课程目标

1、掌握观察者模式和访问者模式的应用场景。
2、掌握观察者模式在具体业务场景中的应用。
3、了解访问者模式的双分派。
4、观察者模式和访问者模式的优、缺点。

内容定位

1、有Swing开发经验的人群更容易理解观察者模式。
2、访问者模式被称为最复杂的设计模式。

1. 观察者模式

观察者模式（Observer Pattern），又叫发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。定义一种一对多的依赖关系，一个主题对象可被多个观察者对象同时监听，使得每当主题对象状态变化时，所有依赖于它的对象都会得到通知并被自动更新。属于行为型模式。

> **原文**：Defines a one-to-many dependency relationship between objects so that each time an object's state changes，its dependent objects are notified and automatically updated.

观察者模式的核心是将观察者与被观察者解耦，以类似于消息/广播发送的机制联动两者，使被观察者的变动能通知到感兴趣的观察者们，从而做出相应的响应。

1.1. 观察者模式的应用场景

观察者模式在现实生活应用也非常广泛，比如：起床闹钟设置、APP角标通知、GPer生态圈消息通知、邮件通知、广播通知、桌面程序的事件响应等（如下图）。

APP 角标通知

起床闹钟设置

在软件系统中，当系统一方行为依赖于另一方行为的变动时，可使用观察者模式松耦合联动双方，使得一方的变动可以通知到感兴趣的另一方对象，从而让另一方对象对此做出响应。观察者模式适用于以下几种应用场景：

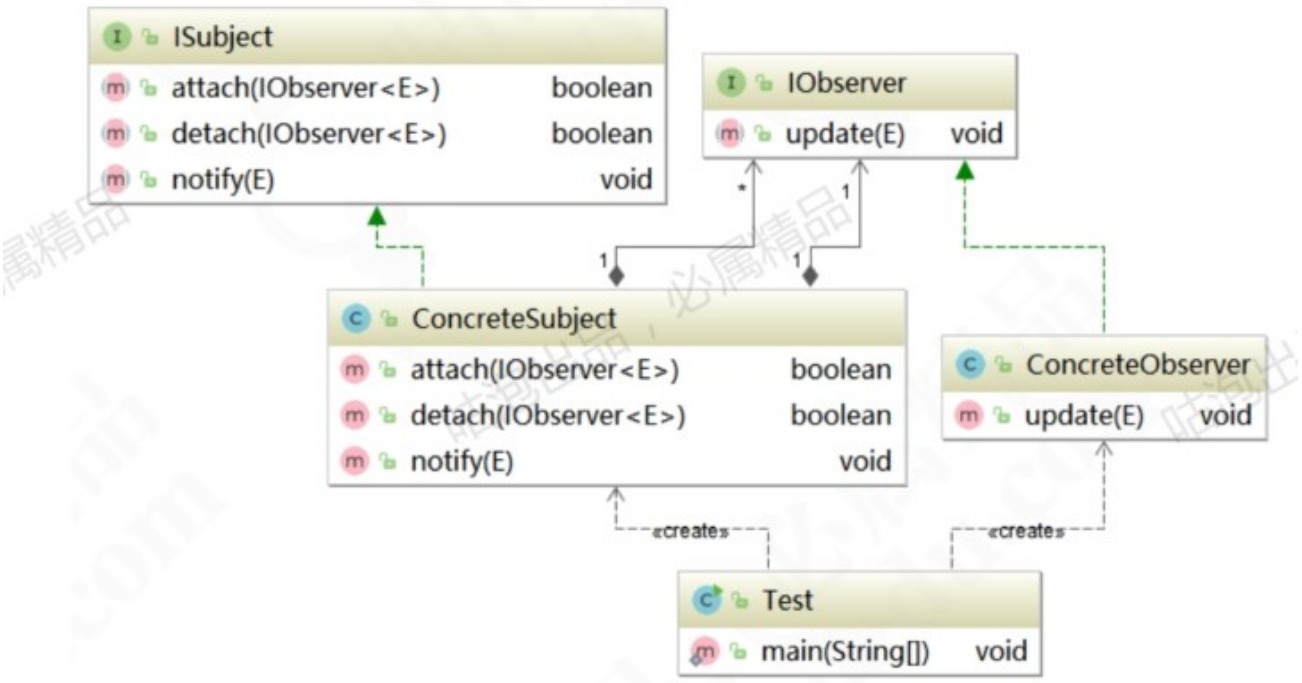
1、当一个抽象模型包含两个方面内容，其中一个方面依赖于另一个方面；

2、其他一个或多个对象的变化依赖于另一个对象的变化；

3、实现类似广播机制的功能，无需知道具体收听者，只需分发广播，系统中感兴趣的对象会自动接收该广播；

4、多层级嵌套使用，形成一种链式触发机制，使得事件具备跨域（跨越两种观察者类型）通知。

下面来看下观察者模式的通用UML类图：



从UML类图中，我们可以看到，观察者模式主要包含三种角色：

抽象主题（Subject）：指被观察的对象（Observable）。该角色是一个抽象类或接口，定义了增加、删除、通知观察者对象的方法；
具体主题（ConcreteSubject）：具体被观察者，当其内状态变化时，会通知已注册的观察者；
抽象观察者（Observer）：定义了响应通知的更新方法；
具体观察者（ConcrereObserver）：在得到状态更新时，会自动做出响应。

1.2. 观察者模式在业务场景中的应用

当小伙伴们在GPer生态圈中提问的时候，如果有设置指定老师回答，对应的老师就会收到邮件通知，这就是观察者模式的一种应用场景。我们有些小伙伴可能会想到MQ，异步队列等，其实JDK本身就提供这样的API。我们用代码来还原一下这样一个应用场景，创建GPer类：

```
package cn.sitedev.gper;

import java.util.Observable;

/**
 * JDK提供的一种观察者的实现方式，被观察者
 */
public class GPer extends Observable {
    private String name = "GPer生态圈";
    private static final GPer INSTANCE = new GPer();

    private GPer() {
    }
```

```
14
15      public static GPer getInstance() {
16          return INSTANCE;
17      }
18
19      public String getName() {
20          return name;
21      }
22
23      public void publishQuestion(Question question) {
24          System.out.println(question.getUserName() + "在" + this.name + "上提交了一个问是
25          setChanged();
26          notifyObservers(question);
27      }
28  }
```

创建问题Question类：

```
1  package cn.sitedev.gper;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Data;
5  import lombok.NoArgsConstructor;
6
7  @Data
8  @AllArgsConstructor
9  @NoArgsConstructor
10 public class Question {
11     private String userName;
12     private String content;
13 }
```

创建老师Teacher类：

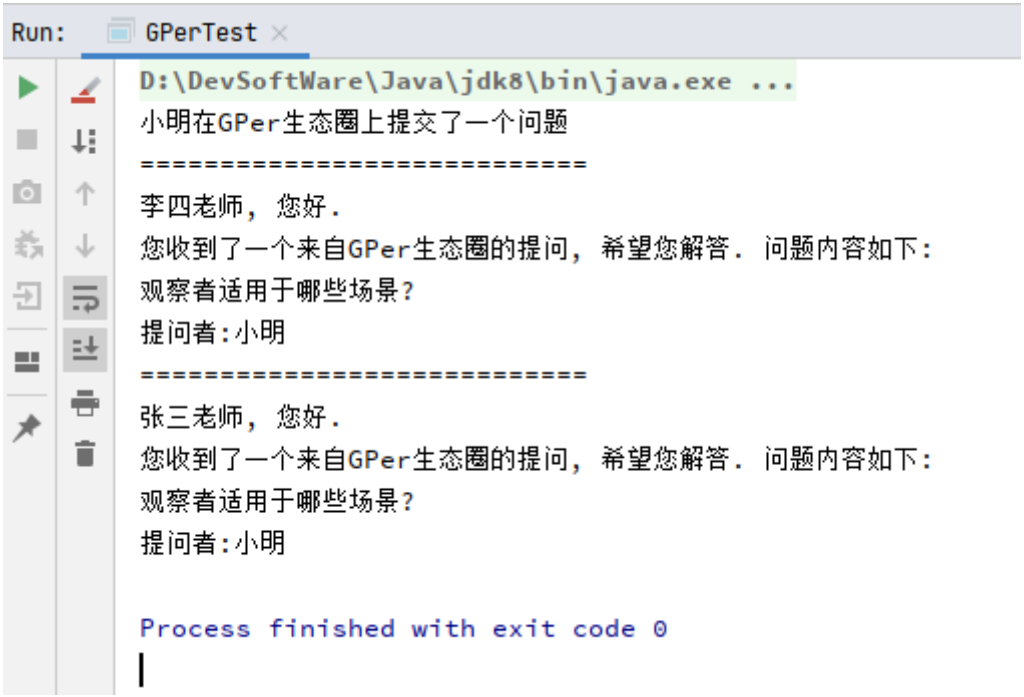
```
1  package cn.sitedev.gper;
2
3  import java.util.Observable;
4  import java.util.Observer;
5
6  /**
```

```
 * 观察者
 */
public class Teacher implements Observer {
    private String name;

    public Teacher(String name) {
        this.name = name;
    }

    @Override
    public void update(Observable o, Object arg) {
        GPer gPer = (GPer) o;
        Question question = (Question) arg;
        System.out.println("============================");
        System.out.println(this.name + "老师，您好.\n您收到了一个来自" + gPer.getName()
                "问题内容如下:\n" + question.getContent() + "\n提问者:" + question.getU
    }
}
```

客户端测试代码：

```
package cn.sitedev.gper;

public class GPerTest {
    public static void main(String[] args) {
        GPer gPer = GPer.getInstance();
        Teacher teacher = new Teacher("张三");
        Teacher teacher2 = new Teacher("李四");
        gPer.addObserver(teacher);
        gPer.addObserver(teacher2);

        // 业务逻辑代码
        Question question = new Question();
        question.setUserName("小明");
        question.setContent("观察者适用于哪些场景?");

        gPer.publishQuestion(question);
    }
}
```

运行结果：



1.3. 基于Guava API轻松落地观察者模式

给大家推荐一个实现观察者模式非常好用的框架。API使用也非常简单，举个例子，先引入maven依赖包：

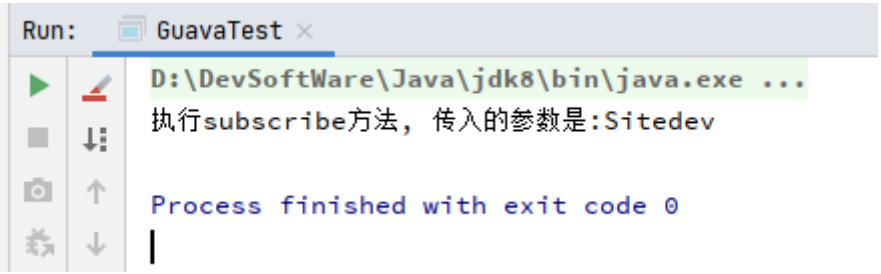
```
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>20.0</version>
        </dependency>
```

创建侦听事件GuavaEvent：

```
package cn.sitedev.guava;

import com.google.common.eventbus.Subscribe;

public class GuavaEvent {
    @Subscribe
    public void subscribe(String str) {
        // 业务逻辑
        System.out.println("执行subscribe方法，传入的参数是:" + str);
    }
}
```

客户端测试代码：

```
package cn.sitedev.guava;

import com.google.common.eventbus.EventBus;

public class GuavaTest {
    public static void main(String[] args) {
        EventBus eventBus = new EventBus();
        GuavaEvent guavaEvent = new GuavaEvent();
        eventBus.register(guavaEvent);
        eventBus.post("Sitedev");
    }
}
```



1.4. 使用观察者模式设计鼠标事件响应API

下面再来设计一个业务场景，帮助小伙伴更好的理解观察者模式。JDK源码中，观察者模式也应用非常多。例如java.awt.Event就是观察者模式的一种，只不过Java很少被用来写桌面程序。我们自己用代码来实现一下，以帮助小伙伴们更深刻地了解观察者模式的实现原理。首先，创建 EventListener 接口：

```
package cn.sitedev.mouseevent;

/**
 * 观察者抽象
 */
public interface EventListener {
}
```

创建Event类：

```java
package cn.sitedev.mouseevent;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;

import java.lang.reflect.Method;

/**
 * 标准事件源格式的定义
 */
@Getter
@AllArgsConstructor
@NoArgsConstructor
public class Event {
    // 事件源，动作是由谁发出的
    private Object source;
    // 事件触发，要通知谁(观察者)
    private EventListener target;
    // 观察者给的回应
    private Method callback;
    // 事件的名称
    private String trigger;
    // 事件的触发事件
    private long time;

    public Event(EventListener target, Method callback) {
        this.target = target;
        this.callback = callback;
    }

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

    public Event setTrigger(String trigger) {
        this.trigger = trigger;
        return this;
    }

    public Event setTime(long time) {
        this.time = time;
```

```
        return this;
    }
}
```

创建EventContext类：

```
package cn.sitedev.mouseevent;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

/**
 * 被观察者的抽象
 */
public class EventContext {
    protected Map<String, Event> events = new HashMap<>();

    public void addListener(String eventType, EventListener target, Method callback) {
        events.put(eventType, new Event(target, callback));
    }

    public void addListener(String eventType, EventListener target) {
        try {
            this.addListener(eventType, target,
                    target.getClass().getMethod("on" + toUpperFirstCase(eventType), Eve
        } catch (NoSuchMethodException e) {
            return;
        }
    }

    private String toUpperFirstCase(String eventType) {
        char[] chars = eventType.toCharArray();
        chars[0] -= 32;
        return String.valueOf(chars);
    }

    private void trigger(Event event) {
        event.setSource(this);
        event.setTime(System.currentTimeMillis());
```

```
36
37          try {
38              if (event.getCallback() != null) {
39                  // 用反射调用回调函数
40                  event.getCallback().invoke(event.getTarget(), event);
41              }
42          } catch (IllegalAccessException e) {
43              e.printStackTrace();
44          } catch (InvocationTargetException e) {
45              e.printStackTrace();
46          }
47      }
48
49      protected void trigger(String trigger) {
50          if (!this.events.containsKey(trigger)) {
51              return;
52          }
53          this.trigger(this.events.get(trigger).setTrigger(trigger));
54      }
55  }
```

创建MouseEventType接口：

```
1  package cn.sitedev.mouseevent;
2
3  public interface MouseEventType {
4      // 单击
5      String ON_CLICK = "click";
6
7      // 双击
8      String ON_DOUBLE_CLICK = "doubleClick";
9
10     // 弹起
11     String ON_UP = "up";
12
13     // 按下
14     String ON_DOWN = "down";
15
16     // 移动
17     String ON_MOVE = "move";
18
```

```
    // 滚动
    String ON_WHEEL = "wheel";

    // 悬停
    String ON_OVER = "over";

    // 失焦
    String ON_BLUR = "blur";

    // 获焦
    String ON_FOCUS = "focus";

}
```

创建 Mouse类：

```
package cn.sitedev.mouseevent;

/**
 * 具体的被观察者
 */
public class Mouse extends EventContext {
    public void click() {
        System.out.println("调用单击方法");
        this.trigger(MouseEventType.ON_CLICK);
    }

    public void doubleClick() {
        System.out.println("调用双击方法");
        this.trigger(MouseEventType.ON_DOUBLE_CLICK);
    }

    public void up() {
        System.out.println("调用弹起方法");
        this.trigger(MouseEventType.ON_UP);
    }

    public void down() {
        System.out.println("调用按下方法");
        this.trigger(MouseEventType.ON_DOWN);
    }
```

```
26
27      public void move() {
28          System.out.println("调用移动方法");
29          this.trigger(MouseEventType.ON_MOVE);
30      }
31
32      public void wheel() {
33          System.out.println("调用滚动方法");
34          this.trigger(MouseEventType.ON_WHEEL);
35      }
36
37      public void over() {
38          System.out.println("调用悬停方法");
39          this.trigger(MouseEventType.ON_OVER);
40      }
41
42      public void blur() {
43          System.out.println("调用失焦方法");
44          this.trigger(MouseEventType.ON_BLUR);
45      }
46
47      public void focus() {
48          System.out.println("调用获焦方法");
49          this.trigger(MouseEventType.ON_FOCUS);
50      }
51
52
53  }
```

创建回调方法MouseEventListener类：

```
1  package cn.sitedev.mouseevent;
2
3  /**
4   * 观察者
5   */
6  public class MouseEventListener implements EventListener {
7      public void onClick(Event event) {
8          System.out.println("==========触发鼠标单击事件============\n" + event);
9      }
10
```

```
    public void onDoubleClick(Event event) {
        System.out.println("==========触发鼠标双击事件==========\n" + event);
    }

    public void onUp(Event event) {
        System.out.println("==========触发鼠标弹起事件==========\n" + event);
    }

    public void onDown(Event event) {
        System.out.println("==========触发鼠标按下事件==========\n" + event);
    }

    public void onMove(Event event) {
        System.out.println("==========触发鼠标移动事件==========\n" + event);
    }

    public void onWheel(Event event) {
        System.out.println("==========触发鼠标滚动事件==========\n" + event);
    }

    public void onOver(Event event) {
        System.out.println("==========触发鼠标悬停事件==========\n" + event);
    }

    public void onBlur(Event event) {
        System.out.println("==========触发鼠标失焦事件==========\n" + event);
    }

    public void onFocus(Event event) {
        System.out.println("==========触发鼠标获焦事件==========\n" + event);
    }
}
```

客户端测试代码：

```
package cn.sitedev.mouseevent;

public class MouseEventTest {
    public static void main(String[] args) {
        MouseEventListener listener = new MouseEventListener();

```

```
        Mouse mouse = new Mouse();
        mouse.addListener(MouseEventType.ON_CLICK, listener);
        mouse.addListener(MouseEventType.ON_MOVE, listener);

        mouse.click();
        mouse.move();
        mouse.blur();
        mouse.focus();
    }
}
```

1.5. 观察者模式在源码中的应用

Spring 中的ContextLoaderlistener实现了ServletContextlistener接口，ServletContextListener接口又继承了EventListener，在JDK中EventListener有非常广泛的应用。

我们可以看一下源代码，ContextLoaderListener：

```
public class ContextLoaderListener extends ContextLoader implements ServletContextListe

    public ContextLoaderListener() {
    }

    public ContextLoaderListener(WebApplicationContext context) {
        super(context);
    }

    @Override
    public void contextInitialized(ServletContextEvent event) {
```

```
        initWebApplicationContext(event.getServletContext());
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        closeWebApplicationContext(event.getServletContext());
        ContextCleanupListener.cleanupAttributes(event.getServletContext());
    }

}
```

ServletContextListener接口源码如下：

```
public interface ServletContextListener extends EventListener {

    public void contextInitialized(ServletContextEvent sce);

    public void contextDestroyed(ServletContextEvent sce);
}
```

EventListener 接口源码如下：

```
public interface EventListener {
}
```

1.6. 观察者模式的优缺点

优点：
1、观察者和被观察者是松耦合（抽象耦合）的，符合依赖倒置原则；
2、分离了表示层（观察者）和数据逻辑层（被观察者），并且建立了一套触发机制，使得数据的变化可以响应到多个表示层上；
3、实现了一对多的通讯机制，支持事件注册机制，支持兴趣分发机制，当被观察者触发事件时，只有感兴趣的观察者可以接收到通知。

缺点：
1、如果观察者数量过多，则事件通知会耗时较长；
2、事件通知呈线性关系，如果其中一个观察者处理事件卡壳，会影响后续的观察者接收该事件；
3、如果观察者和被观察者之间存在循环依赖，则可能造成两者之间的循环调用，导致系统崩溃。

2. 访问者模式

访问者模式（Visitor Pattern）是一种将数据结构与数据操作分离的设计模式。是指封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。属于行为型模式。

> **原文**：Represent an operation to be performed on the elements of an object structure.Visitor lets you define a new operation without changing the classes of the elements on which it operates.

访问者模式被称为最复杂的设计模式，并且使用频率不高，设计模式的作者也评价为：大多情况下，你不需要使用访问者模式，但是一旦需要使用它时，那就真的需要使用了。访问者模式的基本思想是，针对系统中拥有固定类型数的对象结构（元素），在其内提供一个accept()方法用来接受访问者对象的访问。不同的访问者对同一元素的访问内容不同，使得相同的元素集合可以产生不同的数据结果。accept（）方法可以接收不同的访问者对象，然后在内部将自己（元素）转发到接收到的访问者对象的visit（）方法内。访问者内部对应类型的visit()方法就会得到回调执行，对元素进行操作。也就是通过两次动态分发（第一次是对访问者的分发accept（）方法,第二次是对元素的分发visit（）方法），才最终将一个具体的元素传递到一个具体的访问者。如此一来，就解耦了数据结构与操作，且数据操作不会改变元素状态。

访问者模式的核心是，解耦数据结构与数据操作，使得对元素的操作具备优秀的扩展性。可以通过扩展不同的数据操作类型（访问者）实现对相同元素集的不同的操作。

2.1. 访问者模式的应用场景

访问者模式在生活场景中也是非常当多的，例如每年年底的KPI考核，KPI考核标准是相对稳定的，但是参与KPI考核的员工可能每年都会发生变化，那么员工就是访问者。我们平时去食堂或者餐厅吃饭，餐厅的菜单和就餐方式是相对稳定的，但是去餐厅就餐的人员是每天都在发生变化的，因此就餐人员就是访问者。

参与 KPI 考核的人员

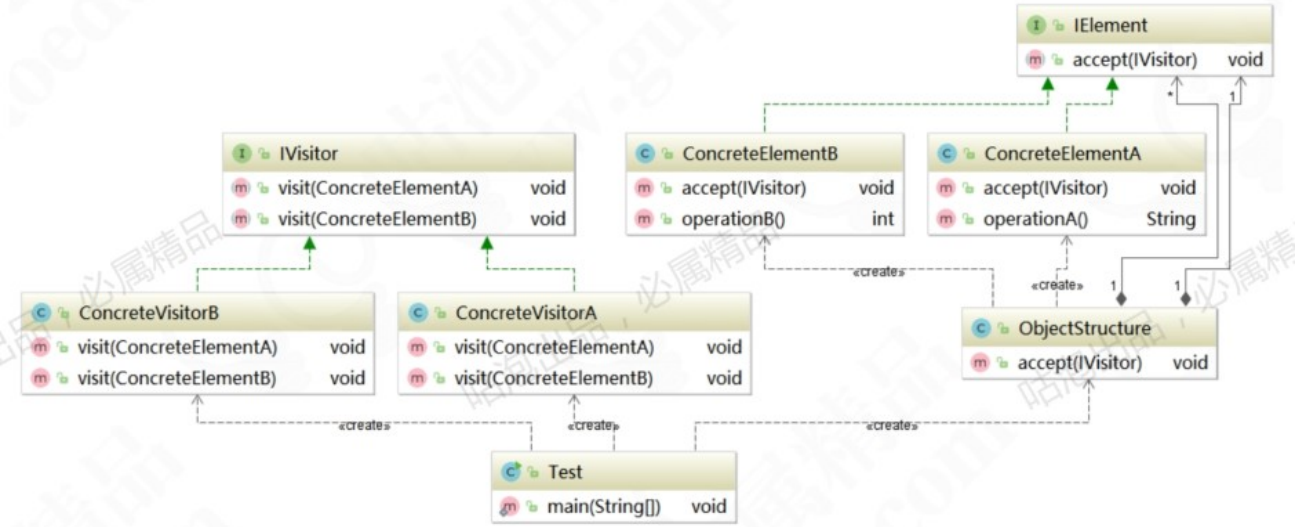

餐厅就餐人员

当系统中存在类型数目稳定（固定）的一类数据结构时，可以通过访问者模式方便地实现对该类型所有数据结构的不同操作，而又不会数据产生任何副作用（脏数据）。

简言之，就是对集合中的不同类型数据（类型数量稳定）进行多种操作，则使用访问者模式。

下面总结一下访问者模式的适用场景：
1、数据结构稳定，作用于数据结构的操作经常变化的场景；
2、需要数据结构与数据操作分离的场景；
3、需要对不同数据类型（元素）进行操作，而不使用分支判断具体类型的场景。

首先来看下访问者模式的通用UML类图：



从UML类图中，我们可以看到，访问者模式主要包含五种角色：

抽象访问者（Visitor）：接口或抽象类，该类地冠以了对每一个具体元素（Element）的访问行为visit()方法，其参数就是具体的元素（Element）对象。理论上来说，Visitor的方法个数与元素（Element）个数是相等的。如果元素（Element）个数经常变动，会导致Visitor的方法也要进行变动，此时，该情形并不适用访问者模式；
具体访问者（ConcreteVisitor）：实现对具体元素的操作；
抽象元素（Element）：接口或抽象类，定义了一个接受访问者访问的方法accept()，表示所有元素类型都支持被访问者访问；
具体元素（Concrete Element）：具体元素类型，提供接受访问者的具体实现。通常的实现都为：visitor.visit（this）；
结构对象（ObjectStruture）：该类内部维护了元素集合，并提供方法接受访问者对该集合所有元素进行操作。

2.2. 利用访问者模式实现KPI考核的场景

每到年底，管理层就要开始评定员工一年的工作绩效，员工分为工程师和经理；管理层有CEO和CTO。那么CTO关注工程师的代码量、经理的新产品数量；CEO关注的是工程师的KPI和经理的KPI以及新产品数量。

由于CEO和CTO对于不同员工的关注点是不一样的，这就需要对不同员工类型进行不同的处理。访问者模式此时可以派上用场了。

```
1 package cn.sitedev.kpi;
```

```java

import java.util.Random;

/**
 * 员工基类
 */
public abstract class Employee {
    protected String name;
    // 员工kpi
    protected int kpi;

    public Employee(String name) {
        this.name = name;
        this.kpi = new Random().nextInt(10);
    }

    // 核心方法，接受访问者的访问
    public abstract void accept(IVisitor vistor);
}
```

Employee类定义了员工基本信息及一个accept()方法，accept()方法表示接受访问者的访问，由具体的子类来实现。访问者是个接口，传入不同的实现类，可访问不同的数据。下面看看工程师Engineer类的代码：

```java
package cn.sitedev.kpi;

import java.util.Random;

// 工程师
public class Engineer extends Employee {
    public Engineer(String name) {
        super(name);
    }

    @Override
    public void accept(IVisitor vistor) {
        vistor.visit(this);
    }

    // 工程师一年的代码量
    public int getCodeLines() {
```

```
        return new Random().nextInt(100000);
    }
}
```

经理Manager类的代码：

```
package cn.sitedev.kpi;

import java.util.Random;

// 经理
public class Manager extends Employee {
    public Manager(String name) {
        super(name);
    }

    @Override
    public void accept(IVisitor vistor) {
        vistor.visit(this);
    }

    // 一年做的产品数量
    public int getProducts() {
        return new Random().nextInt(10);
    }
}
```

工程师是考核的是代码数量，经理考核的是产品数量，二者的职责不一样。也正是因为有这样的差异性，才使得访问模式能够在这个场景下发挥作用。Employee、Engineer、Manager这3个类型就相当于数据结构，这些类型相对稳定，不会发生变化。
然后将这些员工添加到一个业务报表类中，公司高层可以通过该报表类的showReport()方法查看所有员工的业绩，具体代码如下：

```
package cn.sitedev.kpi;

import java.util.LinkedList;
import java.util.List;

// 员工业务报表类
public class BusinessReport {
```

```
    private List<Employee> employees = new LinkedList<>();

    public BusinessReport() {
        employees.add(new Manager("经理1"));
        employees.add(new Engineer("工程师1"));
        employees.add(new Engineer("工程师2"));
        employees.add(new Engineer("工程师3"));
        employees.add(new Manager("经理2"));
        employees.add(new Engineer("工程师4"));
    }

    /**
     * 为访问者展示报表
     *
     * @param visitor 公司高层,如CEO, CTO
     */
    public void showReport(IVisitor visitor) {
        for (Employee employee : employees) {
            employee.accept(visitor);
        }
    }
}
```

下面看看访问者类型的定义，访问者声明了两个visit（ ）方法，分别对工程师和经理访问，具体代码如下：

```
package cn.sitedev.kpi;

public interface IVisitor {
    // 访问工程师类型
    void visit(Engineer engineer);

    // 访问经理类型
    void visit(Manager manager);
}
```

首先定义一个IVisitor 接口，该接口有两个visit（ ）方法，参数分别是Engineer、Manager，也就是说对于Engineer和Manager的访问会调用两个不同的方法，以此达到差异化处理的目的。这两个访问者具体的实现类为CEOVisitor类和CTOVisitor类，代码如下：

```
```

```java
package cn.sitedev.kpi;

// CEO 访问者
public class CEOVisitor implements IVisitor {
    @Override
    public void visit(Engineer engineer) {
        System.out.println("工程师:" + engineer.name + ", KPI:" + engineer.kpi);
    }

    @Override
    public void visit(Manager manager) {
        System.out.println("经理:" + manager.name + ", KPI:" + manager.kpi + ", 新产品数
    }
}
```

在CEO的访问者中，CEO关注工程师的KPI，经理的KPI和新产品数量，通过两个visitor()方法分别进行处理。如果不使用访问者模式，只通过一个visit()方法进行处理，那么就需要在这个visit（ ）方法中进行判断，然后分别处理，代码大致如下：

```java
package cn.sitedev.kpi;

public class ReportUtil {
    public void visit(Employee employee) {
        if (employee instanceof Manager) {
            Manager manager = (Manager) employee;
            System.out.println("经理:" + manager.name + ", KPI:" + manager.kpi + ", 新产
        } else if (employee instanceof Engineer) {
            Engineer engineer = (Engineer) employee;
            System.out.println("工程师:" + engineer.name + ", KPI:" + engineer.kpi);
        }
    }
}
```

这就导致了if-else逻辑的嵌套以及类型的强制转换，难以扩展和维护，当类型较多时，这个ReportUtil 就会很复杂。而使用访问者模式，通过同一个函数对不同对元素类型进行相应对处理，使结构更加清晰、灵活性更高。

再添加一个CTO的访问者类：

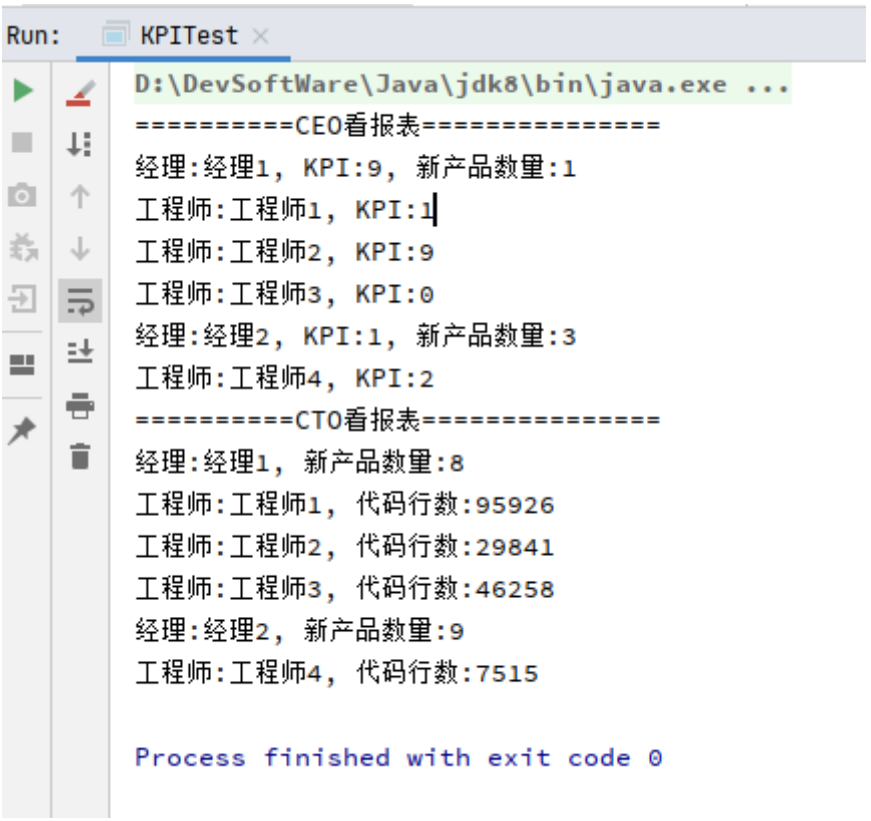
```java
package cn.sitedev.kpi;
```

```
2
3  // CTO 访问者
4  public class CTOVisitor implements IVisitor {
5      @Override
6      public void visit(Engineer engineer) {
7          System.out.println("工程师:" + engineer.name + ", 代码行数:" + engineer.getCode
8      }
9
10     @Override
11     public void visit(Manager manager) {
12         System.out.println("经理:" + manager.name + ", 新产品数量:" + manager.getProduc
13     }
14 }
```

重载的visit（ ）方法会对元素进行不同的操作，而通过注入不同的访问者又可以替换掉访问者的具体实现，使得对元素的操作变得更灵活，可扩展性更高，同时也消除了类型转换、if-else等“丑陋”的代码。下面是客户端代码：

```
1  package cn.sitedev.kpi;
2
3  public class KPITest {
4      public static void main(String[] args) {
5          // 构建报表
6          BusinessReport report = new BusinessReport();
7          System.out.println("==========CEO看报表==============");
8          report.showReport(new CEOVisitor());
9
10         System.out.println("==========CTO看报表==============");
11         report.showReport(new CTOVisitor());
12     }
13 }
```

运行结果如下：



在上述案例中，Employee扮演了Element角色，而Engineer和Manager都是ConcreteElement；CEOVisitor 和CTOVisitor都是具体的Visitor对象；而BusinessReport就是ObjectStructure。

访问者模式最大的优点就是增加访问者非常容易，我们从代码中可以看到，如果要增加一个访问者，只要新实现一个访问者接口的类，从而达到数据对象与数据操作相分离的效果。如果不实用访问者模式，而又不想对不同的元素进行不同的操作，那么必定需要使用if-else和类型转换，这使得代码难以升级维护。

我们要根据具体情况来评估是否适合使用访问者模式，例如，我们的对象结构是否足够稳定，是否需要经常定义新的操作，使用访问者模式是否能优化我们的代码，而不是使我们的代码变得更复杂。

2.3. 从静态分派到动态分派

变量被声明时的类型叫做变量的静态类型（Static Type），有些人又把静态类型叫做明显类型（Apparent Type）；而变量所引用的对象的真实类型又叫做变量的实际类型（Actual Type）。比如：

```
List list=null;
list=new Arraylist（）；
```

2.3.1. 静态分派

静态分派（Static Dispatch）就是按照变量的静态类型进行分派，从而确定方法的执行版本，静态分派在编译时期就可以确定方法的版本。而静态分派最典型的应用就是方法重载，来看下面这段代

码。

```
package cn.sitedev.staticdispatch;

public class Main {
    public void test(String string) {
        System.out.println("string");
    }

    public void test(Integer integer) {
        System.out.println("integer");
    }

    public static void main(String[] args) {
        String string = "1";
        Integer integer = 1;

        Main main = new Main();

        main.test(string);
        main.test(integer);
    }
}
```

在静态分派判断的时候，我们根据多个判断依据（即参数类型和个数）判断出了方法的版本，那么这个就是多分派的概念，因为我们有一个以上的考量标准。所以Java是静态多分派的语言。

2.3.2. 动态分派

对于动态分派，与静态相反，它不是在编译期确定的方法版本，而是在运行时才能确定。而动态分派最典型的应用就是多态的特性。举个例子，来看下面的这段代码。

```
package cn.sitedev.dynamicdispatch;

public class Main {
    public static void main(String[] args) {
        Person man = new Man();
        Person woman = new Woman();

        man.test();
        woman.test();
```

```
    }
}

interface Person {
    void test();
}

class Man implements Person {

    @Override
    public void test() {
        System.out.println("man");
    }
}

class Woman implements Person {
    @Override
    public void test() {
        System.out.println("woman");
    }
}
```

这段程序输出结果为依次打印男人和女人，然而这里的test（ ）方法版本，就无法根据 Man和Woman的静态类型去判断了，他们的静态类型都是Person接口，根本无从判断。显然，产生的输出结果，就是因为test()方法的版本是在运行时判断的，这就是动态分派。动态分派判断的方法是在运行时获取到Man和Woman的实际引用类型，再确定方法的版本，而由于此时判断的依据只是实际引用类型，只有一个判断依据，所以这就是单分派的概念，这时我们的考量标准只有一个，即变量的实际引用类型。相应的，这说明Java是动态单分派的语言。

2.4. 访问者模式中的伪动态双分派

通过前面分析，我们知道Java是静态多分派、动态单分派的语言。Java底层不支持动态的双分派。但是通过使用设计模式，也可以在Java 语言里实现伪动态双分派。在访问者模式中使用的就是伪动态双分派。所谓动态双分派就是在运行时依据两个实际类型去判断一个方法的运行行为，而访问者模式实现的手段是进行了两次动态单分派来达到这个效果。

还是回到前面的KPI考核业务场景当中，BusinessReport类中的 showReport()方法：

```
    public void showReport(IVisitor visitor) {
        for (Employee employee : employees) {
            employee.accept(visitor);
        }
```

```
    }
```

这里就是依据Employee和IVisitor 两个实际类型决定了showReport()方法的执行结果，从而决定了accept()方法的动作。

分析accept()方法的调用过程
1.当调用accept（ ）方法时，根据Employee的实际类型决定是调用Engineer 还是Manager的accept()方法。
2.这时accept（ ）方法的版本已经确定，假如是Engineer，它的accept（ ）方法是调用下面这行代码。

```
    public void accept(IVisitor vistor) {
        vistor.visit(this);
    }
```

此时的this是Engineer类型，所以对应的是IVisitor 接口的visit（Engineer engineer）方法，此时需要再根据访问者的实际类型确定 visit（ ）方法的版本，如此一来，就完成了动态双分派的过程。

以上的过程就是通过两次动态双分派，第一次对accept（ ）方法进行动态分派，第二次对访问者的visit（ ）方法进行动态分派，从而达到了根据两个实际类型确定一个方法的行为的效果。

而原本我们的做法，通常是传入一个接口，直接使用该接口的方法，此为动态单分派，就像策略模式一样。在这里，showReport()方法传入的访问者接口并不是直接调用自己的visit()方法，而是通过Employee的实际类型先动态分派一次，然后在分派后确定的方法版本里再进行自己的动态分派。

> 注意：这里确定accept（IVisitor visitor）方法是静态分派决定的，所以这个并不在此次动态双分派的范畴内，而且静态分派是在编译期就完成的，所以accept（IVisitor visitor）方法的静态分派与访问者模式的动态双分派并没有任何关系。动态双分派说到底还是动态分派，是在运行时发生的，它与静态分派有着本质上的区别，不可以说一次动态分派加一次静态分派就是动态双分派，而且访问者模式的双分派本身也是另有所指。

而this的类型不是动态确定的，你写在哪个类当中，它的静态类型就是哪个类，这是在编译期就确定的，不确定的是它的实际类型，请小伙伴也要区分开来。

2.5. 访问者模式在源码中的应用

首先来看JDK的NIO模块下的FileVisitor，它接口提供了递归遍历文件树的支持。这个接口上的方法表示了遍历过程中的关键过程，允许你在文件被访问、目录将被访问、目录已被访问、发生错误等等过程上进行控制；换句话说，这个接口在文件被访问前、访问中和访问后，以及产生错误的时候都有相应的钩子程序进行处理。

调用FileVisitor中的方法，会返回访问结果FileVisitResult对象值，用于决定当前操作完成后接下来该如何处理。FileVisitResult的标准返回值存放到FileVisitResult枚举类型中：

> FileVisitResult.CONTINUE：这个访问结果表示当前的遍历过程将会继续。
>
> FileVisitResult.SKIP_SIBLINGS：这个访问结果表示当前的遍历过程将会继续，但是要忽略当前文件/目录的兄弟节点。
>
> FileVisitResult.SKIP_SUBTREE：这个访问结果表示当前的遍历过程将会继续，但是要忽略当前目录下的所有节点。
>
> FileVisitResult.TERMINATE：这个访问结果表示当前的遍历过程将会停止。

```
public interface FileVisitor<T> {

    FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
        throws IOException;

    FileVisitResult visitFile(T file, BasicFileAttributes attrs)
        throws IOException;

    FileVisitResult visitFileFailed(T file, IOException exc)
        throws IOException;

    FileVisitResult postVisitDirectory(T dir, IOException exc)
        throws IOException;
}
```

通过它去遍历文件树会比较方便，比如查找文件夹内符合某个条件的文件或者某一天内所创建的文件，这个类中都提供了相对应的方法。我们来看一下它的实现其实也非常简单：

```
public class SimpleFileVisitor<T> implements FileVisitor<T> {

    protected SimpleFileVisitor() {
    }

    @Override
    public FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
        throws IOException
    {
        Objects.requireNonNull(dir);
        Objects.requireNonNull(attrs);
        return FileVisitResult.CONTINUE;
    }

```

```
    @Override
    public FileVisitResult visitFile(T file, BasicFileAttributes attrs)
        throws IOException
    {
        Objects.requireNonNull(file);
        Objects.requireNonNull(attrs);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(T file, IOException exc)
        throws IOException
    {
        Objects.requireNonNull(file);
        throw exc;
    }

    @Override
    public FileVisitResult postVisitDirectory(T dir, IOException exc)
        throws IOException
    {
        Objects.requireNonNull(dir);
        if (exc != null)
            throw exc;
        return FileVisitResult.CONTINUE;
    }
}
```

下面再来看访问者模式在Spring中的应用，Spring IoC中有个BeanDefinitionVisitor类，其中有一个visitBeanDefinition()方法，我们来看他的源码：

```
public class BeanDefinitionVisitor {

    @Nullable
    private StringValueResolver valueResolver;

    public BeanDefinitionVisitor(StringValueResolver valueResolver) {
        Assert.notNull(valueResolver, "StringValueResolver must not be null");
        this.valueResolver = valueResolver;
    }

```

```
    protected BeanDefinitionVisitor() {
    }

    public void visitBeanDefinition(BeanDefinition beanDefinition) {
        visitParentName(beanDefinition);
        visitBeanClassName(beanDefinition);
        visitFactoryBeanName(beanDefinition);
        visitFactoryMethodName(beanDefinition);
        visitScope(beanDefinition);
        if (beanDefinition.hasPropertyValues()) {
            visitPropertyValues(beanDefinition.getPropertyValues());
        }
        if (beanDefinition.hasConstructorArgumentValues()) {
            ConstructorArgumentValues cas = beanDefinition.getConstructorArgumentValues
            visitIndexedArgumentValues(cas.getIndexedArgumentValues());
            visitGenericArgumentValues(cas.getGenericArgumentValues());
        }
    }
    ...
```

我们看到在visitBeanDefinition()方法中，分别访问了其他的数据，比如父类的名字、自己的类名、在IoC容器中的名称等各种信息。

2.6. 访问者模式的优缺点

优点：
1、解耦了数据结构与数据操作，使得操作集合可以独立变化；
2、扩展性好：可以通过扩展访问者角色，实现对数据集的不同操作；
3、元素具体类型并非单一，访问者均可操作；
4、各角色职责分离，符合单一职责原则。

缺点：
1、无法增加元素类型：若系统数据结构对象易于变化，经常有新的数据对象增加进来，则访问者类必须增加对应元素类型的操作，违背了开闭原则；
2、具体元素变更困难：具体元素增加属性，删除属性等操作会导致对应的访问者类需要进行相应的修改，尤其当有大量访问者类时，修改范围太大；
3、违背依赖倒置原则：为了达到”区别对待"，访问者依赖的是具体元素类型，而不是抽象。