观察者模式

> *整理自：《java与模式》之观察者模式*

在阎宏博士的《JAVA与模式》一书中开头是这样描述观察者（Observer）模式的：

观察者模式是对象的行为模式，又叫发布-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。

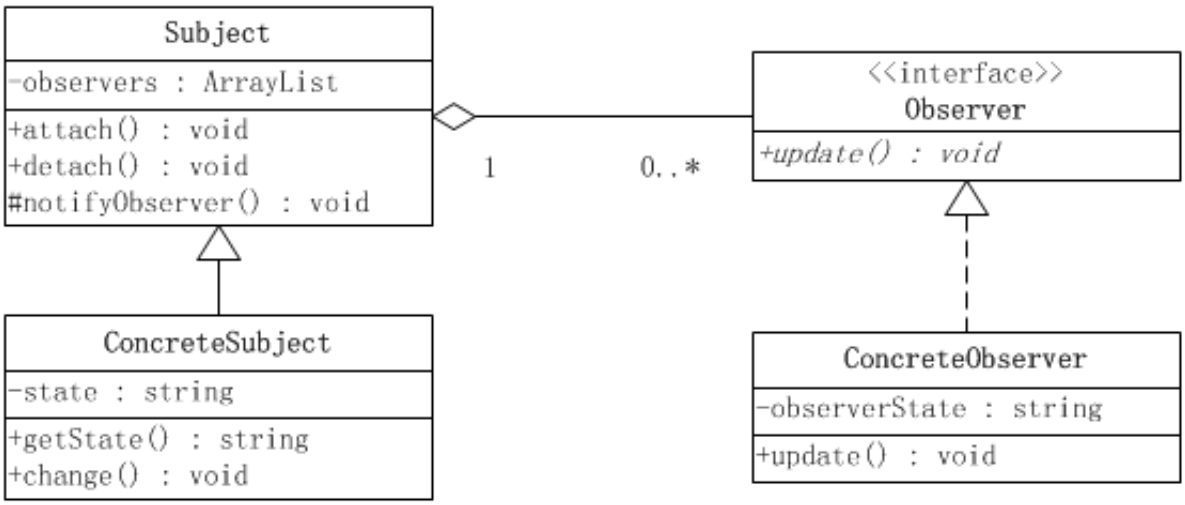
观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

观察者模式的结构

一个软件系统里面包含了各种对象，就像一片欣欣向荣的森林充满了各种生物一样。在一片森林中，各种生物彼此依赖和约束，形成一个个生物链。一种生物的状态变化会造成其他一些生物的相应行动，每一个生物都处于别的生物的互动之中。

同样，一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其他的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作。观察者模式是满足这一要求的各种设计方案中最重要的一种。

下面以一个简单的示意性实现为例，讨论观察者模式的结构。



观察者模式所涉及的角色有：

- **抽象主题(Subject)角色：**抽象主题角色把所有对观察者对象的引用保存在一个聚集（比如ArrayList对象）里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，抽象主题角色又叫做抽象被观察者(Observable)角色。
- **具体主题(ConcreteSubject)角色：**将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色又叫做具体被观察者(Concrete Observable)角色。
- **抽象观察者(Observer)角色：**为所有的具体观察者定义一个接口，在得到主题的通知时更新自己，这个接口叫做更新接口。
- **具体观察者(ConcreteObserver)角色：**存储与主题的状态自恰的状态。具体观察者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 像协调。如果需要，具体观察者角色可以保持一个指向具体主题对象的引用。

源代码

抽象主题角色类

```
public abstract class Subject {
    /**
     * 用来保存注册的观察者对象
     */
    private    List<Observer> list = new ArrayList<Observer>();
    /**
     * 注册观察者对象
     * @param observer    观察者对象
```

```
     */
    public void attach(Observer observer){

        list.add(observer);
        System.out.println("Attached an observer");
    }
    /**
     * 删除观察者对象
     * @param observer    观察者对象
     */
    public void detach(Observer observer){

        list.remove(observer);
    }
    /**
     * 通知所有注册的观察者对象
     */
    public void nodifyObservers(String newState){

        for(Observer observer : list){
            observer.update(newState);
        }
    }
}
```

具体主题角色类

```
public class ConcreteSubject extends Subject{

    private String state;

    public String getState() {
        return state;
    }

    public void change(String newState){
        state = newState;
        System.out.println("主题状态为: " + state);
        //状态发生改变，通知各个观察者
        this.nodifyObservers(state);
```

```
    }
}
```

抽象观察者角色类

```
public interface Observer {
    /**
     * 更新接口
     * @param state    更新的状态
     */
    public void update(String state);
}
```

具体观察者角色类

```
public class ConcreteObserver implements Observer {
    //观察者的状态
    private String observerState;

    @Override
    public void update(String state) {
        /**
         * 更新观察者的状态，使其与目标的状态保持一致
         */
        observerState = state;
        System.out.println("状态为: "+observerState);
    }

}
```

客户端类

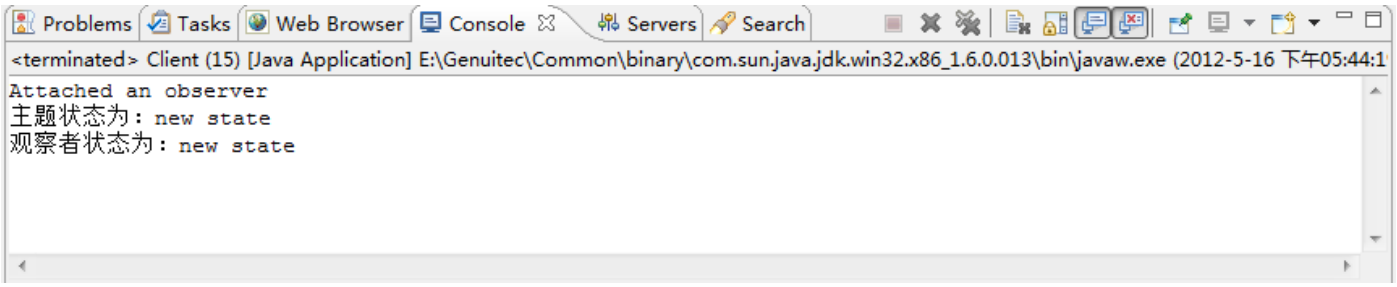
```
public class Client {

    public static void main(String[] args) {
        //创建主题对象
        ConcreteSubject subject = new ConcreteSubject();
        //创建观察者对象
        Observer observer = new ConcreteObserver();
```

```
        //将观察者对象登记到主题对象上
        subject.attach(observer);
        //改变主题对象的状态
        subject.change("new state");
    }

}
```

运行结果如下



在运行时，这个客户端首先创建了具体主题类的实例，以及一个观察者对象。然后，它调用主题对象的attach()方法，将这个观察者对象向主题对象登记，也就是将它加入到主题对象的聚集中去。

这时，客户端调用主题的change()方法，改变了主题对象的内部状态。主题对象在状态发生变化时，调用超类的notifyObservers()方法，通知所有登记过的观察者对象。

推模型和拉模型

在观察者模式中，又分为推模型和拉模型两种方式。

- **推模型**

主题对象向观察者推送主题的详细信息，不管观察者是否需要，推送的信息通常是主题对象的全部或部分数据。

- **拉模型**

主题对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到主题对象中获取，相当于是观察者从主题对象中拉数据。一般这种模型的实现中，会把主题对象自身通过update()方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

根据上面的描述，发现前面的例子就是典型的推模型，下面给出一个拉模型的实例。

拉模型的抽象观察者类

拉模型通常都是把主题对象当做参数传递。

```
public interface Observer {
    /**
     * 更新接口
     * @param subject 传入主题对象，方面获取相应的主题对象的状态
     */
    public void update(Subject subject);
}
```

拉模型的具体观察者类

```
public class ConcreteObserver implements Observer {
    //观察者的状态
    private String observerState;

    @Override
    public void update(Subject subject) {
        /**
         * 更新观察者的状态，使其与目标的状态保持一致
         */
        observerState = ((ConcreteSubject)subject).getState();
        System.out.println("观察者状态为: "+observerState);
    }

}
```

拉模型的抽象主题类

拉模型的抽象主题类主要的改变是nodifyObservers()方法。在循环通知观察者的时候，也就是循环调用观察者的update()方法的时候，传入的参数不同了。

```
public abstract class Subject {
    /**
     * 用来保存注册的观察者对象
     */
```

```
    private    List<Observer> list = new ArrayList<Observer>();
    /**
     * 注册观察者对象
     * @param observer    观察者对象
     */
    public void attach(Observer observer){

        list.add(observer);
        System.out.println("Attached an observer");
    }
    /**
     * 删除观察者对象
     * @param observer    观察者对象
     */
    public void detach(Observer observer){

        list.remove(observer);
    }
    /**
     * 通知所有注册的观察者对象
     */
    public void nodifyObservers(){

        for(Observer observer : list){
            observer.update(this);
        }
    }
}
```

拉模型的具体主题类

跟推模型相比，有一点变化，就是调用通知观察者的方法的时候，不需要传入参数了。

```
public class ConcreteSubject extends Subject{

    private String state;

    public String getState() {
        return state;
    }
```

```
    public void change(String newState){
        state = newState;
        System.out.println("主题状态为: " + state);
        //状态发生改变，通知各个观察者
        this.nodifyObservers();
    }
}
```

两种模式的比较

- 推模型是假定主题对象知道观察者需要的数据；而拉模型是主题对象不知道观察者具体需要什么数据，没有办法的情况下，干脆把自身传递给观察者，让观察者自己去按需要取值。
- 推模型可能会使得观察者对象难以复用，因为观察者的update()方法是按需要定义的参数，可能无法兼顾没有考虑到的使用情况。这就意味着出现新情况的时候，就可能提供新的update()方法，或者是干脆重新实现观察者；而拉模型就不会造成这样的情况，因为拉模型下，update()方法的参数是主题对象本身，这基本上是主题对象能传递的最大数据集合了，基本上可以适应各种情况的需要。

JAVA提供的对观察者模式的支持

在JAVA语言的java.util库里面，提供了一个Observable类以及一个Observer接口，构成JAVA语言对观察者模式的支持。

Observer接口

这个接口只定义了一个方法，即update()方法，当被观察者对象的状态发生变化时，被观察者对象的notifyObservers()方法就会调用这一方法。

```
public interface Observer {

    void update(Observable o, Object arg);
}
```

Observable类

被观察者类都是java.util.Observable类的子类。java.util.Observable提供公开的方法支持观察者对象，这些方法中有两个对Observable的子类非常重要：一个是setChanged()，另一

个是notifyObservers()。第一方法setChanged()被调用之后会设置一个内部标记变量，代表被观察者对象的状态发生了变化。第二个是notifyObservers()，这个方法被调用时，会调用所有登记过的观察者对象的update()方法，使这些观察者对象可以更新自己。

```
public class Observable {
    private boolean changed = false;
    private Vector obs;

    /** Construct an Observable with zero Observers. */

    public Observable() {
    obs = new Vector();
    }

    /**
     * 将一个观察者添加到观察者聚集上面
     */
    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
    if (!obs.contains(o)) {
        obs.addElement(o);
    }
    }

    /**
     * 将一个观察者从观察者聚集上删除
     */
    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }

    public void notifyObservers() {
    notifyObservers(null);
    }

    /**
     * 如果本对象有变化（那时hasChanged 方法会返回true）
     * 调用本方法通知所有登记的观察者，即调用它们的update()方法
     * 传入this和arg作为参数
```

```
 */
public void notifyObservers(Object arg) {

    Object[] arrLocal;

synchronized (this) {

    if (!changed)
            return;
        arrLocal = obs.toArray();
        clearChanged();
    }

    for (int i = arrLocal.length-1; i>=0; i--)
        ((Observer)arrLocal[i]).update(this, arg);
}

/**
 * 将观察者聚集清空
 */
public synchronized void deleteObservers() {
obs.removeAllElements();
}

/**
 * 将“已变化”设置为true
 */
protected synchronized void setChanged() {
changed = true;
}

/**
 * 将“已变化”重置为false
 */
protected synchronized void clearChanged() {
changed = false;
}

/**
 * 检测本对象是否已变化
 */
```

```
    public synchronized boolean hasChanged() {
    return changed;
    }

    /**
     * Returns the number of observers of this <tt>Observable</tt> objec
t.
     *
     * @return  the number of observers of this object.
     */
    public synchronized int countObservers() {
    return obs.size();
    }
}
```

这个类代表一个被观察者对象，有时称之为主题对象。一个被观察者对象可以有数个观察者对象，每个观察者对象都是实现Observer接口的对象。在被观察者发生变化时，会调用Observable的notifyObservers()方法，此方法调用所有的具体观察者的update()方法，从而使所有的观察者都被通知更新自己。

怎样使用JAVA对观察者模式的支持

这里给出一个非常简单的例子，说明怎样使用JAVA所提供的对观察者模式的支持。在这个例子中，被观察对象叫做Watched；而观察者对象叫做Watcher。Watched对象继承自java.util.Observable类；而Watcher对象实现了java.util.Observer接口。另外有一个Test类扮演客户端角色。

源代码

被观察者Watched类源代码

```
public class Watched extends Observable{

    private String data = "";

    public String getData() {
        return data;
    }
```

```
    public void setData(String data) {

        if(!this.data.equals(data)){
            this.data = data;
            setChanged();
        }
        notifyObservers();
    }


}
```

观察者类源代码

```
public class Watcher implements Observer{

    public Watcher(Observable o){
        o.addObserver(this);
    }

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

        System.out.println("状态发生改变: " + ((Watched)o).getData());
    }

}
```

测试类源代码

```
public class Test {

    public static void main(String[] args) {

        //创建被观察者对象
        Watched watched = new Watched();
        //创建观察者对象，并将被观察者对象登记
        Observer watcher = new Watcher(watched);
        //给被观察者状态赋值
        watched.setData("start");
```

```
        watched.setData("run");
        watched.setData("stop");

    }

}
```

Test对象首先创建了Watched和Watcher对象。在创建Watcher对象时，将Watched对象作为参数传入；然后Test对象调用Watched对象的setData()方法，触发Watched对象的内部状态变化；Watched对象进而通知实现登记过的Watcher对象，也就是调用它的update()方法。