转载请注明链接:https://blog.csdn.net/feather_wch/article/details/78538902

观察者模式是JDK中使用最多的模式。本文进行详细的介绍。

> 鸣谢: 《Head First 设计模式》

观察者模式详解

版本: 2018/8/23-1(19:00)

介绍

1、什么是观察者模式?

> 1. 观察者模式是JDK中使用最多的模式。
> 2. 观察者模式类似于报社的运行方式，订阅者一开始在报社注册了订阅服务，当报社每天获得新报纸之后，会给所有用户派送报纸，如果用户不想继续看报纸了，也可以去报社退订。这种就是观察者模式： `出版者 + 订阅者 = 观察者模式`
> 3. 出版者就是主题Subject，订阅者就是观察者Observer: `Subject + Observer = 观察者模式`

定义

2、观察者模式的定义

> 定义了对象之间的 `一对多依赖` ， 当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。

应用场景

3、观察者模式在Android中的应用场景

> 1. RecyclerView中就是用了观察者。
>   1. setAdapter()会将RV设置为Adapter的观察者。如果Adapter数据发生改变会通知观察者。
>   2. 如：Adapter的notifyItemChanged()内部会调用，Observable会调用

特点

4、观察者模式的优点

> 1. 交互对象之间耦合度低
> 2. Subject/Observable拥有数据并执行操作，避免让多个对象操作同一份数据，结构清晰，安全。
> 3. 复用性：可以独立复用观察者和被观察者。

5、观察者模式的缺点

> 1. 通知观察者的时间消耗：如果观察者过多，会触发通知比较慢。
> 2. 多个观察者间的循环可能发生死锁：比如观察者在接收通知的方法去调用Subject的通知方法，可能会导致无限循环。
> 3. 内存问题：Subject持有观察者的引用，如果未能及时解注册，可能会导致观察着无法回收。

实例

自定义接口实现

6、观察者模式的实例：天气气象站数据在多个公告板的展示

> 1. 被观察者：天气气象站就是Subject，当数据变动时，通知所有公告牌进行显示。
> 2. 观察者：公告牌就是Observer，设计一个Observer接口和element接口(抽象公告牌的共同部分)。
>
> WetherDate(Subject)-被观察者：

```
public interface Subject {
        public void registerObserver(Observer observer);
        public void removeObserver(Observer observer);
        public void notifycation();
}

public class WetherData implements Subject{
        float temp = 0;
        float humidity = 0;
        boolean isChanged = false;
        ArrayList<Observer> observerList = new ArrayList<>();
    //注册观察者就是在链表中添加观察者
        public void registerObserver(Observer observer) {
                observerList.add(observer);
        }
        //移除观察者
        public void removeObserver(Observer observer) {
                int index = observerList.indexOf(observer);
                observerList.remove(index);
        }
    //通知所有观察者
        public void notifycation() {
                if(isChanged) {
                        for(Observer observer:observerList) {
                                observer.update(temp, humidity);
                        }
                        isChanged = false;
                }
        }
        public void measureDataChanged(float temp, float humidity) {
                this.temp = temp;
                this.humidity = humidity;
                isChanged = true;
                notifycation();
        }
}
```

> Obeserver-观察者:

```
//观察者接口
public interface Observer {
        public void update(float temp, float humidity);
}
//告示牌通用功能
public interface DisplayElement {
        public void display();
}
//中文版告示牌
public class DisplayChinese implements Observer, DisplayElement{
        float temp = 0;
        float humidity = 0;
        public DisplayChinese(WetherData wetherData) {
                wetherData.registerObserver(this);
        }
        public void display() {
                System.out.println("中文数据：当前温度:"+temp+"℃/当前湿度:"+humidity+"%");
        }
        public void update(float temp, float humidity) {
                this.temp = temp;
                this.humidity = humidity;
                display();
        }
}
//英文版告示牌
public class DisplayEnglish implements Observer, DisplayElement{
        float temp = 0;
        float humidity = 0;
        public DisplayEnglish(WetherData wetherData) {
                wetherData.registerObserver(this);
        }
        public void display() {
                System.out.println("English data: temp:"+temp+"℃/humi:"+humidity+"%");
        }
        public void update(float temp, float humidity) {
                this.temp = temp;
                this.humidity = humidity;
                display();
        }
}
```

> 天气测量(测试):

```
public class WetherMeasure {
        public static void main(String[] args) {
                WetherData wetherData = new WetherData();
                DisplayEnglish beijingDisplay = new DisplayEnglish(wetherData);
                DisplayChinese nanjingDisplay = new DisplayChinese(wetherData);
                wetherData.measureDataChanged(20, 60);
        }
}
```

> 运行结果(结果正确):

```
English data：temp:20.0℃/humi:60.0%
中文数据：当前温度:20.0℃/当前湿度:60.0%
```

Java内置观察者模式

7、Java内置了观察者模式

> 1. `java.util.Observable` 包中提供
>
> 1. 观察者： `Observer` 类
> 2. 被观察者： `Observable`

8、Observerable如何发出通知?

> 1. 先调用 `setChanged()` 方法
> 2. 调用 `notifyObservers()`

9、观察者如何获取数据?

> 1. 与常规实现的方法不同，Java内置观察者的方法是观察者去主动获取数据。
> 2. 比如在 `update` 中去 `get` 数据，而不是由被观察者来推送数据。

10、Observable/Observer实现的实例

> JavaWetherData:

```
public class JavaWetherData extends Observable{
        float temp = 0;
        float humi = 0;
        //提供获取数据接口
        public float getTemp() {
                return temp;
        }
        public float getHumi() {
                return humi;
        }
        public void measureData(float temp, float humi) {
                this.temp = temp;
                this.humi = humi;
                setChanged(); //改变标志
                notifyObservers();//推送通知
        }
}
```

> JavaDisplayEnglish: 实现Java提供的 `Observer` 接口

```
public class JavaDisplayEnglish implements Observer, DisplayElement{
        float temp = 0;
        float humi = 0;
        public JavaDisplayEnglish(Observable observable) {
                observable.addObserver(this); //在被观察者中进行注册
        }
        public void display() {
                System.out.println("(Java内置观察者)English data: temp:"+temp+"℃/humi:"+humi+"%
        }
        //更新数据-主动去获取数据
        public void update(Observable arg0, Object arg1) {
                if(arg0 instanceof JavaWetherData) { //确保参数正确
                        JavaWetherData wetherData = (JavaWetherData)arg0;
                        temp = wetherData.getTemp();
                        humi = wetherData.getHumi();
                        display();
                }
        }
}
```

> 数据测试:

```
JavaWetherData javaWetherData = new JavaWetherData();
JavaDisplayEnglish javaDisplayEnglish = new JavaDisplayEnglish(javaWetherData);
javaWetherData.measureData(23, 79);
```

> 结果(结果正确):

```
English data: temp:23.0℃/humi:79.0%
```

11、Java内置Observable缺点

> 1. Observable是一个类而不是interface，影响了复用和使用。
> 2. 违反了 多用组合，少用继承 的原则。

设计原则

12、设计原则四：交互对象之间松耦合设计

13、观察者模式是如何利用设计原则的?

> 1. 交互对象之间松耦合设计
> 2. 针对接口编程
>
>   Observer观察者利用主题的接口向Subject进行注册。
>
>   Subject使用Observer的接口通知观察者

> 3. 多组合少继承
>   对象之间的关系不是通过继承产生，而是在运行时通过组合来产生。

参考资料

1. RecyclerView的原理