# 代理模式(Proxy)

1. 代理模式：通过代理对象访问目标对象，可以在目标对象实现的基础上，增强额外的功能操作即拓展目标对象的功能。
2. 被代理的对象可以是远程对象、创建开销大的对象或需要安全控制的对象
3. 代理模式有不同的形式，主要有三种：静态代理、动态代理（JDK代理、接口代理）和Cglib代理（可以在内存动态的创建对象，而不需要实现接口，他属于动态代理的范畴）

## 静态代理

1. 静态代理在使用时，需要定义接口或父类，被代理对象（目标对象）与代理对象一起实现相同的接口或者是继承相同父类。
2. 定义接口
3. 创建目标对象实现接口
4. 使用静态代理方式，代理对象也要实现接口
5. 调用的时候通过调用代理对象的方法来实现调用目标对象
6. Tip：代理对象与目标对象要实现相同的接口,然后通过调用相同的方法来调用目标对象的方法;

### 实例：

1.定义接口

**public** **interface** HouseProvider {

**void** rentHouse();

}

1. 创建目标对象

**public** **class** HouseProviderImpl **implements** HouseProvider {

@Override

**public** **void** rentHouse() {

// **TODO** Auto-generated method stub

System.***out***.println("房屋出租");

}

}

1. 创建代理对象

**public** **class** Houseproxy **implements** HouseProvider {

**private** HouseProvider target;

**public** Houseproxy(HouseProvider target) {

**this**.target = target;

}

@Override

**public** **void** rentHouse() {

// **TODO** Auto-generated method stub

System.***out***.println("中介租房");

target.rentHouse();

}

}

1. 创建客户端

**public** **class** Client {

**public** **static** **void** main(String[] args) {

HouseProvider house = **new** HouseProviderImpl();

Houseproxy houseproxy = **new** Houseproxy(house);

houseproxy.rentHouse();

}

}

### 优缺点：

1. 在不修改目标对象的方法的前提下，能通过代理对象对目标对象进行拓展；
2. 因为目标对象和代理对象需要实现相同的接口，一旦接口增加方法，目标对象和代理对象都需要维护;

## 动态代理

1. 动态代理不需要实现接口，但是目标对象需要实现接口,否则不能用动态代理;
2. 代理对象的生成，是利用JDK的API，动态的在内存中构建代理对象
3. 动态代理也叫做JDK代理、接口代理

JDK生成代理对象的API

1. 代理对象所在的包：java.lang.reflect.Proxy
2. JDK实现代理只需要newProxyInstance方法,该方法需要传递三个参数，

**public** **static** Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

### 实例

1. 创建接口

**public** **interface** HouseProvider {

**void** buyHouse();

}

1. 创建目标对象

**public** **class** HouseProviderImpl **implements** HouseProvider {

@Override

**public** **void** buyHouse() {

// **TODO** Auto-generated method stub

System.***out***.println("房屋出售");

}

}

1. 构建ProxyFactory

**public** **class** ProxyFactory {

//目标对象

**private** Object target;

**public** ProxyFactory(Object target) {

**this**.target = target;

}

/\*\*

\* public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

\* loader:指定目标对象使用的类加载器，获取加载器的方式固定;

\* interfaces:目标对象实现的接口类型,使用泛型方法确认类型

\* h:事情处理:执行目标对象的方法时，会触发事情处理器方法

\*/

**public** Object getProxyInstance() {

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(), **new** InvocationHandler() {

@Override

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

// **TODO** Auto-generated method stub

Object result = method.invoke(target, args);

**return** result;

}

});

}

}

1. 创建client

**public** **class** Client {

**public** **static** **void** main(String[] args) {

HouseProviderImpl houseProviderImpl = **new** HouseProviderImpl();

HouseProvider proxyInstance = (HouseProvider)**new** ProxyFactory(houseProviderImpl).getProxyInstance();

proxyInstance.buyHouse();

}

}

## Cglib代理

1. 静态代理和JDK代理模式都要求目标对象实现一个接口，但是有时候目标对象只是一个单独的对象，并没有实现任何接口，这个时候可以使用可以使用目标对象子类实现代理，可以使用Cglib代理;
2. Cglib代理也叫子类代理，他是在内存中构建的一个子类对象从而实现对目标对象功能的拓展，CgLib代理也归属与动态代理;
3. Cglib是一个强大的高性能的代码生成包，它可以产生在运行期拓展java类与实现java接口。它广泛的被许多AOP的框架使用，例如spring AOP，事项方法拦截
4. 在Aop编程中如何选择代理模式：
5. 目标对象需要实现接口，用JDK代理
6. 目标对象不需要实现接口，使用Cglib代理
7. Cglib包的底层是通过使用字节码处理框架ASM来转换字节码并生成新的类;

实现：

1. 需要引入Cglib的jar文件
2. 在内存中动态构建子类，注意代理的类不能为final，否则报错 java.lang.IIIegalArgumentException
3. 目标对象的方法如果是final/static,那么就不会被拦截，即不会执行目标对象额外的业务方法

### 实例：

1. 创建目标对象

**public** **class** HouseProvider {

**public** **void** show() {

System.***out***.println("你好，房屋出售");

}

}

1. 构建ProxyFactory

**public** **class** ProxyFactory **implements** MethodInterceptor {

//目标对象

**private** Object target;

**public** ProxyFactory(Object target) {

**super**();

**this**.target = target;

}

**public** Object getProxyInstance() {

//1.创建一个工具类

Enhancer enhancer = **new** Enhancer();

//2.设置父类

enhancer.setSuperclass(target.getClass());

//3.设置回调函数

enhancer.setCallback(**this**);

//4.创建子类对象,即代理对象

**return** enhancer.create();

}

@Override

**public** Object intercept(Object arg0, Method method, Object[] arg2, MethodProxy arg3) **throws** Throwable {

// **TODO** Auto-generated method stub

System.***out***.println("Cglib代理");

Object result = method.invoke(target, arg2);

**return** result;

}

}

1. 创建client

**public** **class** Client {

**public** **static** **void** main(String[] args) {

HouseProvider provider = **new** HouseProvider();

HouseProvider proxyInstance = (HouseProvider)**new** ProxyFactory(provider).getProxyInstance();

proxyInstance.show();

}

}

## 代理模式的变体

1. 防火墙代理

内网通过代理穿透防火墙，实现对内网的访问.

1. 缓存代理

如当请求图片文件等资源时，先到缓存代理取，如果取不到资源，再到公网或者数据库去，然后缓存；

1. 远程代理

远程对象的本地代表，通过它可以把远程对象本地对象来调用。远程代理通过网络和真正的远程对象沟通信息；

1. 同步代理，主要使用在多线程编程中，完成多线程间同步工作；

# 2.单例设计模式

单例模式是采用一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得实例的方法(静态方法)

### 饿汉式（两种方式）

#### 实现步骤：

1. 私有化构造函数 （防止new）
2. 类的内部创建对象
3. 暴露一个静态的公共方法

#### 优缺点：

1. 优点：写法简单，在类装载的时候完成实例化，避免了线程同步问题
2. 缺点：在类装载的时候就完成实例化，没有达到懒加载的效果。如果从开始至终从未使用过这个实例，则会造成内存的浪费。

3.总结：这种单例模式可以用,可能造成内存浪费；

#### 实现：

#### 静态常量方式

**public** **class** Singleton1 {

//1.构造私有化

**private** Singleton1() {

}

//2.在本类中创建类的实例对象

**private** **static** Singleton1 *singleton* = **new** Singleton1();

//3.暴露一个静态的公共方法，返回实例对象

**public** **static** Singleton1 newInstance() {

**return** *singleton*;

}

}

#### 静态代码块方式

**public** **class** Singleton2 {

**private** **static** Singleton2 *singleton*;

//1.构造私有化

**private** Singleton2() {

}

//2.使用静态代码块创建对象实例

**static** {

*singleton* = **new** Singleton2();

}

//3.暴露静态的公共方法，返回对象实例

**Public static** Singleton2 newInstance() {

**return** *singleton*;

}

}

### 懒汉式

#### 懒汉式（线程不安全）

**public** **class** Singleton3 {

**private** **static** Singleton3 *instance*;

//1.私有化构造函数

**private** Singleton3() {

}

//2.提供公共的静态方法，返回实例对象

**public** **static** Singleton3 getInstance() {

**if** (**null** == *instance*) {

*instance* = **new** Singleton3();

}

**return** *instance*;

}

}

优缺点：

1. 起到了懒加载的效果，但是只能在单线程下使用；
2. 如果在多线程下，一个线程进入**if** (**null** == *instance*)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这是便会产生多个实例。所以在多线程环境不可以使用这种方式。
3. 结论：在实际开发中，不要使用这种方式；

#### 懒汉式（线程安全，同步方法）

**public** **class** Singleton3 {

**private** **static** Singleton3 *instance*;

//1.私有化构造函数

**private** Singleton3() {

}

//2.加入同步代码块，解决线程不安全问题

**public** **static** **synchronized** Singleton3 getInstance() {

**if** (**null** == *instance*) {

*instance* = **new** Singleton3();

}

**return** *instance*;

}

}

优缺点：

1. 解决了线程不安全问题；
2. 效率低，每个线程在想获得类的实例的的时候，执行getInstance()方法都要进行同步，而其实这个方法只执行一次实例化代码就够了，后面的想获取改类的实例，之间return就行了。方法进行同步效率太低。
3. 结论：在实际开发中，不推荐使用这种方式

#### 懒汉式(线程安全，同步代码块)

**public** **class** Singleton4 {

**private** **static** Singleton4 *instance*;

//1.私有化构造函数

**private** Singleton4() {

}

//2.加入同步代码块，解决线程不安全问题

**public** **static** **synchronized** Singleton4 getInstance() {

**if** (**null** == *instance*) {

**synchronized** (Singleton4.**class**) {

*instance* = **new** Singleton4();

}

}

**return** *instance*;

}

}

优缺点：

1. 这种方式，本意是想对第前面方式的改进，因为前面同步方法效率太低，改为同步产生实例化的代码块；
2. **但是这种同步并不能起到线程同步的作用；**如果一个线程进入了**if** (**null** == *instance*) 判断语句块，还未来得及往下执行，另一个线程也会通过这个判断语句,这时便会产生多个实例；
3. **结论：在时间开发中，不能使用这种方式**

## 双重检验方式

**public** **class** Singleton4 {

**private** **static** Singleton4 *instance*;

//1.私有化构造函数

**private** Singleton4() {

}

//2.加入同步代码块，解决线程不安全问题

**public** **static** **synchronized** Singleton4 getInstance() {

**if** (**null** == *instance*) {

**synchronized** (Singleton4.**class**) {

**if** (**null** == *instance*) {

*instance* = **new** Singleton4();

}

}

}

**return** *instance*;

}

优缺点：

1. Double-Check概念是多线程开发中常使用到的，如代码中所示，我们进行了两次**if** (**null** == *instance*)检查，这样可以保证线程安全了。
2. 这样，实例化代码只用执行一次，后面再次访问，判断**if** (**null** == *instance*)，返回实例对象，也避免反复进行方法同步；
3. 线程安全；延迟加载；效率更高;
4. **结论：在实际开发中推荐使用此种单例模式**

## 静态内部内方式

/\*\*

\* 知识储备：需要对静态内部类有了解

\* 1.当Singleton类被加载时，内部类SingeltonInstance不会被加载

2.当调用Singeleton中getInstance时使用内部类中的静态变量时会导致内部类装载,只会装载一次;

\*/

**public** **class** Singleton5 {

//1.私有化构造函数

**private** Singleton5() {

}

//2.在静态内部类中实例会对象

**private** **static** **class** SingletonInstance{

**private** **static** **final** Singleton5 ***INSTANCE*** = **new** Singleton5();

}

//3.返回对象

**public** **static** Singleton5 getInstance() {

**return** SingletonInstance.***INSTANCE***;

}

}

优缺点：

1. 这种方式采用了类装载的机制来保证初始化实例时只有一个线程；
2. 静态内部类的方式在Singleton5 类被加载时并不会立即实例化,而是在需要实例化时，调用getInstance方法，才会被装载SingletonInstance类，从而完成Singleton5 的实例化；
3. 类的静态属性只会在第一次加载类的时候初始化，所有在这里，JVM帮助我们保证了线程的安全性，在类进行初始化时，别的线程无法进入的；
4. 优点：避免了线程不安全，利用静态内部类特点实现延迟加载，效率高；
5. 结论：推荐使用

## 使用枚举方式

**public** **enum** Singleton {

***INSTANCE***;

**public** **void** method() {

//方法实现

System.***out***.println("hello");

}

**public** **static** **void** main(String[] args) {

Singleton instance = Singleton.***INSTANCE***;

instance.method();

}

}

优缺点：

1. 这借助JKD1.5中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而且还能防止序列化重新创建新的对象;
2. **结论：推荐使用**

## 单例模式的使用说明：

1. 单例模式包装类系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统的性能。
2. 当想实例化一个单例类的时候,必须要记住使用相应的获取对象的方法，而不是使用new。
3. 单例模式使用的场景：需要频繁的进行创建和销毁的对象、创建对象时耗时过多或耗费资源过多（即：重量级对象）,但又经常用到的对象、工具类对象、频繁访问数据库或文件的对象（比如数据源、session工厂等）

# 3.工厂模式

## 简单工厂模式

1. 简单工厂模式是属于创建型模式，是工厂模式的一种。**简单工厂模式是由一个工厂对象决定创建出哪一种产品的实例**。简单工厂模式是工厂模式家族中的最简单的实用模式。

2. 简单工厂模式：定义了一个创建对象的类，由这个类来封装实例化对象的行为

3. 在软件开发中，当我们会用到大量的创建某种、某类或者某批对象时，就会使用到工厂模式

### 实现:

**通过继承或接口实现多态;**

**public** **interface** ComputerProvider {

}

**public** **class** ComputerFactory {

**public** **static** ComputerProvider getComputer(String type) {

**if**("Dell".equals(type)) {

**return** **new** SimpleDellProviderImpl();

}**else** **if**("Lenovo".equals(type)){

**return** **new** SimpleLenovoproviderImpl();

}

**return** **null**;

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

SimpleDellProviderImpl computer =(SimpleDellProviderImpl) ComputerFactory.*getComputer*("Dell");

System.***out***.println(computer);

}

}

### 优缺点：

1. 容易理解，操作简单
2. 违反了设计模式的ocp原则,**即对扩展开放,对修改关闭**。即当我们给类新增功能的时候，尽量不修改代码或者尽可能少修改代码；
3. 如果我们需要增加一个电脑的种类需要ComputerFactory 修改代码;

## 工厂方法模式

1. 定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将对象的实例化推迟到子类

**public** **abstract** **class** AbstractComputer {

**abstract** Computer create();

}

**public** **class** SimpleLenovoFactory **extends** AbstractComputer {

@Override

**public** Computer create() {

// **TODO** Auto-generated method stub

**return** **new** Computer("Lenovo");

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

SimpleDellFactory simpleDellFactory = **new** SimpleDellFactory();

Computer make = simpleDellFactory.create();

}

## 抽象工厂模式

1. 定义一个接口用于创建相关或有依赖关系的对象，而无需指明具体的类；
2. 抽象工厂模式可以将简单工厂模式和工厂方法模式进行整合
3. 从设计层面来看，抽象工厂模式就是对简单工厂模式的改进（或者称为进一步抽象）
4. 将工厂抽象成两层，抽象工厂和具有实现的工厂子类。程序可以根据创建对象的类型使用对应的工厂子类。更便于代码的维护和拓展。

**public** **interface** FactoryProvider {

Computer create();

}

**public** **class** DellDeskTopFactoryImpl **implements** FactoryProvider {

@Override

**public** Computer create() {

// **TODO** Auto-generated method stub

**return** **new** Computer("Dell 台式机");

}

}

**public** **class** DellLaptopFactoryImpl **implements** FactoryProvider {

@Override

**public** Computer create() {

// **TODO** Auto-generated method stub

**return** **new** Computer("Dell笔记本");

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

DellDeskTopFactoryImpl desktop = **new** DellDeskTopFactoryImpl();

Computer create = desktop.create();

System.***out***.println(create.getBrand());

}

应用：

Calenadar；

总结：

1. 工厂模式的意义：将实例化对象的代码提取出来，放到一个类中统一管理和维护，达到和主项目的依赖关系的解耦。从而提高项目的拓展和维护性。
2. 三种工厂模式
3. 设计模式的依赖抽象原则
4. 创建对象实例时，不要直接new类，而是把这个类的new类的动作放到一个工厂的方法中，并返回。（变量不要直接持有具体类的引用）
5. 不要让类继承具体类，而是继承抽象类或者实现接口
6. 不要覆盖基类中已经实现的方法

# 4.建造者模式

## 1.传统方式

**public** **interface** HouseProvider {

**void** buildBasic();//打地基

**void** buildWalls();//砌墙

**void** roofed();//封顶

House build();

}

**public** **class** HouseProviderImpl **implements** HouseProvider{

@Override

**public** House build() {

// **TODO** Auto-generated method stub

buildBasic();

buildWalls();

roofed();

**return** **new** House("商品房");

}

@Override

**public** **void** buildBasic() {

// **TODO** Auto-generated method stub

System.***out***.println("打地基完成");

}

@Override

**public** **void** buildWalls() {

// **TODO** Auto-generated method stub

System.***out***.println("砌墙完成");

}

@Override

**public** **void** roofed() {

// **TODO** Auto-generated method stub

System.***out***.println("封顶完成");

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

House build = **new** HouseProviderImpl().build();

}

}

### 优缺点

1. 优点：容易理解,操作简单
2. 缺点：设计的程序结构过于简单，没有设计缓存层对象，程序的拓展和维护不好（这种设计，把产品（房子）和创建产品的流程（建房子的过程）封装在了一起，耦合性增强了）
3. 解决方案：将产品和产品构建过程解耦=》建造者模式；

## 建造者模式

1. 建造者模式（Builder Pattern）又叫生成器模式，是一种对象构建模式。他可以将复制的对象的建造过程抽取出来（抽象类）级别，使这个抽象过程的不同实现方法可以构造出不同表现(属性)的对象;
2. 建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就来构建他们，用户不需要知道内部的具体构建细节。

建造者模式的四个角色：

1. Product（产品角色）：一个具体的产品对象；
2. Builder(产品建造者)：创建一个Product对象的各个部件指定的接口|抽象类；
3. ConcreteBuilder（具体建造者）：实现接口，构建和转配各个部件
4. Director(指挥者):构建一个使用Builder接口的对象。主要用于创建一个复杂的对象；

主要有两个作用：

1. 隔离客户和对象的生成过程；
2. 复制控制产品对象的生产过程；

### 实现：

1. 产品

**public** **class** House {

**private** String type;

**public** String getType() {

**return** type;

}

**public** **void** setType(String type) {

**this**.type = type;

}

**public** House() {

}

**public** House(String type) {

**this**.type = type;

}

}

1. 产品构建者

**public** **interface** HouseBuild {

**void** buildBasic();//打地基

**void** buildWalls();//砌墙

**void** roofed();//封顶

}

1. 具体建造者

**public** **class** HouseBuilder **implements** HouseBuild{

@Override

**public** **void** buildBasic() {

// **TODO** Auto-generated method stub

System.***out***.println("打地基完成");

}

@Override

**public** **void** buildWalls() {

// **TODO** Auto-generated method stub

System.***out***.println("砌墙完成");

}

@Override

**public** **void** roofed() {

// **TODO** Auto-generated method stub

System.***out***.println("封顶完成");

}

}

1. Director

**public** **class** HouseDirector {

**private** HouseBuild houseBuild;

**public** HouseDirector(HouseBuild houseBuild) {

**this**.houseBuild = houseBuild;

}

**public** **void** setHouseBuild(HouseBuild houseBuild) {

**this**.houseBuild = houseBuild;

}

**public** House constructHouse() {

**return** **new** House();

}

}

1. 使用

**public** **class** Client {

**public** **static** **void** main(String[] args) {

House constructHouse = **new** HouseDirector(**new** HouseBuilder()).constructHouse();

}

}

### 实例：

StringBuilder

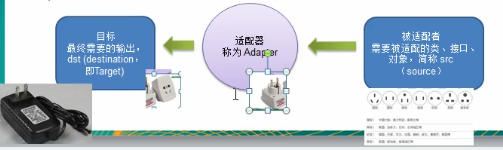
# 5.适配器模式

现实生活实例：充电器;充电器本身相当于adapter,220v交流电相当于被适配者，我们的目标是5v直流电;

1. 适配器模式（Adapter Pattern）将某个类的接口转换成客户端期望的另一个接口表示，主要目的是为了兼容性,让原本接口不匹配不能一起工作的两个类可以协同工作,器别名为包装器(Wrapper)
2. 适配器模式属于结构型模式
3. 主要分为三大类:类适配器模式、对象适配器模式、接口适配器模式

适配器模式工作原理：

1. 适配器模式：将一个类的接口转换为另一种接口，让原本接口不兼容的类可以兼容
2. 从用户的角度看不到被适配者，是解耦的；
3. 用户调用适配器转换出来的目标接口方法，适配器在调用被适配者的相关接口方法；
4. 用户收到反馈结果，感觉只是和目标接口交互；



实现：Adappter类通过继承被适配（src）类，实现目标类（dst）接口，完成，

完成src—dst的适配；

## 类适配模式

### 实现

1. 被适配者

/\*\*

\* 被适配者(需要被适配的类、接口、对象简称为src（Source）)

\* **@return**

\*/

**public** **class** Voltage220v {

**public** **int** ouput220v(){

**return** 220;

}

}

1. 目标对象的接口

/\*\*

\* 目标对象的接口dst(dstination及target);

\* **@author** wwang

\*

\*/

**public** **interface** Voltage5v {

**int** output5v();

}

1. 适配器(Adapter)

**public** **class** VoltageAdapter **extends** Voltage220v **implements** Voltage5v {

@Override

**public** **int** output5v() {

// **TODO** Auto-generated method stub

**int** ouput220v = **this**.ouput220v();

**int** v = ouput220v/44;

**return** v;

}

}

1. 提供目标访问对象

**public** **class** Phone {

**public** **void** charging(Voltage5v voltage5v) {

**if**(voltage5v.output5v()== 5) {

System.***out***.println("正常充电");

}**else** {

System.***out***.println("充电异常");

}

}

}

1. 用户调用

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Phone phone = **new** Phone();

phone.charging(**new** VoltageAdapter());

}

}

### 优缺点：

1. Java是单继承机制，所以类适配器需要继承被适配的类（src）或接口这一点算是一个缺点，因为这要求目标对象（dst）必须是接口，有一定的局限性；
2. Src类中的方法在Adapter中都会暴露出来，也增加了使用的成本；
3. 由于其继承了src类，所以它可以更加需求重写src类的方法，是的Adapter的灵活性增强了;

## 对象适配模式

基本思路和类的适配器模式相同，只是将Adapter类做修改，不是集成src类，而是持有src类的实例，以解决兼容性的问题。即持有src类，实现dst类接口，完成src-dst的适配；

### 实现：

1. 被适配类

/\*\*

\* 被适配者(需要被适配的类、接口、对象简称为src（Source）)

\* **@return**

\*/

**public** **class** Voltage220v {

**public** **int** ouput220v(){

**return** 220;

}

}

1. 目标对象接口

/\*\*

\* 目标对象的接口dst(dstination及target);

\* **@author** wwang

\*

\*/

**public** **interface** Voltage5v {

**int** output5v();

}

1. 适配器(Adapter)

**public** **class** VoltageAdapter **implements** Voltage5v {

**private** Voltage220v voltage220v;

**public** VoltageAdapter(Voltage220v voltage220v) {

**this**.voltage220v = voltage220v;

}

@Override

**public** **int** output5v() {

// **TODO** Auto-generated method stub

**int** ouput220v = voltage220v.ouput220v();

**int** v = ouput220v/44;

**return** v;

}

}

1. 提供目标访问对象

**public** **class** Phone {

**public** **void** charging(Voltage5v voltage5v) {

**if**(voltage5v.output5v()== 5) {

System.***out***.println("正常充电");

}**else** {

System.***out***.println("充电异常");

}

}

}

1. 用户调用

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Phone phone = **new** Phone();

phone.charging(**new** VoltageAdapter(**new** Voltage220v()) {

});

}

}

### 优缺点：

1. 根据“合成复用原则”，在系统中尽量使用关联关系来替代继承关系，因此大部分结构型模式都是对象结构型模式。
2. 对象适配器模式适配器模式常用的一种;

## 接口适配器模式：

1. 适配器模式(default adapter Patern)或缺省适配器模式
2. 当不需要全部实现接口提供的方法时，可以先设计一个抽象类实现接口，并为该接口中的每个方法提供一个陌生人的实现（空方法），那么改抽象类的子类可以有选择的覆盖父类的某些方法来实现需求;
3. 适用于一个接口不想使用其所有的方法的情况;

1.接口

**public** **interface** CalculateProvider {

**int** add(**int** a,**int** b);

**int** sub(**int** a,**int** b);

**int** mul(**int** a,**int** b);

**int** div(**int** a,**int** b);

}

1. 抽象类实现接口

**public** **abstract** **class** AbstractAdapter **implements** CalculateProvider {

@Override

**public** **int** add(**int** a, **int** b) {

// **TODO** Auto-generated method stub

**return** 0;

}

@Override

**public** **int** sub(**int** a, **int** b) {

// **TODO** Auto-generated method stub

**return** 0;

}

@Override

**public** **int** mul(**int** a, **int** b) {

// **TODO** Auto-generated method stub

**return** 0;

}

@Override

**public** **int** div(**int** a, **int** b) {

// **TODO** Auto-generated method stub

**return** 0;

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

AbstractAdapter abstractAdapter = **new** AbstractAdapter() {

@Override

**public** **int** mul(**int** a, **int** b) {

// **TODO** Auto-generated method stub

**return** a \* b;

}

};

abstractAdapter.mul(0, 3);

}

}

实例：

1. Springmvc框架通过适配器设计模式获取到对应的controller的源码;
2. Spring定义了一个适配器接口，使得每一种controller有一种对应的适配器实现类
3. 适配器代替controller执行相应的方法
4. 拓展controller时，只需要增加一个适配器类就完成了Springmvc的拓展了

总结：

1. 类适配器：以类给到，在adapter里，就是将src当做类，继承
2. 对象适配器：以对象给到，在adapter里，将src作为一个对象，持有
3. 接口适配器：以接口给到，在adapter里，将src作为一个接口，实现;
4. adapter模式最大的作用是将原本不兼容的接口融合在一起工作
5. 实际开发中，实现起来不拘泥于三种经典形式