Java 基于可观察者技术的 MVC 编程的实现方式

Java according to can technical MVC in observer plait the distance realizes way

曹大有　赵 韬　（郧阳师范高等专科学校 计算机科学系 湖北丹江口 442700）

摘要：MVC 架构是将应用程序对象的模型与显示它的 GUI 元素相分离，在 Java Swing 组件体系中应用很广泛。文中讨论了如何基于 Java Observable 类、Observer 接口和 PropertyChangeSupport 类、PropertyChangeListener 接口来实现 GUI 界面的 MVC 编程过程，并对实现过程进行了详细的讨论。

关键词：MVC　业务层　业务对象　GUI 层

1　问题提出

在建立用户图形界面时，我们可能有多种方式来修改一个整型数据，并且这个整型数据的变化有可能会引起若干个不同的显示区域的更新。例如：假设我们需要创建一个改变整型数据的滑动条，而这一整型数据又以某种标签的形式显示。当该整型数据改变后，我们需要标签也能立即得到更新，但我们又不希望标签知道关于滑动条的任何信息。如果我们不知道为什么标签不应该了解滑动条，我们可以设想将滑动条换成使用文本域来输入所需要的数据，这时将会怎样？我们不必在每次改变输入源时去修改标签。

这时最好是创建一个可观察的整型变量，这将允许其它对象对它给予关注。当该整型变量改变时，它将通知那些关注它的对象（称为观察者），它已改变了。在上面标签的例子中，标签将得到数值已经改变的通知，它将查询整型变量的新值，并重新绘制其本身。这样就使得标签可能正确地显示数据，而不管使用什么手段来改变数值。这一概念称为模型视图控制器（MVC）。

Java Foundation Class（简称 JFC）系统是基于模型－视图－控制器的概念设计的。模型视图控制器 MVC 的概念源于 Smalltalk，主要用于设计用户界面。在这样的界面中，应用程序由三个类组成，即：模型、视图和控制器。模型表示数据或应用对象，用于操作和向用户展示内容，它是对数据源和所有基于对这些数据操作的封装。视图是模型的屏幕表示，它是表示模型当前状态的对象，它将用户的请求传给控制器。控制器定义了用户界面与用户输入进行交互的方法，它是操作模型的对象。

使用 MVC 设计模式的主要优点是模型和视图分离，清晰地分解表示和业务层。这样一来，我们就可以把业务逻辑的表示部分分离出来，进而我们能够建立或改变视图而不必改变模型或操作模型的控制器逻辑部分，而 MVC 还允许使用多个视图表示同一个模型。

Java 是通过观察者模式来支持这种责任的分离。我们可以通过使用 java.util 包中的 Observable 类和 Observer 接口，也可以通过使用 java.beans 包中的 PropertyChangeSupport 类和 PropertyChangeListener 接口来完成 MVC 模式的编程。

2　基于 Observable 类和 Observer 接口 MVC 编程的分层架构

Observable（可观察）类允许一个对象当其改变时能通知其它对象，它是所有可观察类的超类。创建可观察者子类最重要的方法是 setChanged( )和 notifyObservers( )。setChanged( )方法用来标记可观察者的对象已经改变，以便使用户可以调用 notifyObservers( )方法来通知观察者。

notifyObservers( )方法检查 changed 标志是否设置，如果没有，则该方法不发出任何通知，它有两个版本，即：public void notifyObservers( )和 public void notifyObservers(Object arg)。

下述代码段用来设置 changed 标志并通知观察者所发生的改变：

```
setChanged();notifyObservers();
```

(1) 可观察者整型类 ObservableInt 的实现程序片段

```
public class ObservableInt extends Observable{
    int value;public ObservableInt(){ value=0;} //构造方法
    public ObservableInt(int newValue){ value=newValue;} //构造方法
    public synchronized void setValue(int newValue){ //属性改变事件
        if(newValue! =value){ value=newValue;setChanged();notifyObservers();}}
    public synchronized int getValue(){ return value;}} //返回属性值
```

可观察类有一个对等的接口称为观察者(Observer)。任何需要接收有关一个可观察对象的修改情况的类,都必须实现观察者接口。观察者接口只包括一个称为 update()的方法,该方法在对象改变时调用。update()方法的原型为:

```
void update(Observable obj, Object arg);
```

其中 obj 是刚刚发生变化的可观察对象,arg 是由可观察对象在调用 notifyObservers(Object arg)方法时所传送的值。如果调用 notifyObservers()时不带参数,arg 则为 null。

(2) 一个 Label 类实例 IntLabel 程序片段

该 Label 类实现了观察者接口,它可以使用一个整型变量获得各种变化的通知,并使用新值立即更新自己。

```
public class IntLabel extends JLabel implements Observer{
    private ObservableInt intValue; //可观察者属性
    public IntLabel(ObservableInt theInt){ //构造方法
        intValue = theInt; intValue. addObserver(this); //加观察者
        setText("" + intValue.getValue());} //设置标签内容
    public void update(Observable obs,Object arg){ //更新通知事件
        setText("" + intValue.getValue());}} //设置标签内容
```

现在我们已经有了一个以 ObservableInt 形式定义的模型对象,一个以 IntLabel 形式定义的视图对象,下面定义控制器 IntSlider 类。

```
public class IntSlider extends JSlider implements Observer{
    private ObservableInt intValue; //可观察者属性
    public IntSlider(ObservableInt newValue){ //构造方法
        super(); intValue = newValue; intValue.addObserver(this); //加观察者
        setValue(intValue. getValue()); //设置 JSlider 值
        addChangeListener(new ChangeListener(){
            public void stateChanged(ChangeEvent event){ //JSlider 属性改变事件
                intValue.setValue(getValue());}});}
    public void update(Observable obs,Object arg){ //更新通知事件
        setValue(intValue. getValue());}} //设置 JSlider 值
```

上面看上去要做的工作很多,但要使用它们是非常容易的。如下面的小程序所示:

```
public class ObservableApplet1 extends JApplet{
    ObservableInt myIntValue; //可观察者属性
    public void init(){ //JApplet 的初始化并且加入控件
        myIntValue = new ObservableInt(5); setLayout(new GridLayout(2,0));
        add(new IntSlider(myIntValue)); add(new IntLabel(myIntValue));}}
```

(3) 创建一个修改 Observable Int 的文本线

这样当我们运行它时,只要改变滑动条,标签就会立即更新,然而标签对滑动条并不了解,而滑动条对标签也不了解。当然根据 MVC 的观点,我们新增加一个修改 ObservableInt 类对象的文本域也是很容易的。全部工作就是创建一个修改 ObservableInt 的文本域。程序清单如下所示:

```
public class IntTextField extends JTextField
```

```
implements Observer{
    private ObservableInt intValue; //可观察者属性
    public IntTextField(ObservableInt theInt){//构造方法
        super(""+theInt.getValue(),3);intValue=theInt; //设置属性值
        intValue.addObserver(this); //加观察者
        addActionListener(new ActionListener(){//事件监听器
            public void actionPerformed(ActionEvent event){
            Integer intStr;
            try{ intStr=new Integer(getText());
                intValue.setValue(intStr.intValue()); //改变可观察者值
            }catch(Exception oops){{{{{}}}}};}
    public void update(Observable obs,Object arg){ //更新通知事件
        setText(""+intValue.getValue());}}
```

要使用这个类，我们只需要向小程序中添加一个代码就行。如下面所示：

```
public class ObservableApplet2 extends JApplet{
    ObservableInt myIntValue; //可观察者属性
    public void init(){ //JApplet 的初始化并且加入控件
        myIntValue=new ObservableInt(5);setLayout(new GridLayout(3,0));
        add(new IntSlider(myIntValue));add(new IntLabel(myIntValue));
        add(new IntTextField(myIntValue));}}
```

而且这些组件之间都不互相了解，但只要修改了 ObservableInt 类对象的值，它们都会随之更新。

在上面的设计中，我们将模型对象与用于表示它的 GUI 元素相分离，实现这一设计必须有两个关键步骤：

① 实现 Observer 接口的观察者类必须向自己关注的对象注册自己，收到事件通知后，观察者类必须能够做出合适的操作。

② Observable 类的被观察者子类在它们的数据发生变化的时候，必须记住去通知相关的观察者。

3　基于 PropertyChangeSupport 类和 Property ChangeListener 接口 MVC 编程的分层架构

当然有时候，我们期望观察的类可能无法扩展 Observable 类，特别是当该类已经扩展了除 Object 类以外的类时。这个时候，我们可以为这个类提供一个 Observable 对象，将对这个类的关键方法的调用转发给该 Observable 对象。实际上 java.awt 包中 Component 类就是采用了这种方法，它借助了 PropertyChangeSupport 对象而不是 Observable 对象。

PropertyChangeSupport 类非常类似 Observable 类，不过该类位于 java.bean 包中。该 JavaBean API 有助于创建可重用的组件，在 GUI 组件中被广泛应用。当然它也可以用于其它地方。Component 类使用 PropertyChangeSupport 对象，从而使得关注其变化的观察者能够向其注册，以便在标签、面板以及其它 GUI 组件属性发生变化的时候，观察者能够得到通知。

3.1　如何利用 PropertyChangeSupport 对象管理监听器对象

Java Bean 中有一个属性称之为绑定属性，它用于告诉有关的属性监听器，它的值已经改变了。要实现绑定属性，必须实现以下两个关键步骤：

(1) 每当属性值发生改变时，bean 必须给所有注册的监听器发送一个 PropertyChange 的事件。

(2) 要使有关的监听器能够对自己进行注册，bean 必须实现以下两个方法：

```
void addPropertyChangeListener(PropertyChangeListener listener);
void removePropertyChangeListener(PropertyChangeListener listener);
```

但是在 java.beans 包中有一个非常有用的类 PropertyChangeSupport，它可以负责为我们管理监听器。若要使用这个类，可以为可观察者类增加一个该类的数据字段，即：

```
private PropertyChangeSupport changeSupport=new
    PropertyChangeSupport(this);
```

然后将增加和删除属性变更监听器的任务委托给

该对象。

```
public void addPropertyChangeListener(PropertyChangeListener listener){
    changeSupport.addPropertyChangeListener(listener);}
public void removePropertyChangeListener(PropertyChangeListener listener){
    changeSupport.removePropertyChangeListener(listener);}
```

当属性值发生改变时，使用 PropertyChangeSupport 对象的 firePropertyChange()方法，为所有注册的监听器传递一个事件，该方法配有 3 个参数：属性名、老值和新值。如：

```
changeSupport.firePropertyChange("propertyName",oldValue,newValue);
```

而且这些值必须是对象，如果不是对象，就要使用对象封装器。在属性值以生改变后，要将变化的情况通知给观察者，必须实现 PropertyChangeListener 接口，该接口中只有一个方法：

```
void propertyChange(PropertyChangeEvent event);
```

观察者类中可以通过 PropertyChangeEvent 类的：getOldValue()和 getNewValue()方法来获得数值。当然这些方法返回的也是对象。

3.2 用 PropertyChangeSupport 类和 PropertyChangeListener 接口对上面的例子重构

可观察者整型类 IntBean 的实现为：

```
public class IntBean{
    private PropertyChangeSupport changeSupport;int value;
    public IntBean(int newValue){ value=newValue;} //构造方法
    public void setValue(int newValue){
        int oldValue = value; value = newValue; //保存旧值设置新值
        changeSupport.firePropertyChange("value", new Integer(oldValue), new Integer(newValue));} //发送属性改变事件
    public int getValue(){ return value;} //返回当前值
    public void addPropertyChangeListener(PropertyChangeListener listener){
        if(listener==null){return;}
        if(changeSupport==null){changeSupport=new PropertyChangeSupport(this);}
        changeSupport.addPropertyChangeListener(listener);} //加监听者
    public void removePropertyChangeListener(PropertyChangeListener listener){
        if(listener==null||changeSupport==null){return;}
        changeSupport.removePropertyChangeListener(listener);}} //删除监听者
```

观察者 Label 类实例 IntBeanLabel 的实现为：

```
public class IntBeanLabel extends JLabel{
    private IntBean intValue; //可观察者属性
    public IntBeanLabel(IntBean theInt){ intValue=theInt;
        intValue.addPropertyChangeListener(new PropertyChangeListener(){
            public void propertyChange(PropertyChangeEvent event){
                setText(""+((Integer)event.getNewValue()).intValue());}});
        setText(""+intValue.getValue());}} //设置观察者内容
```

控制器 IntBeanSlider 类的实现为：

```
public class IntBeanSlider extends JSlider{
    private IntBean intValue; //可观察者属性
    public IntBeanSlider(IntBean newValue){
        super();intValue=newValue;
        intValue.addPropertyChangeListener(new PropertyChangeListener(){
            public void propertyChange(PropertyChangeEvent event){
                setValue(((Integer)event.getNewValue()).intValue());}});
        addChangeListener(new ChangeListener(){
            public void stateChanged(ChangeEvent event){
                intValue.setValue(getValue());}});}} //设置观察者内容
```

使用它们同样很容易，如下面的小程序所示：

```
public class IntBeanApplet1 extends JApplet{
    IntBean myIntBean; //可观察者属性
    public void init(){ //JApplet 的初始化并且加入控件
        myIntBean = new IntBean(5); setLayout(new GridLayout(2,0));
        add(new IntBeanSlider(myIntBean));
add(new IntBeanLabel(myIntBean));}}
```

增加一个新的控制器同样容易，如 IntBeanTextField 所示：

```
public class IntBeanTextField extends JTextField{
    private IntBean intValue; //可观察者属性
    public IntBeanTextField(IntBean theInt){ //构造方法
        super("" + theInt.getValue(),3); intValue = theInt;
        intValue.addPropertyChangeListener(new PropertyChangeListener(){
            public void propertyChange(PropertyChangeEvent event){ //属性改变事件
                setText("" + ((Integer)event.getNewValue()).intValue());}});
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent event){ //事件监听器
                Integer intStr;
                try{ intStr = new Integer(getText());
                    intValue.setValue(intStr.intValue()); //设置可观察者属性值
                }catch(Exception oops){}}});}}
```

使用见 IntBeanApplet2 所示：

```
public class IntBeanApplet2 extends JApplet{
    IntBean myIntBean; //可观察者属性
    public void init(){ //JApplet 的初始化并且加入控件
        myIntBean = new IntBean(5); setLayout(new GridLayout(3,0));
        add(new IntBeanSlider(myIntBean));
add(new IntBeanLabel(myIntBean));
        add(new IntBeanTextField(myIntBean));}}
```

该设计的的关键之处在于受关注的类 IntBean 借助 PropertyChangeSupport 对象使得其它对象能够观察自己的变化，该 PropertyChangeSupport 对象为 IntBean 类提供了被监听的能力。

4 总结

MVC 的设计者曾设想将组件的外观（视图）与它的使用方式（控制器）相分离，但实际上，GUI 组件的外观与其对用户的支持是紧密耦合的，Java Swing 并没有对视图与控制器进行分离。MVC 的真正价值在于：推动应用程序将其模型分离出来，形成模型自己的域。

这种设计的优点是：层间接口清晰，上下层相互独立。对代码分层本质上就是对责任分层，这样便于代码的维护。例如，在 GUI 层我们可以再创建一个 GUI，但我们无需修改业务对象层中的代码。而在业务层，我们可以再添加一个可能会导致 IntBean 类对象变化的事件源，而我们无需修改 GUI 层。观察者模式提供的这种机制可以自动更新 GUI 层的对象。

在确定使用观察者模式之后，我们可以决定写出自己的注册和监听机制。不过，Java 类库已经分别在两个包中提供完成这两种机制的类。我们可以使用 java.util 包中的 Observable 类和 Observer 接口，也可以使用 java.beans 包中的 PropertyChangeSupport 类和 PropertyChangeListener 接口。

参考文献

1 Steven John Metsker 著，龚 波、冯 军、程群梅等译，设计模式 Java 手册[M]，北京：机械工业出版社，2006.

2 John Zukowski（美）著，邱仲潘等译，Java 2 从入门到精通[M]，北京：电子工业出版社，2000.

3 Cay S. Horstmann、Gary Cornell（美）著，程峰、黄若波等译，Java 2 核心技术 卷 I：基础知识（第六版）[M]，北京：机械工业出版社，2004.