《Head First 设计模式》：观察者模式

作者：jingqueyimu
原文链接：https://ld246.com/article/1593786827043
来源网站：链滴

正文

一、定义

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会到通知并自动更新。

要点：

- 观察者模式定义了对象之间一对多的关系。
- 观察者模式让主题（可观察者）和观察者之间松耦合。
- 主题对象管理某些数据，当主题内的数据改变时，会以某种形式通知观察者。
- 观察者可以订阅（注册）主题，以便在主题数据改变时能收到更新。
- 观察者如果不想收到主题的更新通知，可以随时取消订阅（注册）。

二、实现步骤

1、创建主题父类/接口

主题父类/接口主要提供了注册观察者、移除观察者、通知观察者三个方法。

```
/**
 * 主题
 */
public class Subject {

    /**
```

```
     * 观察者列表
     */
    private ArrayList<Observer> observers;

    public Subject() {
        observers = new ArrayList<>();
    }

    /**
     * 注册观察者
     */
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    /**
     * 移除观察者
     */
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    /**
     * 通知所有观察者,并推送数据（也可以不推送数据，而是由观察者过来拉取数据）
     */
    public void notifyObservers(Object data) {
        for (Observer o : observers) {
            o.update(data);
        }
    }
}
```

2、创建观察者接口

观察者接口主要提供了更新方法，以供主题通知观察者时调用。

```
/**
 * 观察者接口
 */
public interface Observer {

    /**
     * 根据主题推送的数据进行更新操作
     */
    public void update(Object data);
}
```

3、创建具体的主题，并继承主题父类/实现主题接口

```
/**
 * 主题A
 */
public class SubjectA extends Subject {
```

```
    /**
     * 主题数据
     */
    private String data;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
        // 数据发生变化时，通知观察者
        notifyObservers(data);
    }
}
```

4、创建具体的观察者，并实现观察者接口

通过观察者类的构造函数，注册成为主题的观察者。

（1）观察者 A

```
/**
 * 观察者A
 */
public class ObserverImplA implements Observer {

    private Subject subject;

    public ObserverImplA(Subject subject) {
        // 保存主题引用，以便后续取消注册
        this.subject = subject;
        // 注册观察者
        subject.registerObserver(this);
    }

    @Override
    public void update(Object data) {
        System.out.println("Observer A：" + data.toString());
    }
}
```

（2）观察者 B

```
/**
 * 观察者B
 */
public class ObserverImplB implements Observer {

    private Subject subject;
```

```
    public ObserverImplB(Subject subject) {
        // 保存主题引用，以便后续取消注册
        this.subject = subject;
        // 注册观察者
        subject.registerObserver(this);
    }

    @Override
    public void update(Object data) {
        System.out.println("Observer B： " + data.toString());
    }
}
```

5、使用主题和观察者对象

```
public class Test {

    public static void main(String[] args) {
        // 主题
        SubjectA subject = new SubjectA();
        // 观察者A
        ObserverImplA observerA = new ObserverImplA(subject);
        // 观察者B
        ObserverImplB observerB = new ObserverImplB(subject);
        // 模拟主题数据变化
        subject.setData("I'm Batman!!!");
        subject.setData("Why so serious...");
    }
}
```

三、举个栗子

1、背景

你的团队刚刚赢得一纸合约，负责建立 Weather-O-Rama 公司的下一代气象站——Internet 气象观站。

该气象站建立在 WeatherData 对象上，由 WeatherData 对象负责追踪目前的天气状况（温度、湿、气压）。并且具有三种布告板，分别显示目前的状况、气象统计以及简单的预报。当 WeatherData 对象获得最新的测量数据时，三种布告板必须实时更新。

并且，这是一个可扩展的气象站，Weather-O-Rama 气象站希望公布一组 API，好让其他开发人员以写出自己的气象布告板，并插入此应用中。

2、实现

（1）创建主题父类

```
/**
 * 主题
 */
```

```
public class Subject {

    /**
     * 观察者列表
     */
    private ArrayList<Observer> observers;

    public Subject() {
        observers = new ArrayList<>();
    }

    /**
     * 注册观察者
     */
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    /**
     * 移除观察者
     */
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    /**
     * 通知所有观察者，并推送数据
     */
    public void notifyObservers(float temperature, float humidity, float pressure) {
        for (Observer o : observers) {
            o.update(temperature, humidity, pressure);
        }
    }
}
```

（2）创建观察者接口

```
/**
 * 观察者接口
 */
public interface Observer {

    /**
     * 更新观测值
     */
    public void update(float temperature, float humidity, float pressure);
}
```

（3）创建气象数据类，并继承主题父类

```
/**
 * 气象数据
 */
```

```
public class WeatherData extends Subject {

    /**
     * 温度
     */
    private float temperature;
    /**
     * 湿度
     */
    private float humidity;
    /**
     * 气压
     */
    private float pressure;

    public void measurementsChanged() {
        // 观测值变化时，通知所有观察者
        notifyObservers(temperature, humidity, pressure);
    }

    /**
     * 设置观测值（模拟观测值变化）
     */
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
}
```

（4）创建布告板，并实现观察者接口

```
/**
 * 目前状态布告板
 */
public class CurrentConditionsDisplay implements Observer {

    private Subject weatherData;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        // 注册观察者
        weatherData.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }
```

```
    public void display() {
        System.out.println("Current conditions：" + temperature + "F degress and " + humidity
 "% humidity");
    }
}

/**
 * 统计布告板
 */
public class StatisticsDisplay implements Observer {

    private Subject weatherData;
    private ArrayList<Float> historyTemperatures;

    public StatisticsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        // 注册观察者
        weatherData.registerObserver(this);
        historyTemperatures = new ArrayList<>();
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.historyTemperatures.add(temperature);
        display();
    }

    public void display() {
        if (historyTemperatures.isEmpty()) {
            return;
        }
        Collections.sort(historyTemperatures);
        float avgTemperature = 0;
        float maxTemperature = historyTemperatures.get(historyTemperatures.size() - 1);
        float minTemperature = historyTemperatures.get(0);
        float totalTemperature = 0;
        for (Float temperature : historyTemperatures) {
            totalTemperature += temperature;
        }
        avgTemperature = totalTemperature / historyTemperatures.size();
        System.out.println("Avg/Max/Min temperature：" + avgTemperature + "/" + maxTempe
ature + "/" + minTemperature);
    }
}

/**
 * 预测布告板
 */
public class ForecastDisplay implements Observer {

    private Subject weatherData;
    private float temperature;
    private float humidity;
```

```
    private float pressure;

    public ForecastDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        // 注册观察者
        weatherData.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        display();
    }

    public void display() {
        System.out.println("Forecast：waiting for implementation...");
    }
}
```

（5）测试

```
public class Test {

    public static void main(String[] args) {
        // 气象数据
        WeatherData weatherData = new WeatherData();
        // 目前状态布告板
        CurrentConditionsDisplay currentConditionsDisplay = new CurrentConditionsDisplay(we
therData);
        // 统计布告板
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        // 预测布告板
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        // 模拟气象观测值变化
        weatherData.setMeasurements(80, 65, 30.4F);
        weatherData.setMeasurements(82, 70, 29.2F);
        weatherData.setMeasurements(78, 90, 29.2F);
    }
}
```