

比如：

某超类的子类中有公有的方法，并且逻辑基本相同，可以使用模板模式。必要时可以使用钩子方法约束其行为。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 13、模板方法模式.py
@time: 2020/3/14 19:55
@author:LDC
'''
"""
模板方法模式
在多个算法或框架具有类似或相同的逻辑的时候，可以使用模板方法模式，以实现代码重用。

模板方法模式是一种基于继承的代码复用技术，它是一种类行为型模式。
模板方法模式是结构最简单的行为型设计模式，在其结构中只存在父类与子类之间的继承关系。
通过使用模板方法模式，可以将一些复杂流程的实现步骤封装在一系列基本方法中，
在抽象父类中提供一个称之为模板方法的方法来定义这些基本方法的执行次序，而通过其子类来覆盖某些步骤，
从而使得相同的算法框架可以有不同的执行结果。
模板方法模式提供了一个模板方法来定义算法框架，而某些具体步骤的实现可以在其子类中完成。

```

```python

"""


class Register(object):
    '''用户登录/注册模板接口'''

    def register(self):
        pass

    def login(self):
        pass

    def auth(self):
        # 模板方法：定义好具体的算法步骤或框架
        self.register()
        self.login()


class RegisterByQQ(Register):
    '''qq注册'''
    # 子类1：按需重新定义模板方法中的算法操作，即重新定义登录和注册方法
    def register(self):
        print("---用qq注册-----")

    def login(self):
        print('----用qq登录-----')


class RegisterByWeiChat(Register):
    '''微信注册'''
    # 子类2：按需重新定义模板方法中的算法操作，即重新定义登录和注册方法
    def register(self):
        print("---用微信注册-----")

    def login(self):
        print('----用微信登录-----')


if __name__ == "__main__":
    register1 = RegisterByQQ()
    register1.auth()

    register2 = RegisterByWeiChat()
    register2.auth()


"""
总结：
主要角色：
接口：通常是抽象基类，定义模板方法中需要的各项操作。
模板方法：即模板算法，定义好各项操作的执行顺序或算法框架。
真实对象：子类通过重新实现接口中的各项操作，以便让模板方法实现不同的功能。
优缺点：
优点：因为子类的实现是根据基类中的模板而来的，所以可以实现代码重用，
因为有时候我们需要修改的只是模板方法中的部分操作而已。
缺点：此模式的维护有时候可能会很麻烦，因为模板方法是固定的，
一旦模板方法本身有修改的时候，就可能对其他的相关实现造成影响。
```

```
"""
```

输出：

```
---用qq注册-----
----用qq登录-----
---用微信注册-----
----用微信登录-----
```

2、观察者模式【Observer】

意图：

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。[典型的发布订阅]

适用性：

当一个抽象模型有两个方面, 其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

当对一个对象的改变需要同时改变其它对象, 而不知道具体有多少对象有待改变。

当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之, 你不希望这些对象是紧密耦合的。

比如：

市里新修了一个图书馆，现在招募一个图书管理员叫T，T知道图书馆里的图书更新和借阅等信息。现在有三个同学甲乙丙想去了解以后几个月的图书馆图书信息和借阅信息，于是它们去T那里注册登记。当图书馆图书更新后，T就给注册了的同学发送图书更新信息。三个月后，丙不需要知道图书更新信息了，于是就去T那儿注销了它的信息。所以，以后，只有甲乙会收到消息。几个月后，丁也去图书馆注册了信息，所以以后甲乙丁会收到图书更新信息。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 14、观察者模式.py
@time: 2020/3/14 20:35
@author:LDC
'''
"""
观察者模式
观察者（Observer）模式又名发布-订阅（Publish/Subscribe）模式
当我们希望一个对象的状态发生变化，那么依赖于它的所有对象都能相应变化(获得通知)，
那么就可以用到Observer模式，  其中的这些依赖对象就是观察者的对象，
那个要发生变化的对象就是所谓'观察者'

"""

```

```python

class ObserverBase(object):
    # 观察者基类，放哨者
    def __init__(self):
        self._observerd_list = []  # 被通知对象

    def attach(self, observe_subject):
        """
        添加要观察的对象
        """
        if observe_subject not in self._observerd_list:
            self._observerd_list.append(observe_subject)
            print("【{}】已经将【{}】加入观察队列...".format(self.name, observe_subject))

    def detach(self, observe_subject):
        """
        解除观察关系
        :param observe_subject:
        :return:
        """
        try:
            self._observerd_list.remove(observe_subject)
            print("不再观察【{}】".format(observe_subject))
        except ValueError:
            pass

    def notify(self):
        """
        通知所有被观察者
        :return:
        """
        for observer in self._observerd_list:
            observer.update(self)


class Observer(ObserverBase):
    # 观察者类
    def __init__(self, name):
        super(Observer, self).__init__()
        self.name = name
        self._msg = ''

    @property  # 外部执行o.msg 去掉括号
    def msg(self):
        # 当前状况
        return self._msg

    @msg.setter  # 设置属性(一个方法变成一个静态的属性)
    def msg(self, content):
        self._msg = content
        self.notify()


class ATeamViews(object):
    """
    A军观察者
    """
```

```
76
77      def update(self, observer_subject):
78          print("A军：收到【{}】消息【{}】".format(observer_subject.name,
    observer_subject.msg))
79
80
81  class BTeamViews(object):
82      """
83      B军观察者
84      """
85
86      def update(self, observer_subject):
87          print("B军：收到【{}】消息【{}】".format(observer_subject.name,
    observer_subject.msg))
88
89
90  if __name__ == '__main__':
91      observer_A = Observer("A军放哨者")
92      observer_B = Observer("B军放哨者")
93
94      A_jun = ATeamViews()
95      B_jun = BTeamViews()
96
97      observer_A.attach(A_jun)
98      observer_A.attach(B_jun)
99      observer_B.attach(B_jun)
100
101     observer_A.msg = "\033[32;1mB军来了...\033[0m"
102
103     observer_B.msg = "\033[31;1m前方发现A军，请紧急撤离，不要告诉A军\033[0m"
104
105 """
106 总结：
107 通过代码了解其实就是发布订阅模式
108 优点：
109 独立封装，互不影响：观察者和被观察者都是独自封装好的，观察者之间并不会相互影响
110 热插拔：在软件运行中，可以动态添加和删除观察者
111 """
```

输出：

```
1  【A军放哨者】已经将【<__main__.ATeamViews object at 0x03571A30>】加入观察队列...
2  【A军放哨者】已经将【<__main__.BTeamViews object at 0x03571A50>】加入观察队列...
3  【B军放哨者】已经将【<__main__.BTeamViews object at 0x03571A50>】加入观察队列...
4  A军：收到【A军放哨者】消息【B军来了...】
5  B军：收到【A军放哨者】消息【B军来了...】
6  B军：收到【B军放哨者】消息【前方发现A军，请紧急撤离，不要告诉A军】
7
```

3、状态模式【State】

意图：

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

适用性：

一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。

一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常, 有多个操作包含这一相同的条件结构。State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

比如：

在目前主流的RPG（Role Play Game，角色扮演游戏）中，使用状态模式可以对游戏角色进行控制，游戏角色的升级伴随着其状态的变化和行为的变化。对于游戏程序本身也可以通过状态模式进行总控，一个游戏活动包括开始、运行、结束等状态，通过对状态的控制可以控制系统的行为，决定游戏的各个方面，因此可以使用状态模式对整个游戏的架构进行设计与实现。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 15、状态模式.py
@time: 2020/3/15 10:44
@author:LDC
'''
"""
状态模式
当控制一个对象的状态转换的条件表达式过于复杂时,把状态的判断逻辑转移到表示不同状态的一系列类当中,
这可以把复杂的判断逻辑简化
"""


class State(object):
    """定义一个状态基类"""

    def toggle_amfm(self):
        # 定义一个切换状态方法
        pass

    def scan(self):
        pass


class AMState(State):
    # 定义一个AM频道类
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["1250", "1380", "1510"]
        self.pos = 0
```

```
        self.name = "AM"

    def scan(self):
        # 定义一个共用的方法，扫描刻度盘下一个状态
        self.pos += 1
        if self.pos == len(self.stations):
            self.pos = 0
        print("扫描中...状态是", self.stations[self.pos], self.name)

    def toggle_amfm(self):
        # 定义一个切换AM/FM频道的方法
        print("转换到FM频道")
        self.radio.state = self.radio.fm_state


class FMState(State):
    # 定义一个FM频道类
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["81.3", "89.1", "103.9"]
        self.pos = 0
        self.name = "FM"

    def scan(self):
        # 定义一个共用的方法，扫描刻度盘下一个状态
        self.pos += 1
        if self.pos == len(self.stations):
            self.pos = 0
        print("扫描中...状态是", self.stations[self.pos], self.name)

    def toggle_amfm(self):
        # 定义一个切换AM/FM频道的方法
        print("转换到AM频道")
        self.radio.state = self.radio.am_state


class Radio(object):
    # 定义一个收音机,有一个扫描按钮，和切换AM/Fm的开关
    def __init__(self):
        self.am_state = AMState(self)
        self.fm_state = FMState(self)
        self.state = self.am_state

    def toggle_amfm(self):
        self.state.toggle_amfm()

    def scan(self):
        self.state.scan()


if __name__ == '__main__':
    radio = Radio()
    actions = [radio.scan] * 2 + [radio.toggle_amfm] + [radio.scan] * 2
    actions = actions * 2
    for action in actions:
        action()

"""
```

```
总结：
就是通过一个对象的方法调用已经存在其他类对象的状态
"""

```

输出：

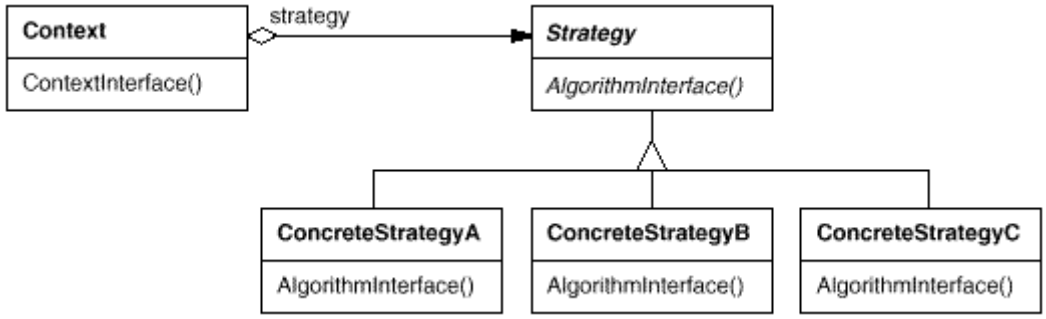
```
扫描中...状态是 1380 AM
扫描中...状态是 1510 AM
转换到FM频道
扫描中...状态是 89.1 FM
扫描中...状态是 103.9 FM
扫描中...状态是 81.3 FM
扫描中...状态是 89.1 FM
转换到AM频道
扫描中...状态是 1250 AM
扫描中...状态是 1380 AM
```

4、策略模式【Strategy】

意图：

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护的问题。



适用性：

许多相关的类仅仅是行为有异。

“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。

需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时，可以使用策略模式。

算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

一个类定义了多种行为, 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句

比如：

商场搞活动，可以按正常收费、打折收费、返利收费等支付方式，通过 一个具体的策略类来控制支付方式（也就是不同的算法）

代码：

```

```

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 16、策略模式.py
@time: 2020/3/15 12:03
@author:LDC
'''
"""
策略模式
策略模式(Strategy Pattern):它定义了算法家族,分别封装起来,
让他们之间可以相互替换,此模式让算法的变化,不会影响到使用算法的客户.

商场搞活动，可以按正常收费、打折收费、返利收费等支付方式，通过 一个具体的策略类来控制支付方式（也就是不同的算法）
"""


class PaySuper(object):
    # 定义一个支付抽象类

    def pay(self, money):
        pass


class PayNormal(PaySuper):
    # 正常支付子类
    def pay(self, money):
        return money


class PayRebate(PaySuper):
    # 打折支付子类
    def __init__(self, discount):
        self.discount = discount

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


class PayReturn(PaySuper):
    # 返利支付子类
    def __init__(self, money_condition, money_return):
        self.money_condition = money_condition
        self.money_return = money_return

    def pay(self, money):
        # 支付满money_conditionc才返利money_return
        if money >= self.money_condition:
            return money - self.money_return
        return money


class Context(object):
    # 定义一个具体的策略类，管理支付方式
    def __init__(self, paysuper):
        self.paysuper = paysuper
```

```
    def get_result(self, money):
        return self.paysuper.pay(money)


if __name__ == '__main__':
    money = round(float(input('商品原价：')), 2)
    strategy = dict()
    strategy['1'] = Context(PayNormal())  # 正常支付
    strategy['2'] = Context(PayRebate(0.8))  # 打折支付
    strategy['3'] = Context(PayReturn(100, 10))  # 返利支付
    while True:
        mode = input("选择支付方式：1.原价  2.打8折  3.满100减10 4. 取消支付\r\n")
        if mode not in strategy:
            break
        pay_mode = strategy[mode]
        print("需要支付{}元".format(pay_mode.get_result(money)))


"""
总结：
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
"""
```

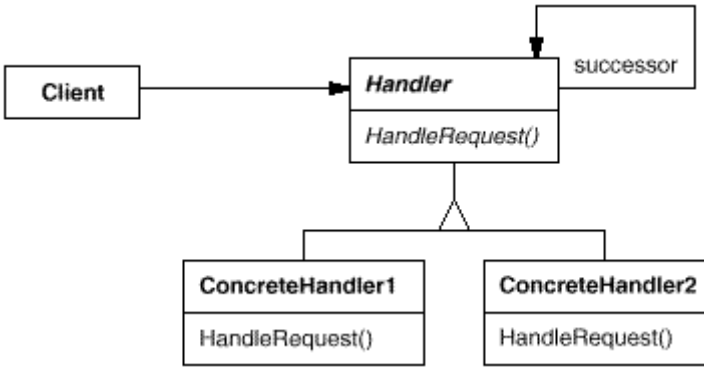
输出：

```
商品原价：100
选择支付方式：1.原价  2.打8折  3.满100减10 4. 取消支付
2
需要支付80.0元
选择支付方式：1.原价  2.打8折  3.满100减10 4. 取消支付
3
需要支付90.0元
选择支付方式：1.原价  2.打8折  3.满100减10 4. 取消支付
1
需要支付100.0元
选择支付方式：1.原价  2.打8折  3.满100减10 4. 取消支付

```

5、职责链模式【Chain of Responsibility】

意图：

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。



适用性：

有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。

你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

可处理一个请求的对象集合应被动态指定。

比如：

比如费用报销找上级领导审批，不同的级别可以审批不同的金额。这时候就可以使用职责链模式。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 职责链模式.py
@time: 2020/3/15 17:33
@author:LDC
'''
"""
职责链模式
在调用时要定义好哪个实例是哪个实例的职责上一级，
请求沿着定义的链条传递给可以处理请求的对象
1、有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定；
2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；
3、处理一个请求的对象集合应被动态指定。

比如费用报销找上级领导审批，不同的级别可以审批不同的金额。这时候就可以使用职责链模式。
"""


class HandlerBase(object):
    # 处理基类
    def successor(self, successor):  # 下一个处理者
        self._successor = successor


class RequestHandlerL1(HandlerBase):
    # 第一级请求处理者
    name = "TeamLeader"  # 小组领导

    def handle(self, request):
        if request < 500:
            print("审批者【{}】,处理请求金额【{}】元，审批结果【审批通过】".format(self.name, request))
        else:
```

```
            print("\033[31;1m【%s】无权审批,交给下一个审批者\033[0m" % self.name)
            self._successor.handle(request)


class RequestHandlerL2(HandlerBase):
    # 第一级请求处理者
    name = "DeptManager"  # 副经理

    def handle(self, request):
        if request < 5000:
            print("审批者【{}】,处理请求金额【{}】元，审批结果【审批通过】".format(self.name, request))
        else:
            print("\033[31;1m【%s】无权审批,交给下一个审批者\033[0m" % self.name)
            self._successor.handle(request)


class RequestHandlerL3(HandlerBase):
    # 第一级请求处理者
    name = "CEO"  # 首席执行官

    def handle(self, request):
        if request < 10000:
            print("审批者【{}】,处理请求金额【{}】元，审批结果【审批通过】".format(self.name, request))
        else:
            print("\033[31;1m【%s】钱太多了，不批\033[0m" % self.name)
            # self._successor.handle(request)


class RequestAPI(object):
    # 定义一个请求接口类
    h1 = RequestHandlerL1()
    h2 = RequestHandlerL2()
    h3 = RequestHandlerL3()

    h1.successor(h2)
    h2.successor(h3)

    def __init__(self, name, amount):
        self.name = name
        self.amount = amount

    def handle(self):
        # 统一请求接口
        self.h1.handle(self.amount)


if __name__ == '__main__':
    r1 = RequestAPI('ldc', 8000)
    r1.handle()
    print(r1.__dict__)


"""
总结:
接收者和发送者都没有对方的明确信息，且链中的对象自己并不知道链的结构，
职责链可简化对象的相互连接，他们仅需保持一个指向后继者的引用，
```

```
92 而不需要保持他所有候选接收者的引用，大大降低了耦合度，可以随时增加或修改处理一个请求的结构
93 """
```

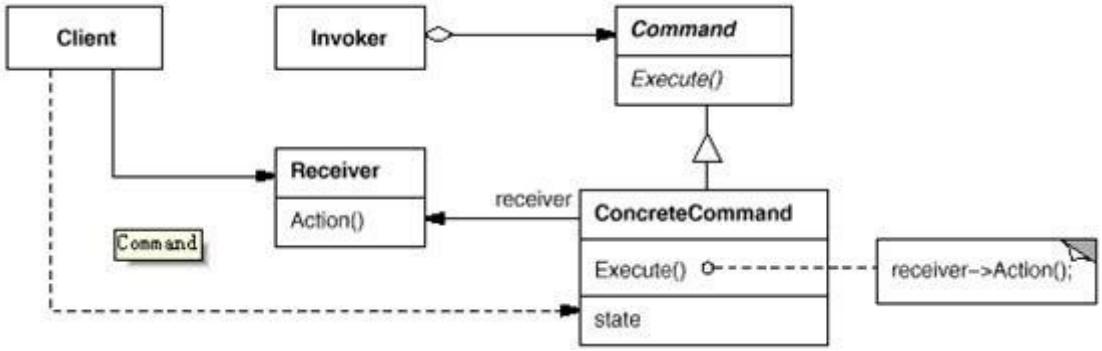
输出：

```
【TeamLeader】无权审批,交给下一个审批者
【DeptManager】无权审批,交给下一个审批者
审批者【CEO】,处理请求金额【8000】元，审批结果【审批通过】
{'name': 'ldc', 'amount': 8000}
```

6、命令模式【Command】

意图：

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。



抽象出待执行的动作以参数化某对象，你可用过程语言中的回调（callback）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。**Command 模式是回调机制的一个面向对象的替代品。**

适用性：

在不同的时刻指定、排列和执行请求。一个Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

支持取消操作。Command的Excute 操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command 接口必须添加一个Unexecute操作，该操作取消上一次Execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用Unexecute和Execute来实现重数不限的“取消”和“重做”。

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在Command接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用Execute操作重新执行它们。

用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务(transaction)的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

比如：

1、命令设计模式帮助我们将一个操作（**撤销、重做、复制、粘贴等**）封装成一个对象。简而言之，这意味着创建一个类，包含实现该操作所需要的所有逻辑和方法。这样做的优势如下所述。

2、我们并不需要直接执行一个命令。命令可以按照希望执行。 调用命令的对象与指导如何执行命令的对象解耦。调用者无需知道命令的任何实现细节。 如果有意义，可以把多个命令组织起来，这样调用者能够按顺序执行它们。例如，在实现一个**多层撤销命令**时，这是很有用的。

3、**事务型行为**和**日志记录**：事务型行为和日志记录对于为变更记录一份持久化日志是很重要的。操作系统用它来从系统崩溃中恢复，关系型数据库用它来实现事务，文件系统用它来实现快照，而安装程序（向导程序）用它来恢复取消的安装。

4、**宏**：在这里，宏是指一个动作序列，可在任意时间点按要求进行录制和执行。流行的编辑器（比如，Emacs和Vim）都支持宏。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 18、命令模式.py
@time: 2020/3/15 17:58
@author:LDC
'''
"""
命令模式
结构组成：
命令角色，具体命令角色，接收者角色，请求者（调用者）角色（Invoker），客户角色（Client）
"""
import abc


class Command(metaclass=abc.ABCMeta):
    # 命令抽象类
    @abc.abstractmethod
    def execute(self):
        # 命令对象对外只提供execute方法
        pass


class VmReciver(object):
    # 命令接收者，真正执行命令的地方

    def start(self):
        print("启动虚拟机")

    def stop(self):
        print("关闭虚拟机")


class StartVmCommand(Command):
    # 具体命令角色：开启虚拟机的命令
    def __init__(self, recevier):
        self.recevier = recevier

    def execute(self):
        # 真正执行命令的时候命令接收者开启虚拟机
        self.recevier.start()


```

```
class StopVmCommand(Command):
    # 具体命令角色：关闭虚拟机的命令
    def __init__(self, recevier):
        self.recevier = recevier

    def execute(self):
        # 真正执行命令的时候命令接收者开启虚拟机
        self.recevier.stop()


class ClientInvoker(object):
    # 请求者（调用者）角色,客户角色
    def __init__(self, command):
        self.command = command

    def do(self):
        self.command.execute()


if __name__ == '__main__':
    recevier = VmReciver()  # 创建接收者角色
    start_command = StartVmCommand(recevier)  # 创建一个开启虚拟机命令
    client = ClientInvoker(start_command)  # 创建一个调用者角色
    client.do()

    # 调用其它命令
    stop_command = StopVmCommand(recevier)
    client.command = stop_command
    client.do()
"""
总结：
优点：
降低对象之间的耦合度。
新的命令可以很容易地加入到系统中。
可以比较容易地设计一个组合命令。
调用同一方法实现不同的功能
缺点：
使用命令模式可能会导致某些系统有过多的具体命令类。
因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。
"""

```

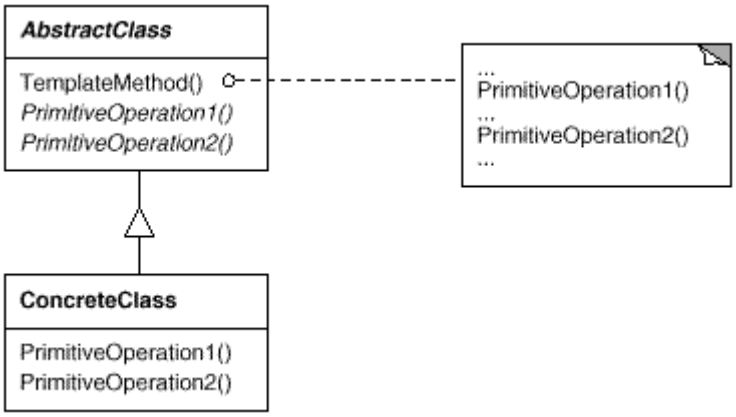
输出：

```
启动虚拟机
关闭虚拟机
```

7、访问者模式【Visitor】

意图：

1)对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。
2)需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。



适用性：

访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

比如：

安排不同年份的财务报表给不同的角色分析，这就是访问者模式的魅力；访问者模式的核心是在保持原有数据结构的基础上，实现多种数据的处理方法，该方法的角色就是访问者。

代码：

```python
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 19、访问者模式.py
@time: 2020/3/15 19:26
@author:LDC
'''
"""
访问者模式
安排不同年份的财务报表给不同的角色分析，这就是访问者模式的魅力；
访问者模式的核心是在保持原有数据结构的基础上，实现多种数据的处理方法，
该方法的角色就是访问者。
"""


class Finance(object):
    # 定义一个财务类
    def __init__(self):
        self.salesvolume = None  # 销售额
        self.cost = None  # 成本
        self.history_salesvolume = None  # 历史销售额
        self.history_cost = None  # 历史成本

    def set_salesvolume(self, value):
        self.salesvolume = value

    def set_cost(self, value):
        self.cost = value

    def set_history_salesvolume(self, value):
        self.history_salesvolume = value
```

```
34
35      def set_history_cost(self, value):
36          self.history_cost = value
37
38      def accept(self):
39          pass
40
41
42  class FinanceYear(Finance):
43      def __init__(self, year):
44          Finance.__init__(self)
45          self.analyst = []
46          self.year = year
47
48      def add_analyst(self, worker):
49          # 增加分析师来分析数据
50          self.analyst.append(worker)
51
52      def accept(self):
53          # 分析师分析数据
54          for a in self.analyst:
55              a.visit(self)
56
57
58  class Accounting(object):
59      # 会计
60      def __init__(self):
61          self.name = '会计'
62          self.duty = '计算报表'
63
64      def visit(self, year_data):
65          print('我现在分析的是{}年的数据'.format(year_data.year))
66          print('我的身份是:{}，职责：{}'.format(self.name, self.duty))
67          print('本年度纯利润:{}'.format(year_data.salesvolume -
    year_data.cost))
68          print('----------------------------------------')
69
70
71  class Audit:
72      # 财务总监
73      def __init__(self):
74          self.name = '财务总监'
75          self.duty = '分析业绩'
76
77      def visit(self, year_data):  # 要把具体哪一年的数据传给分析师，让分析师去分析
78          print('我现在分析的是{}年的数据'.format(year_data.year))
79          print('我的身份是:{}，职责：{}'.format(self.name, self.duty))
80          if year_data.salesvolume - year_data.cost >
    year_data.history_salesvolume - year_data.history_cost:
81              msg = '较同期上涨'
82          else:
83              msg = '较同期下跌'
84          print('本年度公司业绩:{}'.format(msg))
85          print('---------------------------------')
86
87
88  class Advisor:
89      # 战略顾问
```

```
    def __init__(self):
        self.name = '战略顾问'
        self.duty = '制定明年策略'

    def visit(self, year_data):
        print('我现在分析的是{}年的数据'.format(year_data.year))
        print('我的身份是:{}，职责：{}'.format(self.name, self.duty))
        if year_data.salesvolume > year_data.history_salesvolume:
            msg = '行业上涨，扩大规模'
        else:
            msg = '行业下跌，减少规模'
        print('本年度公司业绩:{}'.format(msg))
        print('------------------------------')


class AnalyseData(object):
    # 执行分析
    def __init__(self):
        self.datalist = []  # 需要处理的数据列表

    def add_data(self, year_data):
        self.datalist.append(year_data)

    def remove_data(self, year_data):
        self.datalist.remove(year_data)

    def visit(self):
        for d in self.datalist:
            d.accept()


if __name__ == '__main__':
    w = AnalyseData()  # 计划安排财务，总监，顾问对2020年数据进行分析
    finance_2020 = FinanceYear(2020)  # 2020年财务数据
    finance_2020.set_salesvolume(200)
    finance_2020.set_cost(90)
    finance_2020.set_history_salesvolume(190)
    finance_2020.set_history_cost(80)

    accounting = Accounting()
    audit = Audit()
    advisor = Advisor()

    finance_2020.add_analyst(accounting)  # 会计参与2020年的数据分析，然后执行了自己的visit方法
    finance_2020.add_analyst(audit)
    finance_2020.add_analyst(advisor)
    # finance_2020.accept() # 也可以直接这样调用
    w.add_data(finance_2020)
    w.visit()
"""
总结：
访问者模式优点
1)使得数据结构和作用于结构上的操作解耦，使得操作集合可以独立变化。
2)添加新的操作或者说访问者会非常容易。
3)将对各个元素的一组操作集中在一个访问者类当中。
4)使得类层次结构不改变的情况下，可以针对各个层次做出不同的操作，而不影响类层次结构的完整性。
```

```
5)可以跨越类层次结构，访问不同层次的元素类，做出相应的操作。
6)如果操作的逻辑改变，我们只需要改变访问者的实现就够了，而不用去修改其他所有的类。
7)添加新的访问者到系统变得容易。只需要改变一下访问者接口以及其实现。已经存在的访问者不会被干扰影响。



访问者模式缺点
1)实现起来比较复杂，会增加系统的复杂性。
2)visit()方法的返回值的类型在设计系统式就需要明确。
不然，就需要修改访问者的接口以及所有接口实现。
另外如果访问者接口的实现太多，系统的扩展性就会下降。

"""

```

输出：

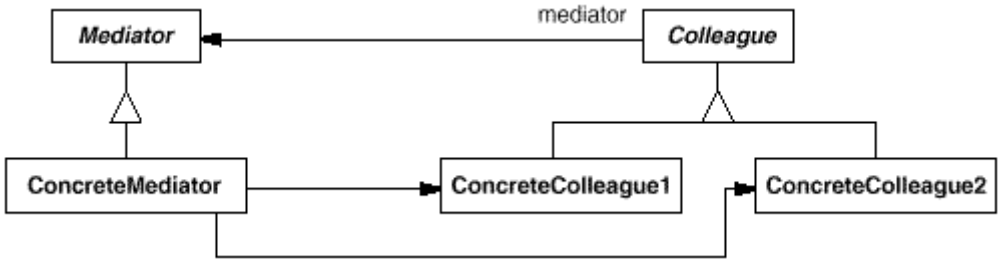
```
我现在分析的是2020年的数据
我的身份是:会计，职责：计算报表
本年度纯利润:110
---------------------------------------
我现在分析的是2020年的数据
我的身份是:财务总监，职责：分析业绩
本年度公司业绩:较同期下跌
---------------------------------
我现在分析的是2020年的数据
我的身份是:战略顾问，职责：制定明年策略
本年度公司业绩:行业上涨，扩大规模
------------------------------
```

8、调停者模式【Mediator】

也叫中介者模式。

意图：

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。



适用性：

一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。

想定制一个分布在多个类中的行为，而又不想生成太多的子类。

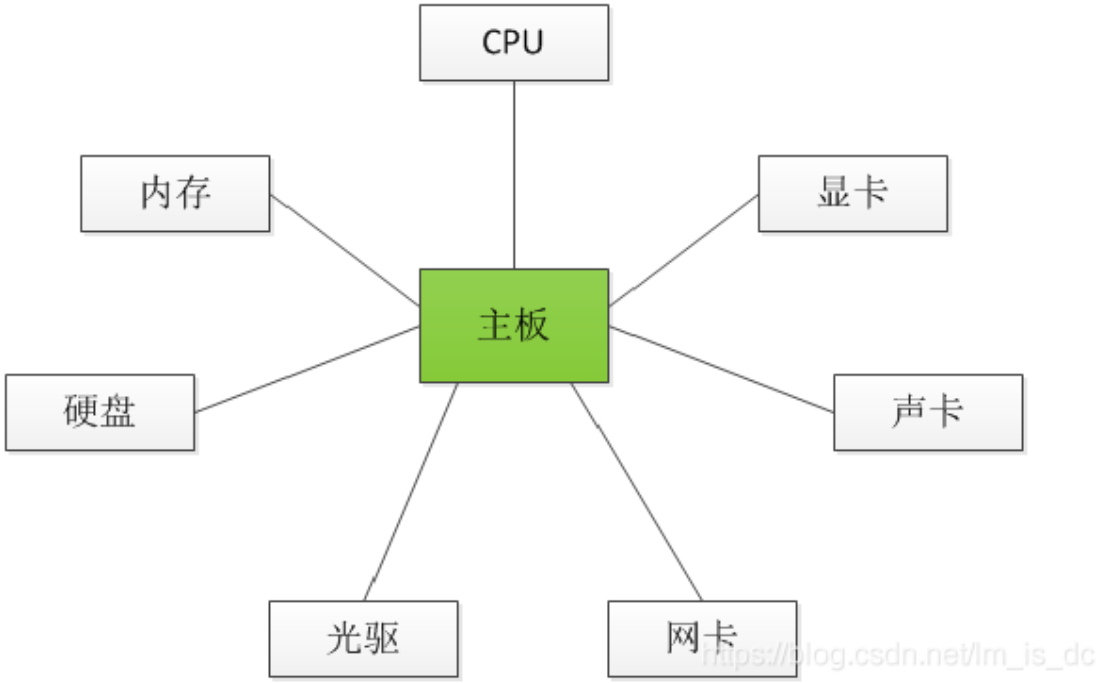
比如：

1、消息中间件

各个组件互相通信时，通过消息中间件来操作，不直接与组件联系。

2、电脑主板

电脑里面各个配件之间的交互，主要是通过主板来完成的。如果电脑里面没有了主板，那么各个配件之间就必须自行相互交互，以互相传送数据。而且由于各个配件的接口不同，相互之间交互时，还必须把数据接口进行转换才能匹配上。



代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 20、调停者模式.py
@time: 2020/3/15 21:59
@author:LDC
'''
"""
调停者模式/中介者模式
将类与类之间的复杂相互调用，由一个类来协调。解耦
以生产者和消费者之间的销售作为一个中介者，用对象来表示生产和购买及流通这个过程
"""


class Mediator(object):
    # 定义一个抽象中介者类
    def __init__(self):
        self.name = '中介者'
        self.consumer = None
        self.producer = None

    def sale(self):
        # 销售
        self.consumer.shopping(self.producer.name)

    def shopping(self):
```

```
        # 购买
        self.producer.sale(self.consumer.name)

    def profit(self):
        # 利润
        print('中介: 我净赚: {}'.format(self.consumer.price -
self.producer.price))

    def complete(self):
        # 交易
        self.shopping()  # 先进货后购买
        self.sale()
        self.profit()


class Consumer(object):
    # 消费者
    def __init__(self, product, price):
        self.name = "消费者"
        self.product = product
        self.price = price

    def shopping(self, name):
        # 买东西
        print("{}：购买了{}的{},价格{}".format(self.name, name, self.product,
self.price))


class Producer(object):
    # 生产者
    def __init__(self, product, price):
        self.name = "生产者"
        self.product = product
        self.price = price

    def sale(self, name):
        # 卖东西
        print("{}:向{}销售{},价格{}".format(self.name, name, self.product,
self.price))


if __name__ == '__main__':
    mediator = Mediator()  # 创建一个中介者
    consumer = Consumer('手机', 3000)  # 创建一个消费者
    producer = Producer('手机', 2500)  # 创建一个生产者
    mediator.consumer = consumer
    mediator.producer = producer
    mediator.complete()

"""
总结：

调停者模式的优点
调停者对象的存在保证了对象结构上的稳定，
也就是说，系统的结构不会因 为新对象的引入造成大量的修改工作。
    ●松散耦合
    调停者模式通过把多个组件对象之间的交互封装到调停者对象里面，从而使得组件对象之间松散耦
合，
```

```
    基本上可以做到互补依赖。这样一来，组件对象就可以独立地变化和复用，而不再像以前那样“牵一处而动全身”了。
    ●集中控制交互
    多个组件对象的交互，被封装在调停者对象里面集中管理，使得这些交互行为发生变化的时候，
    只需要修改调停者对象就可以了，当然如果是已经做好的系统，那么就扩展调停者对象，而各个组件类不需要做修改。
    ●多对多变成一对多
    没有使用调停者模式的时候，组件对象之间的关系通常是多对多的，引入调停者对象以后，
    调停者对象和组件对象的关系通常变成双向的一对多，这会让对象的关系更容易理解和实现。
调停者模式的缺点
    调停者模式的一个潜在缺点是，过度集中化。如果组件对象的交互非常多，而且比较复杂，当这些复杂性全部集中到调停者的时候，
    会导致调停者对象变得十分复杂，而且难于管理和维护。
"""

```

输出：

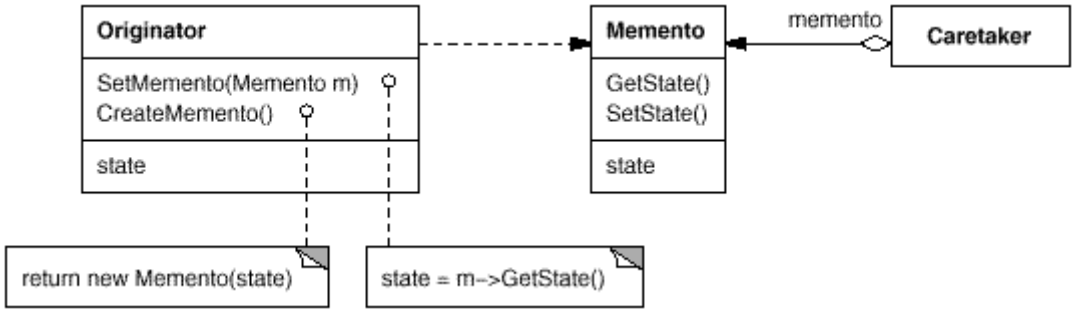
```
生产者:向消费者销售手机,价格2500
消费者：购买了生产者的手机,价格3000
中介： 我净赚：500
```

9、备忘录模式【Memento】

意图：

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

在备忘录模式中，如果要保存的状态多，可以创造一个备忘录管理者角色来管理备忘录。



适用性：

必须保存一个对象在某一个时刻的(部分)状态, 这样以后需要时它才能恢复到先前的状态。

如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

比如：

```
1、需要保存和恢复数据的相关状态场景。如保存游戏状态的场景：撤销场景，事务回滚等；
2、副本监控场景。备忘录可以当做一个临时的副本监控，实现非实时和准实时的监控。
```

代码：

```
# encoding: utf-8
'''
```

```
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 21、备忘录模式.py
@time: 2020/3/15 22:40
@author:LDC
'''
"""
备忘录模式
备忘录模式，在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，
这样以后就可将该对象恢复到原先保存的状态；
在备忘录模式中，如果要保存的状态多，可以创造一个备忘录管理者角色来管理备忘录。
比如保存游戏进度的功能
"""
import random


class Memento(object):
    # 定义一个备忘录类
    vitality = 0  # s生命值
    attack = 0  # 攻击
    defense = 0  # 防御

    def save_state(self, vitality, attack, defense):
        # 保存状态
        self.vitality = vitality
        self.attack = attack
        self.defense = defense
        return self


class GameCharacter(object):
    # 定义一个游戏角色抽象类
    vitality = 0  # s生命值
    attack = 0  # 攻击
    defense = 0  # 防御

    def display_state(self):
        # 显示状态
        print("\033[31;1m当前状态\033[0m")
        print("生命值：{}".format(self.vitality))
        print("攻击值：{}".format(self.attack))
        print("防守值：{}".format(self.defense))

    def init_state(self, vitality, attack, defense):
        # 初始化状态
        self.vitality = vitality
        self.attack = attack
        self.defense = defense

    def save_state(self, memento):
        # 保存状态
        print('\033[35;1m保存当前的状态到备忘录中\033[0m')
        return memento.save_state(self.vitality, self.attack, self.defense)

    def recover_state(self, memento):
        # 恢复状态
        print('\033[36;1m准备恢复备忘录中的状态\033[0m')
```

```
        self.vitality = memento.vitality
        self.attack = memento.attack
        self.defense = memento.defense

    def fight(self):
        # 攻击
        pass


class FightCharactor(GameCharacter):
    # 定义一个士兵
    def fight(self):
        self.vitality -= random.randint(1, 10)
        self.attack += random.randint(1, 10)
        self.defense -= random.randint(1, 10)


if __name__ == '__main__':
    memento = Memento()
    a_charactoer = FightCharactor()
    a_charactoer.init_state(100, 90, 80)
    a_charactoer.display_state()
    state = a_charactoer.save_state(memento)
    a_charactoer.fight()
    a_charactoer.display_state()
    a_charactoer.recover_state(state)
    a_charactoer.display_state()
"""
总结:
优点
使用备忘录可以把复杂的对象内部信息对其他的对象屏蔽起来。
缺点
当需要保存的状态数据很大很多时，会消耗较多资源。
"""

```

输出:

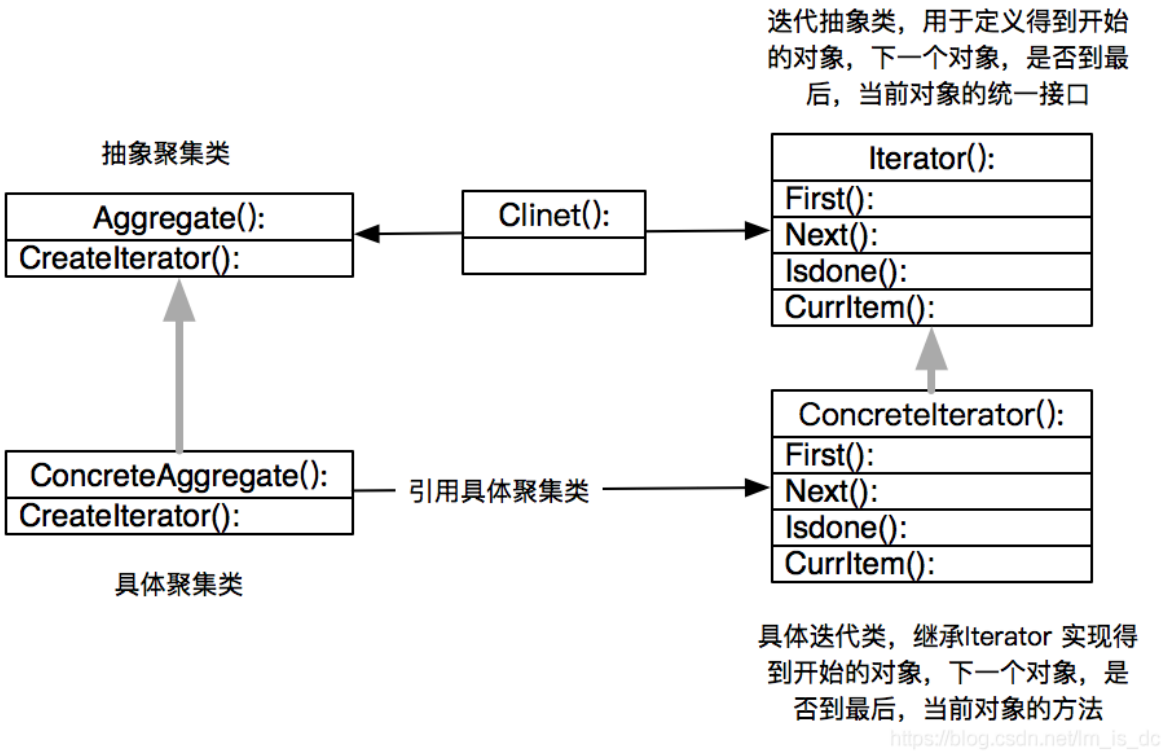
```
当前状态
生命值：100
攻击值：90
防守值：80
保存当前的状态到备忘录中
当前状态
生命值：90
攻击值：96
防守值：72
准备恢复备忘录中的状态
当前状态
生命值：100
攻击值：90
防守值：80

```

10、迭代器模式【Iterator】

意图：

提供一种方法顺序访问一个集合对象中各个元素, 而又不需暴露该对象的内部表示。

主要解决： 不同的方式来遍历整个集合对象。



适用性：

访问一个聚合对象的内容而无需暴露它的内部表示。

支持对聚合对象的多种遍历。

为遍历不同的聚合结构提供一个统一的接口(即, 支持多态迭代)。

比如：

遍历一个集合。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 22、迭代器模式.py
@time: 2020/3/15 23:25
@author:LDC
'''
"""
迭代器模式
迭代器模式(Iterator Pattern):提供方法顺序访问一个聚合对象中各元素，而又不暴露该对象的内部表示.
"""

```

```python

class Iterator(object):
    # 迭代器抽象类
    def first(self):
        # 第一个
        pass

    def next(self):
        # 下一个
        pass

    def is_done(self):
        # 是否完成
        pass

    def current_item(self):
        # 当前项目
        pass


class Aggregate(object):
    # 聚集抽象类
    def create_iterator(self):
        # 创建迭代器
        pass


class ConcreteIterator(Iterator):
    # 具体迭代器类，顺序
    def __init__(self, aggregate):
        self.aggregate = aggregate
        self.curr = 0

    def first(self):
        return self.aggregate[0]

    def next(self):
        ret = None
        self.curr += 1
        if self.curr < len(self.aggregate):
            ret = self.aggregate[self.curr]
        return ret

    def is_done(self):
        return True if self.curr + 1 >= len(self.aggregate) else False

    def current_item(self):
        return self.aggregate[self.curr]


class ConcreteAggregate(Aggregate):
    # 具体聚集类

    def __init__(self):
        self.i_list = []

    def create_iterator(self):
        return ConcreteIterator(self)
```

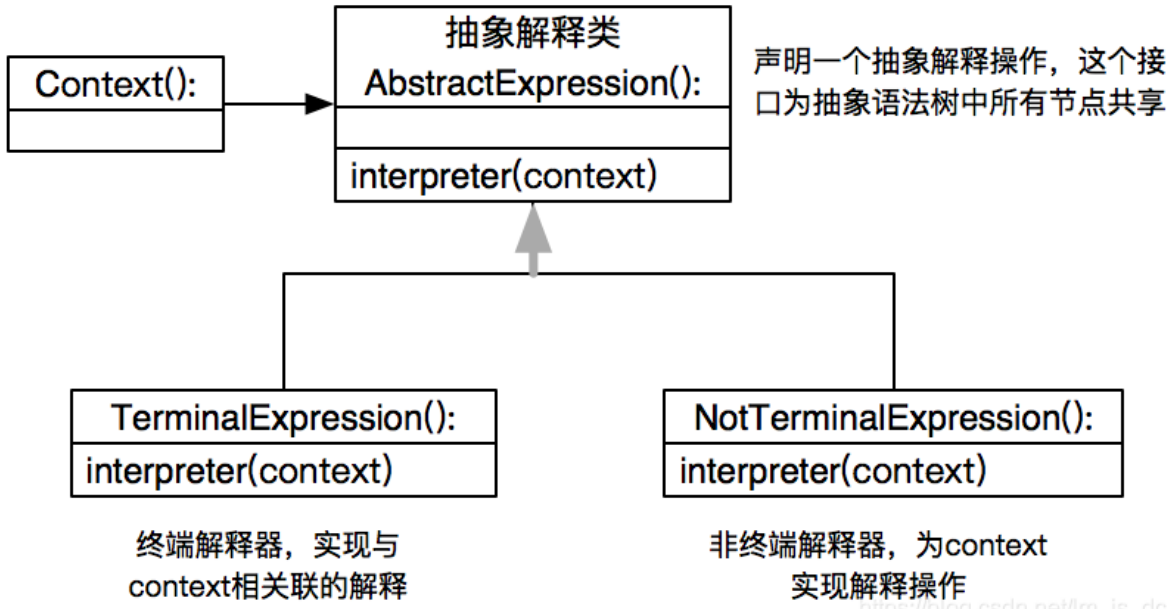
```python


class ConcreteIteratorDesc(Iterator):
    # 具体迭代器类，倒序
    def __init__(self, aggregate):
        self.aggregate = aggregate
        self.curr = len(aggregate) - 1

    def first(self):
        return self.aggregate[-1]

    def next(self):
        ret = None
        self.curr -= 1
        if self.curr >= 0:
            ret = self.aggregate[self.curr]
        return ret

    def is_done(self):
        return True if self.curr - 1 < 0 else False

    def current_item(self):
        return self.aggregate[self.curr]


if __name__ == '__main__':
    ca = ConcreteAggregate()  # 创建具体聚集
    ca.i_list.append("房子")
    ca.i_list.append("沙发")
    ca.i_list.append("衣柜")
    ca.i_list.append("床")

    itor = ConcreteIterator(ca.i_list)  # 创建一个迭代器
    print(itor.first())
    while not itor.is_done():
        print(itor.next())

    print('**********倒序**********')
    itor_desc = ConcreteIteratorDesc(ca.i_list)  # 创建一个迭代器
    print(itor_desc.first())
    while not itor_desc.is_done():
        print(itor_desc.next())


"""
总结:
当需要对聚集对象有多种方式遍历时，可以考虑使用迭代器模式
迭代器模式分离了集合的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合内部结构，
又可以让外部代码透明的访问集合内部的数据

优点： 1、它支持以不同的方式遍历一个聚合对象。
2、迭代器简化了聚合类。
3、在同一个聚合上可以有多个遍历。
4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，
```

```
类的个数成对增加，这在一定程度上增加了系统的复杂性。
"""
```

11、解释器模式【Interpreter】

意图：

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。



适用性：

当有一个语言需要解释执行，并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：

该文法简单。对于复杂的文法，文法的类层次变得庞大而无法管理。此时使用语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式，这样可以节省空间而且还可能节省时间。

效率不是一个关键问题。最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下，转换器仍可用解释器模式实现，该模式仍是有用的。

比如：

对于一些固定文法构建一个解释句子的解释器。

编译器、运算表达式计算。

代码：

```
# encoding: utf-8
'''
@contact: 1257309054@qq.com
@wechat: 1257309054
@Software: PyCharm
@file: 23、解释器模式.py
```

```
@time: 2020/3/15 23:59
@author:LDC
'''
import datetime
import time

"""
解释器模式
解释器模式(Interpreter Pattern):给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子.
比如：实现一段简单的中文编程
"""


class Code(object):
    # 自定义语言

    def __init__(self, text=None):
        self.text = text


class InterpreterBase(object):
    # 自定义计算器基类
    def run(self, code):
        pass


class Interpreter(InterpreterBase):
    # 实现解释器方法，通过表达式字典实现中文编程
    def run(self, code):
        code = code.text
        code_dict = {'获取当前时间戳': time.time(),
                     "获取当前日期": datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                     }
        print(code_dict.get(code))


if __name__ == '__main__':
    zw = Code()
    interpreter = Interpreter()
    zw.text = "获取当前时间戳"
    interpreter.run(zw)
    zw.text = "获取当前日期"
    interpreter.run(zw)


"""
总结：
优点：
1、可扩展性比较好，灵活。
2、增加了新的解释表达式的方式。
3、易于实现简单文法。

缺点：
1、可利用场景比较少。
2、对于复杂的文法比较难维护。
3、解释器模式会引起类膨胀。
```

```
63 """
64
```

输出：

```
1584289349.8031545
2020-03-16 00:22:29
```

后记

【后记】为了让大家能够轻松学编程，我创建了一个公众号【轻松学编程】，里面有让你快速学会编程的文章，当然也有一些干货提高你的编程水平，也有一些编程项目适合做一些课程设计等课题。

也可加我微信【1257309054】，拉你进群，大家一起交流学习。 如果文章对您有帮助，请我喝杯咖啡吧！

https://www.cnblogs.com/Sam-2018/p/principle.html

https://www.jianshu.com/p/4cc96686abb8

https://blog.csdn.net/qq_21467113/article/details/89480740

https://blog.csdn.net/weixin_30692143/article/details/96118986?depth_1-utm_source=distribute.pc_relevant.none-task&utm_source=distribute.pc_relevant.none-task

https://blog.csdn.net/burgess_zheng/article/details/86762248#%C2%A0%20%C2%A0%20%C2%A0%20%C2%A0%201.%E5%B7%A5%E5%8E%82%E6%96%B9%E6%B3%95%E6%A8%A1%E5%BC%8F%EF%BC%88Factory%20Method%EF%BC%89 https://blog.csdn.net/burgess_zheng/article/details/86762248

https://www.cnblogs.com/ppap/p/11103324.html

https://www.cnblogs.com/littlefivebolg/p/9927328.html

https://www.cnblogs.com/welan/p/9127000.html

https://blog.csdn.net/hbu_pig/article/details/80807322

https://www.cnblogs.com/mrwuzs/p/12026311.html

https://www.cnblogs.com/baxianhua/p/10904122.html

https://www.cnblogs.com/qq_841161825/articles/10144601.html

https://www.cnblogs.com/onepiece-andy/p/python-iterator-pattern.html

https://www.cnblogs.com/onepiece-andy/p/python-interpreter-pattern.html