# 1. 六大原则

## 1.1. 单一职责原则

![](data:image/jpeg;charset=UTF-8;base64,)

### 1.1.1. 简介

**姓名** ：单一职责原则

**英文名** ：Single Responsibility Principle

**座右铭** ：There should never be more than one reason for a class to change. 应当有且仅有一个原因引起类的变更。。。意思就是不管干啥，我都只干一件事，你叫我去买菜，我就只买菜，叫我顺便去倒垃圾就不干了，就这么拽

**脾气** ：一个字“拽”，两个字“特拽“

**伴侣** ：老子职责单一，哪来的伴侣？

**个人介绍** ：在这个人兼多责的社会里，我显得那么的特立独行，殊不知，现在社会上发生的很多事情都是因为没有处理好职责导致的，比如，经常有些父母带着小孩，一边玩手机，导致小孩弄丢、发生事故等等

### 1.1.2. 单一职责应用范围

单一职责原则适用的范围有接口、方法、类。按大家的说法，接口和方法必须保证单一职责，类就不必保证，只要符合业务就行。

#### 1.1.2.1. 方法

设想一下这个场景：假设我们要做一个用户修改名字以及修改密码的功能，可以有多种实现方案，比如下面列举 2 种实现方式

##### 1.1.2.1.1. 第一种实现方式

/\*\*  
 \* 错误的示范  
 \*/  
enum OprType {  
 /\*\*  
 \* 更新密码  
 \*/  
 UPDATE\_PASSWORD,  
 /\*\*  
 \* 更新名字  
 \*/  
 UPDATE\_NAME;  
}  
  
interface UserOpr {  
 boolean updateUserInfo(User user, OprType oprType);  
}  
  
class UserOprImpl implements UserOpr {  
  
 @Override  
 public boolean updateUserInfo(User user, OprType oprType) {  
 if (oprType == OprType.UPDATE\_NAME) {  
 // update name  
 } else if (oprType == OprType.UPDATE\_PASSWORD) {  
 // update password  
 }  
 return true;  
 }  
}

##### 1.1.2.1.2. 第二种实现方式

/\*\*  
 \* 正确的示范  
 \*/  
interface UserOpr2 {  
 boolean updatePassword(User user, String password);  
 boolean updateUserInfo(User user);  
}  
  
class UserOprImpl2 implements UserOpr2 {  
  
 @Override  
 public boolean updatePassword(User user, String password) {  
 user.setPassword(password);  
 // update password  
 return true;  
 }  
  
 @Override  
 public boolean updateUserInfo(User user) {  
 // update user info  
 return true;  
 }  
}

2 种实现有什么区别呢？ 第一种实现通过 OprType 类型的不同来做不同的事情，把修改密码和修改名字耦合在一起，容易引起问题，只要稍不注意，传错枚举值就悲剧了，在代码中也没法很直接看到是做什么操作，也就是这个方法的职责不明确。而第二种实现，把修改密码和修改名字分离开来，也就是把修改密码和修改名字都当做独自的职责处理，这样子就很清晰明了，你调用哪个方法，就很明确的知道这个方法是实现什么逻辑。结论是啥呢？用第二种方式实习才符合单一职责原则。现实中看到很多像第一种实现的代码，而且是枚举有十来个的情况，看代码真费劲。

#### 1.1.2.2. 接口

设想一下这个场景，假设我们让小明去倒垃圾，小红去买菜，小红回来后再叫小红去洗碗。下面也举 2 个实现的例子。

##### 1.1.2.2.1. 第一种实现方式

/\*\*  
 \* 错误的示范  
 \*/  
interface Housework {  
 void shopping();  
 void pourGarbage();  
}  
  
class XiaoMing implements Housework {  
  
 @Override  
 public void shopping() {  
 // 不购物  
 }  
  
 @Override  
 public void pourGarbage() {  
 System.out.println("pourGarbage ...");  
 }  
}  
  
class XiaoHong implements Housework {  
  
 @Override  
 public void shopping() {  
 System.out.println("shopping ...");  
 }  
  
 @Override  
 public void pourGarbage() {  
 // 从不倒垃圾  
 }  
}

中途回来小红去洗碗，要怎么实现？按这个写法，就在 Housework 接口添加 washingUp() 方法，然后小明和小红依次都实现洗碗这个方法，只是小明不做具体实现代码，这样子是不是觉得很别扭，不符合单一职责原则的，修改一个地方，不影响其他不需要改变的地方，只对需要用到的地方做修改。小明本来就不用洗碗，却要去实现洗碗这个方法。

##### 1.1.2.2.2. 第二种实现方式

/\*\*  
 \* 正确的示范  
 \*/  
interface Shopping {  
 void doShopping();  
}  
  
interface PourGarbage {  
 void doPourGarbage();  
}  
  
interface WashingUp {  
 void doWashingUp();  
}  
  
class XiaoMing2 implements PourGarbage {  
  
 @Override  
 public void doPourGarbage() {  
 System.out.println("pourGarbage ...");  
 }  
}  
  
class XiaoHong2 implements Shopping, WashingUp {  
  
 @Override  
 public void doShopping() {  
 System.out.println("shopping ...");  
 }  
  
 @Override  
 public void doWashingUp() {  
 System.out.println("washing up ...");  
 }  
}

可以看到，这种实现把不同的家务都当做不同的职责，分离开来，这种实现可以按需实现做家务的类型，小明只需要去倒垃圾，就实现 PourGarbage 接口，小红去购物和洗碗，就实现 Shopping 和 WashingUp 接口，完全不会影响到对方，这才是完美的根据单一职责原则编写出来的代码。

#### 1.1.2.3. 类

类这个看了一些资料都说没法硬性要求一定按单一职责原则分，或者说类的职责可大可小，没有很明确的像上面接口那样按照单一职责原则分就很清晰也很有道理。 设想一下这个场景：我们要实现一个用户注册、登录、注销操作，可以像如下 2 种实现方式

##### 1.1.2.3.1. 第一种实现方式

从用户的角度考虑，这些操作都是用户的行为，可以放在一个统一的类 UserBiz

class UserBiz {  
  
 public boolean register(User user){  
 // 注册操作  
 return true;  
 }  
  
 public boolean login(User user) {  
 // 登录操作  
 return true;  
 }  
  
 public boolean logout(User user) {  
 // 注销操作  
 return true;  
 }  
  
}

##### 1.1.2.3.2. 第二种实现方式

有人又说，不是说单一职责么？从业务操作考虑，需要把注册、登录、注销分开

class UserRegisterBiz {  
  
 public boolean register(User user){  
 // 注册操作  
 return true;  
 }  
  
}  
  
class UserLoginBiz {  
  
 public boolean login(User user) {  
 // 登录操作  
 return true;  
 }  
  
}  
  
class UserLogoutBiz {  
  
 public boolean logout(User user) {  
 // 注销操作  
 return true;  
 }  
  
}

感觉像是在抬杠，其实这个没有好坏之分，根据具体业务具体分析，你说你的登录、注册、注销操作代码很多，需要分开，那就分开，无可厚非。

### 1.1.3. 好处

1. 类的复杂性降低，实现什么职责都有清晰明确的定义
2. 可读性提高，复杂性降低，那当然可读性提高了
3. 可维护性提高，可读性提高，那当然更容易维护了
4. 变更引起的风险降低，变更是必不可少的，如果接口的单一职责做得好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助 (来自《设计模式之禅》)

### 1.1.4. 总结

这个单一职责原则，目的就是提高代码的可维护性、可读性、扩展性，如果为了单一职责而破坏了这 3 个特性，可能会得不偿失。

## 1.2. 里氏替换原则

![](data:image/jpeg;charset=UTF-8;base64,)

### 1.2.1. 简介

**姓名** ：里氏替换原则

**英文名** ：Liskov Substitution Principle

**座右铭** ： 1. If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T. 如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1都代换成o2时，程序P的行为没有发生变化，那么类型S是类型T的子类型。

1. Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. 所有引用基类的地方必须能透明地使用其子类的对象。

这 2 个定义来自《设计模式之禅》，比较干巴巴，不认真思考起来可能不太容易懂。简单来说就是定义了什么是父子。在现实生活中，什么是父子？就是生你的那个男人和你的关系就是父子（父女）。而这里定义的就是假如 A 能胜任 B 干的所有事情，那 B 就是 A 的父亲，也就是儿子要会父亲的所有能活，儿子活得再烂也要有父亲的水平。

**价值观** ：很显然，比较传统，严父出孝子。儿子必须要有父亲的能耐，最好青出于蓝胜于蓝。

**伴侣** ：估计有个贤惠的老婆，才能有这么优秀的儿子。

**个人介绍** ：我比较严厉，也是为了生存没办法，只有一辈一辈地变优秀，一直坚持下去，家族就会越来越好。这样就可以富过三代，你看你们人类不是经常说富不过三代。。。扎心了老铁，老子还是富零代。

### 1.2.2. 老爹开车，前方注意

里氏替换原则定义了什么是父子，还有一点要注意的，就是儿子不能在父亲会的技能上搞“创新”。 比如父亲会做红烧排骨，儿子在新东方烹饪学校中学到了一招，在红烧排骨里面加糖和醋，变成红烧糖醋排骨，更加美味，看代码，儿子在父亲的基础红烧排骨上加了糖醋，好像没啥问题。

class Father1 {  
  
 public void braisedRibs(){  
 System.out.println("红烧排骨");  
 }  
  
}  
  
  
class Son1 extends Father1 {  
  
 public void braisedRibs(){  
 System.out.println("红烧糖醋排骨");  
 }  
  
}

运行下面代码，会打印：红烧排骨

Father1 father1 = new Father1();  
father1.braisedRibs();

我们上面说过，所有在使用父亲的地方，都能够替换成儿子，并且效果是一样的，那接下来我们改一下代码。

Son1 son1 = new Son1();  
son1.braisedRibs();

结果是啥？打印出：红烧糖醋排骨，出乎意料吧。。。这结果完全不一样。想一下上面说的：老爸会的老子也要会，很明显，上面的例子老子不会红烧排骨，只会红烧糖醋排骨，所以这根本不是父子关系。

那应该怎么实现呢？其实红烧排骨和红烧糖醋排骨这压根就是 2 道菜，你去餐馆吃饭的时候，你点红烧排骨服务员给你送来红烧糖醋排骨，或者你点红烧糖醋排骨服务员给你送来红烧排骨，你这时候不生气，算我输。

来看看 Son2，Son2 将红烧糖醋改为 braisedSweetAndSourPorkRibs （翻译不好找 Google 算账去哈，反正不是我翻译的）。

class Son2 extends Father1 {  
   
 public void braisedSweetAndSourPorkRibs(){  
 System.out.println("红烧糖醋排骨");  
 }  
   
}

测试一下是不是好儿子

Son2 son2 = new Son2();  
son2.braisedRibs();  
son2.braisedSweetAndSourPorkRibs();  
  
打印出：  
红烧排骨  
红烧糖醋排骨

这才是 Father1 的好儿子嘛，不仅会红烧排骨，还会红烧糖醋排骨。所以说里氏替换原则就是在定义父子关系，大家都遵守这个定义，就会一代比一代好，不遵守大家也看到了，把前辈传下来的都毁于一旦了。

### 1.2.3. 优缺点

下面再贴一下书本上的一些优缺点

#### 1.2.3.1. 优点

1. 代码共享，减少创建类的工作量，每个子类都拥有父类的方法和属性；
2. 提高代码的重用性；
3. 子类可以形似父类，但又异于父类，“龙生龙，凤生凤，老鼠生来会打洞”是说子拥有父的“种”，“世界上没有两片完全相同的叶子”是指明子与父的不同；
4. 提高代码的可扩展性，实现父类的方法就可以“为所欲为”了，君不见很多开源框架的扩展接口都是通过继承父类来完成的；
5. 提高产品或项目的开放性。

#### 1.2.3.2. 缺点

1. 继承是侵入性的。只要继承，就必须拥有父类的所有属性和方法；
2. 降低代码的灵活性。子类必须拥有父类的属性和方法，让子类自由的世界中多了些约束；
3. 增强了耦合性。当父类的常量、变量和方法被修改时，需要考虑子类的修改，而且在缺乏规范的环境下，这种修改可能带来非常糟糕的结果————大段的代码需要重构。 (来自《设计模式之禅》)

### 1.2.4. 总结

好了，里氏替换原则的大概原理讲得差不多，大家只要记住是在定义“父子关系”，就像游戏规则一样，定义后让大家遵守，会让大家的程序在后面越来越复杂的时候也能清晰，而不会越来越乱。

## 1.3. 依赖倒置原则

![](data:image/jpeg;charset=UTF-8;base64,)

### 1.3.1. 简介

**姓名** ：依赖倒置原则

**英文名** ：Dependence Inversion Principle

**价值观** ：大男子主义的典型代表，什么都得通过老大或者老爸同意。

**伴侣** ：一定是个温柔体贴的女子。

**个人介绍** ：

1. High level modules should not depend upon low level modules.Both should depend upon abstractions. 高层模块不应该依赖低层模块，两者都应该依赖其抽象（模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的）
2. Abstractions should not depend upon details. 抽象不应该依赖细节（接口或抽象类不依赖于实现类）
3. Details should depend upon abstractions. 细节应该依赖抽象（实现类依赖接口或抽象类）

给大家讲个故事，我胡乱想的，如有雷同，肯定是英雄所见略同。那必须交个朋友。

一个小村里，有两家饭馆，虽然挂着不同的牌子，挨在一起，但是老板确是表兄弟。这两兄弟抠得很，为了节省成本，密谋了一个想法：在两家饭馆谁家忙的时候，可以让不忙的那家的员工过去支援一下。这样子，本来每家饭馆都需要 2 个洗碗工，总共需要 4 个，他们就只招了 3 个，省了 1 个洗碗工的成本，当然不止洗碗工，还有服务员等等。两兄弟约定了规则：

1. A 饭馆需要支援的时候，B 饭馆老板，让 B 饭馆老板选哪个员工去支援，不能直接让 A 饭馆的员工直接找 B 饭馆的员工去帮忙，但可以让 A 饭馆员工找 B饭馆老板告知需要支援。
2. 虽然老板权利大，但是也不能说 A 饭馆老板直接叫 B 饭馆的员工去帮忙。
3. 员工没有真实的老板，今天为 A 饭馆工作就是 A 饭馆的员工，没有跟定哪个老板。

大概通过这个小故事，描述了依赖倒置原则的基本内容。

### 1.3.2. 代码体现

下面通过代码来模拟这个故事。

#### 1.3.2.1. 错误的示范

这个错误的示范将就看哈，可能有些问题没描述清楚。

##### 1.3.2.1.1. 老板和员工抽象

abstract class Boss {  
  
 abstract void support();  
  
 abstract void askHelp(Boss boss);  
}  
  
abstract class Staff {  
  
 private String name;  
  
 abstract void service();  
  
 abstract void askHelp(Boss boss);  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

##### 1.3.2.1.2. 老板具体类

class BossA extends Boss {  
  
 private StaffA staffA;  
  
 public BossA(StaffA staffA) {  
 this.staffA = staffA;  
 }  
  
 @Override  
 void support() {  
 staffA.service();  
 }  
  
 @Override  
 void askHelp(Boss boss) {  
 boss.support();  
 }  
  
}  
  
class BossB extends Boss {  
  
 private StaffB staffB;  
  
 public BossB(StaffB staffB) {  
 this.staffB = staffB;  
 }  
  
 @Override  
 void support() {  
 staffB.service();  
 }  
  
 @Override  
 void askHelp(Boss boss) {  
 boss.support();  
 }  
}

##### 1.3.2.1.3. 员工具体类

class StaffA extends Staff {  
  
 public StaffA(String name) {  
 this.setName(name);  
 }  
  
 @Override  
 void service() {  
 System.out.println(this.getName() + "提供服务");  
 }  
  
 @Override  
 void askHelp(Boss boss) {  
 boss.support();  
 }  
}  
  
class StaffB extends Staff {  
  
 public StaffB(String name) {  
 this.setName(name);  
 }  
  
 @Override  
 void service() {  
 System.out.println(this.getName() + "提供服务");  
 }  
  
 @Override  
 void askHelp(Boss boss) {  
 boss.support();  
 }  
}

##### 1.3.2.1.4. 测试代码

/\*\* 初始化老板和员工 \*/  
StaffA staffA = new StaffA("A 员工");  
StaffB staffB = new StaffB(" B 员工");  
Boss bossA = new BossA(staffA);  
Boss bossB = new BossB(staffB);  
  
/\*\* A 老板向 B 老板求支援 \*/  
bossA.askHelp(bossB); // 打印出：B 员工提供服务  
  
/\*\* B 员工向 A 老板求支援 \*/  
staffB.askHelp(bossA); // 打印出：A 员工提供服务

好像看起来实现了要求了，但是其实这段代码没有按照上面的 3 点规则编写，破坏了第 3 点规则，老板们的员工没有用员工的抽象类，破坏了细节依赖抽象这一点。设想一下，假如现在 A 老板把 A 员工辞退了，重新招了个 C 员工，那么怎么实现呢？是不是需要再新增一个 StaffC 类，然后再修改 BossA 类代码，把 StaffA 换成 StaffC。这样超级麻烦，在平时写项目中要时刻考虑这一点：在具体实现类使用其他类，是不是可以用其抽象类？

### 1.3.3. 正确的示范

看了上面那个憋屈的代码，再来看下面简洁的代码，才会发现依赖倒置原则是多么强大。

#### 1.3.3.1. 老板和员工抽象类

abstract class Boss2 {  
  
 private Staff2 staff;  
  
 public Boss2(Staff2 staff) {  
 this.staff = staff;  
 }  
  
 abstract void support();  
  
 abstract void askHelp(Boss2 boss);  
  
 public void setStaff(Staff2 staff) {  
 this.staff = staff;  
 }  
  
 public Staff2 getStaff() {  
 return staff;  
 }  
}  
  
abstract class Staff2 {  
  
 private String name;  
  
 abstract void service();  
  
 abstract void askHelp(Boss2 boss);  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
}

#### 1.3.3.2. 老板类

class BossImpl extends Boss2 {  
  
 public BossImpl(Staff2 staff) {  
 super(staff);  
 }  
  
 @Override  
 void support() {  
 this.getStaff().service();  
 }  
  
 @Override  
 void askHelp(Boss2 boss) {  
 boss.support();  
 }  
}

#### 1.3.3.3. 员工类

class StaffImpl extends Staff2{  
  
 public StaffImpl(String name) {  
 this.setName(name);  
 }  
  
 @Override  
 void service() {  
 System.out.println(this.getName() + "提供服务");  
 }  
  
 @Override  
 void askHelp(Boss2 boss) {  
 boss.support();  
 }  
}

#### 1.3.3.4. 测试类

/\*\* 正确示范 \*/  
Staff2 staffA2 = new StaffImpl("A 员工");  
Staff2 staffB2 = new StaffImpl("B 员工");  
Boss2 bossA2 = new BossImpl(staffA2);  
Boss2 bossB2 = new BossImpl(staffB2);  
  
/\*\* A 老板向 B 老板求支援 \*/  
bossA2.askHelp(bossB2); // 打印出：B 员工提供服务  
  
/\*\* B 员工向 A 老板求支援 \*/  
staffB2.askHelp(bossA2); // 打印出：A 员工提供服务  
  
/\*\* A 老板辞退了 A 员工，换成了 C 员工 \*/  
Staff2 staffC2 = new StaffImpl("C 员工");  
bossA2.setStaff(staffC2);  
  
/\*\* B 员工向 A 老板求支援 \*/  
staffB2.askHelp(bossA2); // 打印出：C 员工提供服务

这代码相比上面错误的示范，简洁了很多，实现的功能却更灵活，这就是依赖倒置原则强大的地方，它可以将类的耦合性降低，提供灵活的处理。

### 1.3.4. 最佳实践

1. 变量的表面类型尽量是接口或者是抽象类
2. 任何类都不应该从具体类派生
3. 尽量不要覆写基类的方法
4. 结合里氏替换原则使用 （来自《设计模式之禅》）

### 1.3.5. 总结

总的来说，要实现依赖倒置原则，要有『面向接口编程』这个思维，掌握好这个思维后，就可以很好的运用依赖倒置原则。

## 1.4. 接口隔离原则

![](data:image/jpeg;charset=UTF-8;base64,)

### 1.4.1. 简介

**姓名** ：接口隔离原则

**英文名** ：Interface Segregation Principle

**价值观** ：宁缺毋滥

**个人介绍** ：

1. Clients should not be forced to depend upon interfaces that they don't use.（客户端不应该依赖它不需要的接口。）
2. The dependency of one class to another one should depend on the smallest possible interface.（类间的依赖关系应该建立在最小的接口上。）

也用一个故事来讲这 2 句干巴巴的定义。

一小伙子跑到大城市的工厂打工，工作了一年半载，越来越觉得没劲，每天干那么多活，又领那么一点工资，和他老爸抱怨这段时间的困扰，老爸想着，家里有个小作坊，自己也一年不如一年了，要不就让儿子回老家管理这小作坊。小伙子熬不过这个年，就跑回老家跟着老爸打理小作坊。

![](data:image/jpeg;charset=UTF-8;base64,) (来自Google Image)

小作坊主要是做布娃娃的，如上图，工作在于打扮包装布娃娃，工序有给布娃娃扎辫子、穿衣服、包装入箱、打标签。整个完整的流程都是一个人做的。有很多个工人每天都在做这个事情。

老爸向小伙子诉苦，感觉招工挺多人的，生产力还是提不上去。小伙子记着老爸的话，在工厂里面观察了几天，他发现每个工人都要做这 4 个打扮包装布娃娃的工序，有些工人扎辫子很快但穿衣服很慢，有些工人扎辫子很慢但穿衣服快，他用了笔记本记下来：李大姨扎辫子快，王大妈穿衣服快，就这样把每个人有效率的工作都记录下来。

一天晚上吃饭，小伙子跟老爸说了自己观察到的现象，也把本子拿给老爸看，跟老爸商量：可不可以做个尝试，不要每个人负责打扮包装布娃娃全步骤，而是按工序分开，每个人只负责一个工序，每个工人只干一件事，更容易熟能生巧。老爸听着觉得有道理。

第二天早上，就到小作坊里，召集了所有工人，按小伙子的笔记上面的名单分工，大家都做好各自负责的内容，像流水线一样，做好了就放到下个工序的地方，让下个工序的人去做。到了下班，小伙子清点了今天工作的成果，包装完成的娃娃比前一天多了 50% 。晚上小伙子跟老爸喝着百威吃起大肉庆祝一番。

这个故事你看了可能想骂爹骂娘，跟上面的定义有啥毛关系？故事只是把大家带入这个场景，我们在工作中，着手开发之前不都得先理清好需求背景，这就是要讲接口隔离原则的背景，通过代码来给大家讲解一下如何用好接口隔离原则。

### 1.4.2. 父亲的运营模式

先看代码

interface Work {  
  
 void hairBraiding();  
 void getDressed();  
 void packingIntoTheBox();  
 void makeTag();  
  
}  
  
class WangMather implements Work{  
  
 @Override  
 public void hairBraiding() {  
 System.out.println("王大妈给布娃娃扎辫子");  
 }  
  
 @Override  
 public void getDressed() {  
 System.out.println("王大妈给布娃娃穿衣服");  
 }  
  
 @Override  
 public void packingIntoTheBox() {  
 System.out.println("王大妈把布娃娃装入箱子");  
 }  
  
 @Override  
 public void makeTag() {  
 System.out.println("王大妈给箱子打标签");  
 }  
}  
  
class LiAunt implements Work {  
  
 @Override  
 public void hairBraiding() {  
 System.out.println("李大姨给布娃娃扎辫子");  
 }  
  
 @Override  
 public void getDressed() {  
 System.out.println("李大姨给布娃娃穿衣服");  
 }  
  
 @Override  
 public void packingIntoTheBox() {  
 System.out.println("李大姨把布娃娃装入箱子");  
 }  
  
 @Override  
 public void makeTag() {  
 System.out.println("李大姨给箱子打标签");  
 }  
}  
  
// 测试代码  
WangMather wangMather = new WangMather();  
wangMather.hairBraiding();  
wangMather.getDressed();  
wangMather.packingIntoTheBox();  
wangMather.makeTag();  
  
LiAunt liAunt = new LiAunt();  
liAunt.hairBraiding();  
liAunt.getDressed();  
liAunt.packingIntoTheBox();  
liAunt.makeTag();

在父亲管理下的小作坊，是大家各自完成好一个布娃娃，工作互不交接，在这种运营模式下，我们把所有工作都合并在一个接口 Work 是没有问题的。有人可能要问，不是说接口隔离么？这里面 Work 接口的 4 个方法都可以分离开，它们都是各自的工作内容。稍等一下，我们现在是基于老父亲运营的模式下实现，如果小作坊一直都是这种模式运营，这段代码有问题么？其实没问题的，我们根据当时的业务考虑，在这种情况下，把 Work 抽成 4 个接口不是不可以，只是不现实，每个工人都去实现一模一样的 4 个接口在老父亲运营模式下是不切实际。

### 1.4.3. 儿子的运营模式

接下来介绍儿子的运营模式。儿子提倡的是每个工人职责分明，只负责一个事情，在这种情况下，如果还是用老父亲的 Work 接口会有什么问题呢？上面我们说了，李大姨扎辫子快，王大妈穿衣服快，所以李大姨被分配去给布娃娃扎辫子，王大妈被分配去给布娃娃穿衣服。我们沿用老父亲的 Work 接口实现，代码如下

class WangMather2 implements Work{  
  
 @Override  
 public void hairBraiding() {  
 }  
  
 @Override  
 public void getDressed() {  
 System.out.println("王大妈给布娃娃穿衣服");  
 }  
  
 @Override  
 public void packingIntoTheBox() {  
 }  
  
 @Override  
 public void makeTag() {  
 }  
}  
  
class LiAunt2 implements Work {  
  
 @Override  
 public void hairBraiding() {  
 System.out.println("李大姨给布娃娃扎辫子");  
 }  
  
 @Override  
 public void getDressed() {  
 }  
  
 @Override  
 public void packingIntoTheBox() {  
 }  
  
 @Override  
 public void makeTag() {  
 }  
}

看出问题来了么？李大姨仅仅参与扎辫子工作，王大妈参与了穿衣服工作，但是却都要依旧实现其他 3 个多余的接口。所以在儿子的运营模式下，老父亲的 Work 接口需要重新分配，以工序的角度分配，而不是以完成一个布娃娃的角度分配。总共有 4 个工序：扎辫子、穿衣服、包装入箱、打标签，我们需要定义 4 个接口，让员工去实现各自负责的工序接口。代码如下

interface Hair {  
 void hairBraiding();  
}  
  
interface Dress {  
 void getDressed();  
}  
  
interface Box {  
 void packingIntoTheBox();  
}  
  
interface Tag {  
 void makeTag();  
}  
  
/\*\*  
 \* 李大姨给布娃娃扎辫子快  
 \*/  
class LiAunt3 implements Hair {  
  
 @Override  
 public void hairBraiding() {  
 System.out.println("李大姨给布娃娃扎辫子");  
 }  
}  
  
/\*\*  
 \* 王大妈给布娃娃穿衣服快  
 \*/  
class WangMather3 implements Dress{  
  
 @Override  
 public void getDressed() {  
 System.out.println("王大妈给布娃娃穿衣服");  
 }  
  
}  
  
/\*\*  
 \* 陈大叔包装快  
 \*/  
class ChenUncle implements Box {  
  
 @Override  
 public void packingIntoTheBox() {  
 System.out.println("陈大叔给布娃娃装箱");  
 }  
}  
  
/\*\*  
 \* 黄大姐贴标签快  
 \*/  
class HuangSister implements Tag {  
  
 @Override  
 public void makeTag() {  
 System.out.println("黄大姐给箱子打标签");  
 }  
}  
  
// 测试代码  
LiAunt3 liAunt3 = new LiAunt3();  
WangMather3 wangMather3 = new WangMather3();  
ChenUncle chenUncle = new ChenUncle();  
HuangSister huangSister = new HuangSister();  
liAunt3.hairBraiding();  
wangMather3.getDressed();  
chenUncle.packingIntoTheBox();  
huangSister.makeTag();

这段代码看起来就很清晰了，在儿子的运营模式下，大家都是只做一道工序，这样子实现就非常合理。看了这个过程，你理解了接口隔离原则了么？再看一看上面的定义：客户端不应该依赖它不需要的接口。闭上眼睛，静默 3 秒，感受一下。 我们也可以回忆一下在工作中编写的代码，是不是有遵守接口隔离原则？在特定的场景下，如果很多类实现了同一个接口，并且都只实现了接口的极少部分方法，这时候很有可能就是接口隔离性不好，就要去分析能不能把方法拆分到不同的接口。

### 1.4.4. 总结

接口隔离原则最最最重要一点就是要**根据实际情况，具体业务具体分析**，不能犯了上面说到的错误：在老父亲的运营模式下，按儿子的工序划分接口去实现，那样子会得不偿失。

## 1.5. 迪米特法则

![](data:image/jpeg;charset=UTF-8;base64,)

### 1.5.1. 简介

**姓名**：迪米特法则

**英文名**：Law of Demeter

**小名**：最少知识原则

**小名英文名**：Least Knowledge Principle

**价值观**：妈妈说不和陌生人说话

**个人介绍**：

1. Each unit should have only limited knowledge about other units: only units "closely" related to the current unit. (每个单元对于其他的单元只能拥有有限的知识：只是与当前单元紧密联系的单元)
2. Each unit should only talk to its friends; don't talk to strangers. (每个单元只能和它的朋友交谈：不能和陌生单元交谈)
3. Only talk to your immediate friends. (只和自己直接的朋友交谈) (来自维基百科)

还是脑洞大开来个小故事。这故事还是比较现实一些，其实也不算是故事，就是咱们经常经历的事情，现在知识付费已经广受欢迎，18 年底更是快速猛涨，各种各样的培训、读书、音频学习软件如雨后春笋一般涌现出来。我们就拿一个读书的例子。

有一天，设计模式老师讲解了迪米特法则，同学们听得云里雾里的，老师怕同学们没掌握这个知识点，就给同学们布置了一个作业，需要同学们按迪米特法则实现。

作业是这样子的：平常在零碎的时间里，喜欢看一些书籍，一般都是电子书，现在我们看书的操作是这样的：唤醒手机，打开阅读软件，选择书籍，然后阅读。总共 3 个步骤，涉及了 3 样东西：手机、软件、书籍。同学们用代码实现这个过程。

第二天上课，同学们纷纷交了作业，老师随手一番，就看到了 2 个鲜明的例子，很明显，就是一好一坏。老师便给同学们讲解了这 2 个例子，让学生感受一番迪米特法则。

### 1.5.2. 错误例子

public class LODErrorTest {  
  
 public static void main(String[] args) {  
 Phone phone = new Phone();  
 phone.readBook();  
 }  
  
}  
  
  
/\*\*  
 \* 错误的示范  
 \*/  
class Phone {  
 App app = new App();  
 Book book = new Book("设计模式");  
 public void readBook() {  
 app.read(book);  
 }  
  
}  
  
  
class App {  
   
 public void read(Book book) {  
 System.out.println(book.getTitle());  
 }  
  
}  
  
class Book {  
  
 private String title;  
  
 public Book(String title) {  
 this.title = title;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
}

代码是完成了读书这个过程，看样子是功能实现了，细看会发现代码的逻辑不对。哪里不对呢？书籍和应用对象都在手机上，现实是我们唤醒手机，这时手机是没有书籍的，只有当我们打开阅读软件，才有书籍可以看，没有阅读软件，书籍是看不了的。因此，手机和书籍没有一毛钱关系，书籍不应该在手机里面。正常的设计是：手机里面有阅读软件，阅读软件里面有书籍，这才符合迪米特法则，按定义来说：手机和阅读软件是朋友，阅读软件和书籍是朋友，可是朋友的朋友不是朋友，也就是手机和书籍不是朋友，所以它们不应该有交集，应该离得远远的。

思考一下现实：工作中如果缺少代码复核这个步骤，就会出现这样子，后果是怎么样呢？会给后人挖坑，而且是大坑，因为和现实中的逻辑是对不上的，况且后人不知道当时的业务背景，只能看代码去熟悉，就会一步错、步步错，所以要好好把控代码质量这一关，因为代码千人千面，没法要求代码风格全部一致，但至少需要实现逻辑是清晰易懂的。

### 1.5.3. 正确例子

public class LODRightTest {  
  
 public static void main(String[] args) {  
 Phone2 phone2 = new Phone2();  
 phone2.readBook();  
 }  
  
}  
  
/\*\*  
 \* 正确的示范  
 \*/  
class Phone2 {  
  
 private App2 app2 = new App2();  
  
 public void readBook() {  
 app2.read();  
 }  
  
}  
  
  
class App2 {  
 private Book2 book2 = new Book2("设计模式");  
  
 public void read() {  
 System.out.println(book2.getTitle());  
 }  
  
}  
  
class Book2 {  
  
 private String title;  
  
 public Book2(String title) {  
 this.title = title;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
}

这段代码就符合迪米特法则，手机中有阅读软件，阅读软件中有书籍，手机没有书籍任何影子。正确代码不用细讲，用心去感受就可以体会到。

### 1.5.4. 总结

迪米特法则主要讲述的观点是高内聚、低耦合。我理解为：**是你的，就别给别人；不是你的，就别拿**。上面定义的朋友也是这个意思。

## 1.6. 开闭原则

![](data:image/jpeg;charset=UTF-8;base64,)

### 1.6.1. 简介

**姓名** ：开闭原则

**英文名** ：Open Closed Principle

**价值观** ：老顽童就是我，休想改变我

**个人介绍** ：

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.（软件中的对象（类，模块，函数等等）应该对于扩展是开放的，但是对于修改是封闭的） （来自维基百科）

停更了三四天了，这几天比较忙，不仅仅是工作上，更多是精神上。周日突然老胃病又复发了，一直疼到凌晨 4，5 点。因为这次疼得蛮厉害的，所以准备去医院看一下医生，这时候才体验到大城市就医之苦。周日晚下载了微医 App （不是做广告哈），也不知道哪家医院好，在深圳两年半还没去过医院，随便选个三甲医院：北京大学深圳医院，看了消化内科门诊的医生列表，整整这一周主任医生都预约满了，顿时很崩溃，打电话给医院预约，最快只能预约 17 号，are you kidding？App 上有个『立即问诊』功能，在线把状况告诉医生，医生一天之内接诊，需要花 60 块，我就尝试一下，没想到第二天医生回复后，说可以下午去医院看，他可以临时加号。就这样跳过了预约，直接看病，不知道你是否也苦于看病烦，可以尝试这个方法，当然，如果你有更好的方法，可以留言让更多的人了解到。

跑题了跑题了，今天是想和大家分享设计模式最后一个原则：开闭原则。这个原则要求就是允许扩展，拒绝修改。既然上面讲到看医生，那就用一个跟看病有关的例子。

### 1.6.2. 故事从这里开始

小明去医院看病，医生开了阿司匹林药，小明去了收费台，付了钱，总共 20 块钱。例子的代码如下：

public class OcpTest {  
  
 public static void main(String[] args) {  
 Hospital hospital = new Hospital();  
 IPatient xiaoMing = new Patient("小明");  
 hospital.sellMedicine(xiaoMing);  
 }  
  
}  
  
  
class Medicine {  
 private String name;  
 private BigDecimal price;  
  
 public Medicine(String name, BigDecimal price) {  
 this.name = name;  
 this.price = price;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public BigDecimal getPrice() {  
 return price;  
 }  
  
 public void setPrice(BigDecimal price) {  
 this.price = price;  
 }  
}  
  
class Hospital {  
  
 private Medicine medicine = new Medicine("阿司匹林", new BigDecimal(20));  
  
 public void sellMedicine(IPatient patient) {  
 BigDecimal money = patient.pay(medicine);  
 System.out.println(patient.getName() + " 花了 " + money.setScale(2, BigDecimal.ROUND\_UP) + " 块钱买了药：" + medicine.getName());  
 }  
  
}  
  
interface IPatient {  
 String getName();  
 BigDecimal pay(Medicine medicine);  
}  
  
class Patient implements IPatient{  
  
 private String name;  
  
 public Patient(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public BigDecimal pay(Medicine medicines) {  
 return medicines.getPrice();  
 }  
  
 @Override  
 public String getName() {  
 return name;  
 }  
  
}

第二天和朋友聚会聊起这事，小红说道：不对呀，前几天我在医院也拿了阿司匹林药，才 14 块钱呢。小花说：奇怪了，我买的是 16 块钱。小杰回应：怎么我买的是 18 块。怎么这药这么多个价格。小明 Google 搜了一下，发现价格跟社保有关，几个人便发现，原来他们都是“不同人”：小明没有社保，小红社保是一档，小花社保是二挡，小杰社保是三挡。（假设社保一档打 7 折，社保二挡打 8 折，社保三挡打 9 折，虚拟的哈） 发现了这秘密后，作为和 IT 工作相关的人，便讨论起医院系统具体实现是怎么实现的。小红说：这很简单呢，药品给不同人提供不同的价格。代码如下：

class Medicine {  
 private String name;  
 private BigDecimal price;  
  
 public Medicine(String name, BigDecimal price) {  
 this.name = name;  
 this.price = price;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public BigDecimal getPrice() {  
 return price;  
 }  
   
 public BigDecimal getPrice1() {  
 return price.multiply(new BigDecimal(0.7));  
 }  
   
 public BigDecimal getPrice2() {  
 return price.multiply(new BigDecimal(0.8));  
 }  
   
 public BigDecimal getPrice3() {  
 return price.multiply(new BigDecimal(0.9));  
 }  
  
 public void setPrice(BigDecimal price) {  
 this.price = price;  
 }  
}

小花说：药片本身的价格是不会变的，只是给不同人不同价格，所以可以在病人获取价钱的时候去区分。代码如下：

class Patient implements IPatient{  
  
 private String name;  
 private int level;  
  
 public Patient(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public BigDecimal pay(Medicine medicines) {  
 if (level == 1) {  
 return medicines.getPrice().multiply(new BigDecimal(0.7));  
 } else if (level == 2) {  
 return medicines.getPrice().multiply(new BigDecimal(0.8));  
 } else if (level == 3) {  
 return medicines.getPrice().multiply(new BigDecimal(0.9));  
 }  
 return medicines.getPrice();  
 }  
  
 @Override  
 public String getName() {  
 return name;  
 }  
  
}

小杰陷入了沉思。。。 小明发话：你们说的方法都可以实现，但是总感觉不对劲，如果以后有社保四挡，还是要修改原来的代码，前 2 天设计模式老师讲的开闭原则忘记了么？里面说要对扩展开放，对修改封闭。我觉得这个药片价格是因为我们人而变的，那是不是我们可以把没社保的归为一类人，一档社保的也为一类，以此类推。我觉得这样实现更好，增加多 3 类病人，分别是一档社保、二挡社保、三挡社保。代码如下：

class OneLevelSocialSecurityPatient implements IPatient {  
  
 private String name;  
  
 public OneLevelSocialSecurityPatient(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public BigDecimal pay(Medicine medicine) {  
 return medicine.getPrice().multiply(new BigDecimal(0.7));  
 }  
  
 @Override  
 public String getName() {  
 return this.name;  
 }  
}  
  
class TwoLevelSocialSecurityPatient implements IPatient {  
  
 private String name;  
  
 public TwoLevelSocialSecurityPatient(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public BigDecimal pay(Medicine medicine) {  
 return medicine.getPrice().multiply(new BigDecimal("0.8"));  
 }  
  
 @Override  
 public String getName() {  
 return this.name;  
 }  
}  
  
class ThreeLevelSocialSecurityPatient implements IPatient {  
  
 private String name;  
  
 public ThreeLevelSocialSecurityPatient(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public BigDecimal pay(Medicine medicine) {  
 return medicine.getPrice().multiply(new BigDecimal("0.9"));  
 }  
  
 @Override  
 public String getName() {  
 return this.name;  
 }  
}  
  
// 测试代码  
public static void main(String[] args) {  
 Hospital hospital = new Hospital();  
 IPatient xiaoMing = new Patient("小明");  
 hospital.sellMedicine(xiaoMing);  
  
 IPatient xiaoHong = new OneLevelSocialSecurityPatient("小红");  
 hospital.sellMedicine(xiaoHong);  
  
 IPatient xiaoHua = new TwoLevelSocialSecurityPatient("小花");  
 hospital.sellMedicine(xiaoHua);  
  
 IPatient xiaoJie = new ThreeLevelSocialSecurityPatient("小杰");  
 hospital.sellMedicine(xiaoJie);  
}

看了他们的对话和代码，是不是能知道哪种方式更好了？对于小红来说，她没理清价格变化的原因，价格变化不在于药片；小花理清了，但是实现方式差了点，以后如果新增了四挡社保，她的实现要修改原有的代码，不符合开闭原则；小明的方法就符合开闭原则，如果新增四挡社保人员，他的方法只需要再额外扩展一个四挡社保人员就可以，不用动用其他代码。

用了这个大家可能不太喜欢的看病的场景来描述这个开闭原则，不要忌讳哈，希望大家都健健康康，远离医院。

### 1.6.3. 总结

重申一下：**对扩展开放，对修改封闭**。如果有同学经常看一些开源框架源码就会发现，有很多很多抽象类和接口，debug 进去很绕，其实这些抽象类和接口很多都是为了扩展用，因为作为开源框架，不得不实现各种可想象到的方案，而这些都基于开闭原则来实现的。以后有机会也可以写一下源码的文章分享给大家。

# 2. 五大创建型模式

## 2.1. 单例模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 2.1.1. 简介

**姓名**：单例模式

**英文名**：Singleton Pattern

**价值观**：我的生活我主宰（只允许自己实例化，不愿意被其他对象实例化）

**个人介绍**：

Ensure a class has only one instance, and provide a global point of access to it.（确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。） (来自《设计模式之禅》)

这里的关注点有 3 个，分别是： 1. 只有一个实例 2. 自行实例化（也就是主动实例化） 3. 向整个系统提供这个实例

### 2.1.2. 你要的故事

我们脑洞大开来用一个故事讲解一番。

小明家里有一辆小汽车，具体什么牌子就不知道了，咱也不关注，反正他家里就这么一辆车，小明比较懒，只要一出门都会开车，例如去旅游、去学校、去聚会都会开车去。下面模拟小明出去的场景。

class Car {  
 public void run() {  
 System.out.println("走。。。。");  
 }  
}  
  
  
class XiaoMing {  
 public Car travel() {  
 System.out.println("小明去旅游");  
 Car car = new Car();  
 car.run();  
 return car;  
 }  
  
 public Car goToSchool() {  
 System.out.println("小明去学校");  
 Car car = new Car();  
 car.run();  
 return car;  
 }  
  
 public Car getTogether() {  
 System.out.println("小明参加聚会");  
 Car car = new Car();  
 car.run();  
 return car;  
 }  
}  
  
public class SingletonErrorTest {  
  
 public static void main(String[] args) {  
 XiaoMing xiaoMing = new XiaoMing();  
 Car car1 = xiaoMing.travel();  
 Car car2 = xiaoMing.goToSchool();  
 Car car3 = xiaoMing.getTogether();  
 }  
  
}

上面小汽车只有一个方法，就是走。小明去旅游、去学校、参加聚会都开着他唯一的一辆汽车车去。是不是有人有疑问？为什么每个方法都返回 Car 对象？其实只是想在下面做一次检查，检查小明去旅游、去学校和参加聚会的车是不是同一辆。下面是检查代码：

System.out.println("car1 == car2 ? " + (car1 == car2));  
System.out.println("car2 == car3 ? " + (car2 == car3));

最终结果是啥？很明显是 2 个 false。小明去旅游、去学校和参加聚会的车都不相同，小明不是只有 1 辆车？关键在于 Car car = new Car(); 这一句代码，其实这一句是创建一辆车，每次都重新创建一辆。那应该怎么实现小明只有一辆车呢？这时候就引入了**单例模式**。

上面我们说到了单例模式需要具备的 3 个点：**只有 1 个实例**，很显然，上面的代码不止 1 个实例，而是有 3 个 Car 实例；**自行实例化**，Car 本身没有主动实例化，而是在小明需要用到的时候才实例化；**向整个系统提供这个实例**，因为 Car 没有主动实例化，所以它没法向外部暴露提供自己出来。

我们的代码完全不符合单例模式的要求。我们要通过修改，使之符合单例模式的 3 个要点。首先需要实现的是第 2 点，把 Car 实例化从小明转为 Car 本身，如下代码

class Car1{  
  
 private static Car1 car1 = new Car1();  
  
 private Car1() {  
  
 }  
  
 public void run(){  
 System.out.println("走。。。。");  
 }  
}

上面代码使用 private 修饰构造方法，使得 Car1 不能被其他使用方实例化，通过 Car1 car1 = new Car1(); 主动实例化自己。

接下来再实现第 3 点，向整个系统暴露这个实例，也就是暴露它自己。每个使用方都调用 Car1.getInstance() 方法来获取实例。

class Car1{  
  
 private static Car1 car1 = new Car1();  
  
 public static Car1 getInstance() {  
 return car1;  
 }  
   
 private Car1() {  
  
 }  
  
 public void run(){  
 System.out.println("走。。。。");  
 }  
}

上面代码就实现了单例模式的 2 和 3 要点，第 1 要点要怎么实现呢？告诉你，不用实现，只要满足了 2 和 3 要点就可以，第 1 要点是用来检验是否是单例模式的好思路。我们检验一下

class Car1{  
  
 private static Car1 car1 = new Car1();  
  
 public static Car1 getInstance() {  
 return car1;  
 }  
  
 private Car1() {  
  
 }  
  
 public void run(){  
 System.out.println("走。。。。");  
 }  
}  
  
class XiaoMing1 {  
 public Car1 travel() {  
 System.out.println("小明去旅游");  
 Car1 car = Car1.getInstance();  
 car.run();  
 return car;  
 }  
  
 public Car1 goToSchool() {  
 System.out.println("小明去学校");  
 Car1 car = Car1.getInstance();  
 car.run();  
 return car;  
 }  
  
 public Car1 getTogether() {  
 System.out.println("小明参加聚会");  
 Car1 car = Car1.getInstance();  
 car.run();  
 return car;  
 }  
}  
  
public class SingletonRightHungryTest {  
  
 public static void main(String[] args) {  
 XiaoMing1 xiaoMing1 = new XiaoMing1();  
 Car1 car1 = xiaoMing1.travel();  
 Car1 car2 = xiaoMing1.goToSchool();  
 Car1 car3 = xiaoMing1.getTogether();  
  
 System.out.println("car1 == car2 ? " + (car1 == car2));  
 System.out.println("car2 == car3 ? " + (car2 == car3));  
 }  
  
}

上面代码最后两行打印出来的结果是啥？是我们想要的：2 个 true。说明小明这几次外出开的车都是同一辆。这是最简单的单例模式的实现方式，我们经常称作**饿汉式单例模式**。为什么起这么古怪的名字呢？其实和对应的**懒汉式单例模式**有关，这是 2 个实现方式的差别，饿汉式单例模式实现方式在类加载到内存的时候，就创建好对象了，而懒汉式则是在第一次使用的时候才创建对象，也就是把创建对象的时机从加载延迟到第一次使用，所以才有懒饿之分。

下面我们来看怎么实现懒汉式单例模式。先描述一下场景：小明还没有汽车，他也不知道什么时候要买汽车，突然某一天，他想去旅游，觉得是时候买辆车了，然后他就买车去旅游了，旅游回来又开车去学校和参加聚会。

class Car2{  
  
 private static Car2 car2;  
  
 public static synchronized Car2 getInstance() {  
 if (null == car2) {  
 System.out.println("买车啦。。。");  
 car2 = new Car2();  
 }  
 return car2;  
 }  
  
 private Car2() {  
  
 }  
  
 public void run(){  
 System.out.println("走。。。。");  
 }  
}  
  
class XiaoMing2  
{  
 public Car2 travel() {  
 System.out.println("小明去旅游");  
 Car2 car = Car2.getInstance();  
 car.run();  
 return car;  
 }  
  
 public Car2 goToSchool() {  
 System.out.println("小明去学校");  
 Car2 car = Car2.getInstance();  
 car.run();  
 return car;  
 }  
  
 public Car2 getTogether() {  
 System.out.println("小明参加聚会");  
 Car2 car = Car2.getInstance();  
 car.run();  
 return car;  
 }  
}  
  
public class SingletonRightLazyTest {  
  
 public static void main(String[] args) {  
 XiaoMing2 xiaoMing2 = new XiaoMing2();  
 Car2 car1 = xiaoMing2.travel();  
 Car2 car2 = xiaoMing2.goToSchool();  
 Car2 car3 = xiaoMing2.getTogether();  
  
 System.out.println("car1 == car2 ? " + (car1 == car2));  
 System.out.println("car2 == car3 ? " + (car2 == car3));  
 }  
  
}  
  
小明去旅游  
买车啦。。。  
走。。。。  
小明去学校  
走。。。。  
小明参加聚会  
走。。。。  
car1 == car2 ? true  
car2 == car3 ? true

上面附带了打印出来的结果，小明要去旅游的时候，才去买车。这就是**懒汉式单例模式**的实现方式。

要注意懒汉式单例模式有个很关键的一点就是 getInstance() 方法带上了 synchronized，这个是为什么呢？

首先得了解关键字 synchronized 的作用是什么：用于修饰执行方法同步，也就是说多线程并发的情况下，在一个时间点，只允许一个线程执行这个方法。

不加上这个会有什么结果？在多线程并发情况下，如果有 2 个线程同时执行到 if(null == car2)，那么都判断为 true，这时 2 个线程都会执行 car2 = new Car2()，这样子就不是单例了。

### 2.1.3. 总结

单例模式可以说是设计模式中最简单的一个，也是在工作中很多场景下经常用到的，比如：项目的配置文件加载、各种工具类等等。我们对于单例模式最重要的一点就是要考虑**多线程并发**，没有考虑这点就容易引发单例对象不单例的情况。而单例给我们带来最大的好处就是**节约内存**。

上面实现的两种方法是单例模式中最最最简单的 2 种实现，相信也是用得最多的实现方式。网上有不少网友分享了单例模式的很多种实现方法，大家也可以去了解，在了解之前务必已经搞懂文中这 2 种最简单的实现方式，不然会头晕的。

## 2.2. 工厂方法

![](data:image/jpeg;charset=UTF-8;base64,)

### 2.2.1. 简介

**姓名**：工厂方法

**英文名**：Factory method Pattern

**价值观**：扩展是我的专属

**个人介绍**：

Define an interface for creating an object,but let subclasses decide which class to instantiate.Factory Method lets a class defer instantiation to subclasses. (定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。) (来自《设计模式之禅》)

### 2.2.2. 你要的故事

还记得上一篇 [单例模式](https://mp.weixin.qq.com/s/aJSf8yESPeex78S6W0FWWw) 中的故事么？小明开着汽车去旅游、去学校、去聚会。这一次还是延续小明的故事，一个故事能讲 2 个设计模式，不容易呀。。。（每次想故事都想破脑袋，每一篇文章至少有 3 个故事从脑子里闪过，但最终留下的只有一个适合，为了就是能比较清晰简单的说明设计模式中的关键要点。）

#### 2.2.2.1. 简单工厂

小明家里以前不算很富裕，但是还是有一个不错的车库，什么汽车、摩托车、自行车啥的都放在这个车库里。小明每次要出去，都会到车库里面挑合适的车出发。比如，小明最近期末考试了，骑摩托车去学校考试，考完试之后，小明就准备去旅游，这次决定自驾游，开着自己家的小汽车去。这个场景我们用代码描述下。

public class SimpleFactoryTest {  
  
 public static void main(String[] args) {  
 XiaoMing xiaoMing = new XiaoMing();  
 // 小明骑摩托车去学校  
 IVehicle motorcycle = GarageFactory.getVehicle("motorcycle");  
 xiaoMing.goToSchool(motorcycle);  
  
 // 小明开汽车去旅游  
 IVehicle car = GarageFactory.getVehicle("car");  
 xiaoMing.travel(car);  
 }  
  
}  
  
/\*\*  
 \* 车库  
 \*/  
class GarageFactory {  
  
 public static IVehicle getVehicle(String type) {  
 if ("car".equals(type)) {  
 return new Car();  
 } else if ("motorcycle".equals(type)) {  
 return new Motorcycle();  
 }  
 throw new IllegalArgumentException("请输入车类型");  
 }  
  
}  
  
/\*\*  
 \* 交通工具  
 \*/  
interface IVehicle {  
 void run();  
}  
  
/\*\*  
 \* 汽车  
 \*/  
class Car implements IVehicle {  
  
 @Override  
 public void run() {  
 System.out.println("开汽车去。。。。");  
 }  
}  
  
/\*\*  
 \* 摩托车  
 \*/  
class Motorcycle implements IVehicle {  
  
 @Override  
 public void run() {  
 System.out.println("骑摩托车去。。。。");  
 }  
}  
  
  
class XiaoMing {  
  
 public void goToSchool(IVehicle vehicle) {  
 System.out.println("小明去学校");  
 vehicle.run();  
 }  
  
 public void travel(IVehicle vehicle) {  
 System.out.println("小明去旅游");  
 vehicle.run();  
 }  
  
}

上面代码看懂了么？ 小明家里有一个车库 GarageFactory，里面放着汽车 Car 和摩托车 Motorcycle，小明要出去的时候，就到车库选择车，通过传递参数给 GarageFactory.getVehicle()，指明要什么车，然后小明就骑着车出发了。

这个代码真正的术语叫：**简单工厂模式**（Simple Factory Pattern），也叫做**静态工厂模式**。它是工厂方法中的一个实现方式，从字面理解就可以知道，它是最简单的工厂方法实现方式。它有一点点小缺陷，就是**扩展性不够好**，在上面代码中，小明只能骑摩托车或者开汽车，如果小明要骑单车出去呢？势必得在 GarageFactory 中添加 if 是自行车的逻辑。这违反了哪条规则了？是不是那个**允许扩展，拒绝修改的**[**开闭原则**](https://mp.weixin.qq.com/s/szBP4A1TFQL1RVZvXEknRA)？

不是说简单工厂这种实现方式不好，而是扩展性不够，在平时的开发中，简单工厂模式也用得不少。在这个小明家里车不多的情况下，用一个车库也是合适的。

#### 2.2.2.2. 工厂方法

小明老爸近几年赚了不少，车迷的两父子一直买车，家里的车越来越多，这时候，他们决定多建几个车库，按车类型放置。比如，有一个汽车库，一个摩托车库。这时候小明要开汽车就去汽车库，要骑摩托车就去摩托车库。代码实现如下。

public class FactoryMethodTest {  
  
 public static void main(String[] args) {  
 XiaoMing xiaoMing = new XiaoMing();  
 // 小明骑摩托车去学校  
 VehicleGarage motorcycleGarage = new MotorcycleGarage();  
 IVehicle motorcycle = motorcycleGarage.getVehicle();  
 xiaoMing.goToSchool(motorcycle);  
  
 // 小明开汽车去旅游  
 VehicleGarage carGarage = new CarGarage();  
 IVehicle car = carGarage.getVehicle();  
 xiaoMing.travel(car);  
 }  
  
}  
  
interface VehicleGarage {  
 IVehicle getVehicle();  
}  
  
/\*\*  
 \* 汽车车库  
 \*/  
class CarGarage implements VehicleGarage {  
  
 @Override  
 public IVehicle getVehicle() {  
 return new Car();  
 }  
}  
  
/\*\*  
 \* 摩托车车库  
 \*/  
class MotorcycleGarage implements VehicleGarage {  
  
 @Override  
 public IVehicle getVehicle() {  
 return new Motorcycle();  
 }  
}

上面代码重用了简单工厂实现方式的交通接口以及摩托车和汽车的实现类。代码中有 2 个车库，一个是汽车车库 CarGarage，一个是摩托车库 MotorcycleGarage。如果小明要骑自行车，只需要建一个自行车车库，完全不用去修改汽车车库或者摩托车车库，就非常符合开闭原则，扩展性大大的提高。

### 2.2.3. 总结

工厂方法模式可以说在你能想到的开源框架源码中必定会使用的一个设计模式，因为开源框架很重要一点就是要有扩展性，而工厂方法模式恰恰具有**可扩展性**。弄懂了工厂方法模式，以后看开源代码就很得心应手啦。

## 2.3. 抽象工厂

![](data:image/jpeg;charset=UTF-8;base64,)

### 2.3.1. 简介

**姓名** ：抽象工厂

**英文名** ：Abstract Factory Pattern

**价值观** ：不管你有多少产品，给我就是了

**个人介绍** ：

Provide an interface for creating families of related or dependent objects without specifying their concrete classes. (为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。) （来自《设计模式之禅》）

今天讲的是抽象工厂模式，小伙伴可能有疑问，抽象工厂和工厂方法之间都有工厂，那肯定是有什么联系的，具体是什么关系呢？简单的说：**工厂方法是在解决一个产品多个层级方面的事情；而抽象工厂致力于解决多个产品多个层级方面的事情**。举个例子：汽车是由很多零件组成的，比如引擎、轮胎、方向盘等等。现在如果我们是轮胎生产方，要生产宝马轮胎和奔驰轮胎，要用工厂方法还是抽象工厂实现呢？答案是：工厂方法。轮胎是一个产品，宝马轮胎和奔驰轮胎是 2 个不同层级的轮胎，所以用工厂方法解决就足够。假如现在我们是汽车生产方，要生产宝马汽车和奔驰汽车，汽车又包含轮胎和方向盘等等，要用哪个来实现？既然是上面的是工厂方法，那这个就用抽象工厂，因为这涉及到多个产品（轮胎、方向盘等等）和 2 个层级（宝马和奔驰）。这里还没有讲抽象工厂的概念就说了工厂方法和抽象方法的区别，是不是有点陌生？嗯，先记住这个概念，分清楚两者的区别。在不同场景使用不同的设计模式。

上面定义中：为创建一组相关或相互依赖的对象提供一个接口。这样子理解这句话，比如上面说的轮胎和方向盘，宝马汽车用的轮胎和方向盘需要都是宝马品牌的，也就是说在安装宝马汽车的轮胎和方向盘的时候，得用宝马生产的轮胎和方向盘，重要的一点是：轮胎和方向盘是互相依赖的，不能在宝马汽车上安装奔驰轮胎和宝马方向盘，因为有这个依赖关系，所以我们需要提供一个额外的接口，来保证宝马汽车使用的轮胎和方向盘都是宝马生产的。这就是抽象工厂干的事情。

### 2.3.2. 你要的故事

上面用汽车安装轮胎和方向盘的例子，那这里为了让大家能深入理解，就不用其他例子了。在一个设计模式讲解的过程中，我觉得用一个案例来讲解可以减少读者的阅读理解成本，为了写设计模式这一系列文章，看了不少设计模式方面的书籍，有些书籍在讲解一个设计模式的时候，用了不止一个例子，读完之后印象不是很深刻。这个系列写完之后，想要的效果是：不需要记住设计模式的定义，把这些故事以及故事对应是讲哪个设计模式都记住了，就真正掌握了这些内容了。

public class AbstractFactoryTest {  
  
 public static void main(String[] args) {  
 // 宝马员工安装轮胎和方向盘  
 AbstractCarFactory bmwCarFacatory = new BMWCarFactory();  
 bmwCarFacatory.installWheel();  
 bmwCarFacatory.installSteeringWheel();  
  
 // 奔驰员工安装轮胎和方向盘  
 AbstractCarFactory mercedesCarFacatory = new MercedesCarFacatory();  
 mercedesCarFacatory.installWheel();  
 mercedesCarFacatory.installSteeringWheel();  
 }  
  
}  
  
/\*\*  
 \* 汽车抽象工厂  
 \*/  
interface AbstractCarFactory {  
  
 void installWheel();  
  
 void installSteeringWheel();  
  
}  
  
/\*\*  
 \* 宝马工厂  
 \*/  
class BMWCarFactory implements AbstractCarFactory {  
  
 @Override  
 public void installWheel() {  
 WheelFacatory wheelFacatory = new BMWWheelFacatory();  
 String wheel = wheelFacatory.createWheel();  
 System.out.println("安装轮胎：" + wheel);  
 }  
  
 @Override  
 public void installSteeringWheel() {  
 SteeringWheelFacatory steeringWheelFacatory = new BMWSteeringWheelFacatory();  
 String steeringWheel = steeringWheelFacatory.createSteeringWheel();  
 System.out.println("安装方向盘：" + steeringWheel);  
 }  
}  
  
/\*\*  
 \* 奔驰工厂  
 \*/  
class MercedesCarFacatory implements AbstractCarFactory {  
  
 @Override  
 public void installWheel() {  
 WheelFacatory wheelFacatory = new MercedesWheelFacatory();  
 String wheel = wheelFacatory.createWheel();  
 System.out.println("安装轮胎：" + wheel);  
 }  
  
 @Override  
 public void installSteeringWheel() {  
 SteeringWheelFacatory steeringWheelFacatory = new MercedesSteeringWheelFacatory();  
 String steeringWheel = steeringWheelFacatory.createSteeringWheel();  
 System.out.println("安装方向盘：" + steeringWheel);  
 }  
}  
  
/\*\*  
 \* 轮胎工厂  
 \*/  
interface WheelFacatory {  
  
 String createWheel();  
  
}  
  
/\*\*  
 \* 宝马轮胎工厂  
 \*/  
class BMWWheelFacatory implements WheelFacatory {  
  
 @Override  
 public String createWheel() {  
 System.out.println("宝马轮胎工厂生产轮胎");  
 return "宝马轮胎";  
 }  
}  
  
/\*\*  
 \* 奔驰轮胎工厂  
 \*/  
class MercedesWheelFacatory implements WheelFacatory {  
  
 @Override  
 public String createWheel() {  
 System.out.println("奔驰轮胎工厂生产轮胎");  
 return "奔驰轮胎";  
 }  
}  
  
/\*\*  
 \* 方向盘工厂  
 \*/  
interface SteeringWheelFacatory {  
  
 String createSteeringWheel();  
  
}  
  
/\*\*  
 \* 宝马方向盘工厂  
 \*/  
class BMWSteeringWheelFacatory implements SteeringWheelFacatory {  
  
 @Override  
 public String createSteeringWheel() {  
 System.out.println("宝马方向盘工厂生产方向盘");  
 return "宝马方向盘";  
 }  
}  
  
/\*\*  
 \* 奔驰方向盘工厂  
 \*/  
class MercedesSteeringWheelFacatory implements SteeringWheelFacatory {  
  
 @Override  
 public String createSteeringWheel() {  
 System.out.println("奔驰方向盘工厂生产方向盘");  
 return "奔驰方向盘";  
 }  
}

还是和以往一样，思维开拓一下，这里列举的是给汽车安装轮胎和方向盘，汽车不止这些，如果要加个安装引擎呢？要怎么实现？这里我就不写出来了，让小伙伴尝试一下，写出来了就理解抽象模式这个设计模式啦。

### 2.3.3. 总结

简单工厂、工厂方法、抽象工厂这几个工厂相关的设计模式的基本内容都讲完了，这几个模式都是为了解耦，为了可扩展。这里要着重说一下，三者之间没有好坏之分，只有在具体的场景才能发挥它们各自的优势。**在单产品多层级，层级数量不多的情况下，可以使用简单工厂，层级多且需要支持扩展，可以使用工厂方法；在多产品多层级，可以使用抽象工厂。**

## 2.4. 建造者模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 2.4.1. 简介

**姓名** ：建造者模式

**英文名** ：Builder Pattern

**价值观** ：专治丢三落四

**个人介绍** ：

Separate the construction of a complex object from its representation so that the same construction process can create different representations. 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。 （来自《设计模式之禅》）

今天给大家介绍的是建造者模式。建造者模式的使用场景是：创建复杂的对象。什么才能算复杂对象？如果一个对象只需要通过 new XXX() 的方式创建，那就算是一个简单对象；如果需要 new XXX()，并且还要设置很多属性，那这种就可以称为复杂对象了，因为它的构建过程比较复杂。采用建造者模式，可以把这个复杂的构建过程抽离开，使它不依赖创建者。下面我们通过故事来讲解。

### 2.4.2. 你要的故事

还记得小时候刚开始学煮汤，不了解怎么煮，第一次是煮了板栗排骨汤，因为板栗比较难熟，所以是跟排骨一起下锅，然后煮了 40 分钟，加了盐就可以出锅啦。第二次煮了冬瓜排骨汤，不懂得冬瓜容易熟，就和排骨一起下锅，煮了也差不多 40 分钟，下了盐和香菜，发现怎么这汤有点浓，冬瓜咋都不见了（全都煮透了），我妈看到这锅汤，才跟我说冬瓜容易熟，得先熬排骨，再放冬瓜进去煮。那时才发现熬汤特么还有这差异。

上面是背景哈，接下来我们来实现这个煲汤过程，冬瓜排骨汤是先加排骨，熬制 30 分钟，加冬瓜，熬制 18 分钟，加盐加香菜；板栗排骨汤是先加排骨和板栗，熬制 40 分钟，再加盐。这汤都需要加肉、加菜、熬制、加配料。我们使用下面代码实现这个煲冬瓜排骨汤和板栗排骨汤的过程。

public class NoBuilderTest {  
  
 public static void main(String[] args) {  
 // 熬制冬瓜排骨汤  
 DongGuaPaiGuSoup dongGuaPaiGuSoup = new DongGuaPaiGuSoup();  
 // 加排骨  
 dongGuaPaiGuSoup.addMeat();  
 // 熬制 30 分钟  
 dongGuaPaiGuSoup.waitMinute(30);  
 // 加冬瓜  
 dongGuaPaiGuSoup.addVegetables();  
 // 熬制 10 分钟  
 dongGuaPaiGuSoup.waitMinute(10);  
 // 加盐加香菜  
 dongGuaPaiGuSoup.addIngredients();  
  
 // 熬制板栗排骨汤  
 BanLiPaiGuSoup banLiPaiGuSoup = new BanLiPaiGuSoup();  
 // 加排骨  
 banLiPaiGuSoup.addMeat();  
 // 加板栗  
 banLiPaiGuSoup.addVegetables();  
 // 熬制 40 分钟  
 banLiPaiGuSoup.waitMinute(40);  
 // 加盐  
 banLiPaiGuSoup.addIngredients();  
 }  
  
}  
  
/\*\*  
 \* 煲汤接口  
 \*/  
interface Soup {  
  
 /\*\* 加肉 \*/  
 void addMeat();  
 /\*\* 加菜 \*/  
 void addVegetables();  
 /\*\* 熬制 \*/  
 void waitMinute(int minute);  
 /\*\* 加配料 \*/  
 void addIngredients();  
  
}  
  
/\*\*  
 \* 冬瓜排骨汤  
 \*/  
class DongGuaPaiGuSoup implements Soup {  
  
 @Override  
 public void addMeat() {  
 System.out.println("加排骨");  
 }  
  
 @Override  
 public void addVegetables() {  
 System.out.println("加冬瓜");  
 }  
  
 @Override  
 public void waitMinute(int minute) {  
 System.out.println("熬制 " + minute + " 分钟");  
 }  
  
 @Override  
 public void addIngredients() {  
 System.out.println("加盐、加香菜");  
 }  
}  
  
/\*\*  
 \* 板栗排骨汤  
 \*/  
class BanLiPaiGuSoup implements Soup {  
  
 @Override  
 public void addMeat() {  
 System.out.println("加排骨");  
 }  
  
 @Override  
 public void addVegetables() {  
 System.out.println("加板栗");  
 }  
  
 @Override  
 public void waitMinute(int minute) {  
 System.out.println("熬制 " + minute + " 分钟");  
 }  
  
 @Override  
 public void addIngredients() {  
 System.out.println("加盐");  
 }  
}

上面代码简单实现了煲冬瓜排骨汤和板栗排骨汤。煲汤我们要关注的点是：各操作的顺序，是先加肉先煮再加菜，还是肉和菜一起放进锅煮。上面代码中，这个过程是谁控制的？是煲汤的人，所以顺序由煲汤的人决定，甚至有可能忘记放配料啥的，这样子的汤就味道不够好。那怎么去解决这些问题？

我们通过建造者模式可以解决上面的 2 个问题：煲汤顺序问题和忘记加配料这种丢三落四行为。我们将这个煲汤顺序从煲汤者分离开来，让煲汤者只需要决定煲什么汤就好，让建造者来保证煲汤顺序问题和防止漏加配料。

我们用一个 SoupBuilder 来规范化煲汤过程，方法 buildSoup 给实现者提供一个设置煲汤顺序的地方。因为冬瓜排骨汤和板栗排骨汤熬制的过程不一样，所以分别用 DongGuaPaiGuSoupBuilder 和 BanLiPaiGuSoupBuilder 来具体实现冬瓜排骨汤和板栗排骨汤的熬制过程，也就是消除熬制过程和煲汤者的依赖关系。 Director 则相当于一个菜单，提供为熬汤者来选择熬什么汤。具体代码如下所示。

public class BuilderTest {  
  
 public static void main(String[] args) {  
 Director director = new Director();  
 // 熬制冬瓜排骨汤  
 director.buildDongGuaPaiGuSoup();  
 // 熬制板栗排骨汤  
 director.buildBanLiPaiGuSoup();  
 }  
  
}  
  
/\*\*  
 \* 煲汤建造接口  
 \*/  
interface SoupBuilder {  
 void buildSoup();  
 Soup getSoup();  
}  
  
/\*\*  
 \* 冬瓜排骨汤建造者  
 \*/  
class DongGuaPaiGuSoupBuilder implements SoupBuilder {  
  
 private DongGuaPaiGuSoup dongGuaPaiGuSoup = new DongGuaPaiGuSoup();  
  
 @Override  
 public void buildSoup() {  
 // 加排骨  
 dongGuaPaiGuSoup.addMeat();  
 // 熬制 30 分钟  
 dongGuaPaiGuSoup.waitMinute(30);  
 // 加冬瓜  
 dongGuaPaiGuSoup.addVegetables();  
 // 熬制 10 分钟  
 dongGuaPaiGuSoup.waitMinute(10);  
 // 加盐加香菜  
 dongGuaPaiGuSoup.addIngredients();  
 }  
  
 @Override  
 public Soup getSoup() {  
 return dongGuaPaiGuSoup;  
 }  
}  
  
/\*\*  
 \* 板栗排骨汤建造者  
 \*/  
class BanLiPaiGuSoupBuilder implements SoupBuilder {  
  
 BanLiPaiGuSoup banLiPaiGuSoup = new BanLiPaiGuSoup();  
  
 @Override  
 public void buildSoup() {  
 // 加排骨  
 banLiPaiGuSoup.addMeat();  
 // 加板栗  
 banLiPaiGuSoup.addVegetables();  
 // 熬制 40 分钟  
 banLiPaiGuSoup.waitMinute(40);  
 // 加盐  
 banLiPaiGuSoup.addIngredients();  
 }  
  
 @Override  
 public Soup getSoup() {  
 return banLiPaiGuSoup;  
 }  
}  
  
/\*\*  
 \* 生产方  
 \*/  
class Director {  
 private DongGuaPaiGuSoupBuilder dongGuaPaiGuSoupBuilder = new DongGuaPaiGuSoupBuilder();  
 private BanLiPaiGuSoupBuilder banLiPaiGuSoupBuilder = new BanLiPaiGuSoupBuilder();  
  
 /\*\*  
 \* 熬制冬瓜排骨汤  
 \*/  
 public DongGuaPaiGuSoup buildDongGuaPaiGuSoup() {  
 dongGuaPaiGuSoupBuilder.buildSoup();  
 return (DongGuaPaiGuSoup) dongGuaPaiGuSoupBuilder.getSoup();  
 }  
  
 /\*\*  
 \* 熬制板栗排骨汤  
 \*/  
 public BanLiPaiGuSoup buildBanLiPaiGuSoup() {  
 banLiPaiGuSoupBuilder.buildSoup();  
 return (BanLiPaiGuSoup) banLiPaiGuSoupBuilder.getSoup();  
 }  
  
}

通过用建造者实现，是不是保证了熬制汤的顺序并且一定会加够料？感受一下其中的奥秘吧。

### 2.4.3. 总结

通过建造者模式，可以把本来强依赖的东西解绑，不仅仅解决依赖问题，还提高了封装性，让使用者不用明白内部的细节，用上面的例子说就熬汤不用关心怎么熬制的过程，就像我们想喝冬瓜排骨汤，告诉妈妈，妈妈熬制，我们并不知道是怎么熬制的。

## 2.5. 原型模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 2.5.1. 简介

**姓名** ：原型模式

**英文名** ：Prototype Pattern

**价值观** ：效率第一

**个人介绍** ：

Specify the kinds of objects to create using a prototypical instance,and create new objects by copying this prototype. 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。 （来自《设计模式之禅》）

又到了一个系列的最后一篇文章了，今天是创建型模式的最后一篇。什么是创建型模式呢？创建型模式是对类的实例化过程进行抽象，使对象的创建和使用分离，从而使代码更加灵活。

我们平时使用最多的一种创建对象方式就是 new ABC()，直接通过构造方法来创建一个对象。通过原型模式来创建对象则不用调用构造方法，就可以创建一个对象。下面来揭开它的面纱。

### 2.5.2. 你要的故事

前几天有出版社的老师邀请写书，鉴于深知自己水平还不足以出书，所以没有合作，还在努力学习，以后有能力有机会再考虑这方面的事情。

今天的故事就从出书讲起。我们知道一本新书发版的时候，会复印很多册，如果销售得好，会有很多个印刷版本。我们来了解复印一批书籍这个过程是怎么实现的。小明写下了下面这段代码。

public class NoPrototypeTest {  
  
 public static void main(String[] args) {  
 for (int i = 1; i <= 10; i ++) {  
 Book book = new Book("娱乐至死", "尼尔波兹曼", "社会科学", "XXXX");  
 System.out.println("复印书籍：" + book.getName() + "，第 " + i + " 本");  
 }  
 }  
  
}  
  
class Book {  
 private String name;  
 private String author;  
 private String type;  
 private String content;  
  
 public Book(String name, String author, String type, String content) {  
 this.name = name;  
 this.author = author;  
 this.type = type;  
 this.content = content;  
 System.out.println("实例化书籍：" + this.name);  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getAuthor() {  
 return author;  
 }  
  
 public void setAuthor(String author) {  
 this.author = author;  
 }  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
}  
  
// 打印结果：  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 1 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 2 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 3 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 4 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 5 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 6 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 7 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 8 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 9 本  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 10 本

上面小明的代码复印了 10 本《娱乐至死》，代码逻辑没有问题，有个问题就是复印一本就实例化一次书籍，这个实例化可以减少么？使用原型模式可以实现。小明根据这些提示，重新修改了代码。

public class PrototypeTest {  
  
 public static void main(String[] args) {  
 Book2 book1 = new ConcreteBook("娱乐至死", "尼尔波兹曼", "社会科学", "XXXX");  
 System.out.println("复印书籍：" + book1.getName() + "，第 " + 1 + " 本");  
 for (int i = 2; i <= 10; i ++) {  
 Book2 book2 = (Book2) book1.clone();  
 System.out.println("复印书籍：" + book2.getName() + "，第 " + i + " 本");  
 }  
  
  
 }  
  
}  
  
/\*\*  
 \* 抽象类  
 \*/  
abstract class Book2 implements Cloneable {  
  
 private String name;  
 private String author;  
 private String type;  
 private String content;  
  
 public Book2(String name, String author, String type, String content) {  
 this.name = name;  
 this.author = author;  
 this.type = type;  
 this.content = content;  
 System.out.println("实例化书籍：" + this.name);  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getAuthor() {  
 return author;  
 }  
  
 public void setAuthor(String author) {  
 this.author = author;  
 }  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
  
 @Override  
 protected Object clone() {  
 try {  
 return super.clone();  
 } catch (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 return null;  
 }  
}  
  
/\*\*  
 \* 具体类  
 \*/  
class ConcreteBook extends Book2 {  
  
 public ConcreteBook(String name, String author, String type, String content) {  
 super(name, author, type, content);  
 }  
}  
  
打印结果：  
实例化书籍：娱乐至死  
复印书籍：娱乐至死，第 1 本  
复印书籍：娱乐至死，第 2 本  
复印书籍：娱乐至死，第 3 本  
复印书籍：娱乐至死，第 4 本  
复印书籍：娱乐至死，第 5 本  
复印书籍：娱乐至死，第 6 本  
复印书籍：娱乐至死，第 7 本  
复印书籍：娱乐至死，第 8 本  
复印书籍：娱乐至死，第 9 本  
复印书籍：娱乐至死，第 10 本

看，打印结果和第一次实现的结果完全不一样，这一次只实例化了一次，后面复印的书籍都没有实例化。我们看看代码的变化，代码中最最主要的就是 Book2 实现了 Cloneable 接口，这个接口有个 clone() 方法，通过实现这个方法，可以实现对象的拷贝，就是不用调用构造方法，直接通过对内存的拷贝来创建一个新的对象。这就是原型模式的实现方式，**通过原型模式可以提高创建对象的效率**。

### 2.5.3. 总结

通过原型模式，绕过构造方法创建对象，利用内存直接拷贝对象，提高对象的创建性效率。在有大量的对象创建或者类初始化消耗多资源的场景下可以利用原型模式来优化。当然在实现的过程中，要注意**浅拷贝与深拷贝**的问题，防止写出 bug，文章主要介绍原型模式，就不详细说这个问题了，留给大家去扩展了解。

# 3. 十一大行为型模式

## 3.1. 模板方法

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.1.1. 简介

**姓名** ：模板方法

**英文名** ：Template Method Pattern

**价值观** ：在我的掌控下，任由你发挥

**个人介绍** ：

Define the skeleton of an algorithm in an operation,deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. 定义一个操作中的算法的框架，而将一些步骤延迟到子类中。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。 （来自《设计模式之禅》）

解释一下上面的介绍，意思是由父类来定义框架，让子类来具体实现。

### 3.1.2. 你要的故事

刚过完春节，大家都买新鞋了么？今天要讲的故事和鞋子有关。一双鞋子从表面来看，由鞋底、鞋垫、鞋面、鞋带组成，同一系列的鞋子这几个部分都是一样的，用同样的材料做出来，不同系列的鞋子就大相径庭了。根据模板方法模式，组装一双鞋子的制造过程可以归并为固定的框架，至于用什么材料，那由每个系列的鞋子去具体实现。我们先看定义组装鞋子的框架代码。

/\*\*  
 \* 定义鞋子制造的工序框架  
 \*/  
abstract class ShoeInstallTemplate {  
  
 public abstract void installSole();  
 public abstract void installInsole();  
 public abstract void installVamp();  
 public abstract void installShoelace();  
  
 public void installShot(){  
 System.out.println("组装一双鞋，步骤如下：");  
 // 组装鞋底  
 installSole();  
 // 组装鞋垫  
 installInsole();  
 // 组装鞋面  
 installVamp();  
 // 组装鞋带  
 installShoelace();  
 }  
  
}

定义了一个组装鞋子框架的抽象类 ShoeInstallTemplate，里面有 4 个工序未具体实现，由鞋子制造商去实现，因为只有鞋子制造商才知道鞋子要用什么材料来做。 下面举 2 个比较出名的鞋子：Adidas 的 Boost 系列和 Nike 的 Jordan 系列。下面分别实现这 2 个系列鞋子的制造代码。

/\*\*  
 \* Adidas Boost 鞋制造  
 \*/  
class AdidasBoostShoeInstall extends ShoeInstallTemplate {  
 @Override  
 public void installSole() {  
 System.out.println("组装白色 Boost 鞋底");  
 }  
  
 @Override  
 public void installInsole() {  
 System.out.println("组装黑色 Boost 鞋垫");  
 }  
  
 @Override  
 public void installVamp() {  
 System.out.println("组装黑色 Boost 鞋面");  
 }  
  
 @Override  
 public void installShoelace() {  
 System.out.println("组装黑色 Boost 鞋带");  
 }  
}  
  
/\*\*  
 \* Nike Jordan 鞋制造  
 \*/  
class NikeJordanShoeInstall extends ShoeInstallTemplate {  
  
 @Override  
 public void installSole() {  
 System.out.println("组装黑色 Jordan 鞋底");  
 }  
  
 @Override  
 public void installInsole() {  
 System.out.println("组装黑色 Jordan 鞋垫");  
 }  
  
 @Override  
 public void installVamp() {  
 System.out.println("组装红色 Jordan 鞋面");  
 }  
  
 @Override  
 public void installShoelace() {  
 System.out.println("组装红色 Jordan 鞋带");  
 }  
}

实现了制造商制造鞋子的代码之后，我们通过代码测试怎么制造 Boost 和 Jordan 鞋子。

public class TemplateMethodTest {  
  
 public static void main(String[] args) {  
 ShoeInstallTemplate adidasBoost = new AdidasBoostShoeInstall();  
 adidasBoost.installShot();  
  
 ShoeInstallTemplate nikeJordan = new NikeJordanShoeInstall();  
 nikeJordan.installShot();  
 }  
  
}  
  
打印结果：  
组装一双鞋，步骤如下：  
组装白色 Boost 鞋底  
组装黑色 Boost 鞋垫  
组装黑色 Boost 鞋面  
组装黑色 Boost 鞋带  
  
组装一双鞋，步骤如下：  
组装黑色 Jordan 鞋底  
组装黑色 Jordan 鞋垫  
组装红色 Jordan 鞋面  
组装红色 Jordan 鞋带

模板方法模式就这么简单。是不是掌握了？

### 3.1.3. 总结

模板方法是一个比较实用的模式，为什么说实用呢？举个现实的例子，Java 能有如今的发展，离不开各大开源框架，比如 Dubbo，有看过源码的朋友就知道，里面大量代码运用了模板方法设计模式，为什么 Dubbo 可以支持很多种注册中心？其实本质就是用了模板方法设计模式，使得可以扩展多种注册中心。掌握好模板方法，对读源码有非常大的帮助，很多人包括我在内，在刚开始阅读源码的时候，有相当长的一段时间怀疑人生，怎么这些代码那么绕？调来调去的。当你了解了常用的设计模式之后，看源代码就可以直截了当的知道是用什么设计模式，为什么用这个设计模式？原来是为了什么什么。。。有了这层思考，就像有一条线将以前散落在各地的知识点连接起来，成了可以推敲的知识。

## 3.2. 中介者模式

### 3.2.1. 简介

![](data:image/jpeg;charset=UTF-8;base64,)

**姓名** ：中介者模式

**英文名** ：Mediator Pattern

**价值观** ：让你体验中介是无所不能的存在

**个人介绍** ：

Define an object that encapsulates how a set of objects interact.Mediator promotes loose coupling by keeping objects from referring to each other explicitly,and it lets you vary their interaction independently. 用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。 （来自《设计模式之禅》）

### 3.2.2. 你要的故事

看了这小伙子的名字，大家会很直观的想到那些拿了我们半个月租的租房中介同学。在这不讲讲房租中介同学，以后可没机会了。大家现在找房子，不管是买还是租，一登录什么安居客、58同城，是不是有 80% 是经纪人房源，说 80% 还是比较保守的，经历了 4 次找房，发现个人房源越来越少。每个网站都有个选项：经纪人房源。如下图：

![](data:image/png;charset=UTF-8;base64,)

房源

(图片截自：安居客网站)

经纪人就扮演着中介的角色，和本文要讲的中介者模式完全吻合。我们在找房子的时候，经纪人扮演什么角色呢？我们通过个人房源和经纪人房源的租房案例来简单描述经纪人的角色。

#### 3.2.2.1. 个人房源

我们通过个人房源找房子的方式是这样的：在网上找个人房源的房东，然后挨个联系，和房东约定好时间去看房，我们跟房东的关系是一对多的关系。小明就在网上看了个人房源，联系了房东，分别去看了农民房和小区房，用代码表示如下。

public class PersonalTest {  
  
 public static void main(String[] args) {  
 Tenant xiaoMing = new Tenant("小明");  
 xiaoMing.lookAtHouse();  
 }  
  
}  
  
class Tenant {  
 private String name;  
 private XiaoQuFangLandlord xiaoQuFangLandlord2 = new XiaoQuFangLandlord();  
 private NongMinFangLandlord nongMinFangLandlord2 = new NongMinFangLandlord();  
  
 public Tenant(String name) {  
 this.name = name;  
 }  
  
 public void lookAtHouse() {  
 System.out.println(this.name +"想看农民房");  
 nongMinFangLandlord2.supply();  
 System.out.println(this.name + "想看小区房");  
 xiaoQuFangLandlord2.supply();  
 }  
  
}  
  
/\*\*  
 \* 房东  
 \*/  
abstract class Landlord {  
 // 提供房子  
 public abstract void supply();  
}  
  
class XiaoQuFangLandlord extends Landlord {  
  
 @Override  
 public void supply() {  
 System.out.println("小区房的房东提供一间小区房");  
 }  
}  
  
class NongMinFangLandlord extends Landlord {  
  
 @Override  
 public void supply() {  
 System.out.println("农民房的房东提供一间小区房");  
 }  
}  
  
打印结果如下：  
小明想看农民房  
农民房的房东提供一间小区房  
小明想看小区房  
小区房的房东提供一间小区房

小明分别联系小区房的房东和农民房的房东，然后依次去看了农民房和小区房。这样子有个弊端就是小明和房东是强关联的关系，其实小明只是去看一下房，看完不想租就和房东没啥关系了。这个时候经纪人就派上用场了，经纪人的主要任务就是把房子租出去，所以他和房东应该是强关系，直到把房子成功租出去了，才和房东脱离关系，而小明也不用去挨个找房东看房子了，这个职责转给经纪人，小明只需要联系一个人，那就是经纪人，跟他说我要看小区房和农民房，经纪人就带他去看。下面就介绍经纪人房源的方式，也就是本文要讲的中介者模式。

#### 3.2.2.2. 经纪人房源

用经纪人房源找房子，小明就省心很多了，小明就只联系了一个经纪人，跟他描述了自己要的房源：小区房和农民房都可以，经纪人里面和他约定了一个下午的时间，把小明所有想看的房让他看完，最终小明决定租了一间房。看代码。

public class MediatorTest {  
  
 public static void main(String[] args) {  
 System.out.println("小明想要看小区房和农民房");  
 Tenant2 xiaoMing = new Tenant2("小明", Arrays.asList("XiaoQuFang", "NongMinFang"));  
 xiaoMing.lookAtHouse();  
 }  
  
  
}  
  
/\*\*  
 \* 租客  
 \*/  
class Tenant2 {  
 private String name;  
 private List<String> wantTypes;  
  
 private RentingMediator rentingMediator = new RentingMediator();  
  
 public Tenant2(String name, List<String> wantTypes) {  
 this.name = name;  
 this.wantTypes = wantTypes;  
 }  
  
 public void lookAtHouse() {  
 rentingMediator.supplyHouse(wantTypes);  
 }  
  
}  
  
/\*\*  
 \* 中介抽象类  
 \*/  
abstract class Mediator {  
 // 看房  
 public abstract void supplyHouse(List<String> types);  
}  
  
/\*\*  
 \* 租房中介  
 \*/  
class RentingMediator extends Mediator {  
  
 private XiaoQuFangLandlord xiaoQuFangLandlord;  
 private NongMinFangLandlord nongMinFangLandlord;  
  
 public RentingMediator() {  
 xiaoQuFangLandlord = new XiaoQuFangLandlord();  
 nongMinFangLandlord = new NongMinFangLandlord();  
 }  
  
 @Override  
 public void supplyHouse(List<String> types) {  
 System.out.println("经纪人提供了如下房源");  
 if (types.contains("XiaoQuFang")) {  
 xiaoQuFangLandlord.supply();  
 }  
 if (types.contains("NongMinFang")) {  
 nongMinFangLandlord.supply();  
 }  
 }  
}  
  
打印结果：  
小明想要看小区房和农民房  
经纪人提供了如下房源  
小区房的房东提供一间小区房  
农民房的房东提供一间小区房

在代码中，我们可以看到小明和经纪人是一对一关系，经纪人和房东是一对多关系。小明找房经历也轻松多了，只花了一下午就把房子都看了并看中了。这也是中介者模式的优点，**减少了不必要的依赖，降低了类间的耦合**。

### 3.2.3. 总结

**中介者模式通过在互相依赖的对象中间加了一层，让原本强依赖的对象变成弱依赖**。在软件编程中，有一个中介者模式的典型的例子，就是 MVC 框架，也称三层架构，通过 Controller (控制层) 将 Model (业务逻辑层) 和 View (视图层) 的依赖给分离开，协调 Model 和 View 中的数据和界面交互工作。看看你工作中的代码，想想看有没有哪些对象之间的关系特紧密特混乱，考虑是不是可以通过中介者模式来把依赖关系剥离，让代码更清晰。

## 3.3. 命令模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.3.1. 简介

**姓名** ：命令模式

**英文名** ：Command Pattern

**价值观** ：军令如山

**个人介绍** ：

Encapsulate a request as an object,thereby letting you parameterize clients with different requests,queue or log requests,and support undoable operations. 将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。 （来自《设计模式之禅》）

### 3.3.2. 你要的故事

作为一个程序猿，我们每天都在经历着命令模式，技术经理把需求任务分配给工程师开发，有时因为第三方或者其他不可抗拒的因素导致需求停止开发。这种工作模式就是命令模式。好了，开始故事了。小明在 XX 科技公司做一个安静的程序猿，有一天技术经理给他分配了一个任务：新增黑名单，也就是在他们系统的某个模块里面可以手工对电话打黑名单标签的功能。小明接到任务后就立马开发，在开发了 2 天之后，因为战略原因，技术经理大明暂停了这个开发任务，接下来我们通过非命令模式和命令模式 2 种代码实现来体现这个过程。在这个场景中，为了简单，我们假定技术经理大明手下只有小明一个开发人员。

#### 3.3.2.1. 非命令模式

非命令模式也就是不使用命令模式的代码实现。代码中，我们出现了 Developer 开发人，开发同学是接受技术经理传达的任务，技术经理让他开发哪个需求就开发哪个需求，如果项目有问题需要中断，也需要技术经理评估后传达给开发同学，所以 Developer 有 2 个方法，分别是 develop() 开发需求和 suspend() 暂停需求。 Requirement 则为需求类，TechnicalManager1 则为技术经理类，他有一个方法 action()，通过这个方法来指定开发同学开发任务或者暂停任务。

public class NoCommandTest {  
  
 public static void main(String[] args) {  
 Developer xiaoMing = new Developer("小明");  
 Requirement requirement = new Requirement("新增黑名单");  
 TechnicalManager1 technicalManager2 = new TechnicalManager1("大明");  
 technicalManager2.setDeveloper(xiaoMing);  
 technicalManager2.action(requirement, "develop");  
 System.out.println("开发了 2 天，需求变故，需要暂停。。。");  
 technicalManager2.action(requirement, "suspend");  
 }  
  
}  
  
/\*\*  
 \* 开发人员  
 \*/  
class Developer {  
  
 private String name;  
  
 public Developer(String name) {  
 this.name = name;  
 }  
  
 public void develop(Requirement requirement) {  
 System.out.println(this.name + " 开始开发需求：" + requirement.getName());  
 }  
  
 public void suspend(Requirement requirement) {  
 System.out.println(this.name + " 停止开发需求：" + requirement.getName());  
 }  
  
 public String getName() {  
 return name;  
 }  
  
}  
  
/\*\*  
 \* 需求  
 \*/  
class Requirement {  
 private String name;  
  
 public Requirement(String name) {  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
}  
  
/\*\*  
 \* 技术经理  
 \*/  
class TechnicalManager1 {  
  
 private String name;  
  
 private Developer developer;  
  
 public TechnicalManager1(String name) {  
 this.name = name;  
 }  
  
 public void setDeveloper(Developer developer) {  
 this.developer = developer;  
 }  
  
 public void action(Requirement requirement, String type) {  
 if ("develop".equals(type)) {  
 this.developer.develop(requirement);  
 } else if ("suspend".equals(type)) {  
 this.developer.suspend(requirement);  
 }  
 }  
  
}  
  
打印结果：  
小明 开始开发需求：新增黑名单  
开发了 2 天，需求变故，需要暂停。。。  
小明 停止开发需求：新增黑名单

通过代码，我们可以发现技术经理和开发同学是强依赖关系。如果技术经理下达了一个任务，要求小明写一下周报，这时候得怎么写？是不是小明需要一个写周报的方法，大明也需要新增一个处理事务类型？有没有更好的方法让技术经理不需要做任何改变？命令模式就来解决这个问题。

#### 3.3.2.2. 命令模式

在这个例子中，不管大明叫小明做什么事情，其实都是一样的，就是下达任务命令，让小明去执行命令。我们可以利用命令模式把下达任务这个抽象起来，当做父类，下达开发命令、下达暂停命令、下达写周报等等都是不同的子命令。代码如下。

public class CommandTest {  
  
 public static void main(String[] args) {  
 Developer xiaoMing = new Developer("小明");  
 Command developCommand = new DevelopCommand(xiaoMing);  
 Command suspendCommand = new SuspendCommand(xiaoMing);  
 Requirement requirement = new Requirement("新增黑名单");  
 TechnicalManager2 technicalManager = new TechnicalManager2("大明");  
 technicalManager.setCommand(developCommand);  
 technicalManager.action(requirement);  
 System.out.println("开发了 2 天，需求变故，需要暂停。。。");  
 technicalManager.setCommand(suspendCommand);  
 technicalManager.action(requirement);  
  
 }  
  
}  
  
/\*\*  
 \* 命令  
 \*/  
abstract class Command {  
  
 protected Developer developer;  
  
 public Command(Developer developer) {  
 this.developer = developer;  
 }  
  
 public abstract void execute(Requirement requirement);  
}  
  
/\*\*  
 \* 开始开发  
 \*/  
class DevelopCommand extends Command {  
  
 public DevelopCommand(Developer developer) {  
 super(developer);  
 }  
  
 @Override  
 public void execute(Requirement requirement) {  
 this.developer.develop(requirement);  
 }  
}  
  
/\*\*  
 \* 开发中断  
 \*/  
class SuspendCommand extends Command {  
  
 public SuspendCommand(Developer developer) {  
 super(developer);  
 }  
  
 @Override  
 public void execute(Requirement requirement) {  
 this.developer.suspend(requirement);  
 }  
}  
  
/\*\*  
 \* 技术经理  
 \*/  
class TechnicalManager2 {  
  
 private String name;  
 private Command command;  
  
 public TechnicalManager2(String name) {  
 this.name = name;  
 }  
  
 public void action(Requirement requirement) {  
 this.command.execute(requirement);  
 }  
  
 public void setCommand(Command command) {  
 this.command = command;  
 }  
}  
  
打印结果：  
小明 开始开发需求：新增黑名单  
开发了 2 天，需求变故，需要暂停。。。  
小明 停止开发需求：新增黑名单

代码中用 Command 来抽象下达任务，而技术经理 TechnicalManager2 并没有和 Developer 有直接的关系，而是 TechnicalManager2 和 Command 建立的联系，Command 和 Developer 建立了联系。这样子把大明和小明的强依赖关系给剥离开，而新增一个下达写周报的任务也很简单，在 Developer 中新增一个处理写周报的方法，新增一个写周报的 Command 子类，就可以了，TechnicalManager2 如上面所愿不用修改。这就是完整的一个命令模式代码。

### 3.3.3. 总结

从文章中我们就可以看到，利用命令模式能够**进行类的解耦**，让调用者和接受者没有任何关系，也通过对行为的抽象，让新增其他行为变得清晰容易，也就是**可扩展性大大增加**。

## 3.4. 责任链模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.4.1. 简介

**姓名** ：责任链模式

**英文名** ：Chain of Responsibility Pattern

**价值观** ：责任归我

**个人介绍** ： Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.Chain the receiving objects and pass the request along the chain until an object handles it. 使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。 （来自《设计模式之禅》）

### 3.4.2. 你要的故事

快要金三银四了，很多同学蠢蠢欲动想要去外面看看世界，而大家有没有提前了解各大企业的面试流程呢？这里我就给大家科普一下大多数互联网企业的面试流程，正好责任链模式用得上。

在互联网企业中，程序员这个岗位的招聘流程大同小异，而一般至少都会有 3 轮面试，分别是 2 轮技术面和 1 轮 HR 面。而这几轮面试都是层层递进的，最开始面试一般是组长面试，通过之后就是部门领导面试，再通过之后就是 HR 面试，HR 面试通过之后就可以成功拿到 Offer 了。

故事从小明参加某公司的面试开始，某公司的招聘流程就是上面说的 3 轮面试。招聘流程的面试官分别是：第一面是组长老刚，第二面是部门经理老孙，第三面也就是终面是 HR 老刘。为什么说这个场景符合责任链模式呢？首先不管是组长还是部门经理还是 HR，他们都作为面试官，面试官赋予他们的权利是去面试来公司面试的同学，而面试的结果是可传递性的，也就是如果面试通过，就会到下一轮面试，最终成为一条传递链。我们用代码模拟这个过程。

public class ChainOfResponsibilityTest {  
  
 public static void main(String[] args) {  
 Interviewee interviewee = new Interviewee("小明");  
 TeamLeader teamLeader = new TeamLeader("老刚");  
 DepartMentManager departMentManager = new DepartMentManager("老孙");  
 HR hr = new HR("老刘");  
 // 设置面试流程  
 teamLeader.setNextInterviewer(departMentManager);  
 departMentManager.setNextInterviewer(hr);  
 // 开始面试  
 teamLeader.handleInterview(interviewee);  
 }  
  
}  
  
/\*\*  
 \* 面试者  
 \*/  
class Interviewee {  
  
 private String name;  
  
 private boolean teamLeaderOpinion;  
 private boolean departMentManagerOpinion;  
 private boolean hrOpinion;  
  
 public Interviewee(String name) {  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public boolean isTeamLeaderOpinion() {  
 return teamLeaderOpinion;  
 }  
  
 public void setTeamLeaderOpinion(boolean teamLeaderOpinion) {  
 this.teamLeaderOpinion = teamLeaderOpinion;  
 }  
  
 public boolean isDepartMentManagerOpinion() {  
 return departMentManagerOpinion;  
 }  
  
 public void setDepartMentManagerOpinion(boolean departMentManagerOpinion) {  
 this.departMentManagerOpinion = departMentManagerOpinion;  
 }  
  
 public boolean isHrOpinion() {  
 return hrOpinion;  
 }  
  
 public void setHrOpinion(boolean hrOpinion) {  
 this.hrOpinion = hrOpinion;  
 }  
}  
  
/\*\*  
 \* 面试官  
 \*/  
abstract class Interviewer {  
  
 protected String name;  
 protected Interviewer nextInterviewer;  
  
 public Interviewer(String name) {  
 this.name = name;  
 }  
  
 public Interviewer setNextInterviewer(Interviewer nextInterviewer) {  
 this.nextInterviewer = nextInterviewer;  
 return this.nextInterviewer;  
 }  
  
 public abstract void handleInterview(Interviewee interviewee);  
  
}  
  
/\*\*  
 \* 组长  
 \*/  
class TeamLeader extends Interviewer {  
  
 public TeamLeader(String name) {  
 super(name);  
 }  
  
 @Override  
 public void handleInterview(Interviewee interviewee) {  
 System.out.println("组长[" + this.name + "]面试[" + interviewee.getName() + "]同学");  
 interviewee.setTeamLeaderOpinion(new Random().nextBoolean());  
 if (interviewee.isTeamLeaderOpinion()) {  
 System.out.println("[" + interviewee.getName() + "]同学组长轮面试通过");  
 this.nextInterviewer.handleInterview(interviewee);  
 } else {  
 System.out.println("[" + interviewee.getName() + "]同学组长轮面试不通过");  
 }  
 }  
}  
  
/\*\*  
 \* 部门经理  
 \*/  
class DepartMentManager extends Interviewer {  
  
 public DepartMentManager(String name) {  
 super(name);  
 }  
  
 @Override  
 public void handleInterview(Interviewee interviewee) {  
 System.out.println("部门经理[" + this.name + "]面试[" + interviewee.getName() + "]同学");  
 interviewee.setDepartMentManagerOpinion(new Random().nextBoolean());  
 if (interviewee.isDepartMentManagerOpinion()) {  
 System.out.println("[" + interviewee.getName() + "]同学部门经理轮面试通过");  
 this.nextInterviewer.handleInterview(interviewee);  
 } else {  
 System.out.println("[" + interviewee.getName() + "]同学部门经理轮面试不通过");  
 }  
 }  
}  
  
/\*\*  
 \* HR  
 \*/  
class HR extends Interviewer {  
  
 public HR(String name) {  
 super(name);  
 }  
  
 @Override  
 public void handleInterview(Interviewee interviewee) {  
 System.out.println("HR[" + this.name + "]面试[" + interviewee.getName() + "]同学");  
 interviewee.setHrOpinion(new Random().nextBoolean());  
 if (interviewee.isHrOpinion()) {  
 System.out.println("[" + interviewee.getName() + "]同学HR轮面试通过, 恭喜拿到 Offer");  
 } else {  
 System.out.println("[" + interviewee.getName() + "]同学HR轮面试不通过");  
 }  
 }  
}  
  
打印结果：  
组长[老刚]面试[小明]同学  
[小明]同学组长轮面试通过  
部门经理[老孙]面试[小明]同学  
[小明]同学部门经理轮面试通过  
HR[老刘]面试[小明]同学  
[小明]同学HR轮面试通过, 恭喜拿到 Offer

上面代码打印结果是小明通过层层筛选，通过了面试，拿到了 Offer。下面的图来展现这次面试的流程。

![](data:image/jpeg;charset=UTF-8;base64,)

流程图

讲解一下代码，Interviewee 是面试者，对于企业来说这个面试者来面试的过程会有 3 个关键标识，就是 3 位面试官对这位面试者的评价，只有都评价好才能通过面试拿到 Offer，所以 Interviewee 类有 3 位面试官的面试结果。Interviewer 是面时官，企业中面试官不是一个职位，而是一个工作，帮企业招到合适的人才，所以它是一个抽象类，有一个抽象方法就是去面试应聘者，具体面试过程又各面试官实现，而因为这个面试会有结果反馈，结果好的会进入下一轮面试，所以会有下一个面试官的对象引用，责任链模式也就在这里体现。TeamLeader、DepartMentManager、HR 则为公司的不同职位，而这些职位当公司需要招聘时，都需要去充当面试官，所以都继承了 Interviewer。这整个过程就构成了责任链模式代码示例，希望在金三银四各位蠢蠢欲动的朋友们都能闯到最后一关拿下 HR 姐姐。

### 3.4.3. 总结

责任链模式很好的**把处理的逻辑封装起来**，在代码中，我们看到的只是组长面试，但是其实背后隐藏着部门经理和HR的面试。责任链是不是看起来很熟悉？在开发 Java Web 项目的时候是不是有用到过？Filter 过滤器里面就是用责任链模式来实现的。上面代码还用到了另一个模式，不明确指出来了，考验大家学习这些设计模式的效果，知道的同学可以留言说一下。

## 3.5. 策略模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.5.1. 简介

**姓名** ：策略模式

**英文名** ：Strategy Pattern

**价值观** ：集计谋于一身

**个人介绍** ： Define a family of algorithms,encapsulate each one,and make them interchangeable. 定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。 （来自《设计模式之禅》）

### 3.5.2. 你要的故事

先看一张拍得不好看的图片

![](data:image/jpeg;charset=UTF-8;base64,)

洗衣机面板

每天上完班回到家第一件事情是干啥？有人一进门就躺在沙发上闭目养神、有人一进门躺在沙发上玩手机、有人一进门就陪自己的小宠物玩等等。而我进门第一件事就是洗澡，洗完澡很容易就把一整天的疲惫感给消除掉，然后就可以开始美好的下班时光。现实没那么美好，洗完澡后还要洗衣服，大学手洗了 4 年的衣服，一出来工作，宿舍第一必需品就是洗衣机。细看洗衣机，有很多种洗衣类型，比如：标准、大物、快洗、轻柔。洗衣类型的区别在于洗衣服的过程不一样，洗衣过程包括有浸泡、洗涤、漂洗、脱水，还有洗衣服的时间也不一样。细想可以发现这 4 种洗衣类型其实是洗衣服的 4 种不同的策略，也即是 4 种不同的算法。根据这个思路，我们可以用代码实现它，定义一个接口 WashingStrategy 定义洗衣服类型，而这些类型都有各自的洗衣过程，比如标准洗衣类型就包括浸泡、洗涤、漂洗、脱水，而快洗则只包括洗涤、漂洗、脱水。而我们洗衣服则需要选择某个洗衣类型后，洗衣机就开始工作了。过程如下代码所示。

public class StrategyTest {   
  
 public static void main(String[] args) {  
 WashingStrategy washingStrategy = new StandardWashingStrategy();  
 WashingMachine washingMachine = new WashingMachine(washingStrategy);  
 washingMachine.washingClothes();  
 }  
  
}  
  
/\*\*  
 \* 洗衣类型  
 \*/  
interface WashingStrategy {  
 void washing();  
}  
  
/\*\*  
 \* 洗衣机  
 \*/  
class WashingMachine {  
 private WashingStrategy washingStrategy;  
  
 public WashingMachine(WashingStrategy washingStrategy) {  
 this.washingStrategy = washingStrategy;  
 }  
  
 public void washingClothes() {  
 this.washingStrategy.washing();  
 }  
}  
  
/\*\*  
 \* 标准  
 \*/  
class StandardWashingStrategy implements WashingStrategy{  
  
 @Override  
 public void washing() {  
 System.out.println("标准流程：");  
 System.out.println("[浸泡] 10 分钟");  
 System.out.println("[洗涤] 2 次，每次 15 分钟");  
 System.out.println("[漂洗] 1 次，每次 10 分钟");  
 System.out.println("[脱水] 5 分钟");  
 System.out.println("总共耗时：55 分钟");  
 }  
}  
  
/\*\*  
 \* 快洗  
 \*/  
class QuickWashingStrategy implements WashingStrategy {  
  
 @Override  
 public void washing() {  
 System.out.println("快洗流程：");  
 System.out.println("[洗涤] 1 次，每次 10 分钟");  
 System.out.println("[漂洗] 1 次，每次 10 分钟");  
 System.out.println("[脱水] 5 分钟");  
 System.out.println("总共耗时：25 分钟");  
 }  
}  
  
/\*\*  
 \* 大物  
 \*/  
class BigClothesWashingStrategy implements WashingStrategy {  
  
 @Override  
 public void washing() {  
 System.out.println("大物流程：");  
 System.out.println("[浸泡] 30 分钟");  
 System.out.println("[洗涤] 3 次，每次 15 分钟");  
 System.out.println("[漂洗] 2 次，每次 10 分钟");  
 System.out.println("[脱水] 5 分钟");  
 System.out.println("总共耗时：100 分钟");  
 }  
}  
  
/\*\*  
 \* 轻柔  
 \*/  
class SoftWashingStrategy implements WashingStrategy {  
  
 @Override  
 public void washing() {  
 System.out.println("轻柔流程：");  
 System.out.println("[浸泡] 10 分钟");  
 System.out.println("[漂洗] 2 次，每次 15 分钟");  
 System.out.println("[脱水] 5 分钟");  
 System.out.println("总共耗时：45 分钟");  
 }  
}  
  
标准流程：  
[浸泡] 10 分钟  
[洗涤] 2 次，每次 15 分钟  
[漂洗] 1 次，每次 10 分钟  
[脱水] 5 分钟  
总共耗时：55 分钟

是不是感觉策略模式很简单呢？上面代码就是完整的策略模式示例，是不是感觉有些问题，这 4 种洗衣类型对象完全暴露给了用户，这也是策略模式的缺点。往往策略模式不会单独使用，会和其他设计模式一起使用，比如和简单工厂模式一起使用就可以解决这个对外暴露对象的问题，看下面代码。

/\*\*  
 \* 洗衣类型选择  
 \*/  
class WashingFactory {  
  
 public static WashingStrategy getWashingStrategy(String type) {  
 if ("Quick".equals(type)) {  
 return new QuickWashingStrategy();  
 }  
 if ("BigClothes".equals(type)) {  
 return new BigClothesWashingStrategy();  
 }  
 if ("Soft".equals(type)) {  
 return new SoftWashingStrategy();  
 }  
 return new StandardWashingStrategy();  
 }  
}  
  
public class StrategyTest {  
  
 public static void main(String[] args) {  
 WashingStrategy washingStrategy2 = WashingFactory.getWashingStrategy("Soft");  
 WashingMachine washingMachine2 = new WashingMachine(washingStrategy2);  
 washingMachine2.washingClothes();  
 }  
}  
  
打印结果：  
轻柔流程：  
[浸泡] 10 分钟  
[漂洗] 2 次，每次 15 分钟  
[脱水] 5 分钟  
总共耗时：45 分钟

代码中使用 WashingFactory 来封装 4 种策略，使得策略没有对外暴露，我们也了解到设计模式之间具有互补的关系，有些时候并不是单独存在的。

### 3.5.3. 总结

策略模式是一个很好的封装各种复杂处理的设计模式，让使用者根据自己的选择来选中策略，而不用修改其他代码。当策略太多的时候，可能造成使用方变得复杂、难管理多个策略的问题，利用工厂方法模式可以很好的解决这个难题。这其中也是一个见招拆招的问题，设计模式在真正运用中也是这样子的，遇到问题使用恰当的设计模式去解决问题。

## 3.6. 迭代器模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.6.1. 简介

**姓名** ：迭代器模式

**英文名** ：Iterator Pattern

**价值观** ：人生没有回头路

**个人介绍** ：

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. 它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。 （来自《设计模式之禅》）

### 3.6.2. 你要的故事

大家伙听歌频率高么？是不是经常听歌曲来放松心情？我是经常会听歌，心情不好的时候听歌，心情好的时候也听歌。。。今天讲的迭代器模式，我们就拿听歌这件事来说说，大家都知道听歌有几种模式：单曲循环、列表循环、随机等等。。。现在网易云音乐还多了一个心动模式。

既然说到迭代器模式，那这里就要着重讲讲列表循环这个听歌模式，其他的就先抛到脑后。在列表循环中，歌曲从第一条播放到最后一条，也就是一个遍历歌单的过程。我们有 2 种实现方式，一种是没有迭代器，通过获取歌单，用 for 循环遍历每一个歌曲，然后播放；另外一种是使用迭代器，获取歌单的一个迭代器，通过迭代器来遍历每一个歌曲，然后播放。下面我们就用代码来实现这 2 种方式。

#### 3.6.2.1. 木有迭代器

public class NoIteratorTest {  
  
 public static void main(String[] args) {  
 NetEaseMusic1 netEaseMusic1 = new NetEaseMusic1();  
 netEaseMusic1.listenToMusicByLoop();  
 }  
}  
  
/\*\*  
 \* 网易云音乐  
 \*/  
class NetEaseMusic1 {  
  
 private IList1 songList;  
  
 public NetEaseMusic1() {  
 songList = new SongList1(3);  
 songList.add(new Song("让我留在你身边", "陈奕迅"));  
 songList.add(new Song("你曾是少年", "SHE"));  
 songList.add(new Song("Perfect", "Ed Sheeran"));  
 }  
  
 /\*\*  
 \* 列表循环  
 \*/  
 public void listenToMusicByLoop() {  
 for (int i = 0; i < songList.size(); i++) {  
 System.out.println("听歌：" + ((ISong)songList.get(i)).getSongInfo());  
 }  
 }  
  
}  
  
/\*\*  
 \* 容器接口  
 \*/  
interface IList1 {  
  
 void add(Object object);  
  
 Object get(int index);  
  
 int size();  
}  
  
/\*\*  
 \* 歌单  
 \*/  
class SongList1 implements IList1 {  
  
 private ISong[] songs;  
 private int index;  
 private int size;  
  
 public SongList1(int size) {  
 songs = new ISong[size];  
 index = 0;  
 size = 0;  
 }  
  
 @Override  
 public void add(Object object) {  
 songs[index++] = (ISong) object;  
 size ++;  
 }  
  
 @Override  
 public Object get(int index) {  
 if (index < size) {  
 return songs[index];  
 }  
 return null;  
 }  
  
 @Override  
 public int size() {  
 return size;  
 }  
}  
  
  
/\*\*  
 \* 歌曲接口  
 \*/  
interface ISong {  
 String getSongInfo();  
}  
  
/\*\*  
 \* 歌曲  
 \*/  
class Song implements ISong{  
  
 private String name;  
 private String singer;  
  
 public Song(String name, String singer) {  
 this.name = name;  
 this.singer = singer;  
 }  
  
 @Override  
 public String getSongInfo() {  
 return this.name + "--" + this.singer;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getSinger() {  
 return singer;  
 }  
  
 public void setSinger(String singer) {  
 this.singer = singer;  
 }  
  
}  
  
打印结果：  
听歌：让我留在你身边--陈奕迅  
听歌：你曾是少年--SHE  
听歌：Perfect--Ed Sheeran

我们定义了 ISong 接口，里面有个 getSongInfo() 方法来获取歌曲信息，用 Song 类来定义歌曲。没有用 Java 自带的容器来存放歌曲，这里实现了一个自定义容器接口 IList1，定义 SongList1 来做歌曲的容器，为什么不用 Java 自带的 ArrayList 等等？因为 Java 自带的已经实现了迭代器功能了，我们这里自定义其实就是在模仿自带的容器的实现。NetEaseMusic1 类是充当网易云音乐客户端，在 listenToMusicByLoop() 方法中，我们可以看到是获取了歌单 songList，然后一个一个遍历，这是没有使用迭代器的代码。

下面看一下使用迭代器的代码是怎么样的。

### 3.6.3. 用迭代器实现遍历

public class IteratorTest {  
  
 public static void main(String[] args) {  
 NetEaseMusic2 netEaseMusic2 = new NetEaseMusic2();  
 netEaseMusic2.listenToMusicByLoop();  
 }  
  
}  
  
/\*\*  
 \* 网易云音乐  
 \*/  
class NetEaseMusic2{  
  
 private IList2 songList;  
  
 public NetEaseMusic2() {  
 songList = new SongList2(3);  
 songList.add(new Song("让我留在你身边", "陈奕迅"));  
 songList.add(new Song("你曾是少年", "SHE"));  
 songList.add(new Song("Perfect", "Ed Sheeran"));  
 }  
  
 /\*\*  
 \* 列表循环  
 \*/  
 public void listenToMusicByLoop() {  
 IIterator iterator = songList.iterator();  
 while (iterator.hasNext()) {  
 System.out.println("听歌：" + ((ISong)iterator.next()).getSongInfo());  
 }  
  
 }  
  
}  
  
/\*\*  
 \* 容器接口  
 \*/  
interface IList2 {  
  
 IIterator iterator();  
  
 void add(Object object);  
  
 Object get(int index);  
  
 int size();  
}  
  
/\*\*  
 \* 歌单  
 \*/  
class SongList2 implements IList2 {  
  
 private ISong[] songs;  
 private int index;  
 private int size;  
  
 public SongList2(int size) {  
 songs = new ISong[size];  
 index = 0;  
 size = 0;  
 }  
  
 @Override  
 public IIterator iterator() {  
 return new IteratorImpl(this);  
 }  
  
 @Override  
 public void add(Object object) {  
 songs[index++] = (ISong) object;  
 size ++;  
 }  
  
 @Override  
 public Object get(int index) {  
 if (index < size) {  
 return songs[index];  
 }  
 return null;  
 }  
  
 @Override  
 public int size() {  
 return size;  
 }  
}  
  
  
/\*\*  
 \* 迭代器  
 \*/  
interface IIterator {  
 Object next();  
 boolean hasNext();  
}  
  
/\*\*  
 \* 迭代器实现类  
 \*/  
class IteratorImpl implements IIterator {  
  
 private IList2 list;  
 private int index;  
  
 public IteratorImpl(IList2 list) {  
 this.list = list;  
 this.index = 0;  
 }  
  
 @Override  
 public Object next() {  
 return list.get(index++);  
 }  
  
 @Override  
 public boolean hasNext() {  
 if (index < list.size()) {  
 return true;  
 }  
 return false;  
 }  
}  
  
打印结果：  
听歌：让我留在你身边--陈奕迅  
听歌：你曾是少年--SHE  
听歌：Perfect--Ed Sheeran

代码中我们自定义了一个迭代器接口 IIterator 和迭代器具体实现类 IteratorImpl，有关键的 2 个方法，hasNext() 判断是否有存在下一个元素，next() 获取下一个元素。而 IList2 接口则比 IList1 接口多了一个获取迭代器的方法 iterator()，这让网易云音乐在遍历歌单的时候，不用直接使用 songList 来遍历，而可以通过 songList.iterator() 获取迭代器来实现遍历的过程。NetEaseMusic2.listenToMusicByLoop() 这个方法里面就直接获取迭代器来遍历了。

### 3.6.4. 总结

迭代器模式是所有设计模式中使用最广泛的，有不少开发同学知道迭代器，但是不知道它是设计模式的。虽然迭代器的代码会比没有迭代器的代码复杂，但是加上迭代器可以让容器有统一的遍历代码风格，不用各自去实现遍历方法，有更好的封装性。在 Java 中，迭代器已经运用很广泛，比如 Java 中访问 MySQL 获取数据就是用迭代器来遍历数据的。好了，迭代器模式就讲到这，大家知道的知识就不多说啦。

## 3.7. 观察者模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.7.1. 简介

**姓名** ：观察者模式

**英文名** ：Observer Pattern

**价值观** ：盯着你怎么着

**个人介绍** ： Define a one-to-many dependency between objects so that when one object changes state,all its dependents are notified and updated automatically. 定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。 （来自《设计模式之禅》）

### 3.7.2. 你要的故事

想来想去，就拿我们现在生活中最常体会到的事情来讲观察者模式--朋友圈。小明、小红、小东 3 人是好朋友，最近他们的父母都给安排了手机，刚用上手机那是相当的兴奋呀。他们立马从 QQ 转投到微信的怀抱，对微信的朋友圈玩的不亦乐乎，什么事情都往上面发。突然有一天，小明和小红因为一些小事争执闹别扭了，原因就是他们对一道数学题有不同的见解。就跟我们小时候和朋友玩得好好的，突然因为一点小事就闹翻了。小红比较孩子气，立马就屏蔽了小明的朋友圈，不想再看到有关小明相关的信息。故事就是这么一回事，关注点就在这朋友圈上。朋友圈就是运用观察者模式的一个很好的样例。为什么这么说？我们发朋友圈的时候，那些没有屏蔽我们朋友圈的好友，会收到信息推送。也就是没有屏蔽我们朋友圈的好友其实是订阅了我们朋友圈，好友相当于观察者，我们是被观察的对象。符合观察者模式这个关系。

我们通过代码来描述小明、小红、小东他们在朋友圈玩的场景。利用观察者模式，需要观察对象和被观察对象，所以我们先定义 2 个接口，分别是 Observable (可被观察接口) 和 Observer (观察者接口)。

实现 Observable 接口的对象说明是可被订阅观察的，所以它需要 addObserver() 新增订阅者方法和 removeObserver() 移除订阅者方法，另外还有一个是必须的，就是通知各个订阅者消息的方法 notifyObservers()。那 Observable 接口代码如下所示。

interface Observable {  
 void addObserver(Observer observer);  
 void removeObserver(Observer observer);  
 void notifyObservers(String message);  
}

实现 Observer 接口的对象说明是可以去订阅观察的，也就是说可以接收被订阅的对象发出来的消息，那就需要一个接收消息的方法 update()。代码如下所示。

interface Observer {  
 void update(String name, String message);  
}

**为了让大家不混淆，先把观察者和被观察者分离开**，其实在这个例子中，观察者和被观察者是同一个对象 User 的。这里就分开，分成 User 和 Friend，后面会给出正确的代码，稍安勿躁哈。这里 User 作为被观察者，实现了 Observable 接口，而 Friend 作为观察者，实现了 Observer 接口。代码如下。

class User implements Observable {  
  
 private List<Observer> friends;  
 private String name;  
  
 public User(String name) {  
 this.name = name;  
 this.friends = new LinkedList<>();  
 }  
  
 public void sendMessage(String message) {  
 this.notifyObservers(message);  
 }  
  
 @Override  
 public void addObserver(Observer observer) {  
 this.friends.add(observer);  
 }  
  
 @Override  
 public void removeObserver(Observer observer) {  
 this.friends.remove(observer);  
 }  
  
 @Override  
 public void notifyObservers(String message) {  
 this.friends.forEach(friend -> {  
 friend.update(this.name, message);  
 });  
 }  
}  
  
class Friend implements Observer {  
  
 private String name;  
  
 public Friend(String name) {  
 this.name = name;  
 }  
 @Override  
 public void update(String name, String message) {  
 System.out.println("【" + this.name + "】看到【" + name + "】发的朋友圈：" + message);  
 }  
}  
  
public class ObserverTest {  
  
 public static void main(String[] args) {  
 User xiaoMing = new User("小明");  
 Friend xiaoHong = new Friend("小红");  
 Friend xiaoDong = new Friend("小东");  
 xiaoMing.addObserver(xiaoHong);  
 xiaoMing.addObserver(xiaoDong);  
 xiaoMing.sendMessage("今天真开心");  
 // 小红和小明闹别扭了，小红取消订阅小明的朋友圈  
 xiaoMing.removeObserver(xiaoHong);  
 xiaoMing.sendMessage("希望明天也像今天一样开心");  
 }  
  
}  
  
打印结果：  
【小红】看到【小明】发的朋友圈：今天真开心  
【小东】看到【小明】发的朋友圈：今天真开心  
【小东】看到【小明】发的朋友圈：希望明天也像今天一样开心

看到代码执行结果，小红和小东都订阅了小明的朋友圈，小明发了朋友圈：今天真开心。他们俩都收到了，因为小红和小明闹别扭，小红取消订阅小明的朋友圈，所以小明后来发的朋友圈，小红没收到。

上面代码其实是不对的，不应该用 User 和 Friend 2 个类来定义。如果小明订阅小红和小东的朋友圈呢？这样实现比较麻烦，主要是为了分清 观察者 和 被观察者 这 2 个概念，通过上面的例子应该分清楚了 2 个概念了，那就可以来看正确的代码，小明、小红、小东他们其实都是观察者和被观察者，所以我们用 User2 来定义他们就可以，User2 实现了 Observable 和 Observer 接口。代码如下。

class User2 implements Observable, Observer {  
  
 private List<Observer> friends;  
 private String name;  
  
 public User2(String name) {  
 this.name = name;  
 this.friends = new LinkedList<>();  
 }  
  
 @Override  
 public void addObserver(Observer observer) {  
 this.friends.add(observer);  
 }  
  
 @Override  
 public void removeObserver(Observer observer) {  
 this.friends.remove(observer);  
 }  
  
 @Override  
 public void notifyObservers(String message) {  
 this.friends.forEach(friend -> {  
 friend.update(this.name, message);  
 });  
 }  
  
 @Override  
 public void update(String name, String message) {  
 System.out.println("【" + this.name + "】看到【" + name + "】发的朋友圈：" + message);  
 }  
  
 public void sendMessage(String message) {  
 this.notifyObservers(message);  
 }  
}  
  
public class ObserverTest {  
  
 public static void main(String[] args) {  
 User2 xiaoMing2 = new User2("小明");  
 User2 xiaoHong2 = new User2("小红");  
 User2 xiaoDong2 = new User2("小东");  
 xiaoMing2.addObserver(xiaoHong2);  
 xiaoMing2.addObserver(xiaoDong2);  
 xiaoMing2.sendMessage("今天真开心");  
 xiaoMing2.removeObserver(xiaoHong2);  
 xiaoMing2.sendMessage("希望明天也像今天一样开心");  
  
 xiaoHong2.addObserver(xiaoMing2);  
 xiaoHong2.addObserver(xiaoDong2);  
 xiaoHong2.sendMessage("今天和小明吵架了，屏蔽他的朋友圈");  
  
 xiaoDong2.addObserver(xiaoMing2);  
 xiaoDong2.addObserver(xiaoHong2);  
 xiaoDong2.sendMessage("小明和小红吵架了，夹在中间好尴尬");  
 }  
  
}  
  
打印结果:  
【小红】看到【小明】发的朋友圈：今天真开心  
【小东】看到【小明】发的朋友圈：今天真开心  
【小东】看到【小明】发的朋友圈：希望明天也像今天一样开心  
【小明】看到【小红】发的朋友圈：今天和小明吵架了，屏蔽他的朋友圈  
【小东】看到【小红】发的朋友圈：今天和小明吵架了，屏蔽他的朋友圈  
【小明】看到【小东】发的朋友圈：小明和小红吵架了，夹在中间好尴尬  
【小红】看到【小东】发的朋友圈：小明和小红吵架了，夹在中间好尴尬

从代码中，我们看到小明、小红、小东 3 个人互相订阅朋友圈，当然中途小红屏蔽了小明的朋友圈。这就是 观察者 和 被观察者 刚好是同一个对象的实现。

### 3.7.3. 总结

观察者模式 是一个比较特殊的设计模式，它定义了触发机制，观察者只要订阅了被观察者，就可以第一时间得到被观察者传递的信息。在工作中，使用观察者模式的场景也比较多，比如消息队列消费，Android 开发中的事件触发机制等等。好，观察者模式就到这。

## 3.8. 状态模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.8.1. 简介

**姓名** ：状态模式

**英文名** ：State Pattern

**价值观** ：有啥事让状态我来维护

**个人介绍** ： Allow an object to alter its behavior when its internal state changes.The object will appear to change its class. 当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。 （来自《设计模式之禅》）

### 3.8.2. 你要的故事

现在有好多个人贷款软件，比如：支付宝、360借条(打广告。。。)等等。贷款会有一个用户状态流程，游客->注册用户->授信用户->借款用户(这里简化了状态，只用 4 个)。每个状态拥有的权限不一样，如下图所示。

![](data:image/jpeg;charset=UTF-8;base64,)

状态

从上图可以看到，一个用户有 3 种行为，分别是注册、授信、借款。当注册成功后，用户的状态就从『游客』改变为『注册用户』；当授信成功后，用户的状态就从『注册用户』改变为『授信用户』；当借款成功后，用户的状态就从『授信用户』改变为『借款用户』。现在我们就来实现用户注册、授信、借款的过程，因为每个状态的权限不一样，所以这里需要根据用户的状态来限制用户行为。

很快，我们就完成下面的代码。

class User {  
 private String state;  
  
 public String getState() {  
 return state;  
 }  
  
 public void setState(String state) {  
 this.state = state;  
 }  
  
 public void register() {  
 if ("none".equals(state)) {  
 System.out.println("游客。注册中。。。");  
 }else if ("register".equals(state)) {  
 System.out.println("注册用户。不需要再注册。");  
 } else if ("apply".equals(state)) {  
 System.out.println("授信用户。不需要再注册。");  
 } else if ("draw".equals(state)) {  
 System.out.println("借款用户。不需要再注册。");  
 }  
 }  
  
 public void apply() {  
 if ("none".equals(state)) {  
 System.out.println("游客。不能申请授信。");  
 }else if ("register".equals(state)) {  
 System.out.println("注册用户。授信申请中。。。");  
 } else if ("apply".equals(state)) {  
 System.out.println("授信用户。不需要再授信。");  
 } else if ("draw".equals(state)) {  
 System.out.println("借款用户。不需要再授信。");  
 }  
 }  
  
 public void draw(double money) {  
 if ("none".equals(state)) {  
 System.out.println("游客。申请借款【" + money + "】元。不能申请借款。");  
 } else if ("register".equals(state)) {  
 System.out.println("注册用户。申请借款【" + money + "】元。还没授信，不能借款。");  
 } else if ("apply".equals(state)) {  
 System.out.println("授信用户。申请借款【" + money + "】元。申请借款中。。。");  
 } else if ("draw".equals(state)) {  
 System.out.println("授信用户。申请借款【" + money + "】元。申请借款中。。。");  
 }  
 }  
}  
  
public class NoStateTest {  
  
 public static void main(String[] args) {  
 User user = new User();  
 user.setState("register");  
 user.draw(1000);  
 }  
  
}  
  
打印结果：  
注册用户。申请借款【1000.0】元。还没授信，不能借款。

上面代码实现了用户 register (注册)，apply (授信)，draw (借款) 这 3 种行为，每个行为都会根据状态 state 来做权限控制。看起来有点繁琐，扩展性不高，假设新增了一个状态，那么注册、授信、借款这 3 种行为的代码都要修改。下面通过状态模式来解决这个问题。

我们把状态给抽出来，作为一个接口，因为在每种状态中都可能有注册、授信、借款行为，所以把这 3 个行为作为状态接口的方法，让每个状态子类都实现相应的行为控制。如下代码所示。

interface State {  
  
 void register();  
  
 void apply();  
  
 void draw(double money);  
}  
  
/\*\*  
 \* 游客  
 \*/  
class NoneState implements State {  
  
 @Override  
 public void register() {  
 System.out.println("游客。注册中。。。");  
 }  
  
 @Override  
 public void apply() {  
 System.out.println("游客。不能申请授信。");  
 }  
  
 @Override  
 public void draw(double money) {  
 System.out.println("游客。申请借款【" + money + "】元。不能申请借款。");  
 }  
}  
  
/\*\*  
 \* 注册状态  
 \*/  
class RegisterState implements State {  
  
 @Override  
 public void register() {  
 System.out.println("注册用户。不需要再注册。");  
 }  
  
 @Override  
 public void apply() {  
 System.out.println("注册用户。授信申请中。。。");  
 }  
  
 @Override  
 public void draw(double money) {  
 System.out.println("注册用户。申请借款【" + money + "】元。还没授信，不能借款。");  
 }  
}  
  
/\*\*  
 \* 授信状态  
 \*/  
class ApplyState implements State {  
  
 @Override  
 public void register() {  
 System.out.println("授信用户。不需要再注册。");  
 }  
  
 @Override  
 public void apply() {  
 System.out.println("授信用户。不需要再授信。");  
 }  
  
 @Override  
 public void draw(double money) {  
 System.out.println("授信用户。申请借款【" + money + "】元。申请借款中。。。");  
 }  
}  
  
/\*\*  
 \* 借款状态  
 \*/  
class DrawState implements State {  
  
 @Override  
 public void register() {  
 System.out.println("借款用户。不需要再注册。");  
 }  
  
 @Override  
 public void apply() {  
 System.out.println("借款用户。不需要再授信。");  
 }  
  
 @Override  
 public void draw(double money) {  
 System.out.println("申请借款【" + money + "】元。申请借款中。。。");  
 }  
}  
  
class User1 {  
 private State state;  
  
 public State getState() {  
 return state;  
 }  
  
 public void setState(State state) {  
 this.state = state;  
 }  
  
 public void register() {  
 this.state.register();  
 }  
  
 public void apply() {  
 this.state.apply();  
 }  
  
 public void draw(double money) {  
 this.state.draw(money);  
 }  
}  
  
public class StateTest {  
 public static void main(String[] args) {  
 User1 user1 = new User1();  
 user1.setState(new RegisterState());  
 user1.apply();  
 user1.draw(1000);  
 user1.setState(new ApplyState());  
 user1.draw(2000);  
 }  
  
}  
  
  
打印结果：  
注册用户。授信申请中。。。  
注册用户。申请借款【1000.0】元。还没授信，不能借款。  
授信用户。申请借款【2000.0】元。申请借款中。。。

看上面代码，我们抽象了 State 接口，4 种状态分别用 NoneState (游客)、RegisterState (注册)、ApplyState (授信)、DrawState (借款) 表示。而每个状态都有 3 种行为，它们各自对这些行为进行权限控制。这样子实现可以让权限逻辑分离开，分散到每个状态里面去，如果以后要业务扩展，要新增状态，那就很方便了，只需要再实现一个状态类就可以，不会影响到其他代码。这也是为什么《阿里巴巴 Java 开发手册》里面讲的，当超过 3 层的 if-else 的逻辑判断代码，推荐用状态模式来重构代码。

### 3.8.3. 总结

状态模式 很好的减低了代码的复杂性，从而提高了系统的可维护性。在业务开发中可以尝试使用，比如在迭代开发中，业务逻辑越来越复杂，从而不得不使用很多 if-else 语句来实现时，就可以考虑一下是不是可以用 状态模式 来重构，特别是一些有状态流程转换方面的业务。看到这篇文章，想想工作中是不是有些复杂的代码可以重构，赶紧行动起来。

## 3.9. 备忘录模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.9.1. 简介

**姓名** ：备忘录模式

**英文名** ：Memento Pattern

**价值观** ：凡事要有备份

**个人介绍** ： Without violating encapsulation,capture and externalize an object's internal state so that the object can be restored to this state later. 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。 （来自《设计模式之禅》）

### 3.9.2. 你要的故事

点开看这篇文章的各位，都是。。。程序界的大佬。作为程序猿，免不了『上线』这件小事。每逢上线必祭天。。。上线这件事我们很多人都操作过，每家公司有不同的上线流程以及上线的技术能力。按照发布的平台的完善程度大概分为 3 种。 1. 发布平台牛逼的公司：只需要按下『一键部署』按钮，就搞定上线，按下『回滚』按钮，就搞定回滚上一个版本。 2. 发布平台稍差点的公司：可能就得多个步骤操作了，上线：备份并关闭应用、部署并启动新应用；回滚：关闭新应用、恢复旧应用并启动。 3. 没有发布平台的公司：那就全程手工操作，上线：关闭旧应用、复制旧应用到备份空间、复制新应用到部署环境、启动新应用；回滚：关闭新应用、删除新应用、从备份空间复制旧应用到部署环境、启动旧应用。

其实通过发布平台完善程度可以侧面反映企业的技术成熟程度。怎么说呢？发布系统的操作难易程度在我们作为程序猿心中，都有一个可接受的范围。假设刚开始是单体应用，一个 tomcat 和一个 war 就搞定，你需要发布平台么？并不需要，咱 ctrl+C、ctrl+V、shutdown、startup 就行，还弄什么发布平台。当系统有多套服务时，每次上线都需要部署 10 个机器的应用，这时你能忍么？要是还是手工操作，那发布一次系统得花费好长时间，要是再搞不好，回滚一次，一晚上都没了；这时就会逼迫开发出一个简易的发布平台，把对多台机器的操作步骤放到发布平台上。当系统是以微服务的架构发展时，每个服务都有上百个实例，那这时就不能简单的把操作步骤搬到发布平台了，还得简化步骤，最终变成上面说的 一键部署 和一键回滚。

上面的 3 种我都亲身经历过。。。在刚出来实习时候，就经历了第 3 种情况，因为是单体应用，一个应用搞定所有东西，手动部署已满足要求。到了银行工作，接触到了云平台，那时就只需要一个按钮就唰唰唰的部署了，也是上面说的第 1 种。而现在，正在经历第 2 种发布平台，只是简单的把操作步骤搬到了系统上，目前的情况是机器越来越多，操作步骤没删减的话，每次发布会花费很多时间，这也会去促进开发出更方便使用的发布平台。

回到今天的主题，今天讲的是备忘录模式，从字面上理解，就是讲备份东西，有了备份就可以恢复。上面讲了一大堆发布的东西，也是咱们工作中接触蛮多的事情，发布的最核心就是要支持部署新应用以及回滚老应用，**回滚特别重要，它能够保证在新应用出现异常的情况下，马上恢复到旧应用可用的状态，减少异常的影响面**。发布这东东也很符合备忘录模式，下面通过模拟发布步骤代码来讲备忘录模式。这里讲的不是上面说的第 1 种，这里围绕着第 3 种发布步骤写，不管哪种发布平台，它们的底层都是一样的。

先定义应用实例这个类，应用一般会有应用名、版本号信息。

/\*\*  
 \* 应用实例  
 \*/  
class App {  
 private String content;  
 private String version;  
  
 public App(String content, String version) {  
 this.content = content;  
 this.version = version;  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
  
 public String getVersion() {  
 return version;  
 }  
  
 public void setVersion(String version) {  
 this.version = version;  
 }  
  
 @Override  
 public String toString() {  
 return "App{" +  
 "content='" + content + '\'' +  
 ", version='" + version + '\'' +  
 '}';  
 }  
}

定义 AppBackup 来充当备忘录角色，它有一个属性就是 App，也就是备份的应用。

/\*\*  
 \* 应用备份（充当备忘录角色）  
 \*/  
class AppBackup {  
  
 private App app;  
  
 public AppBackup(App app) {  
 this.app = app;  
 }  
  
 public App getApp() {  
 return app;  
 }  
  
 public void setApp(App app) {  
 this.app = app;  
 }  
}

有了备忘录，也需要一个空间来存放备忘录，并对外提供备忘录。

/\*\*  
 \* 备份空间  
 \*/  
class Space {  
 private AppBackup appBackup;  
  
 public AppBackup getAppBackup() {  
 return appBackup;  
 }  
  
 public void setAppBackup(AppBackup appBackup) {  
 this.appBackup = appBackup;  
 }  
}

有了这些备份机制，还需要有一个程序猿来部署，这位同学需要掌握发布步骤的所有过程，部署新应用以及回滚旧应用。

/\*\*  
 \* 部署应用的同学  
 \*/  
class Deployer {  
  
 // 要部署的应用  
 private App app;  
  
 public App getApp() {  
 return app;  
 }  
  
 // 设置部署应用  
 public void setApp(App app) {  
 this.app = app;  
 }  
  
 // 创建应用的备份  
 public AppBackup createAppBackup() {  
 return new AppBackup(app);  
 }  
  
 // 从备忘录恢复应用  
 public void setAppBackup(AppBackup appBackup) {  
 this.app = appBackup.getApp();  
 }  
  
 // 显示应用的信息  
 public void showApp() {  
 System.out.println(this.app.toString());  
 }  
  
 // 暂停应用  
 public void stopApp() {  
 System.out.println("暂停应用：" + this.app.toString());  
 }  
  
 // 启动应用  
 public void startApp() {  
 System.out.println("启动应用：" + this.app.toString());  
 }  
}

再献上测试代码。

public class MementoTest {  
  
 public static void main(String[] args) {  
 Deployer deployer = new Deployer();  
 deployer.setApp(new App("apply-system", "1.0.0"));  
  
 System.out.println("1. 暂停旧应用");  
 deployer.stopApp();  
  
 System.out.println("2. 备份旧应用");  
 Space space = new Space();  
 space.setAppBackup(deployer.createAppBackup());  
  
 System.out.println("3. 拷贝新应用到服务器");  
 deployer.setApp(new App("apply-system", "2.0.0"));  
 deployer.showApp();  
  
 System.out.println("4. 启动新应用");  
 deployer.startApp();  
  
 System.out.println("5. 有异常，暂停新应用");  
 deployer.stopApp();  
  
 System.out.println("6. 回滚旧应用，拷贝备份的旧应用到服务器");  
 deployer.setAppBackup(space.getAppBackup());  
 deployer.showApp();  
  
 System.out.println("7. 启动备份的旧应用");  
 deployer.startApp();  
 }  
  
}  
  
打印结果：  
1. 暂停旧应用  
暂停应用：App{content='apply-system', version='1.0.0'}  
2. 备份旧应用  
3. 拷贝新应用到服务器  
App{content='apply-system', version='2.0.0'}  
4. 启动新应用  
启动应用：App{content='apply-system', version='2.0.0'}  
5. 有异常，暂停新应用  
暂停应用：App{content='apply-system', version='2.0.0'}  
6. 回滚旧应用，拷贝备份的旧应用到服务器  
App{content='apply-system', version='1.0.0'}  
7. 启动备份的旧应用  
启动应用：App{content='apply-system', version='1.0.0'}

备忘录模式代码实现搞定。有同学会不会觉得挺麻烦的，为什么要有AppBackup？我们看看个人介绍，在对象之外保存状态，AppBackup 就是对象之外的对象，用来保存旧应用。

### 3.9.3. 总结

备忘录模式定义了一个备份机制。在很多场景都有类似备忘录模式的实现，比如数据库的事务的回滚机制。在平常业务开发中并没有经常使用这个设计模式，但是我们有使用它的思想，比如我们用数据库或者其他中间件做备份数据，其中备份思想是一致的。

## 3.10. 解释器模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.10.1. 简介

**姓名** ：解释器模式

**英文名** ：Interpreter Pattern

**价值观** ：不懂解释到你懂​

**个人介绍** ： Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. 给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。 （来自《设计模式之禅》）

### 3.10.2. 你要的故事

解释器顾名思义就是对 2 个不同的表达方式进行转换，让本来不懂的内容解释成看得懂的。比如翻译官就是解释器，把英文翻译成中文，让我们明白外国人说什么。咱们工作中也有很多类似的场景，开发系统避免不了使用数据库，数据库有特定的语法，我们称为 SQL (Structured Query Language)，而我们系统开发语言和 SQL 的语法不一样，这中间就需要做一层转换，像把 Java 语言中的 userDao.save(user) 变成 insert into user (name,age) values ('小明', 18)，这一层转换也可以称为解释器。很多框架实现了这个功能，比如 Hibernate，我们称这些框架为 ORM。

今天，我们就来简单的实现 SQL 拼接解释器，通过参数组装成我们要的 SQL 语句。好多开发同学都吐槽工作天天在 CRUD，也就是只干增删改查的活，对于 SQL 我们经常用的也就是这 4 种语法：insert 语句、delete 语句、update 语句、select 语句。这 4 种语法各有不同，也即需要不同的解释器去解析。利用今天要讲的解释器模式，我们来实现一番。

解释器模式中，会有一个上下文类，这个类用于给解释器传递参数。这里我们 SQL 解释器需要的参数分别是 1. tableName ：数据库名 2. params ：修改时更新后的数据 3. wheres ：where 语句后的条件

class Context {  
 private String tableName;  
 private Map<String, Object> params = new HashMap<>();  
 private Map<String, Object> wheres = new HashMap<>();  
  
 public String getTableName() {  
 return tableName;  
 }  
  
 public void setTableName(String tableName) {  
 this.tableName = tableName;  
 }  
  
 public Map<String, Object> getParams() {  
 return params;  
 }  
  
 public void setParams(Map<String, Object> params) {  
 this.params = params;  
 }  
  
 public Map<String, Object> getWheres() {  
 return wheres;  
 }  
  
 public void setWheres(Map<String, Object> wheres) {  
 this.wheres = wheres;  
 }  
}

解释器主角来了，定义 SQL 解释器抽象类，它有一个抽象方法 interpret，通过这个方法来把 context 中的参数解释成对应的 SQL 语句。

/\*\*  
 \* SQL 解释器  
 \*/  
abstract class SQLExpression {  
  
 public abstract String interpret(Context context);  
  
}

我们上面说了 SQL 语句用的比较多的就是 4 种，每一种其实就是一个解释器，因为语法不一样，解释的逻辑也就不一样，我们就利用 SQLExpression 解释器抽象类，来实现 4 个具体的 SQL 解释器，分别如下：

Insert SQL 解释器代码实现：

/\*\*  
 \* Insert SQL 解释器  
 \*/  
class InsertSQLExpression extends SQLExpression {  
  
 @Override  
 public String interpret(Context context) {  
 StringBuilder insert = new StringBuilder();  
 insert.append("insert into ")  
 .append(context.getTableName());  
  
 // 解析 key value  
 StringBuilder keys = new StringBuilder();  
 StringBuilder values = new StringBuilder();  
 keys.append("(");  
 values.append("(");  
 for (String key : context.getParams().keySet()) {  
 keys.append(key).append(",");  
 values.append("'").append(context.getParams().get(key)).append("',");  
 }  
 keys = keys.replace(keys.length() - 1, keys.length(), ")");  
 values = values.replace(values.length() - 1, values.length(), ")");  
  
 // 拼接 keys values  
 insert.append(keys)  
 .append(" values ")  
 .append(values);  
  
 System.out.println("Insert SQL : " + insert.toString());  
 return insert.toString();  
 }  
}

Update SQL 解释器代码实现：

/\*\*  
 \* Update SQL 解释器  
 \*/  
class UpdateSQLExpression extends SQLExpression {  
  
 @Override  
 public String interpret(Context context) {  
 StringBuilder update = new StringBuilder();  
 update.append("update ")  
 .append(context.getTableName())  
 .append(" set ");  
  
 StringBuilder values = new StringBuilder();  
 for (String key : context.getParams().keySet()) {  
 values.append(key)  
 .append(" = '")  
 .append(context.getParams().get(key))  
 .append("',");  
 }  
  
 StringBuilder wheres = new StringBuilder();  
 wheres.append(" 1 = 1 ");  
 for (String key : context.getWheres().keySet()) {  
 wheres.append(" and ")  
 .append(key)  
 .append(" = '")  
 .append(context.getWheres().get(key))  
 .append("'");  
 }  
  
 update.append(values.substring(0, values.length() - 1))  
 .append(" where ")  
 .append(wheres);  
  
 System.out.println("Update SQL : " + update.toString());  
 return update.toString();  
 }  
}

Select SQL 解释器代码实现：

/\*\*  
 \* Select SQL 解释器  
 \*/  
class SelectSQLExpression extends SQLExpression {  
  
 @Override  
 public String interpret(Context context) {  
 StringBuilder select = new StringBuilder();  
 select.append("select \* from ")  
 .append(context.getTableName())  
 .append(" where ")  
 .append(" 1 = 1 ");  
 for (String key : context.getWheres().keySet()) {  
 select.append(" and ")  
 .append(key)  
 .append(" = '")  
 .append(context.getWheres().get(key))  
 .append("'");  
 }  
 System.out.println("Select SQL : " + select.toString());  
 return select.toString();  
 }  
}

Delete SQL 解释器代码实现

/\*\*  
 \* Delete SQL 解释器  
 \*/  
class DeleteSQLExpression extends SQLExpression {  
  
 @Override  
 public String interpret(Context context) {  
 StringBuilder delete = new StringBuilder();  
 delete.append("delete from ")  
 .append(context.getTableName())  
 .append(" where ")  
 .append(" 1 = 1");  
 for (String key : context.getWheres().keySet()) {  
 delete.append(" and ")  
 .append(key)  
 .append(" = '")  
 .append(context.getWheres().get(key))  
 .append("'");  
 }  
 System.out.println("Delete SQL : " + delete.toString());  
  
 return delete.toString();  
 }  
}

测试代码

public class InterpreterTest {  
 public static void main(String[] args) {  
 Context context = new Context();  
 context.setTableName("user");  
  
 // Insert SQL  
 Map<String, Object> params = new HashMap<>();  
 params.put("name", "小明");  
 params.put("job", "Java 工程师");  
 context.setParams(params);  
 SQLExpression sqlExpression = new InsertSQLExpression();  
 String sql = sqlExpression.interpret(context);  
  
 // Delete SQL  
 Map<String, Object> wheres = new HashMap<>();  
 wheres.put("name", "小明");  
 context.setParams(null);  
 context.setWheres(wheres);  
 sqlExpression = new DeleteSQLExpression();  
 sql = sqlExpression.interpret(context);  
  
 // Update SQL  
 params = new HashMap<>();  
 params.put("job", "Java 高级工程师");  
 wheres = new HashMap<>();  
 wheres.put("name", "小明");  
 context.setParams(params);  
 context.setWheres(wheres);  
 sqlExpression = new UpdateSQLExpression();  
 sql = sqlExpression.interpret(context);  
  
 // Select SQL  
 wheres = new HashMap<>();  
 wheres.put("name", "小明");  
 context.setParams(null);  
 context.setWheres(wheres);  
 sqlExpression = new SelectSQLExpression();  
 sql = sqlExpression.interpret(context);  
 }  
  
}  
  
打印结果：  
  
Insert SQL : insert into user(name,job) values ('小明','Java 工程师')  
Delete SQL : delete from user where 1 = 1 and name = '小明'  
Update SQL : update user set job = 'Java 高级工程师' where 1 = 1 and name = '小明'  
Select SQL : select \* from user where 1 = 1 and name = '小明'

上面实现了整个解释器模式的代码，其实咱们在开发中，SQL 解析没有这么去实现，更多是用一个工具类把上面的各个 SQL 解释器的逻辑代码分别实现在不同方法中，如下代码所示。因为咱们可以预见的就这 4 种语法类型，基本上不用什么扩展，用一个工具类就足够了。

class SQLUtil {  
  
 public static String insert(String tableName, Map<String, Object> params) {  
 StringBuilder insert = new StringBuilder();  
 insert.append("insert into ")  
 .append(tableName);  
  
 // 解析 key value  
 StringBuilder keys = new StringBuilder();  
 StringBuilder values = new StringBuilder();  
 keys.append("(");  
 values.append("(");  
 for (String key : params.keySet()) {  
 keys.append(key).append(",");  
 values.append("'").append(params.get(key)).append("',");  
 }  
 keys = keys.replace(keys.length() - 1, keys.length(), ")");  
 values = values.replace(values.length() - 1, values.length(), ")");  
  
 // 拼接 keys values  
 insert.append(keys)  
 .append(" values ")  
 .append(values);  
  
 System.out.println("Insert SQL : " + insert.toString());  
 return insert.toString();  
 }  
  
 public static String update(String tableName, Map<String, Object> params, Map<String, Object> wheres) {  
 StringBuilder update = new StringBuilder();  
 update.append("update ")  
 .append(tableName)  
 .append(" set ");  
  
 StringBuilder values = new StringBuilder();  
 for (String key : params.keySet()) {  
 values.append(key)  
 .append(" = '")  
 .append(params.get(key))  
 .append("',");  
 }  
  
 StringBuilder wheresStr = new StringBuilder();  
 wheresStr.append(" 1 = 1 ");  
 for (String key : wheres.keySet()) {  
 wheresStr.append(" and ")  
 .append(key)  
 .append(" = '")  
 .append(wheres.get(key))  
 .append("'");  
 }  
  
 update.append(values.substring(0, values.length() - 1))  
 .append(" where ")  
 .append(wheresStr);  
  
 System.out.println("Update SQL : " + update.toString());  
 return update.toString();  
 }  
  
 public static String select(String tableName, Map<String, Object> wheres) {  
 StringBuilder select = new StringBuilder();  
 select.append("select \* from ")  
 .append(tableName)  
 .append(" where ")  
 .append(" 1 = 1 ");  
 for (String key : wheres.keySet()) {  
 select.append(" and ")  
 .append(key)  
 .append(" = '")  
 .append(wheres.get(key))  
 .append("'");  
 }  
 System.out.println("Select SQL : " + select.toString());  
 return select.toString();  
 }  
  
 public static String delete(String tableName, Map<String, Object> wheres) {  
 StringBuilder delete = new StringBuilder();  
 delete.append("delete from ")  
 .append(tableName)  
 .append(" where ")  
 .append(" 1 = 1");  
 for (String key : wheres.keySet()) {  
 delete.append(" and ")  
 .append(key)  
 .append(" = '")  
 .append(wheres.get(key))  
 .append("'");  
 }  
 System.out.println("Delete SQL : " + delete.toString());  
  
 return delete.toString();  
 }  
}

### 3.10.3. 总结

上面用解释器模式实现了 SQL 解释器，然后又指明了实际上咱们开发中大多数是直接一个 SQLUtil 工具类就搞定，并不是说解释器模式没用，想表达的观点是：解释器在工作中很少使用，工作中我们一般遵循的是能用就好策略，满足当前需求，加上一些易扩展性就足够了。解释器模式有比较大的扩展性，就如上面，再加上个建表语句 create table 只需要加一个 CreateTableSQLExpression 就可以轻松实现，不用去改动其他解释器代码。今天的解释器就到讲到这。觉得不错点个“在看”鼓励鼓励一下。

## 3.11. 访问者模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 3.11.1. 简介

**姓名** ：访问者模式

**英文名** ：Visitor Pattern

**价值观** ：来访者便是客，招待就是

**个人介绍** ： Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. 封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。 （来自《设计模式之禅》）

### 3.11.2. 你要的故事

先声明一下，下面故事全瞎编的。。。

我们是否还记得 N 年前反腐开始的时候，有一段时间提倡官员宴请吃饭只能几菜几汤，不能超出。我记得那会刚读大一，军事理论的老师说到这个问题，也发表了他的一些想法，他觉得这么做比较刻板。今天的故事就和宴请有关。现在中国企业发展越来越大，在社会中担任的责任也越来越大，政府也越来越重视企业，官员去参观企业是常有的事，而企业宴请官员也变得格外的常见。

故事的背景就是企业宴请各级官员。不同级别的官员宴请的菜式就不一样，每家企业的菜式丰富程度也不一样。我们这里的访问对象就用 Alibaba 和 Tencent 这 2 家公司，而访问者就用郭嘉领导人和省领导人做举例。这 2 家公司都跟喜来登酒店合作，Alibaba 合作方案是：宴请省级领导人及以下官员则十菜一汤，宴请郭嘉领导人及以上官员则十四菜两汤；Tencent 合作方案是：宴请省领导人及以下官员则八菜一汤，宴请郭嘉领导人及以上官员则十六菜两汤。

下面看看如何用访问者模式来实现上面的故事。

首先定义一个抽象类：企业。企业有一个共有的特性就是接受上级领导的访问。

/\*\*  
 \* 企业  
 \*/  
abstract class Company {  
  
 public abstract void accept(Vistor vistor);  
  
}

上面故事我们举例了 2 家企业，分别是 Alibaba 和 Tencent，这里实现这 2 家公司的宴请方案，并实现接待访问者方法。

Alibaba 宴请郭嘉领导人及以上官员是十四菜两汤，宴请省领导及以下是十菜一汤。

/\*\*  
 \* Alibaba 企业  
 \*/  
class AlibabaCompany extends Company {  
  
 @Override  
 public void accept(Vistor vistor) {  
 vistor.visit(this);  
 }  
  
 public String entertainBelowProvincialLeader(String leader) {  
 return "Alibaba 接待" + leader + "：十菜一汤";  
 }  
  
 public String entertainAboveNationalLeader(String leader) {  
 return "Alibaba 接待" + leader + "：十四菜两汤";  
 }  
  
}

Tencent 宴请郭嘉领导人及以上是十六菜两汤，宴请省领导及以下是八菜一汤。

/\*\*  
 \* Tencent 企业  
 \*/  
class TencentCompany extends Company {  
  
 @Override  
 public void accept(Vistor vistor) {  
 vistor.visit(this);  
 }  
  
 public String entertainBelowProvincialLeader(String leader) {  
 return "Tencent 接待" + leader + "：八菜一汤";  
 }  
  
 public String entertainAboveNationalLeader(String leader) {  
 return "Tencent 接待" + leader + "：十六菜两汤";  
 }  
}

这里定义访问者接口，访问者接口有 2 个方法，分别是访问 Alibaba 企业和访问 Tencent 企业。

/\*\*  
 \* 访问者接口  
 \*/  
interface Vistor {  
  
 void visit(AlibabaCompany alibabaCompany);  
  
 void visit(TencentCompany tencentCompany);  
  
}

上面故事中有 2 个访问者，一个是郭嘉领导人，另一个是省领导人，因为不同企业对应不同访问者有不同的宴请方案，所以这里访问企业是需要调用对应企业的宴请方式。

省领导人访问企业时，需要调用企业对省领导及以下官员的宴请方案，为entertainBelowProvincialLeader()

/\*\*  
 \* 省领导访问  
 \*/  
class ProvincialLeaderVistor implements Vistor {  
  
 @Override  
 public void visit(AlibabaCompany alibabaCompany) {  
 System.out.println(alibabaCompany.entertainBelowProvincialLeader("省领导"));  
 }  
  
 @Override  
 public void visit(TencentCompany tencentCompany) {  
 System.out.println(tencentCompany.entertainBelowProvincialLeader("省领导"));  
 }  
}

郭嘉领导人访问企业时，需要调用企业对郭嘉领导人的宴请方案，为entertainAboveNationalLeader()

/\*\*  
 \* 郭嘉领导访问  
 \*/  
class NationalLeaderVistor implements Vistor {  
  
 @Override  
 public void visit(AlibabaCompany alibabaCompany) {  
 System.out.println(alibabaCompany.entertainAboveNationalLeader("省领导"));  
 }  
  
 @Override  
 public void visit(TencentCompany tencentCompany) {  
 System.out.println(tencentCompany.entertainAboveNationalLeader("郭嘉领导"));  
 }  
}

上面是访问者和被访问者的代码，因为企业是在喜来登酒店宴请领导人，所以这里还需要一个酒店，酒店里面有企业合作的名单，以及负责宴请各路领导的方法提供。

/\*\*  
 \* 酒店  
 \*/  
class Hotel {  
 private List<Company> companies = new ArrayList<>();  
  
 public void entertain(Vistor vistor) {  
 for (Company company : companies) {  
 company.accept(vistor);  
 }  
 }  
  
 public void add(Company company) {  
 companies.add(company);  
 }  
}

下面提供测试代码，看看运行的结果怎样。

public class VisitorTest {  
  
 public static void main(String[] args) {  
 AlibabaCompany alibabaCompany = new AlibabaCompany();  
 TencentCompany tencentCompany = new TencentCompany();  
 ProvincialLeaderVistor provincialLeaderVistor = new ProvincialLeaderVistor();  
 NationalLeaderVistor nationalLeaderVistor = new NationalLeaderVistor();  
  
 Hotel xilaideng = new Hotel();  
 xilaideng.add(alibabaCompany);  
 xilaideng.add(tencentCompany);  
  
 xilaideng.entertain(provincialLeaderVistor);  
 xilaideng.entertain(nationalLeaderVistor);  
 }  
  
}  
  
打印结果：  
Alibaba 接待省领导：十菜一汤  
Tencent 接待省领导：八菜一汤  
Alibaba 接待郭嘉领导：十四菜两汤  
Tencent 接待郭嘉领导：十六菜两汤

完整的访问者模式代码已经呈现，花 1 分钟思考一番，理解整个代码后我们来看看下面的总结。

### 3.11.3. 总结

访问者模式有比较好的扩展性，看看访问者代码，我们如果要新增一个访问者：市领导人，只需新增市领导人类，便可实现。当然也有它不好的地方，就是把被访问者暴露给访问者，使得访问者可以直接了解被访问者的所有东西。明白了优缺点，才能更好的在实际中运用，一般访问者模式运用于要求遍历多个不同的对象的场景。

# 4. 七大结构型模式

## 4.1. 适配器模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.1.1. 简介

**姓名** ：适配器模式

**英文名** ：Adapter Pattern

**价值观** ：老媒人，牵线搭桥

**个人介绍** ： Convert the interface of a class into another interface clients expect.Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. 将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。 （来自《设计模式之禅》）

### 4.1.2. 你要的故事

大家有买过港式的 Apple 产品么？在深圳的同学估计买过，毕竟港式的 Apple 产品基本比国内便宜 500 以上。我手机和平板都是在香港买的，买来后这充电器是没法直接充电的，因为港版的电子产品都是英式的插头，而咱们国内是中式的，所以用上港版电子产品的同学免不了要用上这么一个转换器：将英式的插孔转为中式的插孔，方可插入咱家里的插座充电。这个转换器就是今天想讲的适配器。

没见过的同学可以看看图片熟悉一下，下图右边为港版苹果手机充电器，插头比较大，左边为某品牌转换器，插头为中国家用标准形状。 ![](data:image/jpeg;charset=UTF-8;base64,)

下图为使用时的图片 ![](data:image/jpeg;charset=UTF-8;base64,)

在这描述一下这个场景。用港式插头要在国内充电，因为插头和插座大小对不上，所以需要加一个适配器，这个适配器充当插头和插座，它的插头可以插入国内标准的插座，它的插座可以插入港式标准的插头，这样子就可以用港式充电器在国内为手机充电。

下面用适配器模式代码实现这个场景。

首先需要找到被适配的对象是什么？在这里我们的被适配对象是英式充电器。

/\*\*  
 \* 英式充电器  
 \*/  
class BritishCharger {  
  
 public void chargeByBritishStandard(){  
 System.out.println("用英式充电器充电");  
 }  
  
}

在这个场景的目的是什么？在中国为港式手机充电，因此目的是让英式充电器能够在中国标准的插座充电。

/\*\*  
 \* 使用中式插座充电  
 \*/  
interface Target {  
  
 void chargeByChineseStandard();  
  
}

接下来是这个设计模式的主角：适配器。它需要连接中式插座以及英式充电器，在中间做适配功能。

/\*\*  
 \* 充电器适配器  
 \*/  
class ChargerAdapter implements Target {  
  
 private BritishCharger britishCharger;  
  
 public ChargerAdapter(BritishCharger britishCharger) {  
 this.britishCharger = britishCharger;  
 }  
  
 @Override  
 public void chargeByChineseStandard() {  
 System.out.println("使用中英式插头转换器");  
 britishCharger.chargeByBritishStandard();  
 }  
}

上面是适配器模式的一个简单的例子，要学习适配器模式也可以看看 Java 的 IO 实现源码，里面是应用适配器模式的官方很好的代码。

### 4.1.3. 总结

适配器很好的将 2 个无法关联的类结合起来，在中间起桥梁作用。另外新增适配器代码不会影响原来被适配者的正常使用，他们可以一起被使用。在工作中和外部系统对接的时候，大可能外部系统的数据格式和自己系统的数据格式并不相同，这时候就可以利用适配器模式来实现。

## 4.2. 桥接模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.2.1. 简介

**姓名** ：桥接模式

**英文名** ：Bridge Pattern

**价值观** ：解耦靠我

**个人介绍** ： Decouple an abstraction from its implementation so that the two can vary independently. 将抽象和实现解耦，使得两者可以独立地变化。 （来自《设计模式之禅》）

### 4.2.2. 你要的故事

现在手机二分天下，安卓手机和苹果手机目前占有率高居 98.45%，其中安卓手机占有率为 70.21%，苹果手机占有率为 28.24%，如下图所示。

![](data:image/jpeg;charset=UTF-8;base64,) （数据从 [netmarketshare](https://netmarketshare.com/operating-system-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22deviceType%22%3A%7B%22%24in%22%3A%5B%22Mobile%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Trend%22%2C%22attributes%22%3A%22share%22%2C%22group%22%3A%22platform%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22platformsMobile%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222018-04%22%2C%22dateEnd%22%3A%222019-03%22%2C%22plotKeys%22%3A%5B%7B%22platform%22%3A%22Android%22%7D%2C%7B%22platform%22%3A%22iOS%22%7D%2C%7B%22platform%22%3A%22Unknown%22%7D%2C%7B%22platform%22%3A%22Series%2040%22%7D%2C%7B%22platform%22%3A%22Windows%20Phone%20OS%22%7D%5D%2C%22segments%22%3A%22-1000%22%7D) 来）

因为有这 2 个系统，所以很多软件商都不得不开发 2 个系统的 APP。我们就拿这个案例来讲，目前手机有安卓手机和苹果手机，软件有谷歌浏览器和火狐浏览器，通过手机打开软件这一过程来讲讲桥接模式。

从个人介绍可见，需要抽象化和实现化，然后使用桥接模式将抽象和实现解耦。

抽象化：把一类对象共有的东西抽象到一个类里面，该类作为这类对象的基类。在这里我们可以抽象化的便是**手机**。

实现化：将接口或抽象类的未实现的方法进行实现。在这里我们可以实现化的就是**软件**。

将抽象和实现解耦：有了上面的抽象化和实现化，通过桥接模式来实现解耦。在这里，我们把打开软件 open() 放到软件实现中，而抽象的手机利用模板方法模式定义 openSoftware() 供手机子类去实现，手机子类也是调用软件的 open() 方法，并没有自己实现打开逻辑，也就是解耦了这个打开软件过程。

下面给出案例的代码。

Phone 手机抽象类代码。属性 system 代表系统名称，software 代表要打开的软件，openSoftware() 对外提供打开软件的方法。

abstract class Phone {  
  
 private String system;  
 private Software software;  
  
 public abstract void openSoftware();  
  
 public String getSystem() {  
 return system;  
 }  
  
 public void setSystem(String system) {  
 this.system = system;  
 }  
  
 public Software getSoftware() {  
 return software;  
 }  
  
 public void setSoftware(Software software) {  
 this.software = software;  
 }  
  
}

AndroidPhone 安卓系统手机代码。

class AndroidPhone extends Phone {  
  
 public AndroidPhone(Software software){  
 this.setSystem("Android");  
 this.setSoftware(software);  
 }  
  
 @Override  
 public void openSoftware() {  
 this.getSoftware().open(this);  
 }  
}

IOSPhone IOS 系统手机代码（也就是苹果手机）。

class IOSPhone extends Phone {  
  
 public IOSPhone(Software software) {  
 this.setSystem("IOS");  
 this.setSoftware(software);  
 }  
  
 @Override  
 public void openSoftware() {  
 this.getSoftware().open(this);  
 }  
}

Software 软件接口代码。它有一个方法 open()，用于打开该软件。

interface Software {  
 void open(Phone phone);  
}

Chrome 谷歌浏览器软件代码。

class Chrome implements Software {  
  
 @Override  
 public void open(Phone phone) {  
 System.out.println("打开 " + phone.getSystem() + " 手机的 Chrome 浏览器");  
 }  
  
}

FireFox 火狐浏览器软件代码。

class FireFox implements Software {  
  
 @Override  
 public void open(Phone phone) {  
 System.out.println("打开 " + phone.getSystem() + " 手机的 Firefox 浏览器");  
 }  
  
}

测试代码如下。

public class BridgeTest {  
  
 public static void main(String[] args) {  
 Software chrome = new Chrome();  
 Software firefox = new FireFox();  
  
 Phone androidPhone = new AndroidPhone(chrome);  
 androidPhone.openSoftware();  
  
 androidPhone.setSoftware(firefox);  
 androidPhone.openSoftware();  
  
 Phone iosPhone = new IOSPhone(chrome);  
 iosPhone.openSoftware();  
  
 iosPhone.setSoftware(firefox);  
 iosPhone.openSoftware();  
 }  
  
}  
  
打印结果：  
打开 Android 手机的 Chrome 浏览器  
打开 Android 手机的 Firefox 浏览器  
打开 IOS 手机的 Chrome 浏览器  
打开 IOS 手机的 Firefox 浏览器

桥接模式代码已经写完。为什么叫桥接模式呢？因为它将打开软件的具体实现放到了软件实现里面，而不是放在了手机，通过聚合方式去调用软件打开的方法，这就像一条桥一样连接手机和软件。

### 4.2.3. 总结

桥接模式利用了聚合的优点去解决继承的缺点，使得抽象和实现进行分离解耦。正由于解耦，使得有更好的扩展性，加手机类型或者加软件都非常容易，也不会破坏原有的代码。

## 4.3. 组合模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.3.1. 简介

**姓名** ：组合模式

**英文名** ：Composite Pattern

**价值观** ：专门解决各种树形疑难杂症

**个人介绍** ： Compose objects into tree structures to represent part-whole hierarchies.Composite lets clients treat individual objects and compositions of objects uniformly. 将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。 （来自《设计模式之禅》）

### 4.3.2. 你要的故事

今天咱们再讲讲咱们程序猿的组织架构。技术类的组织架构比较单一，基本上都是这样：经理--->组长--->工程师，如下图所示。

![](data:image/jpeg;charset=UTF-8;base64,)

IT组织架构

各个公司的 title 可能不太一样，但是基本是差不多这种架构，按职业发展，从入职到能独立开发需求便为工程师，从独立开发需求到能带小团队开发便为组长，从带小团队开发到能带几个团队一起协作开发便为经理。

假设目前有一家公司，技术部就 4 个人，大熊担任经理，中熊担任组长，小熊1和小熊2担任工程师。下面的代码都围绕这个假设编写。

#### 4.3.2.1. 非组合模式

我们先来一个非正常的实现方案：从组织架构里，有 3 个角色，分别是经理、组长、工程师，那么我们就按角色去实现一番。

Manager 为经理类，经理下有多个组长 leaders。

/\*\*  
 \* 经理  
 \*/  
class Manager {  
  
 private String name;  
 private List<Leader> leaders;  
  
 public Manager(String name) {  
 this.name = name;  
 this.leaders = new LinkedList<>();  
 }  
  
 public void add(Leader leader) {  
 this.leaders.add(leader);  
 }  
  
 public void remove(Leader leader) {  
 this.leaders.remove(leader);  
 }  
  
 public void display(int index) {  
 for (int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("经理：" + this.name);  
 leaders.forEach(leader -> {  
 leader.display(index+1);  
 });  
 }  
  
}

Leader 为组长类，组长下有多个工程师 engineers。

/\*\*  
 \* 组长  
 \*/  
class Leader {  
  
 private String name;  
 private List<Engineer> engineers;  
  
 public Leader(String name) {  
 this.name = name;  
 this.engineers = new LinkedList<>();  
 }  
  
 public void add(Engineer engineer) {  
 this.engineers.add(engineer);  
 }  
  
 public void remove(Engineer engineer) {  
 this.engineers.remove(engineer);  
 }  
  
 public void display(int index) {  
 for (int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("组长：" + this.name);  
 engineers.forEach(engineer -> {  
 engineer.display(index + 1);  
 });  
 }  
}

Engineer 为工程师类，工程师没有下属。

/\*\*  
 \* 工程师  
 \*/  
class Engineer {  
  
 private String name;  
  
 public Engineer(String name) {  
 this.name = name;  
 }  
  
 public void display(int index) {  
 for (int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("工程师：" + this.name);  
 }  
  
}

测试代码

public class NoCompositeTest {  
  
 public static void main(String[] args) {  
 Manager manager = new Manager("大熊");  
 Leader leader = new Leader("中熊");  
 Engineer engineer1= new Engineer("小熊1");  
 Engineer engineer2 = new Engineer("小熊2");  
  
 manager.add(leader);  
 leader.add(engineer1);  
 leader.add(engineer2);  
  
 manager.display(0);  
 }  
  
}  
  
打印结果：  
经理：大熊  
----组长：中熊  
--------工程师：小熊1  
--------工程师：小熊2

这份代码看完之后，有什么想法？是不是感觉代码有点冗余？经理和组长的代码几乎一致，而工程师类和经理类、组长类也有共同点，唯一的区别就是工程师没有下属，因此没有对下属的增删操作方法。

#### 4.3.2.2. 安全模式

通过上面一层思考，这 3 个角色有相通性，我们可以抽象出一个 Employee2 类，把 3 个角色共同的特性放到 Employee2 类中，经理和组长合并共用一个类，因为在这个例子里，这 2 个角色完全一样的。下面看代码。

Employee2 抽象类，它有这 3 个角色共有的特性，名称设置获取以及显示数据。

abstract class Employee2 {  
  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public abstract void display(int index);  
  
}

Leader2 领导类，把上面的经理类和组长类都合并到这个领导类，因为他们都是领导层。

class Leader2 extends Employee2 {  
  
 private List<Employee2> employees;  
  
 public Leader2(String name) {  
 this.setName(name);  
 this.employees = new ArrayList<>();  
 }  
  
 public void add(Employee2 employee) {  
 this.employees.add(employee);  
 }  
  
 public void remove(Employee2 employee) {  
 this.employees.remove(employee);  
 }  
  
 @Override  
 public void display(int index) {  
 for(int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("领导：" + this.getName());  
 this.employees.forEach(employee -> {  
 employee.display(index + 1);  
 });  
 }  
}

Engineer2 工程师类，工程师类比较简单，因为名称设置获取在抽象类 Employee2 有了，所以就只需实现显示数据的功能。

class Engineer2 extends Employee2 {  
  
 public Engineer2(String name) {  
 this.setName(name);  
 }  
  
 @Override  
 public void display(int index) {  
 for(int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("工程师：" + this.getName());  
 }  
}

测试代码

public class CompositeTest {  
  
 public static void main(String[] args) {  
 // 安全模式  
 Leader2 leader1 = new Leader2("大熊");  
 Leader2 leader2 = new Leader2("中熊");  
 Engineer2 engineer1 = new Engineer2("小熊1");  
 Engineer2 engineer2 = new Engineer2("小熊2");  
  
 leader1.add(leader2);  
 leader2.add(engineer1);  
 leader2.add(engineer2);  
  
 leader1.display(0);  
 }  
  
}  
  
打印结果：  
领导：大熊  
----领导：中熊  
--------工程师：小熊1  
--------工程师：小熊2

看下运行结果和上面是一致的，这份代码比第一份代码有更好的封装性，也更符合面向对象的编程方式，经理和组长被合并成 Leader2，也就是咱们今天讲的组合模式，Leader2 为组合对象。上面讲的是安全模式，安全模式指的是抽象类 Employee2 只提供了 3 个角色中共有的特性，安全是相对透明模式所说的，因为这里领导类 Leader2 和工程师类 Engineer2 都只提供了自己能提供的方法，Engineer2 不会有多余的方法，而透明模式则不是。下面讲讲透明模式。

#### 4.3.2.3. 透明模式

透明模式把组合对象（即领导类）使用的方法放到抽象类中，而因为工程师没有下属，则不具体实现对应的方法。代码如下。

Employee3 抽象类，将组合对象的属性 employees 和 方法 add()、 remove() 都放到这个类里面。

abstract class Employee3 {  
  
 private String name;  
 private List<Employee3> employees;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public List<Employee3> getEmployees() {  
 return employees;  
 }  
  
 public void setEmployees(List<Employee3> employees) {  
 this.employees = employees;  
 }  
  
 public abstract void add(Employee3 employee);  
  
 public abstract void remove(Employee3 employee);  
  
 public abstract void display(int index);  
  
}

Leader3 领导类，具体实现 Employee3 提供的所有方法。

class Leader3 extends Employee3 {  
  
 public Leader3(String name) {  
 this.setName(name);  
 this.setEmployees(new ArrayList<>());  
 }  
  
 @Override  
 public void add(Employee3 employee) {  
 this.getEmployees().add(employee);  
 }  
  
 @Override  
 public void remove(Employee3 employee) {  
 this.getEmployees().remove(employee);  
 }  
  
 @Override  
 public void display(int index) {  
 for(int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("领导：" + this.getName());  
 this.getEmployees().forEach(employee -> {  
 employee.display(index + 1);  
 });  
 }  
}

Engineer3 工程师类，只具体实现 Employee3 中的 display() 方法，add() 和 remove() 方法不是工程师具备的，所以留空，不做具体实现。

class Engineer3 extends Employee3 {  
  
 public Engineer3(String name) {  
 this.setName(name);  
 }  
  
 @Override  
 public void add(Employee3 employee) {  
 // 没有下属  
 }  
  
 @Override  
 public void remove(Employee3 employee) {  
 // 没有下属  
 }  
  
 @Override  
 public void display(int index) {  
 for(int i = 0; i < index; i++) {  
 System.out.print("----");  
 }  
 System.out.println("工程师：" + this.getName());  
 }  
}

测试代码：

public class CompositeTest {  
  
 public static void main(String[] args) {  
 // 透明模式  
 Leader3 leader3 = new Leader3("大熊");  
 Leader3 leader31 = new Leader3("中熊");  
 Engineer3 engineer31 = new Engineer3("小熊1");  
 Engineer3 engineer32 = new Engineer3("小熊2");  
  
 leader3.add(leader31);  
 leader31.add(engineer31);  
 leader31.add(engineer32);  
  
 leader3.display(0);  
  
 }  
  
打印结果：  
领导：大熊  
----领导：中熊  
--------工程师：小熊1  
--------工程师：小熊2  
}

安全模式把 3 个角色的共同点抽象到 Employee2 中，透明模式则把 3 个角色中的领导者（组合对象）的内容抽象到 Employee3 中。透明模式有些不好的地方在于工程师也有领导者的下属对象和相应的方法，其实工程师并没有这些功能。安全模式把领导者和工程师分开，每个对象都只提供自己具有的功能，这样子在使用的时候也就更安全。

### 4.3.3. 总结

我们根据 IT 组织架构，从简单的每个角色对应一个类的实现，再到抽象出每个角色共同的功能、组合领导类的安全模式，接着再到抽象起来领导类（组合）所有功能的透明模式，分析了组合模式的完整过程，也讲了安全模式和透明模式的差异。组合模式让对象更加有层次，将对象的划分更加清晰，特别是树形结构的层次，利用组合模式会更加简化。

## 4.4. 装饰模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.4.1. 简介

**姓名** ：装饰模式

**英文名** ：Decorator Pattern

**价值观** ：人靠衣装，类靠装饰

**个人介绍** ： Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality. 动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。 （来自《设计模式之禅》）

### 4.4.2. 你要的故事

夏天到了，吃货们期待的各种各样冷冻零食就要大面积面向市场，什么冰淇淋、雪糕、冰棍等等。今天的装饰模式不讲这些都受欢迎的零食，讲讲那乌黑滴龟苓膏。不知道大伙们喜不喜欢吃龟苓膏，我是挺喜欢的，不喜欢的人很多都闲它苦，应该没有人愿意在没加任何糖类的情况下吃龟苓膏。很多糖水店会提供几种龟苓膏，比如蜂蜜龟苓膏、牛奶龟苓膏。下面我们空想出一个场景来。

天气到了 30℃，小明和小红加班到了 10 点，一起下班，路过一家糖水店，小红萌生了吃糖水解解热的想法，小明最近涨薪，就提出要请小红吃糖水，他们进去糖水店，小红想着好久没吃龟苓膏了，就想吃吃，怀念一下童年那段在农村夜晚吃龟苓膏的时光。糖水店里面有 3 种龟苓膏，一种是普通龟苓膏（这老板居然提供不加任何糖分的，过分了），一种是蜂蜜龟苓膏，另外一种是牛奶龟苓膏，小明点了一份蜂蜜龟苓膏，小红想加蜂蜜和牛奶，就咨询了老板娘，能否同时加蜂蜜和牛奶，老板娘用那东北腔爽快地回复小红：行。小明和小红就等龟苓膏上桌。。。脑洞到这。

我们来把这个故事套入到装饰模式里去。上面故事里老板卖 3 种龟苓膏，而都不满足小红的需求，小红想要的是蜂蜜牛奶龟苓膏，如果用继承来实现龟苓膏，那就无法满足小红的要求了，因为继承直接固定了龟苓膏的做法，加什么就是什么，要蜂蜜牛奶龟苓膏，那就需要另外一个龟苓膏类代表蜂蜜牛奶龟苓膏；而用装饰模式则不同，下面看看装饰模式的实现代码。

龟苓膏抽象类，该类定义了制作龟苓膏的抽象方法。

/\*\*  
 \* 龟苓膏  
 \*/  
abstract class HerbalJelly {  
  
 /\*\*  
 \* 制作龟苓膏方法  
 \*/  
 public abstract void process();  
  
}

老板提供的最基本的龟苓膏，这种龟苓膏不加任何料，就是那苦苦的龟苓膏，我们称它为普通龟苓膏。

/\*\*  
 \* 普通龟苓膏  
 \*/  
class CommonHerbalJelly extends HerbalJelly {  
  
 @Override  
 public void process() {  
 System.out.println("盛一碗龟苓膏");  
 }  
   
}

另外 2 种龟苓膏：蜂蜜龟苓膏和牛奶龟苓膏，不是用继承实现，而是用装饰器实现，我们可以发现这 2 种龟苓膏都是基于上面普通龟苓膏添加不同的糖类食品制作而成。下面实现一个抽象类充当装饰器。

/\*\*  
 \* 龟苓膏装饰器  
 \*/  
abstract class Decorator extends HerbalJelly {  
  
 private HerbalJelly herbalJelly;  
  
 public Decorator(HerbalJelly herbalJelly) {  
 this.herbalJelly = herbalJelly;  
 }  
  
 @Override  
 public void process() {  
 this.herbalJelly.process();  
 }  
}

接下来就根据上面的龟苓膏装饰器来实现蜂蜜龟苓膏和牛奶龟苓膏。

/\*\*  
 \* 蜂蜜龟苓膏  
 \*/  
class HoneyHerbalJelly extends Decorator{  
  
 public HoneyHerbalJelly(HerbalJelly herbalJelly) {  
 super(herbalJelly);  
 }  
  
 @Override  
 public void process() {  
 super.process();  
 System.out.println("加蜂蜜");  
 }  
}  
  
/\*\*  
 \* 牛奶龟苓膏  
 \*/  
class MilkHerbalJelly extends Decorator{  
  
 public MilkHerbalJelly(HerbalJelly herbalJelly) {  
 super(herbalJelly);  
 }  
  
 @Override  
 public void process() {  
 super.process();  
 System.out.println("加牛奶");  
 }  
}

下面提供我们的测试代码，还记得上面说的，小明要了一碗蜂蜜龟苓膏，小红则要了一碗蜂蜜牛奶龟苓膏。

public class DecoratorTest {  
  
 public static void main(String[] args) {  
 CommonHerbalJelly commonHerbalJelly = new CommonHerbalJelly();  
 HoneyHerbalJelly honeyHerbalJelly = new HoneyHerbalJelly(commonHerbalJelly);  
 // 小明的蜂蜜龟苓膏  
 honeyHerbalJelly.process();  
  
 MilkHerbalJelly milkHerbalJelly = new MilkHerbalJelly(honeyHerbalJelly);  
 // 小红的蜂蜜牛奶龟苓膏  
 milkHerbalJelly.process();  
 }  
  
}  
  
打印结果：  
盛一碗龟苓膏  
加蜂蜜  
盛一碗龟苓膏  
加蜂蜜  
加牛奶

我们看到，小明的龟苓膏只加蜂蜜，小红的龟苓膏加了蜂蜜和牛奶，这样就很简单的满足了小红的要求。

### 4.4.3. 总结

装饰模式在一些类与类之间有叠加效应（也就是给一个类增加附加功能）的场景中非常好用，它可以说是继承的替代品，有更好的扩展性，也比较灵活。在 Java JDK 源码中也大面积用到了装饰模式，比如：java.io.BufferedInputStream(InputStream)。学完基础知识后，可以看看源码中是怎么实现的，巩固知识。

## 4.5. 外观模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.5.1. 简介

**姓名** ：外观模式

**英文名** ：Facade Pattern

**价值观** ：统一口径、一致对外

**个人介绍** ： Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. 要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。 （来自《设计模式之禅》）

### 4.5.2. 你要的故事

作为开发同学，我们平时打交道最多的就是需求同学和测试同学，公司小的时候，什么事情都全靠吼，工作也直接一对一，一个需求下来，需求同学先跟开发同学一起跟进这个需求，需求开发完成了，需求同学和测试同学沟通了需求的测试要点，测试同学就开测。这个过程中需求一直跟到上线。我们用代码来描述这个过程。

开发同学，负责开发需求。

/\*\*  
 \* 开发同学  
 \*/  
class Developer {  
  
 public void develop(String name) {  
 System.out.println("开发需求：" + name);  
 }  
  
}

测试同学，负责测试需求。

/\*\*  
 \* 测试同学  
 \*/  
class Tester {  
  
 public void test(String name) {  
 System.out.println("测试需求：" + name);  
 }  
  
}

需求同学，负责提需求，也负责跟进需求的开发、测试，直到上线。

/\*\*  
 \* 需求同学  
 \*/  
class Demander {  
  
 private Developer developer = new Developer();  
 private Tester tester = new Tester();  
  
 public void demand(String name) {  
 System.out.println("提需求：" + name);  
 developer.develop(name);  
 tester.test(name);  
 }  
  
}

测试代码。

public class FacadeTest {  
  
 public static void main(String[] args) {  
 Demander demander = new Demander();  
 demander.demand("开发一个跟淘宝一样的系统");  
 }  
}  
  
打印结果：  
提需求：开发一个跟淘宝一样的系统  
开发需求：开发一个跟淘宝一样的系统  
测试需求：开发一个跟淘宝一样的系统

公司小的时候，这样干没啥问题，咱关注的是业务的迭代速度和沟通成本，大家都是在一块办公，随时吼一声完事。当公司发展到一定程度，比如有 100 来人，其中需求 10 人、开发 70 人、测试 20 人，那就没法靠吼来沟通了，需要有一个比较规范化的沟通机制。一般会这样子引进，开发会把一些沟通能力较强、把控开发流程能力较好的同学升职为组长，负责保证一个需求的正常开发，他们会直接面对需求同学，直接沟通需求的开发要点，然后组长安排开发同学和测试同学跟进这个需求直到上线，也就是把需求同学以前的工作分配到开发组长，让他把控整个流程，这样就不会使得开发同学、测试同学、需求同学之间互相频繁沟通影响效率。这样子我们看看代码实现。

多了一个技术组长的类，负责跟进整个需求的开发测试过程。

/\*\*  
 \* 技术组长  
 \*/  
class Leader {  
  
 private Developer developer = new Developer();  
 private Tester tester = new Tester();  
  
 public void processDemand(String name) {  
 developer.develop(name);  
 tester.test(name);  
 }  
  
}

需求同学就不用直接和开发同学、测试同学沟通了，就跟技术组长对接就好。

/\*\*  
 \* 需求同学  
 \*/  
class Demander2 {  
  
 public Leader leader = new Leader();  
 public void demand(String name) {  
 System.out.println("提需求：" + name);  
 leader.processDemand(name);  
 }  
  
}

测试代码。

public class FacadeTest {  
  
 public static void main(String[] args) {  
 Demander2 demander2 = new Demander2();  
 demander2.demand("开发一个跟微信一样的系统");  
 }  
   
}  
  
打印结果：  
提需求：开发一个跟微信一样的系统  
开发需求：开发一个跟微信一样的系统  
测试需求：开发一个跟微信一样的系统

这个就是我们的外观模式，我们的技术组长就是外观模式的象征，他专门对外提供接收需求服务，然后安排需求给开发同学和测试同学，保证完成。

### 4.5.3. 总结

外观模式通过一个对外统一的接口，隐藏了内部的具体实现，使得外部系统可以更加简单的访问，也减少了外部系统对内部系统的依赖，从上面的例子讲，如果开发同学开发一半生病短时间无法来上班，交接给其他同学，由组长内部安排解决，需求同学并不需要知道。外观模式在微服务交互之间经常使用。

## 4.6. 享元模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.6.1. 简介

**姓名** ：享元模式

**英文名** ：Flyweight Pattern

**价值观** ：共享富贵

**个人介绍** ： Use sharing to support large numbers of fine-grained objects efficiently. 使用共享对象可有效地支持大量的细粒度的对象。 （来自《设计模式之禅》）

### 4.6.2. 你要的故事

还记得笔袋么？可能有人已经忘记了，在写这篇文章之前其实我也忘了，从初中开始就再也没用过笔袋。拿笔袋来讲享元模式再适合不过了。笔袋放各种各样的笔，今天我们不讲别的，就讲蜡笔。前段时间在逛公园的时候，看到一位老师在画画，画的就是蜡笔画，第一次看到真正的蜡笔画，真的很震撼，原来蜡笔也可以把景色画得那么美。当时偷偷拍了一张，看下图。

![](data:image/jpeg;charset=UTF-8;base64,)

我们就拿这幅画来说，里面画了草、树、路、山、天空等等。如果没有用享元模式，我们可能这样子实现。

蜡笔接口。

interface ICrayon {  
  
 void draw(String place);  
  
}

蜡笔。

/\*\*  
 \* 蜡笔  
 \*/  
class Crayon implements ICrayon {  
  
 private String color;  
  
 public Crayon(String color) {  
 System.out.println("---新建【" + color + "】蜡笔" );  
 this.color = color;  
 }  
  
 @Override  
 public void draw(String place) {  
 System.out.println("用" + this.color + "蜡笔画" + place);  
 }  
}

测试代码。这幅画是小明和小红一起画，小明画了草和路，小红画了树和蓝天。

public class NoFlyweightTest {  
  
 public static void main(String[] args) {  
 drawByXiaoMing();  
 drawByXiaoHong();  
 }  
  
 public static void drawByXiaoMing() {  
 ICrayon greenCrayon = new Crayon("绿色");  
 greenCrayon.draw("草");  
  
 ICrayon grayCrayon = new Crayon("灰色");  
 grayCrayon.draw("路");  
 }  
  
 public static void drawByXiaoHong() {  
 ICrayon blueCrayon = new Crayon("蓝色");  
 blueCrayon.draw("蓝天");  
  
 ICrayon greenCrayon = new Crayon("绿色");  
 greenCrayon.draw("树");  
 }  
  
}  
  
打印结果：  
---新建【绿色】蜡笔  
用绿色蜡笔画草  
---新建【灰色】蜡笔  
用灰色蜡笔画路  
---新建【蓝色】蜡笔  
用蓝色蜡笔画蓝天  
---新建【绿色】蜡笔  
用绿色蜡笔画树

我们发现小明和小红都用了绿色蜡笔，而这里新建了 2 次绿色蜡笔，也就是在整个作画过程中，小明和小红并不是共用一套蜡笔，而是各自用一套蜡笔，在现实中是没什么问题的，但是在软件开发中，如果这种情况出现，其实相当于资源浪费，因为每个蜡笔都会占用内存，可以共用的我们尽量共用，节省一些内存空间，特别是出现很多这种可以共享却没有共享的对象时候。下面我们引入享元模式。享元模式实现方法相当于我们蜡笔都放在了笔袋，小明和小红用完就放到笔袋里面，每一种颜色的蜡笔只有一根，也就是他们共用一套蜡笔。代码如下所示。

笔袋代码。我们用了 Map 作为容器，如果容器里面没有想要颜色的蜡笔，则创建新的蜡笔，并存到容器里。

/\*\*  
 \* 笔袋  
 \*/  
class CrayonFactory {  
  
 private static Map<String, ICrayon> data = new HashMap<>();  
  
 public static ICrayon getCrayon(String color) {  
 if (data.containsKey(color)) {  
 return data.get(color);  
 }  
 ICrayon crayon = new Crayon(color);  
 data.put(color, crayon);  
 return crayon;  
 }  
  
}

测试代码。

public class FlyweightTest {  
  
 public static void main(String[] args) {  
 drawByXiaoMing();  
 drawByXiaoHong();  
 }  
  
 public static void drawByXiaoMing() {  
 ICrayon greenCrayon = CrayonFactory.getCrayon("绿色");  
 greenCrayon.draw("草");  
  
 ICrayon grayCrayon = CrayonFactory.getCrayon("灰色");  
 grayCrayon.draw("路");  
 }  
  
 public static void drawByXiaoHong() {  
 ICrayon blueCrayon = CrayonFactory.getCrayon("蓝色");  
 blueCrayon.draw("蓝天");  
  
 ICrayon greenCrayon = CrayonFactory.getCrayon("绿色");  
 greenCrayon.draw("树");  
 }  
}  
  
打印结果：  
---新建【绿色】蜡笔  
用绿色蜡笔画草  
---新建【灰色】蜡笔  
用灰色蜡笔画路  
---新建【蓝色】蜡笔  
用蓝色蜡笔画蓝天  
用绿色蜡笔画树

利用享元模式实现的结果，小红画树所用到的绿色蜡笔跟小明画草的绿色蜡笔一样，小红用到时并没有新建绿色蜡笔。

### 4.6.3. 总结

是不是有一种，原来这就是享元模式的感觉？平时开发过程中经常见到这种因为很多重复的对象，所以利用享元模式来实现的场景。享元模式合理提高了对象的复用性，减少了程序的内存占用，还有一个提高性能的地方就是减少了对象创建的过程。好了，收下这个简单的设计模式。欢迎关注公众号，一起学习进步。

## 4.7. 代理模式

![](data:image/jpeg;charset=UTF-8;base64,)

### 4.7.1. 简介

**姓名** ：代理模式

**英文名** ：Proxy Pattern

**价值观** ：为生活加点料

**个人介绍** ： Provide a surrogate or placeholder for another object to control access to it. 为其他对象提供一种代理以控制对这个对象的访问。 （来自《设计模式之禅》）

### 4.7.2. 你要的故事

咱们从事 IT 行业，随时都可能上网查东西，如果网络速度慢或者网络访问受限制，那是相当的折磨，忍无可忍。而咱在国内网络比较特殊，有个墙围着，俗称防火长城。今天讲到代理模式，就来讲讲这道墙。这墙是这么实现的，我们上网，正常的网络是世界各地的网站我们都能访问，而加上这道墙，相当于在我们上网的时候做了一层代理，这一层代理把禁用的网站给过滤掉，使得我们没法访问被禁用的网站。下面通过代码来讲解。

定义一个互联网接口，里面有一个访问网站的通用方法 access。

/\*\*  
 \* 互联网  
 \*/  
interface Internet {  
  
 String access(String domain);  
  
}

定义世界范围内的网络类，可以访问任何存在的网站。

/\*\*  
 \* 世界网络  
 \*/  
class WorldNetwork implements Internet {  
  
 @Override  
 public String access(String domain) {  
 System.out.println("访问网站：" + domain);  
 return domain + "网站内容";  
 }  
  
}

定义中国的网络类，就是代理类，实现墙的功能。disable 对象存储了在国内禁止访问的网站，用户在访问网站时（也就是调用 access 访问）先判断网站是不是在禁用的网站集合里面，如果是则禁用，如果不是则继续访问。

/\*\*  
 \* 中国网络（就是代理）  
 \*/  
class ChinnessNetwork implements Internet {  
  
 private Set<String> disable;  
  
 private Internet internet;  
  
 public ChinnessNetwork(Internet internet) {  
 this.internet = internet;  
 this.disable = new HashSet<>();  
 this.disable.add("www.google.com");  
 this.disable.add("www.facebook.com");  
 }  
  
 @Override  
 public String access(String domain) {  
 if (disable.contains(domain)) {  
 System.out.println("禁止访问该网站：" + domain);  
 return "禁止访问该网站：" + domain;  
 }  
 return internet.access(domain);  
 }  
}

测试代码，ChinnessNetwork 作为代理类，WorldNetwork 作为被代理类。

public class ProxyTest {  
  
 public static void main(String[] args) {  
 WorldNetwork worldNetwork = new WorldNetwork();  
 ChinnessNetwork chinnessNetwork = new ChinnessNetwork(worldNetwork);  
 chinnessNetwork.access("www.google.com");  
 chinnessNetwork.access("www.baidu.com");  
 }  
  
}  
  
打印结果：  
禁止访问该网站：www.google.com  
访问网站：www.baidu.com

看到结果，万能的 google 被屏蔽在外，我们只能默默用 baidu。

### 4.7.3. 总结

代理模式简单说就是在原来对象的功能基础上加上额外的功能，在工作开发中这个很好用，比如我们要统计系统中各方法执行的时间，就可以用代理模式来实现。开源框架中也用得很多，比如 Spring 的 AOP 等等。

上面简单的分享了代理模式，也是最简单的一个代理模式，名称叫静态代理。在开发中用的最多的是动态代理，基于这篇文章主要讲述设计模式的基础内容，后面找个机会再分享动态代理的内容。