第03课：观察者模式

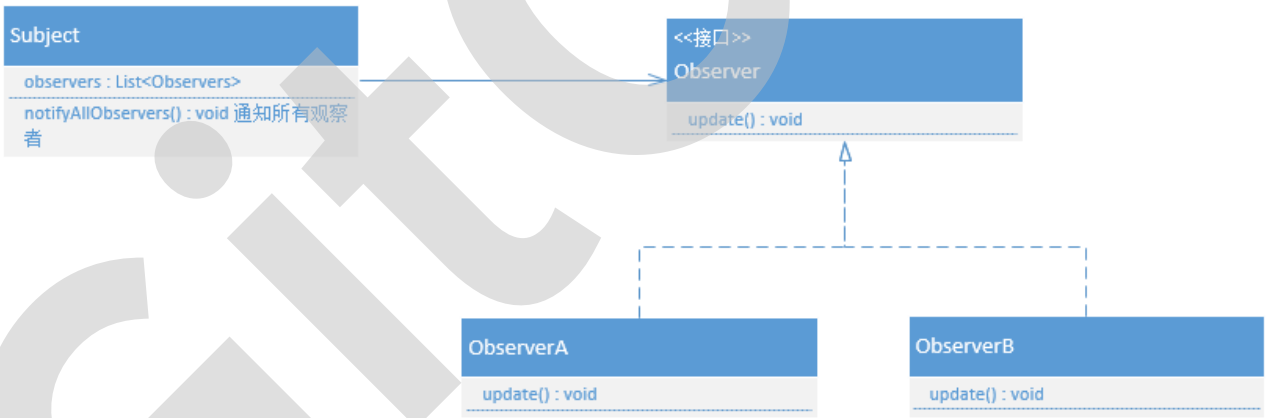
> 观察者模式定义了对象之间的**一对多**依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到**通知**并自动更新。

发布—订阅

在首篇导读中介绍了设计模式的分类，而观察者模式属于行为型模式，行为型模式关注的是对象之间的通讯，观察者模式就是观察者和被观察者之间的通讯。

观察者模式有一个别名叫“订阅—发布模式”。报纸大家都订过吧，当你订阅了一份报纸，每天都会有一份最新的报纸送到你手上，有多少人订阅报纸，报社就会发多少份报纸，这是典型的订阅—发布模式，报社和订报纸的客户就是上面文章开头所说的“一对多”的依赖关系。

观察者模式类图：



其实观察者模式也很简单，从上图可以看出观察者模式就两种角色，一是观察者，二是被观察者（主题），也可以认为是订阅者和发布者。

从逻辑上来说，观察者模式就是：当**被**观察者的状态改变了，就通知观察者，怎么通知呢？从类图中可以看到，被观察者保存了一份所有观察者的列表，只要调用观察者对象的 update() 方法即可。

用发布—订阅报纸的实例来说的话，就是客户们向报社订阅报纸，报社要保存一份所有客户的地址名单，然后有新报纸来了，就按照这个名单一个个派送报纸。

具体的代码实例如下。

观察者（客户）：

```
public abstract class Customer {
    public abstract void update();
}

public class CustomerA extends Customer {
    @Override
    public void update() {
        System.out.println("我是客户A,我收到报纸啦");
    }
}

public class CustomerB extends Customer {
    @Override
    public void update() {
        System.out.println("我是客户B,我收到报纸啦");
    }
}
```

本例子中 Customer 使用了抽象类，而不是接口，跟上面 UML 中不太一样。再次强调，设计模式传达给我们的是一种编程思想，要变通。

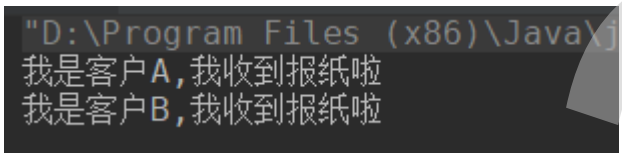
被观察者（报社）：

```
/**
 * 报社（被观察者）
 */
public class NewsOffice {

    private List<Customer> customers = new ArrayList<>();

    public void addCustomer(Customer customer){
        this.customers.add(customer);
    }
    //模拟报纸来了
    public void newspaperCome(){
        this.notifyAllObservers();
    }

    public void notifyAllObservers(){
        for (Customer customer : customers){
            customer.update();
        }
    }
}
```

测试：

```
NewsOffice office= new NewsOffice();

Customer customerA = new CustomerA();
Customer customerB = new CustomerB();
//客户A订阅报纸
office.addCustomer(customerA);
//客户B订阅报纸
office.addCustomer(customerB);
//报纸来了
office.newspaperCome();
```

运行结果：



观察者模式最重要的一点是要搞清楚到底谁是观察者，谁是被观察者，一定要分清楚，这里笔者送你一句金玉良言，只要记住这一句话，观察者模式就不是问题：

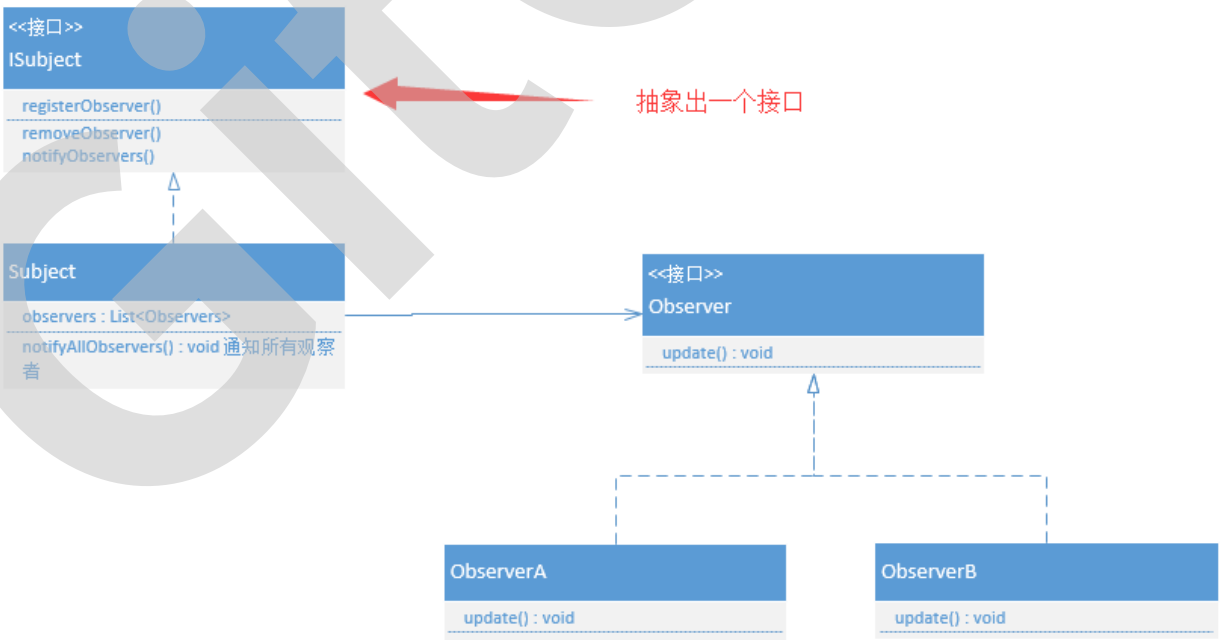
> **收通知的就是观察者。**

如果观察者和被观察者这几个字已被混淆，可以用报纸发布订阅来套。

客户是订阅报纸，收报纸的人（**收通知**），所以客户就是观察者，那么报社就是被观察者了。

改进

上面的观察者模式其实还可以做一个小小的改进，很简单，给被观察者上头抽象出一个 ISubject 接口：



这样做有什么好处呢？

我们将被观察者中通用的方法，比如添加观察者、删除观察者、通知观察者等抽象出来，因为这是所有观察者模式中都要用到的三个方法。

我们举的例子是报社订报纸的例子，如果换了其他案例或场景，观察者不再是订报纸的客户，那么需要 new 一个新的观察者类，实现 ISubject 接口。所有的观察者都有实现了 ISubject 这个接口，那么是不是就有多态的特性可以利用啦！

比如我们可以不用 Customer，而用 ISubject，是不是灵活得多。

```
public void addObserver(ISubject subject){
    //添加被观察者
}
```

很明显这种方式提供了代码的灵活性，改进后的代码如下。

观察者：

```
public abstract class Observer {
    public abstract void update();
}
public class CustomerA extends Observer {
    @Override
    public void update() {
        System.out.println("我是客户A,我收到报纸啦");
    }
}
public class CustomerB extends Observer {
    @Override
    public void update() {
        System.out.println("我是客户B,我收到报纸啦");
    }
}
```

被观察者（客户）：

```
public interface ISubject {

    public void registerObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObservers();
}
/**
 * 报社（被观察者）
 */
public class NewsOffice implements ISubject{

    private List<Observer> observers = new ArrayList<>();

    //模拟报纸来了
    public void newspaperCome(){
        this.notifyObservers();
    }
```

```
    @Override
    public void registerObserver(Observer observer) {
        this.observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers){
            observer.update();
        }
    }
}
```

测试：

```
ISubject office = new NewsOffice();

Observer customerA = new CustomerA();
Observer customerB = new CustomerB();

office.registerObserver(customerA);
office.registerObserver(customerB);

((NewsOffice) office).newspaperCome();
```

看了改进后的例子，你会发现 NewsOffice 中有重复代码：

```
@Override
public void registerObserver(Observer observer) {
    this.observers.add(observer);
}

@Override
public void removeObserver(Observer observer) {
    observers.remove(observer);
}

@Override
public void notifyObservers() {
    for (Observer observer : observers){
        observer.update();
    }
}
```

这三个方法在每一个 ISubject 的实现类中的代码都基本一致，我们并不想重复写，因此可以写一个默认实现类：

```
public abstract class BaseSubject implements ISubject {

    private List<Observer> observers = new ArrayList<>();

    @Override
    public void registerObserver(Observer observer) {
        this.observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers){
            observer.update();
        }
    }
}
/**
 * 报社（被观察者）
 */
public class NewsOffice extends BaseSubject{

    //模拟报纸来了
    public void newspaperCome(){
        this.notifyObservers();
    }

}
```

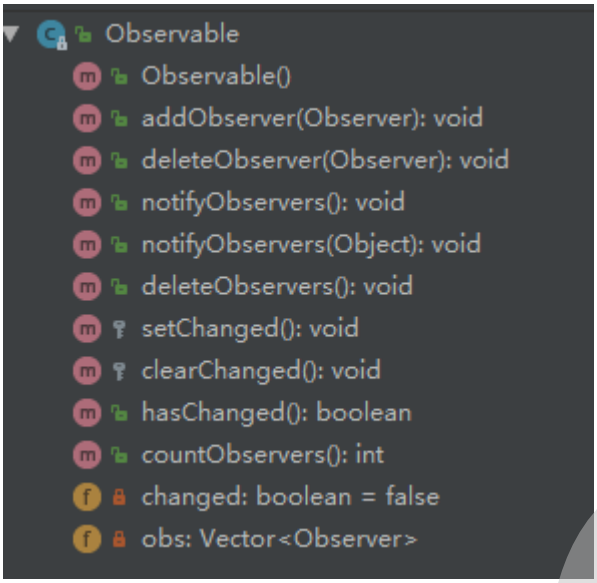
但是其实并**不推荐**这种方式，这是一种偷懒的做法，虽然方便，但是会带来一些问题：BaseSubject 是一个类，所有的被观察者都要继承这个类，如果被观察者既要拥有 BaseSubject 的能力，又想拥有其他类的能力怎么办？要知道 Java 是不支持多继承的，所以还是主要推荐使用接口的方式。

Java 提供的观察者模式

现在我告诉大家一个好消息：

Java 已经为我们提供观察者模式所需要的类：Observer 类和 Subject 类，只不过在 Java 中 Subject 不叫 Subject，而叫 Observable。

我们打开 Observable 类看一下它的结构：



有没有发现我们所熟悉的 addObserver()、notifyObservers() 等方法？

那么 Observable 跟我们自己写的 Subject 有什么区别呢？仔细看一下它的源码你会发现它保存观察者列表不是用 List，而是用：

```
private Vector<Observer> obs
```

同时通知观察者时用了 synchronized 关键字：

```
public void notifyObservers(Object arg) {
        /*
         * a temporary array buffer, used as a snapshot of the state of
         * current Observers.
         */
        Object[] arrLocal;

        synchronized (this) {
            /* We don't want the Observer doing callbacks into
             * arbitrary code while holding its own Monitor.
             * The code where we extract each Observable from
             * the Vector and store the state of the Observer
             * needs synchronization, but notifying observers
             * does not (should not).  The worst result of any
             * potential race-condition here is that:
             * 1) a newly-added Observer will miss a
             *   notification in progress
             * 2) a recently unregistered Observer will be
             *   wrongly notified when it doesn't care
             */
            if (!changed)
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }
```

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

不得不佩服，Java 考虑到了线程安全问题，跟它比起来我们写的代码被碾成了“渣渣辉”。

所以很多情况下直接使用 Java 提供的就可以了，没必要自己去实现。但这不是绝对的，前面已经说过了，Java 是不支持多继承，而 Java 提供的 Observable 是一个类，不是接口，所以会带来一些问题，其中利弊，需要你自己去衡量。

下面给出一个 demo：

```
public class NewsOffice2 extends Observable {

    /**
     * 模拟报纸来了
     */
    public void newspaperCome(){
        this.setChanged();
        this.notifyObservers();
    }
}
public class CustomerC implements Observer {

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("我是客户C，我收到消息啦");
    }
}
```

测试：

```
Observableoffice = new NewsOffice2();
Observer observer = new CustomerC();

office.addObserver(observer);

((NewsOffice2) office).newspaperCome();
```

仔细对比，你会发现 Java 提供的 Observer 的 update 方法给我们传了 Observable 对象：

```
@Override
public void update(Observable o, Object arg) {
    System.out.println("我是客户C，我收到消息啦");
}
```

有时候需要传递数据，这个参数就很有用了。

综合上面各个例子，建议：

- **如果只是简单的场景使用观察者模式，可以直接用 Java 提供的。**
- **复杂的场景建议自己写，基于接口写，同时可以抄袭一下 Java 内置的 Observer 类中关于线程安全的代码。**

案例：View-Model

大名鼎鼎的 MVC 模式大家一定都听过，MVC 分别指 Model、View、Controller。在标准 MVC 模型中，当 Model 改变时，View 视图会自动改变；View 和 Model 之间就是典型的观察者模式。

谁是观察者，谁是被观察者？还记得我在前面说的“金玉良言”吗？**收通知的就是观察者。**在 View 和 Model 中，显然 View 收通知的那一方，那么 View 就是观察者。

在 Web 应用中，由于 View 是在浏览器端展示，而 Model 是在服务端，因此不好体现观察者模式，但是在桌面应用中体现得淋漓尽致。现在我们来写一个简单的例子。

先来一个 User Model（被观察者/主题）：

```
public class User extends Observable {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
        this.setChanged();
        this.notifyObservers();
    }
}
```

setName 的时候，也就是 Model 改变了，通知观察者们。

来一个 View：

```
public class MyFrame extends JFrame implements Observer {

    private Label nameLabel;

    public MyFrame() {
        //创建一个label用于显示name
        this.nameLabel = new Label();
        this.nameLabel.setText("none");
        this.getContentPane().add(nameLabel);
        //设置窗口大小
        this.setLocation(0,0);
        this.setSize(100,100);
    }

    /**
     * model改变会调用该方法

     * @param o
```

```
     * @param arg
     */
    @Override
    public void update(Observable o, Object arg) {
        String name = ((User)o).getName();
        this.nameLabel.setText(name);
    }
}
```

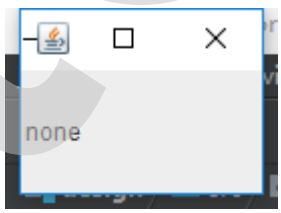
这个 View 是一个桌面窗口，MyFrame() 构造函数中是一些初始化操作，不懂没关系，可以忽略不看，重点是它是一个观察者。

测试：

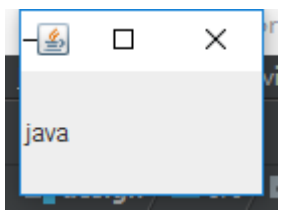
```
Frame frame = new MyFrame();
frame.setVisible(true);

User user = new User();
user.addObserver((Observer) frame);

try {
    Thread.sleep(3000);
    user.setName("java");
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

为了看到效果，我们会在窗口打开后睡眠3秒，然后修改 Model。

运行，显示窗口如下图所示：



三秒后：



除了 MVC，大家可能还听过 MVVM，MVC 是 Model 改变 View 自动更新，而 MVVM 是双向绑定，View 改变，Model 也自动更新。其实原理很简单，双方互为观察者即可。

优缺点

优点：

观察者和被观察之间抽象耦合，自有一套触发机制，被观察者无需知道通知对象是谁，只要是符合观察者接口的就可以。

缺点：

- 观察者只知道被观察发生变化，而无法知道是如何发生变化的，比如是修改了 name 字段还是其他，观察者都不知道。
- 如果有很多个观察者，一个个通知比较耗时。