外观模式

题目链接

外观模式-电源开关

基本概念

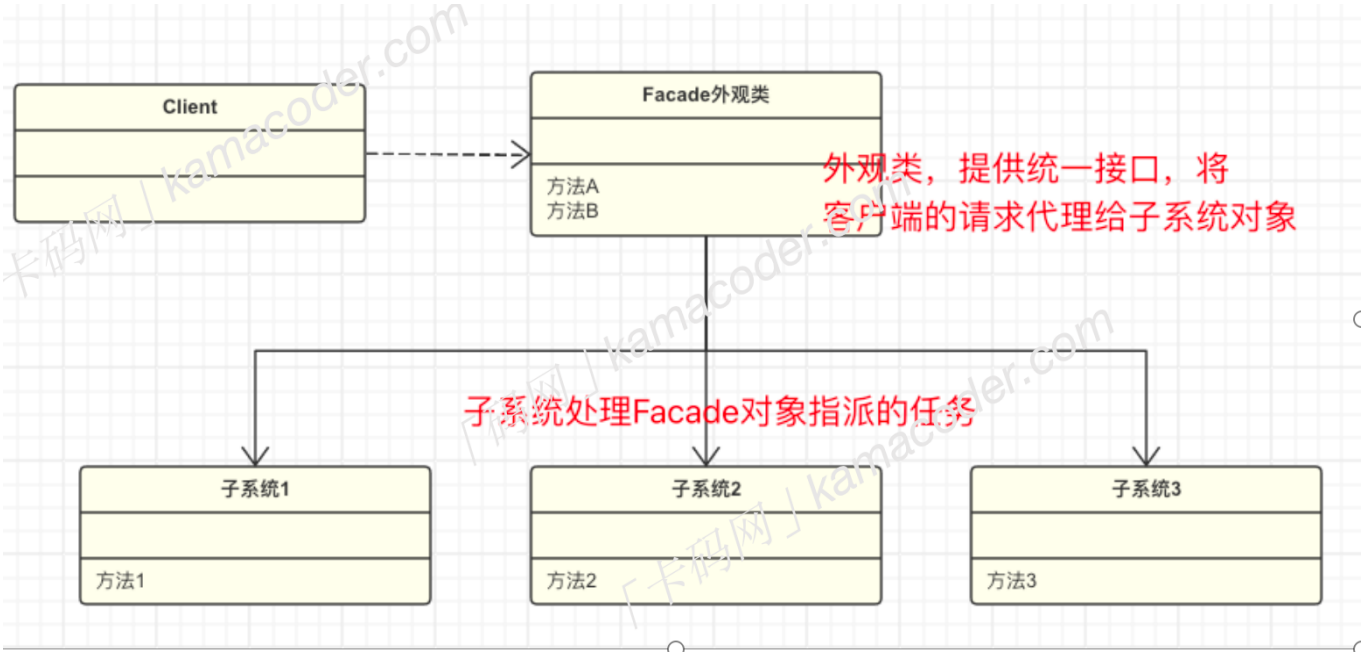
外观模式Facade Pattern，也被称为“门面模式”，是一种结构型设计模式，外观模式定义了一个高层接口，这个接口使得子系统更容易使用，同时也隐藏了子系统的复杂性。

> *门面模式可以将子系统关在“门里”隐藏起来，客户端只需要通过外观接口与外观对象进行交互，而不需要直接和多个子系统交互，无论子系统多么复杂，对于外部来说是隐藏的，这样可以降低系统的耦合度。*

举个例子，假设你正在编写的一个模块用来处理文件读取、解析、存储，我们可以将这个过程拆成三部分，然后创建一个外观类，将文件系统操作、数据解析和存储操作封装在外观类中，为客户端提供一个简化的接口，如果后续需要修改文件处理的流程或替换底层子系统，也只需在外观类中进行调整，不会影响客户端代码。

基本结构

外观模式的基本结构比较简单，只包括“外观”和“子系统类”



- 外观类：对外提供一个统一的高层次接口，使复杂的子系统变得更易使用。
- 子系统类：实现子系统的功能，处理外观类指派的任务。

简易实现

下面使用Java代码实现外观模式的通用结构

```
// 子系统A
class SubsystemA {
    public void operationA() {
        System.out.println("SubsystemA operation");
    }
}

// 子系统B
class SubsystemB {
    public void operationB() {
        System.out.println("SubsystemB operation");
    }
}

// 子系统C
class SubsystemC {
    public void operationC() {
        System.out.println("SubsystemC operation");
    }
}

// 外观类
class Facade {
    private SubsystemA subsystemA;
    private SubsystemB subsystemB;
    private SubsystemC subsystemC;

    public Facade() {
        this.subsystemA = new SubsystemA();
        this.subsystemB = new SubsystemB();
        this.subsystemC = new SubsystemC();
    }

    // 外观方法，封装了对子系统的操作
    public void facadeOperation() {
        subsystemA.operationA();
        subsystemB.operationB();
        subsystemC.operationC();
    }
}

// 客户端
```

```
public class Main {
    public static void main(String[] args) {
        // 创建外观对象
        Facade facade = new Facade();

        // 客户端通过外观类调用子系统的操作
        facade.facadeOperation();
    }
}
```

在上面的代码中，`Facade`类是外观类，封装了对三个子系统`SubSystem`的操作。客户端通过调用外观类的方法来实现对子系统的访问，而不需要直接调用子系统的方法。

优缺点和使用场景

外观模式通过提供一个简化的接口，隐藏了系统的复杂性，降低了客户端和子系统之间的耦合度，客户端不需要了解系统的内部实现细节，也不需要直接和多个子系统交互，只需要通过外观接口与外观对象进行交互。

但是如果需要添加新的子系统或修改子系统的行为，就可能需要修改外观类，这违背了“开闭原则”。

外观模式的应用也十分普遍，下面几种情况都使用了外观模式来进行简化。

- Spring框架是一个广泛使用外观模式的例子。Spring框架提供了一个大量的功能，包括依赖注入、面向切面编程（AOP）、事务管理等。Spring的`ApplicationContext`可以看作是外观，隐藏了底层组件的复杂性，使得开发者可以更轻松地使用Spring的功能。
- JDBC提供了一个用于与数据库交互的接口。`DriverManager`类可以看作是外观，它简化了数据库驱动的加载和连接的过程，隐藏了底层数据库连接的复杂性。
- Android系统的API中也使用了外观模式。例如，`Activity`类提供了一个外观，使得开发者可以更容易地管理应用的生命周期，而无需关心底层的事件和状态管理。

本题代码

```
import java.util.Scanner;

class AirConditioner {
    public void turnOff() {
        System.out.println("Air Conditioner is turned off.");
    }
}
class DeskLamp {
    public void turnOff() {
        System.out.println("Desk Lamp is turned off.");
```

```
    }
}
class Television {
    public void turnOff() {
        System.out.println("Television is turned off.");
    }
}

class PowerSwitchFacade {
    private DeskLamp deskLamp;
    private AirConditioner airConditioner;
    private Television television;

    public PowerSwitchFacade() {
        this.deskLamp = new DeskLamp();
        this.airConditioner = new AirConditioner();
        this.television = new Television();
    }

    public void turnOffDevice(int deviceCode) {
        switch (deviceCode) {
            case 1:
                airConditioner.turnOff();
                break;
            case 2:
                deskLamp.turnOff();
                break;
            case 3:
                television.turnOff();
                break;
            case 4:
                System.out.println("All devices are off.");
                break;
            default:
                System.out.println("Invalid device code.");
        }
    }
}

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

        // 读取输入
        int n = scanner.nextInt();
        int[] input = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            input[i] = scanner.nextInt();
        }

        // 创建电源总开关外观
        PowerSwitchFacade powerSwitch = new PowerSwitchFacade();

        // 执行操作
        for (int i = 0; i < n; i++) {
            powerSwitch.turnOffDevice(input[i]);
        }
    }
}
```

其他语言版本

C++

```
#include <iostream>
#include <vector>

class AirConditioner {
public:
    void turnOff() {
        std::cout << "Air Conditioner is turned off." << std::endl;
    }
};

class DeskLamp {
public:
    void turnOff() {
        std::cout << "Desk Lamp is turned off." << std::endl;
    }
};

class Television {
public:
    void turnOff() {
        std::cout << "Television is turned off." << std::endl;
    }
};

class PowerSwitchFacade {
private:
```

```
    DeskLamp deskLamp;
    AirConditioner airConditioner;
    Television television;

public:
    PowerSwitchFacade() {
    }

    void turnOffDevice(int deviceCode) {
        switch (deviceCode) {
            case 1:
                airConditioner.turnOff();
                break;
            case 2:
                deskLamp.turnOff();
                break;
            case 3:
                television.turnOff();
                break;
            case 4:
                std::cout << "All devices are off." << std::endl;
                break;
            default:
                std::cout << "Invalid device code." << std::endl;
        }
    }
};

int main() {
    // 读取输入
    int n;
    std::cin >> n;
    std::vector<int> input(n);

    for (int i = 0; i < n; i++) {
        std::cin >> input[i];
    }

    // 创建电源总开关外观
    PowerSwitchFacade powerSwitch;

    // 执行操作
    for (int i = 0; i < n; i++) {
        powerSwitch.turnOffDevice(input[i]);
    }
```

```
    return 0;
}
```

Python

```python
class AirConditioner:
    def turn_off(self):
        print("Air Conditioner is turned off.")

class DeskLamp:
    def turn_off(self):
        print("Desk Lamp is turned off.")

class Television:
    def turn_off(self):
        print("Television is turned off.")

class PowerSwitchFacade:
    def __init__(self):
        self.desk_lamp = DeskLamp()
        self.air_conditioner = AirConditioner()
        self.television = Television()

    def turn_off_device(self, device_code):
        if device_code == 1:
            self.air_conditioner.turn_off()
        elif device_code == 2:
            self.desk_lamp.turn_off()
        elif device_code == 3:
            self.television.turn_off()
        elif device_code == 4:
            print("All devices are off.")
        else:
            print("Invalid device code.")

if __name__ == "__main__":
    # 读取输入
    n = int(input())
    input_data = [int(input()) for _ in range(n)]

    # 创建电源总开关外观
    power_switch = PowerSwitchFacade()

    # 执行操作
    for device_code in input_data:
        power_switch.turn_off_device(device_code)
```

Go

```go
package main

import "fmt"

// AirConditioner 类
type AirConditioner struct{}

func (ac *AirConditioner) turnOff() {
    fmt.Println("Air Conditioner is turned off.")
}

// DeskLamp 类
type DeskLamp struct{}

func (dl *DeskLamp) turnOff() {
    fmt.Println("Desk Lamp is turned off.")
}

// Television 类
type Television struct{}

func (tv *Television) turnOff() {
    fmt.Println("Television is turned off.")
}

// PowerSwitchFacade 类
type PowerSwitchFacade struct {
    deskLamp       DeskLamp
    airConditioner AirConditioner
    television     Television
}

func (psf *PowerSwitchFacade) turnOffDevice(deviceCode int) {
    switch deviceCode {
    case 1:
        psf.airConditioner.turnOff()
    case 2:
        psf.deskLamp.turnOff()
    case 3:
        psf.television.turnOff()
    case 4:
        fmt.Println("All devices are off.")
    default:
```

```
        fmt.Println("Invalid device code.")
    }
}

func main() {
    var n int
    fmt.Scan(&n) // 读取输入

    input := make([]int, n)
    for i := 0; i < n; i++ {
        fmt.Scan(&input[i])
    }

    // 创建电源总开关外观
    powerSwitch := PowerSwitchFacade{}

    // 执行操作
    for i := 0; i < n; i++ {
        powerSwitch.turnOffDevice(input[i])
    }
}
```