学生应知信息知识

Java知识手册（四）

狄登峰 主编

目　　录

通过 JavaSwing 看透 MVC 设计模式

一个好的用户界面(GUI)的设计通常可以在现实世界找到相应的表现。例如，如果在您的面前摆放着一个类似于电脑键盘按键的一个简单的按钮，然而就是这么简单的一个按钮，我们就可以看出一个 GUI 设计的规则，它由两个主要的部分构成，一部分使得它具有了按钮应该具有的动作特性，例如可以被按下。另外一部分则负责它的表现，例如这个按钮是代表了 A 还是 B。

看清楚这两点你就发现了一个很强大的设计方法，这种方法鼓励重用 reuse，而不是重新设计 redesign。你发现按钮都有相同的机理，你只要在按钮的顶上喷上不同的字母便能制造出“不同”的按钮，而不用为了每个按钮而重新设计一份图纸。这大大减轻了设计工作的时间和难度。

如果您把上述设计思想应用到软件开发领域，那么取得相似的效果一点都不让人惊奇。一个在软件开发领域应用的非常广泛的技术 Model/View/Controller(MVC)便是这种思想的一个实现。

这当然很不错，但是或许您又开始疑惑这和 java 基础类 JFC(JavaFoundationClass)中的用户界面设计部分(Swing)又有什么关系呢？好的，我来告诉你。

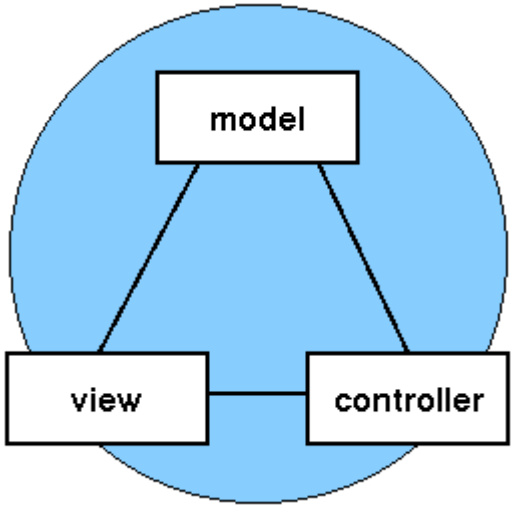
尽管 MVC 设计模式通常是用来设计整个用户界面

(GUI)的，JFC 的设计者们却独创性的把这种设计模式用来设计 Swing 中的单个的组件(Component)，例如表格 Jtable,树 Jtree,组合下拉列表框 JcomboBox 等等等等。这些组件都有一个 Model,一个 View，一个 Controller，而且，这些 model,view,controller 可以独立的改变，就是当组件正在被使用的时候也是如此。这种特性使得开发 GUI 界面的工具包显得非常的灵活。

好，来吧，让我来告诉你它是如何工作的。

MVC 设计模式

就象我刚才指出的一样，MVC 设计模式把一个软件组件区分为三个不同的部分，model,view,controller。



Model 是代表组件状态和低级行为的部分，它管理着自己的状态并且处理所有对状态的操作，model 自己本身并不知道使用自己的 view 和 controller 是谁，系统

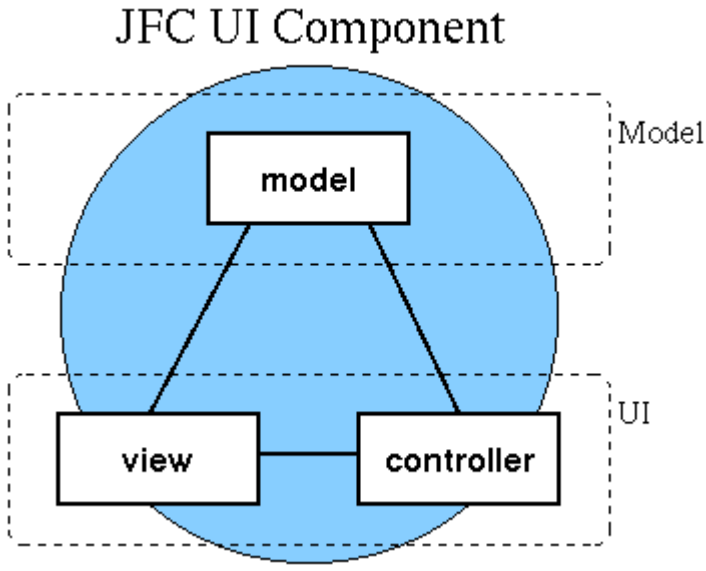
维护着它和 view 之间的关系，当 model 发生了改变系统还负责通知相应的 view。

View 代表了管理 model 所含有的数据的一个视觉上的呈现。一个 Model 可以有一个以上的 View,但是 Swing 中却很少有这样的情况。

Controller 管理着 model 和用户之间的交互的控制。它提供了一些方法去处理当 model 的状态发生了变化时的情况。

使用键盘上的按钮的例子来说明一下：Model 就是按钮的整个机械装置，View/Controller 就是按钮的表面部分。

下面的图解释了如何把一个 JFC 开发的用户界面分为 model,view,controller，注意，view/Controller 被合并到了一起，这是 MVC 设计模式通常的用法，它们提供了组件的用户界面(UI)。



用 Button **的例子详细说明**

为了更好的理解 MVC **设计模式和** Swing **用户界面组件之间的关系，让我们更加深入的进行分析。我将采用最常见的组件** button **来说明。**

我们从 model **来开始。**

Model

一个按钮的 model **所应该具备的行为由一个接口** ButtonModel **来完成。一个按钮** model **实例封装了其内部的状态，并且定义了按钮的行为。它的所有方法可以分为四类：**

1、**查询内部状态**

2、**操作内部状态**

3、添加和删除事件监听器

4、发生事件

其他的用户界面组件有它们各自的与组件相关的 Model，但是所有的组件 Model 都提供这四类方法。

View&Controller

上面的图中讲述一个按钮的 view/controller 由一个接口 ButtonUI 完成。如果一个类实现了这个接口，那么它将会负责创建一个用户界面，处理用户的操作。它的所有方法可以被分为三大类：

1、绘制 Paint

2、返回几何类型的信息

3、处理 AWT 事件

其他用户界面组件有他们自己的组件相关的 View/Controller，但是他们都提供上述三类方法。

程序员通常并不会直接和 model 以及 view/controller 打交道，他们通常隐藏于那些继承自 java.awt.Component 的组件里面了，这些组件就像胶水一样把 MVC 三者合三为一。也正是由于这些继承的组件对象，一个程序员可以很方便的混合使用 Swing 组件和 AWT 组件，然后，我们知道，Swing 组件有很多都是直接继承自相应的 AWT 组件，它能提供比 AWT 组件更加方便易用的功能，所以通常情况下，我们没有必要混

合使用两者。

一个实例

现在我们已经明白了 Java 类与 MVC 各个部分的对应关系，我们可以更加深入一点去分析问题了。下面我们将要讲述一个小型的使用 MVC 模式开发的例子。因为 JFC 十分的复杂，我只能把我的例子局限于一个用户界面组件里面（如果你猜是一个按钮的例子，那么你对了！）

让我们来看看这个例子的所有部分吧。

Button 类

最显而易见的开始的地方就是代表了按钮组件本省的代码，因为这个类是大部分程序员会接触的。

就像我前面提到的，按钮用户界面组件类实际上就是 model 和 view/controller 的之间的黏合剂。每个按钮组件都和一个 model 以及一个 controller 关联，model 定义了按钮的行为，而 view/controller 定义了按钮的表现。而应用程序可以在任何事件改变这些关联。让我们看看得以实现此功能的代码。

```
publicvoidsetModel(ButtonModelbuttonmodel)

{

if(this.buttonmodel!=null)
```

```
{
this.buttonmodel.removeChangeListener(buttonchangelistener);
this.buttonmodel.removeActionListener(buttonactionlistener);
buttonchangelistener=null;
buttonactionlistener=null;
}
this.buttonmodel=buttonmodel;
if(this.buttonmodel!=null)
{
buttonchangelistener=newButtonChangeListener();
buttonactionlistener=newButtonActionListener();
this.buttonmodel.addChangeListener(buttonchangelistener);
this.buttonmodel.addActionListener(buttonactionlistener);
}
updateButton();
}
```

```
publicvoidsetUI(ButtonUIbuttonui)
{
if(this.buttonui!=null)
{
this.buttonui.uninstallUI(this);
}
this.buttonui=buttonui;
if(this.buttonui!=null)
{
this.buttonui.installUI(this);
}
updateButton();
}
publicvoidupdateButton()
{
invalidate();
}
```

在进入下一节之前，你应该多花一些时间来仔细阅读一下 Button 类的源代码。

ButtonModel 类

ButtonModel 维护着三种类型的状态信息：是否被按下（pressed），是否“武装上了”（armed），是否被选择（selected）。它们都是 boolean 类型的值。

一个按钮被按下（pressed）是指当鼠标在按钮上面的时候，按下鼠标但是还没有松开鼠标按钮的状态，及时用户此时把鼠标拖拽到按钮的外面也没有改变这种状态。

一个按钮是否“武装了”（armed）是指按钮被按下，并且鼠标还在按钮的上面。

一些按钮还可能被选择（selected），这种状态通过重复的点击按钮取得 true 或者 false 的值。

下面的代码是状态 pressed 的一个缺省的实现。状态 armed 以及 selected 实现的代码与之类似。ButtonModel 类应该被继承，这样可以覆盖缺省的状态定义，实现有个性的按钮。

privatebooleanboolPressed=false;

publicbooleanisPressed()

```
{
returnboolPressed;
}
publicvoidsetPressed(booleanboolPressed)
{
this.boolPressed=boolPressed;
fireChangeEvent(newChangeEvent(button));
}
```

按钮的模型 buttonmodel **还负责通知其他对象（事件监听器）它们所感兴趣的事件。从下面的代买中我们可以看出当按钮的转台发生改变的时候就会发出一个** ChangeEvent。**下面就是代码：**

```
privateVectorvectorChangeListeners=newVector();
publicvoidaddChangeListener(ChangeListenerchangelistener)
{
vectorChangeListeners.addElement(changelistener);
```

```
}
publicvoidremoveChangeListener(ChangeListenerchangelistener)
{
vectorChangeListeners.removeElement(changelistener);
}
protectedvoidfireChangeEvent(ChangeEventchangeevent)
{
Enumerationenumeration=vectorChangeListeners.elements();
while(enumeration.hasMoreElements())
{
ChangeListenerchangelistener=(ChangeListener)enumeration.nextElem
changelistener.stateChanged(changeevent);
}
}
```

在进入下一节之前，你应该多花一些时间来仔细阅读一下 ButtonModel **类的源代码。**

ButtonUI 类

按钮的 view/controller 是负责构建表示层的。缺省情况下它仅仅是用背景色画一个矩形而已，他们的子类继承了他们并且覆盖了绘制的方法，使得按钮可以有许多不同的表现，例如 MOTIF，Windows95，Java 样式等等。

```
publicvoidupdate(Buttonbutton,Graphicsgraphics)
{
}
publicvoidpaint(Buttonbutton,Graphicsgraphics)
{
Dimensiondimension=button.getSize();
Colorcolor=button.getBackground();
graphics.setColor(color);
graphics.fillRect(0,0,dimension.width,dimension.height);
}
```

ButtonUI 类并不自己处理 AWT 事件，他们会使用一个定制的事件监听器把低级的 AWT 事件翻译为高级

的 Button 模型期望的语义事件。下面就是安装/卸载事件监听器的代码。

```
privatestaticButtonUIListenerbuttonuilistener=null;

publicvoidinstallUI(Buttonbutton)

{

button.addMouseListener(buttonuilistener);

button.addMouseMotionListener(buttonuilistener);

button.addChangeListener(buttonuilistener);

}

publicvoiduninstallUI(Buttonbutton)

{

button.removeMouseListener(buttonuilistener);

button.removeMouseMotionListener(buttonuilistener);

button.removeChangeListener(buttonuilistener);

}
```

View/Controller 实际上就是一些方法。他们不维护

任何自己的状态信息。因此，许多按钮的实例可以共享一个 ButtonUI 实例。ButtonUI 是通过在方面的参数列表里面加上按钮的引用来区分各个不同的按钮。

同样，希望你能多花一些时间来看看 ButtonUI 类，然后咱们进入下一节。

ButtonUIListener 类

ButtonUIListener 类可以帮助 Button 类去转变鼠标或者键盘的输入为对按钮模型的操作。这个监听器类实现了：MouseListener,MouseMotionListener,ChangeListener 接口，并且处理一下事件：

```
publicvoidmouseDragged(MouseEventmouseevent)
{
Buttonbutton=(Button)mouseevent.getSource();
ButtonModelbuttonmodel=button.getModel();
if(buttonmodel.isPressed())
{
if(button.getUI().contains(button,mouseevent.getPoint()))
{
```

```
buttonmodel.setArmed(true);
}
else
{
buttonmodel.setArmed(false);
}
}
}
publicvoidmousePressed(MouseEventmouseevent)
{
Buttonbutton=(Button)mouseevent.getSource();
ButtonModelbuttonmodel=button.getModel();
buttonmodel.setPressed(true);
buttonmodel.setArmed(true);
}
publicvoidmouseReleased(MouseEventmouseevent)
```