# 设计模式

## 一、Design Pattern的基本概念

设计模式是一套被反复使用的、代码设计经验的总结。

**解耦**：就是降低类与类之间的依赖关系。

#### ⑴使用设计模式的目的：

**①重用性**：提高代码的重用性

**②可读性**：使代码便于他人阅读

**③可扩展性**：便于代码新增新功能

**④可靠性**：功能新增功能后，对原来代码不会造成影响。

**⑤高内聚、低耦合**

设计模式的参考书Design Patterns - Elements of Reusable Object-Oriented Software（中文译名：设计模式 - 可复用的面向对象软件元素），这本书的四位作者合称(Gang of Four 即**GOF**)，他们提出的设计模式主要是基于以下的面向对象设计原则：**对接口编程而不是对实现编程，优先使用对象组合而不是继承**。

设计模式总共有23种，可分为三大类：**创建型模式(Creational Patterns)、结构型模式(Structural Patterns)、行为型模式(Behavioral Patterns)**。

#### ⑵设计模式的七大原则：

**①开闭原则(Open Close Priciple)**：(Open for Extension,Close for Modification)对扩展开放，对修改关闭。即在程序需要进行拓展的时候，不能去修改原有的代码。

**②里氏代换原则(Liskov Substitution Principle)**：任何基类可以出现的地方，子类一定可以出现。继承时，子类尽量不要重写父类的方法。

**③依赖倒转原则(Dependence Inversion Priciple)**：针对接口编程，依赖于抽象而不依赖于具体。底层模块尽量有接口或抽象类。

**④接口隔离原则(Interface Segregation Priciple)**：使用多个隔离的接口，比使用单个接口要好。降低类之间的耦合度。

**⑤迪米特法则(Demeter Priciple)**：也称最小知道原则。一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

**⑥合成复用原则(Composite Reuse Priciple)**：在系统中尽量使用关联关系代替继承关系。

**⑦单一职责(Single Responsibility Priciple)**：一个类只负责一个职责。

## 二、创建型模式

**提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象**。

可分为：**原型模式(Prototype Priciple)、单例模式(Singleton Pattern)、工厂模式(Factory Pattern)、抽象工厂模式(Abstract Factory Pattern)、建造者模式(Builder Pattern)**。其中工厂模式可分为**简单工厂模式(Simple Factory Pattern)和工厂方法模式(Factory Method Pattern)**。

### 1.单例模式

单例模式是**一个类只允许产生一个实例化对象**，该类自己创建自己的唯一实例，该类给所有其它对象提供这一实例。

单例模式主要解决一个类频繁的创建和销毁问题。使用单例模式的情况：控制实例数目，节省系统资源的时候。

**单例模式的类的构造器是私有的，且有一个自身的静态实例**。

**单例模式的优缺点：**

**优点**：在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例；避免对资源的多重占用。

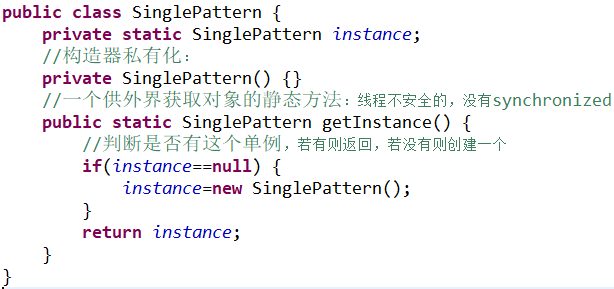
**缺点**：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

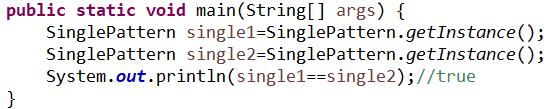
注意一个类的对象的实例化若很耗费资源，则要采用延迟加载(lazy loading)的方式，即不能让该类一加载就实例化对象。

单例模式的实现有多种方式：如下

#### ⑴线程不安全的懒汉式

下面这种方式是线程不安全的，不能在多线程中使用。是延迟加载的方式。



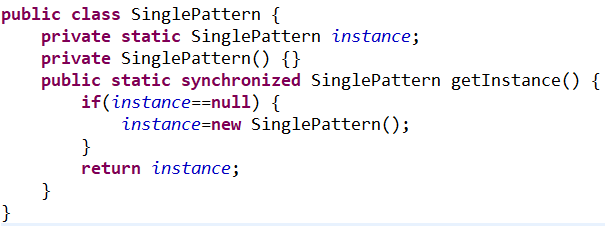


这种方式不算严格意义上的单例模式。如当多个线程同时调用getInstance()方法时，可能会实例化多次并被不同的对象拥有，从而违背了单例的初衷。例子：

|  |  |  |
| --- | --- | --- |
| time | ThreadA | ThreadB |
| t1 | 调用getInstance()方法，发现instance为空 |  |
| t2 |  | 调用getInstance()方法，发现instance为空 |
| t3 |  | 初始化instance对象 |
| t4 |  | 返回instance对象 |
| t5 | 初始化instance对象 |  |
| t6 | 返回instance对象 |  |

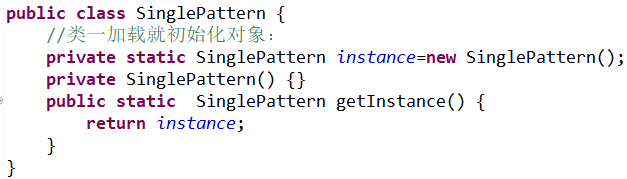
#### ⑵线程安全的懒汉式

这种方式是延迟加载的方式。不推荐使用这种，加了synchronized虽然保证了线程安全和单例，但导致性能开销很大，效率太低。并且加锁其实只需要在第一次初始化的时候用到，之后的调用都没必要再进行加锁。



#### ⑶饿汉式(线程安全的)

下面这种虽然没使用synchronized加锁，但是是线程安全的。这是因为类加载机制(class Loader机制)中一个类只会执行一次静态区域的内容。这种方式没有加锁，执行效率会提高，但是类加载就初始化对象，浪费了内存。不是延迟加载的方式。

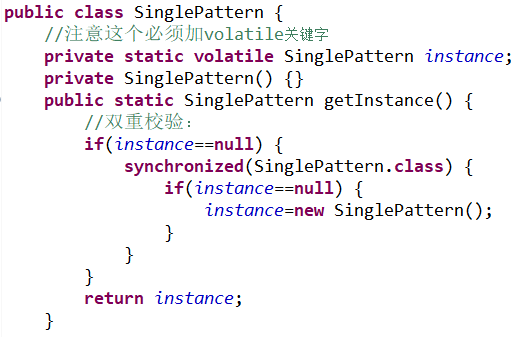


#### ⑷双重校验锁(DCL)

双重校验锁(DCL)：double-checked locking。这种方式采用双锁机制，安全且在多线程情况下能保持高性能。是延迟加载的方式。

这种方式是对懒汉式(线程安全和线程不安全的)的一种优化。这是**先判断对象是否被初始化，若被初始化则直接返回该对象。若没有初始化，则加锁，再次判断对象是否初始化，若没有则创建该对象**。

执行双重检查是因为多个线程同时了通过了第一次检查，并且其中一个线程首先通过了第二次检查并实例化了对象，那么剩余通过了第一次检查的线程就不会再去实例化对象，这种情况除了初始化的时候会出现加锁的情况，后续的所有调用都会避免加锁而直接返回，解决了性能消耗的问题。



疑问这里为什么锁SinglePattern.class而不是锁this。

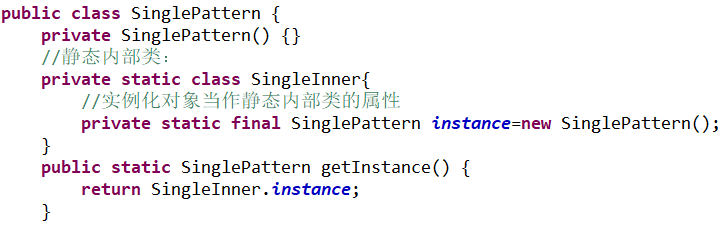
注意上面若不加volatile关键字，则有一个安全隐患。一般实例化一个对象，如上面代码中的instance=new SinglePattern()这个过程的运行步骤可分解为三步：①先分配内存空间，②然后初始化对象，③然后将对象引用指向刚分配的内存空间。 但是有些编译器为了性能的原因，可能会将第二步和第三步进行**重排序**，变成先分配内存空间，然后将对象引用指向刚分配的内存空间，然后再初始化对象。考虑重排序之后，若多个线程进行访问，可可能出现下面这种情况。

|  |  |  |
| --- | --- | --- |
| time | ThreadA | ThreadB |
| t1 | 调用getInstance()方法，发现instance为空 |  |
| t2 | 获得锁 |  |
| t3 | 再次检测到instance为空 |  |
| t4 | 为对象分配内存空间 |  |
| t5 | 将instance指向内存空间 |  |
| t6 |  | 检测到instance不为空 |
| t7 |  | 访问instance(此时对象还未初始化完成) |
| t8 | 初始化对象 |  |

这种情况，在t7时刻线程B访问的是一个初始化还未完成的instance对象。使用volatile关键字可以解决上面的问题，使用了volatile关键字后，重排序被禁止，所有的写（write）操作都将发生在读（read）操作之前。

#### ⑸登记式/静态内部类

下面这种方式同样是采用classLoader机制保证实例化对象只有一个。注意这种方式和饿汉式的区别，饿汉式是类一加载了就会实例化对象，没有达到lazy loading的效果。而下面这种方式SinglePattern类一加载并不会马上实例化instance对象，因为静态内部类没有被主动使用，当调用getInstance()方法时，才会加载给静态内部类，从而实例化instance，这种方式达到了延迟加载(lazy loading)的目的。



⑹其它：使用枚举、ThreadLocal、CAS锁等实现。有空再学。

一般来说，不建议使用第一、二种懒汉式的单例模式，建议使用饿汉式的。对于双重检查锁的单例模式，在某些情况下可能DCL会失效，可以使用静态内部类的单例模式替换。

### 2.原型模式

**原型模式是使用原型实例指定将要创建的对象类型，通过复制这个实例创建新的对象。**

**使用情形**：有些对象的创建比较复杂，且需要频繁的创建，可以使用一个原型对象来指定要创建对象的类型，然后采用复制这个对象的方式创建更多同类型的对象。或者向客户隐藏实例创建的创建过程的场景，提供客户创建未知类型对象的选择。

原型模式是通过给出一个原型对象来指定要创建的对象类型，然后自身实现Cloneable接口重写clone()

**实现原型类的三个条件**：

**①实现Cloneable接口**：实现了Cloneable接口就是告诉虚拟机可以安全的在这个类上使用clone()方法。在 JVM 中，只有实现了 Cloneable 接口的类才可以被拷贝，否则会抛出 CloneNotSupportedException 异常。

**②重写Object类的clone()方法**：Object有一个clone()方法，作用是返回一个对象的拷贝。

**③在重写的clone()方法中调用super.clone()，同时将权限修改为public**：默认情况下，类不具备复制对象的能力，需要调用 super.clone() 来实现。

**采用克隆的方式创建新对象的话，就不再需要通过new实例化对象了。因为Object中的clone()方法是一个native本地方法，它可以直接操作内存中的二进制流，所以性能相对于new实例化来说更佳**。

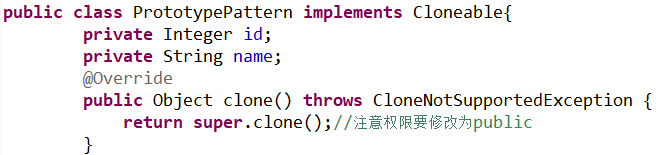
[..\3.javaSE基础\4.集合.docx](file:///F:\1.java笔记（和盈）\3.javaSE基础\4.集合.docx)

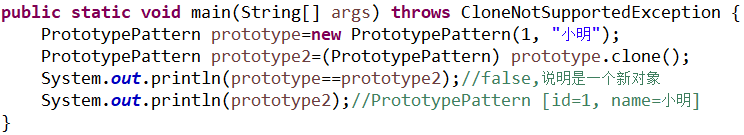


下面暂时忽略深克隆、浅克隆的问题。

当复制大对象时，采用clone的方式创建对象和采用new创建对象，性能差别就很大。

**例子：**





**在下面这些情况下，需要根据现有的实例来生成新的实例：**

①对象种类繁多，无法将他们整合到一个类的时候;

**② 难以根据类生成实例时；**

③想解耦框架与生成的实例时;

想要让生成实例的框架不再依赖于具体的类，这时，不能指定类名来生成实例，而要事先“注册”一个“原型”实例，然后通过复制该实例来生成新的实例。

**原型模式的优缺点：**

**优点：**

**①**创建新对象的实例较复杂时，使用原型模式可以简化创建对象的创建过程，提高创建对象的效率。

**②**可以动态增加或减少产品类。

**③**可以使用深克隆的方式保存对象的状态。

**缺点：**

**①**实现深克隆时需要编写较复杂的代码。

**②**需要为每个类实现Cloneable接口，重写Object类的clone()方法。对新的类来说较好实现，若对于已存在的类，则需要修改其源代码，这违背了”开闭原则”。

**注意使用clone方法创建的新对象的构造函数是不会被执行的**，即会绕过任何构造函数(有参和无参)，因为clone方法的原理是从堆内存中以二进制流的方式进行拷贝，直接分配一块新内存。

很多软件提供的复制、粘贴操作就是原型模式的应用。

### 3.工厂模式

**工厂模式可分为三类：简单工厂模式(Simple Factory Pattern)、工厂方法模式(Factory Method Pattern)、抽象工厂模式(Abstract Factory Pattern)**。

 工厂模式主要是为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

GOF在《设计模式》书中将工厂模式分为工厂方法模式和抽象工厂模式，将简单工厂模式看成是工厂方法模式的一种特例。

#### ⑴简单工厂模式

**就是建立一个工厂类，对实现了同一接口或继承自同一抽象类的一些类进行实例的创建。**

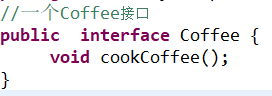
**组成部分：**

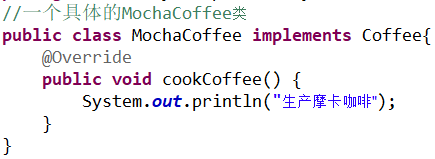
**①抽象产品角色**：一般是具体产品类继承的抽象类或是实现的接口。里面有一些抽象方法等待具体类去重写。

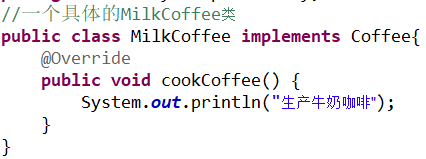
**②具体产品角色**：java中一个具体的类，工厂类中创建的对象就是具体类的对象。

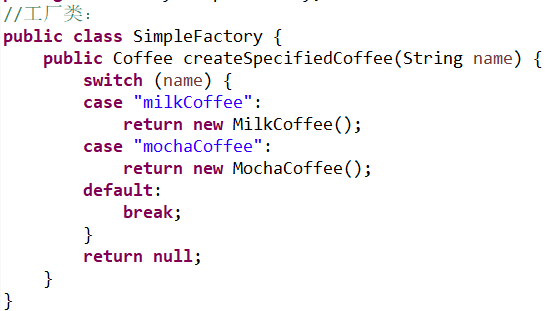
**③工厂角色**：**是简单工厂模式的核心**，用来创建具体的产品类。

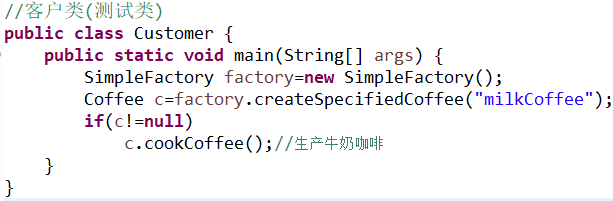
**例子：**









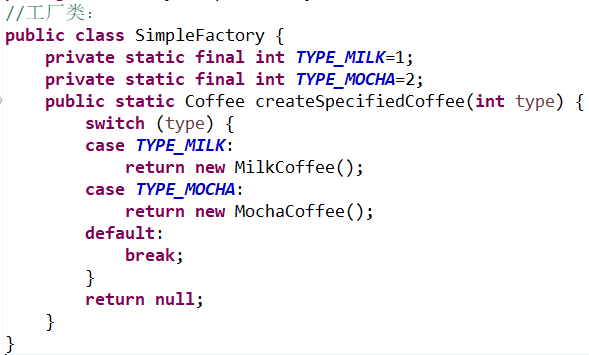


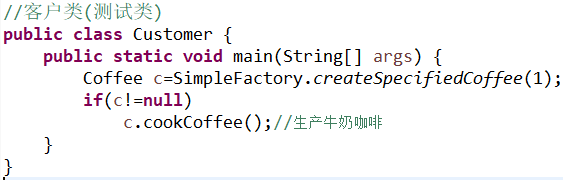
从开闭原则(对扩展开放，对修改封闭)分析简单工厂模式。对于具体产品类角色来说，当要生产新的具体Coffee类时，只需将新Coffee类实现Coffee抽象类重写方法即可，这是符合开闭原则的。而对于工厂类来说，则要修改工厂类的源码，再添加一个对应的case子句，这是不符合开闭原则的。

简单工厂模式也称为静态工厂模式，这是因为工厂中如上面的createSpecifiedCoffee()方法通常是静态方法，这样工厂在调用方法创建具体产品类的对象时则不用创建工厂对象。

简单工厂模式的**缺点**：使用时如果传递的参数出错，则不能创建具体产品的对象，容错率不高，同时当有新产品加入到系统中时，必须修改工厂类，这不符合开闭原则。

也可将上面工厂类的代码改成下面这种：





#### ⑵工厂方法模式

**也称为多态工厂模式**

**组成部分：**

**①抽象工厂角色**：是工厂方法模式的核心。具体工厂角色必须实现的接口或者必须继承的抽象类。

**②具体工厂角色**：一般用来创建具体产品角色的实例对象。

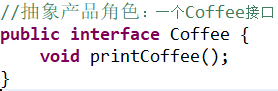
**③抽象产品角色**：它是具体产品继承的抽象类或者是实现的接口。

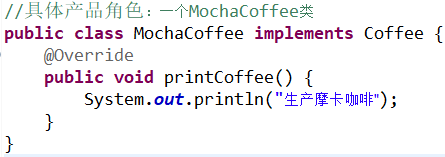
**④具体产品角色**：具体工厂角色所创建的对象就是此角色的对象。

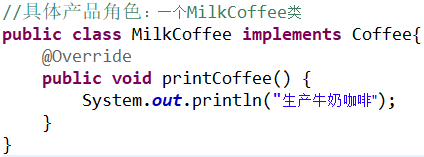
工厂方法模式可以允许系统在不修改抽象工厂的角色下添加新产品，只需添加一个具体工厂角色和具体产品角色即可**，这很好的符合了开闭原则。**

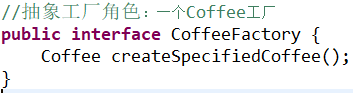
**工厂模式一般包含四种角色，抽象工厂角色、具体工厂角色、抽象产品角色、具体产品角色。其中抽象工厂角色中定义了一个用来创建单一产品的抽象方法，而其子类具体工厂角色则重写这个方法用来创建对应的具体产品类，即具体产品类的创建是由具体工厂类实现的。**

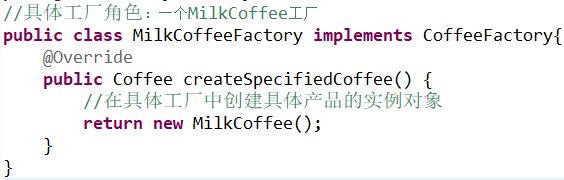
**例子：**

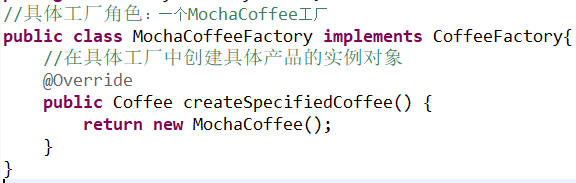


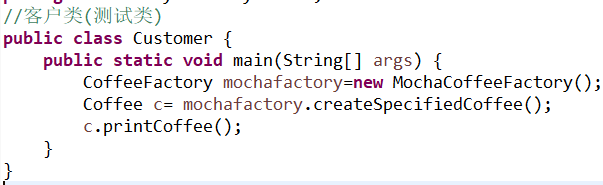












**工厂模式的优缺点：**

**优点：**

**①**在系统添加新产品时，无需修改抽象工厂和抽象产品的类或接口，也无需修改其它具体工厂和具体产品类，只需再添加一个具体工厂和具体产品类即可，符合开闭原则。

**②**工厂角色和产品角色是采用多态的方式。

**③**封装了具体产品对象的创建细节。工厂方法用来创建客户需要的产品对象时，向客户隐藏了哪种具体产品类将被实例化，客户无需关心产品的创建细节，只需知道具体产品对应的具体工厂即可。

**缺点：**

当需要添加很多新产品时，则需要添加多个具体产品类和多个具体工厂类，系统中类的个数成对增加，导致有更多的类需要编译和运行，给系统带来了额外的开销。

若抽象工厂的子只有一个具体工厂，则可以把抽象工厂省略，然后在具体工厂中创建所有产品的对象，并将工厂方法设置为静态方法，这时工厂方法模式则退化成了简单工厂模式。

适用的情形：有空再补充。

### 4.抽象工厂模式

**产品等级结构**：即**产品的继承结构**。比如一个抽象电脑，其子类有小米电脑、联想电脑、华为电脑。则抽象电脑和具体的电脑直接构成了一个产品等级结构。同一类产品。

**产品族**：**在抽象工厂模式中，产品族是指同一个工厂生产的位于不同产品等级结构中的一组产品**。比如华为工厂生产华为手机、华为电脑，其中华为电脑和抽象电脑之间构成一个产品等级结构，而华为手机和抽象手机之间构成另一个产品等级结构，华为手机和华为电脑属于一个产品族。同一个工厂生产的不同类型的产品。

**使用情形**：**①**当工厂需要生产的具体产品对象不止一个时，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式**。②**系统中有多于一个的产品族，而每次使用只使用某一个产品族。**③**属于同一个产品族中的产品在一起使用。

**抽象工厂模式**和**工厂方法模式**的最大**区别**：工厂方法模式针对的是一个产品等级结构，每个具体工厂只生产一个具体产品对象。抽象工厂模式针对的是多个产品等级结构，每个具体工厂生产多个具体产品对象，且具体产品对象不是同一个类型的。工厂方法模式可以看成单产品系列，抽象工厂模式看成多产品系列。

抽象工厂模式的定义：提供一个创建一系列相关或相互依赖对象的接口，而无需指定具体的类。也称Kit模式。

**组成部分：**

**①抽象工厂角色**：是工厂方法模式的核心。一般声明多个用来生成抽象产品的方法。

**②具体工厂角色**：是抽象工厂角色的子类。重写了抽象工厂中声明的生成抽象产品的方法，生成一组具体产品的实例对象，这些产品构成一个产品族，每一个产品都位于某个产品等级结构中。 每个具体工厂创建多个不同类型的具体产品对象。

**③抽象产品角色**：它是具体产品继承的抽象类或者是实现的接口。定义了产品的抽象业务方法。

**④具体产品角色**：具体工厂角色所创建的对象就是此角色的对象。实现了抽象产品接口中定义的业务方法，定义具体工厂生产的具体产品对象。

**抽象工厂模式的优缺点：**

**优点：**

**①**隔离了具体类的生成，使客户不需要知道什么被创建。

**②**使用抽象工厂模式可实现高内聚低耦合的目的。

**③**当一个产品族中的多个对象被设计成一起工作时，能保证客户端始终只使用同一个产品族中的对象。

**④**增加新的具体工厂和产品族很方便，无需修改已有系统中代码，符合开闭原则。

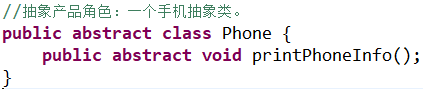
**缺点：**

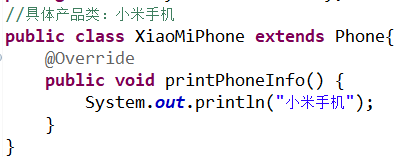
添加新的产品对象时，难以扩展抽象工厂来生产新种类的产品。

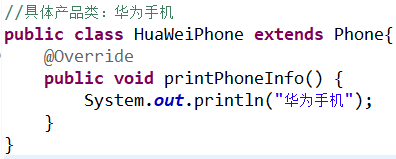
开闭原则的倾斜性：增加新的具体工厂和产品族容易，增加新的产品等级结构麻烦。

**当抽象工厂模式中每一个具体工厂类只创建一个产品对象，即只存在一个产品等级结构时，抽象工厂模式退化成了工厂方法模式**。

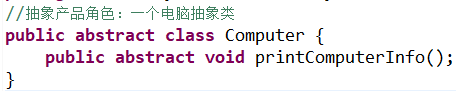
**例子：**

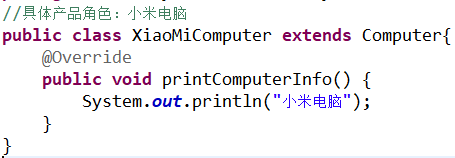


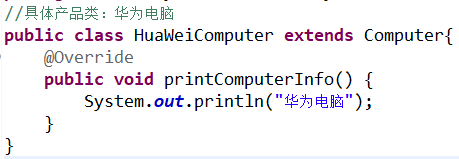




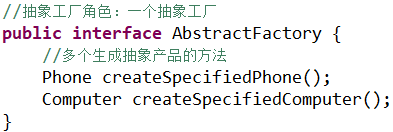
小米手机、华为手机、抽象手机一起构成产品等级结构，这是同一类产品。

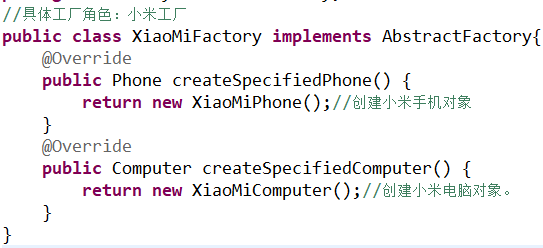


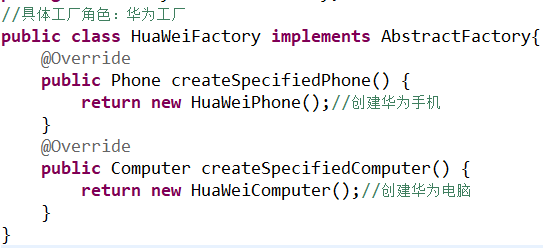




小米电脑、华为电脑、抽象电脑一起构成产品等级结构，这是同一类产品。







小米手机和小米电脑都是在小米工厂生产的，都属于小米的产品，小米手机和小米电脑组成一个产品族。

华为手机和华为电脑都是在华为工厂生产的，都属于华为的产品，华为手机和华为电脑组成一个产品族。



### 5.建造者模式

建造者返回给客户端的是一个已经建造完毕的完整产品对象，而用户无需关心该对象包含的属性以及它们的组装方式。

建造者模式也称**生成器模式**。

**建造者模式的定义**：**将一个复杂对象的构建和对象的表现分离开来，使得同样的构建过程可以创建出不同的产品对象**。

#### ⑴组成部分：

**①抽象建造者**：Builder

声明产品需要的部件。这是一个接口或抽象类，只是用来定义一个规范，不涉及具体的建造。

**②具体建造者**：ConcreteBuilder

继承抽象建造者，实现各部件的方法。

**③指挥者**：Director

负责安排负责对象的建造次序，指挥者和抽象建造者相互关联，在指挥者中调用抽象建造者的相关的方法，返回一个完整的对象。

**④产品角色**：Product

具体的产品对象，由多个部件组成。

**指挥者类的作用**：**①**隔离了客户与生产过程**②**负责控制产品的生成过程。

指挥者针对抽象建造者编程，客户端只需要知道具体建造者的类型，即可通过指挥者调用对应的方法返回一个完整的产品对象。

#### ⑵建造者模式的优缺点：

**优点：**

**①**客户端不必知道产品的内部细节，将产品本身和产品的创建过程解耦，使得相同的创建过程可以创建不同的对象。

**②**每一个具体建造者相互独立，可以很方便替换或增加具体建造者。用户使用不同的建造者就可以得到不同的产品对象。

自己感觉①②两点的优点问题。

**③**可以更加精细的控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使创建过程更清晰。

**④**增加新的具体建造者无需修改原来的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合开闭原则。

**缺点：**

**①**产品必须有共同点，使用范围有限。若产品之间的差异性大，则不适合创建者模式。

**②**若产品内部发生变化，则建造者都要修改，成本较大。若产品内部变化复杂，则会有很多的具体建造者。

#### ⑶适用情形：

**①**需要生成的产品对象有复杂的内部结构，即包含多个成员属性。

**②**需要生成的产品对象属性相互依赖，**需要指定其生成顺序**。

**③对象的创建过程独立于创建该对象的类**。在建造者模式中，将创建过程封装到指挥者中而不是建造者类中。

**④**隔离复杂对象的创建和使用，并使**相同的创建过程可以创建不同的产品对象**。

**应用**：比如游戏开发中通过不同的具体创造者创建不同类型的地图或人物。比如java中的StringBuilder。

建造者模式的简化：若建造者中只有一个具体建造者话，可省略抽象建造者，同时也可省略指挥者，让建造者角色扮演指挥者和建造者双重角色。

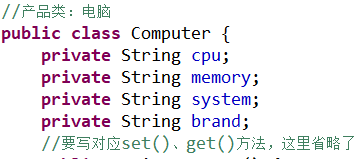
#### ⑸建造者模式和抽象工厂模式的区别：

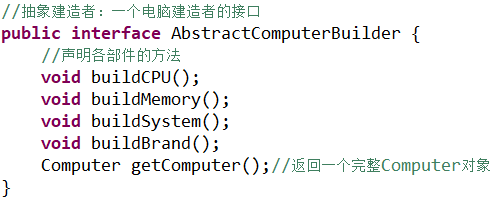
**①**抽象工厂模式返回许多产品对象，每个产品对象分别位于不同的产品等级结构中，这些产品一起组成了一个产品族。而建造者模式返回的是一个组装好的完整产品。

**②**抽象工厂模式中，客户实例化具体工厂类，然后调用相应的工厂方法获取所需的产品对象。而建造者模式中，则是通过指挥者类指导如何生成对象，包括对象的组装过程和建造步骤，它侧重于一步步构造一个复杂对象，然后返回一个完整的对象。

#### ⑹例子：

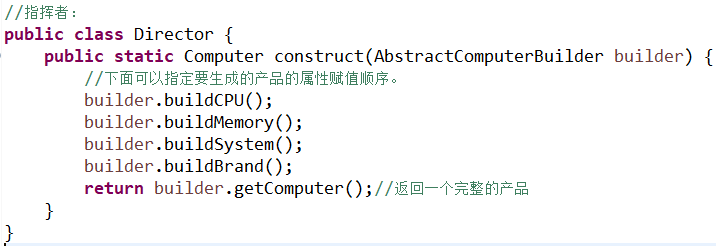
**例子1(建造者模型)：**

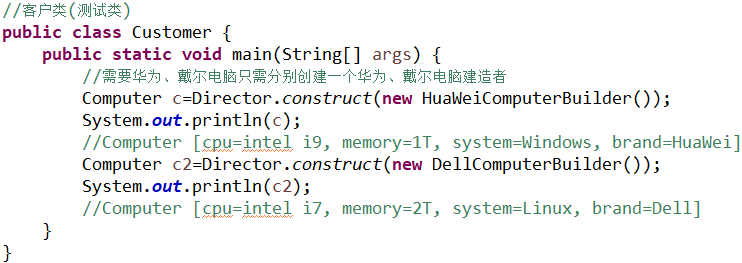






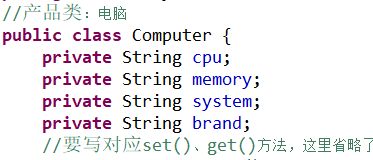


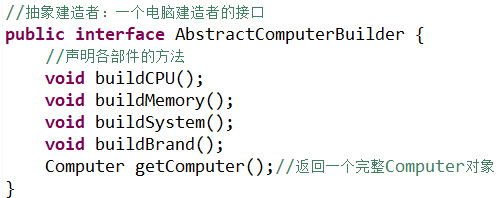


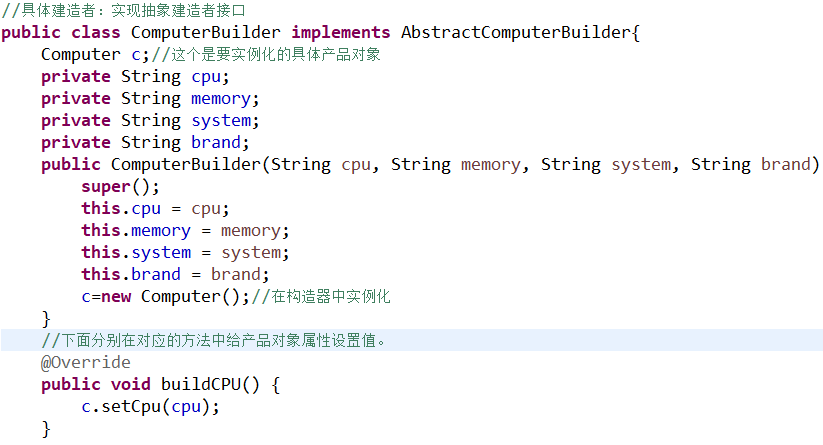


**例子2：**

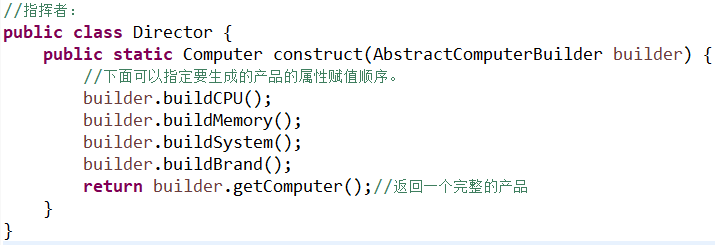
**这个是自己改进的建造者模式，(对于这个是否是建造者模式存疑)**

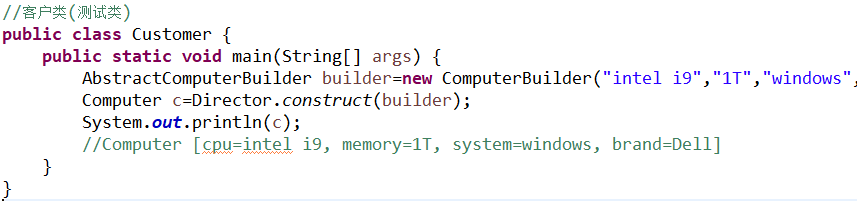








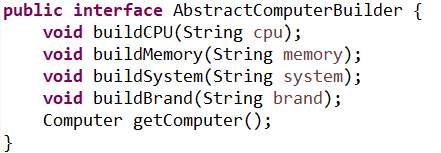


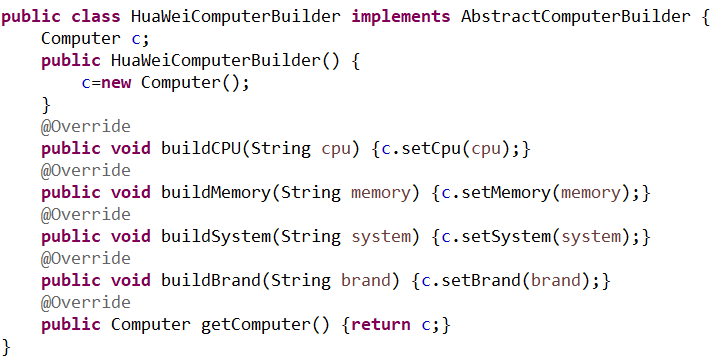


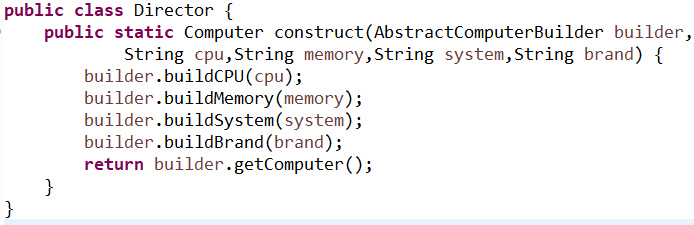
例1是网上标准的建造者模式，完全符合上面对建造者模式的描述。用户使用不同的具体建造者对象，就可以创建不同的产品。但是有个疑问，就是如果具体建造者中初始化属性直接写死的话，对于同一种产品只是属性值不同，则又要创建一个具体建造者，这样会导致具体建造者过多。

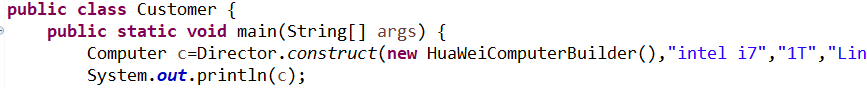
例2是自己改进的建造者模式，给具体建造者创建多个和产品的属性对应的属性，然后采用这些属性给产品赋值。这样在客户需要一个指定的产品时，则需要创建一个指定参数的具体建造者，这样虽然可以解决例1中的问题，但是无法向客户隐藏产品的内部细节，对于上面建造者的优点中前两条则不符合，感觉这样写就有点不想建造者模式。

也可将上面例2中抽象建造者、具体建造者者、指挥者的代码修改成如下：









## 三、结构型模式

**结构型模式(Structural Pattern)是如何将类或者对象结合在一起形成更大的结构**。

**结构型模式**可分为**类结构型模式**和**对象结构型模式**。

**类结构型模式**关心类的组合，这种模式中一般只存在继承关系和实现关系。

**对象结构型模式**关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。根据合成复用原则，系统中尽量使用关联关系而不是继承关系，所以大部分结构型模式都是对象结构模式。

结构型模式有多种，常用的有**适配器模式(Adapter Pattern)、代理模式(Proxy Pattern)、组合模式(Composite Pattern)、外观模式(Facade Pattern)**等。

### 1.适配器模式

适配器模式的**定义**：**将一个接口转换成一个客户希望的接口，适配器模式可以使原本由于接口不兼容而不能一起工作的类可以一起工作**。

在适配器模式中定义一个包装类，包装不兼容接口的对象，这个包装类就是**适配器(Adapter)**，它所包装的对象就是**适配者(Adaptee)，**即**被适配的类**。

适配器模式可分为**类适配器模式**、**对象适配器模式、接口适配器模式。**

#### ⑴组成部分：

**①目标抽象类**：**Target**，期待得到的目标

**②适配器类**：**Adapter**。相当于一个中间者。在适配器类中将目标抽象类和适配者类关联起来。

**③适配者类**：**Adaptee**，需要被适配的对象或类型

**④客户类**：Client

例子：比如一个电饭煲的插头是三脚插头，现有一个插座是两孔插座，要想三脚插头插在两孔插座上，需要一个中间媒介即插头转换器，这个插头转换器就相当于适配器，而三角插头就相当于适配者，两孔插座则相当于抽象目标。

**类适配器模式中，target是客户所期待的接口，target只能是接口，不能为类，即不能为抽象类或具体类，adaptee是需要适配的类，adapter则是实现了抽象目标类接口target，并继承了适配者类adaptee**。

**对象适配器模式中，target是客户所期待的接口，target可以是具体类、抽象类、接口。adaptee则是需要适配的类，adapter实现或继承target抽象目标接口(或者类)，然后在adapter内部包装一个适配者类adaptee。**

**类适配器模式的适配器和适配者是继承关系，和目标抽象类接口则是实现关系；对象适配器模式的适配器和适配者则是引用(也可称为组合)关系，和目标抽象类则可以是继承或实现关系**。

**接口适配器模式**：也称缺省适配器模式。当不想实现一个接口中的所有方法时，可以先设计一个类用来继承该接口，对接口中的所有方法进行重写，重写的方法是都是空的，这样子类需要使用接口中的个别方法时，只需继承这个类，重写需要的方法即可，而不用重写接口中所有的方法。

#### ⑵适配器模式的优缺点：

**优点：**

**①将目标类和适配者类解耦。**通过引入一个适配器类来重用现有的适配者类，而无需修改原来的代码。

**②增加了类的透明性和复用性**。将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。

**③灵活性和扩展性都很好**。可在不更改原有代码的基础上增加新的适配器类，符合开闭原则。

**④可以让任何两个没有关联的类一起运行**。

**类适配器模式还有如下优点**：

该模式下适配器类是适配者类的子类，因此可在适配器类中置换一些适配者类的方法，使得适配器的灵活性更强。

**类适配器模式的缺点：**

java对于类来说不支持多继承，一次最多只能适配一个适配者类，而且目标抽象类只能为接口，不能为类，其具有一定的局限性。不能将一个适配者类和它的子类都适配到目标接口。

**对象适配器模式还有如下优点**：

同一个适配器类可以把适配者类和它的子类都适配到目标接口。

**缺点**：

对象适配器模式很难置换适配者类的方法。

#### ⑶适用情形：

**①**系统需要使用现有的类，而这些类的接口不符合系统的需要。

**②**想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的类。

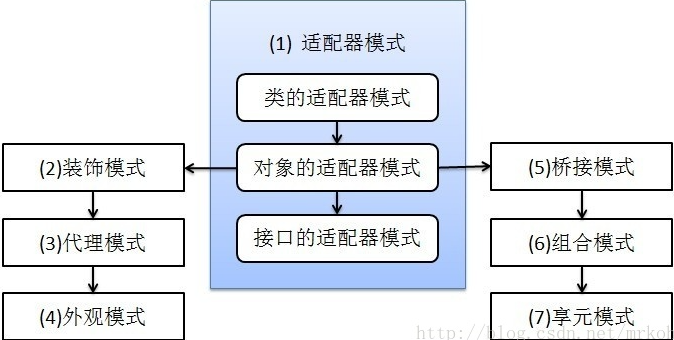
**③**通过接口转换，将一个类加入到另一个体系的类中。

**应用：**

**①Spring AOP框架中，BeforeAdvice、AfterAdvice、ThrowsAdvice**三种通知类型是借助适配器模式实现的。

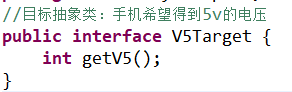
**②**通过 JDBC 连接数据库时，加载数据库驱动利用了适配器模式。不同数据库的驱动都是介于 JDBC 接口和数据库引擎接口（如Oracle）之间的适配器软件。

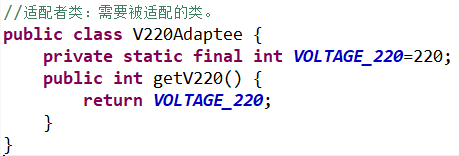
对象适配器模式是另外6种结构型设计模式的起源。

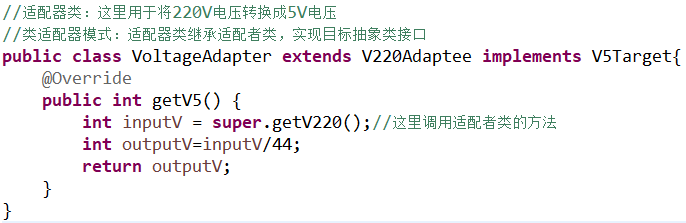


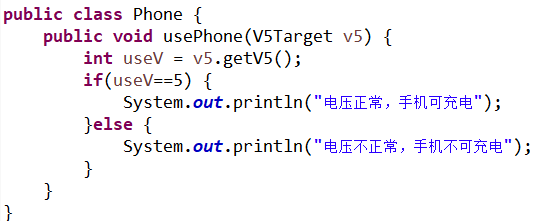
#### ⑷类适配器模式的例子：

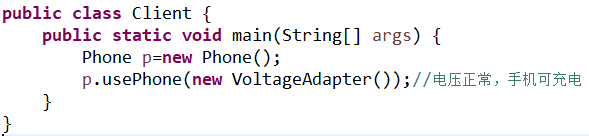
**例1：**





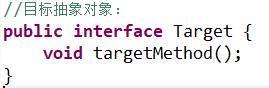


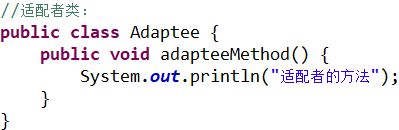


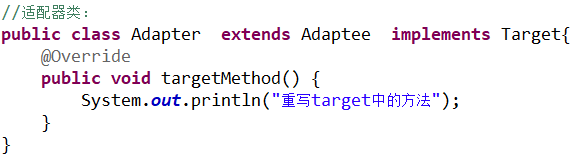


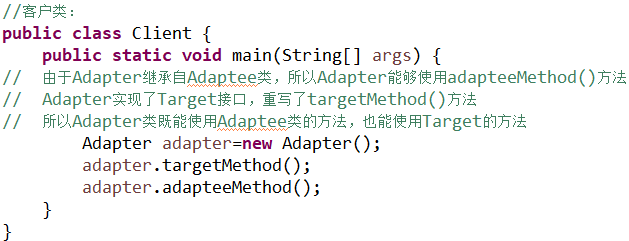
**例2：**

例子：目标抽象类Target有一个targetMethod()方法，适配者类有一个adapteeMethod()方法，现在客户希望使用Adaptee这个类，也希望能够使用targetMethod()这个方法，因此可以采用一个相当于中间者的类即适配器类Adapter将Adaptee类和Target接口衔接起来。





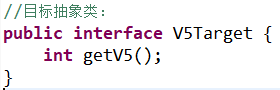


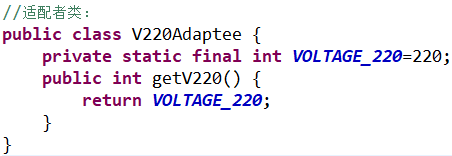


类适配器模式中，由于java中类没有多继承，所以上面的target只能为接口，不能为类。

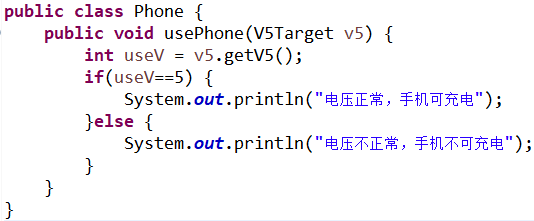
#### ⑸对象适配器模式的例子：

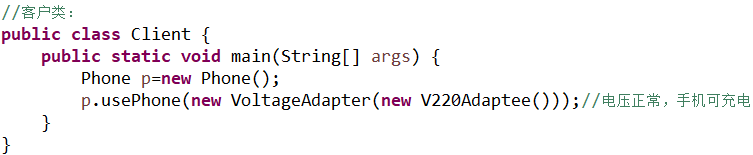
**例1：**



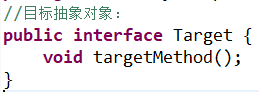


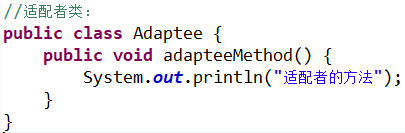


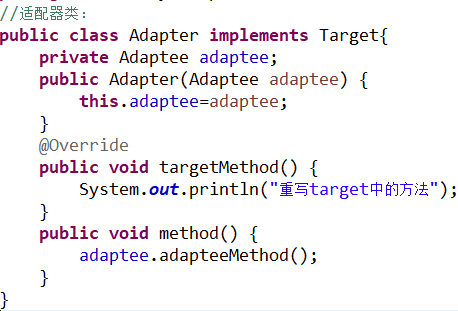


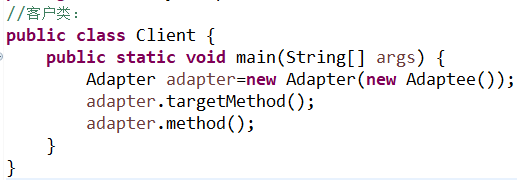


**例2：**



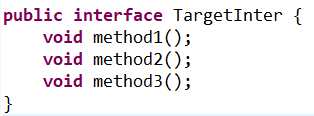


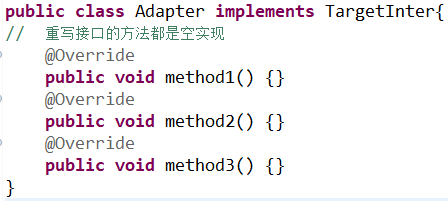


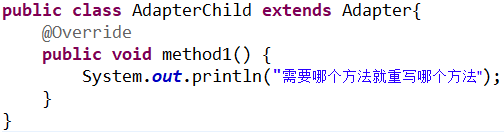


从上面类适配器模式的例子和对象适配器模式的例子中可以知道，这两者的区别主要就是在适配器(Adatper)类中有区别。类适配器的Adapter继承Adaptee类同时还实现Target接口。而对象适配器的Adapter则是实现了Target接口，同时将Adaptee类作为自己属性进行封装。**对于适配者来说，类适配器使用的是继承的方式，直接继承了Adaptee，所以Adaptee的子类不能进行适配；而对象适配器使用的则是组合的方式，Adaptee及其子类都可以进行适配**。

#### ⑹接口适配器模式的例子







### 2.代理模式

代理模式(Proxy Pattern)的**定义**：给某一个对象提供一个代理，并由代理对象控制对原对象的访问。

代理模式和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。

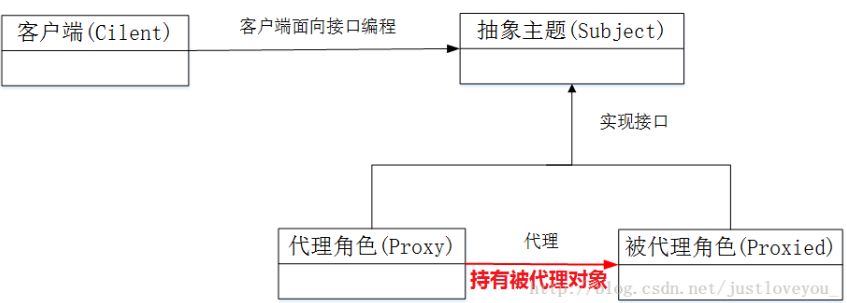
#### ⑴组成部分：

**①抽象主题角色**：**Subject**，声明了真实主题角色类和代理主题角色类的共同接口。可以是接口也可以是类。

**②真实主题角色**：**RealSubject**，定义了真实的业务操作方法。也称被代理类角色

**③代理主题角色**：**Proxy**，该角色包含了对真实主题角色对象的引用，负责对真实主题角色的调用，并在真实主题角色处理前后做预处理和后处理。客户端就是通过代理主题角色间接调用真实主题角色中的方法。

**结构示意图：**



#### ⑵代理模式的优缺点：

**优点**：

能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。

代理类同时能在不修改原有代码的基础上增加一些新功能，**是开闭原则的典型实践。**

**缺点**：

由于在客户和真实主题角色之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。

#### ⑶概念、分类、目的、作用

代理模式根据**使用目的**可分：远程代理、虚拟代理、Copy-on-Writer代理、缓冲(Cache)代理、防火墙(Firewall)代理等。

根据**代理类的创建时期**可分为：**静态代理和动态代理**。其中**动态代理又分为jdk代理和cglib代理**。

**在程序运行前就已经存在的编译好的代理类就是静态代理，**比如说在程序运行前，由程序员创建或特定工具自动生成代理类，并对其编译生成.class文件**；**

**在程序运行期间根据需要动态创建代理类及其实例来完成具体的功能称为动态代理**。

简单说**静态代理中，代理类在程序运行前就已创建，而动态代理中，代理类是在程序运行时根据需要创建**。

在静态代理中，真实主题角色事先已经存在并作为代理主题角色类的内部成员属性。而动态代理则是在事先不知道真实主题角色的情况下使用代理主题角色。

注意：**不管是静态代理还是动态代理，它们都是实现了共同的接口或继承了共同的类。在静态代理中只是自己将代理类实现了抽象业务接口，而动态中代理类则是系统会帮我们实现**。(这是自己根据jdk动态代理总结的)。

情况1：若有N个真实业务类，但代理类中的“预处理、后处理”都是相同的，若采用静态代理模式则要手动创建N代理类，显然不太好。若采用动态代理则可以简单地为各个真实业务类分别生成代理类，共享“预处理，后处理”功能，这样可以大大减小程序规模 ，这是动态代理的一大亮点。

jdk动态代理与静态代理相比，最大的好处是接口中声明的所有方法都被转移到调用处理器invoke()方法中处理。这样，在接口方法数量比较多的时候，可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。

**代理模式的目的**：为真实业务对象提供一个代理对象以控制对真实业务对象的访问。

**代理对象的作用**：

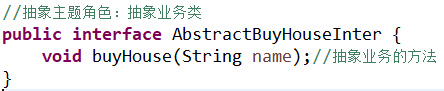
**①**代理对象存在的价值主要用于拦截对真实业务对象的直接访问；

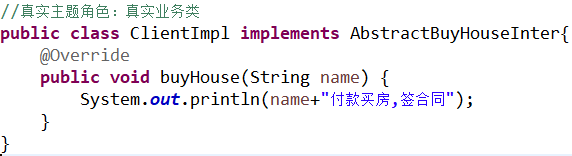
**②**代理对象是对真实业务对象的增强，可以方便对消息进行预处理和后处理。代理类同时能在不修改原有代码的基础上增加一些新功能，符合开闭原则。

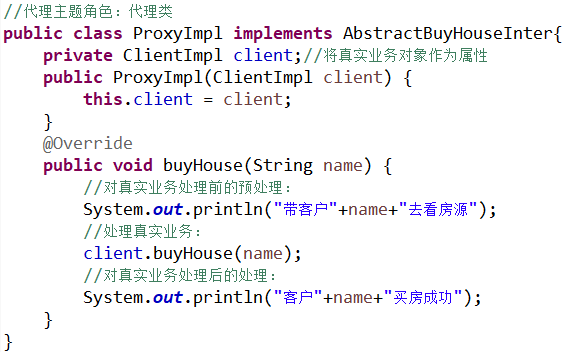
代理对象和真实业务对象实现共同的接口或者继承于同一个类。

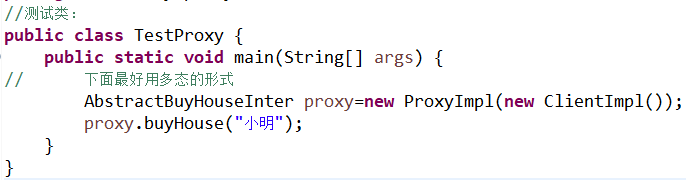
#### ⑷静态代理实例

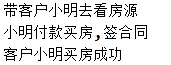
**定义静态代理的步骤**：定义抽象业务接口，定义真实业务类并实现抽象业务接口，定义代理类并实现抽象业务接口，最后调用。











#### ⑸jdk动态代理

**特别注意**：**jdk动态代理生成的代理类继承了Proxy类**，因为java中类不能多继承，**所以 jdk动态代理只能实现接口代理而不能实现类代理**。

jdk动态代理的实现类位于java.lang.reflect 包中，主要涉及两个类。jdk动态代理就是用反射的技术实现的。

**①InvocationHandler接口**：它是代理实例的调用处理程序实现的接口。

该接口只有一个抽象invoke()方法，执行被代理对象的任何接口方法都会经过该方法。如下：

**public** Object **invoke**(Object proxy, Method method, Object[] args)**throws** Throwable;

该方法的作用是实现对真实业务类的代理的访问，第一个参数proxy表示代理对象的引用，第二个参数method表示需要代理的方法，第三个参数表示代理方法的参数数组，返回值为被代理方法的返回值。

这是采用反射的方式完成对真实业务类的业务方法的调用。

**②Proxy类**：**用于动态生成代理类**。该类最常用的方法如下：

**public** **static** Object **newProxyInstance**(ClassLoader loader，Class<?>[] interfaces, InvocationHandler h)

该方法使根据传入的接口类型interfaces返回一个动态创建的代理类的实例。第一个参数类加载器是用来加载代理对象字节码的，它和被代理类使用相同的类加载器，第二个参数是Class的数组，它让代理对象和被代理对象有相同的方法，即代理类和被代理类实现相同的接口。第三个参数是用来给被代理类的方法提供增强的功能。

**public** **static** **boolean** **isProxyClass**(Class<?> cl)：该方法用于判断指定类是否是一个动态代理类。

**public** **static** Class<?> **getProxyClass**(ClassLoader loader，Class<?>... interfaces)：该方法用于获取指定类加载器和接口所对应的动态代理类的Class对象。

**public** **static** InvocationHandler getInvocationHandler(Object proxy)：该方法用于获取指定代理类对象所关联的InvocationHandler对象。

注意jdk动态代理方式生成的代理类继承了Proxy类，同时实现了抽象业务接口，同时**重写了抽象业务接口中的业务方法，还重写了Object类的toString()、equals()、hashCode()方法**。

如何通过反编译的方式查看到动态生成的代理类的源码？

**扩展知识：**

java.lang.**ClassLoader**，这是一个**类加载器类**，这个类的**作用**是负责将类的字节码装载到Java虚拟机中并为其定义类对象，然后该类才能被使用。这个类与普通类的唯一区别就是**其字节码是由JVM在运行时动态生成的而非预存在于任何一个.class 文件中**。

采用Class类**的getClassLoader()**方法可以获得这个类的类加载器。

采用Class类的**getInterfaces()**方法可以获得这个类所实现的所有接口。

**jdk动态代理的一般步骤**：

**①**定义抽象业务接口

**②**定义真实业务类，并实现抽象业务接口

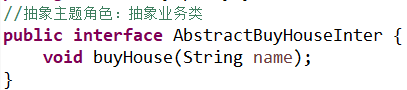
**③**定义一个Handler类，实现InvocationHandler接口，重写对应的invoke()方法。

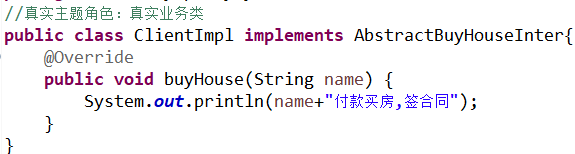
**④**根据Proxy的newProxyInstance()方法创建代理类。

第③④步也可以结合起来。

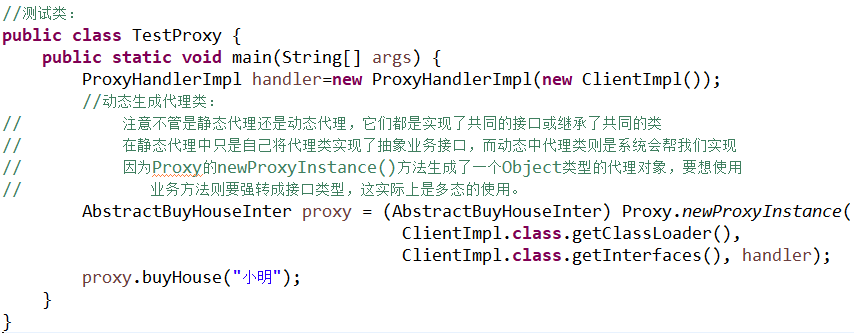
**⑤**最后使用代理。

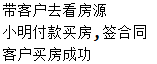
**例1：**



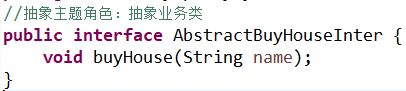


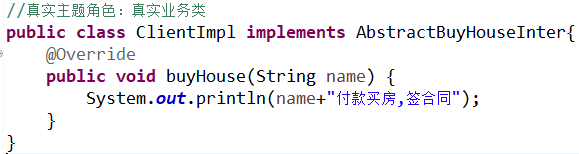




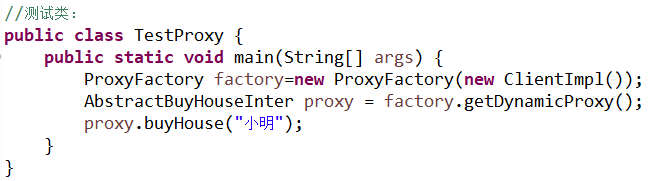


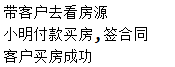
**例2：**









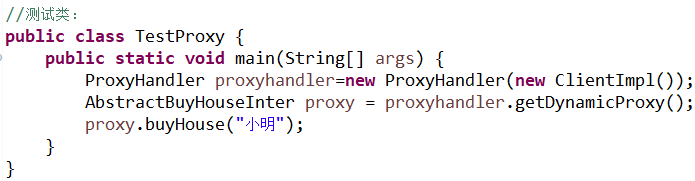


上面例2是例1的简化版，采用例2的方式会使代码更简洁，但是代码就没有例1更好理解。

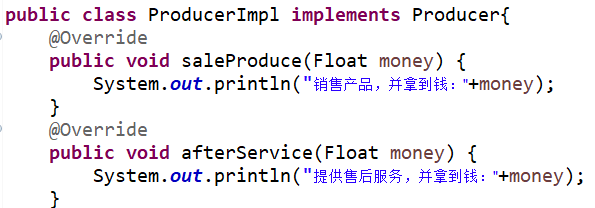
**例3(例2代码的优化版)：**

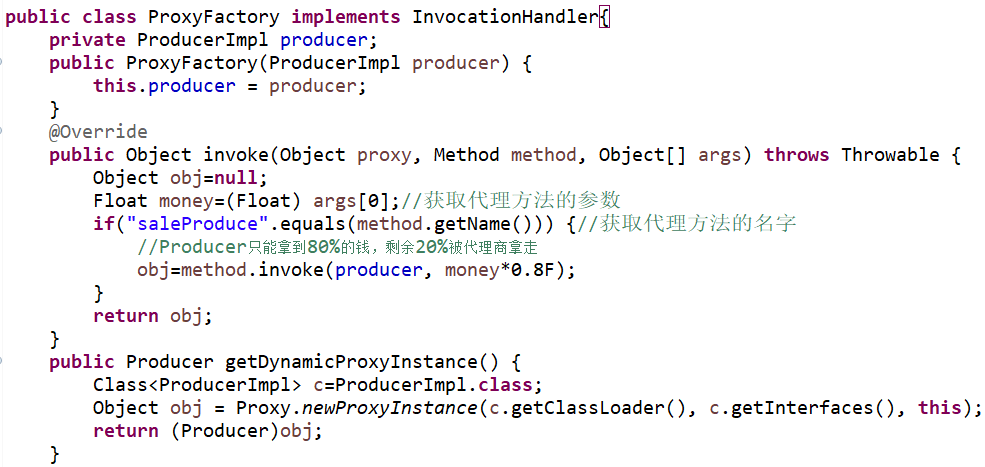
将例2中的代理工厂类的代码修改成下面这种方式，则更直观。真实业务类和抽象业务类的代码同例2。





**例子4**：





#### ⑹cglib动态代理

jdk的动态代理机制只能代理实现了接口的类，而没有实现接口的类就不能采用jdk动态代理；**cglib是针对类来实现代理的，原理是对指定的目标业务类(就是被代理类)生成一个子类，并覆盖其中方法实现增强**，**但因为采用的是继承，所以不能对final修饰的类进行代理，同时对于final修饰的方法也不能进行代理处理。**

CGLIB(Code Generation Library)实现动态代理，并不要求被代理类必须实现接口。CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

简单的说，**cglib动态代理就是通过继承被代理类的方式来进行代理的**。

cglib动态代理使用Enhancer类的create()方法创建，其该方法有如下几种：

create()：没有参数的。

create(Class[] args1,Object[] args2)：第一个参数是被代理对象的class文件。

**使用cglib代理模式时，要先将cglig的包导入。**

**cglib动态代理的一般步骤：**

**①**定义真实业务类(被代理类)。这个可以不用实现抽象业务接口或继承抽象业务类。

**②**定义一个拦截器，并实现MethodInterceptor接口。重写对应的intercept()方法。在这个方法中对业务进行预处理、直接处理业务和后处理。

**③**创建一个Enhancer对象，并调用setSuperClass(被代理类.class)方法；调用setCallback()方法，这个方法的参数是实现了MethodInterceptor接口的对象；调用create()方法创建代理类对象。

**Enhancer e=new Enhancer();**

**e.setSuperClass(被代理类.class);**

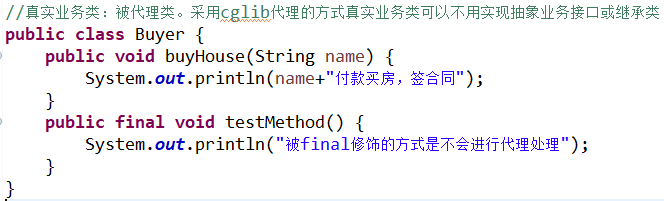
**e.setCallback(实现了MethodInterceptor接口的对象);**

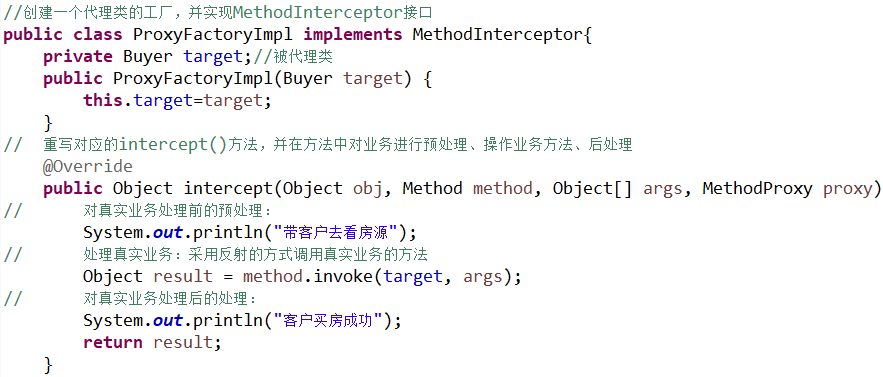
**e.create()：创建一个代理类对象**

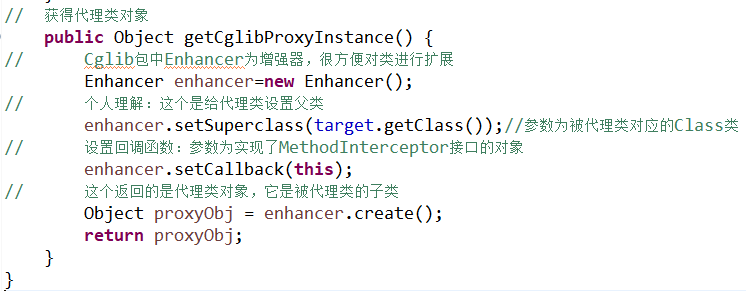
**步骤②③可以合并为一起**

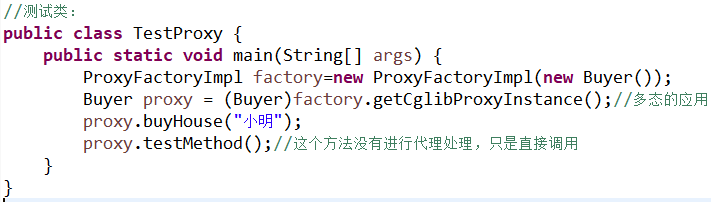
**④**进行测试

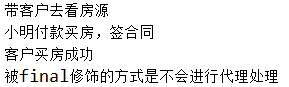
**例子：**











分析结果：若被final修饰的代理方法进行了代理处理，则“被final修饰的方式是不会进行代理处理”被输出的前后应该也要有“带客户去看房源”、“客户买房成功”的输出，这里没有，说明这个方法没有被进行代理处理。所以**被代理类中被final修饰的方法，在cglib动态代理中是不会进行代理处理的**。

#### ⑺总结jdk代理和cglib代理

**①**是否需要实现抽象业务接口：jdk动态代理的真实业务类必须实现抽象业务接口，且抽象业务只能为接口不能为类。而cglib动态代理的真实业务类则可以没有抽象业务接口。

**②**jdk的动态代理机制只能代理实现了接口的类，而没有实现接口的类就不能采用jdk动态代理，对于没有实现接口的类则采用cglib动态代理。即jdk动态代理相当于是针对接口来实现代理的，而cglib是针对类来实现代理的。

**③**jdk的代理类是Proxy类的子类。而cglib的代理类是真实业务类(被代理类)的子类。

**④**jdk的代理类是采用Proxy的newProxyInstance()方法创建的，而cglib的代理类则是采用Enhancer的create()方法创建的。

**⑤**jdk动态代理需要使用InvocationHandler接口，并使用它的invoke()方法，在该方法中对业务进行预处理、直接处理业务和后处理。cglib动态代理则则需要使用MethodInterceptor接口，并使用它的intercept()方法，在该方法中对业务进行预处理、直接处理业务和后处理。

#### ⑻应用：

在Spring AOP中应用了动态代理技术。

在项目前期开发中，没有加入缓存、日志等功能，若后期想加入，则可以采用代理的方式。

**扩展知识：**

**AOP(Aspect Oriented Programming)：面向切面编程**，就是将日志记录、事务处理、异常处理等代码从业务逻辑代码中划分出来，然后将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码。

### 3.组合模式

老师讲过，自己还没学，以后有空再补充。

### 4.外观模式

老师没讲，但是这个模式的使用频率好像还很高，有空自己补充学习。



## 四、行为型模式

### 1.观察者模式

老师讲过，自己还没学，以后有空再补充。

### 2.策略模式

老师讲过，自己还没学，以后有空再补充。

### 3.模板模式

老师讲过，自己还没学，以后有空再补充。

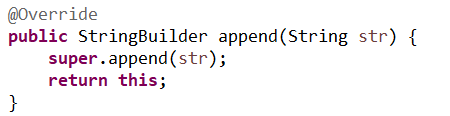
## 弄清楚下面的内容：

oop面向对象的基本特征

高内聚、低耦合什么意思？

lambda表达式

类加载器：获得Class对象，然后调用getClassLoader()



this和super一起使用的例子，上面方法是StringBuilder的。