

课程目标

- 1、掌握装饰者模式的特征和应用场景
- 2、理解装饰者模式和适配器模式的根本区别。
- 3、观察者模式在源码中的应用及实现原理。
- 4、了解装饰者模式和观察者模式的优、缺点。

内容定位

- 1、有重构项目需求的人群一定要掌握装饰者模式。
- 2、有 Swing 开发经验的人群更容易理解观察者模式。

装饰者模式

装饰者模式的应用场景

装饰者模式 (Decorator Pattern) 是指在不改变原有对象的基础之上，将功能附加到对象上，提供了比继承更有弹性的替代方案（扩展原有对象的功能），属于结构型模式。

装饰者模式在我们生活中应用也比较多如给煎饼加鸡蛋；给蛋糕加上一些水果；给房子装修等，为对象扩展一些额外的职责。装饰者在代码程序中适用于以下场景：

- 1、用于扩展一个类的功能或给一个类添加附加职责。
- 2、动态的给一个对象添加功能，这些功能可以再动态的撤销。

来看一个这样的场景，上班族白领其实大多有睡懒觉的习惯，每天早上上班都是踩点，

于是很多小伙伴为了多赖一会儿床都不吃早餐。那么，也有些小伙伴可能在上班路上碰到卖煎饼的路边摊，都会顺带一个到公司茶水间吃早餐。卖煎饼的大姐可以给你的煎饼加鸡蛋，也可以加香肠（如下图，PS：我买煎饼一般都要求不加生菜）。



煎饼侠（大鹏）



水果蛋糕

下面我们用代码还原一下码农的生活。首先创建一个煎饼 Battercake 类:

```
package com.gupaoedu.vip.pattern.decorator.battercake.v1;

/**
 * Created by Tom.
 */
public class Battercake {

    protected String getMsg(){
        return "煎饼";
    }

    public int getPrice(){
        return 5;
    }

}
```

创建一个加鸡蛋的煎饼 BattercakeWithEgg 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v1;

/**
 * Created by Tom.
 */
public class BattercakeWithEgg extends Battercake{
    @Override
```

```

protected String getMsg() {
    return super.getMsg() + "+1 个鸡蛋";
}

@Override
//加一个鸡蛋加 1 块钱
public int getPrice() {
    return super.getPrice() + 1;
}
}

```

再创建一个既加鸡蛋又加香肠的 BattercakeWithEggAndSausage 类：

```

package com.gupaoedu.vip.pattern.decorator.battercake.v1;

/**
 * Created by Tom.
 */
public class BattercakeWithEggAndSausage extends BattercakeWithEgg{
    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 根香肠";
    }

    @Override
    //加一个香肠加 2 块钱
    public int getPrice() {
        return super.getPrice() + 2;
    }
}

```

编写客户端测试代码：

```

package com.gupaoedu.vip.pattern.decorator.battercake.v1;

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

        Battercake battercake = new Battercake();
        System.out.println(battercake.getMsg() + ",总价格: " + battercake.getPrice());

        Battercake battercakeWithEgg = new BattercakeWithEgg();
        System.out.println(battercakeWithEgg.getMsg() + ",总价格: " + battercakeWithEgg.getPrice());
    }
}

```

```

        Battercake battercakeWithEggAndSausage = new BattercakeWithEggAndSausage();
        System.out.println(battercakeWithEggAndSausage.getMsg() + ",总价格: " +
battercakeWithEggAndSausage.getPrice());

    }
}

```

运行结果：

```

煎饼，总价格:5
煎饼+1个鸡蛋，总价格:6
煎饼+1个鸡蛋+1根香肠，总价格:8

Process finished with exit code 0

```

运行结果没有问题。但是，如果用户需要一个加 2 个鸡蛋加 1 根香肠的煎饼，那么用我们现在的类结构是创建不出来的，也无法自动计算出价格，除非再创建一个类做定制。如果需求再变，一直加定制显然是不科学的。那么下面我们就用装饰者模式来解决上面的问题。首先创建一个建煎饼的抽象 Battercake 类：

```

package com.gupaoedu.vip.pattern.decorator.battercake.v2;

/**
 * Created by Tom.
 */
public abstract class Battercake {
    protected abstract String getMsg();
    protected abstract int getPrice();
}

```

创建一个基本的煎饼（或者叫基础套餐）BaseBattercake：

```

package com.gupaoedu.vip.pattern.decorator.battercake.v2;

/**
 * Created by Tom.
 */
public class BaseBattercake extends Battercake {
    protected String getMsg(){
        return "煎饼";
    }
}

```

```
public int getPrice(){ return 5; }
}
```

然后，再创建一个扩展套餐的抽象装饰者 BattercakeDecotator 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public abstract class BattercakeDecorator extends Battercake {
    //静态代理，委派
    private Battercake battercake;

    public BattercakeDecorator(Battercake battercake) {
        this.battercake = battercake;
    }
    protected abstract void doSomething();

    @Override
    protected String getMsg() {
        return this.battercake.getMsg();
    }
    @Override
    protected int getPrice() {
        return this.battercake.getPrice();
    }
}
```

然后，创建鸡蛋装饰者 EggDecorator 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class EggDecorator extends BattercakeDecorator {
    public EggDecorator(Battercake battercake) {
        super(battercake);
    }

    protected void doSomething() {}

    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 个鸡蛋";
    }
}
```

```

@Override
protected int getPrice() {
    return super.getPrice() + 1;
}
}

```

创建香肠装饰者 SausageDecorator 类：

```

package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class SausageDecorator extends BattercakeDecorator {
    public SausageDecorator(Battercake battercake) {
        super(battercake);
    }

    protected void doSomething() {}

    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 根香肠";
    }

    @Override
    protected int getPrice() {
        return super.getPrice() + 2;
    }
}

```

编写客户端测试代码：

```

package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class BattercakeTest {
    public static void main(String[] args) {
        Battercake battercake;
        //路边摊买一个煎饼
        battercake = new BaseBattercake();
        //煎饼有点小，想再加一个鸡蛋
        battercake = new EggDecorator(battercake);
        //再加一个鸡蛋
        battercake = new EggDecorator(battercake);
        //很饿，再加根香肠
    }
}

```

```

battercake = new SausageDecorator(battercake);

//跟静态代理最大区别就是职责不同
//静态代理不一定要满足 is-a 的关系
//静态代理会做功能增强，同一个职责变得不一样

//装饰器更多考虑是扩展
System.out.println(battercake.getMsg() + ",总价: " + battercake.getPrice());
}
}

```

运行结果：

```

煎饼+1个鸡蛋+1个鸡蛋+1根香肠总价:9

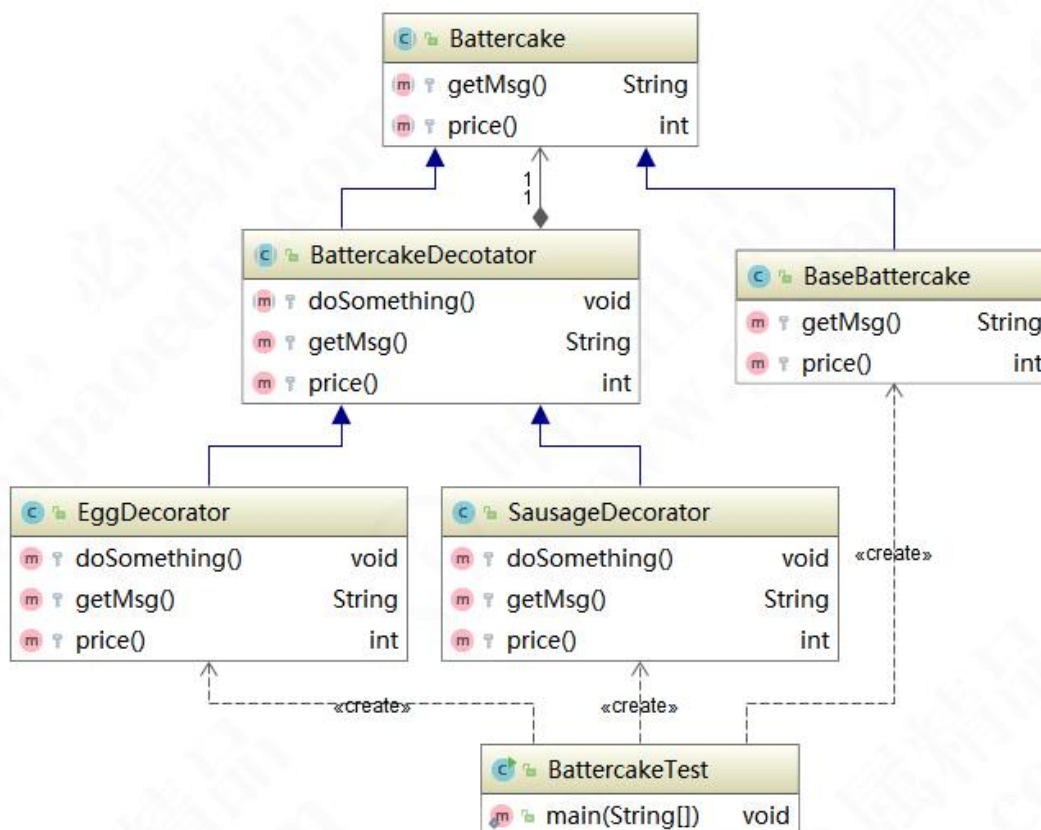
```

```

Process finished with exit code 0

```

来看一下类图：



为了加深印象，我们再来看一个应用场景。是否还有小伙伴记得我们上次讲个的适配器模式，为了实现新功能与老功能兼容，创建一个新的类继承已有的类，实现功能扩展，

遵循开闭原则。今天我们再用装饰者模式再来升级一次代码，同时也做一个更好的对比。

先看原来的代码，Member 类：

```
package com.gupaoedu.vip.pattern.decorator.passport.old;

/**
 * Created by Tom.
 */
public class Member {

    private String username;
    private String password;
    private String mid;
    private String info;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getMid() {
        return mid;
    }

    public void setMid(String mid) {
        this.mid = mid;
    }

    public String getInfo() {
        return info;
    }
}
```



```
public void setInfo(String info) {  
    this.info = info;  
}  
}
```

ResultMsg 类：

```
package com.gupaoedu.vip.pattern.decorator.passport.old;  
  
/**  
 * Created by Tom.  
 */  
public class ResultMsg {  
  
    private int code;  
    private String msg;  
    private Object data;  
  
    public ResultMsg(int code, String msg, Object data) {  
        this.code = code;  
        this.msg = msg;  
        this.data = data;  
    }  
  
    public int getCode() {  
        return code;  
    }  
  
    public void setCode(int code) {  
        this.code = code;  
    }  
  
    public String getMsg() {  
        return msg;  
    }  
  
    public void setMsg(String msg) {  
        this.msg = msg;  
    }  
  
    public Object getData() {  
        return data;  
    }  
  
    public void setData(Object data) {
```

```

        this.data = data;
    }
}

```

ISigninService 接口：

```

package com.gupaoedu.vip.pattern.decorator.passport.old;

/**
 * Created by Tom.
 */
public interface ISigninService {
    ResultMsg regist(String username, String password);
    /**
     * 登录的方法
     * @param username
     * @param password
     * @return
     */
    ResultMsg login(String username, String password);
}

```

SigninService 实现类：

```

package com.gupaoedu.vip.pattern.decorator.passport.old;

/**
 * Created by Tom.
 */
public class SigninService implements ISigninService {

    public ResultMsg regist(String username, String password){
        return new ResultMsg(200, "注册成功", new Member());
    }

    /**
     * 登录的方法
     * @param username
     * @param password
     * @return
     */
    public ResultMsg login(String username, String password){
        return null;
    }
}

```

来看升级以后的代码，创建一个新的接口继承原来的接口：

```
package com.gupaoedu.vip.pattern.decorator.passport.upgrade;

import com.gupaoedu.vip.pattern.decorator.passport.old.ISigninService;
import com.gupaoedu.vip.pattern.decorator.passport.old.ResultMsg;

/**
 * Created by Tom.
 */
public interface ISigninForThirdService extends ISigninService {

    /**
     * QQ 登录
     * @param id
     * @return
     */
    ResultMsg loginForQQ(String id);

    /**
     * 微信登录
     * @param id
     * @return
     */
    ResultMsg loginForWechat(String id);

    /**
     * 记住登录状态后自动登录
     * @param token
     * @return
     */
    ResultMsg loginForToken(String token);

    /**
     * 手机号登录
     * @param telephone
     * @param code
     * @return
     */
    ResultMsg loginForTelephone(String telephone, String code);

    /**
     * 注册后自动登录
     * @param username
     * @param passport
     * @return
     */
}
```

```

    */
    ResultMsg loginForRegist(String username, String passport);
}

```

创建新的逻辑处理类 SigninForThirdService，实现新创建的接口：

```

package com.gupaoedu.vip.pattern.decorator.passport.upgrade;

import com.gupaoedu.vip.pattern.decorator.passport.old.ISigninService;
import com.gupaoedu.vip.pattern.decorator.passport.old.ResultMsg;

/**
 * 第三方登录自由适配
 * Created by Tom.
 */
public class SigninForThirdService implements ISigninForThirdService {

    private ISigninService signin;

    public SigninForThirdService(ISigninService iSignin){
        this.signin = signin;
    }

    public ResultMsg regist(String username, String passport){
        return signin.regist(username,passport);
    }

    public ResultMsg login(String username,String passport){
        return signin.login(username,passport);
    }

    public ResultMsg loginForQQ(String id) {
        return null;
    }

    public ResultMsg loginForWechat(String id) {
        return null;
    }

    public ResultMsg loginForToken(String token) {
        return null;
    }
}

```

```

public ResultMsg loginForTelephone(String telephone, String code) {
    return null;
}

public ResultMsg loginForRegist(String username, String passport) {
    return null;
}
}

```

客户端测试代码：

```

package com.gupaoedu.vip.pattern.decorator.passport;

import com.gupaoedu.vip.pattern.decorator.passport.old.SigninService;
import com.gupaoedu.vip.pattern.decorator.passport.upgrade.ISigninForThirdService;
import com.gupaoedu.vip.pattern.decorator.passport.upgrade.SigninForThirdService;

/**
 * Created by Tom.
 */
public class DecoratorTest {

    public static void main(String[] args) {

        ISigninForThirdService signinForThirdService = new SigninForThirdService(new
        SigninService());

        signinForThirdService.loginForQQ("xdcdfswrwsdfssdfqsdf");

        //为某个类实现动态增加或者覆盖原有方法的情况，采用装饰器模式

    }

}

```

装饰者模式最本质的特征是讲原有类的附加功能抽离出来，简化原有类的逻辑。通过这样两个案例，我们可以总结出来，其实抽象的装饰者是可有可无的，具体可以根据业务模型来选择。

装饰者模式和适配器模式对比

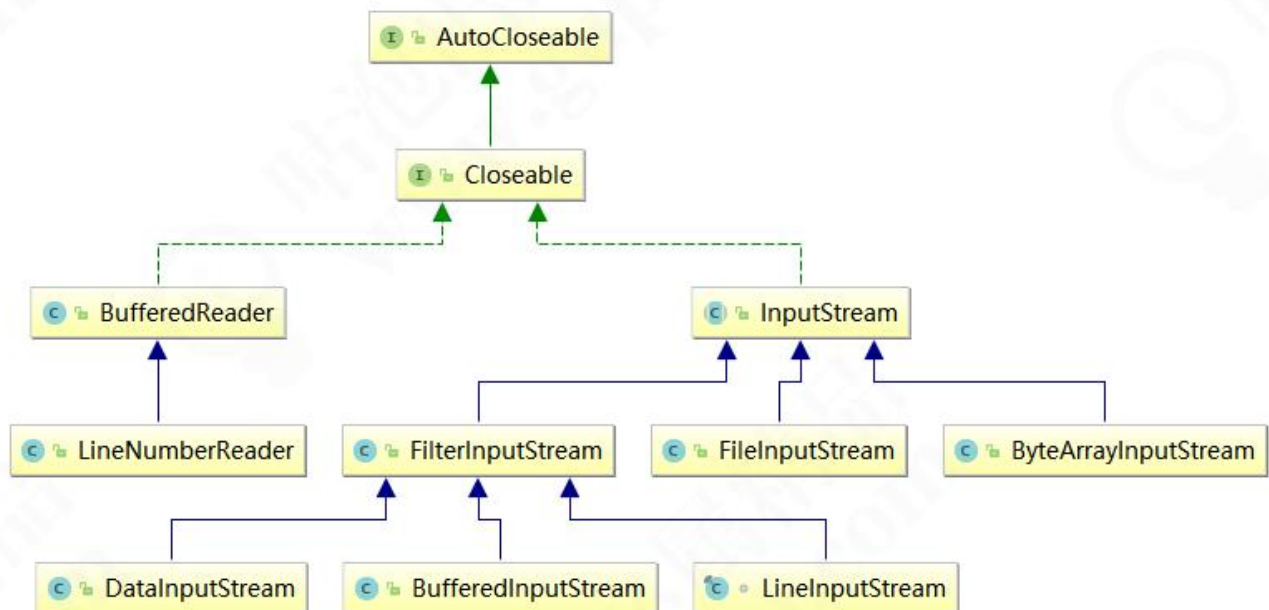
装饰者和适配器模式都是包装模式（Wrapper Pattern），装饰者也是一种特殊的代理模

式。

	装饰者模式	适配器模式
形式	是一种非常特别的适配器模式	没有层级关系 ,装饰器模式有层级关系
定义	装饰者和被装饰者都实现同一个接口，主要目的是为了扩展之后依旧保留 OOP 关系	适配器和被适配者没有必然的联系 ,通常是采用继承或代理的形式进行包装
关系	满足 is-a 的关系	满足 has-a 的关系
功能	注重覆盖、扩展	注重兼容、转换
设计	前置考虑	后置考虑

装饰者模式在源码中的应用

装饰器模式在源码中也应用得非常多，在 JDK 中体现最明显的类就是 IO 相关的类，如 `BufferedReader`、`InputStream`、`OutputStream`，看一下常用的 `InputStream` 的类结构图：



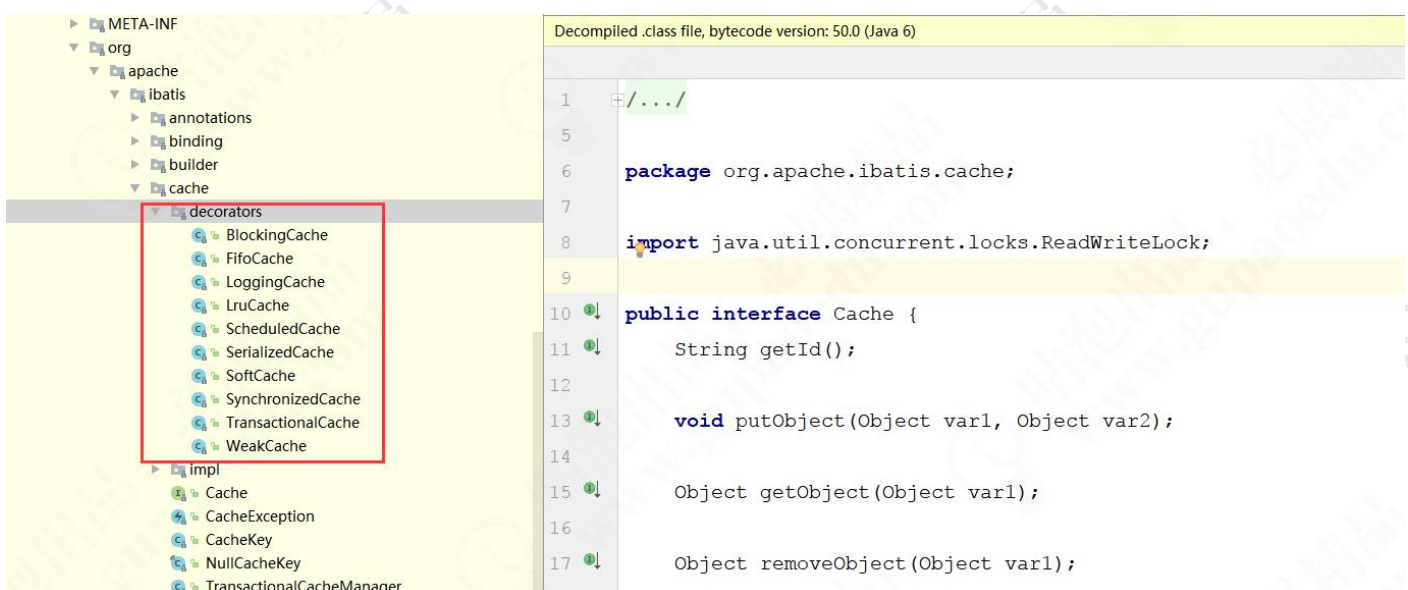
在 Spring 中的 TransactionAwareCacheDecorator 类我们也可以来尝试理解一下，这个类主要是用来处理事务缓存的，来看一下代码：

```
public class TransactionAwareCacheDecorator implements Cache {
    private final Cache targetCache;
    public TransactionAwareCacheDecorator(Cache targetCache) {
        Assert.notNull(targetCache, "Target Cache must not be null");
        this.targetCache = targetCache;
    }
    public Cache getTargetCache() {
        return this.targetCache;
    }
    ...
}
```

TransactionAwareCacheDecorator 就是对 Cache 的一个包装。再来看一个 MVC 中的装饰者模式 HttpHeadersDecorator 类:

```
public class HttpHeadersDecorator extends ServerHttpServletResponse {
    public HttpHeadersDecorator(ServerHttpServletResponse delegate) {
        super(delegate);
    }
    ...
}
```

最后，看看 MyBatis 中的一段处理缓存的设计 org.apache.ibatis.cache.Cache 类，找到它的包定位：



从名字上来看其实更容易理解了。比如 FifoCache 先入先出算法的缓存；LruCache 最近最少使用的缓存；TransactionlCache 事务相关的缓存，都是采用装饰者模式。MyBatis 源码在我们后续的课程也会深入讲解，感兴趣的小伙伴可以详细看看这块的源码，也可以好好学习一下 MyBatis 的命名方式，今天我们还是把重点放到设计模式上。

装饰者模式的优缺点

优点：

- 1、装饰者是继承的有力补充，比继承灵活，不改变原有对象的情况下动态地给一个对象扩展功能，即插即用。
- 2、通过使用不同装饰类以及这些装饰类的排列组合，可以实现不同效果。
- 3、装饰者完全遵守开闭原则。

缺点：

- 1、会出现更多的代码，更多的类，增加程序复杂性。
- 2、动态装饰时，多层装饰时会更复杂。

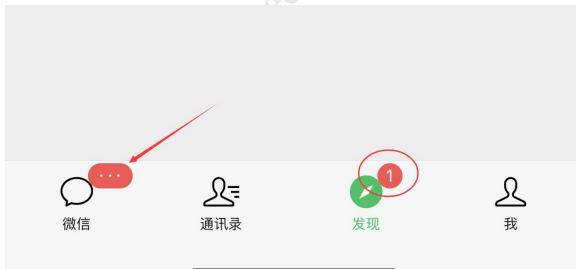
那么装饰者模式我们就讲解到这里，希望小伙伴们认真体会，加深理解。

观察者模式

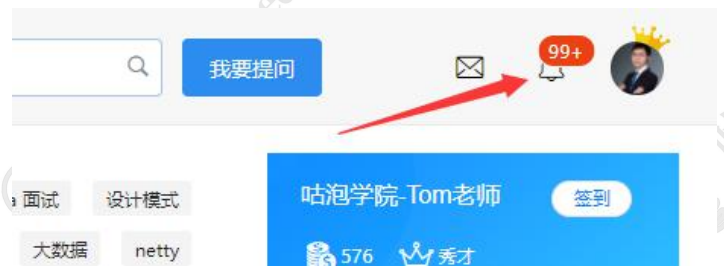
观察者模式的应用场景

观察者模式（Observer Pattern）定义了对象之间的一对多依赖，让多个观察者对象同时监听一个主体对象，当主体对象发生变化时，它的所有依赖者（观察者）都会收到通知并更新，属于行为型模式。观察者模式有时也叫做发布订阅模式。观察者模式主要用于在关联行为之间建立一套触发机制的场景。观察者模式在现实生活应用也非常广泛，比如：微信朋友圈动态通知、GPser 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）。



微信朋友圈通知



GPer 生态圈通知

现在大家都用过我们的 GPer 生态圈了，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：

```
package com.gupaoedu.vip.pattern.observer.gperadvice;
import java.util.Observable;
/**
 * 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：

```
package com.gupaoedu.vip.pattern.observer.gperadvice;
/**
 * 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：

```
package com.gupaoedu.vip.pattern.observer.gperadvice;
import java.util.Observable;
import java.util.Observer;
/**
 * 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：

```

package com.gupaoedu.vip.pattern.observer.gperadvice;
/**
 * Created by Tom
 */
public class ObserverTest {
    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：

```

小明在“GPer生态圈”上提交了一个问题。
=====
Mic老师, 你好!
您收到了一个来自“GPer生态圈”的提问, 希望您解答。问题如下:
观察者模式适用于哪些场景?
提问者: 小明
=====
Tom老师, 你好!
您收到了一个来自“GPer生态圈”的提问, 希望您解答。问题如下:
观察者模式适用于哪些场景?
提问者: 小明
=====
Process finished with exit code 0

```

在下面我们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 `Event` 类：

```
package com.gupaoedu.vip.pattern.observer.events.core;

import java.lang.reflect.Method;

/**
 * 监听器的一种包装, 标准事件源格式的定义
 * Created by Tom.
 */
public class Event {
    //事件源，事件是由谁发起的保存起来
    private Object source;
    //事件触发，要通知谁
    private Object target;
    //事件触发，要做什么动作，回调
    private Method callback;
    //事件的名称，触发的是什么事件
    private String trigger;
    //事件触发的时间
    private long time;

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

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

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

    public Object getSource() {
```

```

        return source;
    }

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

    public long getTime() {
        return time;
    }

    public Object getTarget() {
        return target;
    }

    public Method getCallback() {
        return callback;
    }

    @Override
    public String toString() {
        return "Event{" + "\n" +
            "\tsource=" + source.getClass() + ",\n" +
            "\ttarget=" + target.getClass() + ",\n" +
            "\tcallback=" + callback + ",\n" +
            "\ttrigger='" + trigger + "',\n" +
            "\ttime=" + time + "'\n" +
            '}';
    }
}

```

创建 EventLisenter 类：

```

package com.gupaoedu.vip.pattern.observer.events.core;

import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

/**
 * 监听器，它就是观察者的桥梁
 * Created by Tom.
 */
public class EventLisenter {

```

//JDK 底层的 Lisenter 通常也是这样来设计的

```
protected Map<String,Event> events = new HashMap<String,Event>();
```

//事件名称和一个目标对象来触发事件

```
public void addLisenter(String eventType,Object target){
    try {
        this.addLisenter(
            eventType,
            target,
            target.getClass().getMethod("on" + toUpperFirstCase(eventType),Event.class));
    }catch (Exception e){
        e.printStackTrace();
    }
}
```

```
public void addLisenter(String eventType,Object target,Method callback){
    //注册事件
    events.put(eventType, new Event(target, callback));
}
```

//触发，只要有动作就触发

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

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

//事件名称触发

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

```

//逻辑处理的私有方法，首字母大写
private String toUpperFirstCase(String str){
    char[] chars = str.toCharArray();
    chars[0] -= 32;
    return String.valueOf(chars);
}
}

```

创建 MouseEventType 接口：

```

package com.gupaoedu.vip.pattern.observer.events.mouseevent;

/**
 * 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：

```
package com.gupaoedu.vip.pattern.observer.events.mouseevent;

import com.gupaoedu.vip.pattern.observer.events.core.EventLisenter;

/**
 * Created by Tom.
 */
public class Mouse extends EventLisenter {

    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 MouseEventCallback 类：

```

package com.gupaoedu.vip.pattern.observer.events.mouseevent;

import com.gupaoedu.vip.pattern.observer.events.core.Event;

/**
 * Created by Tom.
 */
public class MouseEventCallback {

    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：

```

package com.gupaoedu.vip.pattern.observer.events;

import com.gupaoedu.vip.pattern.observer.events.mouseevent.Mouse;
import com.gupaoedu.vip.pattern.observer.events.mouseevent.MouseEventCallback;
import com.gupaoedu.vip.pattern.observer.events.mouseevent.MouseEventType;
/**
 * Created by Tom.
 */
public class MouseEventTest {
    public static void main(String[] args) {

        try {
            MouseEventCallback callback = new MouseEventCallback();

            //注册事件
            Mouse mouse = new Mouse();
            mouse.addListener(MouseEventType.ON_CLICK, callback);
            mouse.addListener(MouseEventType.ON_MOVE, callback);
            mouse.addListener(MouseEventType.ON_WHEEL, callback);
            mouse.addListener(MouseEventType.ON_OVER, callback);

            //调用方法
            mouse.click();

            //失焦事件
            mouse.blur();

        }catch (Exception e){

```

```

        e.printStackTrace();
    }
}
}

```

观察者模式在源码中的应用

来看一下 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：

```

package javax.servlet;
import java.util.EventListener;

public interface ServletContextListener extends EventListener {
    public void contextInitialized(ServletContextEvent sce);
    public void contextDestroyed(ServletContextEvent sce);
}

```

EventListener：

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

基于 Guava API 轻松落地观察者模式

在这里，我还推荐给大家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：

```
package com.gupaoedu.vip.pattern.observer.guava;

import com.google.common.eventbus.Subscribe;

/**
 * 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：

```
package com.gupaoedu.vip.pattern.observer.guava;

import com.google.common.eventbus.EventBus;

/**
 * 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");  
}  
}
```

观察者模式的优缺点

优点：

- 1、观察者和被观察者之间建立了一个抽象的耦合。
- 2、观察者模式支持广播通信。

缺点：

- 1、观察者之间有过多的细节依赖、提高时间消耗及程序的复杂度。
- 2、使用要得当，要避免循环调用。