西安交通大學

XI'AN JIAOTONG UNIVERSITY

分析总报告

三种基本事件处理分析

课程名称：	软件系统分析与设计
姓名：	凌晨
学院：	软件学院
专业：	软件工程
学号：	**2214414320**

2023 年 9 月 17 日

西安交通大学实验报告

专业：软件工程
姓名：凌晨
学号：2214414320
日期：2023 年 9 月 17 日

一、 目的和要求

一般地，面向对象分析与设计中存在三种基本事件处理的机制，除了普通的方法调用外，常常也会用到回调函数，而 J2EE 中还提供了一种基于监听方式的事件处理机制，请查阅资料，对 Action 以及 ActionListener 的机制进行分析，完成一个分析示例。

同时，请将这三种方法或其它更多的事件处理方法在代码实现过程中的优劣进行比较和分析，并形成详细的分析总报告。

二、 基本事件处理机制

1. 普通的方法调用

(1) 概念

普通的方法调用又称“同步调用”，是一种阻塞式调用，是最基本的调用方式。即假设在对象 a 中的方法直接调用对象 b 的方法，这个时候程序会等待对象 b 的方法执行完返回结果之后才会继续往下走。

(2) 代码展示

下面是 JAVA 代码展示：

```java
// A.java
package sync;

import java.text.SimpleDateFormat;
import java.util.Date;

public class A {
    public static void main(String[] args) throws InterruptedException {
        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyy-MM-dd hh:mm:ss");
        System.out.println("这是A类中的a方法,现在的时间是:" + sdf.format(date));
        B objectB = new B();
        objectB.b();
        System.out.println("over!");
    }
}

// B.java
package sync;
```

```java
import java.text.SimpleDateFormat;
import java.util.Date;

public class B {
    public void b() throws InterruptedException {
        Thread.sleep(5000);
        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyy-MM-dd hh:mm:ss");
        System.out.println("这是B类中的b方法,现在的时间是:" + sdf.format(date));
    }
}
```

可以看到主程序在 A 类中，先执行打印 A 类中的 a 方法，然后再执行 B 类中的 b 方法，A 类暂停，等待 B 类中的 b 方法执行完毕后，再执行 *System.out.println*(”*over*!”);

不难看出，该程序的执行结果如下：

```
这是 A 类中的 a 方法，现在的时间是:2023-09-10 11:03:35
这是 B 类中的 b 方法, 现在的时间是:2023-09-10 11:03:40
over!
```

执行结果说明分析正确！

2. 异步

(1) 概念

在学习回调函数前，我们需要先简单地了解什么是异步。异步是相对于同步的概念而存在的。对于同步我们已经有了十分清楚的认知，正如刚刚同步调用理解的一样，表现为图 2：

简单理解为就是只能一件事情一件事情的完成。

而异步则表现为多个事情同时开工！表现为图 1：

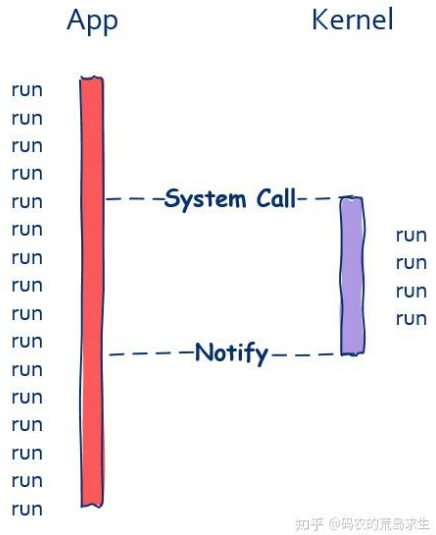

图 1: 异步

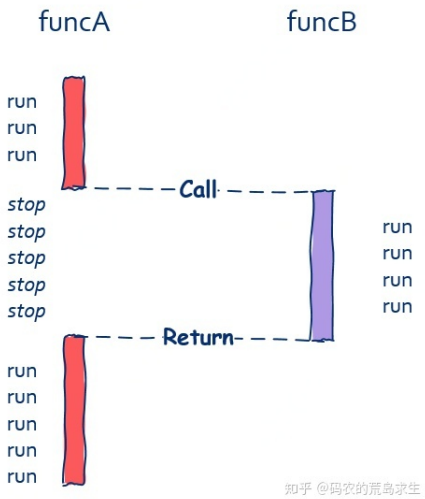

图 2: 同步

(2) 代码展示

下面是 JAVA 代码展示：

```
import java.util.concurrent.Callable;

class Task implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        System.out.println("子线程在进行计算");
        Thread.sleep(3000);
        int sum = 0;
        for (int i = 0; i < 100; i++)
            sum += i;
        System.out.println("子线程完成计算");
        return sum;
    }
}

import java.util.concurrent.*;

public class CallableFuture {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        Task task = new Task();
        Future<Integer> result = executor.submit(task);
        executor.shutdown();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        System.out.println("主线程在执行任务");
        try {
            System.out.println("task运行结果"+result.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        catch (CancellationException e) {
            System.out.println("子线程已经取消任务");
        }

        System.out.println("所有任务执行完毕");
    }
}
```

结果如下：

```
子线程在进行计算
主线程在执行任务
子线程完成计算
task 运行结果：4950
所有任务执行完毕
```

代码讲解：

- Task 是实现接口 Callable，相当于一个子任务，完成的任务较简单，不赘述.
- CallableFuture 相当于主程序，给线程派发任务，同时主程序完成自己的事情，等待线程返回结果。

结果而言，代码讲解准确！

3. 回调函数

(1) 回调

来自维基百科的对回调（Callback）的解析：In computer programming, a callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time. This execution may be immediate as in a synchronous callback, or it might happen at a later time as in an asynchronous callback. 也就是说，把一段可执行的代码像参数传递那样传给其他代码，而这段代码会在某个时刻被调用执行，这就叫做回调。如果代码立即被执行就称为同步回调，如果在之后晚点的某个时间再执行，则称之为异步回调。关于同步和异步，前面已经详细讲述。

(2) 回调函数

来自 Stack Overflow 某位大神简洁明了的表述：A ”callback” is any function that is called by another function which takes the first function as a parameter。也就是说，函数 F1 调用函数 F2 的时候，函数 F1 通过参数给函数 F2 传递了另外一个函数 F3 的指针，在函数 F2 执行的过程中，函数 F2 调用了函数 F3，这个动作就叫做回调（Callback），而先被当做指针传入、后面又被回调的函数 F3 就是回调函数。上述过程体现为下图：

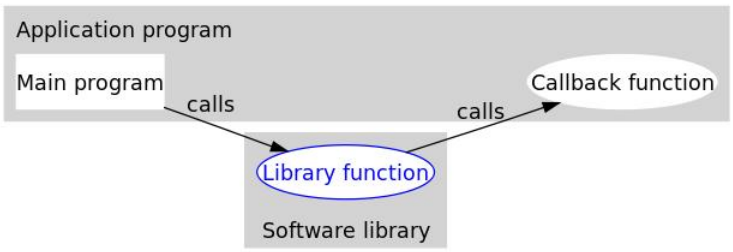

图 3: 回调函数

(3) 代码展示

由于 JAVA 对于指针的概念比较模糊，我们采用 C 语言进行回调函数的讲解, 代码如下：

```c
#include<stdio.h>

int Callback_1() // Callback Function 1
{
    printf("Hello, this is Callback_1 ");
    return 0;
}

int Callback_2() // Callback Function 2
{
    printf("Hello, this is Callback_2 ");
    return 0;
}

int Callback_3() // Callback Function 3
{
    printf("Hello, this is Callback_3 ");
    return 0;
}

int Handle(int (*Callback)())
{
    printf("Entering Handle Function. ");
    Callback();
    printf("Leaving Handle Function. ");
}

int main()
{
    printf("Entering Main Function. ");
    Handle(Callback_1);
    Handle(Callback_2);
    Handle(Callback_3);
    printf("Leaving Main Function. ");
    return 0;
}
```

结果如下：

```
Entering Main Function.
Entering Handle Function.
Hello, this is Callback_1
Leaving Handle Function.
Entering Handle Function.
Hello, this is Callback_2
Leaving Handle Function.
Entering Handle Function.
Hello, this is Callback_3
Leaving Handle Function.
Leaving Main Function.
```

可以看到，Handle() 函数里面的参数是一个指针，在 main() 函数里调用 Handle() 函数的时候，给它传入了函数 $Callback\_1()/Callback\_2()/Callback\_3()$ 的函数名，这时候的函数名就是对应函数的指针，也就是说，回调函数其实就是函数指针的一种用法。

4. 观察者模式

(1) 概念

观察者（Observer）模式的定义：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。观察者模式的主要角色如下。

- 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。

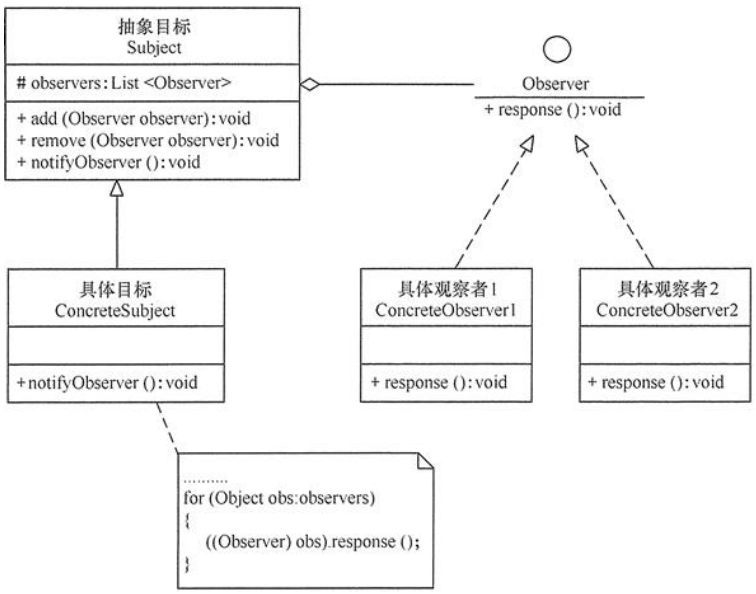

图 4: 观察者模式 UML

(2) 代码展示

```
public abstract class Observable {
    //定义一个观察者数组
    private Vector<Observer> obsVec = new Vector<Observer>();
    //增加一个观察者
    public void addObserver(Oberver obs)
    {
        this.obsVec.add(obs);
     }
    //删除一个观察者
    public void delObserver(Observer obs)
    {
        this.obsVec.remove(obs);
    }
    //通知所有观察者
    public void notifyObservers()
    {
        for (Observer obs:this.obsVec){
             obs.update();
         }
     }
}

public class ObservableImpl extends Observable {
    //具体的业务
    public void doSomething(){
        /*
        do something
```

```
        */
        super.notifyObservers();
      }
}

 public interface Observer{
    //更新方法
    public void update();
}

//创建被观察者
Observable subject = new Observable ();
//创建观察者
Observer obs = new Observer();
//添加到列表
subject.addObserver(obs);
//被观察者开始动作
subject.doSomething();
```

没有具体的例子，只是展示了观察者模式的框架。

5. 基于监听方式的事件处理机制

(1) 监听机制

事件监听机制可以理解为是一种观察者模式，但是并不是完全相同，有数据发布者（事件源）和数据接受者（监听器）；

在 Java 中，事件对象都是继承 java.util.EventObject 对象，事件监听器都是 java.util.EventListener 实例；

因此我们可以把模型简化为下图

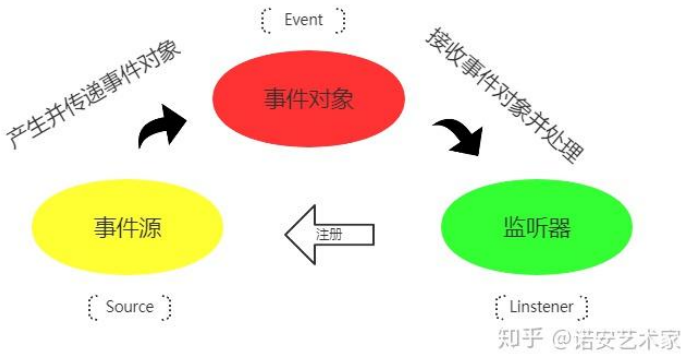

图 5: 监听机制示意图

(2) 代码展示

下面使用 JAVA 代码展示：

```
//创建事件接口
public interface Event {

    // 事件回调
    void callback();
}

//然后创建具体实现
public class ValueEvent implements Event {

    // 事件三要素：事件源、事件发生事件、事件消息
    private Object source;
    private LocalDateTime when;
    private String msg;

    public void setSource(Object source) {
        this.source = source;
    }

    public void setWhen(LocalDateTime when) {
        this.when = when;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public Object getSource() {
        return source;
    }

    public LocalDateTime getWhen() {
        return when;
    }

    public String getMsg() {
        return msg;
    }

    @Override
    public String toString() {
        return "ValueEvent{" +
               "source=" + source +
               ", when=" + when +
               ", msg='" + msg + '\'' +
               '}';
    }

    @Override
```

```
    public void callback() {
        System.out.println(this);
    }
}

//创建监听器接口:

public interface EventListener {

    // 触发事件
    void triggerEvent(Event event);
}
```

实现监听器:

```
public class ValueChangeListener implements EventListener {

    @Override
    public void triggerEvent(Event event) {
        // 调用事件回调方法
        event.callback();
    }
}
//最后编写事件源接口:

public interface EventSource {

    // 注册监听器
    void addListener(EventListener listener);

    // 通知所有监听器
    void notifyListener();
}
//实现事件源接口:

public class ValueSource implements EventSource {

    // 管理所有监听器
    private Vector<EventListener> listeners;

    private String msg;

    public ValueSource() {
        listeners = new Vector<>();
    }

    @Override
    public void addListener(EventListener listener) {
        listeners.add(listener);
    }
```

```
    @Override
    public void notifyListener() {
        for (EventListener listener : listeners) {
            ValueEvent event = new ValueEvent();
            event.setSource(this);
            event.setWhen(LocalDateTime.now());
            event.setMsg("更新数据:" + msg);
        }
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
        notifyListener();
    }
}
//编写测试代码:

public class Main {

    public static void main(String[] args) {
        ValueSource source = new ValueSource();
        source.addListener(new ValueChangeListener());
        source.setMsg("50");
    }
}
```

结果如下：

```
ValueEvent{source=com.wwj.spring.guanchazhe.click.ValueSource@1d81eb93
, when=2021-05-22T13:19:26.806, msg=' 更新数据:50'}
```

下面进行代码解释:

(1) 我们来仔细分析一下这个过程，首先我们创建了一个事件源：

(2) 它相当于观察者模式中的主题对象，也就是被观察者，当被观察者数据发生变化时，通知所有监听器进行处理，所以我们为其注册了一个监听器

(3) 此时我们修改事件源的数据, 就会执行 setMsg 方法

(4) 该方法又调用了 notifyListener 方法，通知所有监听器处理

(5) 在该方法中，首先需要创建事件，并设置事件源，也就是当前对象，设置事件发生时间和消息，最后调用监听器的事件处理方法

(6) 该方法又调用了事件的回调方法, 事件回调方法就输出了当前对象

以上就是整个事件监听机制的流程。

6. 发布-订阅模式

(1) 概念

基于一个事件（主题）通道，希望接收通知的对象 Subscriber 通过自定义事件订阅主题，被激活事件的对象 Publisher 通过发布主题事件的方式通知各个订阅该主题的 Subscriber 对象。因此发布订阅模式与观察者模式相比，发布订阅模式中有三个角色：

- 发布者 Publisher：发布事件，也称事件源。
- 事件调度中心 Event Channel：负责调度事件，可以理解为“上通下达”。
- 订阅者 Subscriber：订阅事件，获取事件。

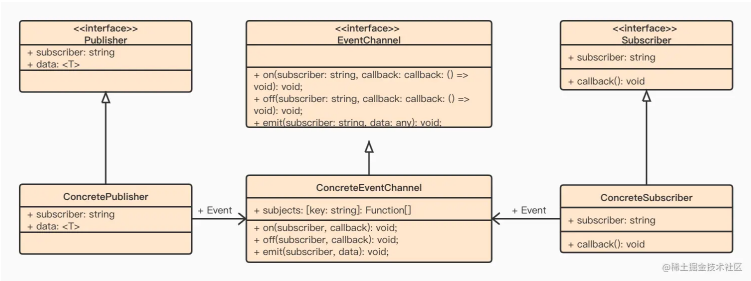

图 6: 发布-订阅模式 UML

(2) 代码展示

代码如下：

```
interface Publisher<T> {
  subscriber: string;
  data: T;
}

interface EventChannel<T> {
  on  : (subscriber: string, callback: () => void) => void;
  off : (subscriber: string, callback: () => void) => void;
  emit: (subscriber: string, data: T) => void;
}

interface Subscriber {
  subscriber: string;
  callback: () => void;
}

interface PublishData {
  [key: string]: string;
}

class ConcreteEventChannel<T> implements EventChannel<T> {
```

```
  // 初始化订阅者对象
  private subjects: { [key: string]: Function[] } = {};

  // 实现添加订阅事件
  public on(subscriber: string, callback: () => void): void {
    console.log(`收到订阅信息，订阅事件：${subscriber}`);
    if (!this.subjects[subscriber]) {
      this.subjects[subscriber] = [];
    }
    this.subjects[subscriber].push(callback);
  };

  // 实现取消订阅事件
  public off(subscriber: string, callback: () => void): void {
    console.log(`收到取消订阅请求，需要取消的订阅事件：${subscriber}`);
    if (callback === null) {
      this.subjects[subscriber] = [];
    } else {
      const index: number = this.subjects[subscriber].indexOf(callback);
      ~index && this.subjects[subscriber].splice(index, 1);
    }
  };

  // 实现发布订阅事件
  public emit (subscriber: string, data: T): void {
    console.log(`收到发布者信息，执行订阅事件：${subscriber}`);
    this.subjects[subscriber].forEach(item => item(data));
  };
}

class ConcretePublisher<T> implements Publisher<T> {
  public subscriber: string = "";
  public data: T;
  constructor(subscriber: string, data: T) {
    this.subscriber = subscriber;
    this.data = data;
  }
}

class ConcreteSubscriber implements Subscriber {
  public subscriber: string = "";
  constructor(subscriber: string, callback: () => void) {
    this.subscriber = subscriber;
    this.callback = callback;
  }
  public callback(): void { };
}
```

测试用例如下：

```
/* 运行示例 */
const pingan8787 = new ConcreteSubscriber(
  "running",
  () => {
    console.log("订阅者 pingan8787 订阅事件成功! 执行回调~");
  }
);

const leo = new ConcreteSubscriber(
  "swimming",
  () => {
    console.log("订阅者 leo 订阅事件成功! 执行回调~");
  }
);

const lisa = new ConcreteSubscriber(
  "swimming",
  () => {
    console.log("订阅者 lisa 订阅事件成功! 执行回调~");
  }
);

const pual = new ConcretePublisher<PublishData>(
  "swimming",
  {message: "pual 发布消息~"}
);

const eventBus = new ConcreteEventChannel<PublishData>();
eventBus.on(pingan8787.subscriber, pingan8787.callback);
eventBus.on(leo.subscriber, leo.callback);
eventBus.on(lisa.subscriber, lisa.callback);

// 发布者 pual 发布 "swimming"相关的事件
eventBus.emit(pual.subscriber, pual.data);
eventBus.off (lisa.subscriber, lisa.callback);
eventBus.emit(pual.subscriber, pual.data);
```

结果如下：

```
[LOG]: 收到订阅信息，订阅事件: running
[LOG]: 收到订阅信息，订阅事件: swimming
[LOG]: 收到发布者信息，执行订阅事件: swimming
[LOG]: 订阅者 leo 订阅事件成功! 执行回调~
[LOG]: 订阅者 lisa 订阅事件成功! 执行回调~
[LOG]: 收到取消订阅请求，需要取消的订阅事件: swimming
[LOG]: 收到发布者信息，执行订阅事件: swimming
[LOG]: 订阅者 leo 订阅事件成功! 执行回调~
```

可以看到，在添加了具体的发布者，订阅者后。订阅者点击订阅，若发布者发布事件，事件调度中心会触发回调。若订阅者取消订阅，则再也不会收到事件通知！

至此我们把几种常见的处理事件的机制全部介绍完毕！

三、 优缺点

1. 四种事件处理对比

只进行了普通方法调用，异步，回调函数，监听的优缺点对比。

(1) 代码编写难度

分析：

- 普通方法调用：代码书写简单-代码逻辑清晰-debug 轻松
- 异步：代码编写不规范可能导致程序有不确定性-异步编写难度较大
- 回调函数：了解接口作用，不必在意细节，提升代码可读性-同时编写不规范反而会降低可读性
- 监听：存在框架结构，编写难度小，规范性高

总结为下表：

处理机制	代码编写难度
普通方法调用	简单
异步	较难
回调函数	较难
监听	较简单

表 1: 代码编写难度

(2) 耦合度

分析：

- 普通方法调用：耦合度极高，一个函数对应一个解决方法，复用性差
- 异步：与同步调用一致
- 回调函数：降低代码耦合度，使其拓展功能成为可能
- 监听：降低代码耦合度，大幅拓展功能

总结为下表：

处理机制	耦合度
普通方法调用	高
异步	高
回调函数	可解耦
监听	可解耦

表 2: 耦合度

(3) 项目适应力-性能

分析：

- 普通方法调用：对于中小型的项目，尤其是单人开发，该方法与其他处理方式无明显不同，甚至因为减少了调用等过程，在性能表现上可能更好
- 异步：异步可以大幅提高项目的运行速度，使用户等待时间大幅减少。在绝大多规模不大项目中，只要程序员按照规范开发项目，保证逻辑合理，异步发挥出最大作用
- 回调函数：可以统一接口，使团队合作成为可能。但是，需要注意回调函数是一种破坏系统结构的设计，一般而言回调函数都会改变系统的运行轨迹，执行顺序和调用顺序，理解成本增加。
- 监听：与回调函数一致，可以统一接口，使团队合作成为可能。同时，由于已经开发出来的多种优秀框架，使监听处理事件机制可以胜任大规模项目。但是，值得注意，对于简单的任务仍需要按照监听机制来编写代码，导致性能下降

总结为下表：

处理机制	小项目	中项目	大项目
普通方法调用	好	较差	差
异步	好	较好	差
回调函数	较好	好	差
监听	差	好	好

表 3: 不同项目对应表现性能

2. 两种设计模式对比

我们进行观察者模式和发布-订阅模式两种设计模式的对比。
流程图如下:

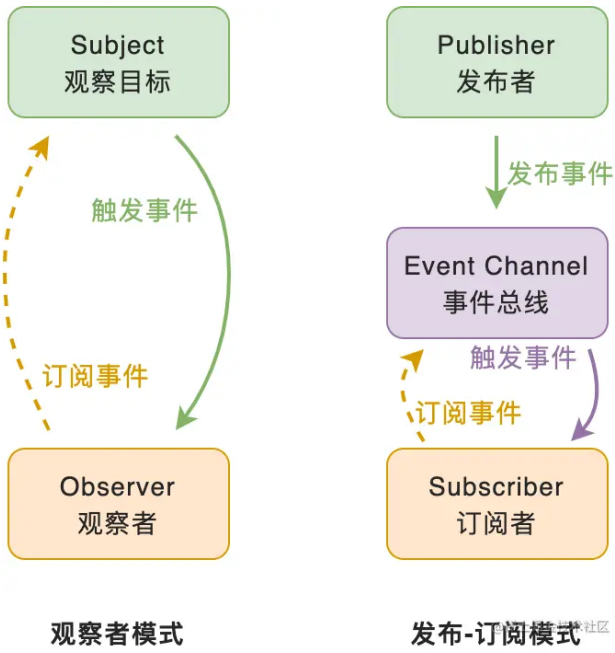

图 7: 流程图对比

可以看到，观察者模式没有通过事件总线，观察者和被观察者直接进行了联系，耦合度比发布-订阅者高。而发布-订阅模式的耦合度较低。观察者模式重点是被观察者，发布-订阅模式重点是发布订阅中心。

同时，发布-订阅模式有观察者模式没有的优点，就是可以通过发布订阅中心进行权限管理。

至此我们把几种常见的处理事件的优缺点介绍完毕！

四、 Action 以及 ActionListener 机制分析

1. 基本概念

(1) Action

Action 不是一个具体的类，而是一个接口，是 Swing 包提供的一种用来封装命令的并将其关联到多个事件源的机制。

动作 (Action) 是封装以下内容的一个对象:

- 命令的说明（一个文本字符串和一个可选的图标）
- 执行命令所需要的参数

该接口定义了如下方法：

```
void actionPerformed(ActionEvent event);
void setEnabled(boolean b);
boolean isEnabled();
void putValue(String key,Object value);
Object getValue(String key);
void addPropertyChangeListener(PropertyChangeListener listener);
```

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

- 第一个方法是 ActionListener 接口中的方法，实际上，Action 接口扩展了 ActionListener 接口，因此，任何需要 ActionListener 对象的地方可以使用 Action 对象.
- 接下来的两个方法允许启用或者禁用这个动作，并检查这个动作当前是否启用
- putValue 和 GetValue 方法允许存储和获取动作对象中的任意名/值对
- 最后两个方法能够让其他对象在动作对象的属性发生变化时得到通知

一般来说，最后两个方法使用的较多，较为重要。

(2) ActionListener

可以在官网上搜到相关定义——The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked. 翻译为：用于接收操作事件的侦听器接口。处理操作事件的类实现此接口，并且使用该类创建的对象使用组件的方法向组件注册。发生操作事件时，将调用该对象的方法。

因此该接口只有一个方法：

$$void \quad addActionListeneractionPerformed(ActionEvent \quad e)$$

method invoked when an action occurs.(该方法当事件发生时触发。)

2. 监听机制

监听机制主要由三部分构成：事件源，事件，监听器，机制如下图：

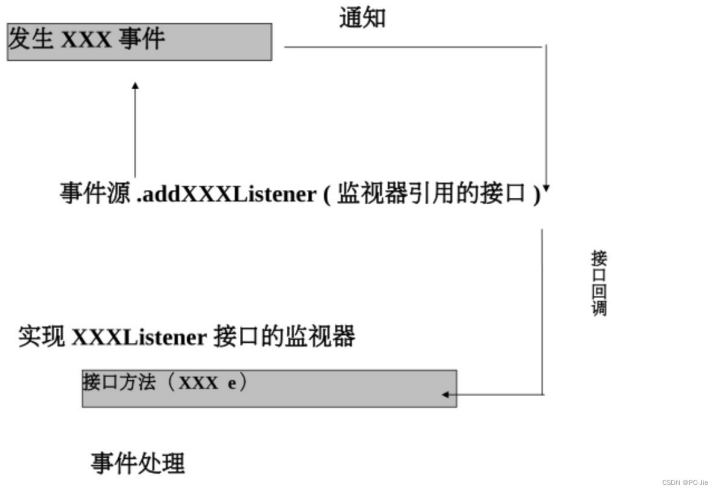

图 8: 监听机制

3. 代码展示

下面使用 JAVA 代码进行讲解，代码如下：

```
package technology;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyFirstActionListener extends JFrame {
  final static long serialVersionUID = 1L;
  Container container = getContentPane();
  JButton button = new JButton("点击我");

  class ButtonListener implements ActionListener {
    int x = 0;

    public void actionPerformed(ActionEvent arg0) {
      MyFirstActionListener.this.button.setText("我被点机了" + (++x) + "次");
    }
  }

  public MyFirstActionListener()
  {
    super("JFrame窗体");
    this.setBounds(200, 100, 200, 200);
    button.addActionListener(new ButtonListener());
    container.add(button);
    this.setVisible(true);
  }

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

展示效果如下：

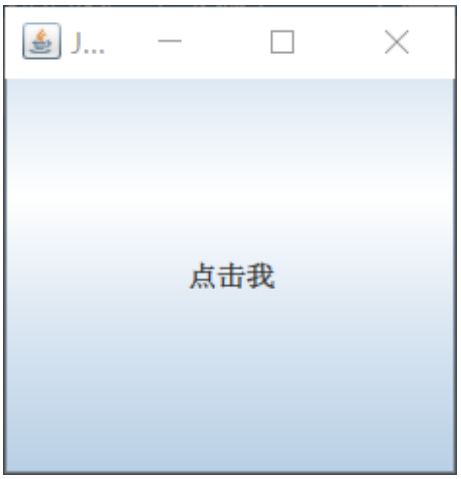

图 9: 未点击展示图

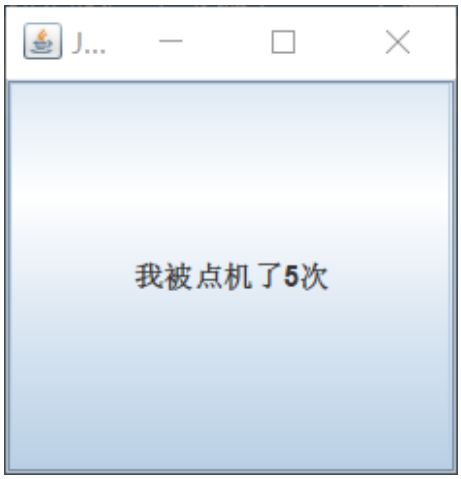

图 10: 已点击展示图

点击按钮时，触发事件，将点击次数增加 1，并且实时更改 swing 界面！

参考文献

[1] https://blog.csdn.net/qq_46311811/article/details/122444702

[2] https://blog.csdn.net/qq_39411709/article/details/126877210

[3] https://www.runoob.com/w3cnote/c-callback-function.html

[4] https://blog.csdn.net/weixin_52451652/article/details/125570819

[5] https://zhuanlan.zhihu.com/p/375905161

[6] https://blog.csdn.net/21aspnet/article/details/88172890

[7] https://zhuanlan.zhihu.com/p/263657828

[8] https://juejin.cn/post/6862803836781002760/