# java设计模式总结

# 1.设计模式的六大原则

**1.单一职责原则**

一个类一个业务，一个方法一个功能。各自有别各自管好自己。

1. **接口隔离原则**

接口和接口之间要业务分类

1. **依赖倒转**

细节依赖抽象，面向接口编程

1. **里氏替换原则、**合成复用原则

多用聚合或者组合

1. **开闭原则**

扩展开放(对提供方)， 对修改关闭(对使用方)

1. **迪米特法则**

最少知道原则

**聚合关系（Aggregation）：**表示的是整体和部分的关系，整体与部分可以分开。 聚合关系是关联关系的特例，所以他具有关联的导航性与多重性。

**组合关系：**也是整体与部分的关系，但是整体与部分不可以分开。

比如A类中包含B类的一个引用b，当A类的一个对象消亡时，b这个引用所指向的对象也同时消亡（没有任何一个引用指向它，成了垃圾对象），这种情况叫做组合，反之b所指向的对象还会有另外的引用指向它，这种情况叫聚合。  
在实际写代码时组合方式一般会这样写：  
A类的构造方法里创建B类的对象，也就是说，当A类的一个对象产生时，B类的对象随之产生，当A类的这个对象消亡时，它所包含的B类的对象也随之消亡。  
聚合方式则是这样：  
A类的对象在创建时不会立即创建B类的对象，而是等待一个外界的对象传给它  
传给它的这个对象不是A类创建的。

# 单例模式

保证整个系统中，对某个类来讲只有一个对象实列

饿汉式：类加载的时候就实例化了 缺点类加载时候就生成可能会造成资源浪费

懒汉式：用到的时候才加载 会出现线程问题

双重检查：解决懒汉式线程问题

静态内部类：比较好的懒汉式方法

枚举：比较好的饿汉式方式

这个需要会写

jdk的runtime源码用到单例模式

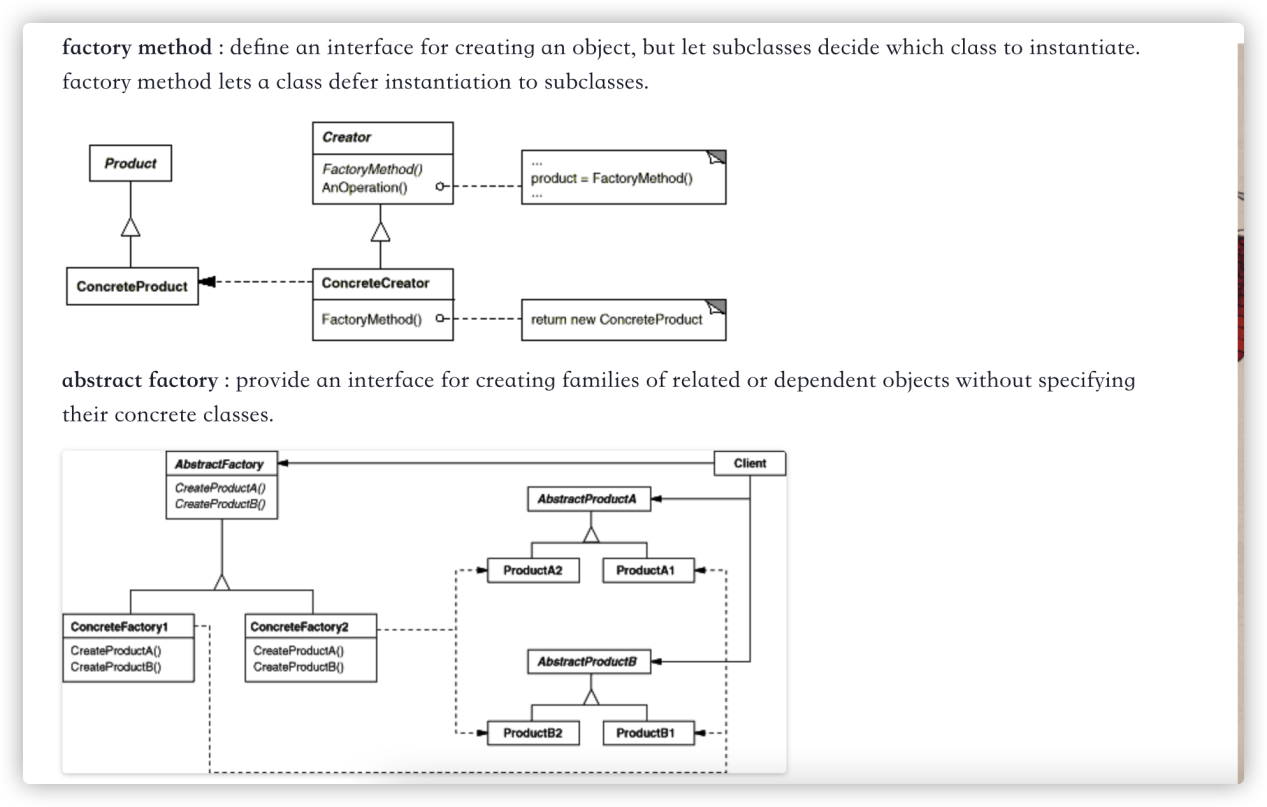
# 工厂模式

简单工厂：不同的产品，差不多的实现步骤，提出一个工厂来统一管理做生产

工厂模式：一个产品对应多个具体产品；一个工厂对应多个具体的工厂

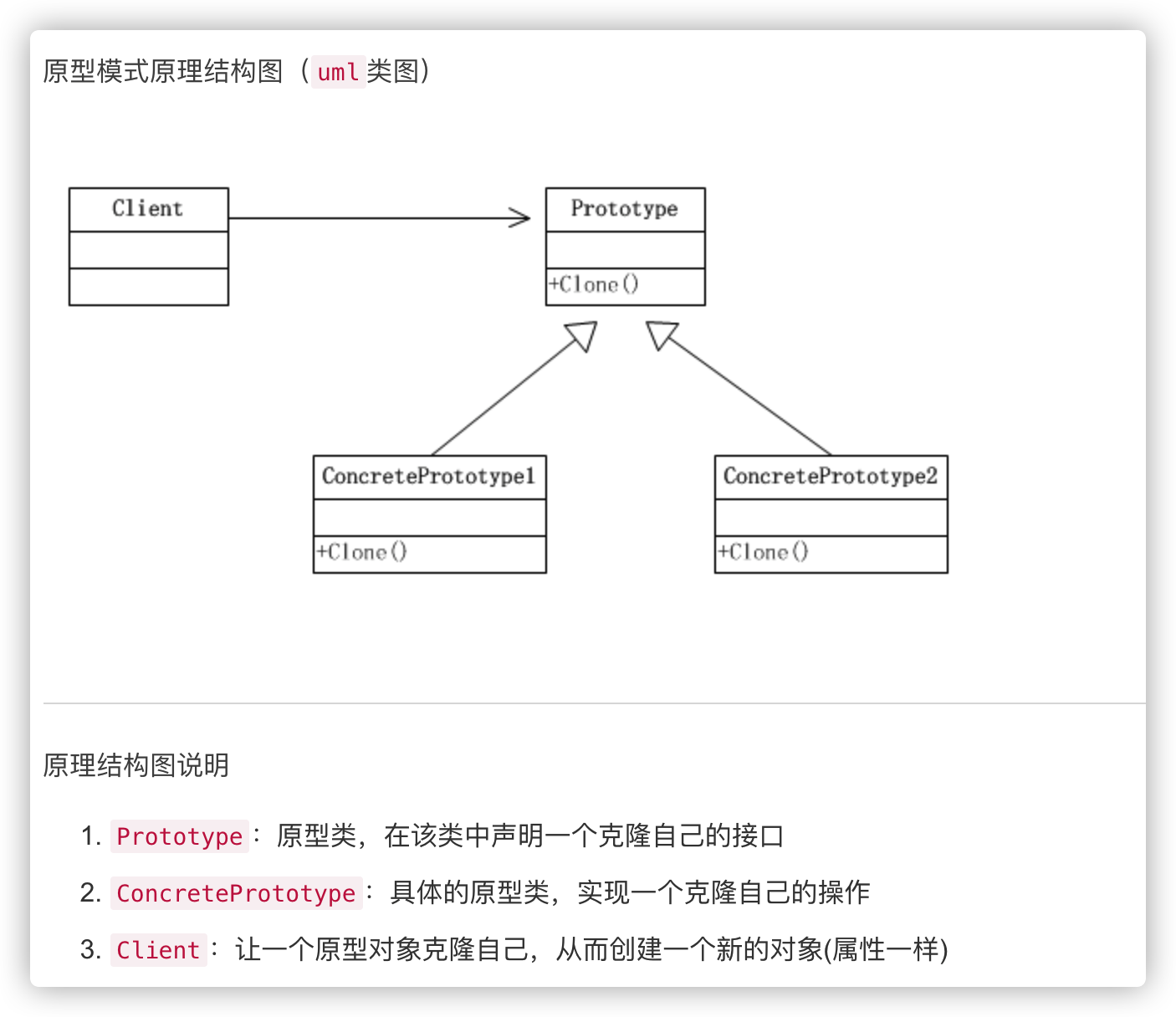
抽象工厂：对工厂再次抽象，有一个大的抽象工厂管理不同的工厂类，工厂类下面再是具体的工厂

JDK Calendar 中使用到了简单工厂模式



# 原形模式

克隆羊的案例



我们需要复制一个对象的时候使用源对象实现clonable接口

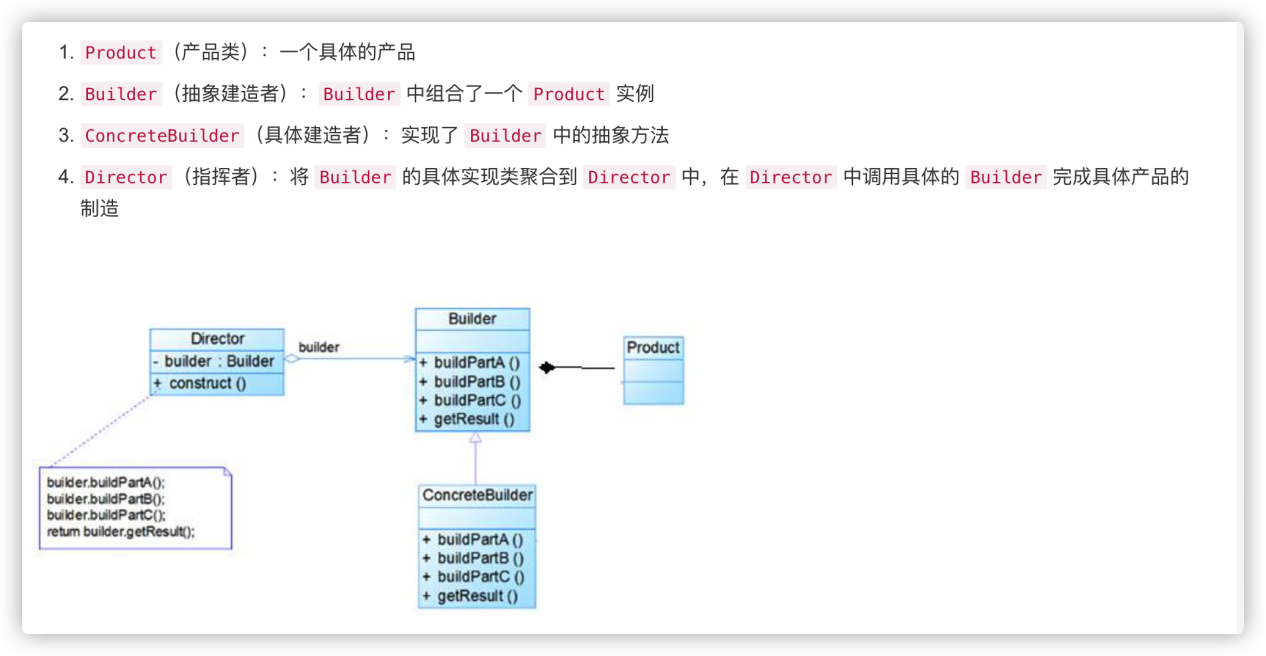
浅拷贝：拷贝数值、对象引用

深拷贝：拷贝数值、具体对象的内容，应用类型是重新生成空间

spring有用到

# 建造者模式

一个业务由不同的功能组合，那么可以创建的一个建造者来统一组合这些功能



# 适配器模式

让两个不相关的类产生关联

1. 对象呢适配器，让个对象去做适配类
2. 接口适配器，抽象方法实现接口来做

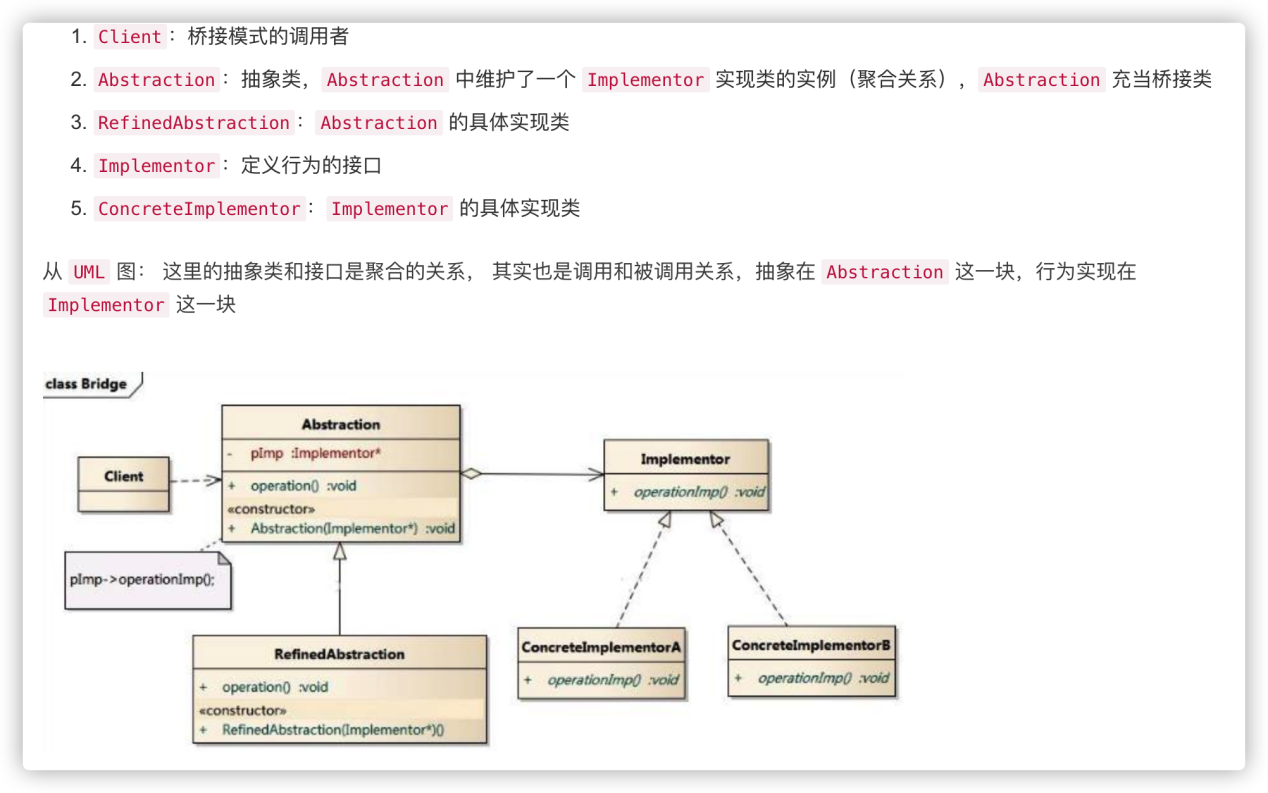
SpringMVC的HandlerAdapter有用到

# 桥接模式

手机品牌案例

业务关联密集的类，可以分类在通过某一类的抽象层做去聚合另一个类业务对象类关联，这样这个聚合了其他类的抽象层充当两个业务的桥梁。在修改添加的时候都在这里操作。

jdbc连接数据库就有用到这个模式

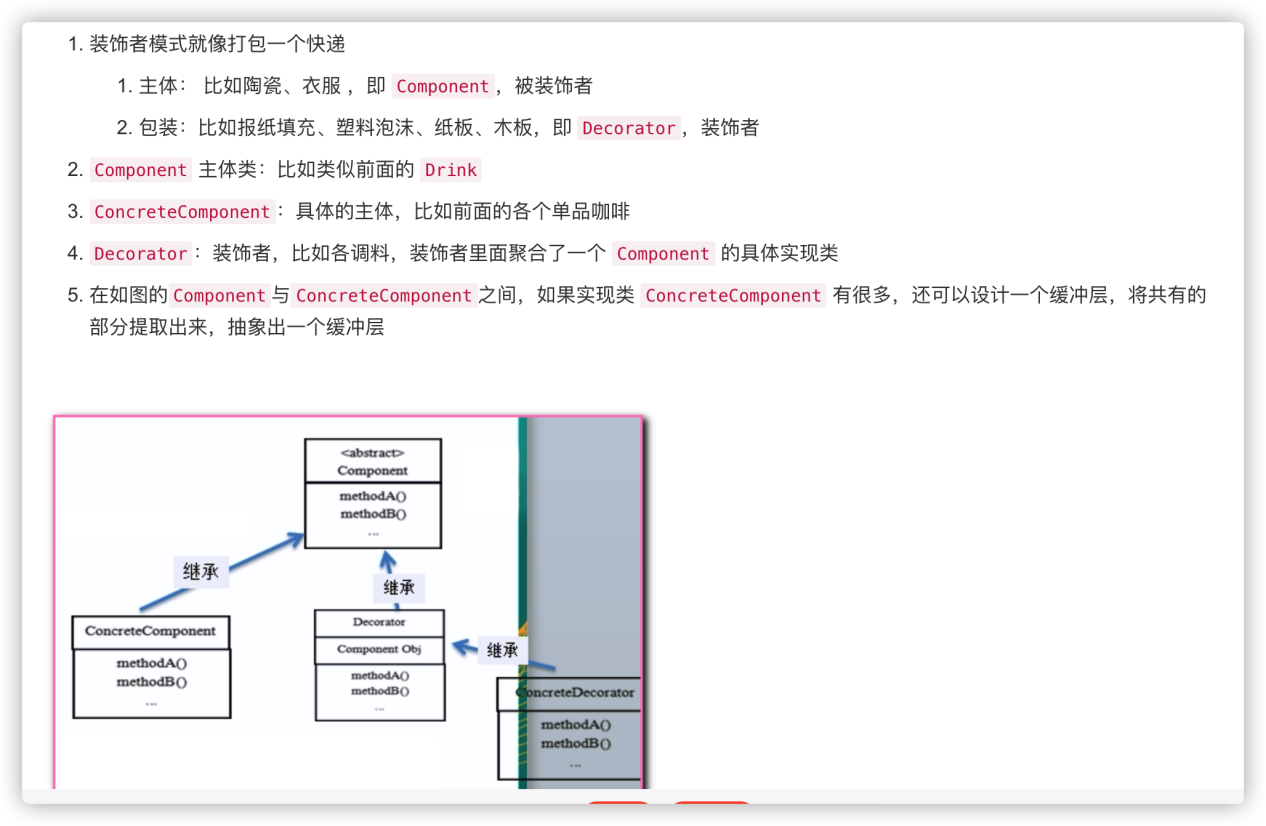


# 装饰者模式

动态的将新功能附加到对象上--->点咖啡的案例

被装饰者抽象层需要继装聚合饰者，当装饰者新加内容时可以及时通知到被装饰者

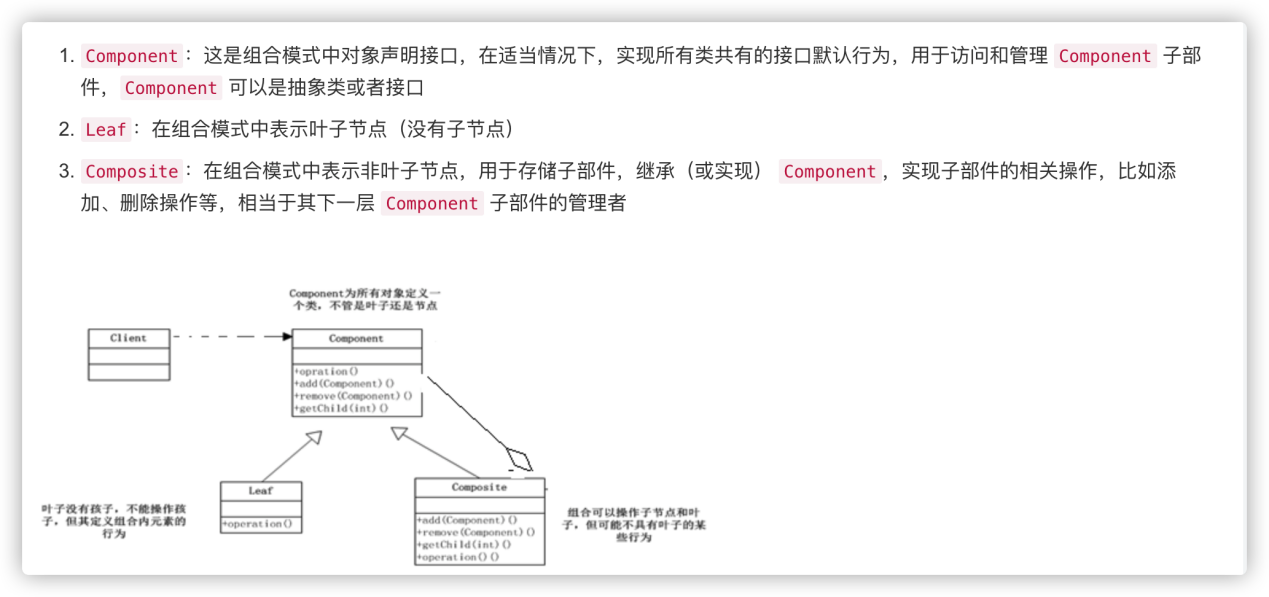
JDKFileInputStream用到



# 组合模式

学院展示案例

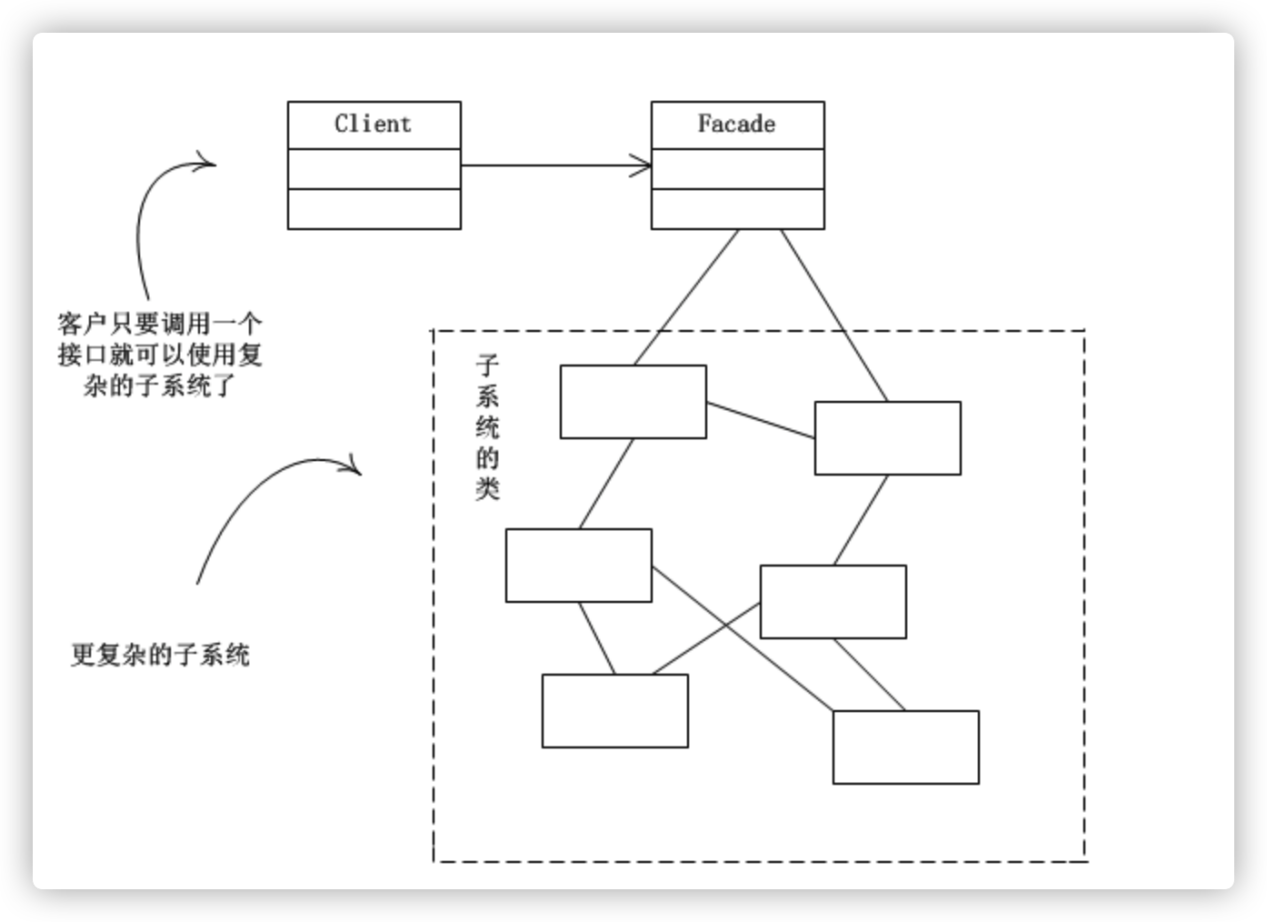
当我们处理的对象可以生成一个树形结构，那么我们规定一个叶子节点，非叶子节点，公共接口来操作



# 外观模式

提供一个统一的类和客户端交流

mybatis有使用到



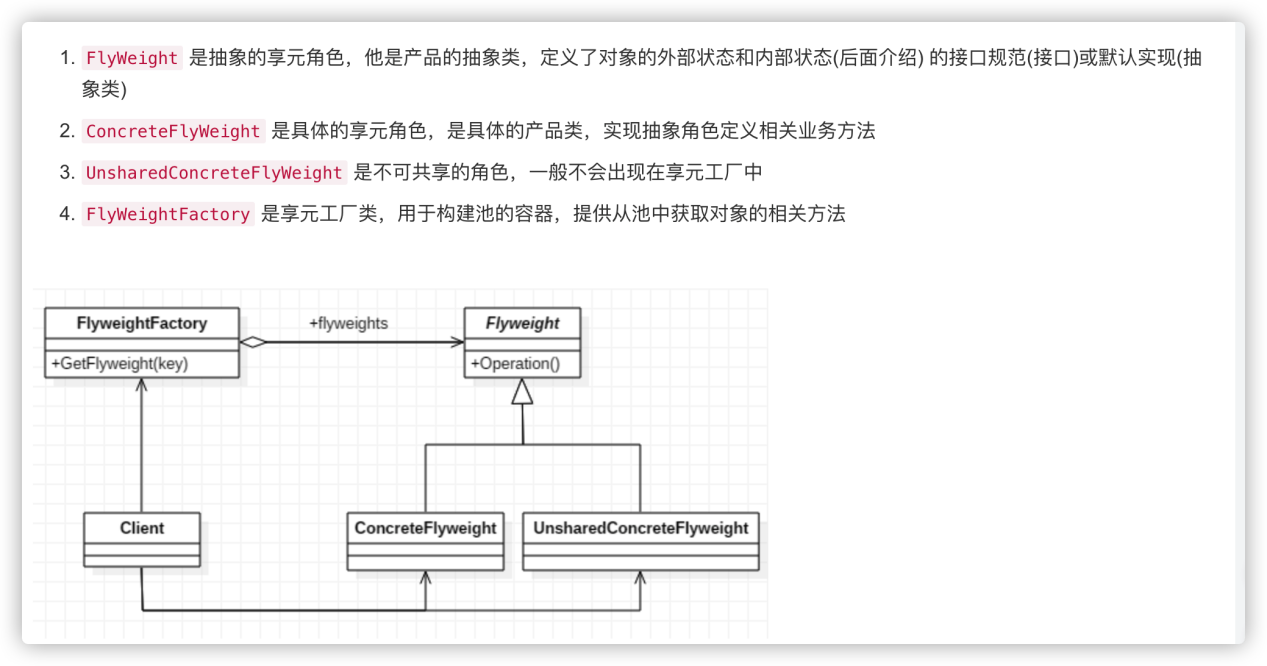
# 享元模式

提出共享的不动大家一起用

外部状态：不共享

内部状态：共享

JDK Interger 如果 Integer.valueOf(x) x 在 -128 --- 127 直接，就是使用享元模式返回,如果不在该范围类，则仍然 new

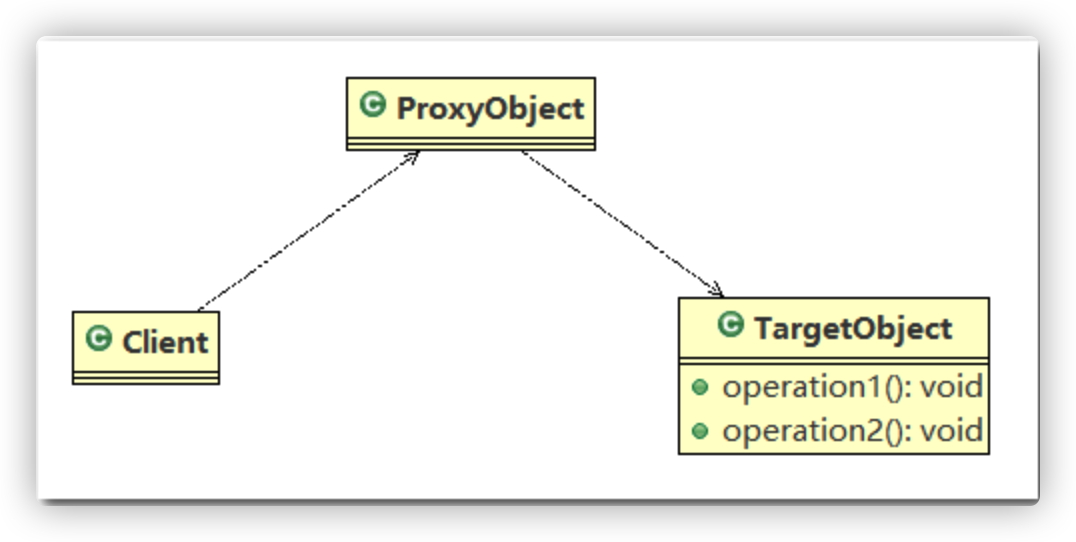


# **代理模式**

1. 静态代理，代理类和目标类都要实现接口
2. 动态代理，jdk的proxy代理，只要目标类实现接口，通过反射代理
3. 动态代理：cglib代理，不需要目标类实现接口也可以 通过目标类子类形式来做的

在AOP编程中如何选择代理模式：

1. 目标对象需要实现接口，用JDK代理
2. 目标对象不需要实现接口，用Cglib代理



代理模式：

1.静态代理

比如继承runnable的类

1.缺点编译期间就得设计好

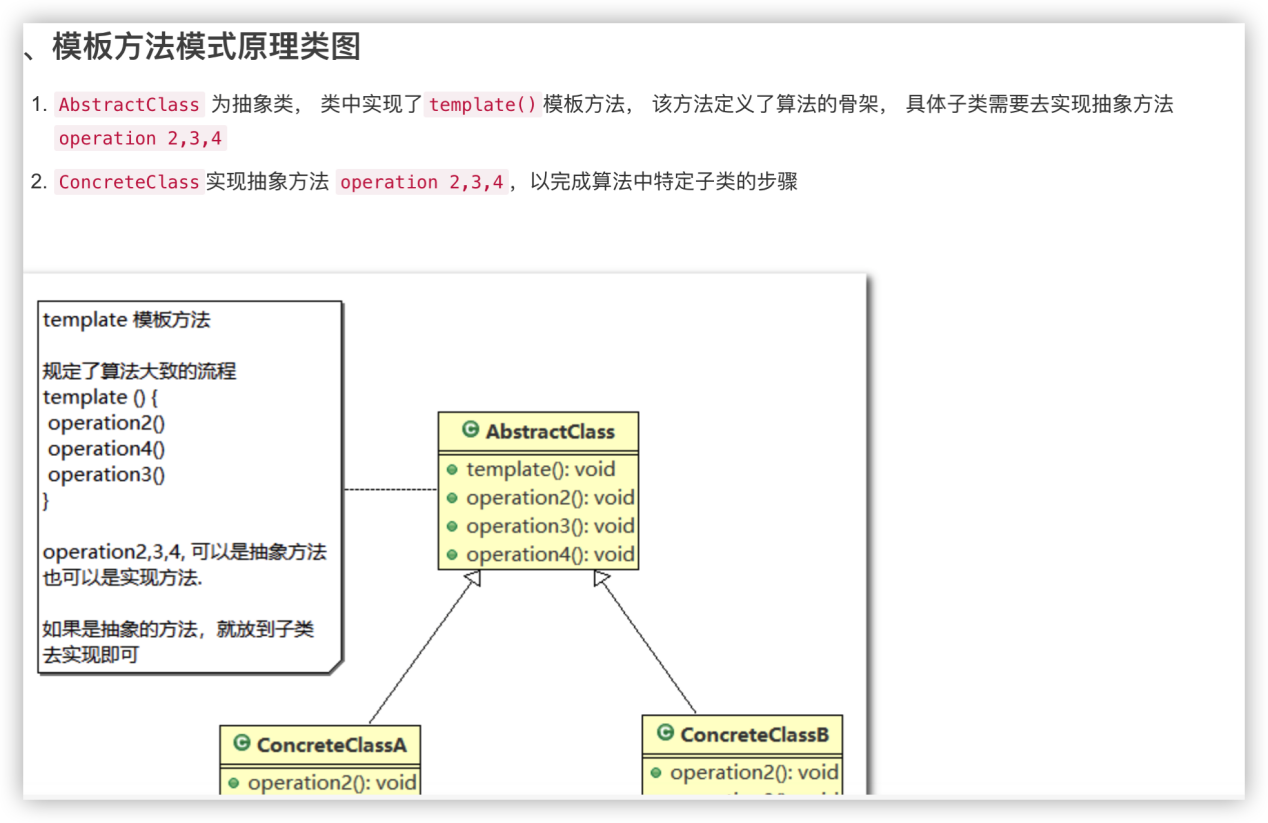
2.每一个代理 只能为 一个接口服务，会产生过多的代理类

2.动态代理

# 模板模式

公用的功能可以做成一个公用的方法，还可以加钩子控制权限

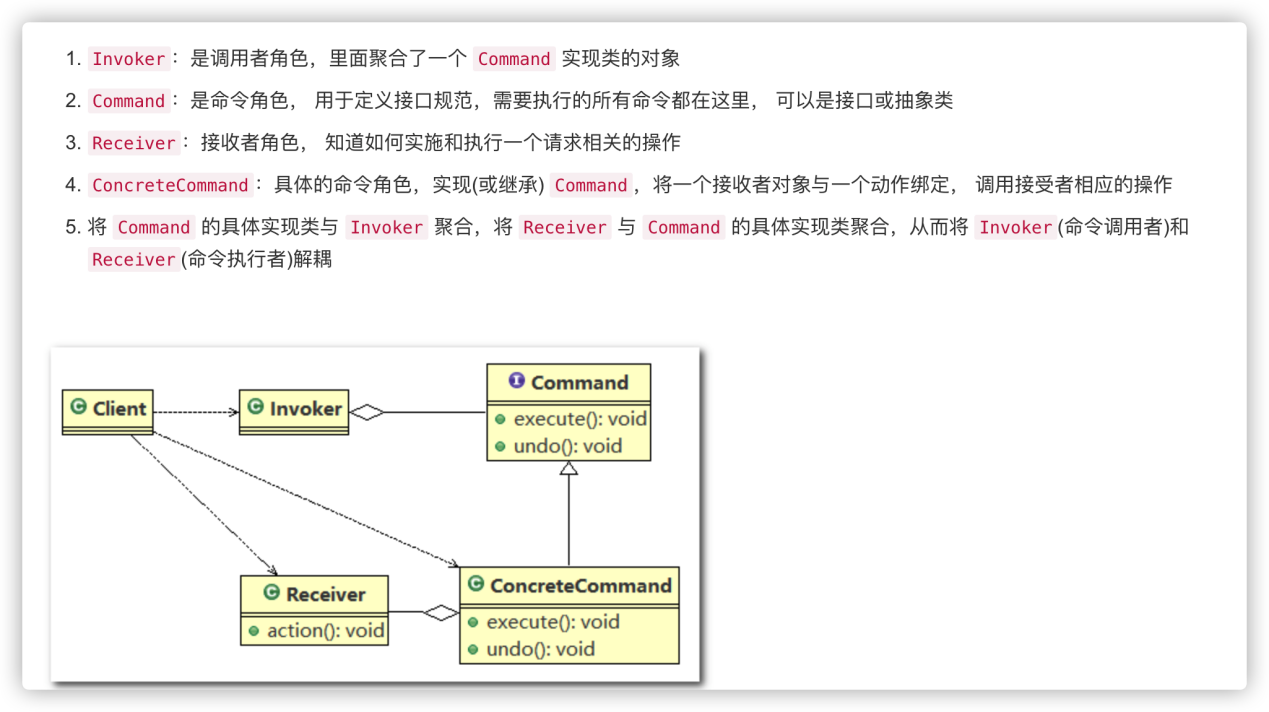
spring的refresh（）方法就用到



# 命令模式

经常需要向某些对象发请求，可以使用命令模式管理这些请求，这样不需要直接操作命令调用的者，实现请求和接受者的解耦

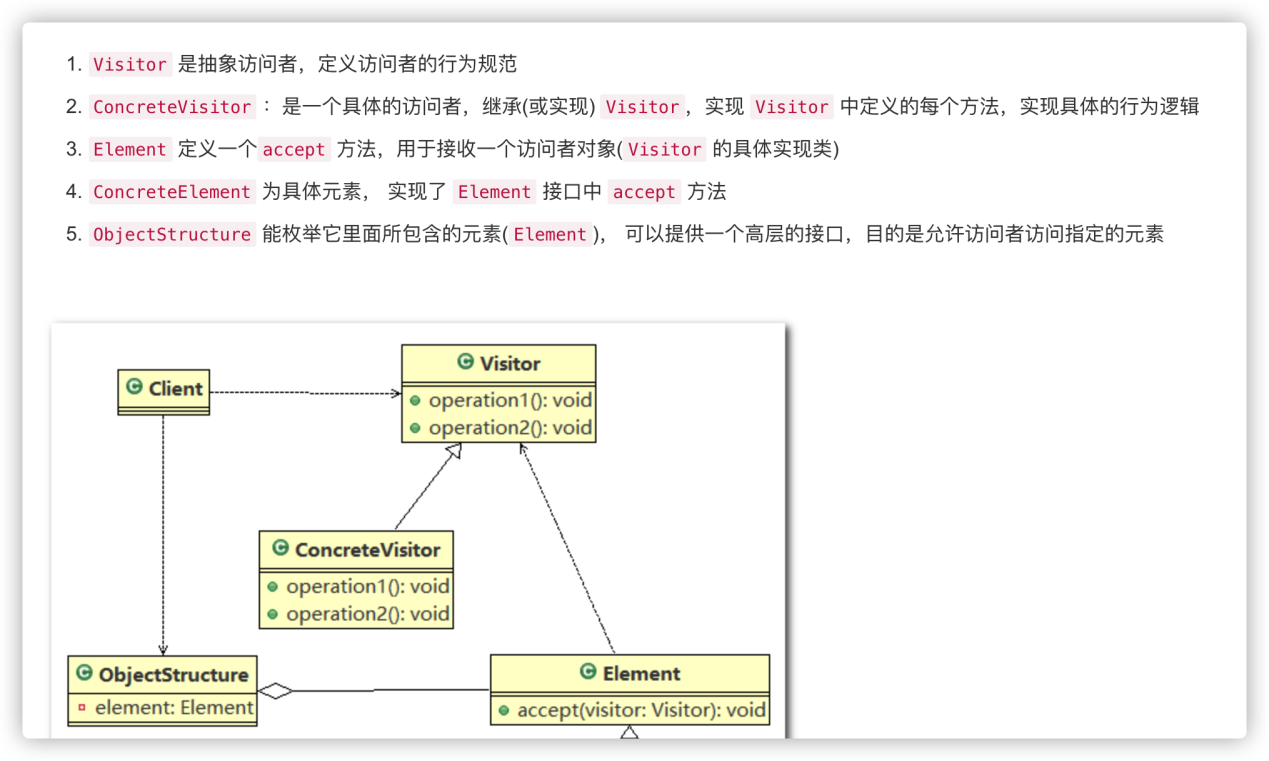
一般一个命令的发起者管理命令，命令、具体的命令、多个命令接受者



# 访问者模式

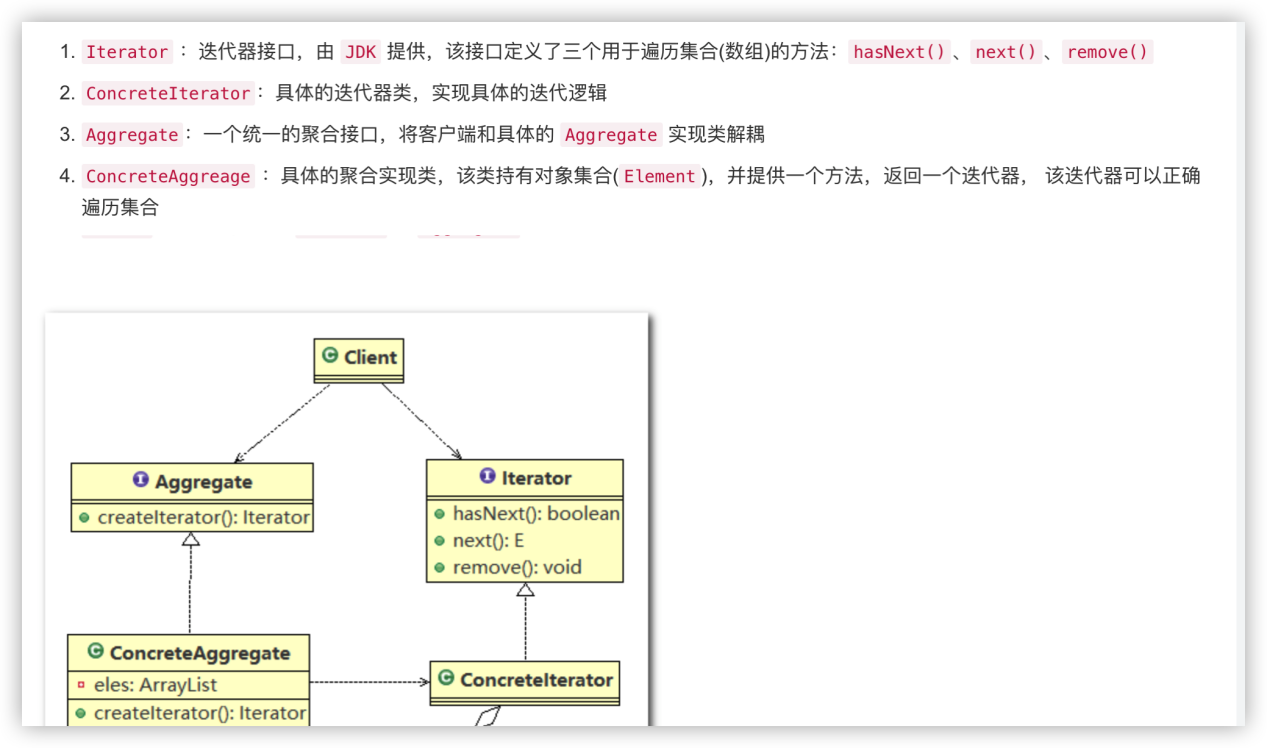
评分案例

当我们一个系统需要经常添加新对象去做某些动作功能时，可以对新加对象抽取一个抽象或者接口层，在这个层里面有聚合需要实现的功能的抽象对象。



# 迭代器模式

继承iterator；就是不同的对象有不同的迭代方式



# 观察者模式

观察者模式类似订牛奶业务

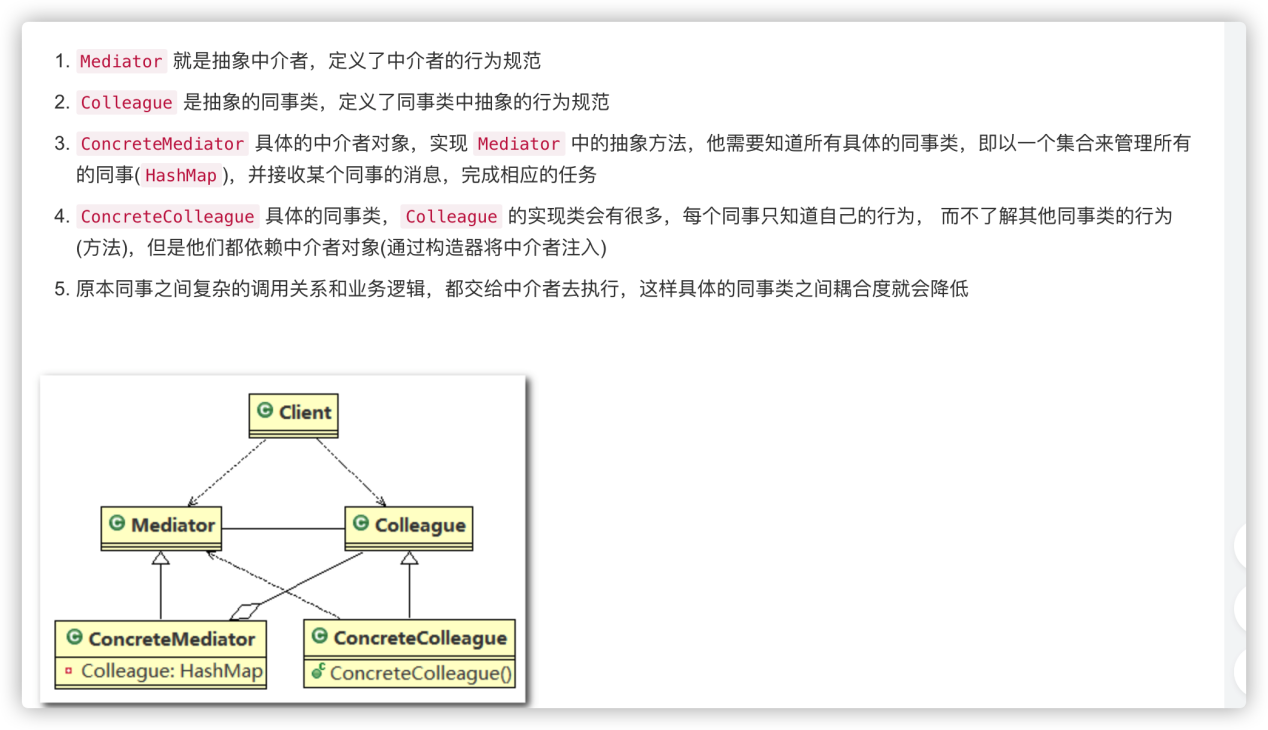
1. 奶站/气象局： Subject
2. 用户/第三方网站： Observer

观察者们去实现观察者接口。观察者接口有聚合到具体的通知业务中来通知观察者

# **中介者模式**

中介租房案例

由中介者去关联某个业务流程来处理问题



# 备忘录

备忘录对象，管理备忘录的类



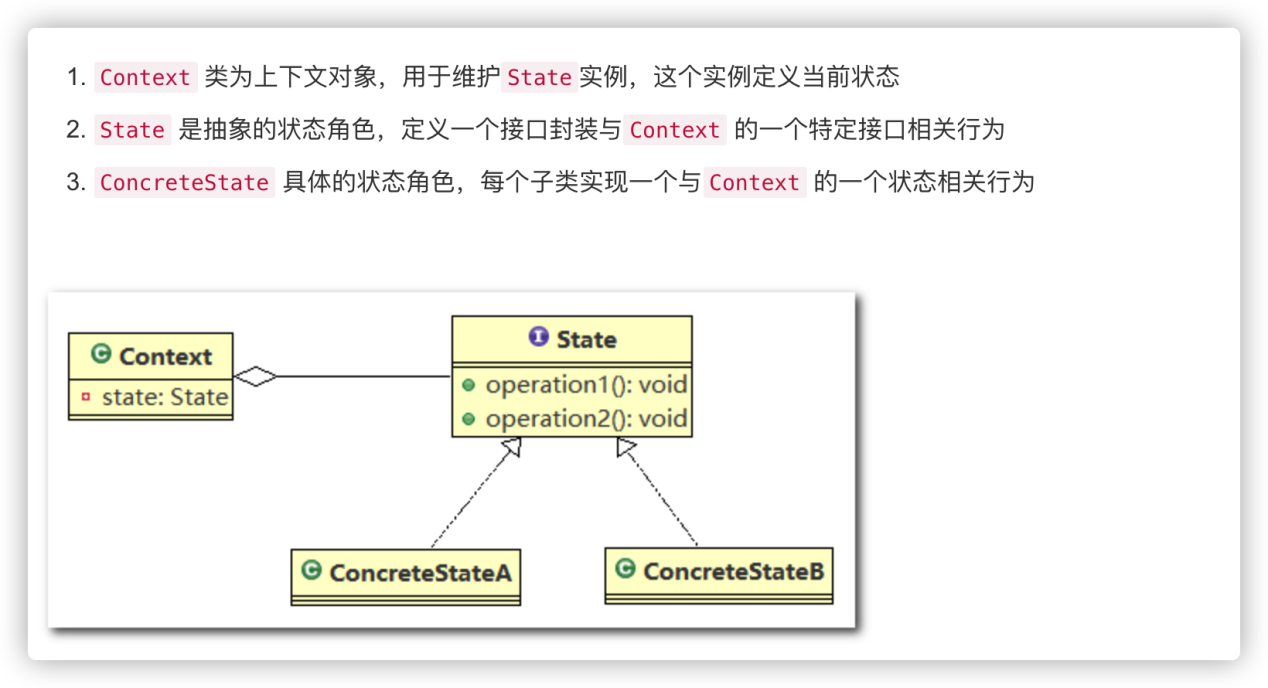
# 解释器模式

解释某些公式

# 状态模式

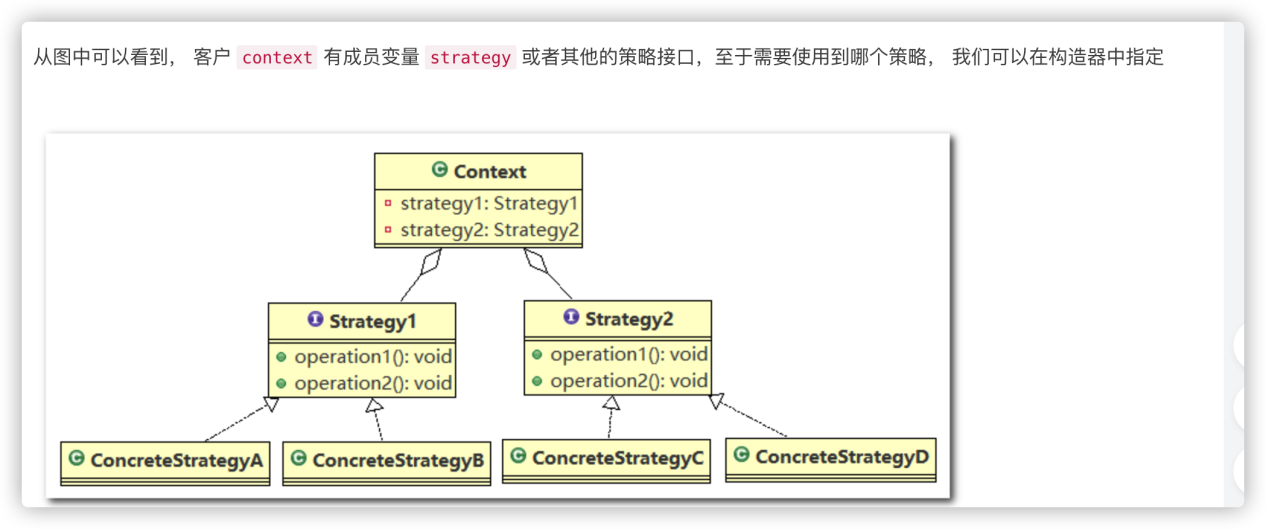
有一个context类来管理状态的变化。状态的变换需要继承状态抽象接口实现各自状态下可以操作的变换步骤。具体的变换由context来管理即可。这样比起使用if-else传统方法：

1. 各个状态值关注自己相关的状态操作，实现状态之间切换解耦
2. 修改更加灵活，新加修改状态的操作只需要在对应操作状态类里面修改即可。



# 策略模式

策略模式： 分别封装行为接口， 实现算法族， 超类里放行为接口对象， 在子类里具体设定行为对象。 原则就是：分离变化部分， 封装接口， 基于接口编程各种功能。 **此模式让行为的变化独立于算法的使用者**



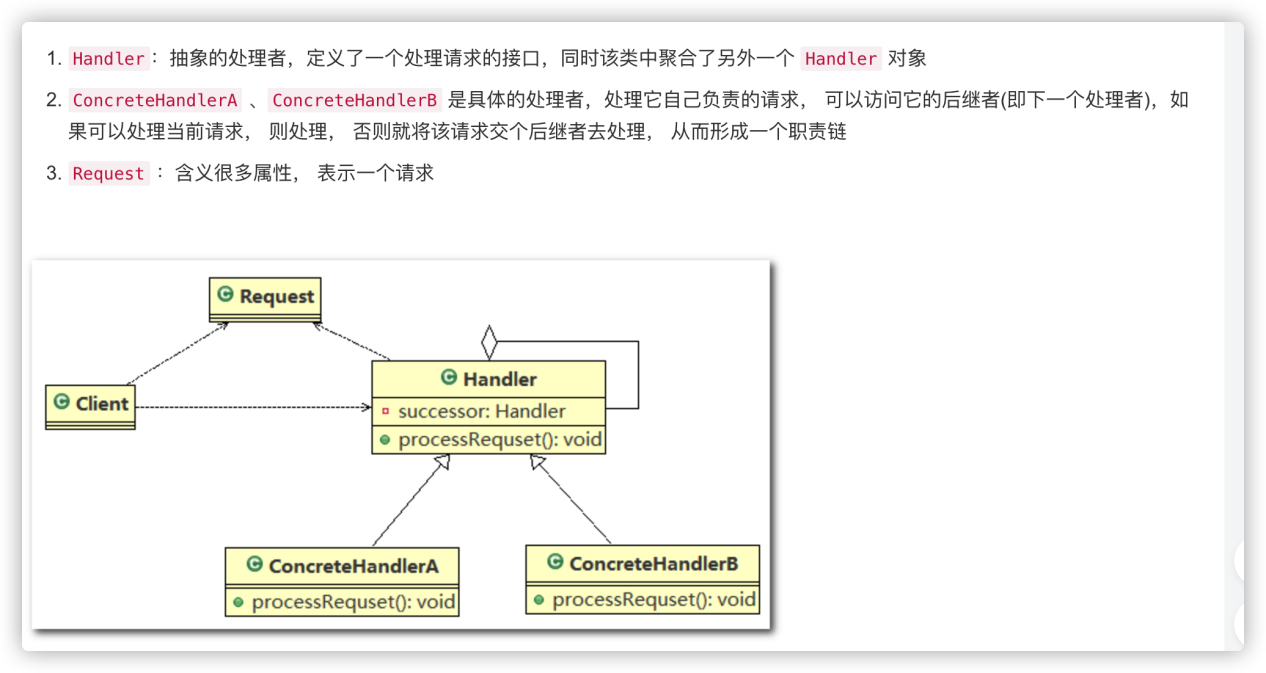
# 责任链模式

为请求创建了一个接收者对象的链。这种模式对请求的发送者和接收者进行解耦

通常可以设计成一个闭环：一个请求可以随意调用链上的任何接口都能找到对应的处理方法

springmvc就使用了这种模式

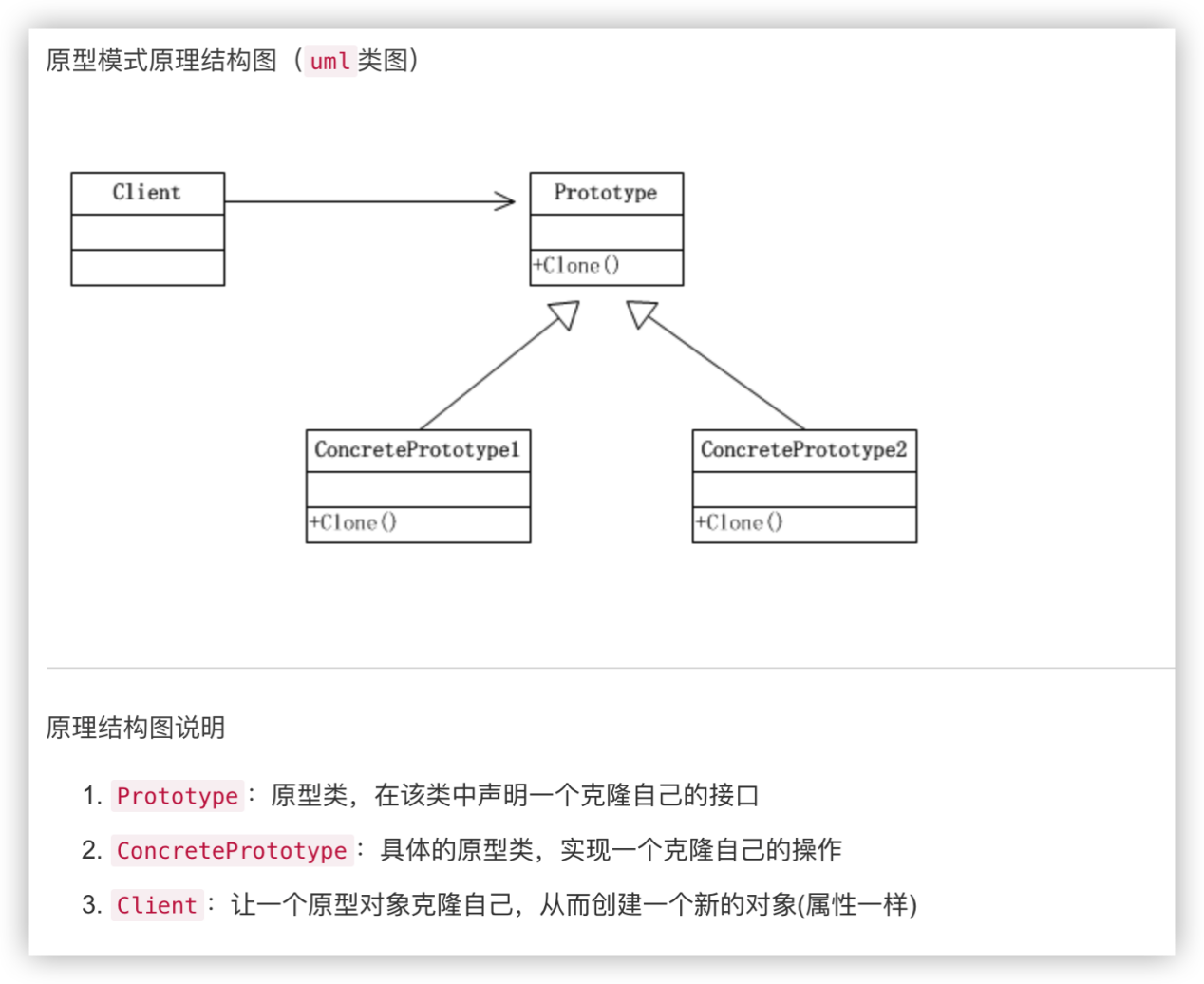
缺点就是：对于业务流程过长的谨慎考虑。



# 设计模式面试题

## 1.原型设计模式问题 ：

1）有请使用UML类图画出原型模式核心角色

  
2）原型设计模式的深拷贝和浅拷贝是什么。并写出深拷贝的两种方式的源码（重写clone方法实现深拷贝、使用序列化来实现深拷贝）

浅拷贝：对于数据类型是基本数据类型的成员变量，浅拷贝会直接进行值传递，也就是将该属性值复制一份给新的对象

对于数据类型是引用数据类型的成员变量，比如说成员变量是某个数组、某个类的对象等，那么浅拷贝会进行引用传递，也就是只是将该成员变量的引用值（内存地址）复制一份给新的对象。因为实际上两个对象的该成员变量都指向同一个实例。在这种情况下，在一个对象中修改该成员变量会影响到另一个对象的该成员变量值

前面我们克隆羊就是浅拷贝，浅拷贝是使用默认的clone()方法来实现：sheep = (Sheep) super.clone();

深拷贝：

1. 复制对象的所有基本数据类型的成员变量值
2. 为所有引用数据类型的成员变量申请存储空间，并复制每个引用数据类型成员变量所引用的对象，直到该对象可达的所有对象。也就是说，对象进行深拷贝要对整个对象进行拷贝
3. 深拷贝实现方式 1：重写clone方法来实现深拷贝
4. 深拷贝实现方式 2：通过对象序列化实现深拷贝(推荐)

3）在Spring框架中哪里使用到原型模式，并对源码进行分析  
beans.xml

<bean id="id01" class="com.demo.spring.bean.Monster" scope="prototype" />

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml"); *// 获取monster[通过id获取monster]* Object bean = applicationContext.getBean("id01"); System.out.println("bean" + bean); *// 输出 "牛魔王" .....* Object bean2 = applicationContext.getBean("id01"); System.out.println("bean2" + bean2); *// 输出 "牛魔王" .....* System.out.println(bean == bean2); *// false*

## 2.状态模式问题

这类代码难以应对变化，在添加一种状态时，我们需要手动添加if/else，在添加一种功能时，要对所有的状态进行判断。因此代码会变得越来越臃肿，并且一旦没有处理某个状态，便会发生极其严重的BUG，难以维护。

## 写出几种熟悉的设计模式，并做简单介绍？

## 4.列出除Singleton外的常用的3种设计模式，并简单描述？

## 5.工厂设计模式的优缺点是什么？

答：工厂设计模式的思路是由一个工厂类根据传入的参数（一般是字符串参数），动态决定应该创建哪一个产品子类（这些产品子类继承自同一个父类或接口）的实例，并以父类形式返回。

优点：客户端不负责对象的创建，而是由专门的工厂类完成；客户端只负责对象的调用，实现了创建和调用的分离，降低了客户端代码的难度。

缺点：如果增加和减少产品子类，需要修改简单工厂类，违背了开闭原则；如果产品子类过多，会导致工厂类非常的庞大，违反了高内聚原则，不利于后期维护。

## 6.说出几个在[JDK](https://www.wkcto.com/article/detail/164)库中使用的设计模式？

答：装饰器设计模式（Decorator design pattern）被用于多个Java IO类中。单例模式（Singleton pattern）用于Runtime，Calendar和其他的一些类中。工厂模式（Factory pattern）被用于各种不可变的类如Boolean，像Boolean.valueOf，观察者模式（Observer pattern）被用于Swing和很多的事件监听中。

## 7. 什么是设计模式？你是否在你的代码里面使用过任何设计模式？

## **8.在 Java 中，什么叫观察者设计模式（observer design pattern）**？

观察者模式是基于对象的状态变化和观察者的通讯，以便他们作出相应的操作。简单的例子就是一个天气系统，当天气变化时必须在展示给公众的视图中进行反映。这个视图对象是一个主体，而不同的视图是观察者。

## **9.使用工厂模式最主要的好处是什么？在哪里使用？**

工厂模式的最大好处是增加了创建对象时的封装层次。如果你使用工厂来创建对象，之后你可以使用更高级和更高性能的实现来替换原始的产品实现或类，这不需要在调用层做任何修改。

## **10.Java 编程为什么不允许从静态方法中访问非静态变量？**

Java 中不能从静态上下文访问非静态数据只是因为非静态变量是跟具体的对象实例关联的，而静态的却没有和任何实例关联。