

设计模式 (下下下)

中介者模式

需求

智能家庭

当客户要看电视时，各个设备可以协同工作，完成看电视这个目的。流程如下

闹钟响起->咖啡机开始泡咖啡->窗帘自动落下->电视机开始播放

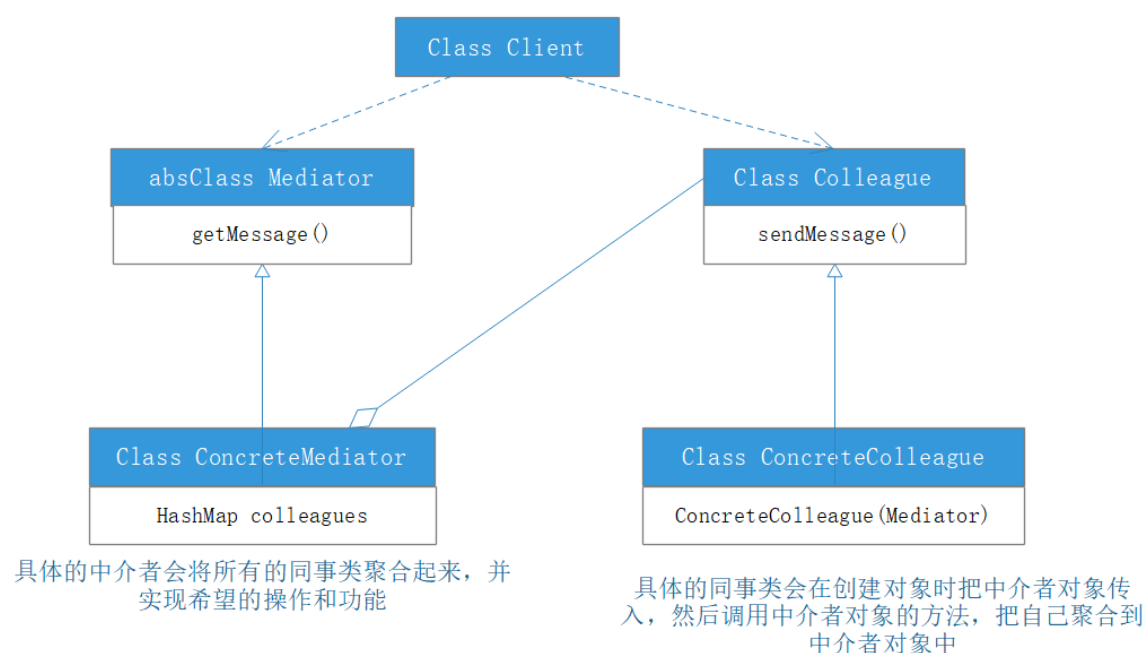
传统模式下带来的弊端：当闹钟响起时，我们要去通知咖啡机，咖啡机开始泡咖啡后又要去通知窗帘，然后又去通知电视机。在这种模式下，各个对象之间的耦合度很高（每种电器需要去组合下一个步骤中所需要使用的电器），并且当我们增加一个电器对象到指向了流程中时，可维护性是很差的

基本介绍

用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显示地相互引用，而是通过中介来引用各个对象。从而使其耦合松散，而且可以独立地改变它们之间的交互

中介者模式在多个类相互耦合，形成了网状结构的复杂耦合时使用，它可以将网状结构拆散为星型结构，即将多对多的关系转化为了多对一（中介）的关系

比如 MVC 模式中，C (Controлле) 是 M (Model) 和 V (View) 的中介者，在前后端交互时起到了中间人的作用



抽象中介者 Mediator：定义统一的接口用于各同事角色之间的通信，其中主要方法是一个（或多个）事件方法

具体中介者 ConcreteMediator：实现了抽象中介者所声明的事件方法。具体中介者知晓所有的具体同事类，并负责具体的协调各同事对象的交互关系，一般用集合来进行管理

抽象同事类 Colleague：定义出中介者到同事角色的接口。同事角色只知道中介者和自己的行为而不知道其余的同事角色。与其他的同事角色通信的时候，一定要通过中介者角色协作

具体同事类 ConcreteColleague：所有的具体同事类均从抽象同事类继承而来。实现自己的业务，在需要与其他同事通信的时候，就与持有的中介者通信，中介者会负责与其他的同事交互

代码实现

操作流程

1. 创建 ConcreteMediator 对象
2. 创建各个同事类对象，比如：闹钟，咖啡机，电视
3. 在创建同事类对象的时候，直接通过构造器，将自己聚合到 ConcreteMediator 中
4. 同事类对象调用 sendMessage() 方法，最终会去调用 ConcreteMediator 的 getMessage() 方法
5. getMessage() 会根据接收到的同事对象发出的消息来协调调用其它同事对象，完成任务

先创建同事抽象类

```
abstract public class Colleague {  
  
    private Mediator mediator;  
    public String name;  
  
    public Colleague(Mediator mediator, String name) {  
        this.mediator = mediator;  
        this.name = name;  
    }  
  
    abstract void sendMessage(String name);  
  
    public Mediator getMediator(){  
        return this.mediator;  
    }  
}
```

然后创建中介抽象类

```
abstract public class Mediator {  
  
    /**  
     * 将中介者对象加入到 Map 中  
     * @param colleague: 值  
     * @param name: 键  
     */  
    abstract public void register(Colleague colleague, String name);  
  
    /**  
     * 接收消息，具体的同事类发出的。是处理逻辑的核心方法
```

```

    */
    abstract public void getMessage(String name);
}

```

接着创建具体中介类

```

public class ConcreteMediator extends Mediator {

    Map<String, Colleague> colleagues;

    public ConcreteMediator(){
        colleagues = new HashMap<>();
    }

    @Override
    public void register(Colleague colleague, String name) {
        colleagues.put(name, colleague);
    }

    /**
     * 根据得到的消息完成对应的任务,协调具体的同事对象
     */
    @Override
    public void getMessage(String name) {

        //闹钟发出消息,说明咖啡机该开始弄咖啡了。以此类推
        if(colleagues.get(name) instanceof Alarm){
            System.out.println("闹钟响起");
        }else if(colleagues.get(name) instanceof CoffeeMachine){
            System.out.println("咖啡机开始弄咖啡");
        }else if(colleagues.get(name) instanceof Curtain){
            System.out.println("窗帘落下");
        }else if(colleagues.get(name) instanceof TV){
            System.out.println("电视打开");
        }
    }
}

```

创建具体同事类,这里以咖啡机为例,其它几个具体实现相同

```

public class CoffeeMachine extends Colleague {

    public CoffeeMachine(Mediator mediator, String name) {
        super(mediator, name);
        //注册自己
        this.getMediator().register(this, name);
    }

    @Override
    void sendMessage(String name) {
        this.getMediator().getMessage(name);
    }
}

```

最后创建客户端进行测试

```

public class Client {
    public static void main(String[] args) {

        Mediator mediator = new ConcreteMediator();

        Alarm alarm = new Alarm(mediator, "闹钟");
        CoffeeMachine coffee = new CoffeeMachine(mediator, "咖啡机");
        Curtain curtain = new Curtain(mediator, "窗帘");
        TV tv = new TV(mediator, "电视");

        alarm.sendMessage("闹钟");
        coffee.sendMessage("咖啡机");
        curtain.sendMessage("窗帘");
        tv.sendMessage("电视");

    }
}

```

闹钟响起
 咖啡机开始弄咖啡
 窗帘落下
 电视打开

在这种模式下，当我们新增一个具体同事类时，如果我们想让这个具体同事类加入到工作流程中，我们只需要修改具体中介类的 `getMessage()` 方法即可；但同样，带来的问题就是：当同事类太多时，中介者将会担负很大的职责，`getMessage()` 可能会变得很庞大并且难以维护

注意事项和细节

如果多个类互相之间相互耦合，形成了网状结构，使用中介者模式可以将网状结构分离成星型结构，进行解耦

优点

- 中介者模式减少了类间依赖，降低了耦合
- 将多对多关系变为了一对多关系，提高了灵活性

缺点

- 中介者承担了较大的责任，一旦中介者出现问题，系统将会受到影响
- 如果具体同事类过多，中介者的逻辑可能会变得庞大而复杂，难以维护

备忘录模式

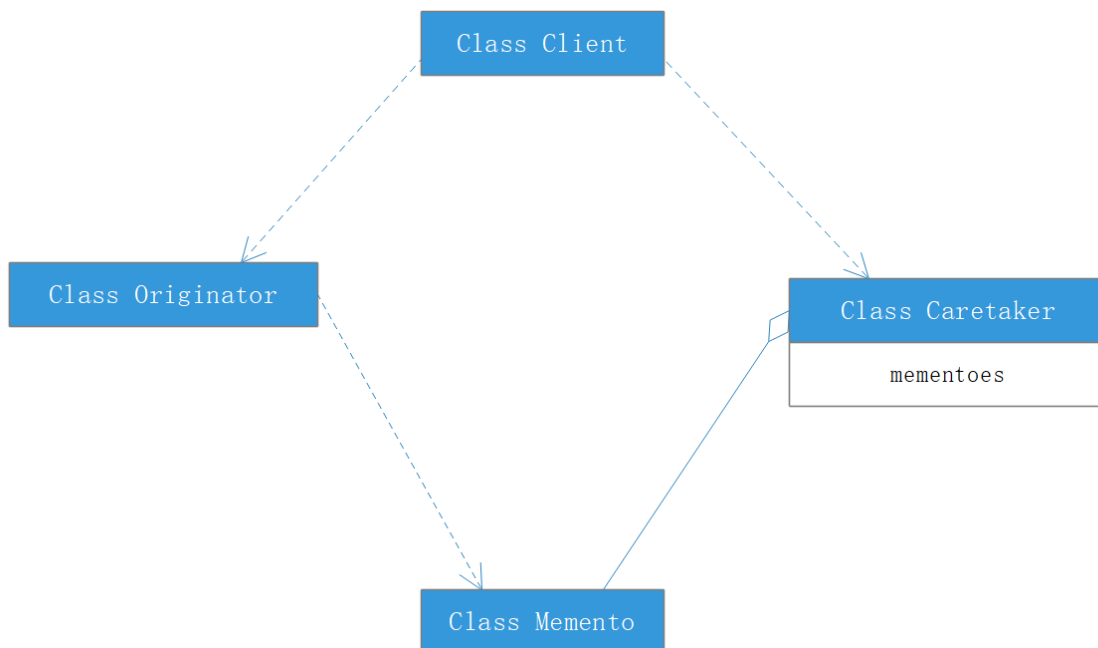
需求

角色状态复原：游戏中的角色在对战时可能会被上 debuff，但是角色又拥有一个备忘录技能，能将角色的状态恢复到被上 debuff 之前

传统模式的解决方案：一个不同的角色类，对应一个保存其对象状态的类 (备份类)。一个角色对象就对应一个保存角色对象状态的对象。在这种模式下，我们只是简单的 new 一个备份对象出来，然后把需要备份的对象的数据放到这个备份对象中，这种模式带来的问题就是：当游戏的对象很多时，需要对应的备份对象，开销很大，并且难以管理

基本介绍

备忘录模式指，在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样就可在此需要的时候将该对象恢复到原来保存的状态



Originator 原发器，被备份对象：记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据

Memento 备忘录：负责存储原发器对象的内部状态，但是具体需要存储哪些数据是由原发器对象来决定的，在需要的时候提供原发器需要的内部状态。PS：这里可以存储状态

Caretaker：对备忘录对象进行管理，但是不能对备忘录对象的内容进行操作

白箱和黑箱

备忘录模式可以分为白箱实现和黑箱实现，两种实现方式与备忘录角色提供的接口有关

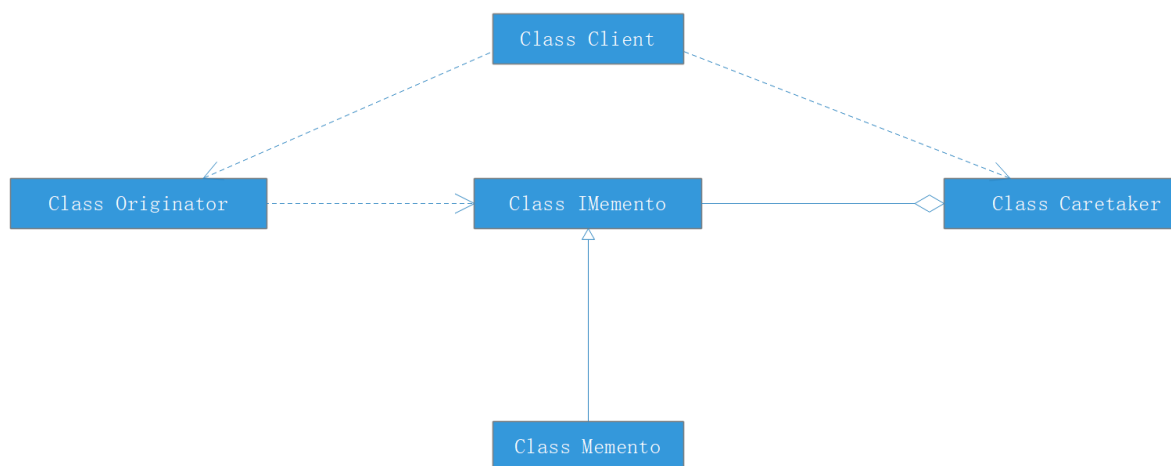
白箱

白箱实现的类图如上图所示，所以不再赘述。

- 备忘录角色对任何对象都提供一个（宽）接口，备忘录角色的内部所存储的状态就对所有对象公开。即白箱实现

- 白箱实现中原发器和备忘录负责人使用备忘录中的相同接口，使得负责人可以访问备忘录全部内容，并不安全
- 白箱实现对备忘录内容的保护靠的是程序员的自律，实现也很简单

黑箱



Memento 作为 原发器的内部类存在

黑箱模式的大致类图如上

备忘录角色 IMemento：空接口，不作任何实现。

发起人 (原发器) Originator：记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据。PS：这里 Memento 做为原发器的私有内部类，来存储备忘录对象。备忘录只能由原发器对象来访问它内部的数据，原发器外部的对象不能访问到备忘录对象的内部数据。

备忘录负责人 (管理者) Caretaker：对备忘录对象进行管理，但是不能对备忘录对象的内容进行操作或检查

窄接口和宽接口的使用

窄接口：在黑箱实现中，备忘录管理者 (Caretaker) 对象 (和其他除原发器对象之外的任何对象) 看到的是备忘录的窄接口 (narrow interface)，这个窄接口只允许它把备忘录对象传给其他的对象

宽接口：在白箱实现中，备忘录管理者对象可以看到一个宽接口 (wide interface)，这个宽接口允许它读取所有的数据，以便根据这些数据恢复这个发起人对象的内部状态

备忘录角色对任何对象都提供一个 (宽) 接口，备忘录角色的内部所存储的状态就对所有对象公开。即白箱实现

Memento 对象给 Originator 角色对象提供一个宽接口，而为其他对象提供一个窄接口，即黑箱实现

代码实现

白箱实现

首先创建原发器类

```

public class Originator {

    //状态信息
    private int buffState;

    public int getBuffState() {
        return buffState;
    }

    public void setBuffState(int buffState) {
        this.buffState = buffState;
    }

    /**
     * 保存当前状态信息，并返回备忘录对象
     * 返回对象用于加入备忘录管理对象中
     * @return
     */
    public Memento saveStateMemento(){
        return new Memento(buffState);
    }

    /**
     * 获得原来的状态信息，并更新当前对象状态信息
     * @param memento: 状态信息
     */
    public void getOldState(Memento memento){
        this.buffState = memento.getBuffState();
    }
}

```

然后创建备忘录类

```

public class Memento {

    private int buffState;

    public Memento(int buffState){
        this.buffState = buffState;
    }

    public int getBuffState() {
        return buffState;
    }
}

```

接着创建备忘录管理类

```

public class Caretaker {

    private List<Memento> mementos;

    public Caretaker(){
        mementos = new ArrayList<>();
    }
}

```

```

public void addMemento(Memento memento){
    mementos.add(memento);
}

public void deleMemento(Memento memento){
    mementos.remove(memento);
}

//获取第 index 个 Originator 对象被保存的状态
public Memento getMemento(int index){
    return mementos.get(index);
}

public int getSize(){
    return mementos.size();
}
}

```

最后创建客户端进行测试

```

public class Client {
    public static void main(String[] args) {

        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        System.out.println("当前英雄状态，有"+originator.getBuffState()+"个
debuff");

        //保存通过原发器状态信息创建的对应该备忘录对象到备忘录管理对象中
        caretaker.addMemento(originator.saveStateMemento());

        //修改状态信息
        originator.setBuffState(4);
        System.out.println("当前英雄状态，有"+originator.getBuffState()+"个
debuff");

        System.out.println("发动技能：时间管理");
        //恢复到上一次的状态。
        //如果在备忘录管理对象的集合中存在多个备忘录对象，则通过下标来选择需要恢复的状态
        originator.getOldState(caretaker.getMemento(caretaker.getSize()-1));
        System.out.println("当前英雄状态，有"+originator.getBuffState()+"个
debuff");
    }
}

```

```

当前英雄状态，有0个debuff
当前英雄状态，有4个debuff
发动技能：时间管理
当前英雄状态，有0个debuff

```

如果想要保存多个 originator 对象的不同时期的集合，可以把集合更改为

Map<Integer, List<Memento>>, k 为原发器对象编号, v 为这个原发器对象的备忘录列表

黑箱实现

黑箱实现中, Memento 对象给 Originator 角色对象提供一个宽接口, 而为其他对象提供一个窄接口, 即黑箱实现

我们先来写窄接口

```
/**
 * 备忘录窄接口
 */
public interface IMemento {
}
```

然后写原发器, 备忘录应该是原发器的私有内部类

```
/**
 * 原发器
 */
public class Originator {
    private int buffState;

    public int getBuffState() {
        return buffState;
    }

    public void setBuffState(int buffState) {
        this.buffState = buffState;
    }

    /**
     * 保存原发器状态到备忘录对象, 并返回备忘录对象以便备忘录管理者管理
     * @return
     */
    public IMemento saveMemento(){
        return new Memento(buffState);
    }

    /**
     * 恢复状态, 根据传入的备忘录对象的状态信息来赋值恢复
     * @param memento
     */
    public void getOldState(IMemento memento){
        this.buffState = ((Memento)memento).getBuffState();
    }

    /**
     * 私有内部类实现备忘录
     */
    private class Memento implements IMemento{

        private int buffState;

        private Memento(int buffState){
```

```

        this.buffState = buffState;
    }

    public int getBuffState() {
        return buffState;
    }
}
}

```

备忘录管理者

```

public class Caretaker {

    private List<IMemento> mementos;

    public Caretaker(){
        mementos = new ArrayList<>();
    }

    public void addMemento(IMemento memento){
        mementos.add(memento);
    }

    public void deleMemento(IMemento memento){
        mementos.remove(memento);
    }

    public IMemento getMemento(int index){
        return mementos.get(index);
    }
}

```

最后创建客户端进行测试

```

public class Client {
    public static void main(String[] args) {

        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        System.out.println("当前英雄状态, 有"+originator.getBuffState()+"个
debuff");

        //保存状态, 并交给备忘录管理者管理
        caretaker.addMemento(originator.saveMemento());
        //修改状态
        originator.setBuffState(8);
        System.out.println("当前英雄状态, 有"+originator.getBuffState()+"个
debuff");

        System.out.println("发动英雄技能: 时间的主人");
        //恢复状态
        originator.getOldState(caretaker.getMemento(0));
        System.out.println("当前英雄状态, 有"+originator.getBuffState()+"个
debuff");
    }
}

```

```
}  
}
```

```
当前英雄状态，有0个debuff  
当前英雄状态，有8个debuff  
发动英雄技能：时间的主人  
当前英雄状态，有0个debuff
```

白箱实现和黑箱实现的区别：外部能否访问备忘录的状态信息，备忘录是否具有安全等级

注意实现和细节

给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便的回到某个历史状态

封装了信息，用户无需关心保存过程和封装细节

缺点

- 如果原发器类的成员变量变量过多，则备份会占用较大资源

为了节约资源，备忘录模式可以和原型模式配合使用

解释器模式

需求

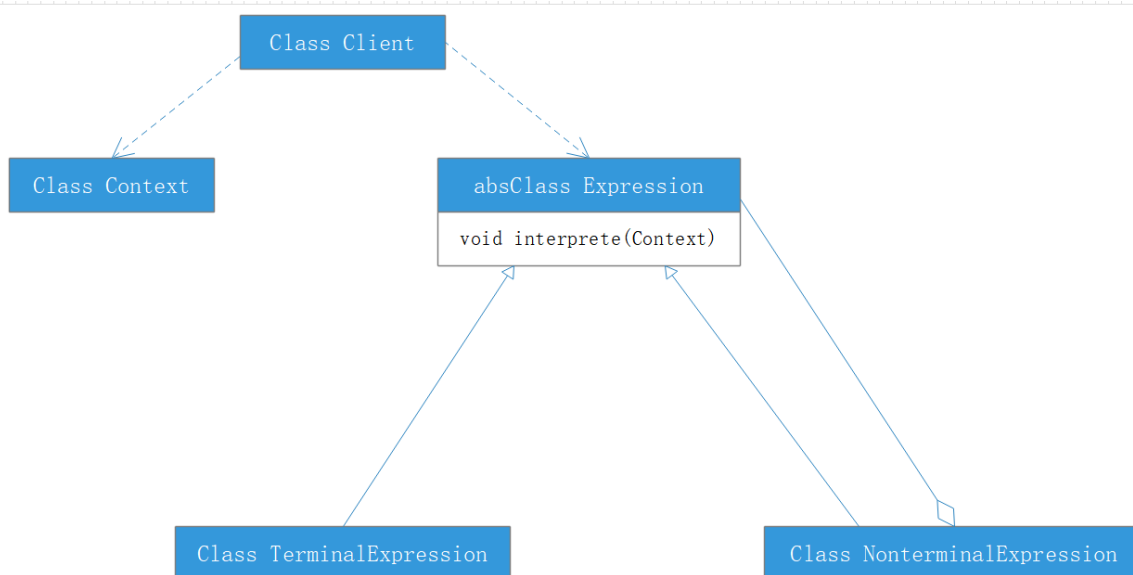
与(and)、或(or)、非(not)的一套解析器

传统方式下，我们可以编写一个方法，接收表达式结构，然后根据用户输入的值得到结果。但这种模式下弊端也很明显，如果我们想要增加新的运算符种类，甚至增加新的运算符，我们都需要去修改原来的代码，不利于扩展

基本介绍

在编译原理中，一个算术表达式通过**词法分析器**形成词法单元，而后这些词法单元再通过**语法分析器**构建语法分析树，最终成为一颗抽象的语法分析树。这里的词法分析器和语法分析器都可以看作是解释器

定义：给定一个语言 (在需求中可以看做是表达式)，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子 (表达式)



Expression (抽象表达式)：定义解释器的接口，约定解释器的解释操作

TerminalExpression (终结符表达式)：用来实现语法规则中和终结符相关的操作，不再包含其它的解释器，如果用组合模式来构建抽象语法树的话，就相当于组合模式中的叶子对象，可以有多种终结符解释器

NonterminalExpression (非终结符表达式)：用来实现语法规则中非终结符相关的操作，通常一个解释器对应一个语法规则，可以包含其它的解释器，如果用组合模式来构建抽象语法树的话，就相当于组合模式中的组合对象，可以有多种非终结符解释器

Context (上下文)：它包含了解释器之外一些其他的全局信息；通常包含各个解释器需要的数据，或是公共的功能

Client (客户端)：指的是使用解释器的客户端，通常在这里去把按照语言的语法做的表达式，转换为使用解释器对象描述的抽象语法树，然后调用解释操作

代码实现

在与或非中，非终结符表达式应该是 `and`，`or`，`not`，这些连接变量的操作，相应的终结符表达式就应该是具体的变量。

先创建一个抽象表达式

```
public interface BoolExp {

    /**
     * 解释上下文表达式，计算出结果
     * @param context: 表达式
     * @return: 结果
     */
    boolean evaluate(Context context);
    BoolExp replace(String var, BoolExp exp);
    BoolExp copy();
}
```

`and` 非终结符表达式

```

public class AndExp implements BoolExp {
    private BoolExp operand1;
    private BoolExp operand2;

    public AndExp(BoolExp oper1, BoolExp oper2) {
        operand1 = oper1;
        operand2 = oper2;
    }

    /**
     * 当运算的 operand1 时，会调用到 context.lookup(name);
     * 返回的是这个变量名所代表的 bool 值
     * @param c
     * @return
     */
    @Override
    public boolean evaluate(Context c) {
        return operand1.evaluate(c) && operand2.evaluate(c);
    }

    @Override
    public BoolExp copy() {
        return new AndExp(operand1.copy(), operand2.copy());
    }

    @Override
    public BoolExp replace(String var, BoolExp exp) {
        return new AndExp(operand1.replace(var, exp), operand2.replace(var,
exp));
    }
}

```

or 非终结符表达式

```

public class OrExp implements BoolExp {
    private BoolExp operand1;
    private BoolExp operand2;

    public OrExp(BoolExp oper1, BoolExp oper2) {
        operand1 = oper1;
        operand2 = oper2;
    }

    @Override
    public boolean evaluate(Context c) {
        return operand1.evaluate(c) || operand2.evaluate(c);
    }

    @Override
    public BoolExp copy() {
        return new OrExp(operand1.copy(), operand2.copy());
    }

    @Override
    public BoolExp replace(String var, BoolExp exp) {
        return new OrExp(operand1.replace(var, exp), operand2.replace(var, exp));
    }
}

```

```
}  
}
```

not 非终结符表达式

```
public class NotExp implements BoolExp {  
    private BoolExp operand1;  
  
    public NotExp(BoolExp oper1) {  
        operand1 = oper1;  
    }  
  
    @Override  
    public boolean evaluate(Context c) {  
        return operand1.evaluate(c);  
    }  
  
    @Override  
    public BoolExp copy() {  
        return new NotExp(operand1.copy());  
    }  
  
    @Override  
    public BoolExp replace(String var, BoolExp exp) {  
        return new NotExp(operand1.replace(var, exp));  
    }  
}
```

上下文

```
public class Context {  
    private Map<String, Boolean> context;  
  
    public Context() {  
        this.context = new HashMap<>();  
    }  
  
    //将变量名及其值加入上下文  
    public void assign(String name, boolean val) {  
        context.put(name, val);  
    }  
  
    public boolean lookUp(String name) {  
        return context.get(name);  
    }  
}
```

终结符表达式

```
public class VarExp implements BoolExp{  
  
    //变量名  
    private String name;  
  
    public VarExp(String name){  
        this.name = name;  
    }  
}
```

```

    }

    @Override
    public boolean evaluate(Context context) {
        return context.lookup(name);
    }

    @Override
    public BoolExp replace(String var, BoolExp exp) {
        if(var.equals(name)) {
            return exp.copy();
        } else {
            return new VarExp(name);
        }
    }

    @Override
    public BoolExp copy() {
        return new VarExp(name);
    }
}

```

对于一个与或非的计算过程，我们希望是这样的

1. 创建上下文
2. 创建非终结符，即变量
3. 将非终结符及其代表的值加入到上下文中，形成表达式
4. 进行非终结符表达式的创建，比如：(z and x) and (y and (not x)) 可以写为

```
new AndExp(new AndExp(z, x), new AndExp(y, new NotExp(x)))
```

5. 传入上下文变量和环境，对表达式进行运算

```
expression.evaluate(context);
```

客户端如下

```

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

        VarExp x = new VarExp("X");
        VarExp y = new VarExp("Y");
        VarExp z = new VarExp("Z");

        context.assign("X", true);
        context.assign("Y", false);
        context.assign("Z", true);

        //测试表达式 : (z and x) and (y and (not x))
        BoolExp expression = new AndExp(new AndExp(z, x), new AndExp(y, new
NotExp(x)));
        boolean result = expression.evaluate(context);
    }
}

```

```
System.out.println("(z and x) and (y and (not x))表达式结果: " + result);

//测试表达式 : (z and x) or (y and (not x))
BoolExp expression1 = new OrExp(new AndExp(z, x), new AndExp(y, new
NotExp(x)));
boolean result1 = expression1.evaluate(context);
System.out.println("(z and x) or (y and (not x))表达式结果: " + result1);
}
}
```

```
(z and x) and (y and (not x))表达式结果: false
(z and x) or (y and (not x))表达式结果: true
```

注意事项和细节

优点

- 易于扩展新的语法和实现新语法，每条新的语法规则需要用一个新的解释器来解释执行。扩展新的语法规则只需要新建解释器即可

缺点

- 采用递归调用方式：每个非终结符表达式只关心与自己有关的表达式，每个表达式需要知道最终的结果，则必须一层一层的递归。这就带来了效率问题
- 不适合语法规则较为复杂的表达式，因为每个语法规则都需要产生一个非终结符表达式

状态模式

需求

英雄状态：

我们假设英雄有如下四个移动状态

- 正常
- 加速
- 减速
- 眩晕

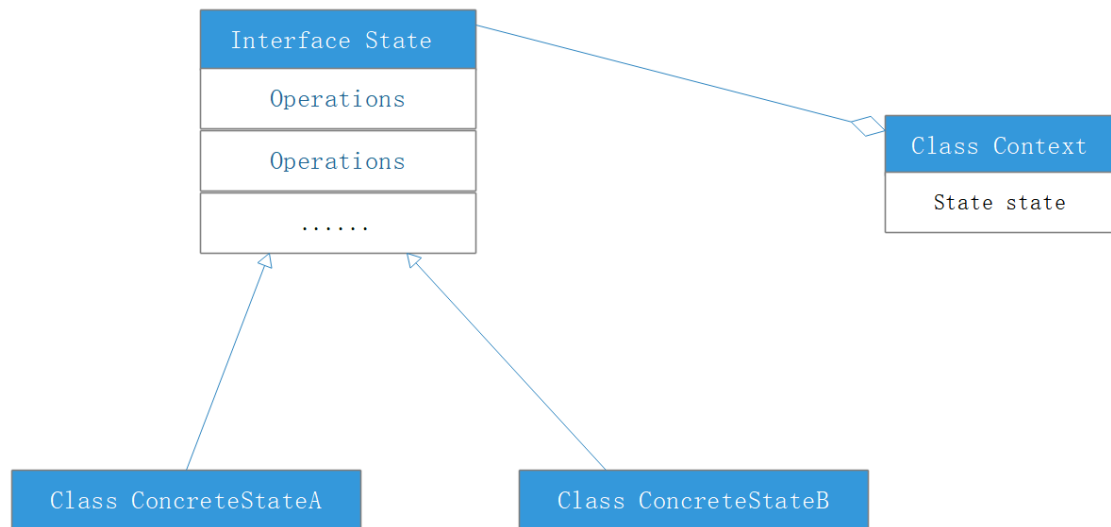
传统模式下，我们可以很好地联想到解决方案：使用 if else 来判断当前的状态码是什么，执行相应的操作即可，但是这样带来的问题就是代码的可维护性和可拓展性会降低

基本介绍

允许对象在内部状态改变时改变它的行为，这样对象看起来就像是修改了它的类

状态模式：主要用来解决对象在多种状态转换时，需要对外输出不同的行为的问题。状态和行为是一一对应的，状态之间可以相互转换

当一个对象的内在状态改变时，其行为也会发生变化，这个对象看起来像是改变了其类。比如人的三个年龄状态，小孩，成年人，老人，状态不同时，行为也会发生变化



Context 上下文：通常用来定义客户端需要的接口，同时维护 State 对象，该对象代表上下文当前状态

State 状态接口：用来封装与上下文的一个特定状态所对应的行为

ConcreteState 具体的状态角色：每个子类实现一个与 Context 的一个状态相关的具体行为

代码实现

首先定义状态接口

```
public interface State {

    /**
     * 处理移动状态，由子类来具体处理
     * @param hero
     */
    void move(Hero hero);
}
```

然后先写一个空的 Hero 类，和带空实现方法的四个具体状态类

保证不报错后，变化 Hero 类内容，这个类就是 Context 类

```
public class Hero {

    /**
     * 正常状态
     */
    public static final State NORMAL_STATE = new NormalState();

    /**
```

```

    * 减速状态
    */
    public static final State SLOW_STATE = new SlowState();

    /**
     * 加速状态
     */
    public static final State QUICK_STATE = new QuickState();

    /**
     * 眩晕状态
     */
    public static final State DIZZ_STATE = new DizzState();

    //表示英雄当前状态
    private State state = NORMAL_STATE;

    //英雄移动线程，便于观察状态变化时的行为变化
    private Thread moveThread;

    /**
     * 设置状态
     * @param state: 新状态
     */
    public void setState(State state){
        this.state = state;
    }

    /**
     * 判断是否在移动
     */
    public boolean isMoving(){
        return moveThread != null && !moveThread.isInterrupted();
    }

    /**
     * 开始移动
     */
    public void startMove(){
        if(isMoving()){
            return;
        }
        final Hero hero = this;
        moveThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while(!moveThread.isInterrupted()){
                    state.move(hero);
                }
            }
        });
        System.out.println("英雄开始移动");
        moveThread.start();
    }

    public void stopMove(){
        if(isMoving()){
            moveThread.interrupt();
        }
    }

```

```

    }
    System.out.println("英雄停止移动");
}
}

```

然后编写四个具体状态类

```

public class DizzState implements State {

    @Override
    public void move(Hero hero) {
        System.out.println("英雄被眩晕");

        try {
            Thread.sleep(2500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //恢复正常状态
        hero.setState(Hero.NORMAL_STATE);
        System.out.println("眩晕状态结束，英雄恢复正常状态");
    }
}

```

```

public class NormalState implements State {
    /**
     * 正常移动不输出信息,, 不然线程循环调用会刷屏
     * @param hero
     */
    @Override
    public void move(Hero hero) {

    }
}

```

```

public class QuickState implements State {

    @Override
    public void move(Hero hero) {
        System.out.println("英雄加速");

        try {
            Thread.sleep(3500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //恢复正常状态
        hero.setState(Hero.NORMAL_STATE);
        System.out.println("加速状态结束，英雄恢复正常状态");
    }
}

```

```

public class SlowState implements State {

    @Override

```

```

public void move(Hero hero) {
    System.out.println("英雄被减速");

    try {
        Thread.sleep(1500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    //恢复正常状态
    hero.setState(Hero.NORMAL_STATE);
    System.out.println("减速状态结束，英雄恢复正常状态");
}
}

```

最后编写客户端进行测试

```

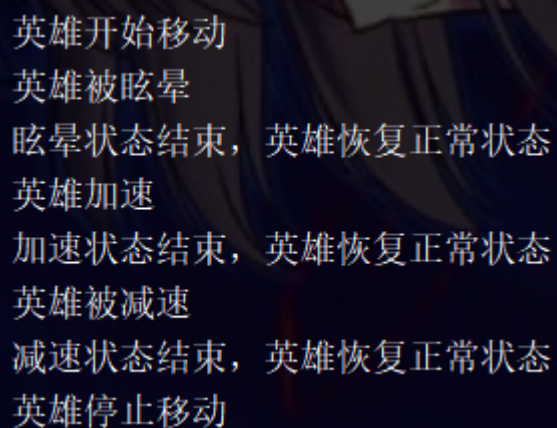
public class Client {
    public static void main(String[] args) throws InterruptedException {

        Hero hero = new Hero();
        hero.startMove();

        hero.setState(Hero.DIZZ_STATE);
        //睡眠，在第一次改变状态又变回普通状态后主线程直接跑完，hero对象中的
        //移动线程就一直执行普通状态的 move 方法了
        Thread.sleep(5000);
        hero.setState(Hero.QUICK_STATE);
        Thread.sleep(5000);
        hero.setState(Hero.SLOW_STATE);
        Thread.sleep(5000);

        hero.stopMove();
    }
}

```



英雄开始移动
 英雄被眩晕
 眩晕状态结束，英雄恢复正常状态
 英雄加速
 加速状态结束，英雄恢复正常状态
 英雄被减速
 减速状态结束，英雄恢复正常状态
 英雄停止移动

很显然，在这种模式下，我们消除了传统模式下的 if else 结构，在增加新状态时只需要让新的状态类实现状态接口，并将其聚合将新的状态类聚合进上下文成为静态常量即可

注意事项和细节

优点

- 结构清晰：避免了复杂的 if else / switch case 结构，提高了可维护性可扩展性
- 遵循单一职责原则和 ocp 原则：每一个子类都只是一种状态。增加新状态时只需要让新的状态类实现状态接口，并将其聚合将新的状态类聚合进上下文成为静态常量即可
- 封装性好：状态变换对客户端屏蔽，客户端只需要知道上下文有哪些状态即可

缺点

- 状态过多的话可能会有太多具体状态类，导致类爆炸

使用场景

- 行为随着状态改变而改变。可以使用状态模式将状态和行为分离开，再通过上下文将其关联起来。可以在运行期间通过改变状态，就能够调用到该状态对应的状态处理对象上，执行新状态相应的行为
- 代码含有大量的分支语句或 switch case 语句，而这些分支的选择依赖于对象的状态

策略模式

需求

鸭子可以有野鸭，北京鸭，玩具鸭，这些鸭子有可能各种做法，比如：啤酒鸭，老鸭汤，北京烤鸭

要求：显示鸭子可以有的做法

传统模式下：我们可以直接写一个鸭子的抽象类并写出具体的做法，然后创建具体鸭子实现抽象类并视情况重写。但是在所举的所有鸭子种类中，玩具鸭如果也继承了鸭子抽象类，则需要重写父类所有的鸭子做法，这显然是非常麻烦的

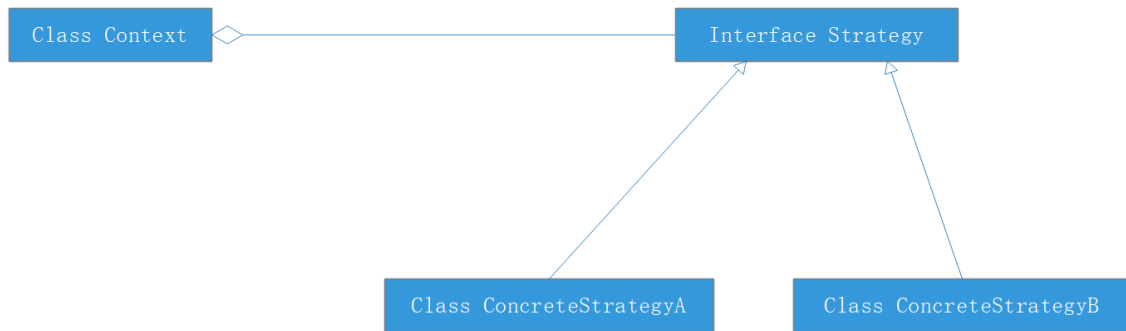
基本介绍

定义一组算法，将每个算法都封装起来，并且使他们之间可以互换。

策略模式的决定权在用户，系统本身提供不同算法的实现，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可

该模式体现了三个设计原则

- 针对接口编程（依赖倒转）
- 远离继承（合成复用）
- 变与不变分离（OCP）



Strategy: 策略接口，定义了一个公共接口，各种不同的具体策略以不同的方式实现这个接口，上下文使用这个接口调用不同的具体策略，一般使用接口或抽象类实现

ConcreteStrategy: 实现策略接口，提供具体的算法实现

Context: 持有一个策略类的引用，并最终交给客户调用

代码实现

在上述需求中，我们将每一种鸭子做法都抽象成一种策略接口，然后进行具体的策略实现。这里我只以能否做成北京烤鸭为例，LaoYaTang 的实现交给读者自行完成

我们先定义一个北京烤鸭的策略接口

```
public interface BeijingKaoYa {  
  
    void doDuck();  
}
```

然后创建两个实现类

```
public class CanBeijingKaoYa implements BeijingKaoYa {  
    @Override  
    public void doDuck() {  
        System.out.println("这鸭子，可以啊，做成北京烤鸭吧");  
    }  
}
```

```
public class NotBeijingKaoYa implements BeijingKaoYa {  
    @Override  
    public void doDuck() {  
        System.out.println("这鸭子八行，不能做成北京烤鸭");  
    }  
}
```

然后创建鸭子的抽象类，这个类及其子类就是上下文 Context，最终会交给客户端调用

```
abstract public class Duck {  
  
    private BeijingKaoYa beijingKaoYa;  
    private LaoYaTang laoYaTang;
```

```

private String name;

public Duck(String name){
    this.name = name;
}
/**
 * 显示鸭子种类
 */
abstract public void whatDuck();

public void canBeiJinKaoYa(){
    if (beiJingKaoYa != null){
        beiJingKaoYa.doDuck();
    }
}

public void canLaoYaTang(){
    if(laoYaTang != null){
        laoYaTang.doDuck();
    }
}

public void setBeiJingKaoYa(BeiJingKaoYa beiJingKaoYa) {
    this.beiJingKaoYa = beiJingKaoYa;
}

public void setLaoYaTang(LaoYaTang laoYaTang) {
    this.laoYaTang = laoYaTang;
}

public String getName() {
    return name;
}
}

```

在上下文中，我们会聚合具体的策略，调用其策略算法，以完成功能需求，控制具体对象的行为
具体北京鸭类

```

public class BeiJingDuck extends Duck {

    public BeiJingDuck(String name) {
        super(name);
        //设置北京鸭能做成北京烤鸭和老鸭汤
        setBeiJingKaoYa(new CanBeiJingKaoYa());
        setLaoYaTang(new CanLaoYaTang());
    }

    @Override
    public void whatDuck() {
        System.out.println("这是"+getName());
    }
}

```

客户端调用

```

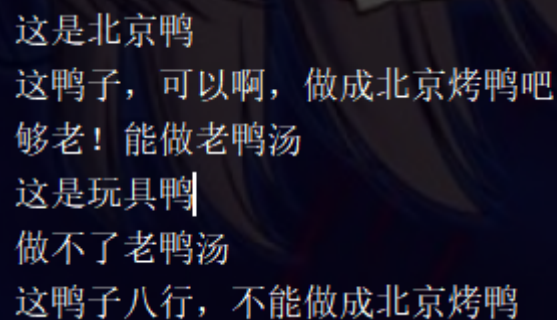
public class Client {
    public static void main(String[] args) {

        Duck peiKing = new BeijingDuck("北京鸭");
        peiKing.whatDuck();
        peiKing.canBeiJinKaoYa();
        peiKing.canLaoYaTang();

        Duck toyDuck = new ToyDuck("玩具鸭");
        toyDuck.whatDuck();
        toyDuck.canLaoYaTang();
        toyDuck.canBeiJinKaoYa();

    }
}

```



这是北京鸭
 这鸭子，可以啊，做成北京烤鸭吧
 够老！能做老鸭汤
 这是玩具鸭
 做不了老鸭汤
 这鸭子八行，不能做成北京烤鸭

在这种模式下，具体的使用者和实现者分离开来，使得我们可以有选择的去使用具体的实现者，但同样这种模式也会带来弊端

注意事项和细节

策略模式是一种简单常用的模式，我们在进行开发的时候，会经常有意无意地使用它（比如 Comparetor 的匿名内部类），一般来说，策略模式不会单独使用，跟模版方法模式、工厂模式等混合使用的情况比较多

优点

- 实现自同一个策略接口之间具体策略类可以自由切换
- 易于扩展，新增一种策略只需要对上下文的抽象类简单修改即可，即将其聚合进上下文中

缺点

- 维护各个策略类会给开发带来额外开销
- 如果很多种策略都有多种具体策略，则很容易产生类爆炸
- 必须对客户端暴露所有的策略类，因为使用哪种策略是由客户端来决定的，并且客户端应该了解各种策略之间的区别，否则后果很严重。比如，客户端要使用一个容器，有链表实现的，也有数组实现的，客户端需要明白链表和数组有什么区别。从这里来说违背了迪米特法则

策略模式和状态模式的区别

二者都可以减少大量的分支结构

状态模式根据状态的变化来选择相应的行为，不同的状态对应不同的类，每个状态对应的类实现了该状态的功能。在实现功能的同时还会维护状态数据的变化。这些对应不同状态的处理类是不可相互替换的

策略模式是根据需求或是客户端参数来选择相应的实现类，实现类之间是平等的，可互换的

责任链模式

需求

采购审批项目，具体流程是

1. 采购员采购教学器材
2. 如果金额 $\leq 5k$ ，由教学主任审批
3. 如果金额 $> 5k$ 并 $\leq 1w$ ，由院长审批
4. 如果金额 $> 1w$ 并 $\leq 3w$ ，由副校长审批
5. 如果金额 $> 3w$ ，由校长审批

传统方式：直接 if else 来调用审批者进行审批。

传统模式存在的很大问题是

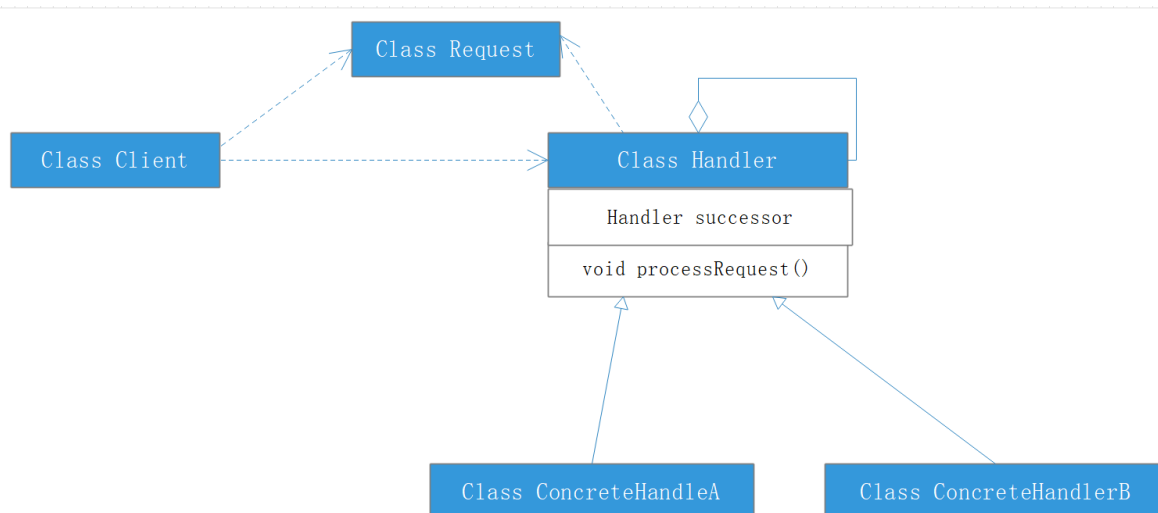
- 如果每个级别审批者可以负责的金额发生变化，则客户端也需要发生变化
- 客户端必须明确知道审批级别和访问方式

这种情况下对一个采购请求进行处理的过程和审批者之间就存在强耦合关系，我们可以使用职责链模式来解耦这种关系

基本介绍

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

所以职责链模式通常每个接收者都包含另一个接收者的引用



Handler：抽象处理类主要包含一个指向下一个处理类的成员变量和一个处理请求的方法。
processRequest() 方法主要就是如果满足处理的条件，则交由本类来处理，否则交给 successor 来处理

ConcreteHandler：具体处理类，在这个类中，实现对在它职责范围内的请求的具体处理。如果无法处理，就继续转发给下一个

Request：请求类

Client：客户端调用请求类，向处理类发送该请求

代码实现

首先，我们封装一个请求类

```
/**
 * 请求类
 */
public class PurchaseRequest {
    //请求类型
    private int type;
    //请求金额
    private double price;
    //id
    private int id;

    public PurchaseRequest(int type, double price, int id) {
        this.type = type;
        this.price = price;
        this.id = id;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }
}
```

```

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}

```

接着是抽象处理类

```

abstract public class Approver {

    //下一个处理者
    Approver successor;

    //名字
    String name;

    public Approver(String name) {
        this.name = name;
    }

    /**
     * 设置下一个处理者
     */
    public void setSuccessor(Approver successor) {
        this.successor = successor;
    }

    /**
     * 处理请求的方法，具体交给子类完成
     */
    abstract void processRequest(PurchaseRequest request);
}

```

教学主任级别的处理类

```

public class DepartmentApprover extends Approver {

    public DepartmentApprover(String name) {
        super(name);
    }

    /**
     * 如果本类的处理方法能够处理，则交给本类处理，否则将请求递交给下一个
     */
}

```

```

@Override
void processRequest(PurchaseRequest request) {
    if(request.getPrice() <= 5000){
        System.out.print("申请编号: "+request.getId()+" ");
        System.out.println("教学主任"+name+"审批 <= 5k的费用");
    }else{
        System.out.println("教学主任"+name+"没这个权限");
        successor.processRequest(request);
    }
}
}

```

院长级别的处理类

```

public class CollegeApprover extends Approver {

    public CollegeApprover(String name){
        super(name);
    }

    @Override
    void processRequest(PurchaseRequest request) {
        if(request.getPrice() > 5000 && request.getPrice() <= 10000){
            System.out.print("申请编号: "+request.getId()+" ");
            System.out.println("院长"+name+"审批 > 5k 并 <= 1w 的费用");
        }else{
            System.out.println("院长"+name+"没这个权限");
            successor.processRequest(request);
        }
    }
}

```

副校长级别的处理类

```

public class ViceschoolMasterApprover extends Approver {

    public ViceschoolMasterApprover(String name) {
        super(name);
    }

    @Override
    void processRequest(PurchaseRequest request) {
        if(request.getPrice() > 10000 && request.getPrice() <= 30000){
            System.out.print("申请编号: "+request.getId()+" ");
            System.out.println("副校长"+name+"审批 > 1w 并 <= 3w 的费用");
        }else{
            System.out.println("副校长"+name+"没这个权限");
            successor.processRequest(request);
        }
    }
}

```

校长级别的处理类

```

public class SchoolManagerApprover extends Approver {

```

```

public SchoolManagerApprover(String name) {
    super(name);
}

@Override
void processRequest(PurchaseRequest request) {
    if(request.getPrice() > 30000){
        System.out.print("申请编号: "+request.getId()+" ");
        System.out.println("校长"+name+"审批 >3w 的费用");
    }else{
        successor.processRequest(request);
    }
}
}

```

客户端测试

```

public class Client {
    public static void main(String[] args) {

        //新建一个请求
        PurchaseRequest request = new PurchaseRequest(1, 349000, 1);

        //新建一个教学主任处理者
        Approver department = new DepartmentApprover("张三");
        //新建院长处理者
        Approver college = new CollegeApprover("李四");
        //以此类推
        Approver viceschool = new ViceSchoolMasterApprover("王五");
        Approver schoolManager = new SchoolManagerApprover("赵六");

        //设置传递链，形成环状，这样在任何情况下都会找到一个人去处理请求
        //但是环状并不是必须的，需要视需求而定
        department.setSuccessor(college);
        college.setSuccessor(viceschool);
        viceschool.setSuccessor(schoolManager);
        schoolManager.setSuccessor(department);

        //选择给谁发送请求
        department.processRequest(request);
    }
}

```

教学主任张三没这个权限
 院长李四没这个权限
 副校长王五没这个权限
 申请编号: 1, 校长赵六审批 >3w 的费用

注意事项和细节

优点

- 请求者与接收者解耦：在职责链模式中，请求者并不知道接收者是谁，也不知道具体如何处理，请求者只是负责发送请求即可。并且职责对象之间也不需要直到其它职责对象内部的实现细节
- 动态组合职责：职责链模式会把功能处理分散到单独的职责对象里面，然后在使用的时候可以动态组合职责形成职责链，可以灵活的改变职责链的传递过程
- 增加新的处理类很方便

缺点

- 产生很多细粒度对象：职责链模式会把功能处理分散到单独的职责对象里面，也就是每个职责对象只是处理一个方面的功能，要把整个业务处理完，可能需要大量的职责对象来进行组合
- 链的有效性：职责链模式的每个职责对象只负责自己处理的那一部分，因此可能会出现某个请求，把整个链传递完了，都没有职责对象处理它。这就需要使用职责链模式的时候注意，需要提供默认的处理或适当的结束职责链，并且注意构建的链的有效性
- 在链比较长的情况下，可能会影响性能

纯与不纯的职责链

纯的职责链：一个纯的职责链模式要求**一个具体处理对象要么承担请求的全部处理逻辑，或把请求完全交给下一个具体处理对象**

不纯的职责链：不纯的职责链模式**允许某个请求被一个具体处理者处理部分后再继续向下传递**

责任链模式与状态模式的区别

二者都可以简化大量的分支结构

职责链模式中，多个处理对象都有机会去处理请求，这些对象链成一条链，并传递，直到有一个对象能够处理这个请求为止。并且链的设置是在客户端完成的

状态模式中，客户端通过设置上下文当前状态来调控上下文对象中的处理逻辑，是一个对象的内在发生改变。并且在状态模式中，需要让每个状态对象知道其下一个处理的对象是谁（比如说，中的减速buff后，英雄应该是什么状态）。而职责链模式只需要按照客户端设置的链传递请求即可

总结

总体来说设计模式分为三大类

- 创建型模式 (**5种**)：单例模式，工厂方法模式，抽象工厂模式，建造者模式，原型模式
- 结构型模式 (**7种**)：适配器模式，装饰者模式，代理模式，外观模式，桥接模式，组合模式，享元模式
- 行为模式 (**11种**)：策略模式，模板方法模式，观察者模式，迭代器模式，责任链模式，命令模式，备忘录模式，状态模式，访问者模式，中介者模式，解释器模式

创建型模式

单例模式 (Singleton)：保证一个类仅有一个实例，并提供一个访问它的全局访问点

工厂方法 (Factory Method): 定义一个创建对象的接口, 让其子类自己决定实例化哪一个工厂类, 工厂模式使其创建过程延迟到子类进行

抽象工厂 (Abstract Factory): 提供一个创建一系列相关或相互依赖对象的接口, 而无需指定它们具体的类

建造者模式 (Builder): 将一个复杂对象的构建与它的表示分离, 使得同样的构建过程可以创建不同的表示

原型模式 (Prototype): 用原型实例指定创建对象的种类, 并且通过拷贝这些原型来创建新的对象

结构型模式

==适配器模式 (Adapter) ==: 适配器模式把一个类的接口变换成客户端所期待的另一种接口, 从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作

装饰模式 (Decrator): 装饰模式是在不必改变原类文件和使用继承的情况下, 动态的扩展一个对象的功能。它是通过创建一个包装对象, 也就是装饰来包裹真实的对象

代理模式 (Proxy): 为一个对象提供一个替身, 以控制对这个对象的访问, 即通过代理对象访问目标对象

外观模式(Facade): 为子系统中的一组接口提供一个一致的界面, 外观模式定义了一个高层接口, 这个接口使得这一子系统更加容易使用

桥接模式 (Bridge): 将抽象部分与实现部分分离, 使它们都可以独立的变化

组合模式 (Composite): 允许你将对象组合成树形结构来表现"整体-部分"层次结构。组合能让客户以一致的方法处理个别对象以及组合对象

享元模式 (Flyweight): 运用共享技术有效地支持大量细粒度的对象

行为型模式

策略模式 (Strategy): 定义一组算法, 将每个算法都封装起来, 并且使他们之间可以互换

模板方法模式 (Template Method): 定义一个操作中的流程骨架, 而将流程中的一些不固定的步骤延迟到子类中, 使得子类可以不改变该流程结构的情况下重定义该流程的某些特定步骤

观察者模式 (Observer): 定义对象间的一种一对多的依赖关系, 当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新

迭代器模式 (Iterator): 提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示

责任链模式 (Chain of Responsibility): 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

命令模式 (Command): 将一个请求封装为一个对象, 从而使你可以用不同的请求对客户进行参数化, 对请求排队和记录请求日志, 以及支持可撤销的操作

备忘录模式 (Memento): 在不破坏封装性的前提下, 捕获一个对象的内部状态, 并在该对象之外保存这个状态。这样就可以将该对象恢复到原先保存的状态

状态模式 (State): 允许对象在内部状态改变时改变它的行为, 这样对象看起来就像是修改了它的类

访问者模式 (Visitor): 表示一个作用于其对象结构中的各元素的操作, 它使你可以在不改变各元素类的前提下定义作用于这些元素的新操作

中介者模式 (Mediator): 用一个中介对象来封装一系列的对象交互, 中介者使各对象不需要显示地相互引用, 而是通过中介来引用各个对象。从而使其耦合松散, 而且可以独立地改变它们之间的交互

解释器模式 (Interpreter): 给定一个语言 (在需求中可以看做是表达式), 定义它的文法表示, 并定义一个解释器, 这个解释器使用该标识来解释语言中的句子 (表达式)

Last Last

设计模式的四个阶段

1. 听说过设计模式, 但是没有去了解过
2. 学了几个模式后, 感觉很简单, 于是到处想着要用自己学过的模式, 但是并不知道应不应该用
3. 学完全部模式时, 感觉很多模式太相似了, 无法很清晰的知道各模式之间的区别、联系, 在使用时分不清要使用那种模式
4. 模式已熟记于心, 无意中便使用了设计模式并且是正确的, 这时便是融会贯通

河冰结合, 非一之日寒; 积土成山, 非斯须之作

设计模式的熟练使用并不是学了即可的, 想要熟练掌握, 必须经过长期的实践, 大量的阅读源码, 才有机会创造出优美的程序

路漫漫其修远兮, 愿诸君共勉