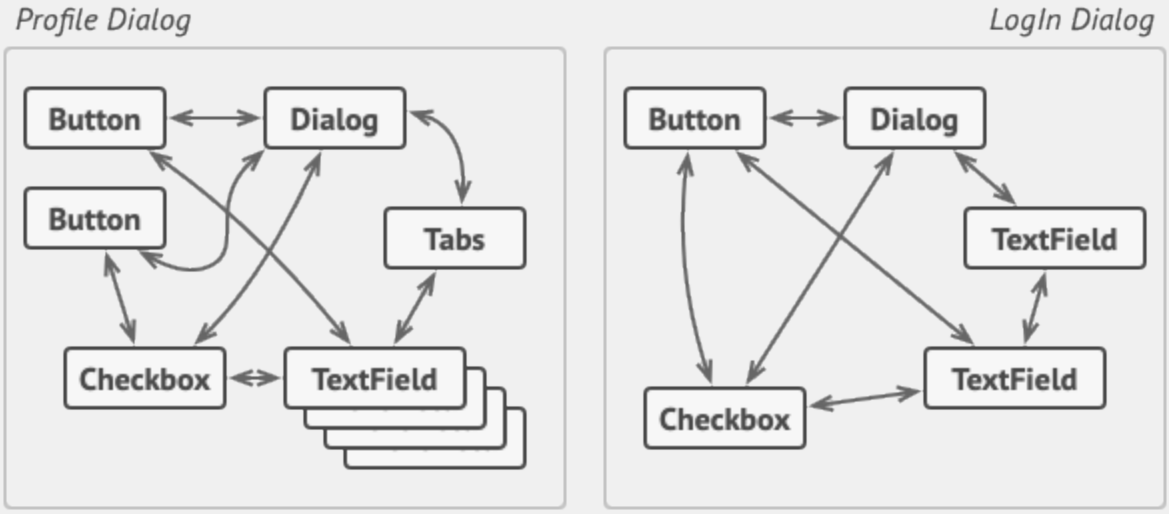
第4章 中介者模式

提出问题

> 面向对象设计将系统功能分布到各个对象中，可能会导致对象之间存在许多连接，最坏的情况就是对象两两之间都存在连接，这种高耦合会导致系统行为改变将变得十分困难。例如一个图形界面由很多控件组成，各个控件之间需要进行交互来完成系统功能，当交互情况变得复杂时，控件之间的耦合性大大提升。思考如何降低这种耦合性？



模式名称

中介者模式（Mediator）

设计意图

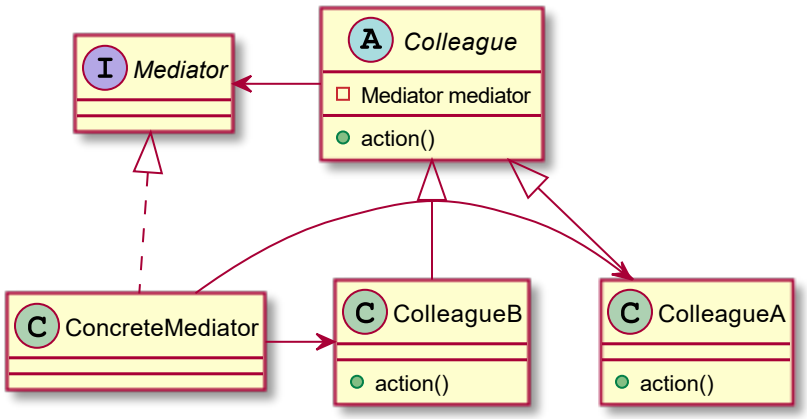
中介者模式（Mediator）用一个中介对象封装一系列对象交互，终结者使各对象不需要显示地交互作用，从而使其耦合松散，而且可以独立地改变他们之间地交互。

> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly,

> and it lets you vary their interaction independently.

设计结构

类图



参与者

- 抽象中介者（Mediator）：定义统一接口，用于同事之间地通信；
- 具体中介者（ConcreteMediator）：从具体同事接收消息，向具体同事对象发出命令，协调各同事；
- 抽象同事（Colleague）：依赖中介者，与其他同事间通信交给中介者转发；
- 具体同事（ConcreteColleague）：负责实现自发行为，转发交给中介。

实现

抽象中介者：

```
public abstract class Mediator {
    private ColleagueA c1;
    private ColleagueB c2;
    public ColleagueA getC1() {
        return c1;
    }
    public void setC1(ColleagueA c1) {
        this.c1 = c1;
    }
    public ColleagueB getC2() {
        return c2;
    }
    public void setC2(ColleagueB c2) {
        this.c2 = c2;
    }
    public abstract void transfer(String message, Colleague c);
}
```

抽象同事：

```
public abstract class Colleague {
    private Mediator mediator;
    public abstract void action(String message);
    public Mediator getMediator() {
        return mediator;
    }
    public void setMediator(Mediator mediator) {
        this.mediator = mediator;
    }
    public abstract void send(String message);
}
```

具体中介者：

```
public class ConcreteMediator extends Mediator{
    @Override
    public void transfer(String message, Colleague c) {
        if(c.equals(this.getC1())) {
            this.getC2().action(message);
        }else {
            this.getC1().action(message);
        }
    }
}
```

具体同事：

```
public class ColleagueA extends Colleague{
    public void setMediator(Mediator mediator) {
        super.setMediator(mediator);
        mediator.setC1(this);
    }
    @Override
    public void action(String message) {
        System.out.printf("A处理信息：%s\n", message);
    }
    @Override
    public void send(String message) {
        this.getMediator().transfer(message, this);
    }
}
```

效果与适用性

优点

- 减少类间依赖，将多对多依赖转化为一对多，降低类间耦合
- 类间各司其职，符合迪米特法则

缺点

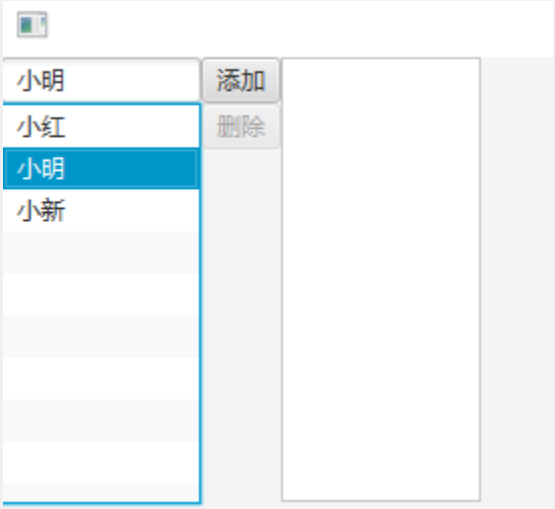
- 同事类越多，中介者会越臃肿，变得复杂难以维护

扩展案例

用户添加对话框

> 对话框时图形用户界面经常要用的窗口组件，一个对话框经常包括多个类型不同的用户交互控件，这些控件交互完成一个系统功能。如下图所示的用户添加对话框包含的主要交互过程：（1）初次显示对话框时两个按钮均处于失效状态；（2）当用户在左侧用户列表中选中一个用户名时，选中的用户名被填到其上面的文本框中，同时 添加 按钮变为有效状态；（3）单击 添加 按钮后，文本框中的用户名添加到右侧列表中，按钮恢复失效状态；（4）选中右侧列表中的人名时， 删除 按钮变为有效状态；（5）当单击 删除 按钮后，右侧

列表删除对应名字，左侧列表中添加对应名字。



这里的交互主要是由控件状态变化引起，设计一个中介接口，定义触发交互的功能：

```
public interface Mediator {
    // 促发一次交互，发起者将本身信息发送给中介者，由中介者判断如何与另外的同事交互
    public void changed(Colleague source);
}
```

在具体中介者中，需要建立同事与中介的联系，据此定义同事接口：

```
public interface Colleague {
    // 建立中介与同事的联系
    public void setMediator(Mediator m);
}
```

具体中介者采用一个面板来放置各个控件，定义如下：

```
public class SelectBox extends HBox implements Mediator{
    private TextField selected;
    private ListColleague addList;
    private ListColleague deleteList;
    private ButtonColleague addBtn;
    private ButtonColleague deleteBtn;

    public SelectBox() {
        // 控件创建、初始化及布局

        // 建立控件与对话框的联系
        addList.setMediator(this);
        deleteList.setMediator(this);
        addBtn.setMediator(this);
        deleteBtn.setMediator(this);
    }

    // 控件之间的交互逻辑实现，根据传递的对象判断哪个操作被促发
    @Override
    public void changed(Colleague c) {
        if(c == addList) {
            ObservableList<Integer> selectedIndices = addList.getSelectionModel().getSelectedIndic
            selected.setText(addList.getItems().get(((Integer)selectedIndices.get(0)).intValue()).
            addBtn.setDisable(false);
            deleteBtn.setDisable(true);
        }else if(c == deleteList) {
            addBtn.setDisable(true);
            deleteBtn.setDisable(false);
        }else if(c == addBtn) {
            ObservableList<Integer> selectedIndices = addList.getSelectionModel().getSelectedIndic
            int index = ((Integer)selectedIndices.get(0)).intValue();
            addList.getItems().remove(index);
            deleteList.getItems().add(selected.getText());
            selected.setText("");
            addBtn.setDisable(true);
            deleteBtn.setDisable(true);
        }else if(c == deleteBtn) {
            ObservableList<Integer> selectedIndices = deleteList.getSelectionModel().getSelectedIn
            int index = ((Integer)selectedIndices.get(0)).intValue();
            System.out.println(deleteList.getItems().get(index));
            addList.getItems().add(deleteList.getItems().get(index));
            deleteList.getItems().remove(index);
            addBtn.setDisable(true);
            deleteBtn.setDisable(true);
        }
    }
```

```
}
```

设计两种同事类：

```
public class ListColleague extends ListView<String> implements Colleague{
    private Mediator m;
    // 业务处理与交互触发之间的逻辑实现
    public ListColleague() {
        ListColleague list = this;
        list.setOnMouseClicked(e -> {
            if(list.getSelectionModel().getSelectedIndices().size() > 0) {
                m.changed(list);
            }
        });
    }
    @Override
    public void setMediator(Mediator m) {
        this.m = m;
    }
}
public class ButtonColleague extends Button implements Colleague{
    private Mediator m;
    // 业务处理与交互触发之间的逻辑实现
    public ButtonColleague(String text) {
        super(text);
        this.setOnAction(e -> {
            m.changed(this);
        });
    }
    @Override
    public void setMediator(Mediator m) {
        this.m = m;
    }
}
```