# 设计模式

相关链接：

http://blog.csdn.net/jason0539/article/details/44956775

http://www.runoob.com/design-pattern/decorator-pattern.html

## 设计模式的六大原则

总原则-开闭原则（Open Close Principle）：

对扩展开放，对修改封闭！！！

1、**单一职责原则**

每个类应该实现单一的职责，否则就应该把类拆分。概括：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

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

任何基类可以出现的地方，子类一定可以出现，是面向对象设计的基本原则之一。

简单的理解为一个软件实体如果使用的是一个父类，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别。

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

针对接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

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

即使用多个隔离的接口，比使用单个接口要好。每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。

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

一个类对自己依赖的类知道的越少越好。无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

6、**合成复用原则**（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

## 分类

### 创建型模式（5种）：

简单工厂模式、抽象工厂模式、单例模式（Singleton）、建造者模式、原型模式

### 结构型模式（7种）：

适配器模式（Adapter）、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式

### 行为型模式（11种）：

策略模式（Strategy）、模板方法模式（Template）、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

# 创建型模式

## 工厂模式

分类：简单工厂模式、工厂方法模式、抽象工厂模式

工厂方法模式：

一个抽象产品类，可派生出多个具体产品类

一个抽象工厂类，可派生出多个具体的工厂类

每个具体工厂类只能创建一个具体产品类的实例。

抽象工厂模式：

多个抽象产品类，每个抽象产品类可派生出多个具体产品类

一个抽象工厂类，可派生出多个具体工厂类

每个具体工厂类可以创建多个具体产品类的实例

工厂方法模式与抽象工厂模式的区别：

工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个

工厂方法模式具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个

### 简单工厂模式

其目的是定义一个用于创建对象的接口。

组成：

-1）抽象产品角色：它一般是具体产品继承的父类或者实现的接口

-2）具体产品角色：工厂类所创建的对象就是此角色的实例。实现抽象产品

-3）工厂类角色：是该模式的核心，存在一定的逻辑，用于创建产品

缺点:

每次增加一个具体产品，都要在工厂类中增加相应的创建实例逻辑。这违背了开闭原则

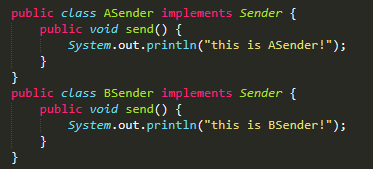
改良：设置成工厂方法模式。工厂类定义成接口，而每新值一产品时，新值对应工厂类的实现。

实例部分：

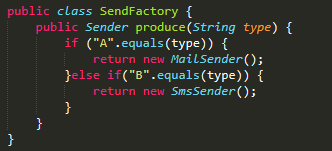
--# 1）抽象产品



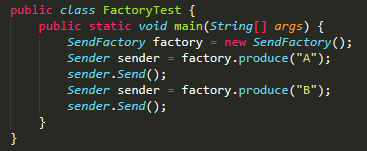
--# 2）具体产品（实现抽象产品）



--# 3）创建工厂类



--# 4）测试



### 工厂方法模式

工厂方法仅修改静态工厂方法模式的工厂类，将工厂类定义成接口，而每新增一具体产品时新增对应的具体工厂类实现。

缺点：当具体产品类非常多时，会创建大量与之对应的具体工厂类实现。

组成：

-1）抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。

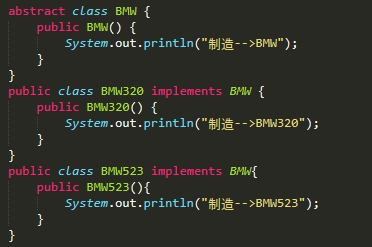
-2）具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

-3）抽象工厂角色：是该模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或继承的父类。在java中它由抽象类或者接口来实现。

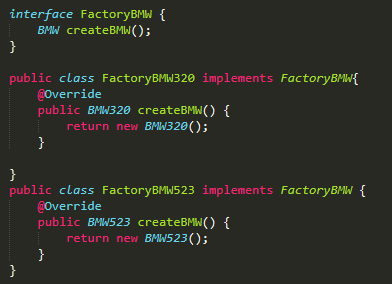
-4）具体工厂角色：它含有和具体业务逻辑有关的代码。用来创建对应具体产品的对象

实例部分：

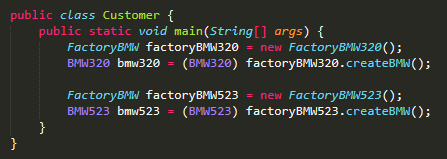
--# 1）产品类及具体产品类



--# 2）创建工厂类及具体产品工厂类



--# 3）客户类



### 抽象工厂模式

当每个抽象产品都有两个及以上的具体子类时，工厂角色怎么知道实例化哪一子类？比如每个抽象产品角色都有两个具体产品。

解决：抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

工厂方法模式有一个问题：类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则。

问题：

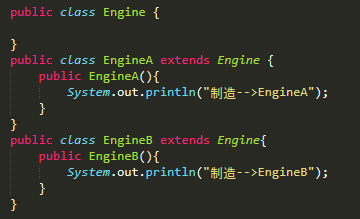
一个产品存在多个形态（如型号、颜色）

一个工厂需要将这些产品零件按照不同形态组装成不同的产品

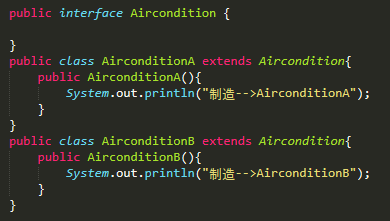
实例：

--#1）产品类：

//发动机以及型号

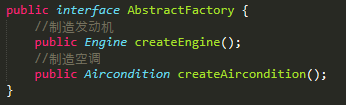


//空调以及型号

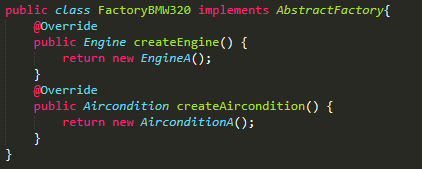


--#2）创建工厂类：

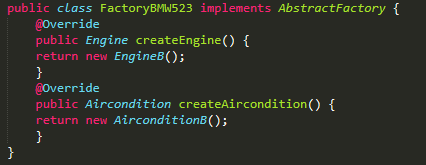
//创建工厂的接口



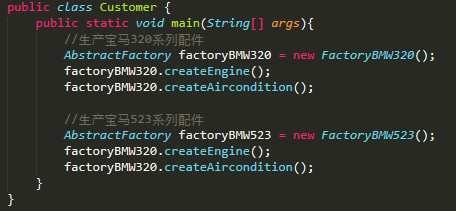
//宝马320系列



//宝马523系列



3）客户端



### 总结！！

无论是简单工厂模式，工厂方法模式，还是抽象工厂模式，他们都属于工厂模式，在形式和特点上也是极为相似的，他们的最终目的都是为了解耦。在使用时，我们不必去在意这个模式到底工厂方法模式还是抽象工厂模式，因为他们之间的演变常常是令人琢磨不透的。经常你会发现，明明使用的工厂方法模式，当新需求来临，稍加修改，加入了一个新方法后，由于类中的产品构成了不同等级结构中的产品族，它就变成抽象工厂模式了；而对于抽象工厂模式，当减少一个方法使的提供的产品不再构成产品族之后，它就演变成了工厂方法模式。

## 单例模式

特点：一个静态属性、私有构造器、静态工厂方法

在Java应用中，单例对象保证在jvm中该对象只有一个实例存在。

好处：

-1）某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

-2）省去了new操作符，降低了系统内存的使用频繁，减轻GC压力。

-3）有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了，单例模式才能保证核心交易服务器独立控制整个流程。

### 分类

可分2类：1、饿汉式单例2、懒汉式单例

饿汉式与懒汉式的区别：

区别：饿汉式单例在类初始化时立即初始化单例；懒汉式单例在第一次调用时实例化

-1）饿汉式就是类一旦加载，就把单例初始化，保证getInstance的时候，单例是已经存在的了；而懒汉式只有当调用getInstance的时候，才回去初始化这个单例。

-2）线程安全方面：饿汉式天生就是线程安全的；懒汉式本身是非线程安全的，为了实现线程安全有几种写法。

-3）资源加载和性能：饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成。懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

### 饿汉式单例

在类初始化时实例化单例。

特点：在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

public class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return instance;

}

}

### 懒汉式单例

1. 实例：

--特点：没有考虑线程安全问题，它是线程不安全的

public class Singleton {

//持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载

private static volatile Singleton single=null;

//私有构造方法，防止被实例化

private Singleton() {}

//静态工程方法，创建实例

public static Singleton getInstance() {

if (single == null) {

single = new Singleton();

}

return single;

}

}

1. 改良：考虑线程安全问题

--1>>在getInstance方法上加同步。 不推荐，在实际并发中很少会出现需要加锁的情况

public static synchronized Singleton getInstance() {

if (single == null) {

single = new Singleton();

}

return single;

}

### 双重检查锁定

--2>>双重检查锁定。判断两次等于null。

--缺点描述: <https://segmentfault.com/a/1190000015549706#articleHeader8> ！！！

public static Singleton getInstance() {

if (singleton == null) { //volatile变量禁止了指令重排序

synchronized (singleton) {

if (singleton == null)

singleton = new Singleton();

}

}

return singleton;

}

### 静态内部类实现

静态内部类：使用lazy-loading机制。Singleton类被装载时，instance并没有立即初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时才会显示装载SingletonHolder类，从而实例化instance。

实例：

public class Singleton {

private static class SingletonHolder {

private static final Singleton instance = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder. instance;

}

}

静态内部类详细说明：

类加载分为加载、链接、初始化三大步骤。其中链接又分为验证、准备和解析三个小步骤。

类中静态的内容在编译阶段被编译到类构造函数<clinit>中，在初始化步骤调用。

外部类被加载时内部类并不需要立即加载内部类，内部类不被加载则不需要进行类初始化，因此单例对象在外部类被加载了以后不占用内存。

实际上，无论是外部类还是静态内部类，对JVM而言，他们是平等的两个InstanceClass对象，只存在访问修饰符限制访问权限的问题，不存在谁包含谁的问题。

至于1、2、3这三种实现又有些区别：

第1种，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，

第2种，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗

第3种，利用了类加载机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗。

### 枚举实现

这种方式是Effective Java作者Josh Bloch提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。

实例：

public enum SingletonEnum {

INSTANCE;

private String str;

SingletonEnum() {

str = new String("123");

}

public String getInstance() {

return str;

}

}

测试：String instance = SingletonEnum.INSTANCE.getInstance();

静态内部类实现和枚举实现的单例：

上面的几种饿汉式单例，其实现原理都是利用类加载时初始化单例，即借助了ClassLoader的线程安全机制。

所谓ClassLoader的线程安全机制，就是ClassLoader的loadClass方法在加载类的时候使用了synchronized关键字。正是如此，这个方法默认在整个装载过程中都是同步的，也就是保证了线程安全。

### CAS实现（额外）

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

实例：

public class Singleton {

private static final AtomicReference<Singleton> INSTANCE = new AtomicReference<Singleton>();

private Singleton() {}

public static Singleton getInstance() {

for (;;) {

Singleton singleton = INSTANCE.get();

if (singleton != null) {

return singleton;

}

singleton = new Singleton();

if (INSTANCE.compareAndSet(null, singleton)) {

return singleton;

}

}

}

}

## 建造者模式

建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性.

角色：

-1）Buidler：为创建一个产品对象的各个部件指定抽象接口。

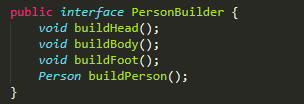
-2）ConcreteBuilder：实现Builder的接口，构造和装配该产品的各个部件，定义并明确它所创建的表示，并提供一个检索产品的接口。

-3）Director：构造一个使用Builder接口的对象，指导构造过程。

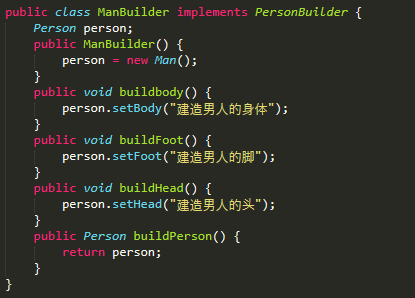
-4）Product：表示被构造的复杂对象。ConcreteBuilder创建该产品的内部表示并定义它的装配过程，包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

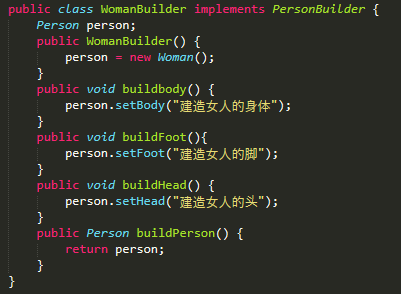
2：实例

--1）角色Builder

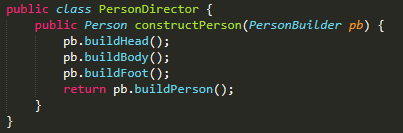


--2）角色ConcreteBuilder





--3）角色Director



--4）角色Product

public class Person {

private String head;

private String body;

private String foot;

set/get方法…

}

public class Man extends Person {

public Man(){

System.out.println(“开始建造男人");

}

}

public class Woman extends Person {

public Woman(){

System.out.println(“开始建造女人");

}

}

--5）测试

public class Test{

public static void main(String[] args) {

PersonDirector pd = new PersonDirector();

Person womanPerson = pd.constructPerson(new ManBuilder());

Person manPerson = pd.constructPerson(new WomanBuilder());

}

}

# 结构型模式

## 适配器模式

定义：将一个类的接口转换成客户希望的另一接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。

解决的问题：Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。

角色：

-1）目标接口（Target）：客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口

-2）需适配类（Adaptee）：需要适配的类或适配者类

-3）适配类（Adapter）：通过包装一个需适配类的对象，把原接口转换成目标接口

实现方式（3种）：

-1）类适配器（采用继承实现）

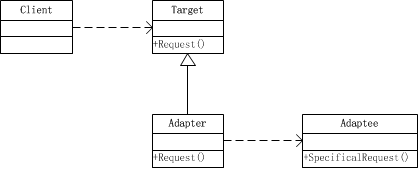
-2）对象适配器（采用对象组合方式实现）

-3）接口适配器（创建一个抽象类实现所有方法，在创建一个适配类继承该抽象类）

### 类适配器

实现原理：通过继承来实现适配器功能

描述：当我们要访问接口A没有的、但接口B中有的方法，在不改变接口A的情况下，我们可以定义一个适配类（类Adapter）来进行中转，它实现接口A、并继承接口B的实现类Adaptee，这样我们就能在适配类中直接引用接口B中的方法。



》》实例：

--1）接口A，没有method2方法

public interface A {

public void method1();

}

--2）接口B，有method2方法

public interface B {

public void method2();

}

public class Adaptee implements B {

public void method2() { ... }

}

--3）适配类，实现接口A并继承接口B的实现类

public class Adapter extends Adaptee implements A {

public void method1() { ... }

}

--4）测试

public class Main {

public static void main(String[] args){

Adapter adapt = new Adapter();

adapt.method1();

adapt.method2(); //访问接口A没有、但接口B有的方法

}

}

### 对象适配器

--适配类：实现接口A，并声明一个接口B实现类对象的引用

public class Adapter implements A {

private static Adaptee adaptee; // 声明一个接口B实现类对象的引用

public Adapter(Adaptee adaptee) { // 构造器

this.adaptee = adaptee;

}

public void method1(){ // method1()方法无法通过继承获得，需要自己声明

System.out.println("method1");

}

public void method2(){ // method2()在源类Adaptee中有，这里直接委派

adaptee.method2();

}

}

--附：其他的类与类适配器的一致

Adapter与Adaptee是委托关系，即对象适配器模式

### 接口适配器模式

原理：通过抽象类实现适配，这种适配稍于上面所述的适配

描述：当存在这样一个接口，它定义了N多个方法，而我们现在却只想使用其中的一到几个方法。为了不对所有方法都进行实现，我们可以使用一个抽象类作为中间件，即适配器。用这个抽象类实现接口，而抽象类中所有方法都置空，那么我们在创建抽象类的继承类，而且重写我们需要使用的那几个方法即可。

》》实例：

--1）定义接口A

public interface A {

void a();

void b();

void c();

void d();

void e();

void f();

}

--2）定义适配器Adapter，实现接口A

public abstract class Adapter implements A {

public void a(){ ... //初始化方法 }

public void b(){ ... //初始化方法 }

public void c(){ ... //初始化方法 }

public void d(){ ... //初始化方法 }

public void e(){ ... //初始化方法 }

public void f(){ ... //初始化方法 }

}

--3）定义实现类，继承适配器Adapter

public class Ashili extends Adapter {

public void a(){

System.out.println("实现A方法被调用");

}

public void d(){

System.out.println("实现d方法被调用");

}

}

--4）测试

public class Clienter {

public static void main(String[] args) {

A a = new Ashili();

a.a();

a.d();

}

}

### 使用场景

类适配器和对象适配器的使用场景：

-1）想要使用一个存在的类，但是它却不符合现有的接口规范，导致无法直接去访问，这时创建一个适配器就能间接去访问这个类中的方法。

-2）我们有一个类，想将其设计为可重用类(可多处访问)，我们可以创建适配器来将这个类来适配其他没有提供合适接口的类。

接口适配器使用场景：

-1）想要使用接口中的某个或些某方法，但是接口中有太多方法，而我们只实现所需要的几个方法即可。

### SpringMVC中的适配器(HandlerAdapter) ！！！

参见Spring文档的HandlerAdapter实现

## 代理模式

定义：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不合适或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介作用。

代理模式就是多一个代理类出来，替原对象进行一些操作。因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。

优点：

-1）职责清晰：真实的角色实际上仅实现实际的业务逻辑，不用关心其它非本职工作

-2）代理对象可以在客户端和目标对象之间起到中介作用

-3）高扩展性

比如：客户--经纪人、明星。客户想要明星唱歌，只是对经纪人进行沟通，经纪人再对明星进行访问

分类：静态代理、动态代理

-1）静态代理：被代理类和代理类都实现相同的接口，且代理类中存在被代理类的引用，在执行实际逻辑方法时调用被代理类的方法，而其它逻辑采用代理中的方法。

-2）动态代理：在实现阶段不用关心代理类，而在运行阶段才指定哪一个对象。

### 静态代理：实现同一接口

角色组成：

-1）抽象角色：指代理角色（如经纪人）和真是角色（如明星）对外提供的公共方法，一般为某个接口。

-2）真实角色：实现抽象角色，定义真实角色所需所要实现的业务逻辑，供代理角色调用。真正的业务逻辑在此！

-3）代理角色：实现抽象角色，声明一个真实角色的引用。通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。将统一流程控制都在代理角色中处理！

测试实例：

--这里定义一个抽象接口(Star)、代理角色实现(ProxyStar)、真实角色实现(RealStar)

--1）抽象角色接口：提供与明星合作的一系列流程

public interface Star {

void confer();//面谈

void signContract();//签合同

void bookTicket();//订票

void sing();//唱歌

void collectMoney();//收尾款

}

--2）真实角色实现类：这里的真实角色中其实只做了一个唱歌的操作，这是真实角色真正的业务逻辑部分

public class RealStar implements Star{

...... //省略

public void sing() { //真实角色的操作：真正的业务逻辑

System.out.println("张学友.sing()");

}

}

--3）代理角色实现：代理角色中代理了真实角色所需要的操作(唱歌)

public class ProxyStar implements Star{

private Star star;//真实对象引用（明星）

public ProxyStar(Star star) {

this.star = star;

}

public void sing() { //调用真实对象的方法

star.sing();

}

...... //省略

public void collectMoney() {

System.out.println("ProxyStar.collectMoney()");

}

}

--4）测试

public static void main(String[] args) {

Star real = new RealStar();

Star proxy = new ProxyStar(real);

proxy.confer(); // ProxyStar.confer()

proxy.signContract(); //ProxyStar.signContract()

proxy.bookTicket(); //ProxyStar.bookTicket()

proxy.sing();//真实对象的操作 //张学友.sing()

proxy.collectMoney(); //ProxyStar.collectMoney()

}

### 动态代理

--不需要定义代理角色的，通过一个处理器来处理代理角色的业务逻辑

测试实例

--1）抽象角色接口：提供与明星合作的一系列流程

public interface Star {

void confer();//面谈

void signContract();//签合同

void bookTicket();//订票

void sing();//唱歌

void collectMoney();//收尾款

}

--2）真实角色实现类：这里的真实角色其实只做唱歌操作，其他操作由代理完成。

public class RealStar implements Star{

public void sing() {

System.out.println("张学友.sing()");//真实角色的操作：真正的业务逻辑

}

...... //省略

}

--3）代理角色的处理器：即实现InvocationHandler接口的invoke方法

public class StarHandler implements InvocationHandler{

private Star realStar;//真实角色的对象引用

public StarHandler(Star realStar) {

this.realStar = realStar;

}

/\*\*

\* 所有的流程控制都在invoke方法中

\* proxy-代理类，method-正在调用的方法，args-方法的参数

\*/

@Override

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

Object object = null;

System.out.println("真实角色调用之前的处理.....");

if (method.getName().equals("sing")) {

object = method.invoke(realStar, args);//激活调用的方法

}

System.out.println("真实角色调用之后的处理.....");

return object;

}

}

--4）测试

public class Main {

public static void main(String[] args) {

Star realStar = new RealStar(); //真实角色

Starhandler starhandler = new Starhandler(realStar); //处理器

//创建代理实例

Star proxy = (Star) Proxy.newProxyInstance(

ClassLoader.getSystemClassLoader(),

realStar.getClass().getInterfaces(),

starhandler);

proxy.sing(); //调用代理类的唱歌方法：其实调用的是真实角色的唱歌方法

}

}

--5）输出

方法调用之前..

ReamStar sing...

真实角色调用之后的处理.....

附录：

其中对Proxy.newProxyInstance方法参数的理解：

newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)

loader --类加载器

interfaces --需代理类已实现的接口

handler --代理的处理器

### JDK和CGLib动态代理

JDKd动态代理需要通过接口创建代理实例，而对于没有实现接口的业务类，只能通过CGLib实现。

CGLib在运行时通过字节码底层继承来实现代理类，在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势织入横切逻辑。

若被代理的对象是个实现类，Spring会使用JDK动态代理（默认）来完成操作；若被代理的对象不是实现类，Spring会强制使用CGLib来实现动态代理。

在JDK1.8的时候，JDK动态代理的速度已经比CGLib动态代理的速度快很多了，但是JDK动态代理和CGLIB动态代理的适用场景还是不一样的哈！

### Spring中AOP实现 ！！！

参见Spring文档的AOP实现

## 装饰器模式

描述：该模式创建一个装饰类，用来包装原有的类，在不改变其结构前提下，提供了额外的功能。它是作为现有的类的一个包装。

设计初衷：通常可以使用继承来实现功能的拓展,如果这些需要拓展的功能的种类很繁多,那么势必生成很多子类,增加系统的复杂性,同时,使用继承实现功能拓展,我们必须可预见这些拓展功能,这些功能是编译时就确定了,是静态的。

何时使用：在不想添加很多子类的情况下扩张类。

关键代码：1.抽象装饰类：实现被装饰类的接口，并设置一个对该接口的引用 2.具体装饰类：继承并具体扩展抽象装饰类

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：多层装饰比较复杂。

角色：

-1）抽象构件角色（Component）：给出一个抽象接口，以规范准备接收附加责任对象

-2）具体构件角色（Concrete Component）：定义一个接收附加责任的类

-3）装饰角色（Decorator）：持有一个构件(Component)对象的实例，并实现一个与抽象构件接口一致的接口

-4）具体装饰角色（Concrete Decorator）：负责给构件对象添加上附加的责任

装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

应用场景：

-1）需要扩展一个类的功能

-2）动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

1）实例：

Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

--1）被装饰类

public class Source implements Sourceable {

public void method() {

System.out.println("the original method!");

}

}

--2）装饰类

public class Decorator implements Sourceable {

private Sourceable source;

public Decorator(Sourceable source){

super();

this.source = source;

}

public void method() {

System.out.println("before decorator!");

source.method();

System.out.println("after decorator!");

}

}

--3）测试

public class DecoratorTest {

public static void main(String[] args) {

Sourceable source = new Source();

Sourceable obj = new Decorator(source);

obj.method();

}

}

》》实例：

--1）创建接口及实现类

public interface Shape {

void draw();

}

public class Rectangle implements Shape {

public void draw() {

System.out.println("Shape: Rectangle");

}

}

public class Circle implements Shape {

public void draw() {

System.out.println("Shape: Circle");

}

}

--2）抽象装饰类：实现接口并设置一个对该接口的引用

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape){

this.decoratedShape = decoratedShape;

}

public void draw(){

decoratedShape.draw();

}

}

--3）具体装饰类：扩展了ShapeDecorator 类

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

@Override

public void draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

private void setRedBorder(Shape decoratedShape){

System.out.println("Border Color: Red");

}

}

public class BlackShapeDecorator extends ShapeDecorator {

public BlackShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

@Override

public void draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

private void setRedBorder(Shape decoratedShape){

System.out.println("Border Color: Black");

}

}

--4）测试

public class DecoratorPatternDemo {

public static void main(String[] args) {

Shape circle = new Circle();

circle.draw();

Shape redCircle = new RedShapeDecorator(new Circle());

redCircle.draw();

Shape redRectangle = new BlackShapeDecorator(new Rectangle());

redRectangle.draw();

}

}

## 外观模式

定义：隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。它向现有的系统添加一个接口，来隐藏系统的复杂性。

1. 实例

--1）创建一个接口

public interface Shape {

void draw();

}

--2）创建该接口的实现类

public class Rectangle implements Shape {

public void draw() {

System.out.println("Rectangle::draw()");

}

}

public class Square implements Shape {

public void draw() {

System.out.println("Square::draw()");

}

}

public class Circle implements Shape {

public void draw() {

System.out.println("Circle::draw()");

}

}

--3）创建一个外观类

public class ShapeMaker {

private Shape circle;

private Shape rectangle;

private Shape square;

public ShapeMaker() {

circle = new Circle();

rectangle = new Rectangle();

square = new Square();

}

public void drawCircle(){

circle.draw();

}

public void drawRectangle(){

rectangle.draw();

}

public void drawSquare(){

square.draw();

}

}

--4）测试：使用该外观类画出各种类型的形状

public class FacadePatternDemo {

public static void main(String[] args) {

ShapeMaker shapeMaker = new ShapeMaker();

shapeMaker.drawCircle();

shapeMaker.drawRectangle();

shapeMaker.drawSquare();

}

}

# 行为型模式

## 策略模式

### 使用策略模式优化if-else代码

平时的if-else代码：

if (a) { ...//省略 }

else if (b) { ...//省略 }

else { ...//省略 }

缺点：条件少还好，一旦else-if过多这里的逻辑将会比较混乱，并容易出错。

例如：

//定义一个Template接口，其show()方法交给具体业务类去实现

public interface Template {

public void show();

}

@Component

public class Template1 implements Template {

public void show() {

System.out.println("Template1...");

}

}

@Component

public class Template2 implements Template {

public void show() {

System.out.println("Template2...");

}

}

//通过不同类型templateType动态获取Template实例，从而执行对应业务方法

@RestController

public class TemplateController {

@Autowired

private Map<String, Template> bcMap;

@RequestMapping("/template")

public void show(String templateType) {

bcMap.get(templateType).show();

}

}

## 责任链模式

相关链接：http://www.runoob.com/design-pattern/chain-of-responsibility-pattern.html

责任链模式为请求创建一个接收者对象的链。

在此模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，以此类推。

### 介绍

意图：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并沿着这条链传递请求，直到对象处理它为止。

主要解决：职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了。

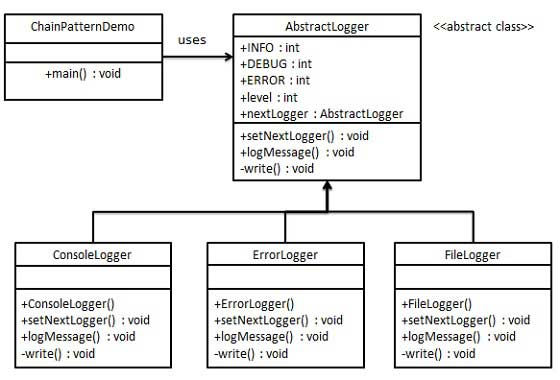
应用实例：1.JS中的事件冒泡 2.Java web中Apache Tomcat对Encoding的处理，Structs2的拦截器，jsp servlet的Filter

优点：1.降低耦合度，它将请求的发送者和接收者解耦 2.简化了对象。使得对象不需要知道链的结构

应用场景：1.可以有多个对象可以处理同一请求，具体哪个对象处理该请求由运行时刻自动确定。 2.在不明确指定接收者的情况下，向多个对象的一个提交一个请求。 3.可动态指定一组对象处理请求。

实例：

我们创建抽象类AbstractLogger，带有详细的日志记录级别。然后我们创建三种类型的记录器，都扩展了AbstractLogger。每个记录器消息的级别是否属于自己的级别，如果是则相应地打印出来，否则将不打印并把消息传给下一个记录器。



--1）创建抽象的记录器类AbstractLogger

public abstract class AbstractLogger {

public static int INFO = 1;

public static int DEBUG = 2;

public static int ERROR = 3;

protected int level; // 级别

protected AbstractLogger nextLogger; // 责任链中的下一个元素

public void setNextLogger(AbstractLogger nextLogger){

this.nextLogger = nextLogger;

}

public void logMessage(int level, String message){

if(this.level <= level){

write(message);

}

if(nextLogger !=null){ // 有下一个记录器

nextLogger.logMessage(level, message);

}

}

abstract protected void write(String message); // 记录事件

}

--2）创建扩展了该记录器类的实现类ConsoleLogger、ErrorLogger、FileLogger

public class ConsoleLogger extends AbstractLogger {

public ConsoleLogger(int level){

this.level = level;

}

@Override

protected void write(String message) {

System.out.println("Standard Console::Logger: " + message);

}

}

public class ErrorLogger extends AbstractLogger {

public ErrorLogger(int level){

this.level = level;

}

@Override

protected void write(String message) {

System.out.println("Error Console::Logger: " + message);

}

}

public class FileLogger extends AbstractLogger {

public FileLogger(int level){

this.level = level;

}

@Override

protected void write(String message) {

System.out.println("File::Logger: " + message);

}

}

--3）创建不同类型的记录器。

赋予它们不同的错误级别，并在每个记录器中设置下一个记录器。每个记录器中的下一个记录器代表的是链的一部分。

public class ChainPattern {

public static AbstractLogger getChainOfLoggers() {

AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);

AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);

AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);

errorLogger.setNextLogger(fileLogger);

fileLogger.setNextLogger(consoleLogger);

return errorLogger;

}

}

--4）测试

public class Main {

public static void main(String[] args) {

AbstractLogger loggerChain = getChainOfLoggers();

loggerChain.logMessage(AbstractLogger.INFO, "This is an information.");

loggerChain.logMessage(AbstractLogger.DEBUG, "This is an debug level information.");

loggerChain.logMessage(AbstractLogger.ERROR, "This is an error information.");

}

}

输出结果：

Standard Console::Logger: This is an information.

File::Logger: This is an debug level information.

Standard Console::Logger: This is an debug level information.

Error Console::Logger: This is an error information.

File::Logger: This is an error information.

Standard Console::Logger: This is an error information.

# Ss