行为模式（一）

2020年4月27日 15:40

行为型模式作业（一）

作业要求：

1、结合实例，绘制责任链模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

2、结合实例，绘制命令模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

3、在军队中，一般根据战争规模的大小和重要性由不同级别的长官（ Officer）来下达作战命令，情报人员向上级递交军情（如敌人的数量），作战命令需要上级批准，如果直接上级不具备下达命令的权力，则上级又传给上级，直到有人可以决定为止。现使用职责链模式来模拟该过程，客户类Client）模拟情报人员，首先向级别最低的班长（Banzhang）递交任务书（Mission），即军情，如果超出班长的权力范围，则传递给排长（Paizhang），排长如果也不能处理，则传递给营长（Yingzhang），如果营长也不能处理，则需要开会讨论。

设置这几级长官的权力范围分别是：

（1）敌人数量<10，班长下达作战命令；

（2）10≤敌人数量<50，排长下达作战命令；

（3）50≤敌人数量<200，营长下达作战命令；

（4）敌人数量≥200，需要开会讨论再下达作战命令。

绘制类图并编程实现

要求：

1、模式结构图要求使用工具软件进行绘制，模式分析需要说明每个角色及作用。 如果可能，每个模式最好使用实际案例来说明。

2、以WORD文档/PDF格式提交。

3、提交的作业如果有多个文件，就提交多个，不要做压缩包。

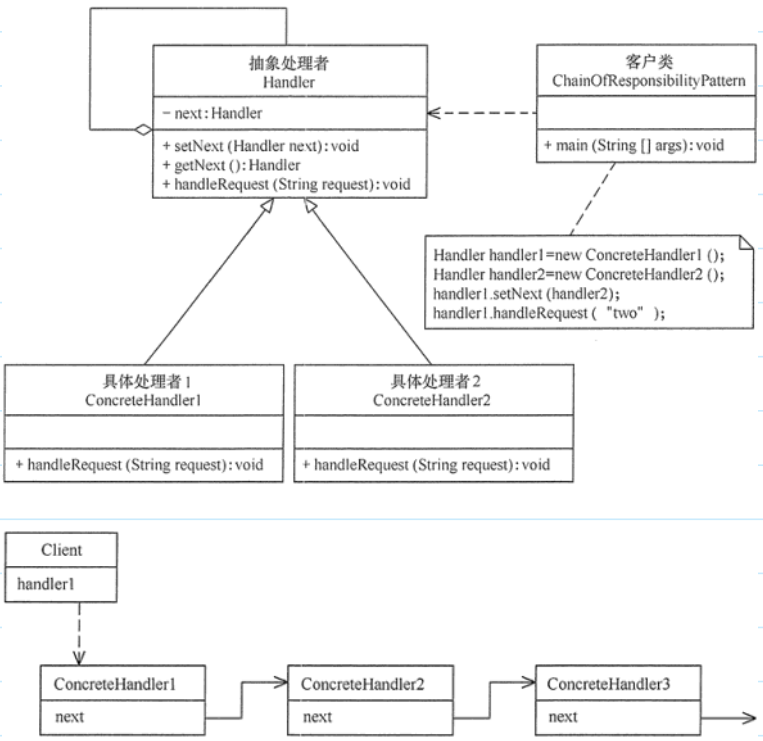
- **作业与笔记github地址**：
  https://github.com/baobaotql/CCNU_DesignPattern

- **Q1:结合实例，绘制责任链模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析**
  A1：
  - **责任链模式定义：**
    为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止
  - **优点**
    - 降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
    - 增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则。
    - 增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改

变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。

- 责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。
- 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

- **缺点**
  - 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
  - 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
  - 职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。
- **模式结构**
  - 抽象处理者（Handler）角色：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。
  - 具体处理者（Concrete Handler）角色：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。
  - 客户类（Client）角色：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。
  - **模式结构图**



  - 在责任链模式中，客户只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程，所以责任链将请求的发送者和请求的处理者解耦了
  - **模式代码**

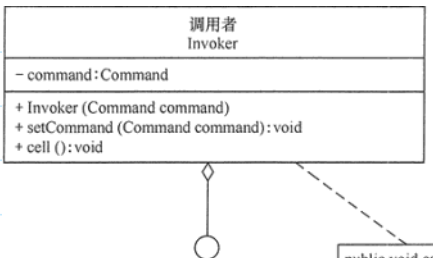
```
package chainOfResponsibility;
```

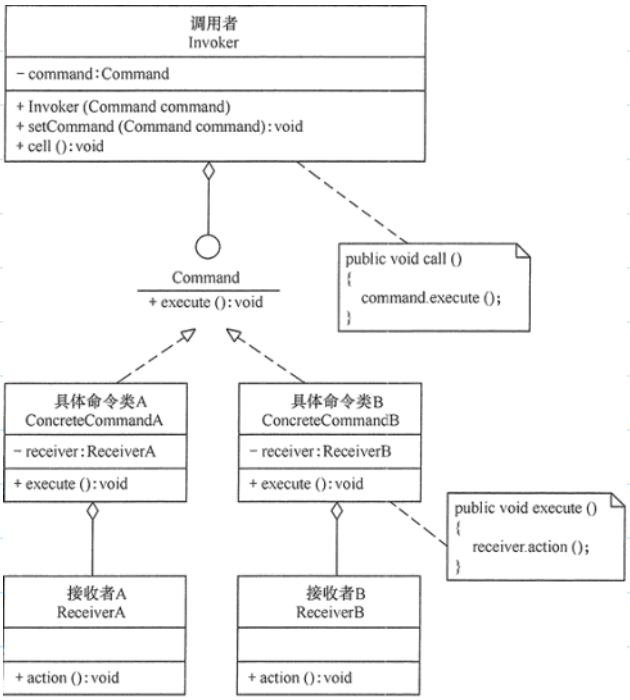
```
public class ChainOfResponsibilityPattern
{
    public static void main(String[] args)
    {
        //组装责任链
        Handler handler1=new ConcreteHandler1();
        Handler handler2=new ConcreteHandler2();
        handler1.setNext(handler2);
        //提交请求
        handler1.handleRequest("two");
    }
}
//抽象处理者角色
abstract class Handler
{
    private Handler next;
    public void setNext(Handler next)
    {
        this.next=next;
    }
    public Handler getNext()
    {
        return next;
    }
    //处理请求的方法
    public abstract void handleRequest(String request);
}
//具体处理者角色1
class ConcreteHandler1 extends Handler
{
    public void handleRequest(String request)
    {
        if(request.equals("one"))
        {
            System.out.println("具体处理者1负责处理该请求！");
        }
        else
        {
            if(getNext()!=null)
            {
                getNext().handleRequest(request);
            }
            else
            {
                System.out.println("没有人处理该请求！");
            }
        }
    }
}
//具体处理者角色2
class ConcreteHandler2 extends Handler
{
    public void handleRequest(String request)
    {
        if(request.equals("two"))
        {
            System.out.println("具体处理者2负责处理该请求！");
        }
        else
```

```
                {
                    if(getNext()!=null)
                    {
                        getNext().handleRequest(request);
                    }
                    else
                    {
                        System.out.println("没有人处理该请求！");
                    }
                }
            }
        }
```

- **Q2:结合实例，绘制命令模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析**

  A2：
  - **命令模式定义：**
    将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理
  - **优点：**
    - 降低系统的耦合度。命令模式能将调用操作的对象与实现该操作的对象解耦。
    - 增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，它满足“开闭原则”，对扩展比较灵活。
    - 可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令。
    - 方便实现 Undo 和 Redo 操作。命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复。
  - **缺点：**
    - 可能产生大量具体命令类。因为计对每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性。
  - **模式结构**
    - 抽象命令类（Command）角色：声明执行命令的接口，拥有执行命令的抽象方法 execute()。
    - 具体命令角色（Concrete Command）角色：是抽象命令类的具体实现类，它拥有接收者对象，并通过调用接收者的功能来完成命令要执行的操作。
    - 实现者/接收者（Receiver）角色：执行命令功能的相关操作，是具体命令对象业务的真正实现者。
    - 调用者/请求者（Invoker）角色：是请求的发送者，它通常拥有很多的命令对象，并通过访问命令对象来执行相关请求，它不直接访问接收者。
    - **模式结构图**



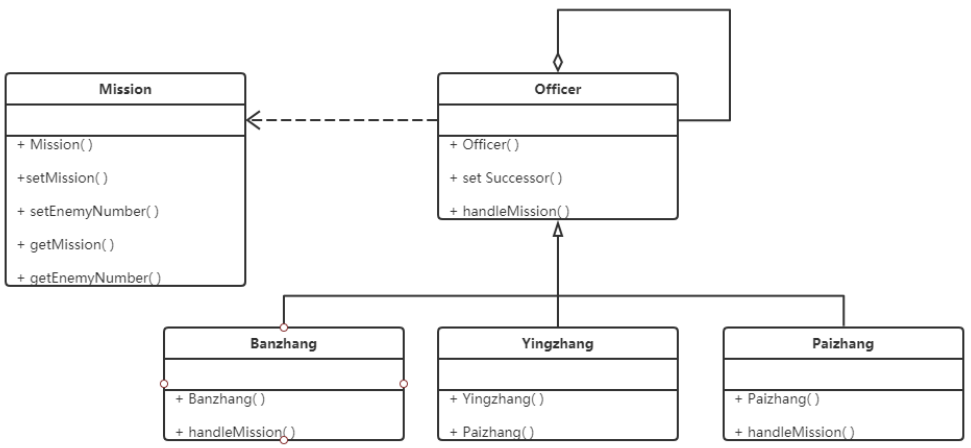


- **模式代码**

```
package command;
public class CommandPattern
{
    public static void main(String[] args)
    {
        Command cmd=new ConcreteCommand();
        Invoker ir=new Invoker(cmd);
        System.out.println("客户访问调用者的call()方法...");
        ir.call();
    }
}
//调用者
class Invoker
{
    private Command command;
    public Invoker(Command command)
    {
        this.command=command;
    }
    public void setCommand(Command command)
    {
        this.command=command;
    }
    public void call()
    {
        System.out.println("调用者执行命令command...");
        command.execute();
    }
}
//抽象命令
interface Command
{
    public abstract void execute();
}
//具体命令
```

```
class ConcreteCommand implements Command
{
    private Receiver receiver;
    ConcreteCommand()
    {
        receiver=new Receiver();
    }
    public void execute()
    {
        receiver.action();
    }
}
//接收者
class Receiver
{
    public void action()
    {
        System.out.println("接收者的action()方法被调用...");
    }
}
```

- **Q3:在军队中,一般根据战争规模的大小和重要性由不同级别的长官(Officer)来下达作战命令,情报人员向上级递交军情(如敌人的数量),作战命令需要上级批准,如果直接上级不具备下达命令的权力,则上级又传给上级,直到有人可以决定为止。现使用职责链模式来模拟该过程,客户类Client)模拟情报人员,首先向级别最低的班长(Banzhang)递交任务书(Mission),即军情,如果超出班长的权力范围,则传递给排长(Paizhang),排长如果也不能处理,则传递给营长(Yingzhang),如果营长也不能处理,则需要开会讨论。**
  设置这几级长官的权力范围分别是:
  (1) 敌人数量<10,班长下达作战命令;
  (2) 10≤敌人数量<50,排长下达作战命令;
  (3) 50≤敌人数量<200,营长下达作战命令;
  (4) 敌人数量≥200,需要开会讨论再下达作战命令。
  A3:




```
//Mission类
public class Mission {
    private String mission;
    private int enemyNumber;
```

```
    public Mission() {
        super();
    }
    public Mission(String mission, int enemyNumber) {
        super();
        this.mission = mission;
        this.enemyNumber = enemyNumber;
    }
    public String getMission() {
        return mission;
    }
    public void setMission(String mission) {
        this.mission = mission;
    }
    public int getEnemyNumber() {
        return enemyNumber;
    }

    public void setEnemyNumber(int enemyNumber) {
        this.enemyNumber = enemyNumber;
    }
}

//Officer类
public abstract class Officer {
    protected String name;
    protected Officer successor;

    public Officer(String name) {
        this.name = name;
    }
    public void setSuccessor(Officer successor) {
        this.successor = successor;
    }
    public abstract void handleMission(Mission mission);
}

//Banzhang类
public class Banzhang extends Officer{
    public Banzhang(String name) {
        super(name);
    }
    @Override
    public void handleMission(Mission mission) {
        if(mission.getEnemyNumber()>0 && mission.getEnemyNumber()<10){
            System.out.println(name+"接到"+mission.getMission()+"的军情"+",敌
人数量为"+mission.getEnemyNumber()+",可以下达作战指令");
        }else{
            if(this.successor!=null)
            {
                this.successor.handleMission(mission);
            }
        }
    }
}
```

```
//Paizhang类
public class Paizhang extends Officer {
    public Paizhang(String name) {
        super(name);
    }
    @Override
    public void handleMission(Mission mission) {
        if (mission.getEnemyNumber() >= 10 && mission.getEnemyNumber() < 50)
{
            System.out
                    .println(name + "接到" + mission.getMission()
                            + "的军情" + ",敌人数量
为" + mission.getEnemyNumber()
                            + ",可以下达作战指令");
        } else {
            if (this.successor != null) {
                this.successor.handleMission(mission);
            }
        }
    }
}

//Yingzhang类
public class Yingzhang extends Officer {
    public Yingzhang(String name) {
        super(name);
    }
    @Override
    public void handleMission(Mission mission) {
        if (mission.getEnemyNumber() >= 50 && mission.getEnemyNumber() < 200)
 {
            System.out
                    .println(name + "接到" + mission.getMission()
                            + "的军情" + ",敌人数量
为" + mission.getEnemyNumber()
                            + ",可以下达作战指令");
        } else {
            System.out
                    .println("接到" + mission.getMission() + "的军情" + ",敌
人数量为"
                            + mission.getEnemyNumber() + ",敌人太多了，开会讨
论！");
        }
    }
}

public class Client {
    public static void main(String[] args) {
        Officer Bangzhang = new Banzhang("王班长");
        Officer Paizhang = new Paizhang("赵排长");
        Officer Yingzhang = new Yingzhang("王营");
        Bangzhang.setSuccessor(Paizhang);
        Paizhang.setSuccessor(Yingzhang);
        Mission m1 = new Mission("mission1", 3);
```

```
        Bangzhang.handleMission(m1);
        Mission m2 = new Mission("mission2", 38);
        Bangzhang.handleMission(m2);
        Mission m3 = new Mission("mission3", 100);
        Bangzhang.handleMission(m3);
        Mission m4 = new Mission("mission4", 400);
        Bangzhang.handleMission(m4);
    }
}
```