设计模式

1.策略模式+工厂模式

(1). 策略模式

定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换

策略模式把对象本身和运算规则区分开来，因此我们整个模式也分为三个部分。

环境类(Context):用来操作策略的上下文环境，也就是我们游客。抽象策略类(Strategy):策略的抽象，出行方式的抽象具体策略类(ConcreteStrategy):具体的策略实现，每一种出行方式的具体实现。

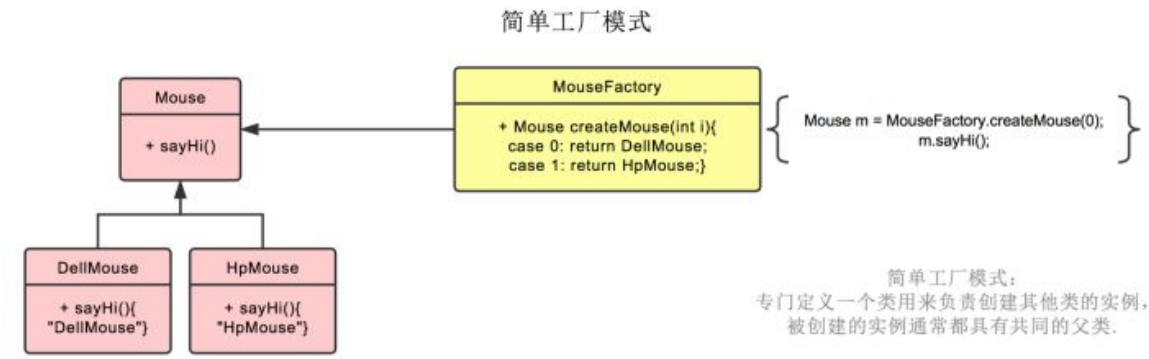
> 个人理解，策略方式可以把client 层面和处理层面分开，对于client的层面，只需要告诉要进行的行为的名字就可以了，比如在使用优惠券时进行八五折，那么就输入八五折相关规定好的字眼就可以了，不需要了解后面的进行操作是怎么样的。策略者模式通常用于各种不同的选择，在优惠券中就可以是1折，2折，各种折扣等。

(2).工厂模式

作用:

1. 当调用者想创建一个对象时，只需要知道名称就可以在工厂获取具体的对象。
2. 扩展性较强，当想增加一个类时，在工厂中进行增加即可。
3. 调用者可以只关注到产品的接口，不需要了解内部的实现。

一、简单工厂模式



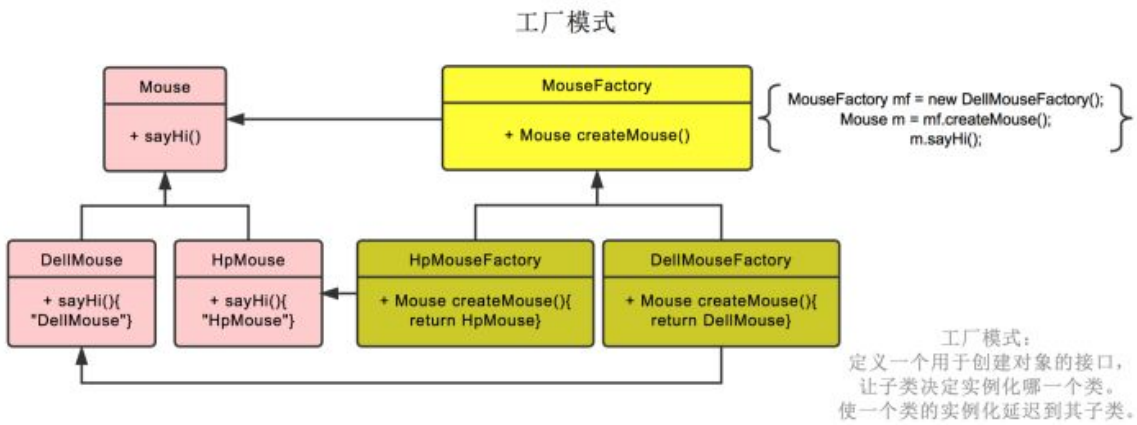
只有一个工厂来创建其他类的实例，在这种情况下是一个项目比较简单的时候可以使用，像dao层中便可以使用一个工厂来创建实例。

在图上的例子便是将鼠标分为了两种类型，一种是惠普的鼠标，一种是戴尔的鼠标.

```
public class MouseFactory {
    private static Map<String, Mouse> mouseMap = new HashMap<>();
    static {

        mouseMap.put("Dell", new DeliMouse());

        mouseMap.put("Hp", new HpMouse());

    }
```

```
    public static Mouse getMouse(String mouseType) {

        return mouseMap.get(mouseType);
    }
}
//工厂类与策略模式进行结合
public class Context {
    private Mouse mouse;

    public Context(Mouse mouse){
        this.mouse = mouse;
    }

    public String executeStrategy(){
        return mouse.createMouse();
    }
}

public class StrategyPattern {
    public static void main(String[] args) {
        //这一部是工厂来的
        Mouse mouse = Factory.getMouse("Hp");
        Context context = new Context(mouse);
        System.out.println(context.executeStrategy());
    }
}
```

二、工厂模式



工厂模式也就是鼠标工厂是个父类，有生产鼠标这个接口。
得力鼠标工厂，惠普鼠标工厂继承它，可以分别生产戴尔鼠标，惠普鼠标。
生产哪种鼠标不再由参数决定，而是创建鼠标工厂时，由戴尔鼠标工厂创建。
后续直接调用鼠标工厂.生产鼠标()即可

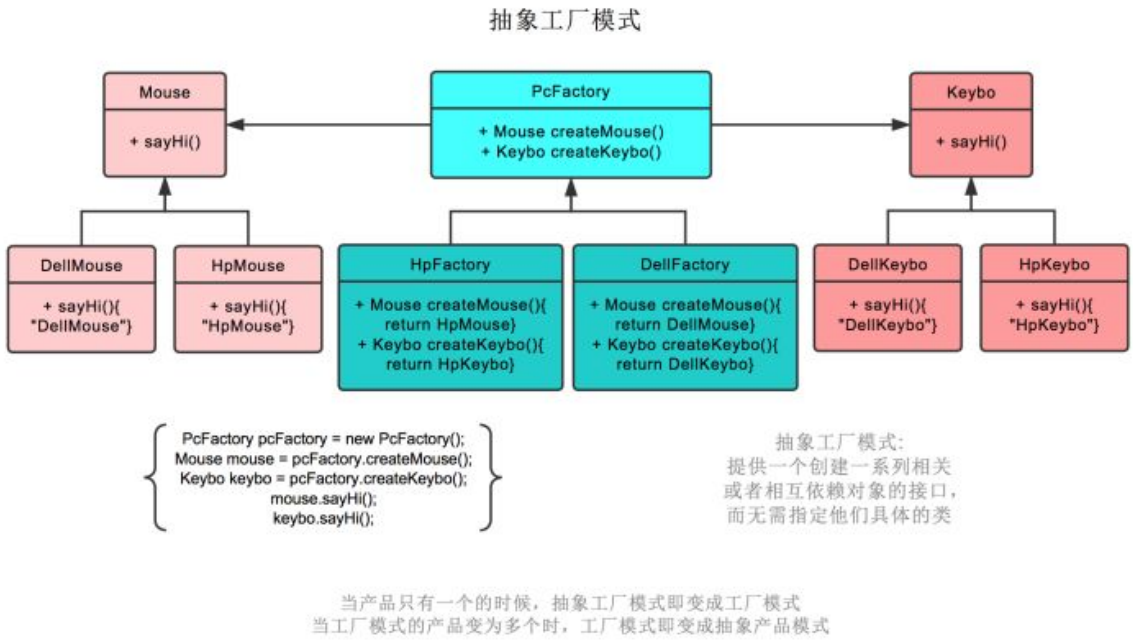
```
//父类工厂类
public class MouseFactory{
private static Map<String, Mouse> mouseMap = new HashMap<>();
    static {

        mouseMap.put("Dell", new DeliMouseFactory());

        mouseMap.put("Hp", new HpMouseFactory());
```

```
    }
    public static MouseFactory getMouse(String mouseType) {

        return mouseMap.get(mouseType);
    }
}
//子类工厂类 惠普鼠标
public class HpMouseFactory extends MouseFactory{
 public static Mouse createMouse(){
     return new HpMouse();
 }
}
    //戴尔鼠标
  public class DellMouseFactory extends MouseFactory{
 public static Mouse createMouse(){
     return new DeliMouse();
}
      //接口
      public interface Mouse{
    void sayHi();
      }

      public class DellMouse extends Mouse{
          public void sayHi(){
        System.out.println("Deli hello");
          }
      }

      public class HpMouse extends Mouse{
          public void sayHi(){
        System.out.println("Hp hello");
          }
      }
```

```
//策略者模式结合
public class Context {
    private Mouse mouse;

    public Context(Mouse mouse){
        this.mouse = mouse;
    }

    public String executeStrategy(){
        return mouse.sayHi();
    }
}
public class StrategyPattern {
    public static void main(String[] args) {
        //这一部是工厂来的
        MouseFactory mouseFactory = MouseFactory.getMouse("Hp");
        Mouse mouse = mouseFactory.getMouse("Hp");
        Context context = new Context(mouse);
        System.out.println(context.executeStrategy());
    }
}

```

```
23
```

三、抽象工厂模式



抽象工厂的实现更为复杂，父类的范围也越大。在抽象工厂模式中，假设我们需要增加一个工厂假设我们增加华硕工厂，则我们需要增加华硕工厂，和戴尔工厂一样，继承PC厂商。
之后创建华硕鼠标，继承鼠标类。创建华硕键盘，继承键盘类。
即可。

2. 观察者模式

观察者模式（Observer），又叫**发布-订阅模式（Publish/Subscribe）**，定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

（1）、 优点

- 观察者和被观察者是抽象耦合的
- 建立了一套触发机制

（4）. 缺点

- 如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间
- 如果观察者和观察目标间有循环依赖，可能导致系统崩溃
- 没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的

代码实现

> 在这里是运用了一个b站粉丝关注和告知粉丝视频发布的简单例子。

```
1  //粉丝就是观察者
2  public interface Observer {
3      //更新
4      public void update();
5      //关注up主
6      public void subscribeChannel(Channel ch);
7      //取消关注
```

```
    public void unSubscribe(Channel ch,Subscriber subscriber);
}

public class Subscriber implements Observer{
    private String name;
    private  List<Channel> channelList = new ArrayList<>();
    private Subscriber subscriber;
    //当up主更新时，会执行该方法
    @Override
    public void update(){
        System.out.println(name + ",你关注的up主发布了视频");
    }
    //将关注的人放在list中
    @Override
    public void subscribeChannel(Channel ch){
        channelList.add(ch);
    }

    public Subscriber(String name) {
        this.name = name;

    }
    //取消关注up主
    @Override
    public void unSubscribe(Channel ch, Subscriber subscriber){
        this.subscriber = subscriber;
        channelList.remove(ch);
        ch.unSubscribe(subscriber);
        System.out.println(name + "你已成功取消关注");
    }
}

```

```
public interface Subject {
    void subscribe(Subscriber sub);
    public void unSubscribe(Subscriber sub);
    public void notifySubscriber();
    public void upload(String title);

}

public class Channel implements Subject{
    List<Subscriber> subs = new ArrayList<>();
    private String title;

    @Override
    public void subscribe(Subscriber sub){
        //增加关注者
        subs.add(sub);
    }

    @Override
    public void unSubscribe(Subscriber sub){
        subs.remove(sub);
    }
    @Override
    public void notifySubscriber(){
```

```
        //通知到每个关注的人
        for(Subscriber sub : subs){
            sub.update();
        }
    }

    @Override
    public void upload(String title){
        this.title = title;
        notifySubscriber();
    }


}

```

```
public class Bilibili {
    public static void main(String[] args) {
        Channel javaLearning = new Channel();
        //两个关注者
        Subscriber s1 = new Subscriber("Ivy");
        Subscriber s2 = new Subscriber("Tom");

        javaLearning.subscribe(s1);
        javaLearning.subscribe(s2);
        //关注了javaLearning up主
        s1.subscribeChannel(javaLearning);
        s2.subscribeChannel(javaLearning);
        //up主上传了名为如何学java的视频
        javaLearning.upload("How to learn java");

        s1.unSubscribe(javaLearning,s1);

        javaLearning.upload("快速入门Javaweb");
    }
}
```

3. 适配器模式

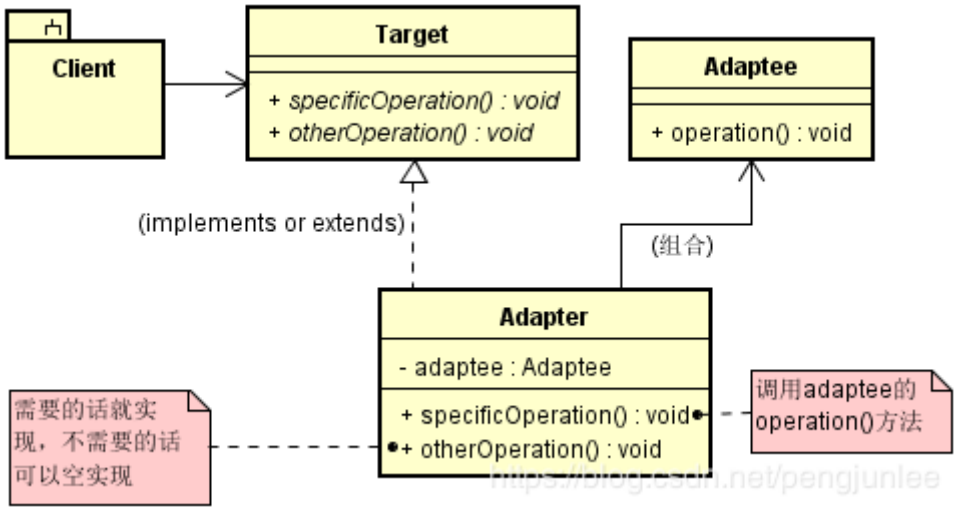
> 适配器模式的内容比较简单，举个例子，像在每个国家使用的电压和插头都是不一样的，在中国是一种，在国外又是一种，所以当我们出国要给一些设备充电时，就需要用到适配器，他是一个中介， 能把我们平常用到的接口转成该国家常使用的接口进行充电。所以基本原理就是这样。

在如下举了一个苹果充电线和安卓充电线的例子，要让苹果手机通过转换器用安卓数据线充上电。

Adaptee: 被适配的类。在下面的例子便是指**苹果手机线**

Adapter: 该类对适配者类和目标接口类进行适配，在对象适配器模式下通过组合方式实现，即：Adapter类继承Target类或者实现Target接口，并在其内部包含一个Adaptee对象的引用，通过对其内部的Adaptee对象的调用实现客户端所需要的接口。 在下面的例子便是`LightningToMicroUsbAdapter`

目标(Target)接口类：客户端所需要的接口。在下面的例子便是安卓手机线




```java
//苹果手机线的接口
interface LightningPhone {
    //充电
    void recharge();
    //使用充电线
    void useLightning();
}
//安卓手机线的接口
interface MicroUsbPhone {
    //充电
    void recharge();
    //使用充电线
    void useMicroUsb();
}
//苹果手机
class Iphone implements LightningPhone {
    private boolean connector;

    @Override
    public void useLightning() {
        connector = true;
        System.out.println("Lightning connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}
//安卓手机
class Android implements MicroUsbPhone {
    private boolean connector;

    @Override
    public void useMicroUsb() {
        connector = true;
        System.out.println("MicroUsb connected");
```

```
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect MicroUsb first");
        }
    }
}
//适配器，作为一个中介将适配者类和目标接口类进行适配，实现目标接口类
/* exposing the target interface while wrapping source object */
class LightningToMicroUsbAdapter implements MicroUsbPhone {
    //内部对Adaptee 对象的引用
    private final LightningPhone lightningPhone;

    public MicroUsbToLightningAdapter(LightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}
//客户端
public class AdapterDemo {
    static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
        phone.useMicroUsb();
        phone.recharge();
    }

    static void rechargeLightningPhone(LightningPhone phone) {
        phone.useLightning();
        phone.recharge();
    }

    public static void main(String[] args) {
        Android android = new Android();
        Iphone iPhone = new Iphone();

        System.out.println("Recharging android with MicroUsb");
        rechargeMicroUsbPhone(android);

        System.out.println("Recharging iPhone with Lightning");
        rechargeLightningPhone(iPhone);

        System.out.println("Recharging iPhone with MicroUsb");
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter (iPhone));
    }
```

```
}
```

作用：

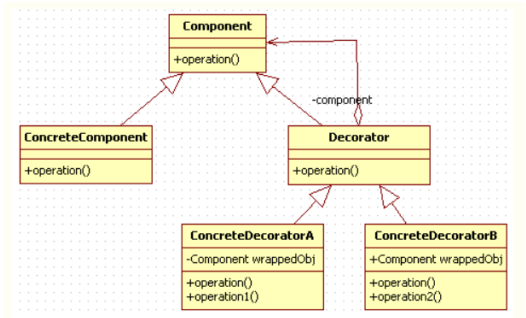
1. 将目标类和适配者类解耦，通过引入一个适配器类重用现有的适配者类，而无需修改原有代码。
2. 一个对象适配器可以把适配者类和它的子类都适配到目标接口。

4.装饰器模式

装饰器的核心便是进行包装，像是一个层层的包装。从一个最基础的物品，通过包装装饰可以获得一个升级版的物品。

举一个生活中贴近的例子，比如说去奶茶店买奶茶，你可以选择一个基础的商品珍珠奶茶，然后进行甜度的调整，配料的添加，加珍珠，芋圆等，都可以进行选择，然后价格也随之叠加，但本质上还是一个珍珠奶茶。

装饰器模式，顾名思义，就是对已经存在的某些类进行装饰，以此来扩展一些功能。其结构图如下：



- Component为统一接口，也是装饰类和被装饰类的基本类型。
- ConcreteComponent为具体实现类，也是被装饰类，他本身是个具有一些功能的完整的类。
- Decorator是装饰类，实现了Component接口的同时还在内部维护了一个ConcreteComponent的实例，并可以通过构造函数初始化。而Decorator本身，通常采用默认实现，他的存在仅仅是一个声明，我要生产出一些用于装饰的子类了。而其子类才是赋有具体装饰效果的装饰产品类。
- ConcreteDecorator是具体的装饰产品类，每一种装饰产品都具有特定的装饰效果。可以通过构造器声明装饰哪种类型的ConcreteComponent，从而对其进行装饰。


```java
//Component 接口
public interface MilkTea{
    //价格
    public double cost();
    //配料
    public String getIngredinets();
}

public class SimpleMilkTea implements MilkTea{
    @Override
    public double cost(){
    return 10;
    }
    //拿一点点的波霸举例
    @Override
    public String getIngredients(){
    return "BoBa"
    }
}

//装饰器抽象类 使用 了MilkTea的接口
public abstract class MilkTeaDecorator implements MilkTea{
    private final MilkTea decoratedMilkTea;
```

```
    public MilkTeaDecorator(MilkTea m){
        this.decoratedMilkTea = m;
    }
    @Override
    public double cost(){
        return decoratedMilkTea.cost();
    }

    @Override
    public String getIngredients(){
        return decoratedMilkTea.getIngredients();
    }

}

//继承了装饰器
class WithMilk extends MilkTeaDecorator{
    public WithMilk(Milktea m){
        super(m);
    }
    //加了牛奶 + 1块钱
    @Override
    public double cost(){
        return super.cost()+1;
    }
    //配料加上了牛奶
    @Override
    public String getIngredient(){
        return super.getIngredients + ",Milk";
    }
}


public class Main{
    public static void main(String[] args){
        MilkTea m = new SimpleMilkTea();

        m = new WithMilk(c);

    }
}
```

总而言之：

```
在不必改变原类文件和使用继承的情况下，动态地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。
装饰对象接受所有来自客户端的请求。它把这些请求转发给真实的对象。装饰对象可以在转发这些请求以前或以后增加一些附加功能。
这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。在面向对象的设计中，通常是通过继承来实现对给定类的功能扩展。
```

----------2020.05.24----------

5.外观模式

```
public class Facade {

    //被委托的对象
    SubSystemA a;
    SubSystemB b;
    SubSystemC c;
    SubSystemD d;

    public Facade() {
        a = new SubSystemA();
        b = new SubSystemB();
        c = new SubSystemC();
        d = new SubSystemD();
    }

    //提供给外部访问的方法
    public void methodA() {
        this.a.dosomethingA();
    }

    public void methodB() {
        this.b.dosomethingB();
    }

    public void methodC() {
        this.c.dosomethingC();
    }

    public void methodD() {
        this.d.dosomethingD();
    }

}
```

```
public class SubSystemA {

    public void dosomethingA() {
        System.out.println("子系统方法A");
    }

}
```

```
public class Client {

    public static void main(String[] args) {
        Facade facade = new Facade();

        facade.methodA();
        facade.methodB();
    }

}
```

优点：

- 减少了系统的相互依赖
- 提高了灵活性。不管系统内部如何变化，只要不影响到外观对象，任你自由活动
- 提高了安全性。想让你访问子系统的哪些业务就开通哪些逻辑，不在外观上开通的方法，你就访问不到

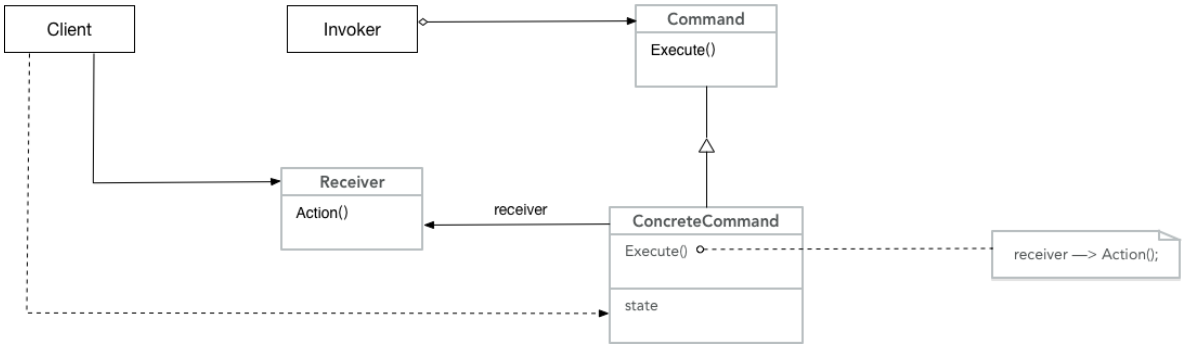
6.命令模式

简单的说，命令模式可将“动作的请求者”从“动作的执行者”对象中解耦。

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化(即，可以用不同的命令对象，去参数化配置客户的请求)；对请求排队或记录请求日志，以及支持可撤销的操作。

这一模式的关键是一个抽象的Command类，它定义了一个执行操作的接口。其最简单的形式是一个抽象的Execute操作。具体的Command子类将接收者作为其一个实例变量，并实现Execute操作，指定接收者采取的动作。而接收者有执行该请求所需的具体信息。
接收者：真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。

> 这的确是我感受到的，因为也不需要你去干什么，就是一个遥控器罢了，你可以给任意的执行



Command：
定义命令的接口，声明执行的方法。

ConcreteCommand：
命令接口实现对象，是“虚”的实现;通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。

Receiver：
接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。

Invoker：
要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。

Client:
创建具体的命令对象，并且设置命令对象的接收者。注意这个不是我们常规意义上的客户端，而是在组装命令对象和接收者，或许，把这个Client称为装配者会更好理解，因为真正使用命令的客户端是从Invoker来触发执行。

> 这里是模拟一个遥控器的操作过程

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

```
class LightOnCommand implements Command{
    Light light;
    LightOnCommand(Light light) {
        this.light = light;
    }
    public void execute(){
        this.light.on();
    }
    public void unexcute(){
        this.light.off();
}
}


```

```
class light{
    ICommand on;
    ICommand off;
    ICommand up;
    ICommand down;

    light(ICommand on, ICommand off, ICommand up, ICommand down){
        this.on = on;
        this.off = off;
        this.up = up;
        this.down = down;
    }

    public void clickOn(){
        this.on.execute();
    }
    public void clickOff(){
        this.on.unexecute();
    }
}
```

用途

可撤销操作的意思就是：放弃该操作，回到未执行该操作前的状态。

有两种基本的思路来实现可撤销的操作：

① 一种是补偿式，又称反操作式

比如被撤销的操作是加的功能， 那撤消的实现就变成减的功能;同理被撤销的操作是打开的功能，那么撤销的实现就变成关闭的功能。

② 另外一种方式是存储恢复式

意思就是把操作前的状态记录下来，然后要撤销操作的时候就直接恢复回去就可以了。

命令模式的关键之处就是把请求封装成为对象，也就是命令对象(一个接收者和一组动作)，然后将它传来传去，就像是一般的对象一样。现在，即使在命令对象被创建许久之后，运算依然可以被调用。事实上，它甚至可以在不同的线程中被调用。我们可以利用这样的特性衍生一些应用，例如：线程池、工作队列、日志请求等。

1. 队列请求

   想象有一个工作队列：你在某一端添加命令，然后另一端则是线程。线程进行下面的动作：从队列中取出一个命令，调用它的execute()方法，等待这个调用完成，然后将此命令对象丢弃，再取出下一个命令……

   请注意，工作队列和命令对象之间是完全解耦的。此刻线程可能在进行财务运算，下一刻却在读取网络数据。工作队列对象不在乎到底做些什么，它们只知道取出命令对象，然后调用其execute()方法。类似地，它们只要实现命令模式的对象，就可以放入队列里，当线程可用时，就调用此对象的execute()方法。
2. 日志请求

   某些应用需要我们将所有的动作都记录在日志中，并能在系统死机之后，重新调用这些动作恢复到之前的状态。