

73-中介模式：什么时候用中介模式？什么时候用观察者模式？

今天，我们来学习23种经典设计模式中的最后一个，中介模式。跟前面刚刚讲过的命令模式、解释器模式类似，中介模式也属于不怎么常用的模式，应用场景比较特殊、有限，但是，跟它俩不同的是，中介模式理解起来并不难，代码实现也非常简单，学习难度要小很多。

如果你对中介模式有所了解，你可能会知道，中介模式跟之前讲过的观察者模式有点相似，所以，今天我们还会详细讨论下这两种模式的区别。

话不多说，让我们正式开始今天的学习吧！

中介模式的原理和实现

中介模式的英文翻译是Mediator Design Pattern。在GoF中的《设计模式》一书中，它是这样定义的：

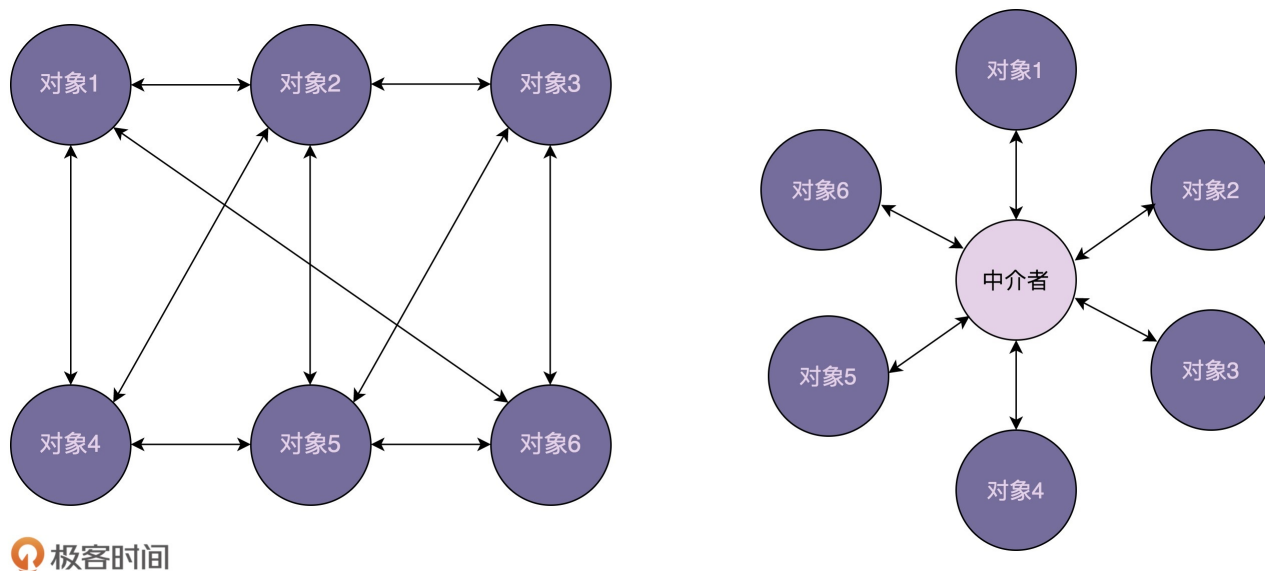
Mediator pattern defines a separate (mediator) object that encapsulates the interaction between a set of objects and the objects delegate their interaction to a mediator object instead of interacting with each other directly.

翻译成中文就是：中介模式定义了一个单独的（中介）对象，来封装一组对象之间的交互。将这组对象之间的交互委派给与中介对象交互，来避免对象之间的直接交互。

还记得我们在[第30节课](#)中讲的“如何给代码解耦”吗？其中一个方法就是引入中间层。

实际上，中介模式的设计思想跟中间层很像，通过引入中介这个中间层，将一组对象之间的交互关系（或者说依赖关系）从多对多（网状关系）转换为一对多（星状关系）。原来一个对象要跟n个对象交互，现在只需要跟一个中介对象交互，从而最小化对象之间的交互关系，降低了代码的复杂度，提高了代码的可读性和可维护性。

这里我画了一张对象交互关系的对比图。其中，右边的交互图是利用中介模式对左边交互关系优化之后的结果，从图中我们可以很直观地看出，右边的交互关系更加清晰、简洁。



提到中介模式，有一个比较经典的例子不得不说，那就是航空管制。

为了让飞机在飞行的时候互不干扰，每架飞机都需要知道其他飞机每时每刻的位置，这就需要时刻跟其他飞机通信。飞机通信形成的通信网络就会无比复杂。这个时候，我们通过引入“塔台”这样一个中介，让每架飞机只跟塔台来通信，发送自己的位置给塔台，由塔台来负责每架飞机的航线调度。这样就大大简化了通信网络。

刚刚举的是生活中的例子，我们再举一个跟编程开发相关的例子。这个例子与UI控件有关，算是中介模式比较经典的应用，很多书籍在讲到中介模式的时候，都会拿它来举例。

假设我们有一个比较复杂的对话框，对话框中有很多控件，比如按钮、文本框、下拉框等。当我们对某个控件进行操作的时候，其他控件会做出相应的反应，比如，我们在下拉框中选择“注册”，注册相关的控件就会显示在对话框中。如果我们在下拉框中选择“登陆”，登陆相关的控件就会显示在对话框中。

按照通常我们习惯的UI界面的开发方式，我们将刚刚的需求用代码实现出来，就是下面这个样子。在这种实现方式中，控件和控件之间互相操作、互相依赖。

```
public class UIControl {
    private static final String LOGIN_BTN_ID = "login_btn";
    private static final String REG_BTN_ID = "reg_btn";
    private static final String USERNAME_INPUT_ID = "username_input";
    private static final String PASSWORD_INPUT_ID = "pswd_input";
    private static final String REPEATED_PASSWORD_INPUT_ID = "repeated_pswd_input";
    private static final String HINT_TEXT_ID = "hint_text";
    private static final String SELECTION_ID = "selection";

    public static void main(String[] args) {
        Button loginButton = (Button)findViewById(LOGIN_BTN_ID);
        Button regButton = (Button)findViewById(REG_BTN_ID);
        Input usernameInput = (Input)findViewById(USERNAME_INPUT_ID);
        Input passwordInput = (Input)findViewById(PASSWORD_INPUT_ID);
        Input repeatedPswdInput = (Input)findViewById(REPEATED_PASSWORD_INPUT_ID);
        Text hintText = (Text)findViewById(HINT_TEXT_ID);
        Selection selection = (Selection)findViewById(SELECTION_ID);

        loginButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String username = usernameInput.text();
                String password = passwordInput.text();
                //校验数据...
                //做业务处理...
            }
        });

        regButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                //获取usernameInput、passwordInput、repeatedPswdInput数据...
                //校验数据...
                //做业务处理...
            }
        });

        //...省略selection下拉选择框相关代码...
    }
}
```

我们再按照中介模式，将上面的代码重新实现一下。在新的代码实现中，各个控件只跟中介对象交互，中介对象负责所有业务逻辑的处理。

```
public interface Mediator {
    void handleEvent(Component component, String event);
}

public class LandingPageDialog implements Mediator {
    private Button loginButton;
    private Button regButton;
    private Selection selection;
    private Input usernameInput;
    private Input passwordInput;
    private Input repeatedPswdInput;
    private Text hintText;

    @Override
    public void handleEvent(Component component, String event) {
        if (component.equals(loginButton)) {
            String username = usernameInput.text();
            String password = passwordInput.text();
            //校验数据...
            //做业务处理...
        } else if (component.equals(regButton)) {
            //获取usernameInput、passwordInput、repeatedPswdInput数据...
            //校验数据...
            //做业务处理...
        } else if (component.equals(selection)) {
            String selectedItem = selection.select();
            if (selectedItem.equals("login")) {
                usernameInput.show();
                passwordInput.show();
                repeatedPswdInput.hide();
                hintText.hide();
                //...省略其他代码
            } else if (selectedItem.equals("register")) {
                //....
            }
        }
    }
}

public class UIControl {
    private static final String LOGIN_BTN_ID = "login_btn";
    private static final String REG_BTN_ID = "reg_btn";
    private static final String USERNAME_INPUT_ID = "username_input";
    private static final String PASSWORD_INPUT_ID = "pswd_input";
    private static final String REPEATED_PASSWORD_INPUT_ID = "repeated_pswd_input";
    private static final String HINT_TEXT_ID = "hint_text";
    private static final String SELECTION_ID = "selection";

    public static void main(String[] args) {
        Button loginButton = (Button)findViewById(LOGIN_BTN_ID);
        Button regButton = (Button)findViewById(REG_BTN_ID);
        Input usernameInput = (Input)findViewById(USERNAME_INPUT_ID);
        Input passwordInput = (Input)findViewById(PASSWORD_INPUT_ID);
        Input repeatedPswdInput = (Input)findViewById(REPEATED_PASSWORD_INPUT_ID);
        Text hintText = (Text)findViewById(HINT_TEXT_ID);
        Selection selection = (Selection)findViewById(SELECTION_ID);

        Mediator dialog = new LandingPageDialog();
        dialog.setLoginButton(loginButton);
    }
}
```

```
dialog.setRegButton(regButton);
dialog.setUsernameInput(usernameInput);
dialog.setPasswordInput(passwordInput);
dialog.setRepeatedPswdInput(repeatedPswdInput);
dialog.setHintText(hintText);
dialog.setSelection(selection);

loginButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        dialog.handleEvent(loginButton, "click");
    }
});

regButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        dialog.handleEvent(regButton, "click");
    }
});

//....
}
```

从代码中我们可以看出，原本业务逻辑会分散在各个控件中，现在都集中到了中介类中。实际上，这样做既有好处，也有坏处。好处是简化了控件之间的交互，坏处是中介类有可能会变成大而复杂的“上帝类”（God Class）。所以，在使用中介模式的时候，我们要根据实际的情况，平衡对象之间交互的复杂度和中介类本身的复杂度。

中介模式 VS 观察者模式

前面讲观察者模式的时候，我们讲到，观察者模式有多种实现方式。虽然经典的实现方式没法彻底解耦观察者和被观察者，观察者需要注册到被观察者中，被观察者状态更新需要调用观察者的update()方法。但是，在跨进程的实现方式中，我们可以利用消息队列实现彻底解耦，观察者和被观察者都只需要跟消息队列交互，观察者完全不知道被观察者的存在，被观察者也完全不知道观察者的存在。

我们前面提到，中介模式也是为了解耦对象之间的交互，所有的参与者都只与中介进行交互。而观察者模式中的消息队列，就有点类似中介模式中的“中介”，观察者模式的中观察者和被观察者，就有点类似中介模式中的“参与者”。那问题来了：中介模式和观察者模式的区别在哪里呢？什么时候选择使用中介模式？什么时候选择使用观察者模式呢？

在观察者模式中，尽管一个参与者既可以是观察者，同时也可以是被观察者，但是，大部分情况下，交互关系往往都是单向的，一个参与者要么是观察者，要么是被观察者，不会兼具两种身份。也就是说，在观察者模式的应用场景中，参与者之间的交互关系比较有条理。

而中介模式正好相反。只有当参与者之间的交互关系错综复杂，维护成本很高的时候，我们才考虑使用中介模式。毕竟，中介模式的应用会带来一定的副作用，前面也讲到，它有可能会产生大而复杂的上帝类。除此之外，如果一个参与者状态的改变，其他参与者执行的操作有一定先后顺序的要求，这个时候，中介模式就可以利用中介类，通过先后调用不同参与者的方法，来实现顺序的控制，而观察者模式是无法实现这样的顺序要求的。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

中介模式的设计思想跟中间层很像，通过引入中介这个中间层，将一组对象之间的交互关系（或者依赖关系）从多对多（网状关系）转换为一对多（星状关系）。原来一个对象要跟n个对象交互，现在只需要跟一个中介对象交互，从而最小化对象之间的交互关系，降低了代码的复杂度，提高了代码的可读性和可维护性。

观察者模式和中介模式都是为了实现参与者之间的解耦，简化交互关系。两者的不同在于应用场景上。在观察者模式的应用场景中，参与者之间的交互比较有条理，一般都是单向的，一个参与者只有一个身份，要么是观察者，要么是被观察者。而在中介模式的应用场景中，参与者之间的交互关系错综复杂，既可以是消息的发送者、也可以同时是消息的接收者。

课堂讨论

在讲观察者模式的时候，我们有讲到EventBus框架。当时我们认为它是观察者模式的实现框架。EventBus作为一个事件处理的中心，事件的派送、订阅都通过这个中心来完成，那是不是更像中介模式的实现框架呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 大头 2020-04-20 07:04:38
想到了现在流行的微服务，注册中心可以理解为广义的中介模式，防止各个服务间错综复杂的调用 [13赞]
- 小晏子 2020-04-20 09:07:34
eventbus更属于观察者模式，首先eventbus中不处理业务逻辑，只提供了对象与对象之间交互的管道；而中介模式为了解决多个对象之间交互的问题，将多个对象的行为封装到一起（中介），然后任意对象和这个中介交互，中介中包含了具体业务逻辑。其次从其实现的思路上，EventBus 和观察者都需要定义 Observer，并且通过 register() 函数注册 Observer，也都需要通过调用某个函数（比如，EventBus 中的 post() 函数）来给 Observer 发送消息。而且eventbus并没有中介模式大而臃肿的上帝类问题。 [5赞]
- 李小四 2020-04-20 11:42:59
设计模式_73:
作业:
个人认为还是观察者模式，当然，引入消息队列的观察者模式可以理解为中介模式的一种，它的业务调用更有规律，它不要求被调用者的顺序。

感想:
中介模式看起来，感觉是要带领我们回到面向过程的老路，就文中所述的“上帝类”，有了这个中介，依然在维护复杂的调用关系。 [3赞]
- 小文同学 2020-04-20 09:56:12
eventbus 是不带业务处理的，而且bus不会随着业务复杂而改变，所以属于观察者模式 [1赞]
- test 2020-04-20 08:49:02
eventbus解决的是消息的发布订阅等，跟具体的业务没关系，所以是观察者模式 [1赞]

- eason2017 2020-04-20 07:55:38
从定义上看，中介模式是解决一组对象之间的交互，而Evenybus并不是解决这块的，解决的是所有观察者和被观察者之间的交互方式。所以，确切的说，它并不算中介模式。不知回答是否正确，请指点，谢谢 [1赞]
- 黄林晴 2020-04-20 07:43:19
打卡
在实际的开发中 UI 控件变化那种感觉不太适合中介模式
因为要把所有的控件view 都传到中介类中才可以获取到输入的内容 感觉比较奇怪，就像只是把某个方法单独提取到一个类中一样 [1赞]
- iLeGeND 2020-04-20 00:41:12
感觉23中设计模式之间本身就有某种耦合 好多不易区分 [1赞]
- 往事随风，顺其自然 2020-04-20 16:40:44
设计模式有点像是咬文嚼字的感觉，其实很多可以通用，编码是一样，非得叫的杂乱，随着技术越来越多这种反而庞杂，以后可以有人把这些凝练成通用就更好了，去除各大门派的花哨部分，取其精华，然后截取精华--截拳道，这样对于后者更加便宜
- 守拙 2020-04-20 14:34:44
我按照老师的demo写了一遍中介模式的dialog实现, 发现不就是Dialog impl OnClickListener嘛...

关于Observer与Mediator的区别

Observer定义了一对多(one-to-many)的依赖关系, Mediator封装了多个对象互相之间的交互方式. Oberv
er定义了单向的订阅关系, Mediator通过引入中间件的方式解决多个对象之间混乱的依赖关系与通信方式.

一个可爱的小栗子:

Observer不能做什么:

1. 观察者: hey订阅号, 你到底什么时候更新?
2. 观察者: 订阅号你到底什么时候更新? 我怎么没收到更新?
3. 观察者: 我知道订阅号更新了!(假话)

Observer能做什么:

1. 观察者乖乖坐等更新;
2. 一段时间后...
3. 订阅号: hey我更新了, 小崽子们!

Mediator不能做什么:

1. 乘客1: hey taxi1, 带我去xx(不会得到任何回应)
2. 乘客2: hey taxi2, 带我去xx(不会得到任何回应)
3. 乘客1: hey taxi2, 带我去xx(不会得到任何回应)

Mediator能做什么:

1. 乘客1: dd车辆调度中心, 我要去xx, 请派车接我!(司机正在赶来)
2. 乘客2: dd车辆调度中心, 我要去xx, 请派优享来接我!(优享司机马上就到!)

课堂讨论:

EventBus基于观察者模式实现, 并不符合中介模式的定义. 中介模式封装一组对象间交互, 将这组对象间的交互委派给中介对象交互. EventBus仅负责发布消息, 并不处理发布者/订阅者的任何交互.

- Demon.Lee 2020-04-20 13:47:41
老师的这个例子, 我还是没看明白哪里体现了“多个对象之间交互”-->“多个对象之间通过中介交互”的变化。比如之前是regButton, loginButton, Selection三者之间是怎么交互的, 我没看出来。然后又是如何把这三个对象的相互调用, 改成了通过中介类交互的, 我也没看明白。我去查阅了其他资料, 发现什么虚拟聊天室什么的代码就体现了上面这一点: 用户A发消息给用户B是直接交互, 改造之后是, 用户A发消息给中介, 中介再把消息转给消息B。有小伙伴们理解了么, 能否分享下。
- 忆水寒 2020-04-20 12:51:28
eventbus没有处理相应的业务逻辑, 是一种观察者模型。
- Heaven 2020-04-20 10:30:15
个人认为是很像的, EventBus其本质上就是为了在交互双方的基础上进行了抽取, 形成了这个框架, 可以被称为是中介, 但是也只是像, 因为观察者模式的这种实现本身就和中介模式类似, 但是也只限于这种实现罢了, 因为比起中介模式那种更加具体的中介类, EventBus将交互双方解耦的更加详细
- 侯金彪 2020-04-20 09:52:51
相同点是中介模式和观察者模式都是为了实现模块解耦, 不同点是eventbus只是单纯的消息通知, 而中介类中需要定义处理逻辑, 中介相对于观察者把业务逻辑处理提前了
个人理解, 请争哥指点