西安交通大學

XI'AN JIAOTONG UNIVERSITY

Action & ActionListener 的实现及与其他事件处理机制的比较分析

课程： 软件系统设计与分析

姓名： 杨豪

班级： 软件 2101

时间： 2022 年 9 月

摘要

事件处理是面向对象分析与设计中最常用的功能之一. 本文先简单论述了事件、事件处理、事件处理机制的概念, 然后以 Java 语言分别实现了基本的事件处理机制: 同步调用、回调函数. 随后着重分析了 Action&ActionListener 及其父类 EventObject 等接口的 Java 内部实现, 尝试用代码实现了一个简单的 ActionListener 机制. 最后总结了以上三种事件处理机制的优缺点.

关键词：事件处理机制; Java; 面向对象; Listener.

目录

1 事件与事件处理机制概述

1.1 事件与事件处理

- 事件 (event): 在软件设计中, 事件是可由软件识别并处理的动作 (action) 或发生情形 (occurrence), 通常来自外部环境, 可以由系统、用户或其他方式生成或触发.
  - 事件的来源可能是用户据通过计算机的外围设备与软件交互 (eg. 通过键盘输入);
  - 软件也可以触发它自己的事件集进入事件循环 (eg. 完成通信任务)
- 事件处理 (event processing): 软件识别某一特定事件进行并对其进行特定的处理方式

1.2 事件处理机制的组成

一个事件机制一般有三个组成部分, 这里以生活中的事为例: 我一边打游戏一边烧水

- 事件源 (source): 即事件的发送者. 在上例中为水壶;
- 事件 (event): 事件源发出的一种信息或状态. 比如上例的警报声, 它代表着水开了;
- 事件侦听者 (listener): 对事件作出反应的对象. 比如上例中打游戏的我

在设计事件机制时一般把侦听者设计为一个函数, 当事件发送时, 调用此函数。比如上例中可以把倒水设计为侦听者.

1.3 简单的事件处理机制及其代码实现

1.3.1 方法调用

方法调用即调用方**等待被调用方执行完毕并返回后**再继续执行, 因此是一种单向的阻塞式同步调用.

该方法用代码实现起来很简洁.

Listing 1: **method call.java**

```java
public class eg1 {
    public static class Caller {
        String name;
        Caller(String s) {
            name = s;
        }
        public void call(Callee Y) {
            System.out.println("This is Caller " + name + ", I will call Callee " + Y.name + " .");
            Y.getcall();
        }
    }
    public static class Callee{
        String name;
        Callee(String s){
            name = s;
        }
        public void getcall(){
            System.out.println("This is Callee " + name + ", I was called.");
            try {
                Thread.sleep(5000);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args)
    {
        Caller x = new Caller("X");
        Callee y = new Callee("Y");
        long t = System.currentTimeMillis();
        x.call(y);
        t -= System.currentTimeMillis();
        System.out.println("Time cost is "+ (-t) + " ms");
    }
}
```

输出结果如下

```
            // 第三行等了约5s后才输出
            This is Caller X, I will call Callee Y .
```

```
3           This is Callee Y, I was called.
4           Time cost is 5020 ms
```

显然, 因为callee中的拖延, caller也被拖了更多的时间.

1.3.2 回调函数

回调函数: 回调是一种**双向**的调用模式, 设 A 和 B 是两个不同的类, 则 B 的该方法在被 A 调用, 完成后也会调用 A 的方法. 这样的调用需要定义一个 interface, 从而将 interface 作为参数传入.

这一思想实现起来比第一种略微麻烦一点:

Listing 2: **Callback.java**

```
1  public class eg2 {
2      public interface CallBack {
3          public void receiveNotice(String Message);
4      }
5
6      public static class Caller implements eg2.CallBack {
7          String name;
8
9          Caller(String s) {
10             name = s;
11         }
12
13         public void receiveNotice(String Message) {
14             System.out.println("Caller " + name + " received a Notice: " + Message);
15         }
16
17         public void call(Callee b) {
18             System.out.println("Caller " + name + " will assign Callee " + b.name + " a task.");
19             b.getcall(this);
20         }
21     }
22
23     public static class Callee {
24         String name;
25
26         Callee(String s) {
27             name = s;
```

```
28          }
29
30          public void getcall(CallBack cb) {
31              System.out.println("This is Callee " + name + ", I was called so I need to finish my
                    task");
32              try {
33                  Thread.sleep(5000);
34              } catch (InterruptedException e) {
35                  e.printStackTrace();
36              }
37              cb.receiveNotice("Callee " + name + " has done his task.");
38          }
39      }
40
41      public static void main(String[] args) {
42          Caller boss = new Caller("boss");
43          Callee worker = new Callee("worker");
44          boss.call(worker);
45      }
46  }
```

输出结果如下

```
1               // 第三行等了约5s后才输出
2               Caller boss will assign Callee worker a task.
3               This is Callee worker, I was called so I need to finish my task.
4               Caller boss received a Notice: Callee worker has done his task.
```

尽管该方法似乎并没有改进什么, 但我们可以使用 java 的多线程编程, 在`getcall`中插入如下的代码, 即以异步的形式实现, 则`Caller`类就可以做自己的事而不必等待回复.

```
1               new Thread(new Runnable()) {
2                   public void run() {
3                       try {
4                           sleep(5000);
5                           cb.receiveNotice("Callee " + name + " has done his task.");
6                       } catch (InterruptedException e) {
7                           e.printStackTrace();
8                       }
9                   }
10              }
```

2 Action & ActionListener 机制分析与实现

2.1 Java 中 EventObject 的基础概念

面向对象分析与设计中还有一种基本的事件处理机制, 即监听, 这种事件处理机制将事件的处理过程分成了四部分:

- Source：事件源，即触发事件的对象, 任何具有行为的对象可以是事件源;
- EventObject：事件对象, 即带有 EventSource 信息的事件对象, 也可以附带除 source 外的其它信息, 是对 EventSource 的包装, 起到传递事件信息的作用;
- EventListener：事件监听器，对该事件的处理, 一个监听器可以监视多个事件源;
- 注册: 将 Object 和 Listener 绑定.

2.1.1 EventObject

`EventObject`是定义于`java.util`的一个类, 直接继承于 Object, 是所有事件对象的父类. `ActionEvent`类的父类`AWTEvent`也是继承于它 (AWT 是 Java 的一个用于编写图形应用程序的开发包).

EventObject 提供了 3 个方法:

```
1   EventObject(Object source)  //source不能为null
2   Object getSource();         //返回事件源
3   String toString();          //返回该事件类名与事件源名
```

2.1.2 EventListener

EventListener 也是定义于 java.util 包、直接继承于 Object 的接口, 是所有监听器的父接口, 无方法.

2.2 Action & ActionListener 的原理

经过查阅资料、源码和自己总结, Action& ActionListener 的实现原理如下

- 定义过程

a. 为 EventSource 定义 EventObject;

– 组件 (如 JFram、Button 等) 触发 Action 等事件时 EventObject 的创建, 即定义过程的第一步, 是由 JVM 自动完成的.

b. 为 EventObject 定义 EventListener;

```
class MyActionListener implements ActionListener{
    @Override
    // 定义发生ActionEvent时要执行什么
    public void actionPerformed(ActionEvent e){
        // 一个Listener可能有多个Source，所以通常先判断Source
        if(e.getSource() == myButtonOne){...}
        else if(e.getSource()==myButtonTwo{...}
        ...
        // if(e instanceof MyEvent)
        // 不关注事件源，而关注事件类型时，通过事件类型判断
    }
}
```

c. 定义 EventSource 的类，指定添加 EventListener 的方法.

- 触发过程: 以 Button 组件为例

a. 通过 addActionListener 方法, 为 Button 添加了 ActionListener 接口实现类;

b. Button 类中定义的与点击相关的某个方法 A 就会把自身作为 source 创建一个事件对象，并将该事件对象传入 addActionListener 方法所添加的监听器中;

```
JButton myButtonOne=new JButton("按钮一");
JButton myButtonTwo=new JButton("按钮二");
MyActionListener listener=new MyActionListener();
myButtonOne.addActionListener(listener);
myButtonTwo.addActionListener(listerner);
```

c. 在 ActionListener 中**根据 source 的类型或实例选择**执行某段代码 (这里要用到前文中提到过的 EventObject 的 getSource() 方法).

2.3 Action & ActionListener 的代码实现

Listing 3: **Action.java**

```
import java.awt.*;
import java.util.*;

public class eg3 {
    static class MyEventObject extends EventObject {
        public MyEventObject(Object source) {
            super(source);    // 通过source构造event
        }
    }

    static class MyListener implements EventListener {
        public void onMyEvent(EventObject e) {
            if (e.getSource() instanceof EventSource) {
                EventSource temp = (EventSource) e.getSource();
                System.out.println("收到来自" + temp.getActioner() + "的事件!");
                CaughtCont ++ ;
            }
        }
    }

    public static MyListener listener = null;
    static int CaughtCont = 0; // 记录监听到的次数

    static class EventSource {
        private String who;
        Vector<MyListener> listeners = new Vector<MyListener>();

        public EventSource(String s) {
            this.who = s;
        }

        public String getActioner() {
            return who;
        }

        public void addMyEventListener(MyListener listener) {
            listeners.add(listener);
        }

        public void say(String words) {
```

```
41              System.out.println(this.getActioner() + ": " + words);
42              for (int i = 0; i < listeners.size(); i++) {
43                  MyListener listener = (MyListener) listeners.elementAt(i);
44                  listener.onMyEvent(new MyEventObject(this));
45              }
46          }
47      }
48
49      public static void main(String[] args) {
50          listener = new MyListener();
51          EventSource A = new EventSource("Alex");
52          EventSource B = new EventSource("Bobby");
53          A.addMyEventListener(listener);
54          A.say("今天天气不错");
55          B.say("适合出去走走");
56          System.out.println();
57          System.out.println("当前共收到" + CaughtCont + "次事件");
58          B.addMyEventListener(listener);
59          System.out.println();
60          A.say("今天天气不错");
61          B.say("适合出去走走");
62          System.out.println();
63          System.out.println("当前共收到" + CaughtCont + "次事件");
64      }
65  }
```

输出结果如下, 代码通过`implements EventListener`和`extends EventObject`实现了监听器并具有可监听多个源的特性.

```
Alex: 今天天气不错
收到来自Alex的事件!
Bobby: 适合出去走走

当前共收到1次事件

Alex: 今天天气不错
收到来自Alex的事件!
Bobby: 适合出去走走
收到来自Bobby的事件!

当前共收到3次事件
```

3 事件处理机制的优劣比较

- 同步调用
  - 优点：代码简洁，实现容易.
  - 缺点: 调用者必须等待被调用者返回后再继续; 难以扩展新的方法.
- 回调函数
  - 优点: 代码相对容易实现; 可利用多线程编程, 调用者无需等待;
  - 缺点: 多线程控制复杂，调试麻烦;
- ActionListener
  - 优点: 支持多线程编程; 代码逻辑比较简单, Java 内有成熟的功能库; 工程库父子继承关系明确, 便于扩展新的方法
  - 缺点: 比较依赖编程语言写好的框架, 如果完全自己实现会比较麻烦

参考文献

[1] 黄文海.Java 多线程编程实战指南（设计模式篇）.[M]. 北京: 电子工业出版社,2006.

[2] action 与 actionlistener 的关系.[DB/OL], https://zhuanlan.zhihu.com/p/65192541,2009.8.21.

[3] Java——事件处理机制.[DB/OL], https://blog.csdn.net/qq_19865749/article/details/70184968,2017.4.7.

[4] Java ActionListener Interface.[DB/OL]. https://www.javatpoint.com/java-actionlistener,2021.

[5] Interface ActionListener java SE 17 & JDK 17 [DB/OL] https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/java/awt/event/ActionListener.html,2022.

A LaTeX 排版参考

本文中所用 LaTeX 代码参考自

- 【LaTeX】自用简洁模板（六）：学校作业
- LaTeX 里「添加程序代码」的完美解决方案