

# 开发模式之一 观察者模式

**Java一共有23种开发模式** 观察者模式是软件设计模式的一种。在此种模式中，一个目标对象管理所有相依赖于它的观察者对象，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。此种模式通常被用来实现事件处理系统。

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知依赖它的对象。观察者模式属于行为型模式。

**观察者（Observer）** 英式发音：[əb'zɜ:və] 美式发音：[əb'zɜ:rver]

**被观察者（subject）** 英式发音：['sʌbdʒɪkt, səb'dʒekt] 美式发音：['sʌbdʒɪkt, səb'dʒekt]

**总结：**

**观察者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体对象，这个主体对象在状态发生变化时，会通知所有观察者对象，使他们能自动更新自己。**

## 主要解决

一个对象状态改变给其他对象通知的问题，而且还要考虑到易用和低耦合，保证高度的协作。

## 优点

- 观察者和被观察者是抽象耦合的。
- 建立一套触发机制。

## 缺点

- 如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

## 使用场景

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
- 一个对象必须通知其他对象，而并不知道这些对象是谁。
- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象.....，可以使用观察者模式创建一种链式触发机制。

总：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

## 代码实现

### 步骤1, 创建抽象Subject类

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public abstract class Subject {
5      //抽象类
6      protected List<Observer> observers = new ArrayList<Observer>();
7      //增加观察者方法
8      public void add(Observer observer){
9          observers.add(observer);
10     }
11     //删除观察者的方法
12     public void remove(Observer observer){
13         observers.remove(observer);
14     }
15     //通知观察者的方法
16     public abstract void notifyObserver(String sum);
17
18 }
```

### 步骤2, 创建 Observer 接口

```
1  public interface Observer {
2      //抽象观察者
3      void response(); //观察者做出的响应
4  }
5
```

### 步骤3, 创建具体被观察类

```
1  public class ConcreteSubject extends Subject{
2      @Override
3      public void notifyObserver(String sum) {
4          //具体目标
5          System.out.println("目标发生改变 "+sum);
6          System.out.println("-----");
7          //因为对象被放入集合中, 所以循环调用每个对象的响应的方法
8          for(Object obs:observers){
9              ((Observer) obs).response();
10         }
11     }
12 }
```

### 步骤4, 创建观察者类1

```
1  public class ConcreteSubject1 implements Observer{
2      //具体观察者1
3      @Override
4      public void response() {
5          System.out.println("具体观察者1做出响应");
6      }
7  }
8
```

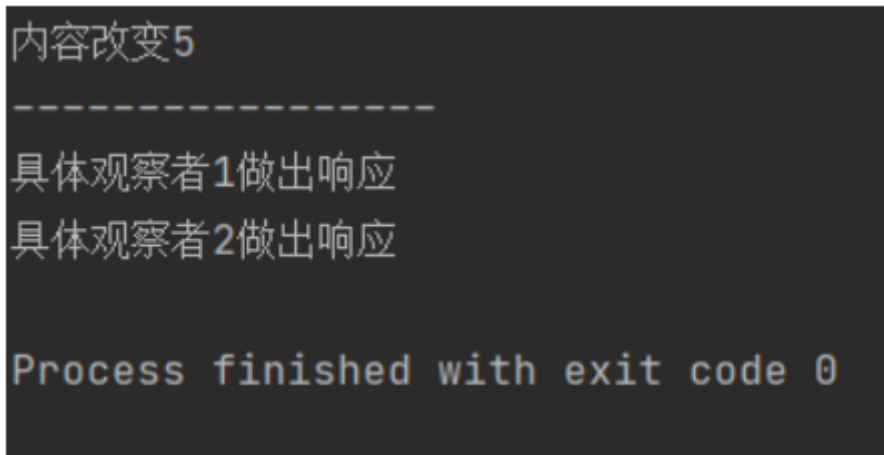
步骤5, 创建观察者类2

```
1 public class ConcreteSubject2 implements Observer{
2     @Override
3     public void response() {
4         System.out.println("具体观察者2做出响应");
5     }
6 }
7
```

测试类

```
1 public class Demo {
2     public static void main(String[] args){
3         Subject subject = new ConcreteSubject();
4         Observer observer1 = new ConcreteSubject1();
5         Observer observer2 = new ConcreteSubject2();
6         //添加观察者, 这里是将观察者对象直接放入集合
7         subject.add(observer1);
8         subject.add(observer2);
9         //通知观察者
10        subject.notifyObserver("5");
11    }
12 }
13
```

执行结果:

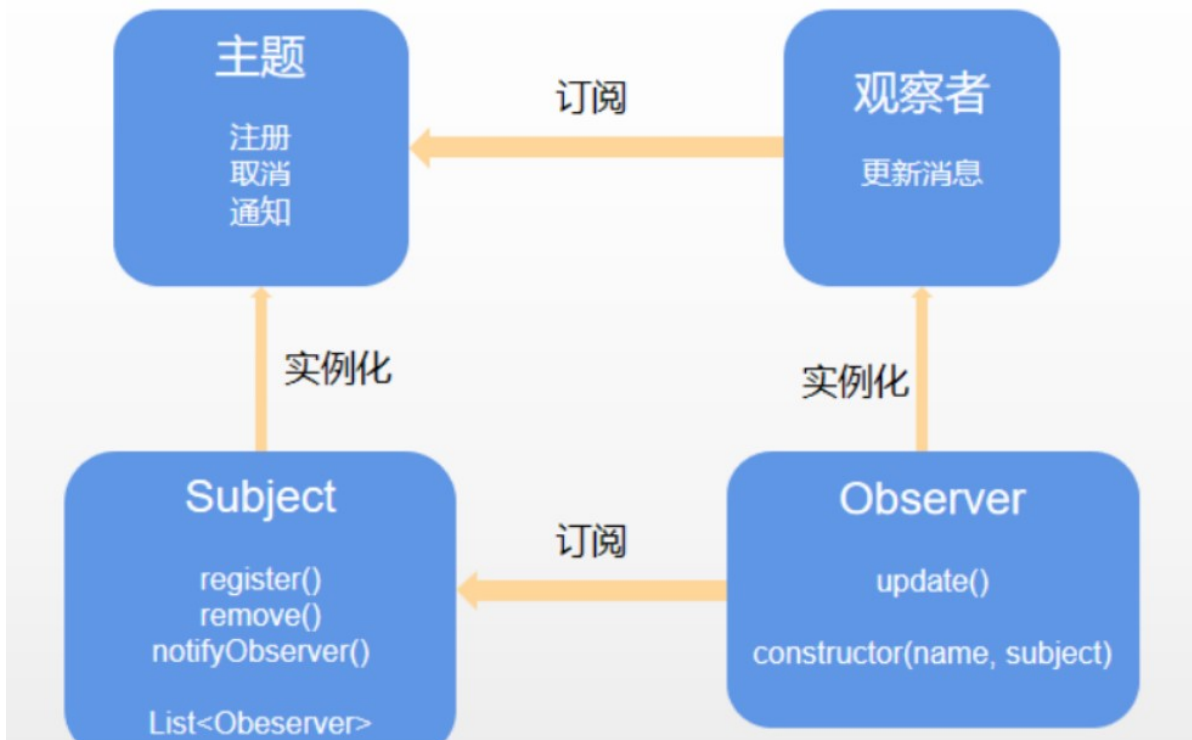


```
内容改变5
-----
具体观察者1做出响应
具体观察者2做出响应

Process finished with exit code 0
```

主题 (Subject) 具有注册和移除观察者、并通知所有观察者的功能, 主题是通过维护一张观察者列表来实现这些操作的。

观察者 (Observer) 的注册功能需要调用主题的 registerObserver() 方法。



在观察者模式，我们又分为**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。

1.定义一个被观察者的实现类WeatherData。

```
1 public class WeatherData extends Observable {
2
3     private int temperature;//温度
4
5     public int getTemperature() {
6         return temperature;
7     }
8
9     public void setTemperature(int temperature) {
10         this.temperature = temperature;
11         this.setChanged();
12         this.notifyObservers(temperature);
13     }
14 }
```

notifyObservers源码：

```
1 public void notifyObservers(Object arg) {
2     Object[] arrLocal;
3
4     synchronized (this) {
5         //如果changed为
6         if (!changed)
```

```

7         return;
8         arrLocal = obs.toArray();
9         clearChanged();
10    }
11
12    for (int i = arrLocal.length-1; i>=0; i--)
13        ((Observer)arrLocal[i]).update(this, arg);
14    }

```

## 2.定义一个观察者的实现类CurrentConditionsDisplay

```

1    public class CurrentConditionsDisplay extends Observable implements Observer {
2        @Override
3        public void update(Observable o, Object arg) {
4            this.displayTemperature(Integer.parseInt(arg.toString()));
5        }
6
7        private String status;
8
9        public String getStatus() {
10            return status;
11        }
12
13        public void setStatus(String status) {
14            this.status = status;
15        }
16
17        private void displayTemperature(int temperature) {
18            if (temperature <= 18) {
19                this.setStatus("要冻死了，还是窝在家里好");
20            }else if(temperature > 18 && temperature < 30){
21                this.setStatus("风和日丽，适合出去浪");
22            }else{
23                this.setStatus("太晒了，都给我晒糊了");
24            }
25            System.out.println("状态:  " + status + "  现在温度:  " + temperature);
26        }
27    }

```

接下来我们定义我们的测试类PullDemo：

```

1    public class PullDemo {
2
3        public static void main(String[] args) {
4            WeatherData weatherData = new WeatherData();
5            //添加一个观察者
6            weatherData.addObserver(new CurrentConditionsDisplay());
7            //天气变化
8            weatherData.setTemperature(20);
9        }
10    }
11

```

运行结果：

```
"D:\Program Files\Java\jdk1.8.0_91\bin\java" ...  
状态： 风和日丽，适合出去浪  现在温度： 20
```

观察者关注被观察者（上文说的主题对象），当被观察的状态发生变化，温度发生变化，会通知被观察对象CurrentConditionsDisplay 来展示被观察者的状态变化的值。

## 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

### 1.定义一个被观察者的实现类WeatherData。

```
1  public class WeatherData extends Observable {  
2  
3      private int temperature;//温度  
4  
5      public int getTemperature() {  
6          return temperature;  
7      }  
8  
9      public void setTemperature(int temperature) {  
10         this.temperature = temperature;  
11         this.setChanged();  
12         this.notifyObservers();  
13     }  
14 }
```

### 2.定义一个观察者的实现类CurrentConditionsDisplay

```
1  public class CurrentConditionsDisplay extends Observable implements Observer {  
2      @Override  
3      public void update(Observable o, Object arg) {  
4          this.displayTemperature(((WeatherData) o).getTemperature());  
5      }  
6  
7      private String status;  
8  
9      public String getStatus() {  
10         return status;  
11     }  
12  
13     public void setStatus(String status) {  
14         this.status = status;  
15     }  
16  
17     private void displayTemperature(int temperature) {  
18         if (temperature <= 18) {  
19             this.setStatus("要冻死了，还是窝在家里好");  
20         }else if(temperature > 18 && temperature < 30){
```

```

21         this.setStatus("风和日丽，适合出去浪");
22     }else{
23         this.setStatus("太晒了，都给我晒糊了");
24     }
25     System.out.println("状态:  " + status + "  现在温度:  " + temperature);
26 }
27 }

```

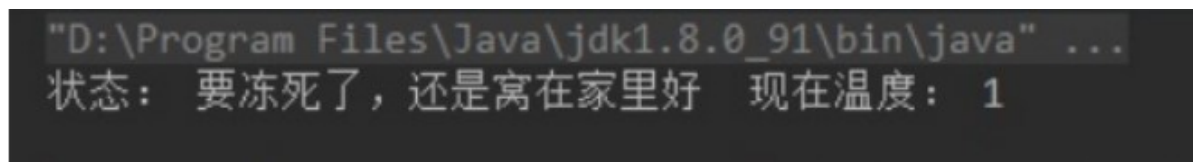
测试类PullDemo：

```

1  public class PullDemo {
2
3      public static void main(String[] args) {
4          WeatherData weatherData = new WeatherData();
5          //添加一个观察者
6          weatherData.addObserver(new CurrentConditionsDisplay());
7          //天气变化
8          weatherData.setTemperature(1);
9      }
10 }
11

```

测试结果：



```

"D:\Program Files\Java\jdk1.8.0_91\bin\java" ...
状态:  要冻死了，还是窝在家里好  现在温度:  1

```

会通知被观察对象CurrentConditionsDisplay 来展示被观察者的状态变化的值。

## 注意事项

1. JAVA 中已经有了对观察者模式的支持类（不重复造轮子）
2. 避免循环引用
3. 如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式

注：异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。