状态模式

题目链接

状态模式-开关台灯

基本结构

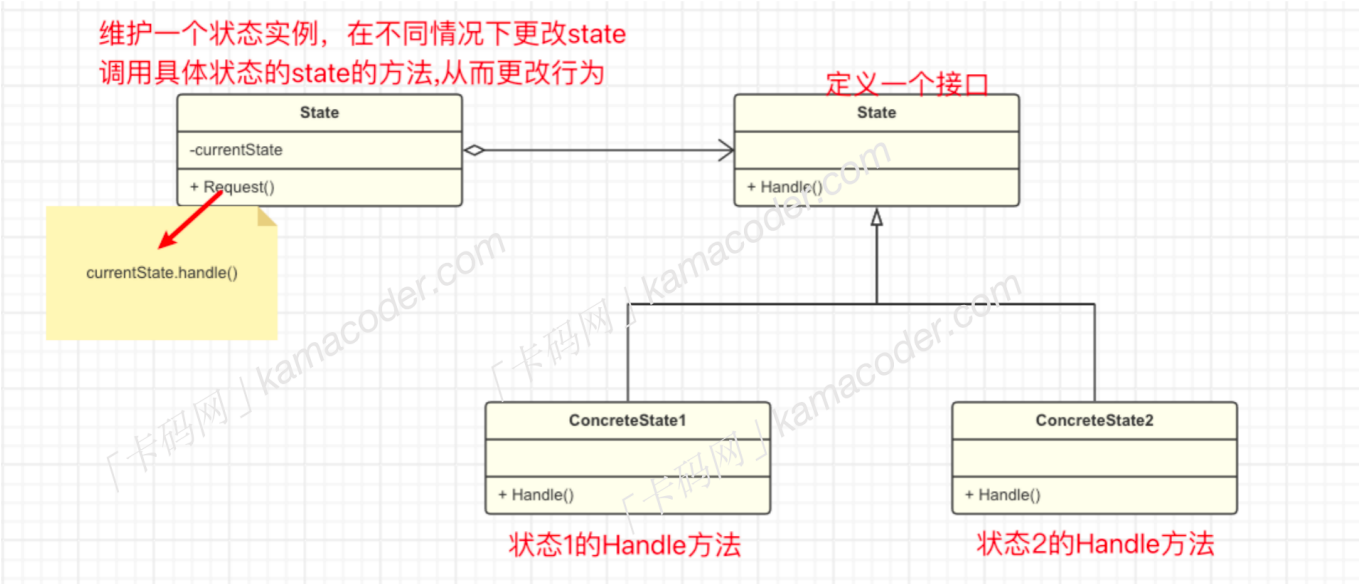
状态模式（State Pattern）是一种行为型设计模式，它适用于一个对象在在不同的状态下有不同的行为时，比如说电灯的开、关、闪烁是不停的状态，状态不同时，对应的行为也不同，在没有状态模式的情况下，为了添加新的状态或修改现有的状态，往往**需要修改已有的代码**，这违背了开闭原则，而且如果对象的状态切换逻辑和各个状态的行为都在同一个类中实现，就可能导致该类的职责过重，不符合单一职责原则。

而状态模式将每个状态的行为封装在一个具体状态类中，使得每个状态类相对独立，并将对象在不同状态下的行为进行委托，从而使得对象的状态可以在运行时动态改变，每个状态的实现也不会影响其他状态。

基本结构：

状态模式包括以下几个重要角色：

- `State`（状态）： 定义一个接口，用于封装与Context的一个特定状态相关的行为。
- `ConcreteState`（具体状态）： 负责处理Context在状态改变时的行为，每一个具体状态子类实现一个与`Context`的一个状态相关的行为。
- `Context`（上下文）：维护一个具体状态子类的实例，这个实例定义当前的状态。



基本使用

1. 定义状态接口：创建一个状态接口，该接口声明了对象可能的各种状态对应的方法。

```
// 状态接口
public interface State {
    void handle();
}
```

2. 实现具体状态类： 为对象可能的每种状态创建具体的状态类，实现状态接口中定义的方法。

```
// 具体状态类1
public class ConcreteState1 implements State {
    @Override
    public void handle() {
        // 执行在状态1下的操作
    }
}

// 具体状态类2
public class ConcreteState2 implements State {
    @Override
    public void handle() {
        // 执行在状态2下的操作
    }
}
```

3. 创建上下文类：该类包含对状态的引用，并在需要时调用当前状态的方法。

```
// 上下文类
public class Context {
    private State currentState;

    public void setState(State state) {
        this.currentState = state;
    }

    public void request() {
        currentState.handle();
    }
}
```

4. 客户端使用：创建具体的状态对象和上下文对象，并通过上下文对象调用相应的方法。通过改变状态，可以改变上下文对象的行为

```
public class Client {
    public static void main(String[] args) {
        Context context = new Context();

        State state1 = new ConcreteState1();
        State state2 = new ConcreteState2();

        context.setState(state1);
        context.request(); // 执行在状态1下的操作

        context.setState(state2);
        context.request(); // 执行在状态2下的操作
    }
}
```

使用场景

状态模式将每个状态的实现都封装在一个类中，每个状态类的实现相对独立，使得添加新状态或修改现有状态变得更加容易，避免了使用大量的条件语句来控制对象的行为。但是如果状态过多，会导致类的数量增加，可能会使得代码结构复杂。

总的来说，状态模式适用于有限状态机的场景，其中对象的行为在运行时可以根据内部状态的改变而改变，在游戏开发中，Unity3D 的 Animator 控制器就是一个状态机。它允许开发人员定义不同的状态（动画状态），并通过状态转换来实现角色的动画控制和行为切换。

本题代码

```
import java.util.Scanner;

// 状态接口
interface State {
    String handle(); // // 处理状态的方法
}

// 具体状态类
class OnState implements State {
    @Override
    public String handle() {
        return "Light is ON";
    }
}

class OffState implements State {
    @Override
    public String handle() {
```

```
        return "Light is OFF";
    }
}

class BlinkState implements State {
    @Override
    public String handle() {
        return "Light is Blinking";
    }
}

// 上下文类
class Light {
    private State state; // 当前状态

    public Light() {
        this.state = new OffState(); // 初始状态为关闭
    }

    public void setState(State state) { // 设置新的状态
        this.state = state;
    }

    public String performOperation() { // 执行当前状态的操作
        return state.handle();
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建一个Scanner对象以读取用户输入
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        scanner.nextLine();

        Light light = new Light();
        // 处理用户输入
        for (int i = 0; i < n; i++) {
            String command = scanner.nextLine().trim();
            // 根据输入修改灯的状态
            switch (command) {
                case "ON":
                    light.setState(new OnState());
                    break;
                case "OFF":
```

```
                    light.setState(new OffState());
                    break;
                case "BLINK":
                    light.setState(new BlinkState());
                    break;
                default:
                    System.out.println("Invalid command: " + command);
                    break;
            }
            // 显示灯的当前状态
            System.out.println(light.performOperation());
        }
    }
}
```

其他语言版本

C++

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

// 状态接口
class State {
public:
    virtual std::string handle() = 0;  // 处理状态的方法
};

// 具体状态类
class OnState : public State {
public:
    std::string handle() override {
        return "Light is ON";
    }
};

class OffState : public State {
public:
    std::string handle() override {
        return "Light is OFF";
    }
};

class BlinkState : public State {
```

```
public:
    std::string handle() override {
        return "Light is Blinking";
    }
};

// 上下文类
class Light {
private:
    State* state;  // 当前状态

public:
    Light() : state(new OffState()) {}  // 初始状态为关闭

    void setState(State* newState) {  // 设置新的状态
        delete state;  // 释放之前的状态对象
        state = newState;
    }

    std::string performOperation() {  // 执行当前状态的操作
        return state->handle();
    }

    ~Light() {
        delete state;  // 释放内存
    }
};

int main() {
    // 读取要输入的命令数量
    int n;
    std::cin >> n;
    std::cin.ignore();  // 消耗掉整数后的换行符

    // 创建一个Light对象
    Light light;

    // 处理用户输入的每个命令
    for (int i = 0; i < n; i++) {
        // 读取命令并去掉首尾空白字符
        std::string command;
        std::getline(std::cin, command);

        // 根据命令执行相应的操作
        if (command == "ON") {
            light.setState(new OnState());
```

```
        } else if (command == "OFF") {
            light.setState(new OffState());
        } else if (command == "BLINK") {
            light.setState(new BlinkState());
        } else {
            // 处理无效命令
            std::cout << "Invalid command: " << command << std::endl;
        }

        // 在每个命令后显示灯的当前状态
        std::cout << light.performOperation() << std::endl;
    }

    return 0;
}
```

Python

```
# 状态接口
class State:
    def handle(self):
        pass

# 具体状态类
class OnState(State):
    def handle(self):
        return "Light is ON"

class OffState(State):
    def handle(self):
        return "Light is OFF"

class BlinkState(State):
    def handle(self):
        return "Light is Blinking"

# 上下文类
class Light:
    def __init__(self):
        self.state = OffState()  # 初始状态为关闭

    def set_state(self, new_state):
        self.state = new_state  # 设置新的状态

    def perform_operation(self):
        return self.state.handle()  # 执行当前状态的操作
```

```
# 处理用户输入
def main():
    # 读取要输入的命令数量
    n = int(input().strip())

    # 创建一个Light对象
    light = Light()

    # 处理用户输入的每个命令
    for _ in range(n):
        # 读取命令并去掉首尾空白字符
        command = input().strip()

        # 根据命令执行相应的操作
        if command == "ON":
            light.set_state(OnState())
        elif command == "OFF":
            light.set_state(OffState())
        elif command == "BLINK":
            light.set_state(BlinkState())
        else:
            # 处理无效命令
            print("Invalid command:", command)

        # 在每个命令后显示灯的当前状态
        print(light.perform_operation())

if __name__ == "__main__":
    main()
```

Go

```
package main

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

// 状态接口
type State interface {
    Handle() string // 处理状态的方法
```

```
}

// 具体状态类
type OnState struct{}

func (s *OnState) Handle() string {
    return "Light is ON"
}

type OffState struct{}

func (s *OffState) Handle() string {
    return "Light is OFF"
}

type BlinkState struct{}

func (s *BlinkState) Handle() string {
    return "Light is Blinking"
}

// 上下文类
type Light struct {
    state State // 当前状态
}

func NewLight() *Light {
    return &Light{state: &OffState{}} // 初始状态为关闭
}

func (l *Light) SetState(state State) { // 设置新的状态
    l.state = state
}

func (l *Light) PerformOperation() string { // 执行当前状态的操作
    return l.state.Handle()
}

// 主函数
func main() {
    scanner := bufio.NewScanner(os.Stdin)

    // 读取用户输入
    scanner.Scan()
    var n int
    fmt.Sscanf(scanner.Text(), "%d", &n)
```

```
    light := NewLight()

    // 处理用户输入
    for i := 0; i < n; i++ {
        scanner.Scan()
        command := strings.TrimSpace(scanner.Text())

        // 根据输入修改灯的状态
        switch command {
        case "ON":
            light.SetState(&OnState{})
        case "OFF":
            light.SetState(&OffState{})
        case "BLINK":
            light.SetState(&BlinkState{})
        default:
            fmt.Println("Invalid command:", command)
            continue
        }

        // 显示灯的当前状态
        fmt.Println(light.PerformOperation())
    }
}
```