观察者模式

1. 开源代码链接

<https://github.com/JackChen-crg/ObserverPatternDemo/commit/0f0a86701f4dadf3a9edf860b1c820b897efc91a>

1. 观察者模式介绍

1、观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。观察者模式是一种对象行为型模式。

2、观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。

这一模式中的关键对象是观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。

作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅(publish-subscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通知。

3、模式结构

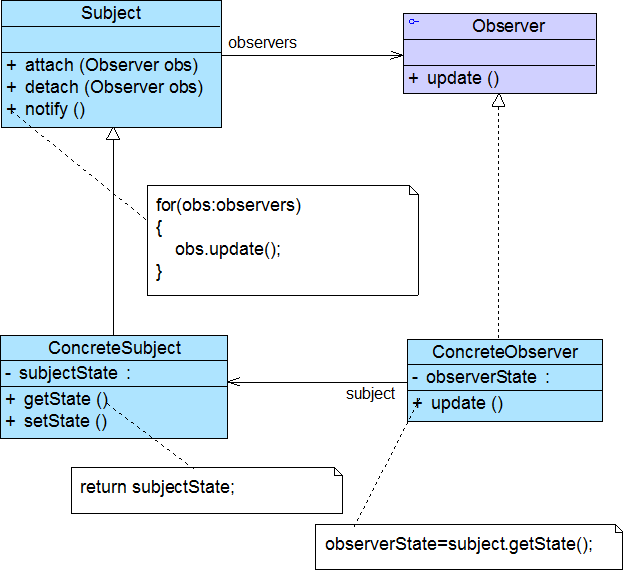
观察者模式包含如下角色：

Subject: 目标

ConcreteSubject: 具体目标

Observer: 观察者

ConcreateObserver：具体观察者



4、观察者模式定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅模式、模型-视图模式、源-监听器模式或从属者模式。观察者模式是一种对象行为型模式。

观察者模式包含四个角色：目标又称为主题，它是指被观察的对象；具体目标是目标类的子类，通常它包含有经常发生改变的数据，当它的状态发生改变时，向它的各个观察者发出通知；观察者将对观察目标的改变做出反应；在具体观察者中维护一个指向具体目标对象的引用，它存储具体观察者的有关状态，这些状态需要和具体目标的状态保持一致。

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目标对象，当这个目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新。

1. 代码分析

他写的是关于温度、湿度、欺压方面一个模式，我进行修改后，写了一个简单的测试观察者模式的小demo

分析：

Observer函数

现在创建一个相关联的观察者。它需要有一个方法能使Subject附属于一个观察者。另外的方法能够接受Subject的变化通知。

**package** com.sweet1118.design.observer;

**public** **interface** Observer {

//method to update the observer, used by subject

**public** **void** update();

//attach with subject to observe

**public** **void** setSubject(Subject sub);

}

Subject函数

基于Subject被观察者的需求，这个是实现一个基本的Subject接口，此接口定了一系列具体的方法需要在随后实现接口的具体类中被实现。

**package** com.sweet1118.design.observer;

**public** **interface** Subject {

//methods to register and unregister observers

**public** **void** register(Observer obj);

**public** **void** unregister(Observer obj);

//method to notify observers of change

**public** **void** notifyObservers();

//method to get updates from subject

**public** Object getUpdate(Observer obj);

}

实现主题

package com.sweet1118.design.observer;

import java.util.ArrayList;

import java.util.List;

public class MyTopic implements Subject {

private List<Observer> observers;

private String message;

private boolean changed;

private final Object MUTEX= new Object();

public MyTopic(){

this.observers=new ArrayList<>();

}

@Override

public void register(Observer obj) {

if(obj == null) throw new NullPointerException("Null Observer");

if(!observers.contains(obj)) observers.add(obj);

}

@Override

public void unregister(Observer obj) {

observers.remove(obj);

}

@Override

public void notifyObservers() {

List<Observer> observersLocal = null;

//synchronization is used to make sure any observer registered after message is received is not notified

synchronized (MUTEX) {

if (!changed)

return;

observersLocal = new ArrayList<>(this.observers);

this.changed=false;

}

for (Observer obj : observersLocal) {

obj.update();

}

}

@Override

public Object getUpdate(Observer obj) {

return this.message;

}

//method to post message to the topic

public void postMessage(String msg){

System.out.println("Message Posted to Topic:"+msg);

this.message=msg;

this.changed=true;

notifyObservers();

}

}

Subject对象

update()方法的实现使用了被观察者的getUpdate()来处理更新的消息。此处应该避免把消息作为参数传递给update()方法。

**package** com.sweet1118.design.observer;

**public** **class** MyTopicSubscriber **implements** Observer {

**private** String name;

**private** Subject topic;

**public** MyTopicSubscriber(String nm){

**this**.name=nm;

}

@Override

**public** **void** update() {

String msg = (String) topic.getUpdate(**this**);

**if**(msg == **null**){

System.***out***.println(name+":: No new message");

}**else**

System.***out***.println(name+":: Consuming message::"+msg);

}

@Override

**public** **void** setSubject(Subject sub) {

**this**.topic=sub;

}

}

主函数（简单测试验证是否正确）

**package** com.sweet1118.design.observer;

**public** **class** ObserverPatternTest {

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

//create subject

MyTopic topic = **new** MyTopic();

//create observers

Observer obj1 = **new** MyTopicSubscriber("观察者1");

Observer obj2 = **new** MyTopicSubscriber("观察者2");

Observer obj3 = **new** MyTopicSubscriber("观察者3");

//register observers to the subject

topic.register(obj1);

topic.register(obj2);

topic.register(obj3);

//attach observer to subject

obj1.setSubject(topic);

obj2.setSubject(topic);

obj3.setSubject(topic);

//check if any update is available

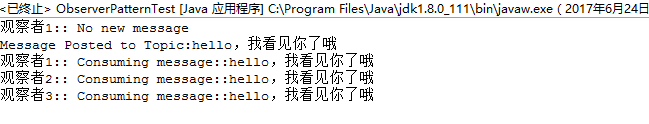
obj1.update();

//now send message to subject

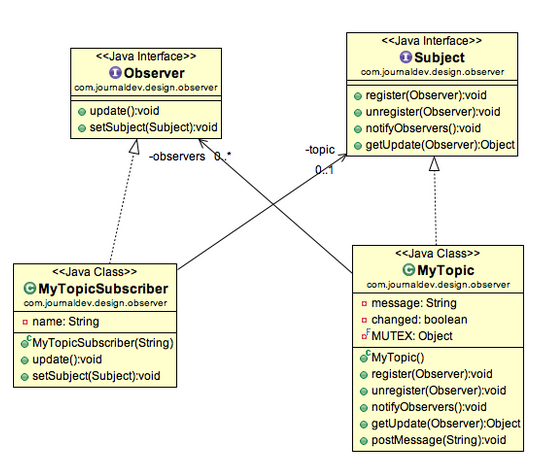
topic.postMessage("hello，我看见你了哦");

}

}

输出结果：

UML图：



1. 总结

实践出真知。

观察者模式的主要优点在于可以实现表示层和数据逻辑层的分离，并在观察目标和观察者之间建立一个抽象的耦合，支持广播通信；其主要缺点在于如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间，而且如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

观察者模式适用情况包括：一个抽象模型有两个方面，其中一个方面依赖于另一个方面；一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变；一个对象必须通知其他对象，而并不知道这些对象是谁；需要在系统中创建一个触发链。

在JDK的java.util包中，提供了Observable类以及Observer接口，它们构成了Java语言对观察者模式的支持。