访问者模式

题目链接

访问者模式-图形的面积

基本概念

访问者模式（Visitor Pattern）是一种行为型设计模式，可以在不改变对象结构的前提下，对对象中的元素进行新的操作。

举个例子，假设有一个动物园，里面有不同种类的动物，比如狮子、大象、猴子等。每个动物都会被医生检查身体，被管理员投喂，被游客观看，医生，游客，管理员都属于访问者。

```
// 定义动物接口
interface Animal {
    void accept(Visitor visitor);
}

// 具体元素类：狮子
class Lion implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 具体元素类：大象
class Elephant implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 具体元素类：猴子
class Monkey implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

如果你想对动物园中的每个动物执行一些操作，比如医生健康检查、管理员喂食、游客观赏等。就可以使用访问者模式来实现这些操作。

```
// 定义访问者接口
interface Visitor {
    void visit(Animal animal);
}

// 具体访问者类：医生
class Vet implements Visitor {
    @Override
    public void visit(Animal animal) {

    }
}

// 具体访问者类：管理员
class Zookeeper implements Visitor {
    @Override
    public void visit(Animal animal) {
    }
}

// 具体访问者类：游客
class VisitorPerson implements Visitor {
    @Override
    public void visit(Animal animal) {
    }
}
```

将这些访问者应用到动物园的每个动物上

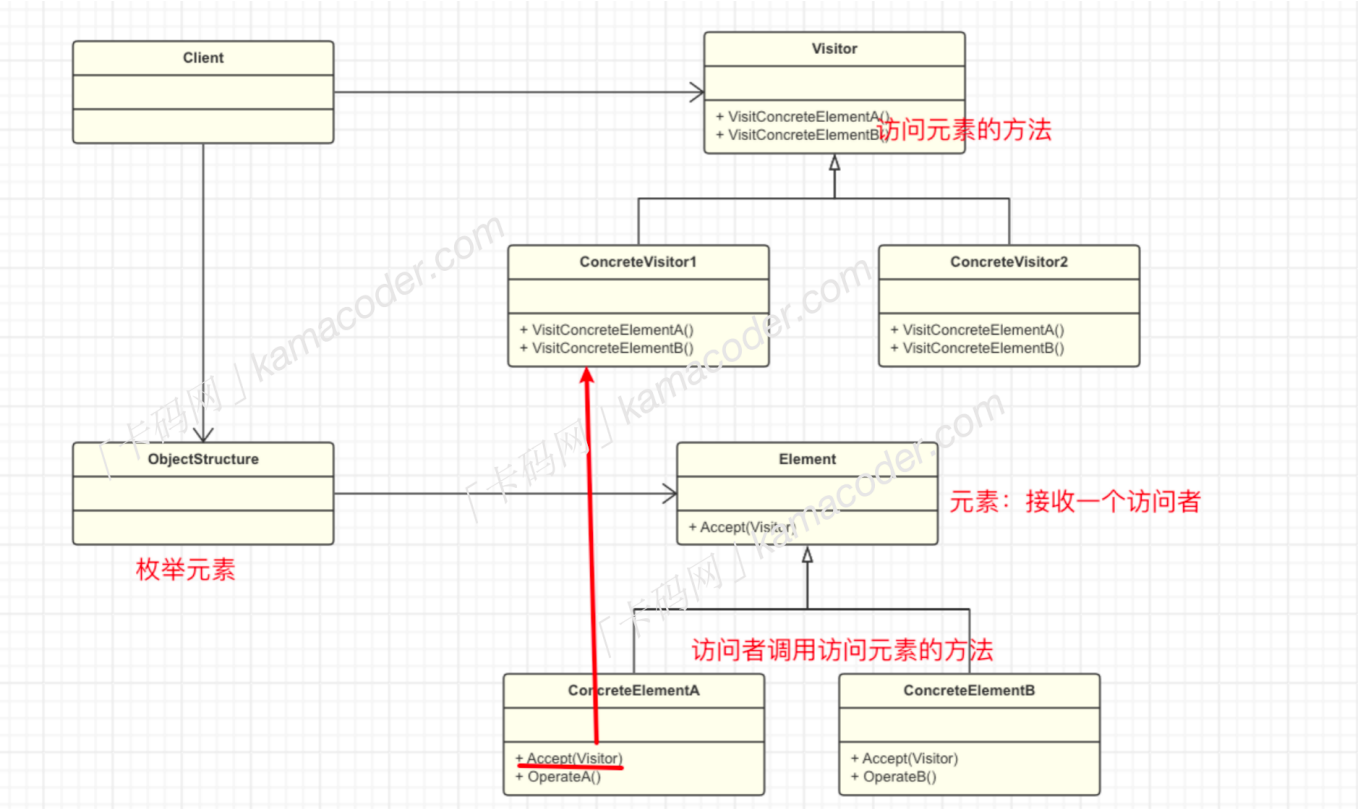
```
public class Main {
    public static void main(String[] args) {
        Animal lion = new Lion();
        Animal elephant = new Elephant();
        Animal monkey = new Monkey();

        Visitor vet = new Vet();
        Visitor zookeeper = new Zookeeper();
        Visitor visitorPerson = new VisitorPerson();

        // 动物接受访问者的访问
        lion.accept(vet);
        elephant.accept(zookeeper);
        monkey.accept(visitorPerson);
```

```
        }
    }
```

基本结构：

访问者模式包括以下几个基本角色：

- **抽象访问者（Visitor）：** 声明了访问者可以访问哪些元素，以及如何访问它们的方法 `visit`。
- **具体访问者（ConcreteVisitor）：** 实现了抽象访问者定义的方法，不同的元素类型可能有不同的访问行为。医生、管理员、游客都属于具体的访问者，它们的访问行为不同。
- **抽象元素（Element）：** 定义了一个accept方法，用于接受访问者的访问。
- **具体元素（ConcreteElement）：** 实现了accept方法，是访问者访问的目标。
- **对象结构（Object Structure）：** 包含元素的集合，可以是一个列表、一个集合或者其他数据结构。负责遍历元素，并调用元素的接受方法。



简易实现：

1. 定义抽象访问者：声明那些元素可以访问

```
// 抽象访问者
interface Visitor {
    void visit(ConcreteElementA element);
    void visit(ConcreteElementB element);
}
```

2. 实现具体访问者：实现具体的访问逻辑

```
// 具体访问者A
class ConcreteVisitorA implements Visitor {
    @Override
    public void visit(ConcreteElementA element) {
        System.out.println("ConcreteVisitorA Visit ConcreteElementA");
    }

    @Override
    public void visit(ConcreteElementB element) {
        System.out.println("ConcreteVisitorA Visit ConcreteElementB");
    }
}

// 具体访问者B
class ConcreteVisitorB implements Visitor {
    @Override
    public void visit(ConcreteElementA element) {
        System.out.println("ConcreteVisitorB Visit ConcreteElementA");
    }

    @Override
    public void visit(ConcreteElementB element) {
        System.out.println("ConcreteVisitorB Visit ConcreteElementB");
    }
}
```

3. 定义元素接口：声明接收访问者的方法。

```
// 抽象元素
interface Element {
    void accept(Visitor visitor);
}
```

4. 实现具体元素：实现接受访问者的方法

```
// 具体元素A
class ConcreteElementA implements Element {
```

```
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 具体元素B
class ConcreteElementB implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

5. 创建对象结构：提供一个接口让访问者访问它的元素。

```
// 对象结构
class ObjectStructure {
    private List<Element> elements = new ArrayList<>();

    public void attach(Element element) {
        elements.add(element);
    }

    public void detach(Element element) {
        elements.remove(element);
    }

    public void accept(Visitor visitor) {
        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

6. 客户端调用

```
public class Main {
    public static void main(String[] args) {
        ObjectStructure objectStructure = new ObjectStructure();
        objectStructure.attach(new ConcreteElementA());
        objectStructure.attach(new ConcreteElementB());

        Visitor visitorA = new ConcreteVisitorA();
        Visitor visitorB = new ConcreteVisitorB();

        objectStructure.accept(visitorA);
        objectStructure.accept(visitorB);
    }
}
```

使用场景

访问者模式结构较为复杂，但是访问者模式将同一类操作封装在一个访问者中，使得相关的操作彼此集中，提高了代码的可读性和维护性。它常用于**对象结构比较稳定，但经常需要在此对象结构上定义新的操作**，这样就无需修改现有的元素类，只需要定义新的访问者来添加新的操作。

本题代码

```
import java.util.Scanner;

// 元素接口
interface Shape {
    void accept(Visitor visitor);
}

// 具体元素类
class Circle implements Shape {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
```

```
}

class Rectangle implements Shape {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 访问者接口
interface Visitor {
    void visit(Circle circle);

    void visit(Rectangle rectangle);
}


// 具体访问者类
class AreaCalculator implements Visitor {
    @Override
    public void visit(Circle circle) {
        double area = 3.14 * Math.pow(circle.getRadius(), 2);
        System.out.println(area);
    }

    @Override
    public void visit(Rectangle rectangle) {
        int area = rectangle.getWidth() * rectangle.getHeight();
        System.out.println(area);
    }
```

```
}

// 对象结构类
class Drawing {
    private Shape[] shapes;

    public Drawing(Shape[] shapes) {
        this.shapes = shapes;
    }

    public void accept(Visitor visitor) {
        for (Shape shape : shapes) {
            shape.accept(visitor);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        scanner.nextLine();
        // 创建一个数组来存储图形对象
        Shape[] shapes = new Shape[n];
        // 根据用户输入创建不同类型的图形对象
        for (int i = 0; i < n; i++) {
            String[] input = scanner.nextLine().split(" ");
            if (input[0].equals("Circle")) {
                int radius = Integer.parseInt(input[1]);
                shapes[i] = new Circle(radius);
            } else if (input[0].equals("Rectangle")) {
                int width = Integer.parseInt(input[1]);
                int height = Integer.parseInt(input[2]);
                shapes[i] = new Rectangle(width, height);
            } else {
                System.out.println("Invalid input");
                return;
            }
        }
        // 创建一个图形集合
        Drawing drawing = new Drawing(shapes);
        // 创建一个面积计算访问者
        Visitor areaCalculator = new AreaCalculator();
        // 访问图形集合并计算面积
        drawing.accept(areaCalculator);
```

```
    }
}
```

其他语言版本

C++

```cpp
#include <iostream>
#include <cmath>
#include <vector>

class Shape;

// 访问者接口
class Visitor {
public:
    virtual void visit(class Circle& circle) = 0;
    virtual void visit(class Rectangle& rectangle) = 0;
};

// 元素接口
class Shape {
public:
    virtual ~Shape() {} // 添加虚析构函数
    virtual void accept(Visitor& visitor) = 0;
};

// 具体元素类
class Circle : public Shape {
private:
    int radius;

public:
    Circle(int radius) : radius(radius) {}

    int getRadius() const {
        return radius;
    }

    void accept(Visitor& visitor) override;
};

// 具体元素类
class Rectangle : public Shape {
private:
```

```
    int width;
    int height;

public:
    Rectangle(int width, int height) : width(width), height(height) {}

    int getWidth() const {
        return width;
    }

    int getHeight() const {
        return height;
    }

    void accept(Visitor& visitor) override;
};

// 具体访问者类
class AreaCalculator : public Visitor {
public:
    void visit(Circle& circle) override;
    void visit(Rectangle& rectangle) override;
};

// 对象结构类
class Drawing {
private:
    std::vector<Shape*> shapes;

public:
    Drawing(const std::vector<Shape*>& shapes) : shapes(shapes) {}

    void accept(Visitor& visitor) {
        for (Shape* shape : shapes) {
            shape->accept(visitor);
        }
    }
};

// 实现 accept 函数
void Circle::accept(Visitor& visitor) {
    visitor.visit(*this);
}

void Rectangle::accept(Visitor& visitor) {
    visitor.visit(*this);
```

```
}

// 实现 visit 函数
void AreaCalculator::visit(Circle& circle) {
    double area = 3.14 * std::pow(circle.getRadius(), 2);
    std::cout << area << std::endl;
}

void AreaCalculator::visit(Rectangle& rectangle) {
    int area = rectangle.getWidth() * rectangle.getHeight();
    std::cout << area << std::endl;
}

int main() {
    int n;
    std::cin >> n;

    std::vector<Shape*> shapes;

    for (int i = 0; i < n; i++) {
        std::string type;
        std::cin >> type;

        if (type == "Circle") {
            int radius;
            std::cin >> radius;
            shapes.push_back(new Circle(radius));
        } else if (type == "Rectangle") {
            int width, height;
            std::cin >> width >> height;
            shapes.push_back(new Rectangle(width, height));
        } else {
            // 处理无效输入
            std::cout << "Invalid input" << std::endl;
            return 1;
        }
    }

    Drawing drawing(shapes);
    AreaCalculator areaCalculator;
    drawing.accept(areaCalculator);

    // 释放动态分配的内存
    for (Shape* shape : shapes) {
        delete shape;
    }
```

```
    return 0;
}
```

Python

```python
from abc import ABC, abstractmethod

# 访问者接口
class Visitor(ABC):
    @abstractmethod
    def visit_circle(self, circle):
        pass

    @abstractmethod
    def visit_rectangle(self, rectangle):
        pass

# 元素接口
class Shape(ABC):
    @abstractmethod
    def accept(self, visitor):
        pass

# 具体元素类
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def get_radius(self):
        return self.radius

    def accept(self, visitor):
        visitor.visit_circle(self)

# 具体元素类
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_width(self):
        return self.width

    def get_height(self):
        return self.height
```

```
    def accept(self, visitor):
        visitor.visit_rectangle(self)

# 具体访问者类
class AreaCalculator(Visitor):
    def visit_circle(self, circle):
        area = 3.14 * circle.get_radius()**2
        print(area)

    def visit_rectangle(self, rectangle):
        area = rectangle.get_width() * rectangle.get_height()
        print(area)

# 对象结构类
class Drawing:
    def __init__(self, shapes):
        self.shapes = shapes

    def accept(self, visitor):
        for shape in self.shapes:
            shape.accept(visitor)

# 示例用法
if __name__ == "__main__":
    n = int(input())
    shapes = []

    for _ in range(n):
        shape_type, *params = input().split()

        if shape_type == "Circle":
            radius = int(params[0])
            shapes.append(Circle(radius))
        elif shape_type == "Rectangle":
            width, height = map(int, params)
            shapes.append(Rectangle(width, height))
        else:
            print("invalid input")
            exit(1)

    drawing = Drawing(shapes)
    area_calculator = AreaCalculator()
    drawing.accept(area_calculator)
```

Go

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "math"
    "strconv"
    "strings"
)

// 元素接口
type Shape interface {
    accept(Visitor)
}

// 具体元素类
type Circle struct {
    radius int
}

func NewCircle(radius int) *Circle {
    return &Circle{radius}
}

func (c *Circle) getRadius() int {
    return c.radius
}

func (c *Circle) accept(visitor Visitor) {
    visitor.visit(c)
}

type Rectangle struct {
    width, height int
}

func NewRectangle(width, height int) *Rectangle {
    return &Rectangle{width, height}
}

func (r *Rectangle) getWidth() int {
    return r.width
}
```

```
func (r *Rectangle) getHeight() int {
    return r.height
}

func (r *Rectangle) accept(visitor Visitor) {
    visitor.visit(r)
}

// 访问者接口
type Visitor interface {
    visit(shape Shape)
}

// 具体访问者类
type AreaCalculator struct{}

func (ac *AreaCalculator) visit(shape Shape) {
    switch concreteShape := shape.(type) {
    case *Circle:
        area := 3.14 * math.Pow(float64(concreteShape.getRadius()), 2)
        fmt.Println(area)
    case *Rectangle:
        area := concreteShape.getWidth() * concreteShape.getHeight()
        fmt.Println(area)
    }
}

// 对象结构类
type Drawing struct {
    shapes []Shape
}

func NewDrawing(shapes []Shape) *Drawing {
    return &Drawing{shapes}
}

func (d *Drawing) accept(visitor Visitor) {
    for _, shape := range d.shapes {
        shape.accept(visitor)
    }
}

func main() {
    var n int
    fmt.Scan(&n)
```

```
    shapes := make([]Shape, n)
    scanner := bufio.NewScanner(os.Stdin)
    for i := 0; i < n; i++ {
        scanner.Scan()
        input := strings.Split(scanner.Text(), " ")
        if input[0] == "Circle" {
            radius, _ := strconv.Atoi(input[1])
            shapes[i] = NewCircle(radius)
        } else if input[0] == "Rectangle" {
            width, _ := strconv.Atoi(input[1])
            height, _ := strconv.Atoi(input[2])
            shapes[i] = NewRectangle(width, height)
        } else {
            fmt.Println("Invalid input")
            return
        }
    }

    drawing := NewDrawing(shapes)
    areaCalculator := &AreaCalculator{}
    drawing.accept(areaCalculator)
}
```