创建型模式

———————————————————————————————————————

结构型模式

代理模式

应用场景

例如：假设有一组对象都实现同一个接口，实现同样的方法，但这组对象中有一部分对象需要有单独的方法，传统的笨办法是在每一个应用端都加上这个单独的方法，但是代码重用性低，[耦合性](http://baike.baidu.com/view/1731634.htm)高。如果用代理的方法则很好的解决了这个问题。

代码示例

假设有一个[Italk](http://baike.baidu.com/view/7639713.htm)接口，有空的方法talk（）（说话），所有的people对象都实现（implements）这个接口，实现talk（）方法，前端有很多地方都将people实例化，执行talk方法，后来发现这些前端里有一些除了要说话以外还要唱歌（sing），那么我们既不能在Italk接口里增加sing()方法，又不能在每个前端都增加sing方法，我们只有增加一个代理类talkProxy，这个代理类里实现talk和sing方法，然后在需要sing方法的客户端调用代理类即可，代码如下：

接口类Italk

public interface **Italk** {

public void talk(String msg);

}

实现类people

public class **People** implements **Italk** {

public String username;

public String age;

public People(String name1, String age1) {

this.username= name1;

this.age = age1;

}

public String getName() {

return username;

}

public void setName(String name) {

this.username= name;

}

public String getAge() {

return age;

}

public void setAge(String age) {

this.age = age;

}

public void talk(String msg) {

System.out.println(msg+"!你好,我是"+username+"，我年龄是"+age);

}

}

代理类talkProxy

public class **TalkProxy** implements **Italk** {

**Italk** talker;

public TalkProxy (Italk talker) {

//super();

this.talker=talker;

}

public void talk(String msg) {

talker.talk(msg);

}

public void talk(String msg,String singname) {

talker.talk(msg);

sing(singname);

}

private void sing(String singname){

System.out.println("唱歌："+singname);

}

}

应用端myProxyTest

public class **MyProxyTest** {

/\*\*[代理模式](http://baike.baidu.com/view/2645890.htm)

\* @param args

\*/

public static void main(String[] args) {

//不需要执行额外方法的

Italk people1 = new People("[湖海散人](http://baike.baidu.com/view/152104.htm)","18");

people1.talk("No ProXY Test");

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

//需要执行额外方法的

TalkProxy talker=new TalkProxy(people1);

talker.talk("ProXY Test","[七里香](http://baike.baidu.com/view/8294.htm)");

}

}

———————————————————————————————————————

行为型模式

**观察者模式 Observer**

　　观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。

　　这个主题对象在状态上发生变化时，会通知所有观察者对象，让它们能够自动更新自己。

**观察者模式的组成**

**抽象主题角色**：把所有对观察者对象的引用保存在一个集合中，每个抽象主题角色都可以有任意数量的观察者。抽象主题提供一个接口，可以增加和删除观察者角色。一般用一个抽象类和接口来实现。

**抽象观察者角色**：为所有具体的观察者定义一个接口，在得到主题的通知时更新自己。

**具体主题角色**：在具体主题内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个子类实现。

**具体观察者角色**：该角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。通常用一个子类实现。如果需要，具体观察者角色可以保存一个指向具体主题角色的引用。

首先定义抽象的观察者：

//抽象观察者角色

public interface Watcher{

public void update(String str);

}

然后定义抽象的主题角色，即抽象的被观察者，在其中声明方法（添加、移除观察者，通知观察者）：

//抽象主题角色，watched：被观察

public interface Watched{

public void addWatcher(Watcher watcher);

public void removeWatcher(Watcher watcher);

public void notifyWatchers(String str);

}

然后定义具体的观察者：

public class ConcreteWatcher implements Watcher{

@Override

public void update(String str) {

System.out.println(str);

}

}

　　之后是具体的主题角色：

import java.util.ArrayList;

import java.util.List;

public class ConcreteWatched implements Watched{

// 存放观察者

private List<Watcher> list = new ArrayList<Watcher>();

@Override

public void addWatcher(Watcher watcher) {

list.add(watcher);

}

@Override

public void removeWatcher(Watcher watcher) {

list.remove(watcher);

}

@Override

public void notifyWatchers(String str) {

// 自动调用实际上是主题进行调用的

for (Watcher watcher : list){

watcher.update(str);

}

}

}

　　编写测试类：

public class Test{

public static void main(String[] args){

Watched girl = new ConcreteWatched();

Watcher watcher1 = new ConcreteWatcher();

Watcher watcher2 = new ConcreteWatcher();

Watcher watcher3 = new ConcreteWatcher();

girl.addWatcher(watcher1);

girl.addWatcher(watcher2);

girl.addWatcher(watcher3);

girl.notifyWatchers("开心");

}

}