**设计模式**

<https://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html>

软件开发不变的真理：

CHANGE, 不论你自认为， 你设计的多么好，想的有多远， 但是经过时间的验证，你总是要更改你的代码，去做业务上的适应。你更改的过程中，会遇到很多问题， 而你遇到的问题是 有些人已经解决的了，那么就可以共享这些经验，建立弹性的设计，易于维护，应对多变的需求。

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。

以往是代码的复用，现在是经验的复用。我们现在要做的就是把这些模式装进我们脑子里，尽量用到我们的工作中。

**设计模式的分类**

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

**设计模式的好处：**

1. 经验复用， 看的远不是你非得视力多好， 站在别人肩膀上也未尝不可。

2. 共享词汇（装）

用一个图片来整体描述一下：



## 策略模式（**strategy**）

**概念**：策略模式是指对一系列的[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)定义，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

（独立，我爱咋改咋改，对你没有影响）

**组成**

a. 策略类，通常由一个接口或者抽象类实现。

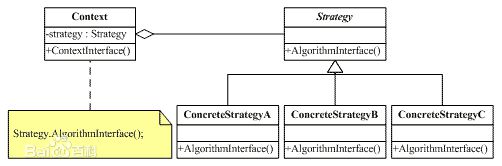
b. 具体策略角色：包装了相关的算法和行为。

c. 环境角色：持有一个策略类的引用，最终给客户端调用。

继承（extend,IS-A）

实现(implement)

有一个(IS-A)



在策略模式的设计中我们会提到这样几个概念： **继承，多态，接口，抽象类**

面向对象的三大特性：**封装、继承、多态**

封装：

1.：隐藏对象的属性和实定义现细节，仅对外公开接口,控制在程序中属性的读和修改的访问级别。首先是抽象，把事物抽象成一个类，其次才是封装，将事物拥有的属性和动作隐藏起来，只保留特定的方法与外界联系

2.封装的目的是：增强安全性和简化编程.封装符合面向对象设计原则的第一条：单一性原则，一个类把自己该做的事情封装起来，而不是暴露给其他类去处理，当内部的逻辑发生变化时，外部调用不用因此而修改，他们只调用开放的(一特定的访问权限)接口，而不用去关心内部的实现

**继承：**

1.目的：实现代码的复用。

2.介绍：当两个类具有相同的特征（属性）和行为（方法）时，可以将相同的部分抽取出来放到一个类中作为父类，其它两个类继承这个父类

**多态 :**

1.概念：相同的事物，调用其相同的方法，参数也相同时，但表现的行为却不同。

2.Java实现多态有三个必要条件：继承、重写、向上转型。

3.多态的实现方式：

（1）基于继承实现的多态（2）基于接口实现的多态

**原则：**

1.会变化的独立出来，不要和经常变化的混在一起。

2.多用组合，少用继承。 （方法/函数不是继承来的，而是和适当的行为对象“组合”来的）

**优缺点**

**优点**

1、 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码转移到父类里面，从而避免重复的代码。

2、 策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为。如果不是用策略模式，那么使用算法或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。但是，这样一来算法或行为的使用者就和算法或行为本身混在一起。决定使用哪一种算法或采取哪一种行为的逻辑就和算法或行为的逻辑混合在一起，从而不可能再独立演化。继承使得动态改变算法或行为变得不可能。

3、 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。

**缺点：**

1、客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。

2、 策略模式造成很多的策略类，每个具体策略类都会产生一个新类。有时候可以通过把依赖于环境的状态保存到客户端里面，而将策略类设计成可共享的，这样策略类实例可以被不同客户端使用。换言之，可以使用享元模式来减少对象的数量。

**设计模式的六大原则**

**1、开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：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的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

为什么叫最少知道原则，就是说：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、合成复用原则（Composite Reuse Principle）**

原则是尽量使用合成/聚合的方式，而不是使用继承。

## 观察者模式（Observer）

1定义：在对象之间定义一对多的依赖，这样一来， 当一个对象改变状态，依赖他的对象都会收到通知，并自动更新 。

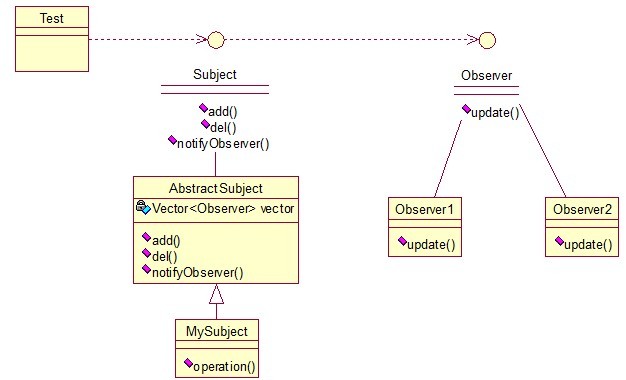
（有时又被称为发布（[publish](https://baike.baidu.com/item/publish) ）-订阅（Subscribe）模式、模型-视图（View）模式、源-收听者(Listener)模式或从属者模式）

2理解 ：当我们试图勾勒出观察者模式的时候 ， 可以利用报纸订阅服务以及出版者和订阅者比拟这一切。

出版者 + 订阅者 = 观察者模式

主题（Subject）+ 观察者（Observer）

对象之间是一种一对多的关系。先来看看关系图：



3 实现：实现的方式不只一种， Subject + Observer的方式最为常见。

4 好处：主题是真正拥有数据的人，观察者是其依赖者，这样数据变化由主题控制，这比许多对象控制同一份数据更好，更干净。

5 设计原则 ： 松耦合的设计，主题和观察者之间松耦合，彼此不清楚彼此

**设计原则：**

找出程序中变化的部分，然后将其和固定不变的方面分离

在观察者模式中，会改变的是主题的状态，以及观察者的数目和类型。

**设计原则：**

针对接口编程，

不针对实现编程

主题与观察者都使用接口，观察者利用主题的接口向主题注册，主题用观察者接口通知观察者。两者间运作正常，同时具有松耦合的优点

**设计原则：**

多用组合少用继承

观察者利用“组合”将许多观察者组合进主题中，对之间的这种关系不是继承产生的，而是在运行时，利用组合产生的。

桐

6 java 内置的对象

java.util.Observable && java.util.Observer

javaBeans

Swing

7观察者模式的效果有以下的优点：

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知，

观察者模式有下面的缺点：

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

8 观察者模式的应用场景：   
1、 对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。   
2、 对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节。

9 回顾：

那些基本概念

回调函数只是一个功能片段，由用户按照回调函数调用约定来实现的一个函数。

**就是程序员A写了一段程序（程序a），其中预留有回调函数接口，并封装好了该程序。程序员B要让a调用自己的程序b中的一个方法，于是，他通过a中的接口回调自己b中的方法。**

**public class A {**

private CallBack callback; //注册一个事件, **其中预留有回调函数接口**

public void register(CallBack callback){

this.callback = callback;

}

public void call(){

// 需要调用的时候回调**vo**

callback.oncall();//**通过a中的接口回调自己b中的方法(oncall)**

}

}

public interface CallBack {

public void oncall();

}

public static void main(String[] args) {

A a = new A();

a.register(new CallBack() {

@Override

public void oncall() {

System.out.println("回调函数被调用");

}

});

a.call();

}

这里在引入 观察者模式的代码

public class Subject {

List<Observer> lists = new ArrayList<Observer>();

//注册一个事件

public void register(Observer observer){

lists.add(observer);

}

public void \_notify(){

for (Observer observer : lists) {

observer.update();

}

}

public void unRegister(Observer observer){

lists.remove(observer);

}

}

public interface Observer {

public void update();

}

public class ConcreteObserver1 implements Observer{

public void update() {

System.out.println("ConcreteObserver1获得更新");

}

}

public class ConcreteObserver2 implements Observer{

public void update() {

System.out.println("ConcreteObserver2获得更新");

}

}

public static void main(String[] args) {

Observer observer1 = new ConcreteObserver1();

Observer observer2 = new ConcreteObserver2();

Subject subject = new Subject();

subject.register(observer1);

subject.register(observer2);

subject.\_notify();

//取消观察者1的注册

subject.unRegister(observer1);

subject.\_notify();

}

目标对象保存了各个观察者的引用，当需要通知时发送通知。

实际上，回调是观察者模式的简单形式。观察者模式也就是将回调上升到设计模式的理论高度上了而已。

将回调例子中的main方法改下：

public static void main(String[] args) {

CallBack callback = new CallBackImp();

A a = new A();

a.register(callback);

a.call();

}

public class CallBackImp implements CallBack{

@Override

public void oncall() {

System.out.println("回调函数被调用");

}

}

观察者模式里面目标类维护了所有观察者的引用，而回调里面只是维护了一个引用。

**什么是事件监听器https://www.cnblogs.com/EasonJim/p/7101203.html**

监听器将监听自己感兴趣的事件一旦该事件被触发或改变，立即得到通知，做出响应。例如：Android程序中的Button事件。

Java的事件监听机制可概括为3点：

1. Java的事件监听机制涉及到**事件源，事件监听器，事件对象**三个组件,监听器一般是接口，用来约定调用方式。
2. 当事件源对象上发生操作时，它将会调用事件监听器的一个方法，并在调用该方法时传递事件对象过去。
3. 事件监听器实现类,通常是由开发人员编写，开发人员通过事件对象拿到事件源，从而对事件源上的操作进行处理。

**举个例子**

这里我为了方便，直接使用JDK，EventListener 监听器，感兴趣的可以去研究下源码，非常简单。

**监听器接口**

public interface EventListener extends java.util.EventListener {

//事件处理

public void handleEvent(EventObject event);

}

**事件对象**

[复制代码](javascript:void(0);)

public class EventObject extends java.util.EventObject{

private static final long serialVersionUID = 1L;

public EventObject(Object source){

super(source);

}

public void doEvent(){

System.out.println("通知一个事件源 source :"+ this.getSource());

}

}

[复制代码](javascript:void(0);)

**事件源**

事件源是事件对象的入口，包含监听器的注册、撤销、通知

[复制代码](javascript:void(0);)

public class EventSource {

//监听器列表，监听器的注册则加入此列表

private Vector<EventListener> ListenerList = new Vector<EventListener>();

//注册监听器

public void addListener(EventListener eventListener){

ListenerList.add(eventListener);

}

//撤销注册

public void removeListener(EventListener eventListener){

ListenerList.remove(eventListener);

}

//接受外部事件

public void notifyListenerEvents(EventObject event){

for(EventListener eventListener:ListenerList){

eventListener.handleEvent(event);

}

}

}

[复制代码](javascript:void(0);)

**测试执行**

[复制代码](javascript:void(0);)

public static void main(String[] args) {

EventSource eventSource = new EventSource();

eventSource.addListener(new EventListener(){

@Override

public void handleEvent(EventObject event) {

event.doEvent();

if(event.getSource().equals("closeWindows")){

System.out.println("doClose");

}

}

});

/\*

\* 传入openWindows事件，通知listener，事件监听器，

对open事件感兴趣的listener将会执行

\*\*/

eventSource.notifyListenerEvents(new EventObject("openWindows"));

}

[复制代码](javascript:void(0);)

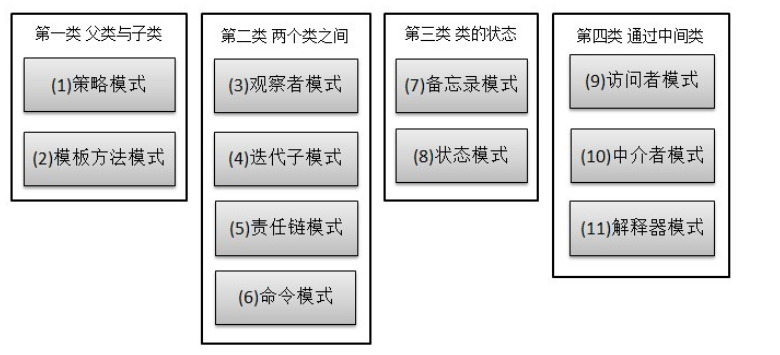
控制台显示：

通知一个事件源 source :openWindows

通知一个事件源 source :openWindows

doOpen something...

到这里你应该非常清楚的了解，什么是事件监听器模式了吧。 那么哪里是回调接口，哪里是回调者，对！EventListener是一个回调接口类，handleEvent是一个回调函数接口，通过回调模型，EventSource 事件源便可回调具体监听器动作。



## 装饰者 （Decorator Pattern）

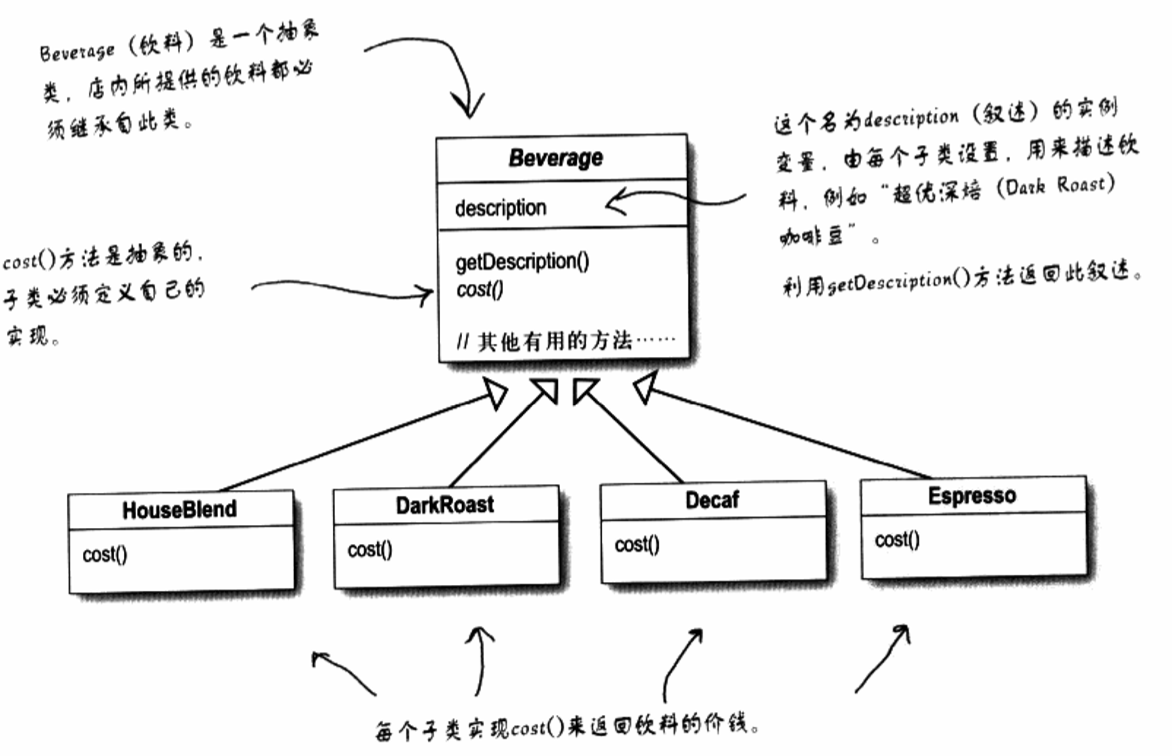
1定义：动态的将责任附加到对象上。想要扩展功能，装饰者提供有别于继承的另一种选择。

2 背景实例 ：

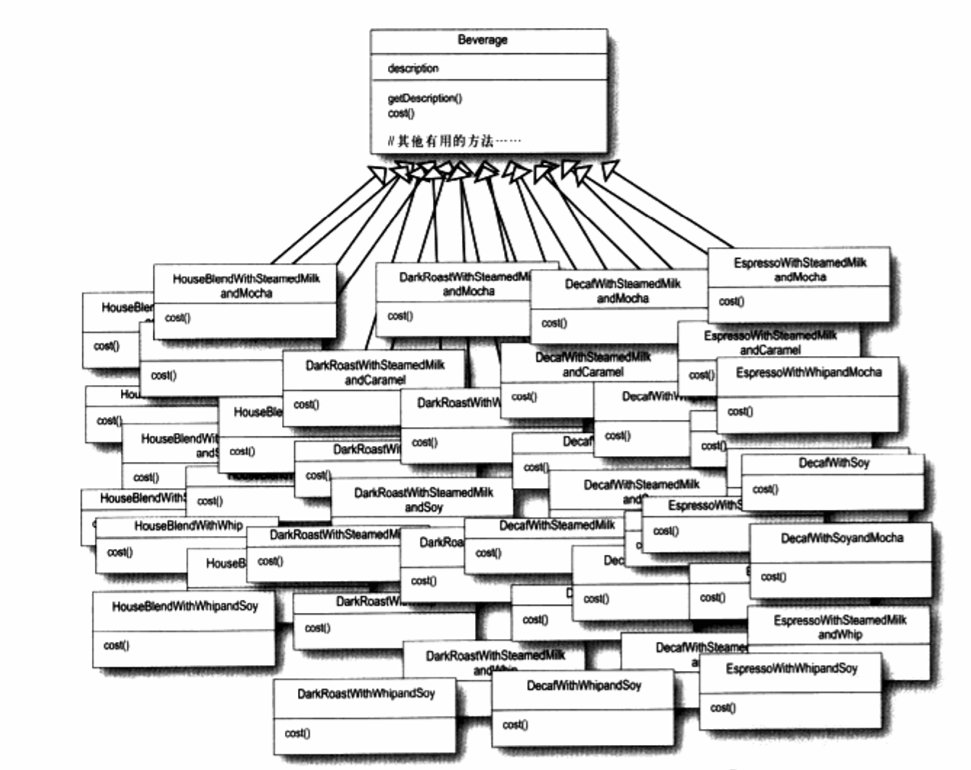
在星巴克喝咖啡时, 可以选择各种调味品, 黑咖啡中可以选择加糖,加奶,加蜂蜜. 这样咖啡的味道被改变,我们可以说黑咖啡被"增强"了。

1. /\*\*
2. \* 创建一个咖啡的类,里面有黑咖啡
3. \*/
4. class Coffee{
5. void blankCoffee(){
6. System.out.println("黑咖啡");
7. }
8. }
9. /\*\*
10. \* 现在要对黑咖啡加糖
11. \*/
12. class Sugar extends Coffee {
13. void blankCoffee() {
14. super.blankCoffee();
15. System.out.println("加糖");
16. }
17. }
18. /\*\*
19. \* 现在要对黑咖啡加奶
20. \*/
21. class Milk extends Coffee {
22. void blankCoffee() {
23. super.blankCoffee();
24. System.out.println("加奶");
25. }
26. }
28. /\*\*
29. \* 现在要对黑咖啡加蜂蜜
30. \*/
31. class Honey extends Coffee {
32. void blankCoffee() {
33. super.blankCoffee();
34. System.out.println("加蜂蜜");
35. }
36. }

这里填一张图 ，表述最初类的继承关系：

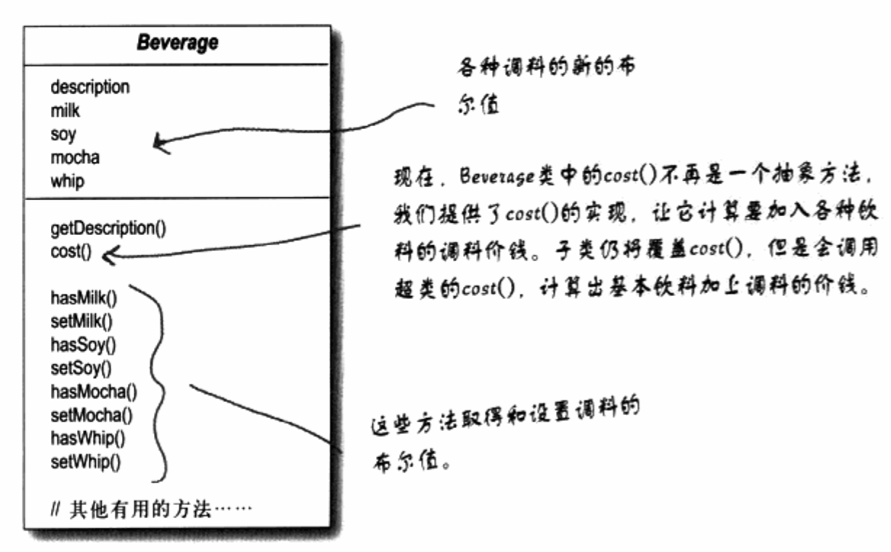


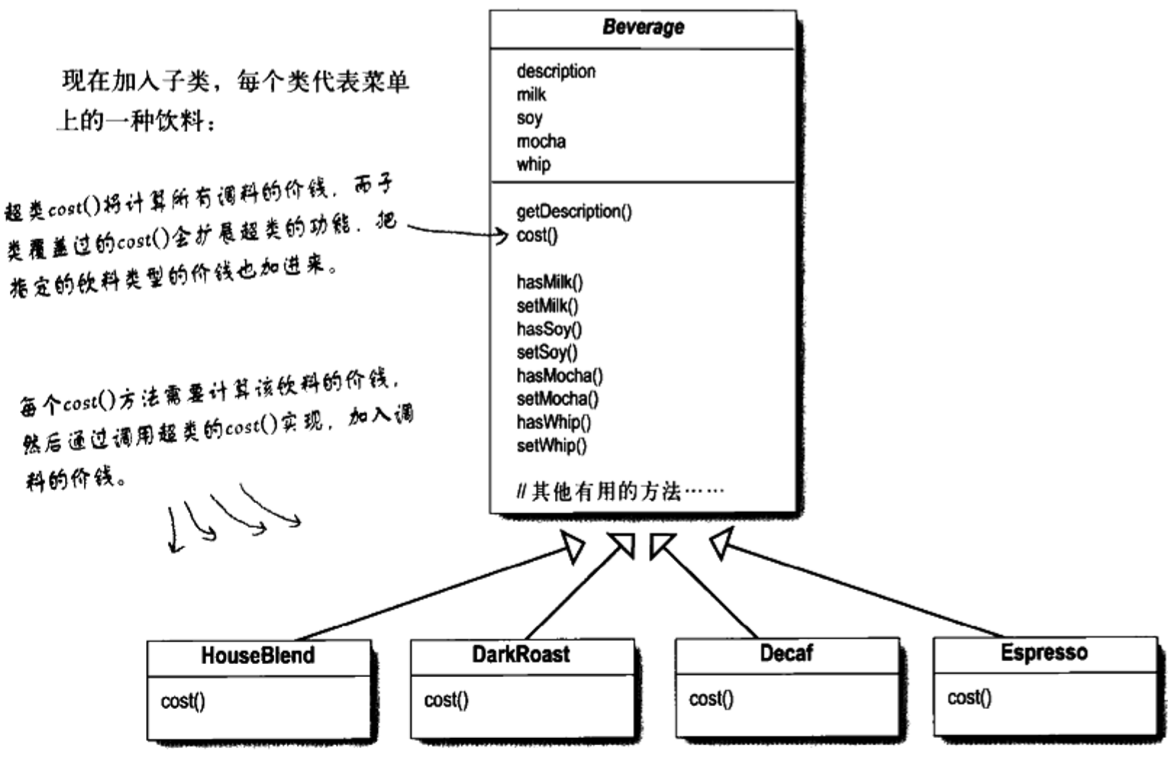
这里再放一张图，描述类的爆炸



症结： **类的爆炸，难以维护，基类加入新功能不适用于所有子类。**

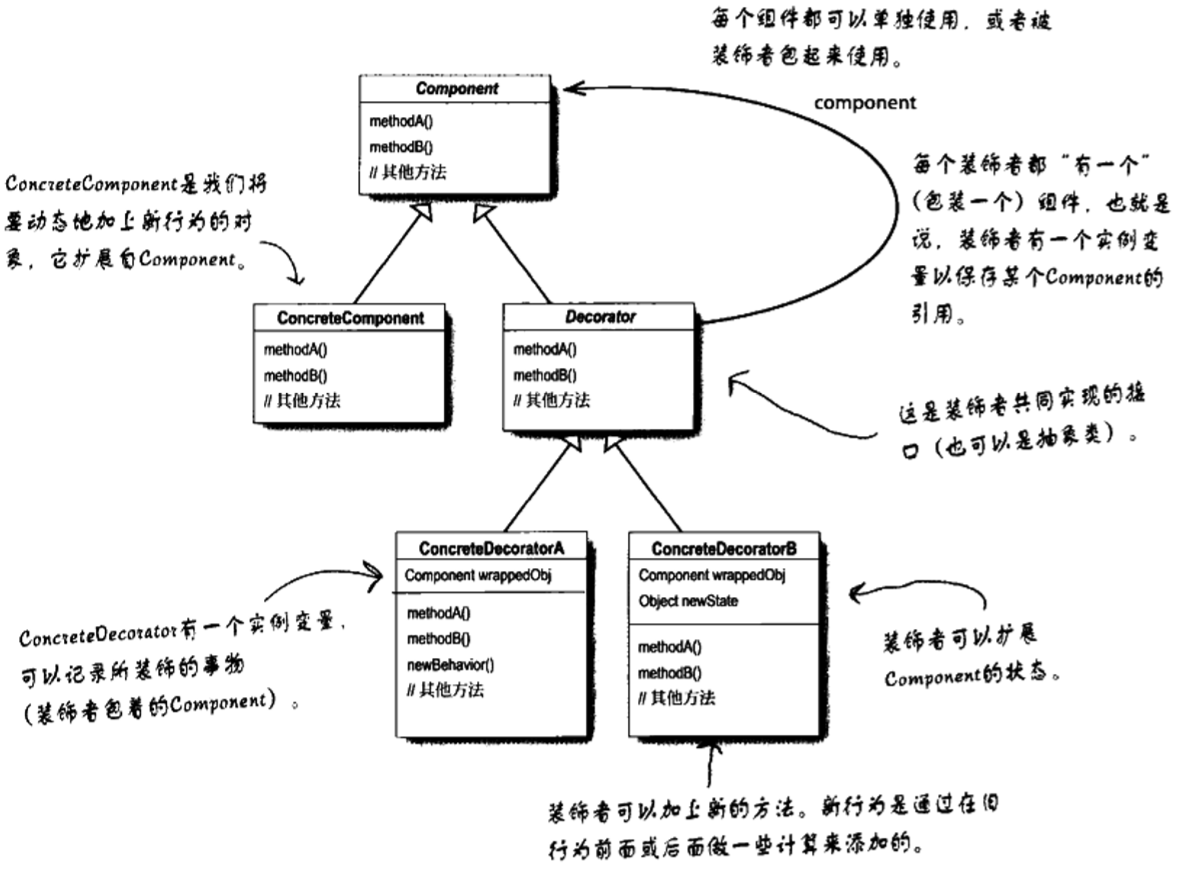
抽象出来方法 ：





症结： **一旦出现新的 调料，我们就要更改代码， 违反了 “类应该对扩展开放，对修改关闭”**

我们可以看到，尽管继承 很好，但是并不能总是实现最有弹性的设计。



利用继承那么类的行为，是在编译时静态决定的，不是来自超类行为就是子类覆盖过后的版本， 利用组合可以在运行时候动态扩展行为，可以把装饰者混合使用。运行时的扩展远比编译时候的继承威力大。

适用性

以下情况使用Decorator模式

1. 需要扩展一个类的功能，或给一个类添加附加职责。

2. 需要动态的给一个对象添加功能，这些功能可以再动态的撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

3. 需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变的不现实。

4. 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。

装饰模式优点

1. Decorator模式与继承关系的目的都是要扩展对象的功能，但是Decorator可以提供比继承更多的灵活性。

2. 通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

装饰模式缺点

1. 这种比继承更加灵活机动的特性，也同时意味着更加多的复杂性。

2. 装饰模式会导致设计中出现许多小类，如果过度使用，会使程序变得很复杂。产生过多相似的对象，不易排错！

3. 装饰模式是针对抽象组件（Component）类型编程。但是，如果你要针对具体组件编程时，就应该重新思考你的应用架构，以及装饰者是否合适。当然也可以改变Component接口，增加新的公开的行为，实现“半透明”的装饰者模式。在实际项目中要做出最佳选择。

装饰模式设计原则

1. 多用组合，少用继承。

利用继承设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为。然而，如果能够利用组合的做法扩展对象的行为，就可以在运行时动态地进行扩展。

2. 类应设计的对扩展开放，对修改关闭。

## 工厂方法模式 **（Factory Method）**

1定义：定义了一个 创建对象的接口， 但由子类决定要实例化的类是哪一个。 工厂方法让类把实例化推迟到子类中。(推迟，说白了父类定义的是抽象方法 子类是具体的实现)

工厂方法 ： abstrasct Product factoryMethod(String type)

创建对象的接口—接口 不要局限于只能是 interface修饰的接口类， 我们说接口其实就是一种抽象， 此地就是那个抽象方法 即是说 工厂方法。

## 抽象工厂模式 **（Abstract Factory Method）**

1定义：提供一个接口， 用于创建相关或依赖对象的家族，而不需要明确指定具体的类

## 简单工厂 –这不是一个设计模式，只是一个好习惯

场景：

1. 创建对象 new
2. 需要创建一组对象

显而易见的我们也写过这样的 code , 集中一个 地方去实例化我们的代码。Store

把这个代码抽象出来我们都能做得到。放到 一个类里面来管理 我们也做的到，这个习惯其实就是简单工厂的 示例 。

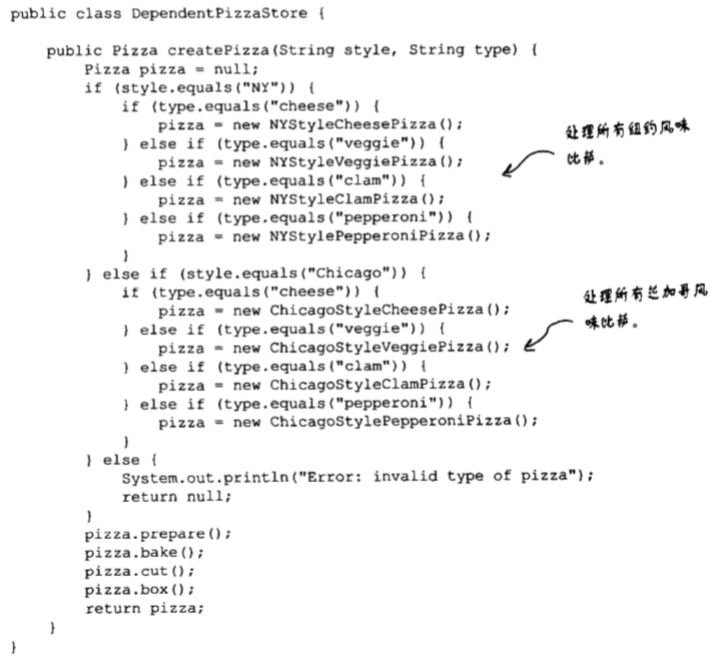
简单工厂：简单工厂模式不是23种里的一种，简而言之，就是有一个专门生产某个产品的类。

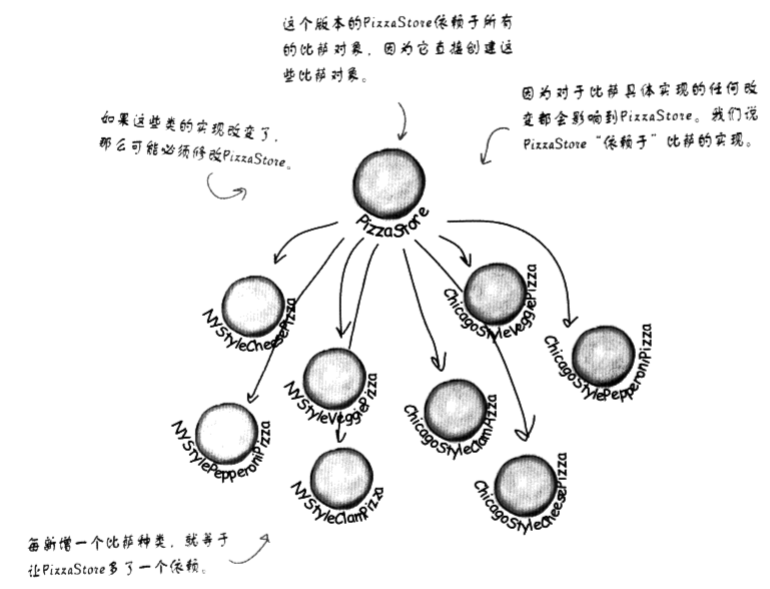
此处引入代码 🡪 MouseFactory

好处 ： 你们说吧

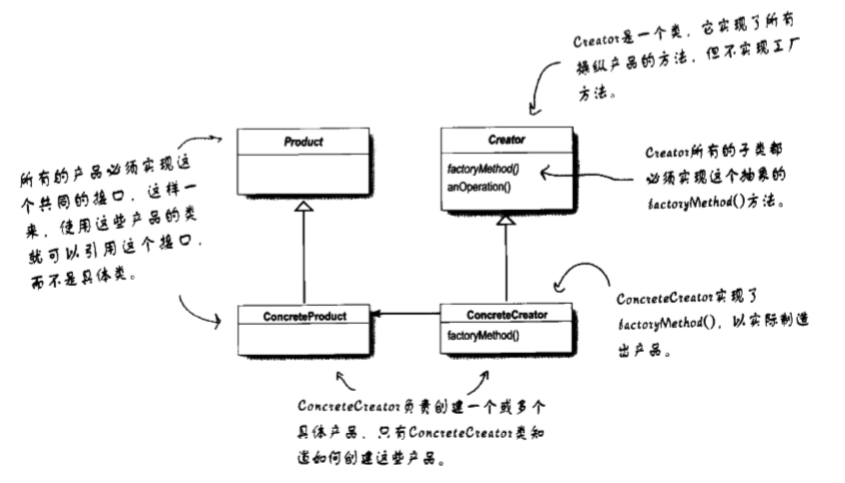
坏处 ：

1. Open-Closed Principle, 比较明显，在进行扩展的时候，我们要更改工厂函数里面的那个分支语句，这样便破坏了OCP，
2. 高度依赖

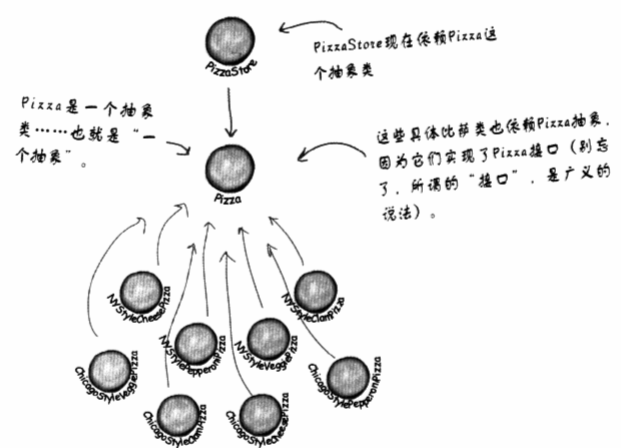




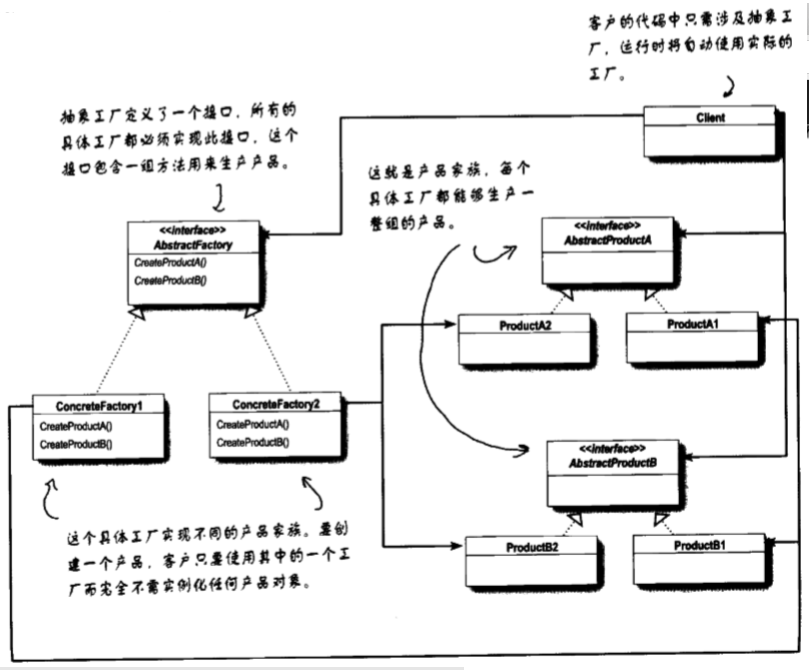
工厂模式



这里有一个依赖倒置的概念：



抽象工厂模式 ：



简单工厂把全部的事情放在一个地方都处理完了， 而工厂方法是创建一个框架， 让子类决定要如何实现

**进化**：

            优化满足OCP        ；        优化满足一族产品

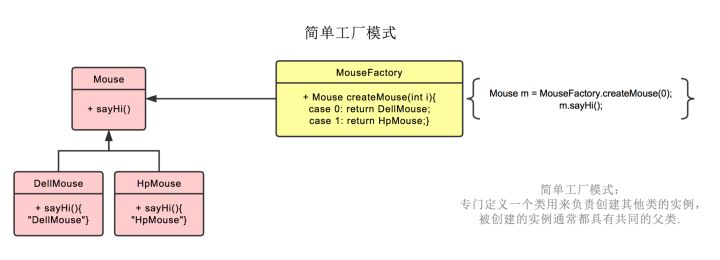
简单工厂 --------------------------->  工厂方法  --------------------------------> 抽象工厂

**退化**：

    增加了和客户端之间的耦合度（客户端需要知道各种工厂）；变得太重了

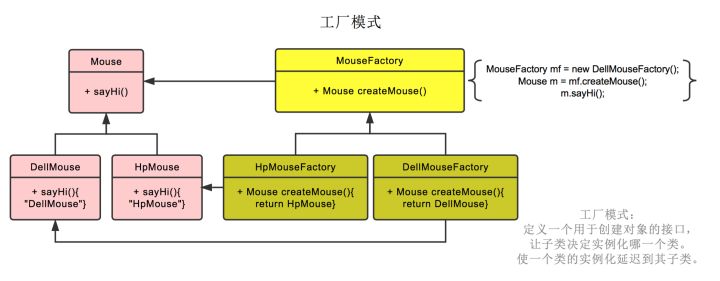
简单工厂 --------------------------->  工厂方法  --------------------------------> 抽象工厂

**简单工厂模式**

他不是23种里的一种，简而言之，就是有一个专门生产某个产品的类。   
比如下图中的鼠标工厂，专业生产鼠标，给参数0，生产戴尔鼠标，给参数1，生产惠普鼠标。 

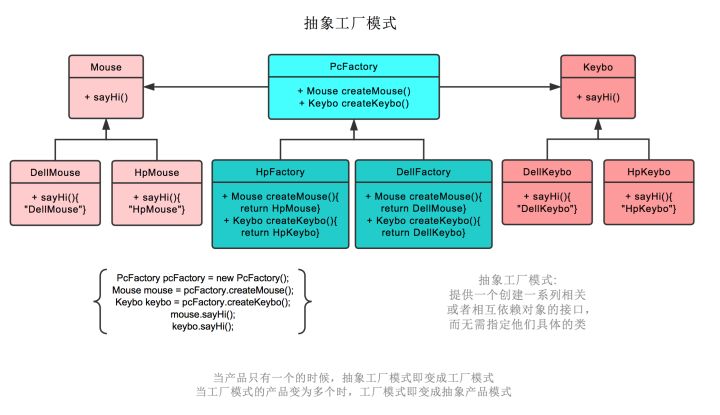
**工厂模式**

也就是鼠标工厂是个父类，有生产鼠标这个接口。   
戴尔鼠标工厂，惠普鼠标工厂继承它，可以分别生产戴尔鼠标，惠普鼠标。   
生产哪种鼠标不再由参数决定，而是创建鼠标工厂时，由戴尔鼠标工厂创建。   
后续直接调用鼠标工厂.生产鼠标()即可



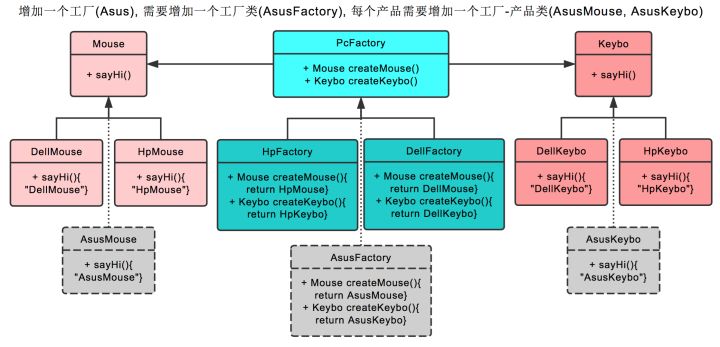
**抽象工厂模式**

也就是不仅生产鼠标，同时生产键盘。   
也就是PC厂商是个父类，有生产鼠标，生产键盘两个接口。   
戴尔工厂，惠普工厂继承它，可以分别生产戴尔鼠标+戴尔键盘，和惠普鼠标+惠普键盘。   
创建工厂时，由戴尔工厂创建。   
后续工厂.生产鼠标()则生产戴尔鼠标，工厂.生产键盘()则生产戴尔键盘



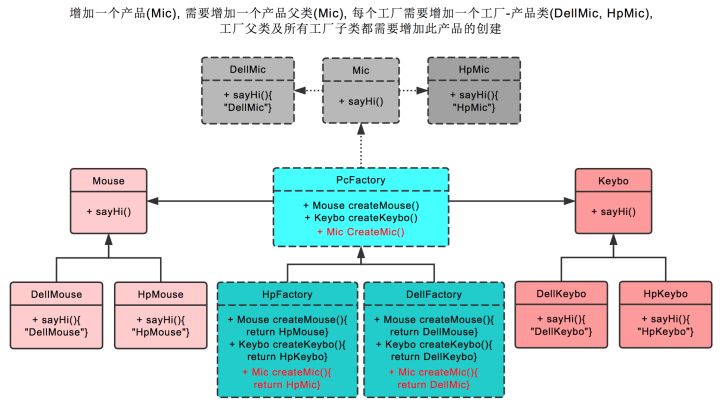
***在抽象工厂模式中，假设我们需要增加一个工厂***

假设我们增加华硕工厂，则我们需要增加华硕工厂，和戴尔工厂一样，继承PC厂商。   
之后创建华硕鼠标，继承鼠标类。创建华硕键盘，继承键盘类。   
即可。



***在抽象工厂模式中，假设我们需要增加一个产品***

假设我们增加耳麦这个产品，则首先我们需要增加耳麦这个父类，再加上戴尔耳麦，惠普耳麦这两个子类。   
之后在PC厂商这个父类中，增加生产耳麦的接口。最后在戴尔工厂，惠普工厂这两个类中，分别实现生产戴尔耳麦，惠普耳麦的功能。



https://www.zhihu.com/question/20367734