行为型模式（三）

2020年5月18日 15:51

行为型设计模式（三）

作业要求：

1、结合实例，绘制备忘录模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

2、结合实例，绘制观察者模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

3、结合实例，绘制状态模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

4、某文字编辑软件须提供撤销(Undo)和重做/恢复(Redo)功能，并且该软件可支持文档对象的多步撤销和重做。开发人员决定采用备忘录模式来实现该功能，在实现过程中引入栈(Stack)作为数据结构。在实现时，可以将备忘录对象保存在两个栈中，一个栈包含用于实现撤销操作的状态对象，另一个栈包含用于实现重做操作的状态对象。在实现撤销操作时，会弹出撤销栈栈顶对象以获取前一个状态并将其设置给应用程序：同样，在实现重做操作时，会弹出重做栈栈顶对象以获取下一个状态并将其设置给应用程序。绘制对应的类图并编程模拟实现。

要求：

1、模式结构图要求使用工具软件进行绘制，模式分析需要说明每个角色及作用。 如果可能，每个模式最好使用实际案例来说明。

2、以WORD文档/PDF格式提交。

3、提交的作业如果有多个文件，就提交多个，不要做压缩包。

- **作业与笔记github地址**：
  https://github.com/baobaotql/CCNU_DesignPattern

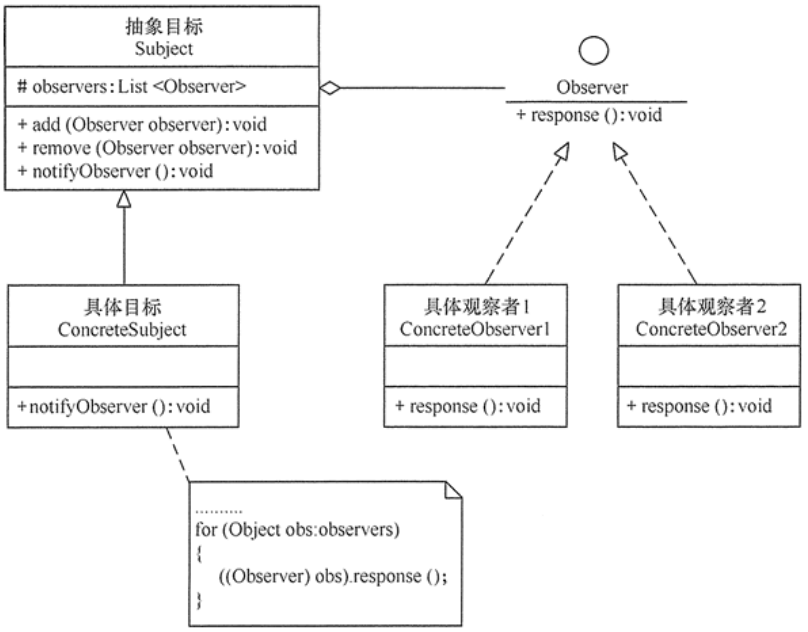
- **Q1:结合实例，绘制观察者模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**

  A1:

  - **观察者模式定义：**
    指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。
  - **优点**
    - 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
    - 目标与观察者之间建立了一套触发机制。
  - **缺点**
    - 目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。
    - 当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。
  - **模式结构**
    - 抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。
    - 具体主题（Concrete　Subject）角色：也叫具体目标类，它实现抽象

目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

- 抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。
- 具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。
- **模式结构图**



- **模式分析**

实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则。

- **模式结构代码**

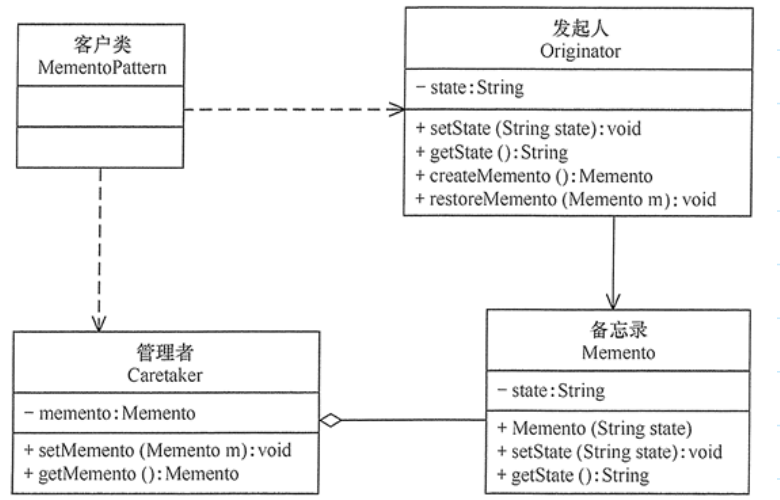
```
package observer;
import java.util.*;
public class ObserverPattern
{
    public static void main(String[] args)
    {
        Subject subject=new ConcreteSubject();
        Observer obs1=new ConcreteObserver1();
        Observer obs2=new ConcreteObserver2();
        subject.add(obs1);
        subject.add(obs2);
        subject.notifyObserver();
    }
}
//抽象目标
abstract class Subject
{
    protected List<Observer> observers=new ArrayList<Observer>();
    //增加观察者方法
    public void add(Observer observer)
    {
        observers.add(observer);
    }
```

```
    //删除观察者方法
    public void remove(Observer observer)
    {
        observers.remove(observer);
    }
    public abstract void notifyObserver(); //通知观察者方法
}
//具体目标
class ConcreteSubject extends Subject
{
    public void notifyObserver()
    {
        System.out.println("具体目标发生改变...");
        System.out.println("--------------");

        for(Object obs:observers)
        {
            ((Observer)obs).response();
        }

    }
}
//抽象观察者
interface Observer
{
    void response(); //反应
}
//具体观察者1
class ConcreteObserver1 implements Observer
{
    public void response()
    {
        System.out.println("具体观察者1作出反应! ");
    }
}
//具体观察者1
class ConcreteObserver2 implements Observer
{
    public void response()
    {
        System.out.println("具体观察者2作出反应! ");
    }
}
```

- **Q2:结合实例，绘制备忘录模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**
  A2:
  - **备忘录模式定义**
    在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式。
  - **优点**
    - 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。

- 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。
- 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

- **缺点**
  - 资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。
- **模式结构**
  - 发起人（Originator）角色：记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息。
  - 备忘录（Memento）角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。
  - 管理者（Caretaker）角色：对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。
  - **模式结构图**



  - **模式代码**

```
package memento;

public class MementoPattern
{
    public static void main(String[] args)
    {
        Originator or=new Originator();
        Caretaker cr=new Caretaker();
        or.setState("S0");
        System.out.println("初始状态:"+or.getState());
        cr.setMemento(or.createMemento()); //保存状态
        or.setState("S1");
        System.out.println("新的状态:"+or.getState());
        or.restoreMemento(cr.getMemento()); //恢复状态
        System.out.println("恢复状态:"+or.getState());
    }
}
//备忘录
class Memento
{
```

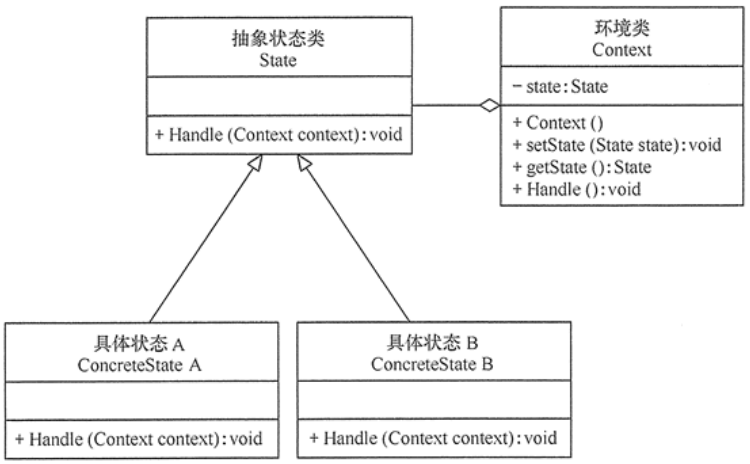
```
    private String state;
    public Memento(String state)
    {
        this.state=state;
    }
    public void setState(String state)
    {
        this.state=state;
    }
    public String getState()
    {
        return state;
    }
}
//发起人
class Originator
{
    private String state;
    public void setState(String state)
    {
        this.state=state;
    }
    public String getState()
    {
        return state;
    }
    public Memento createMemento()
    {
        return new Memento(state);
    }
    public void restoreMemento(Memento m)
    {
        this.setState(m.getState());
    }
}
//管理者
class Caretaker
{
    private Memento memento;
    public void setMemento(Memento m)
    {
        memento=m;
    }
    public Memento getMemento()
    {
        return memento;
    }
}
```

- **Q3:结合实例，绘制状态模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**
- **A3:**
  - **状态模式定义**
    对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状

态对象在其内部状态发生改变时改变其行为

- **优点**
  - 状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”。
  - 减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。
  - 有利于程序的扩展。通过定义新的子类很容易地增加新的状态和转换。
- **缺点**
  - 状态模式的使用必然会增加系统的类与对象的个数。
  - 状态模式的结构与实现都较为复杂，如果使用不当会导致程序结构和代码的混乱。
- **模式结构**
  - 环境（Context）角色：也称为上下文，它定义了客户感兴趣的接口，维护一个当前状态，并将与状态相关的操作委托给当前状态对象来处理。
  - 抽象状态（State）角色：定义一个接口，用以封装环境对象中的特定状态所对应的行为。
  - 具体状态（Concrete State）角色：实现抽象状态所对应的行为。
  - **模式分析**

    状态模式把受环境改变的对象行为包装在不同的状态对象里，其意图是让一个对象在其内部状态改变的时候，其行为也随之改变。现在我们来分析其基本结构和实现方法。
  - **模式结构图**



  - **模式代码**

```
package state;

public class StatePatternClient
{
    public static void main(String[] args)
    {
        Context context=new Context();    //创建环境
        context.Handle();    //处理请求
        context.Handle();
        context.Handle();
        context.Handle();
    }
}
//环境类
class Context
```

```
{
    private State state;
    //定义环境类的初始状态
    public Context()
    {
        this.state=new ConcreteStateA();
    }
    //设置新状态
    public void setState(State state)
    {
        this.state=state;
    }
    //读取状态
    public State getState()
    {
        return(state);
    }
    //对请求做处理
    public void Handle()
    {
        state.Handle(this);
    }
}
//抽象状态类
abstract class State
{
    public abstract void Handle(Context context);
}
//具体状态A类
class ConcreteStateA extends State
{
    public void Handle(Context context)
    {
        System.out.println("当前状态是 A.");
        context.setState(new ConcreteStateB());
    }
}
//具体状态B类
class ConcreteStateB extends State
{
    public void Handle(Context context)
    {
        System.out.println("当前状态是 B.");
        context.setState(new ConcreteStateA());
    }
}
```

- **Q4:某文字编辑软件须提供撤销(Undo)和重做/恢复(Redo)功能,并且该软件可支持文档对象的多步撤销和重做。在实开发人员决定采用备忘录模式来实现该功能,在实现过程中引入栈(Stack)作为数据结构。现时,可以将备忘录对象保存在两个栈中,一个栈包含用于实现撤销操作的状态对象,另一个栈包含用于实现重做操作的状态对象。在实现撤销操作时,会弹出撤销栈栈顶对象以获取前一个状态并将其设置给应用程序;同样,在实现重做操作时,会弹出重做栈栈顶对象以获取下一个状态并将其设置给应用程序。绘制对应的类图并编程模拟实现。**
- **A4:**

- 两种实现（一个与题目要求略不符，属于自己笔记内容）

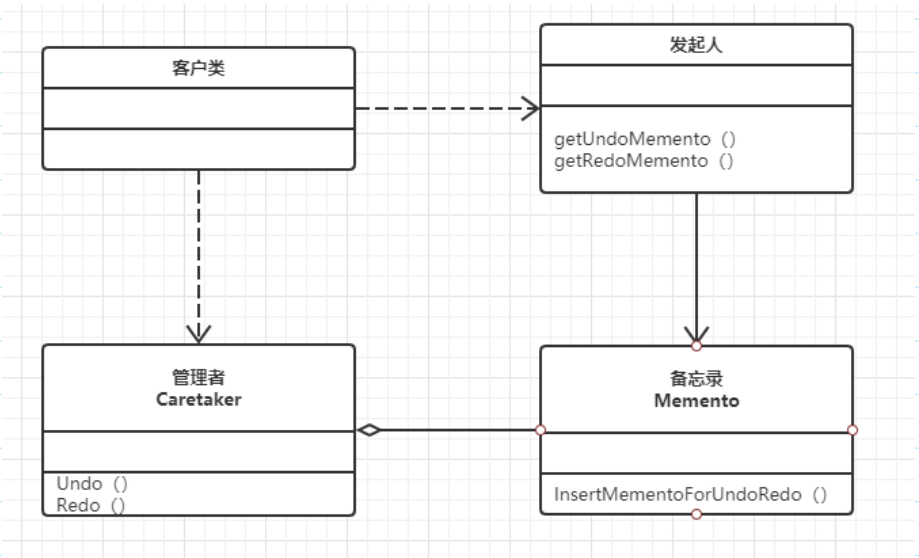
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.undo.*;

class UndoDemo extends JFrame {

    static JTextArea text = new JTextArea();
    static JPanel pnl = new JPanel();
    static JButton unbtn = new JButton("撤销");
    static JButton rebtn = new JButton("恢复");
    static UndoManager undomg = new UndoManager();

    UndoDemo() {

        super("撤销、恢复功能实例");
        setVisible(true);
        setSize(400,300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout(5,5));

        pnl.setLayout(new FlowLayout(5));
        pnl.add(unbtn);
        pnl.add(rebtn);
        add(pnl,BorderLayout.NORTH);
        add(text,BorderLayout.CENTER);

        text.getDocument().addUndoableEditListener(undomg);

        unbtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                if(undomg.canUndo()) {
                    undomg.undo();
                } else {
                    JOptionPane.showMessageDialog(null,"无法撤销","警
告",JOptionPane.WARNING_MESSAGE);
                }
            }
        });

        rebtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                if(undomg.canRedo()) {
                    undomg.redo();
                } else {
                    JOptionPane.showMessageDialog(null,"无法恢复","警
告",JOptionPane.WARNING_MESSAGE);
                }
            }
        });
    }
```

```
    public static void main(String[] args) {
        new UndoDemo();
    }
}
```

程序的运行结果如下：

- **核心代码块**

  Caretaker类将memento (state)保存在两个栈中。Undo栈为Undo操作保存Memento (state)，而Redo栈为/Redo操作保存Memento (state)



Collapse Copy Code

```
class Caretaker
 {
    private Stack<Memento> UndoStack = new Stack<Memento>();
    private Stack<Memento> RedoStack = new Stack<Memento>();

    public Memento getUndoMemento()
    {
        if (UndoStack.Count >= 2)
        {
            RedoStack.Push(UndoStack.Pop());
            return UndoStack.Peek();
        }
        else
            return null;
    }
    public Memento getRedoMemento()
    {
```

```
            if (RedoStack.Count != 0)
            {
                Memento m = RedoStack.Pop();
                UndoStack.Push(m);
                return m;
            }
            else
                return null;
        }
        public void InsertMementoForUndoRedo(Memento memento)
        {
            if (memento != null)
            {
                UndoStack.Push(memento);
                RedoStack.Clear();
            }
        }
        public bool IsUndoPossible()
        {
            if (UndoStack.Count >= 2)
            {
                return true;
            }
            else
                return false;

        }
        public bool IsRedoPossible()
        {
            if (RedoStack.Count != 0)
            {
                return true;
            }
            else
                return false;
        }

    }
```

//undo redo类实现

```
public class UndoRedo : IUndoRedo
    {
        Caretaker _Caretaker = new Caretaker();
        MementoOriginator _MementoOriginator = null;
        public event EventHandler EnableDisableUndoRedoFeature;

        public UndoRedo(Canvas container)
        {
            _MementoOriginator = new MementoOriginator(container);

        }
        public void Undo(int level)
        {
            Memento memento = null;
            for (int i = 1; i <= level; i++)
            {
                memento = _Caretaker.getUndoMemento();
            }
```

```
            if (memento != null)
            {
                _MementoOriginator.setMemento(memento);

            }
            if (EnableDisableUndoRedoFeature != null)
            {
                EnableDisableUndoRedoFeature(null, null);
            }
        }

        public void Redo(int level)
        {
            Memento memento = null;
            for (int i = 1; i <= level; i++)
            {
                memento = _Caretaker.getRedoMemento();
            }
            if (memento != null)
            {
                _MementoOriginator.setMemento(memento);

            }
            if (EnableDisableUndoRedoFeature != null)
            {
                EnableDisableUndoRedoFeature(null, null);
            }
        }

        public void SetStateForUndoRedo()
        {
            Memento memento = _MementoOriginator.getMemento();
            _Caretaker.InsertMementoForUndoRedo(memento);
            if(EnableDisableUndoRedoFeature != null)
            {
                EnableDisableUndoRedoFeature(null,null);
            }
        }

        public bool IsUndoPossible()
        {
            return _Caretaker.IsUndoPossible();

        }
        public bool IsRedoPossible()
        {
           return _Caretaker.IsRedoPossible();
        }
    }
```