## 设计模式的六大原则

### 开闭原则

程序在进行扩展的时候，不能够去修改原有的代码。总而言之，要想使程序的可扩展性更好，易于维护和升级，就得使用接口和抽象类

### 2、里氏代换原则

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

### 3、依赖倒转原则

就是针对接口编程，依赖于抽象而不依赖于具体。是开闭原则的基础

### 4、接口隔离原则

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

### 5、迪米特法则（最少知道原则）

一个实体应当尽量少的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

原则是尽量使用合成/聚合的方式，而不是使用继承

## 工厂方法模式

### 普通工厂模式

即实现同一接口的一些类的实例的创建，需要根据传递的字符串来创建对象

### 2、多工厂模式

普通工厂模式的改进，如果传递的字符串错误，则不能正确的创建对象，而多工厂模式通过提供多个工厂方法，即可创建自己所需的实例

1. **public** **class** SendFactory {
3. **public** Sender produceMail(){
4. **return** **new** MailSender();
5. }
7. **public** Sender produceSms(){
8. **return** **new** SmsSender();
9. }
10. }
11. **public** **class** FactoryTest {
13. **public** **static** **void** main(String[] args) {
14. SendFactory factory = **new** SendFactory();
15. Sender sender = factory.produceMail();
16. sender.Send();
17. }
18. }

## 抽象工厂模式

工厂模式有一个缺陷就是类的创建依赖工厂类，若要新增一个工厂类，就必须修改源代码来实现类的创建。但是通过抽象工厂模式就可以解决这一问题。

首先需要理解的是抽象工厂是创建一个抽象实例的，即抽象Provider是来创建一个抽象Send

Provider Send

SendMailFactory  MailSender

1. **public** **interface** Sender {
2. **public** **void** Send();
3. }

两个实现类：

1. **public** **class** MailSender **implements** Sender {
2. @Override
3. **public** **void** Send() {
4. System.out.println("this is mailsender!");
5. }
6. }
7. **public** **class** SmsSender **implements** Sender {
9. @Override
10. **public** **void** Send() {
11. System.out.println("this is sms sender!");
12. }
13. }

两个工厂类：

**[java]** [view plain](http://blog.csdn.net/zhangerqing/article/details/8194653) [copy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SendMailFactory **implements** Provider {
3. @Override
4. **public** Sender produce(){
5. **return** **new** MailSender();
6. }
7. }
8. **public** **class** SendSmsFactory **implements** Provider{
10. @Override
11. **public** Sender produce() {
12. **return** **new** SmsSender();
13. }
14. }

在提供一个接口：

1. **public** **interface** Provider {
2. **public** Sender produce();
3. }

测试类：

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Provider provider = **new** SendMailFactory();
5. Sender sender = provider.produce();
6. sender.Send();
7. }
8. }

## 单例模式

在多线程应用中，为了实现某一对象只有一个实例的存在

1. **public** **class** Singleton {
3. /\* 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 \*/
4. **private** **static** Singleton instance = **null**;
6. /\* 私有构造方法，防止被实例化 \*/
7. **private** Singleton() {
8. }
10. /\* 静态工程方法，创建实例 \*/
11. **public** **static** Singleton getInstance() {
12. **if** (instance == **null**) {
13. instance = **new** Singleton();
14. }
15. **return** instance;
16. }
18. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
19. **public** Object readResolve() {
20. **return** instance;
21. }
22. }

小技巧：通过私有构造方法，可以防止被实例化

饿汉式：开发常用

Class Singleton{

private Singleton { }; //私有构造方法防止实例化

private static Singleton s = new Singleton( );

public static Singleton getInstance(){

return s;

} }

懒汉式：面试用

Class Singleton{

private Singleton { }; //私有构造方法防止实例化

private static Singleton s;

public static Singleton getInstance(){

if( s == null ){

s = new Singleton( )

}

return s;

}

}

饿汉式和懒汉式之间的区别：

饿汉式是空间换时间，懒汉式是时间换空间（因为每次都要判断是否为空，然后再创建对象）

在多线程中饿汉式是不会创建多个对象的，但是懒汉式是有可能创建多个对象的

单例模式只是为了能够创建一个对象，所以我们也可以使用final来修饰

Class Singleton{

private Singleton { }; //私有构造方法防止实例化

private static final Singleton s = new Singleton( );

}

## 适配器模式

### 类的适配器模式

希望将一个类转换成满足另一个新接口的类，创建一个新类，继承原有的类，实现新的接口即可，但是需要注意的是原有的类中与接口的方法要一致才行

### 对象的适配器模式

在实现新的接口的类中，持有原有的实例，实现其方法即可

1. **public** **class** Wrapper **implements** Targetable {
3. **private** Source source;
5. **public** Wrapper(Source source){
6. **super**();
7. **this**.source = source;
8. }
9. @Override
10. **public** **void** method2() {
11. System.out.println("this is the targetable method!");
12. }
14. @Override
15. **public** **void** method1() {
16. source.method1();
17. }
18. }

### 接口的适配器模式

要实现某一个接口就必须实现接口中的所有方法，为了能够不实现接口中的全部方法，可以先使用一个抽象类来实现接口中的全部方法，当然抽象类中的方法都为空，都未实现，我们可以继承这个抽象类，然后根据自己的需求来实现自己想要的方法

## 装饰模式

可以给一个对象增加新的功能，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，并且装饰对象要持有被装饰对象的实例，然后调用相应的方法即可，这要即可动态的增加新的功能

接口:

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }
4. **public** **class** Source **implements** Sourceable {
5. @Override
6. **public** **void** method() {
7. System.out.println("the original method!");
8. }
9. }
10. **public** **class** Decorator **implements** Sourceable {
12. **private** Sourceable source;
14. **public** Decorator(Sourceable source){
15. **super**();
16. **this**.source = source;
17. }
18. @Override
19. **public** **void** method() {
20. System.out.println("before decorator!");
21. source.method();
22. System.out.println("after decorator!");
23. }
24. }

测试类

1. **public** **class** DecoratorTest {
3. **public** **static** **void** main(String[] args) {
4. Sourceable source = **new** Source();
5. Sourceable obj = **new** Decorator(source);
6. obj.method();
7. }
8. }

## 代理模式

就是像中介一样，使用一个类来代理另一个类

首先分为JDK代理，Cglib代理

JDK代理，有静态也有动态的，但是必须实现一个接口才行

Cglib代理是不需要实现接口的。

JDK的静态代理:

**public** **class** JdkStaticProxy {

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

CountImp count = **new** CountImp();

CountProxy proxy = **new** CountProxy(count);

proxy.query();

}

}

**interface** Count {

**public** **void** query();

}

**class** CountImp **implements** Count{

@Override

**public** **void** query() {

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

System.***out***.println("查询。。。。");

}

}

**class** CountProxy **implements** Count{

**private** CountImp count;

**public** CountProxy(CountImp count){

**this**.count = count;

}

@Override

**public** **void** query() {

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

System.***out***.println("代理前。。。");

count.query();

}

}

JDK动态代理：

**public** **class** JdkProxy {

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

BookProxy bookProxy = **new** BookProxy();

Book proxy = (Book) bookProxy.createProxy(**new** BookImp());

proxy.query();

}

}

**interface** Book{

**public** **void** query();

}

**class** BookImp **implements** Book{

@Override

**public** **void** query() {

System.***out***.println("查询书籍。。。");

}

}

**class** BookProxy **implements** InvocationHandler{

**private** Object target;

**public** Object createProxy(Object target){

**this**.target = target;

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(),

target.getClass().getInterfaces(), **this**);

}

@Override

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

method.invoke(target, args);

System.***out***.println("调用方法成功。。。");

**return** **null**;

}

}

JDK动态代理，代理类需要实现InvocationHandler接口，需要重写invoke（）方法，然后使用Proxy. newProxyInstance(ClassLoader loader, Class<?>[] interfaces,

InvocationHandler h)来创建一个代理对象。

Cglib动态代理：

**public** **class** CglibProxy {

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

PersonProxy proxy = **new** PersonProxy();

Person person = (Person)proxy.getProxy(**new** Person());

person.query();

}

}

**class** Person{

**public** **void** query(){

System.***out***.println("查询人类。。。");

}

}

**class** PersonProxy **implements** MethodInterceptor{

**private** Object target;

**public** Object getProxy(Object obj){

Enhancer enhance = **new** Enhancer();

**this**.target = obj;

enhance.setSuperclass(obj.getClass());

enhance.setCallback(**this**);

**return** enhance.create();

}

@Override

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {

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

System.***out***.println("前置。。。");

Object object = proxy.invokeSuper(obj, args);

System.***out***.println("后置。。。");

**return** object;

}

}

它的原理是对目标类生成一个子类，并覆盖其方法进行增强，但因为采用的是继承，所以不能够对final类进行代理。

## 外观模式

就是降低类与类之间的耦合程度，将所有的类放在一个类中管理，减少它们之间的依赖。举个简单点的例子就是，现在有A,B,C,D四个类，将它们全部放在F类中进行管理，

## 观察者模式

就相当于邮箱订阅消息，有着用户和系统两者之间的关系，系统里存储着众多用户，系统管理着用户，包括增删，以及推送消息

小技巧：用一个抽象类来实现一个接口，可以不全部实现其中的方法，然后某个类来继承这个抽象类，之前抽象类没有实现的接口中的方法这个类会将它实现，这样就可以很好的隐藏代码

## 命令模式

就相当于一个司令发出命令，命令经过传递到达士兵的口中，士兵去执行命令，司令（Invoke）,命令（Command），士兵（receiver），三者各自做自己的事，

1. **public** **interface** Command {
2. **public** **void** exe();
3. }
4. **public** **class** MyCommand **implements** Command {
6. **private** Receiver receiver;
8. **public** MyCommand(Receiver receiver) {
9. **this**.receiver = receiver;
10. }
12. @Override
13. **public** **void** exe() {
14. receiver.action();
15. }
16. }
17. **public** **class** Receiver {
18. **public** **void** action(){
19. System.out.println("command received!");
20. }
21. }
22. **public** **class** Invoker {
24. **private** Command command;
26. **public** Invoker(Command command) {
27. **this**.command = command;
28. }
30. **public** **void** action(){
31. command.exe();
32. }
33. }
34. **public** **class** Test {
36. **public** **static** **void** main(String[] args) {
37. Receiver receiver = **new** Receiver();
38. Command cmd = **new** MyCommand(receiver);
39. Invoker invoker = **new** Invoker(cmd);
40. invoker.action();
41. }
42. }

## 责任链模式

有多个对象，每一个对象持有下一个对象的引用，这样就会形成一条链，请求在这条链上进行，直到某一个对象决定处理这个请求，但是发出者并不清楚到底是哪个对象处理的该请求