课程目标

1、观察者模式在源码中的应用及实现原理。

2、观察者模式的优、缺点。

内容定位

1、有 Swing 开发经验的人群更容易理解观察者模式。

观察者模式

观察者模式（Observer Pattern），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。定义一种一对多的依赖关系，一个主题对象可被多个观察者对象同时监听，使得每当主题对象状态变化时，所有依赖于它的对象都会得到通知并被自动更新。属于行为型模式。

```
原文：Defines a one-to-many dependency relationship between objects so that each time an object's state changes, its dependent objects are notified and automatically updated.
```

观察者模式的核心是将观察者与被观察者解耦，以类似于消息/广播发送的机制联动两者，使被观察者的变动能通知到感兴趣的观察者们，从而做出相应的响应。

观察者模式的应用场景

观察者模式在现实生活应用也非常广泛，比如：起床闹钟设置、APP 角标通知、GPer 生态圈消息通知、邮件通知、广播通知、桌面程序的事件响应等（如下图）。

APP 角标通知

起床闹钟设置

在软件系统中，当系统一方行为依赖于另一方行为的变动时，可使用观察者模式松耦合联动双方，使得一方的变动可以通知到感兴趣的另一方对象，从而让另一方对象对此做出响应。观察者模式适用于以下几种应用场景：

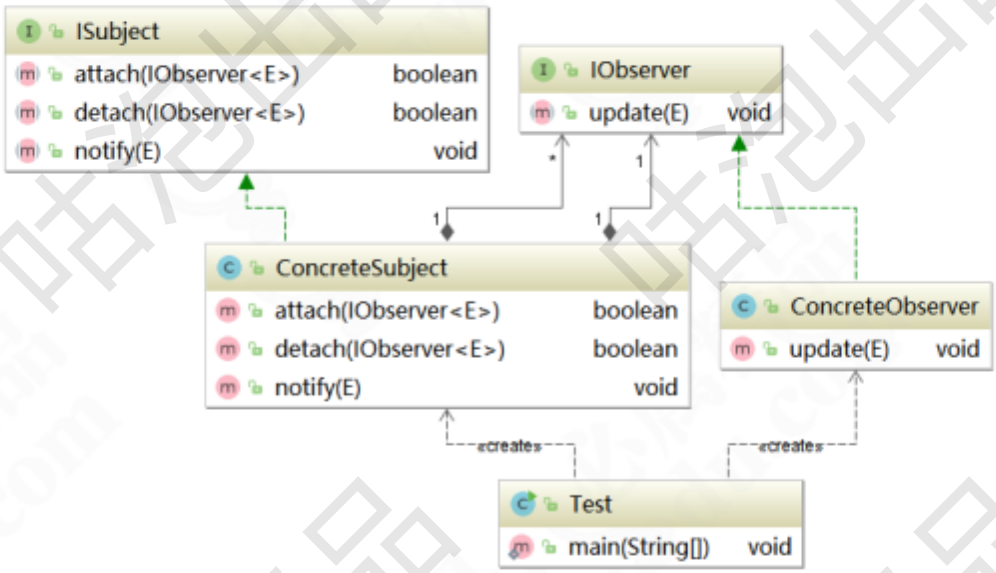
1、当一个抽象模型包含两个方面内容，其中一个方面依赖于另一个方面；

2、其他一个或多个对象的变化依赖于另一个对象的变化；

3、实现类似广播机制的功能，无需知道具体收听者，只需分发广播，系统中感兴趣的对象会自动接收该广播；

4、多层级嵌套使用，形成一种链式触发机制，使得事件具备跨域（跨越两种观察者类型）通知。

下面来看下观察者模式的通用 UML 类图：



从 UML 类图中，我们可以看到，观察者模式主要包含三种角色：

抽象主题（Subject）：指被观察的对象（Observable）。该角色是一个抽象类或接口，定义了增

加、删除、通知观察者对象的方法；

具体主题（ConcreteSubject）：具体被观察者，当其内状态变化时，会通知已注册的观察者；

抽象观察者（Observer）：定义了响应通知的更新方法；

具体观察者（ConcrereObserver）：在得到状态更新时，会自动做出响应。

观察者模式在业务场景中的应用

当小伙伴们在 GPer 生态圈中提问的时候，如果有设置指定老师回答，对应的老师就会收到邮件通知，这就是观察者模式的一种应用场景。我们有些小伙伴可能会想到 MQ，异步队列等，其实 JDK 本身就提供这样的 API。我们用代码来还原一下这样一个应用场景，创建 GPer 类：

```
/**
 * JDK 提供的一种观察者的实现方式，被观察者
 * Created by Tom.
 */
public class GPer extends Observable{
    private String name = "GPer 生态圈";
    private static GPer gper = null;
    private GPer(){}

    public static GPer getInstance(){
        if(null == gper){
            gper = new GPer();
        }
        return gper;
    }
    public String getName() {
        return name;
    }
    public void publishQuestion(Question question){
        System.out.println(question.getUserName() + "在" + this.name + "上提交了一个问题。");
        setChanged();
        notifyObservers(question);
    }
}
```

创建问题 Question 类：

```
/**
 * Created by Tom
 */
public class Question {
    private String userName;
    private String content;

    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
```

```
        this.userName = userName;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
}
```

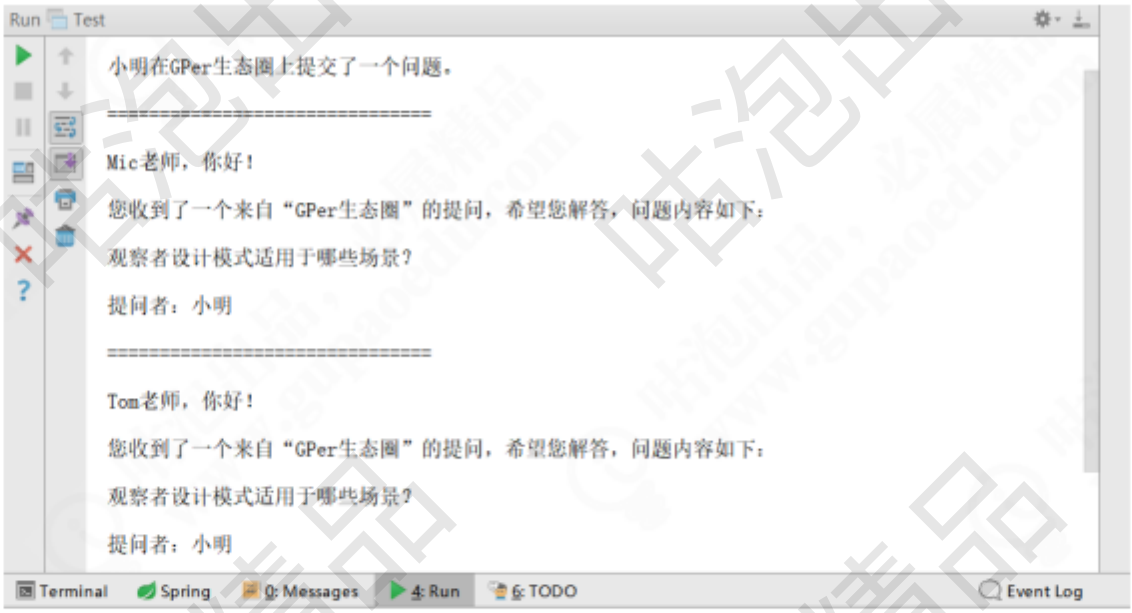
创建老师 Teacher 类：

```
/**
 * 观察者
 * Created by Tom.
 */
public class Teacher implements Observer {
    private String name;
    public Teacher(String name){
        this.name = name;
    }
    public void update(Observable o, Object arg) {
        GPer gper = (GPer)o;
        Question question = (Question)arg;
        System.out.println("==============================");
        System.out.println(name + "老师，你好！\n" +
        "您收到了一个来自“" + gper.getName() + "”的提问，希望您解答，问题内容如下：\n" +
        question.getContent() + "\n" +
        "提问者：" + question.getUserName());
    }
}
```

客户端测试代码：

```
/**
 * Created by Tom
 */
public class Test {
    public static void main(String[] args) {

        GPer gper = GPer.getInstance();
        Teacher tom = new Teacher("Tom");
        Teacher mic = new Teacher("Mic");

        gper.addObserver(tom);
        gper.addObserver(mic);

        //业务逻辑代码
        Question question = new Question();
        question.setUserName("小明");
        question.setContent("观察者模式适用于哪些场景？");

        gper.publishQuestion(gper,question);
    }
}
```

运行结果：



基于 Guava API 轻松落地观察者模式

给大家推荐一个实现观察者模式非常好用的框架。API 使用也非常简单，举个例子，先引入 maven 依赖包：

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>20.0</version>
</dependency>
```

创建侦听事件 GuavaEvent：

```
/**
 * Created by Tom
 */
public class GuavaEvent {
    @Subscribe
    public void subscribe(String str){
        //业务逻辑
        System.out.println("执行 subscribe 方法,传入的参数是:" + str);
    }

}
```

客户端测试代码：

```
/**
 * Created by Tom
 */
public class GuavaEventTest {
    public static void main(String[] args) {
        EventBus eventbus = new EventBus();
        GuavaEvent guavaEvent = new GuavaEvent();
```

```
        eventbus.register(guavaEvent);
        eventbus.post("Tom");
    }

}
```

使用观察者模式设计鼠标事件响应 API

下面再来设计一个业务场景，帮助小伙伴更好的理解观察者模式。JDK 源码中，观察者模式也应用非常多。例如 java.awt.Event 就是观察者模式的一种，只不过 Java 很少被用来写桌面程序。我们自己用代码来实现一下，以帮助小伙伴们更深刻地了解观察者模式的实现原理。首先，创建 EventListener 接口：

```
/**
 * 观察者抽象
 * Created by Tom.
 */
public interface EventListener {

}
```

创建 Event 类：

```
/**
 * 标准事件源格式的定义
 * Created by Tom.
 */
public class Event {
    //事件源，动作是由谁发出的
    private Object source;
    //事件触发，要通知谁（观察者）
    private EventListener target;
    //观察者给的回应
    private Method callback;
    //事件的名称
    private String trigger;
    //事件的触发事件
    private long time;

    public Event(EventListener target, Method callback) {
        this.target = target;
        this.callback = callback;
    }

    public Object getSource() {
        return source;
    }

    public Event setSource(Object source) {
        this.source = source;
        return this;
    }

    public String getTrigger() {
        return trigger;
    }
```

```
    public Event setTrigger(String trigger) {
        this.trigger = trigger;
        return this;
    }

    public long getTime() {
        return time;
    }

    public Event setTime(long time) {
        this.time = time;
        return this;
    }

    public Method getCallback() {
        return callback;
    }

    public EventListener getTarget() {
        return target;
    }

    @Override
    public String toString() {
        return "Event{" +
                "source=" + source +
                ", target=" + target +
                ", callback=" + callback +
                ", trigger='" + trigger + '\'' +
                ", time=" + time +
                '}';
    }
}
```

创建 EventContext 类：

```
/**
 * 被观察者的抽象
 * Created by Tom.
 */
public abstract class EventContext {
    protected Map<String,Event> events = new HashMap<String,Event>();

    public void addListener(String eventType, EventListener target, Method callback){
        events.put(eventType,new Event(target,callback));
    }

    public void addListener(String eventType, EventListener target){
        try {
            this.addListener(eventType, target,
                                    target.getClass().getMethod("on"+toUpperFirstCase(eventType), Event.class));
        }catch (NoSuchMethodException e){
            return;
        }
    }

    private String toUpperFirstCase(String eventType) {
        char [] chars = eventType.toCharArray();
        chars[0] -= 32;
        return String.valueOf(chars);
    }
```

```
    private void trigger(Event event){
        event.setSource(this);
        event.setTime(System.currentTimeMillis());

        try {
            if (event.getCallback() != null) {
                //用反射调用回调函数
                event.getCallback().invoke(event.getTarget(), event);
            }
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    protected void trigger(String trigger){
        if(!this.events.containsKey(trigger)){return;}
        trigger(this.events.get(trigger).setTrigger(trigger));
    }
}
```

创建 MouseEventType 接口：

```
/**
 * Created by Tom.
 */
public interface MouseEventType {
    //单击
    String ON_CLICK = "click";

    //双击
    String ON_DOUBLE_CLICK = "doubleClick";

    //弹起
    String ON_UP = "up";

    //按下
    String ON_DOWN = "down";

    //移动
    String ON_MOVE = "move";

    //滚动
    String ON_WHEEL = "wheel";

    //悬停
    String ON_OVER = "over";

    //失焦
    String ON_BLUR = "blur";

    //获焦
    String ON_FOCUS = "focus";
}
```

创建 Mouse 类：

```
/**
 * 具体的被观察者
 * Created by Tom.
 */
```

```
public class Mouse extends EventContext {

    public void click(){
        System.out.println("调用单击方法");
        this.trigger(MouseEventType.ON_CLICK);
    }

    public void doubleClick(){
        System.out.println("调用双击方法");
        this.trigger(MouseEventType.ON_DOUBLE_CLICK);
    }

    public void up(){
        System.out.println("调用弹起方法");
        this.trigger(MouseEventType.ON_UP);
    }

    public void down(){
        System.out.println("调用按下方法");
        this.trigger(MouseEventType.ON_DOWN);
    }

    public void move(){
        System.out.println("调用移动方法");
        this.trigger(MouseEventType.ON_MOVE);
    }

    public void wheel(){
        System.out.println("调用滚动方法");
        this.trigger(MouseEventType.ON_WHEEL);
    }

    public void over(){
        System.out.println("调用悬停方法");
        this.trigger(MouseEventType.ON_OVER);
    }

    public void blur(){
        System.out.println("调用获焦方法");
        this.trigger(MouseEventType.ON_BLUR);
    }

    public void focus(){
        System.out.println("调用失焦方法");
        this.trigger(MouseEventType.ON_FOCUS);
    }
}
```

创建回调方法 MouseEventLisenter 类：

```
/**
 * 观察者
 * Created by Tom.
 */
public class MouseEventListener implements EventListener {


    public void onClick(Event e){
        System.out.println("==========触发鼠标单击事件==========" + "\n" + e);
    }

    public void onDoubleClick(Event e){
```

```
        System.out.println("==========触发鼠标双击事件==========" + "\n" + e);
    }

    public void onUp(Event e){
        System.out.println("==========触发鼠标弹起事件==========" + "\n" + e);
    }

    public void onDown(Event e){
        System.out.println("==========触发鼠标按下事件==========" + "\n" + e);
    }

    public void onMove(Event e){
        System.out.println("==========触发鼠标移动事件==========" + "\n" + e);
    }

    public void onWheel(Event e){
        System.out.println("==========触发鼠标滚动事件==========" + "\n" + e);
    }

    public void onOver(Event e){
        System.out.println("==========触发鼠标悬停事件==========" + "\n" + e);
    }

    public void onBlur(Event e){
        System.out.println("==========触发鼠标失焦事件==========" + "\n" + e);
    }

    public void onFocus(Event e){
        System.out.println("==========触发鼠标获焦事件==========" + "\n" + e);
    }

}
```

客户端测试代码：

```
    public static void main(String[] args) {
        EventListener listener = new MouseEventListener();

        Mouse mouse = new Mouse();
        mouse.addListener(MouseEventType.ON_CLICK,listener);
        mouse.addListener(MouseEventType.ON_MOVE,listener);

        mouse.click();
        mouse.move();
    }
```

观察者模式在源码中的应用

Spring 中的 ContextLoaderListener 实现了 ServletContextListener 接口，ServletContextListener 接口又继承了 EventListener，在 JDK 中 EventListener 有非常广泛的应用。

我们可以看一下源代码，ContextLoaderListener：

```
package org.springframework.web.context;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ContextLoaderListener extends ContextLoader implements ServletContextListener {
```

```
    public ContextLoaderListener() {
    }

    public ContextLoaderListener(WebApplicationContext context) {
        super(context);
    }

    public void contextInitialized(ServletContextEvent event) {
        this.initWebApplicationContext(event.getServletContext());
    }

    public void contextDestroyed(ServletContextEvent event) {
        this.closeWebApplicationContext(event.getServletContext());
        ContextCleanupListener.cleanupAttributes(event.getServletContext());
    }
}
```

ServletContextListener 接口源码如下：

```
package javax.servlet;
import java.util.EventListener;
public interface ServletContextListener extends EventListener {
    public void contextInitialized(ServletContextEvent sce);
    public void contextDestroyed(ServletContextEvent sce);
}
```

EventListener 接口源码如下：

```
package java.util;
public interface EventListener {
}
```

观察者模式的优缺点

优点：

1、观察者和被观察者是松耦合（抽象耦合）的，符合依赖倒置原则；

2、分离了表示层（观察者）和数据逻辑层（被观察者），并且建立了一套触发机制，使得数据的变化可以响应到多个表示层上；

3、实现了一对多的通讯机制，支持事件注册机制，支持兴趣分发机制，当被观察者触发事件时，只有感兴趣的观察者可以接收到通知。

缺点：

1、如果观察者数量过多，则事件通知会耗时较长；

2、事件通知呈线性关系，如果其中一个观察者处理事件卡壳，会影响后续的观察者接收该事件；

3、如果观察者和被观察者之间存在循环依赖，则可能造成两者之间的循环调用，导致系统崩溃。