Java 设计模式之观察者模式（十六）

一、前言

本篇主题为行为型模式中的第四个模式–观察者模式。上篇 Java 设计模式主题为[《Java 设计模式之迭代器模式（十五）》。

二、简单介绍

2.1 定义

观察者模式是行为模式之一，定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

2.2 参与角色

1. 被观察者（Subject）：当需要被观察的状态发生变化时，需要通知队列中所有观察者对象。Subject 需要维持（添加，删除，通知）一个观察者对象的队列列表。
2. 观察者（Observer）：接口或抽象类。当 Subject 的状态发生变化时，Observer 对象将通过一个 callback 函数得到通知。

2.3 应用场景

1. 聊天室程序，服务器转发信息给所有客户端。
2. 网络游戏，服务器将客户端状态进行分发。
3. 邮件订阅等。

三、实现方式

记得笔者在上中学时，所在的学校是禁止在寝室打扑克牌的，但是还是有不少学生违背这条规定，笔者就是其中之一。我们就以这个故事为例，在这个案例中，老师是被观察者，学生则是观察者。学生观察老师是否来到寝室，从而做出不同的行为。

被观察者和子类：

```
public abstract class Subject {

    protected List<Observer> list = new ArrayList<Observer>();

    public void registerObserver(Observer obs) {
        list.add(obs);
    }

    public void removeObserver(Observer obs) {
        list.add(obs);
    }

    // 通知所有的观察者更新状态
    public void notifyAllObservers() {
        for (Observer obs : list) {
            obs.update(this);
        }
    }
}

public class Teacher extends Subject {

    private String action;

    public String getAction() {
        return action;
    }

    public void setAction(String action) {
        this.action = action;
        // 被观察者状态发生变化，通知观察者
        this.notifyAllObservers();
    }

}
```

Teacher 继承 Subject 类，Teacher 的实例就是被观察的对象。

观察者和实现类：

```
public interface Observer {

    public void update(Subject subject);
}

public class Student implements Observer {

    private String action;

    @Override
    public void update(Subject subject) {
        String teacherAction = ((Teacher) subject).getAction();

        if (teacherAction.equals("老师来了")) {
            action = "假装学习";
        } else if (teacherAction.equals("老师走了")) {
            action = "继续打牌";
        }
    }

    public String getAction() {
        return action;
    }
}
```

客户端：

```
public class Client {

    public static void main(String[] args) {

        Teacher teacher = new Teacher();

        Student s1 = new Student();
        Student s2 = new Student();
        Student s3 = new Student();

        teacher.registerObserver(s1);
        teacher.registerObserver(s2);
        teacher.registerObserver(s3);

        teacher.setAction("老师来了");

        System.out.println(s1.getAction());
        System.out.println(s2.getAction());
        System.out.println(s3.getAction());

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

        teacher.setAction("老师走了");

        System.out.println(s1.getAction());
        System.out.println(s2.getAction());
        System.out.println(s3.getAction());
    }
}
```

打印结果：

```
假装学习
假装学习
假装学习
===========
继续打牌
继续打牌
继续打牌
```

其实，在 JDK 中已经提供了 java.util.Observable 和 java.util.Observer 来实观察者模式。

实现方式如下：

被观察者：

```
public class Teacher extends Observable{

    private String action;

    public String getAction() {
        return action;
    }

    public void setAction(String action) {
        this.action = action;
        // 被观察者状态发生变化，通知观察者
        this.setChanged();
        this.notifyObservers(this.action);
    }
}
```

观察者：

```
public class Student implements Observer {

    private String action;

    @Override
    public void update(Observable o, Object arg) {

        String teacherAction = ((Teacher) o).getAction();

        if (teacherAction.equals("老师来了")) {
            action = "假装学习";
        } else if (teacherAction.equals("老师走了")) {
            action = "继续打牌";
        }

    }

    public String getAction() {
        return action;
    }
}
```

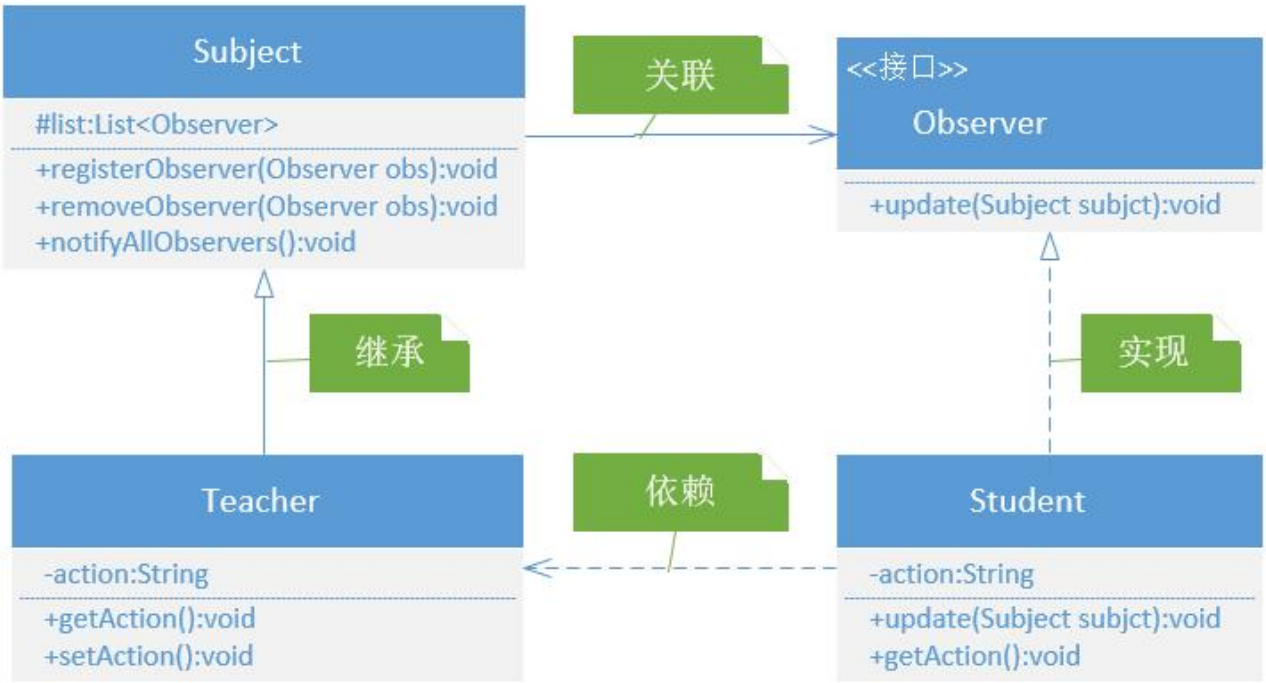
客户端：

```
public class Client {

    public static void main(String[] args) {
        Teacher teacher = new Teacher();

        Student s1 = new Student();
        Student s2 = new Student();
        Student s3 = new Student();

        teacher.addObserver(s1);
        teacher.addObserver(s2);
        teacher.addObserver(s3);

        teacher.setAction("老师来了");

        System.out.println(s1.getAction());
        System.out.println(s2.getAction());
        System.out.println(s3.getAction());

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

        teacher.setAction("老师走了");

        System.out.println(s1.getAction());
        System.out.println(s2.getAction());
        System.out.println(s3.getAction());
    }
}
```

打印结果与上文的一致。

使用 JDK 提供的这 2 个类大大的简化了代码量。

UML 类图表示如下：


Subject
#list:List<Observer>
+registerObserver(Observer obs):void
+removeObserver(Observer obs):void
+notifyAllObservers():void
关联
<<接口>>
Observer
+update(Subject subjct):void
继承
实现
Teacher
-action:String
+getAction():void
+setAction():void
依赖
Student
-action:String
+update(Subject subjct):void
+getAction():void