Observer 模式在 JavaGUI 中的分析与应用

The Analysis and Implementation of Observer Pattern in Java GUI

雷 镇 雷 蕾 周淑秋 王 华 (首都师范大学信息工程学院 100037)

摘要:设计模式描述了对象之间如何通信,而且彼此的数据模型和方法没有任何牵连。保持分离始终是优秀的面向对象编程的目标之一。本文以 Observer 模式为例,具体介绍了 Observer 模式问题的产生及解决 Java GUI 组件间的通信。最后以实例说明运用此设计模式带来软件扩充和复用的方便性。

关键词:设计模式 观察者模式 Java 图形用户界面

1 引言

在软件开发过程中实践面向对象思想时需要综合考虑封装、粒度、灵活性等多种因素,而这些因素往往是冲突的,因此如何进行权衡取舍找到合理的折衷是比较困难的,此时设计人员的经验就显得尤为重要。把经过证明的经验和技巧文档化将是一笔可资利用的财富。这种文档化的经验技巧就是设计模式(Design Pattern)。

设计模式是一种表达、记录和重用软件设计结构和设计经验的方法,它使得软件具备良好的可靠性、可扩展性、可复用性和可维护性。所有结构良好的面向对象软件体系结构中都包含了许多设计模式。设计模式的基本思想是将程序中的可变部分与不变部分进行分离,尽量减少对象之间的耦合度,从而某一个对象的修改,不会导致其他对象的变动,使得由于修改而带来的影响范围达到最小化。

根据目的的不同,模式可分为创建型模式、结构型模型和行为型模型三类。创建型模式与对象的创建有关;结构型模式处理和对象的组合,将一组对象组合成一个大的结构,例如复杂的用户界面;行为模式描述类或对象的交互和职责分配,定义对象间的通信和复杂程序中的流控。

2 Java 中的 Observer 模式

Observer 模式是经典设计模式中应用最为广泛也最为灵活多变的模式之一。Observer 模式是属于行为型的一种设计模式,它定义了对象间一对多的依赖关系,当一个目标(Subject)的状态发生变化,所有依赖他的观察者(Observer)应该能够得到通知并且能够自动更新,观察者设计模式同时解决了实现目标的类和实现观察者的类能够独立重用的问题。Observer 模式的结构如图 1 所示。

其中各元素的含义如下:

(1) Subject:被观察的目标的抽象接口,它提供对观察者(Observer)的注册、注销服务,Notify 方法通知 Observer 目标发生改变;

(2) Object:观察者的抽象接口,Update 方法是当得到 Subject 状态变化的通知后所要采取的动作;

(3) ConcreteSubject:Subject 的具体实现;

(4) ConcreteObserver:Observer 的具体实现

目标与两个观察者相互间的交互如图 2 所示。

Java 是 Sun 公司推出的新的一代面向对象程序设计语言,它具有简单、健壮、安全和跨平台的特点,支持封装、继承和多态。在 Java 语言中,不仅整个 AWT 事件模型都是基于 Observer 模式,而且在 java.util 包中还特别提供了 Observable 类和 Observer 接口支持这种模式。从名字上可以清楚的看出两者在 Observer 设计模式中分别扮演的角色:Observer 是观察者角色,Observable 是被观察目标(subject)角色。Observable 是一个封装 subject 基本功能的类,比如注册 observer (attach 功能),注销 observer(detatch 功能)等。

2.1 Observer 的方法

update(Observable subject,Object arg)监控 subject,当 subject 对象状态发生变化时 Observer 会有什么响应,arg 是传递给 Observable 的 notifyObservers 方

法的参数；

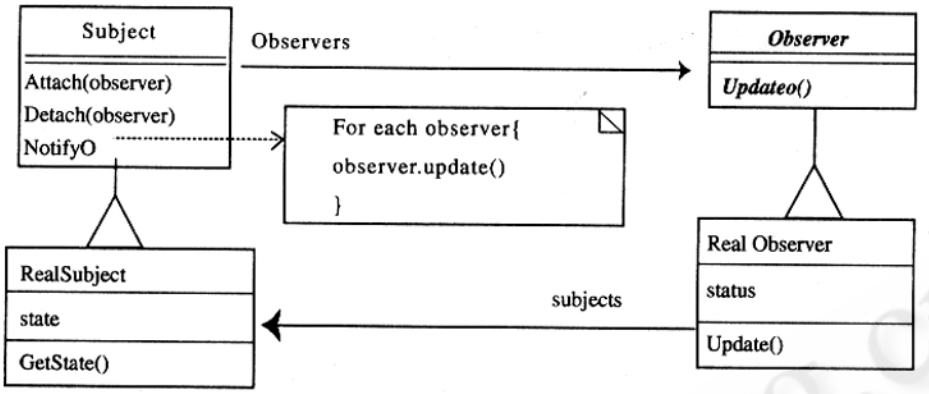

图 1　Observer 模式的结构

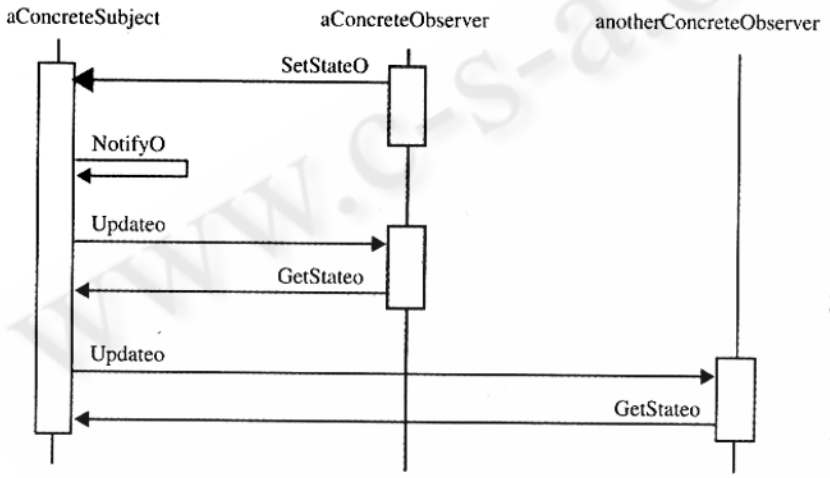

图 2　协作图

2.2 Observable 的方法：

addObserver (Observer observer) observer 向该 subject 注册自己

hasChanged( ) 检查该 subject 状态是否发生变化

setChanged( ) 设置该 subject 的状态为"已变化"

notifyObservers( ) 通知 observer 该 subject 状态发生变化

setChanged( ) 方法是设置 changed 的唯一入口，它的修饰符定义为 protected，就意味着通过定义 Observable 的对象，再设置 changed 属性将变得不可能。所以要想应用 observer 设计模式，必须继承 Observable 类。

```
public? void? notifyObservers( Object? arg )? {
//……省略……
for( inti = arrLocal. length -1;i > =0;i - - )
    ( ( Observer ) arrLocal[ i ] ).
update( this,arg ) ;
    }
```

很容易看出 notifyObservers ( ) 是找出所有已注册的 Observer，再逐个进行"通知"，通过调用 Observer 的 update (Observable, Object) 方法进行通知。update? 第一个参数是 this，我们同时还必须注意到，这段代码是 Observable 类里的代码。这就相当于是在一再强调，发出"通知"的，必须是 observable 自己 (Observable 类或者其派生类)，其他任何类都不行。这就意味着我们的 observable 类继承 Observable 类是必要的，因为如果不继承，而采用组合的话，将无法保证能传递好 this。

3　组件间的通信

以前做一个界面的时候常常会遇到这样的尴尬情况：希望保留各个独立的组件 (类)，但又希望它们之间能够相互通信。通过以上对 Observer 模式的分析可知它可以很好的解决这个矛盾。JDK? 1.1 以后新的事件模型是被成为"基于授权的事件模型"，也就是我们现在所熟悉的 Listener 模型，事件的处理不再由产生事件的对象负责，而由 Listener 负责，只有被注册过的 Listener 才能向组件传递事件动作。尤其在 Swing 组件中设计 MVC 结构时用到了 Observer 模式，众所周知，MVC 表示"模型 - 视图 - 控制器"，即"数据 - 表示逻辑 - 操作"，其中数据可以对应多种表示，这样视图就处在了 observer 的地位，而 model 则是 subject。大家所熟悉的 JTree 和 JTable 就是这种 MVC 结构。下面以一个例子来具体说明用 Observer 模式协调用户界面的各个 UI 组件间的通信。

Frame 包括左右两个 Panel，左边 Panel 中有一个树型选择组件，当用于选择发生变化时，右边 Panel 随之发生相应的变化。由于 Java 单根继承的原因，我们

不能同时继承 JPanel 和 Observable，但可以用对象的

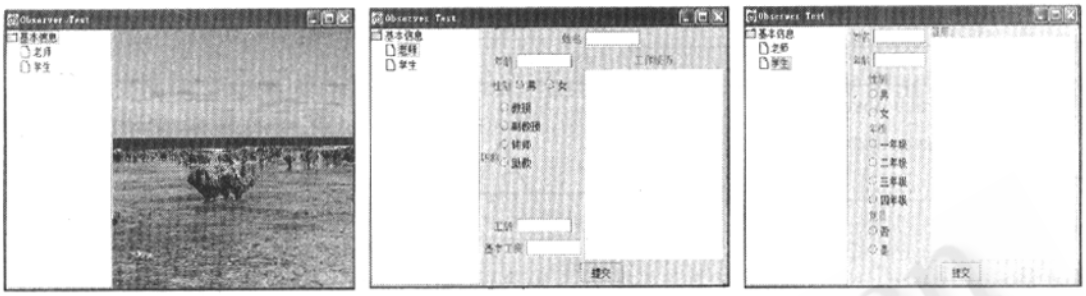

图 3　程序运行效果图

组合把一个 subject 放到我们的类当中，并通过 TreeSelectionListener 触发 subject 的 setChanged 方法，并通过 notifyObservers 方法通知 observer。这里 subject 是继承了 Observable 的一个感应器类。

```
public class Sensor extends Observable {
private Object data;
public Object getData( ) {
return data;
}
public void setData( Object data) {
this. data = data;
setChanged( ); //改变 Observable
}
}
```

我们为左边的 Panel 加入一个 Sensor，并向 Observable 注册 Observer。sensor. addObserver(observer)。当 TreeSelectionListener 触发 subject 的 setChanged 时，调用 sensor. setData 和 sensor. notifyObservers( )方法。右边的 Panel 是实现了 Observer 接口的类：

```
public class RightPanel extends JPanel implements Observer
```

左边 Panel 中向 sensor 注册的 Observer 就是 RightPanel 的一个实例，当 JTree 中用户的选择发生变化时 sensor 通知所有注册的 Observer，也就是 RightPanel 的实例，RightPanel 则调用 public void update (Observable subject, Object obj) 函数（当被观察对象发生变化时即调用该方法），在右边的 Panel 显示与左边选择对应的子 Panel。

程序运行的截图如图 3 所示。

Observer 模式将被观察的对象和观察者分别封装起来，使得它们可以各自独立的改变和复用，并任意增加和删除观察者。一个被观察的对象只知道它有一系列观察者，每个都符合一个具体的类，因此它们之间的耦合是最小的。

4　结束语

在软件开发过程中，经常在某一特点场合中遇到某些以前经常出现或感觉似曾相识的问题，直截了当地解决方案就是套用原有的已经过证明的解决方案，或参考别人成熟的思路来解决，久而久之，通过不断完善并文档化就形成了针对这种问题的模式。通过这种方式，将专家头脑中的经验和技巧固化下来，就可以一次又一次地使用已有地解决方案，无需重复相同地工作。基于对 Observer 模式的分析，我们可以清楚地看到此模式提供了一个完美的机制，能够在应用程序中的对象之间划定清晰的界限。在开发灵活的应用程序方面，设计模式是一个非常强大的工具。通过正确地使用此模式，在确保应用程序可演变性方面就会向前迈出一大步。Observer 模式在界面设计中应用广泛，当您的 UI 和业务要求随时间发生变化时，Observer 模式可确保能够简化您的工作。

参考文献

1 James W. Cooper. Java 设计模式[M]，中国电力出版社，2004。

2 E. Gamma et al. Design Patterns, Elements of reusable Object – Oriented Software[M], Addison – Wesley, 1995.

3 A. J. Riel. Object – Oriented Design Heuristics[M], Addison – wesley, 1996.

4 Steven John Metsker. 设计模式 Java 手册（影印版）[M]，北京中国电力出版社，2004。