命令模式

题目链接

命令模式-自助点餐机

基本概念

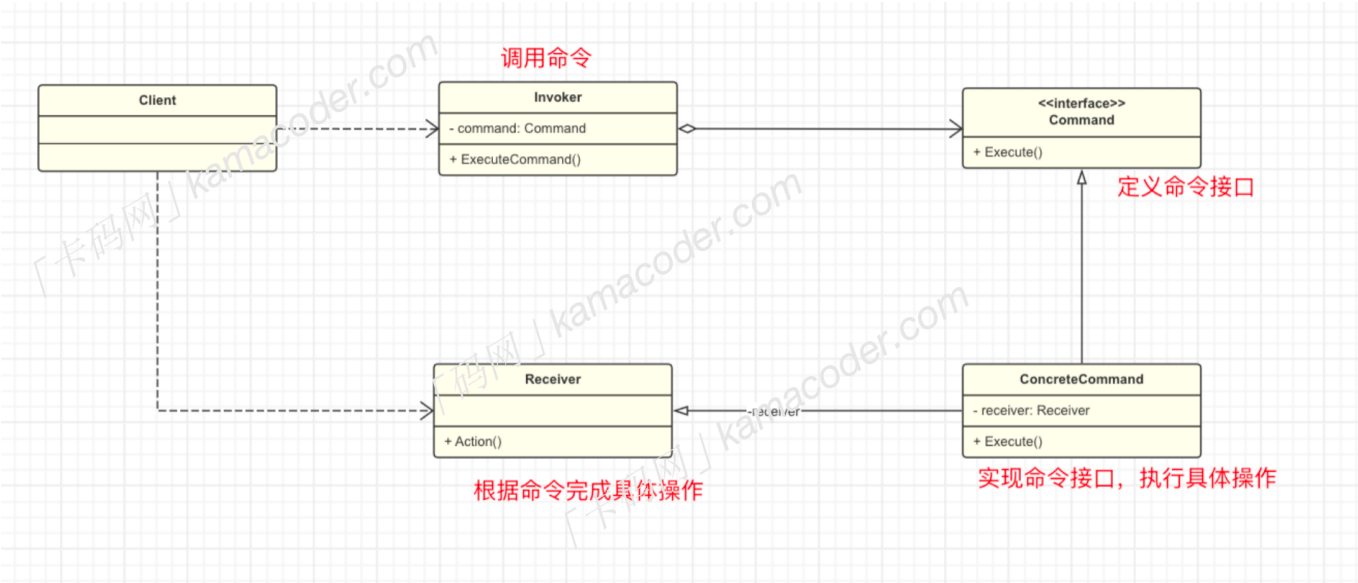
命令模式是一种行为型设计模式，其允许**将请求封装成一个对象**(命令对象，包含执行操作所需的所有信息)，**并将命令对象按照一定的顺序存储在队列中，然后再逐一调用执行，这些命令也可以支持反向操作，进行撤销和重做。**

这样一来，发送者只需要触发命令就可以完成操作，不需要知道接受者的具体操作，从而实现两者间的解耦。

举个现实中的应用场景，遥控器可以控制不同的设备，在命令模式中，可以假定每个按钮都是一个命令对象，包含执行特定操作的命令，不同设备对同一命令的具体操作也不同，这样就可以方便的添加设备和命令对象。

基本结构

命令模式包含以下几个基本角色:



- 命令接口`Command`：接口或者抽象类，定义执行操作的接口。
- 具体命令类`ConcreteCommand`: 实现命令接口，执行具体操作，在调用`execute`方法时使“接收者对象”根据命令完成具体的任务，比如遥控器中的“开机”，“关机”命令。

- 接收者类Receiver: 接受并执行命令的对象，可以是任何对象，遥控器可以控制空调，也可以控制电视机，电视机和空调负责执行具体操作，是接收者。
- 调用者类Invoker: 发起请求的对象，有一个将命令作为参数传递的方法。它不关心命令的具体实现，只负责调用命令对象的 execute() 方法来传递请求，在本例中，控制遥控器的“人”就是调用者。
- 客户端：创建具体的命令对象和接收者对象，然后将它们组装起来。

简易实现

1. 定义执行操作的接口：包含一个execute方法。有的时候还会包括unExecute方法，表示撤销命令。

```
public interface Command {
    void execute();
}
```

2. 实现命令接口，执行具体的操作。

```
public class ConcreteCommand implements Command {
    // 接收者对象
    private Receiver receiver;

    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        // 调用接收者相应的操作
        receiver.action();
    }
}
```

3. 定义接受者类，知道如何实施与执行一个请求相关的操作。

```
public class Receiver {
    public void action() {
        // 执行操作
    }
}
```

4. 定义调用者类，调用命令对象执行请求。

```
public class Invoker {
    private Command command;

    public Invoker(Command command) {
        this.command = command;
    }

    public void executeCommand() {
        command.execute();
    }
}
```

调用者类中可以维护一个命令队列或者“撤销栈”，以支持批处理和撤销命令。

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

// 调用者类：命令队列和撤销请求
class Invoker {
    private Queue<Command> commandQueue; // 命令队列
    private Stack<Command> undoStack;    // 撤销栈

    public Invoker() {
        this.commandQueue = new LinkedList<>();
        this.undoStack = new Stack<>();
    }

    // 设置命令并执行
    public void setAndExecuteCommand(Command command) {
        command.execute();
        commandQueue.offer(command);
        undoStack.push(command);
    }

    // 撤销上一个命令
    public void undoLastCommand() {
        if (!undoStack.isEmpty()) {
            Command lastCommand = undoStack.pop();
            lastCommand.undo(); // 需要命令类实现 undo 方法
            commandQueue.remove(lastCommand);
        } else {
            System.out.println("No command to undo.");
        }
    }
```

```
    // 执行命令队列中的所有命令
    public void executeCommandsInQueue() {
        for (Command command : commandQueue) {
            command.execute();
        }
    }
}
```

5. 客户端使用，创建具体的命令对象和接收者对象，然后进行组装。

```
public class Main {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker(command);

        invoker.executeCommand();
    }
}
```

优缺点和使用场景

命令模式在需要将请求封装成对象、支持撤销和重做、设计命令队列等情况下，都是一个有效的设计模式。

- **撤销操作：** 需要支持撤销操作，命令模式可以存储历史命令，轻松实现撤销功能。
- **队列请求：** 命令模式可以将请求排队，形成一个命令队列，依次执行命令。
- **可扩展性：** 可以很容易地添加新的命令类和接收者类，而不影响现有的代码。新增命令不需要修改现有代码，符合开闭原则。

但是对于每个命令，都会有一个具体命令类，这可能导致类的数量急剧增加，增加了系统的复杂性。

命令模式同样有着很多现实场景的应用，比如Git中的很多操作，如提交（commit）、合并（merge）等，都可以看作是命令模式的应用，用户通过执行相应的命令来操作版本库。Java的GUI编程中，很多事件处理机制也都使用了命令模式。例如，每个按钮都有一个关联的 `Action`，它代表一个命令，按钮的点击触发 `Action` 的执行。

本题代码

```
import java.util.Scanner;

// 命令接口
interface Command {
```

```
    void execute();
}

// 具体命令类 - 点餐命令
class OrderCommand implements Command {
    private String drinkName;
    private DrinkMaker receiver;

    public OrderCommand(String drinkName, DrinkMaker receiver) {
        this.drinkName = drinkName;
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        receiver.makeDrink(drinkName);
    }
}

// 接收者类 - 制作饮品
class DrinkMaker {
    public void makeDrink(String drinkName) {
        System.out.println(drinkName + " is ready!");
    }
}

// 调用者类 - 点餐机
class OrderMachine {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeOrder() {
        command.execute();
    }
}

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

        // 创建接收者和命令对象
        DrinkMaker drinkMaker = new DrinkMaker();
```

```
        // 读取命令数量
        int n = scanner.nextInt();
        scanner.nextLine();

        while (n-- > 0) {
            // 读取命令
            String drinkName = scanner.next();

            // 创建命令对象
            Command command = new OrderCommand(drinkName, drinkMaker);

            // 执行命令
            OrderMachine orderMachine = new OrderMachine();
            orderMachine.setCommand(command);
            orderMachine.executeOrder();
        }
        scanner.close();
    }
}
```

其他语言版本

Java

使用命令模式+工厂模式，进一步将程序进行解耦，主程序不需要知道具体命令类的实现细节，后续增加新命令或饮料类型时，只需修改工厂类，不会影响主程序的结构。

```
import java.util.Scanner;

// 命令接口
interface Command {
    void execute();
}

// 具体命令类 - 点餐命令
class OrderCommand implements Command {
    private String drinkName;
    private DrinkMaker receiver;

    public OrderCommand(String drinkName, DrinkMaker receiver) {
        this.drinkName = drinkName;
        this.receiver = receiver;
    }

    @Override
```

```
    public void execute() {
        receiver.makeDrink(drinkName);
    }
}

// 接收者类 - 制作饮品
class DrinkMaker {
    public void makeDrink(String drinkName) {
        System.out.println(drinkName + " is ready!");
    }
}

// 调用者类 - 点餐机
class OrderMachine {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeOrder() {
        if (command != null) {
            command.execute();
        } else {
            System.out.println("未设置命令.");
        }
    }
}

// 命令工厂类
class CommandFactory {
    private DrinkMaker drinkMaker;

    public CommandFactory(DrinkMaker drinkMaker) {
        this.drinkMaker = drinkMaker;
    }

    public Command createCommand(String drinkName) {
        return new OrderCommand(drinkName, drinkMaker);
    }
}

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

```
        // 创建接收者和工厂对象
        DrinkMaker drinkMaker = new DrinkMaker();
        CommandFactory commandFactory = new CommandFactory(drinkMaker);
        OrderMachine orderMachine = new OrderMachine();

        // 读取命令数量
        int n = scanner.nextInt();
        scanner.nextLine();

        while (n-- > 0) {
            // 读取命令
            String drinkName = scanner.nextLine().trim();

            if (drinkName.isEmpty()) {
                System.out.println("无效输入，请输入饮品名.");
                continue;
            }

            // 使用工厂创建命令对象
            Command command = commandFactory.createCommand(drinkName);

            // 设置命令并执行
            orderMachine.setCommand(command);
            orderMachine.executeOrder();
        }
        scanner.close();
    }
}
```

C++

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

class DrinkMaker; // 前向声明

// 命令接口
class Command {
public:
    virtual void execute() = 0;
    virtual ~Command() = default; // 添加虚析构函数
};

// 具体命令类 - 点餐命令
```

```
class OrderCommand : public Command {
private:
    std::string drinkName;
    DrinkMaker* receiver; // 使用前向声明

public:
    OrderCommand(const std::string& drinkName, DrinkMaker* receiver);
    void execute() override;
};

// 接收者类 - 制作饮品
class DrinkMaker {
public:
    void makeDrink(const std::string& drinkName) {
        std::cout << drinkName << " is ready!" << std::endl;
    }
};

// 实现 OrderCommand 的构造函数和 execute 函数
OrderCommand::OrderCommand(const std::string& drinkName, DrinkMaker*
receiver) : drinkName(drinkName), receiver(receiver) {}

void OrderCommand::execute() {
    receiver->makeDrink(drinkName);
}

// 调用者类 - 点餐机
class OrderMachine {
private:
    Command* command;

public:
    void setCommand(Command* command) {
        this->command = command;
    }

    void executeOrder() {
        command->execute();
    }
};

int main() {
    // 创建接收者和命令对象
    DrinkMaker drinkMaker;

    // 读取命令数量
```

```
    int n;
    std::cin >> n;
    std::cin.ignore(); // 消耗掉换行符

    while (n-- > 0) {
        // 读取命令
        std::string drinkName;
        std::cin >> drinkName;

        // 创建命令对象
        Command* command = new OrderCommand(drinkName, &drinkMaker);

        // 执行命令
        OrderMachine orderMachine;
        orderMachine.setCommand(command);
        orderMachine.executeOrder();

        // 释放动态分配的命令对象
        delete command;
    }

    return 0;
}
```

Python

```
from abc import ABC, abstractmethod

# 命令接口
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

# 具体命令类 - 点餐命令
class OrderCommand(Command):
    def __init__(self, drink_name, receiver):
        self.drink_name = drink_name
        self.receiver = receiver

    def execute(self):
        self.receiver.make_drink(self.drink_name)

# 接收者类 - 制作饮品
class DrinkMaker:
    def make_drink(self, drink_name):
```

```python
        print(f"{drink_name} is ready!")

# 调用者类 - 点餐机
class OrderMachine:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def execute_order(self):
        self.command.execute()

if __name__ == "__main__":
    # 创建接收者和命令对象
    drink_maker = DrinkMaker()

    # 读取命令数量
    n = int(input())

    for _ in range(n):
        # 读取命令
        drink_name = input()

        # 创建命令对象
        command = OrderCommand(drink_name, drink_maker)

        # 执行命令
        order_machine = OrderMachine()
        order_machine.set_command(command)
        order_machine.execute_order()
```

Go

```go
package main

import "fmt"

// Command 接口
type Command interface {
    Execute()
}

// OrderCommand 具体命令类 - 点餐命令
type OrderCommand struct {
    DrinkName string
```

```
    Receiver  *DrinkMaker
}

func (oc *OrderCommand) Execute() {
    oc.Receiver.MakeDrink(oc.DrinkName)
}

// DrinkMaker 接收者类 - 制作饮品
type DrinkMaker struct{}

func (dm *DrinkMaker) MakeDrink(drinkName string) {
    fmt.Println(drinkName + " is ready!")
}

// OrderMachine 调用者类 - 点餐机
type OrderMachine struct {
    Command Command
}

func (om *OrderMachine) SetCommand(command Command) {
    om.Command = command
}

func (om *OrderMachine) ExecuteOrder() {
    om.Command.Execute()
}

func main() {
    // 创建接收者和命令对象
    drinkMaker := &DrinkMaker{}

    // 读取命令数量
    var n int
    fmt.Scan(&n)

    for i := 0; i < n; i++ {
        // 读取命令
        var drinkName string
        fmt.Scan(&drinkName)

        // 创建命令对象
        command := &OrderCommand{DrinkName: drinkName, Receiver:
drinkMaker}

        // 执行命令
        orderMachine := &OrderMachine{}
```

```
        orderMachine.SetCommand(command)
        orderMachine.ExecuteOrder()
    }
}
```

Typescript

```
abstract class Command {
  protected receiver: Receiver;

  constructor(receiver: Receiver) {
    this.receiver = receiver;
  }

  abstract execute(): void;
}

class Receiver {
  milkTea() {
    console.log("MileTea is Ready!");
  }
  coffee() {
    console.log("Coffee is Ready!");
  }
  cola() {
    console.log("Cola is Ready!");
  }
}

class MileTeaCommand extends Command {
  constructor(receiver: Receiver) {
    super(receiver);
  }

  execute(): void {
    this.receiver.milkTea();
  }
}
class CoffeeCommand extends Command {
  constructor(receiver: Receiver) {
    super(receiver);
  }

  execute(): void {
    this.receiver.coffee();
```

```
  }
}
class ColaCommand extends Command {
  constructor(receiver: Receiver) {
    super(receiver);
  }

  execute(): void {
    this.receiver.cola();
  }
}

class Invoke {
  private command: Command;

  constructor(command: Command) {
    this.command = command;
  }

  setOrder(command: Command) {
    this.command = command;
  }

  invokeCommand() {
    this.command.execute();
  }
}

// @ts-ignore
entry(4, (...args) => {
  const machine = new Receiver();
  let waiter: Invoke;
  let command: Command;

  args.forEach((type) => {
    if (type === "MilkTea") {
      command = new MileTeaCommand(machine);
    } else if (type === "Coffee") {
      command = new CoffeeCommand(machine);
    } else if (type === "Cola") {
      command = new ColaCommand(machine);
    }

    if (!waiter) {
      waiter = new Invoke(command);
      waiter.invokeCommand();
```

```
    } else {
      waiter.setOrder(command);
      waiter.invokeCommand();
    }
  });
})("MilkTea")("Coffee")("Cola")("MilkTea");

function entry(count: number, fn: (...args: any) => void) {
  function dfs(...args) {
    if (args.length < count) {
      return (arg) => dfs(...args, arg);
    }

    return fn(...args);
  }
  return dfs;
}
```