Java 设计模式

1、模板模式

> 在模板模式（Template Pattern）中，**一个抽象类公开定义了执行它的行为流程。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。**这种类型的设计模式属于行为型模式。

1.1、问题背景

有个记者去南极采访一群企鹅，他问第一只企鹅：“你每天都干什么？”

企鹅说：“吃饭，睡觉，打豆豆！”

接着又问第2只企鹅，那只企鹅还是说：“吃饭，睡觉，打豆豆！”

记者带着困惑问其他的企鹅，答案都一样，就这样一直问了99只企鹅。

当走到第100只小企鹅旁边时，记者走过去问它：每天都做些什么啊？

那只小企鹅回答："吃饭，睡觉，打豆豆"

1.2、常规方式

“吃饭，睡觉，打豆豆”其实都是独立的行为，为了不相互影响，我们可以**通过函数简单进行封装**：

```
public class Test {
    public static void main(String[] args) {
        System.out.println("littlePenguin:");
        littlePenguin penguin_1 = new
littlePenguin();
        penguin_1.eating();
        penguin_1.sleeping();
        penguin_1.beating();
        // 下同，省略...
```

```
        middlePenguin penguin_2 = new
middlePenguin();
        penguin_2.eating();
        penguin_2.sleeping();
        penguin_2.beating();
    }

}

class littlePenguin {
    public void eating() {
        System.out.println("吃饭");
    }
    public void sleeping() {
        System.out.println("睡觉");
    }
    public void beating() {
        System.out.println("用小翅膀打豆豆");
    }
}
class middlePenguin {
    public void eating() {
        System.out.println("吃饭");
    }
    public void sleeping() {
        System.out.println("睡觉");
    }
    public void beating() {
        System.out.println("用圆圆的肚子打豆豆");
    }
}
```

1.3、优雅设计

这3只企鹅，由于每天吃的都一样，睡觉也都是站着睡，但是打豆豆的方式却不同，**所以我们可以将“吃饭，睡觉，打豆豆”抽象出来，因为“吃饭，睡觉”都一样，所以我们可以直接实现出来，但是他们“打豆豆”的方式不同，所以封装成抽象方法，需要每个企鹅单独去实现“打豆豆”的方式。最后再新增一个方法everyDay()，固定每天的执行流程**：

> 封装抽象类

```
public abstract class penguin {

    public void eating() {
        System.out.println("吃饭");
    }
    public void sleeping() {
        System.out.println("睡觉");
    }

    public abstract void beating();

    public void everyDay() {
        this.eating();
        this.sleeping();
        this.beating();
    }
}
```

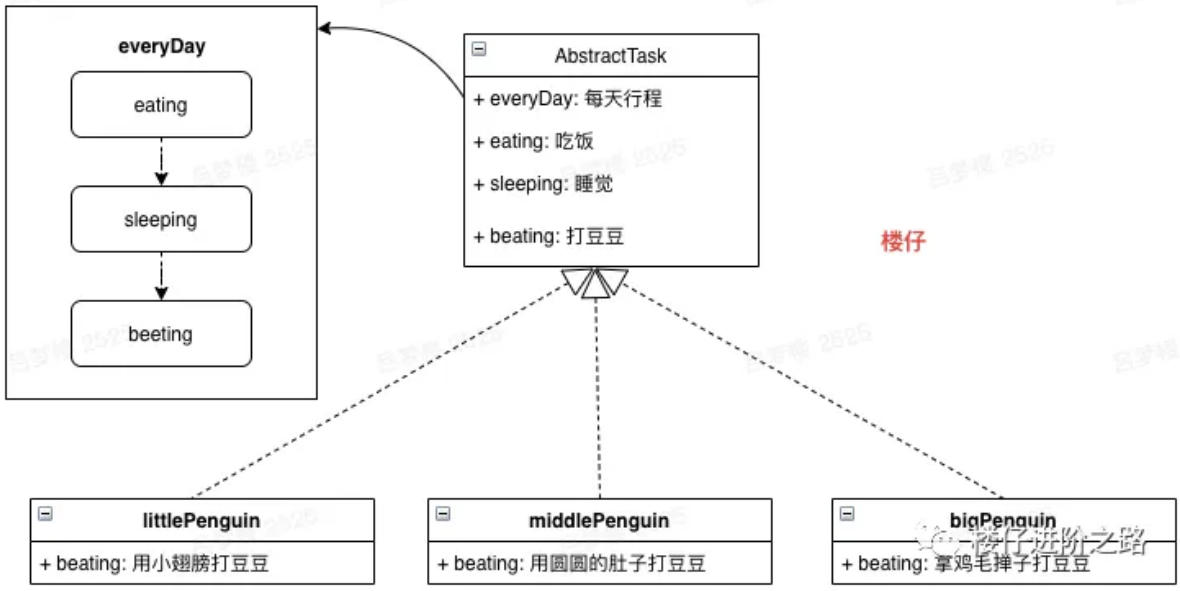
> 实体类继承抽象类并实现自己的方法

```

class littlePenguin extends penguin {
    @Override
    public void beating() {
        System.out.println("用小翅膀打豆豆");
    }
```

```
}
class middlePenguin extends penguin {
    @Override
    public void beating() {
        System.out.println("用圆圆的肚子打豆豆");
    }
}

class bigPenguin extends penguin {
    @Override
    public void beating() {
        System.out.println("拿鸡毛掸子打豆豆");
    }
}
```

> 方法测试

```
public class Test {

    public static void main(String[] args) {
        System.out.println("littlePenguin:");
        littlePenguin penguin1 = new
littlePenguin();
        penguin1.everyDay();
        System.out.println("middlePenguin:");
        middlePenguin penguin2 = new
middlePenguin();
        penguin2.everyDay();
        System.out.println("bigPenguin:");
        bigPenguin penguin3 = new bigPenguin();
        penguin3.everyDay();
    }
}
```



2、策略模式

> 在策略模式（Strategy Pattern）中，**定义了一系列的算法，并将每一个算法封装起来，使它们可以互相替换**，且算法的改变不会影响到使用算法的客户端。通过使用策略模式，可以根据上下文动态地更换策略，使得算法独立于使用它们的客户端。

2.1、问题背景

假设公司需要做一款武侠游戏，我们就是负责游戏的角色模块，需求是这样的：每个角色对应一个名字，每类角色对应一种样子，每个角色拥有一个逃跑、攻击、防御的技能。

2.2、常规方式

> 定义抽象基类

```java
public abstract class Role {
    protected String name;

    protected abstract void display();

    protected abstract void run();

    protected abstract void attack();

    protected abstract void defend();

}
```

> 定义实体类继承基类

```java
public class RoleA extends Role {
    public RoleA(String name) {
        this.name = name;
    }

    @Override
    protected void display() {
        System.out.println("样子1");
    }

    @Override
    protected void run() {
        System.out.println("金蝉脱壳");
    }

    @Override
    protected void attack() {
        System.out.println("降龙十八掌");
    }

    @Override
```

```
    protected void defend() {
        System.out.println("铁头功");
    }

}

public class RoleB extends Role {
    public RoleB(String name) {
        this.name = name;
    }

    @Override
    protected void display() {
        System.out.println("样子2");
    }

    @Override
    protected void run() {
        System.out.println("金蝉脱壳");//从RoleA中拷贝
    }

    @Override
    protected void attack() {
        System.out.println("降龙十八掌");//从RoleA中拷
贝
    }

    @Override
    protected void defend() {
        System.out.println("铁布衫");
    }

}
```

测试类

```
public class Test {

    public static void main(String[] args) {
        RoleA roleA = new RoleA("A");
        RoleB roleB = new RoleB("B");
        roleA.display();
        roleB.display();
    }
}
```

2.3、优雅设计

对于每个角色的**display，attack，defend，run都是有可能变化的，于是我们必须把这写独立出来**。再根据另一个设计原则：针对接口（超类型）编程，而不是针对实现编程。

> 设计具体的接口

```
public interface IAttackBehavior {

    void attack();
}

public interface IDefendBehavior {

    void defend();
}

public interface IDisplayBehavior {

    void display();
}
```

> 根据技能设计具体的封装类

```

```

```
public class AttackJY implements IAttackBehavior {

    @Override
    public void attack() {
        System.out.println("九阳神功！ ");
    }

}

public class DefendTBS implements IDefendBehavior {

    @Override
    public void defend() {
        System.out.println("铁布衫");
    }

}

public class DisplayJCTQ implements IDisplayBehavior
{

    @Override
    public void display() {
        System.out.println("伪装术");
    }

}
```

> 实体角色基类

```
public class Role {

    protected String name;

    protected IDefendBehavior defendBehavior;
```

```
    protected IDisplayBehavior displayBehavior;
    protected IAttackBehavior attackBehavior;

    public Role setName(String name) {
        this.name = name;
        return this;
    }

    public Role setDefendBehavior(IDefendBehavior
defendBehavior) {
        this.defendBehavior = defendBehavior;
        return this;
    }

    public Role setDisplayBehavior(IDisplayBehavior
displayBehavior) {
        this.displayBehavior = displayBehavior;
        return this;
    }

    public Role setAttackBehavior(IAttackBehavior
attackBehavior) {
        this.attackBehavior = attackBehavior;
        return this;
    }

    protected void display() {
        displayBehavior.display();
    }

    protected void attack() {
        attackBehavior.attack();
    }

    protected void defend() {
        defendBehavior.defend();
    }
```

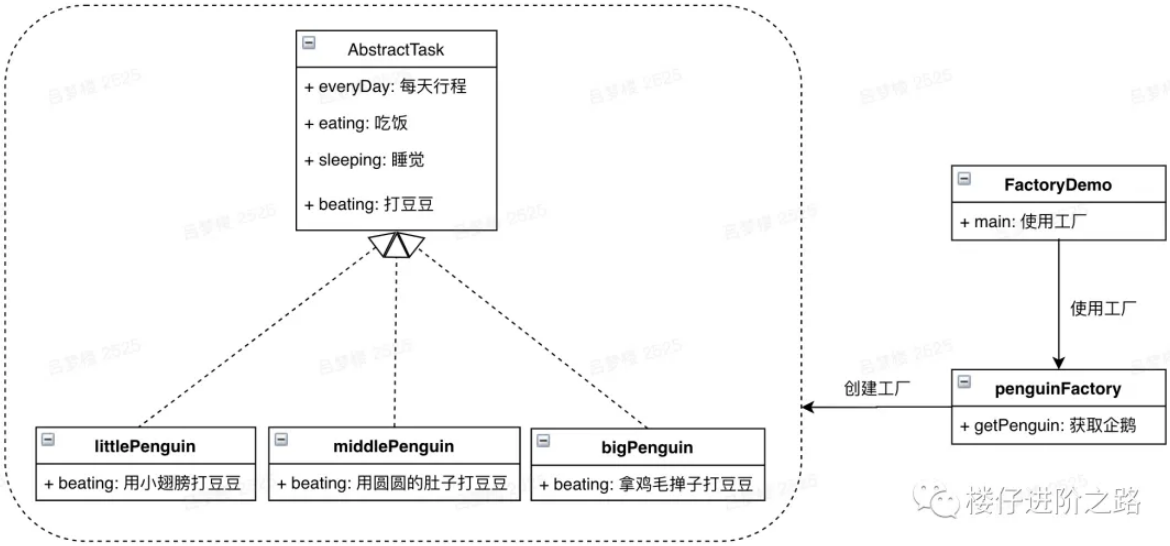
```
}
```

> 测试方法

```
public class Test {

    public static void main(String[] args) {

        Role roleA = new Role();
        roleA.setName("A");

        roleA.setAttackBehavior(new AttackJY())
                .setDefendBehavior(new DefendTBS())
                .setDisplayBehavior(new
DisplayJCTQ());
        System.out.println(roleA.name + ":");
        roleA.attack();
        roleA.defend();
        roleA.display();
    }
}
```

3、工厂模式

> 工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种**创建对象**的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过**使用一个共同的接口来指向新创建的对象。**



其实设计模式一般不会单一使用，通常会和其它模式结合起来使用，这里我们就将**模板模式和工厂模式**结合起来。

因为工厂模式，通常会给这些新创建的对象制定一个公共的接口，我们可以通过抽象类定义：

```
public abstract class penguin {
    public void eating() {
        System.out.println("吃饭");
    }

    public void sleeping() {
        System.out.println("睡觉");
    }

    public abstract void beating();

    public void everyDay() {
        this.eating();
        this.sleeping();
        this.beating();
    }
}
```

因为我们是结合了模板模式，所以这个抽象类中，可以看到模板模式的影子，如果你只关注抽象的接口，比如beating，那么这个就是一个抽象方法，也可以理解为下游需要实现的方法，其它的接口其实可以忽略。再看看每个企鹅具体的实现：

```
public class littlePenguin extends penguin {
    @Override
    public void beating() {
        System.out.println("用小翅膀打豆豆");
    }
}
public class middlePenguin extends penguin {
    @Override
    public void beating() {
        System.out.println("用圆圆的肚子打豆豆");
    }
}
public class bigPenguin extends penguin {
    @Override
    public void beating() {
        System.out.println("拿鸡毛掸子打豆豆");
    }
}
```

在之前模板模式的基础上，构建一个工厂，专门用来拿企鹅。这里使用静态工厂

```
public class PenguinFactory {
    private static final Map<String, penguin> map =
new HashMap<>();
    static {
        map.put("littlePenguin", new
littlePenguin());
        map.put("middlePenguin", new
middlePenguin());
        map.put("bigPenguin", new bigPenguin());
    }
    // 获取企鹅
    public static penguin getPenguin(String name) {
        return map.get(name);
    }
}
```

测试方法

```
public static void main(String[] args) {
    penguin penguin_1 =
PenguinFactory.getPenguin("littlePenguin");
    penguin_1.everyDay();
    penguin penguin_2 =
PenguinFactory.getPenguin("middlePenguin");
    penguin_2.everyDay();
    penguin penguin_3 =
PenguinFactory.getPenguin("bigPenguin");
    penguin_3.everyDay();
}
```

> 实例介绍

首先会**对每个方法中的内容通过模板模式进行抽象**（因为本章主要讲工厂模式，模板模式的代码，我就不贴了），然后**通过工厂模式获取不同的对象**，直接看重构后的代码（目前还是DEMO版）：

```
```

```
public class TaskFactory {

    @Autowired
    public static List<AbstractTask> taskList;

    private static final Map<String, AbstractTask>
map = new HashMap<>();

    static {
        // 存放任务映射关系
        map.put(AbstractTask.OPERATOR_TYPE_FROZEN,
new BatchFrozenTask());
        map.put(AbstractTask.OPERATOR_TYPE_REJECT,
new BatchRejectTask());
        map.put(AbstractTask.OPERATOR_TYPE_CANCEL,
new BatchCancelTask());
    }

    public static void main(String[] args) {
        String operatorType =
AbstractTask.OPERATOR_TYPE_REJECT;
        AbstractTask task =
TaskFactory.map.get(operatorType);
        ParamWrapper<CancelParams> params = new
ParamWrapper<CancelParams>();
        params.rowKey = 11111111;
        params.data = new CancelParams();
        OcApiServerResponse res =
 task.execute(params);
        System.out.println(res.toString());
        return;
    }
}
```

4、建造者模式

> 将一个复杂对象的**构建(构造方法)**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。

4.1、常规方式

> 实体类创建，带有多个**构造方法**

```
public class penguin {
    private String  name;
    private Integer age;
    private String  sex;
    private Integer heignt;

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(Integer age) {
        this.age = age;
    };
    public void setSex(String sex) {
        this.sex = sex;
    }
    public void setHeignt(Integer heignt) {
        this.heignt = heignt;
    };
    public void print() {
        String str = "name:" + name;
        str += (age == null) ? "" : ",age:" + age;
        str += (sex == null) ? "" : ",sex:" + sex;
        str += (heignt == null) ? "" : ",age:" +
heignt;
        System.out.println(str);
    }
```

```
     public penguin(String name) {
        this.name = name;
    }
    public penguin(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    public penguin(String name, Integer age, String
sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }
    public penguin(String name, Integer age, String
sex, Integer heignt) {
        this.name = name;
        this.age = age;
        this.sex = sex;
        this.heignt = heignt;
    }
}
```

> 测试方法

```
public class Test {

    public static void main(String[] args) {
        penguin penguin1 = new penguin("楼仔");
        penguin penguin2 = new penguin("楼仔", 18);
        penguin penguin3 = new penguin("楼仔", 18,
"男");
        penguin penguin4 = new penguin("楼仔", 18,
"男", 180);
        penguin1.print();
        penguin2.print();
        penguin3.print();
        penguin4.print();
    }
}
```

这种方式比较常规，但是如果我只想初始化name和hight，你可能还需要再新增新的构造器，不过有的同学可能会说“我才不用构造器初始化对象，我可以用类提供的setxxx()来设置对应的属性值”，但是这种做法不会觉得很冗余么，每个属性都用setxxx()设置一下，如果有十几个属性值，你是不是要用setxxx()全部设置一遍呢?

4.2、优雅方式

很多场景都会使用，当你需要初始化一个对象，但是对象里面有一堆成员变量，比如有10个左右，你如果给每个成员调用set方法去赋值，代码可读性就太差了，**这时可以采用builder模式，让这些成员的赋值通过链式的方式去set值，然后通过builder去生成一个完整的对象。**

> 给penguin增加一个**静态内部类penguinBuilder类，并修改penguin类的构造函数**

```
public class penguin {

    private String  name;
    private Integer age;
```

```
    private String  sex;
    private Integer height;

    public void print() {
        String str = "name:" + name;
        str += (age == null) ? "" : ",age:" + age;
        str += (sex == null) ? "" : ",sex:" + sex;
        str += (height == null) ? "" : ",age:" +
height;
        System.out.println(str);
    }
    public penguin(penguinBuilder builder) {
        this.age = builder.age;
        this.name = builder.name;
        this.sex = builder.sex;
        this.height = builder.height;
    }

    public static class penguinBuilder {
        private String  name;
        private Integer age;
        private String  sex;
        private Integer height;

        public penguinBuilder setName(String name) {
            this.name = name;
            return this;
        }
        public penguinBuilder setAge(Integer age) {
            this.age = age;
            return this;
        }
        public penguinBuilder setSex(String sex) {
            this.sex = sex;
            return this;
        }
```

```
        public penguinBuilder setHeight(Integer
height) {
            this.height = height;
            return this;
        }
        public penguin build() {
            return new penguin(this);
        }
    }
}
```

> 测试方法

penguinBuilder类中的成员函数返回penguinBuilder对象自身，让它**支持链式调用**，使代码可读性大大增强。

```
public static void main(String[] args) {
    penguin penguin1 =
        new penguin.penguinBuilder().setName("楼仔").
        setSex("男").
        setHeight(170).
        setAge(18).
        build();
    penguin penguin2 =
        new penguin.penguinBuilder().setName("楼仔").
        setAge(18).
        build();
    penguin1.print();
    penguin2.print();
}
```

> 建造者模式要点

1. **定义一个静态内部类Builder，内部的成员变量和外部类一样；**
2. Builder类通过一系列的方法用于**成员变量的赋值**，并**返回当前对象本身（this）**；

3. Builder类**提供一个外部类的创建方法（build、create……）**，该方法内部调用了外部类的一个私有构造函数，**入参就是内部类Builder**；
4. **外部类提供一个私有构造函数供内部类调用，在该构造函数中完成成员变量的赋值，取值为Builder对象中对应的成变量的值。**

5、组合模式

如果你的代码需要处理成“总-分”关系，或者说是树形结构关系，最后通过一次调用完成所有对象的操作行为，那么就可以选择组合模式。

组合模式**依据树形结构来组合对象，用来表示部分以及整体层次。**这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

组合模式其实比较简单，层次很分明，主要包括一个抽象接口、组合对象节点和叶子节点：

- Component抽象组件：**为组合中的所有对象提供一个接口**，不管是叶子对象还是组合对象。
- Component组合节点对象：**实现了接口的所有操作，并且持有子节点对象。**
- Leaf叶子节点对象：叶子节点没有任何子节点，实现了接口中的某些操作。

> Component抽象组件

```
public abstract class penguin {

    protected String name;

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

    public abstract void beating();

    public void add(penguin p) {
```

```
        throw new UnsupportedOperationException();
    }
    public void remove(penguin p) {
        throw new UnsupportedOperationException();
    }
    public penguin getChild(int i) {
        throw new UnsupportedOperationException();
    }
    public List<penguin> getChildren() {
        throw new UnsupportedOperationException();
    }
}
```

这个抽象类其实就是定义了一个公共的行为beating，然后增加了一些方法，**这些方法在“Component组合节点对象”都需要实现**，但是在“Leaf叶子节点对象”可以不用实现。

> Component组合节点对象

```
public class batchPenguin extends penguin {
    private List<penguin> m_penguins = new
ArrayList<>();

    public batchPenguin(String name) {
        super(name);
    }
    @Override
    public void beating() {
        System.out.println(this.name + "打豆豆");
        for (penguin p : m_penguins) {
            p.beating();
        }
    }
    @Override
    public void add(penguin p) {
        m_penguins.add(p);
    }
```

```
    @Override
    public void remove(penguin p) {
        m_penguins.remove(p);
    }
    @Override
    public penguin getChild(int i) {
        return m_penguins.get(i);
    }
    @Override
    public List<penguin> getChildren() {
        return m_penguins;
    }
}
```

测试方法

```
public static void main(String[] args) {
    batchPenguin grandFatherPenguin = new
batchPenguin("grandFatherPenguin");
    batchPenguin fatherPenguin = new
batchPenguin("fatherPenguin");
    batchPenguin motherPenguin = new
batchPenguin("motherPenguin");
    batchPenguin childPenguin1 = new
batchPenguin("childPenguin1");
    batchPenguin childPenguin2 = new
batchPenguin("childPenguin2");
    batchPenguin childPenguin3 = new
batchPenguin("childPenguin3");
    batchPenguin childPenguin4 = new
batchPenguin("childPenguin4");
    fatherPenguin.add(childPenguin1);
    fatherPenguin.add(childPenguin2);
    motherPenguin.add(childPenguin3);
    motherPenguin.add(childPenguin4);
    grandFatherPenguin.add(fatherPenguin);
    grandFatherPenguin.add(motherPenguin);
```

```
    grandFatherPenguin.beating();
}

grandFatherPenguin打豆豆
fatherPenguin打豆豆
childPenguin1打豆豆
childPenguin2打豆豆
motherPenguin打豆豆
childPenguin3打豆豆
childPenguin4打豆豆
```

> Leaf叶子节点对象

```
public class leaf extends penguin{
    public leaf(String name) {
        super(name);
    }
    @Override
    public void beating() {
        System.out.println(name + "打豆豆");
    }
}
```

```
leaf leaf1 = new leaf("leaf1");
leaf leaf2 = new leaf("leaf2");
leaf leaf3 = new leaf("leaf3");
leaf leaf4 = new leaf("leaf4");
childPenguin1.add(leaf1);
childPenguin2.add(leaf2);
childPenguin3.add(leaf3);
childPenguin4.add(leaf4);
```

> 实际场景

下面是小米商城购物车界面，可以看到里面有很多功能模块，你可以直接采用堆砌的方式实现每一个模块，然后依次调用每个模块具体的执行逻辑：

JD 京东自营

小米路由器AC2100 双频路由器 2100M无线家用...

颜色：【新品热销双核全千兆】AC2100

¥199.00

支持7天无理由退货

1

配送 可选京尊达 京东快递

4月1日 [周三] 09:00-15:00

自提 怡景园南门京东快递柜 200m 距离最近

退换无忧 ¥0.80 可享1次免费上门取件

发票 电子(商品明细-个人)

优惠券 无可用

京豆 共90，满1000可用

礼品卡(京东卡/E卡) 1张可用

商品金额 ¥199.00

运费 (总重:0.980kg) + ¥0.00

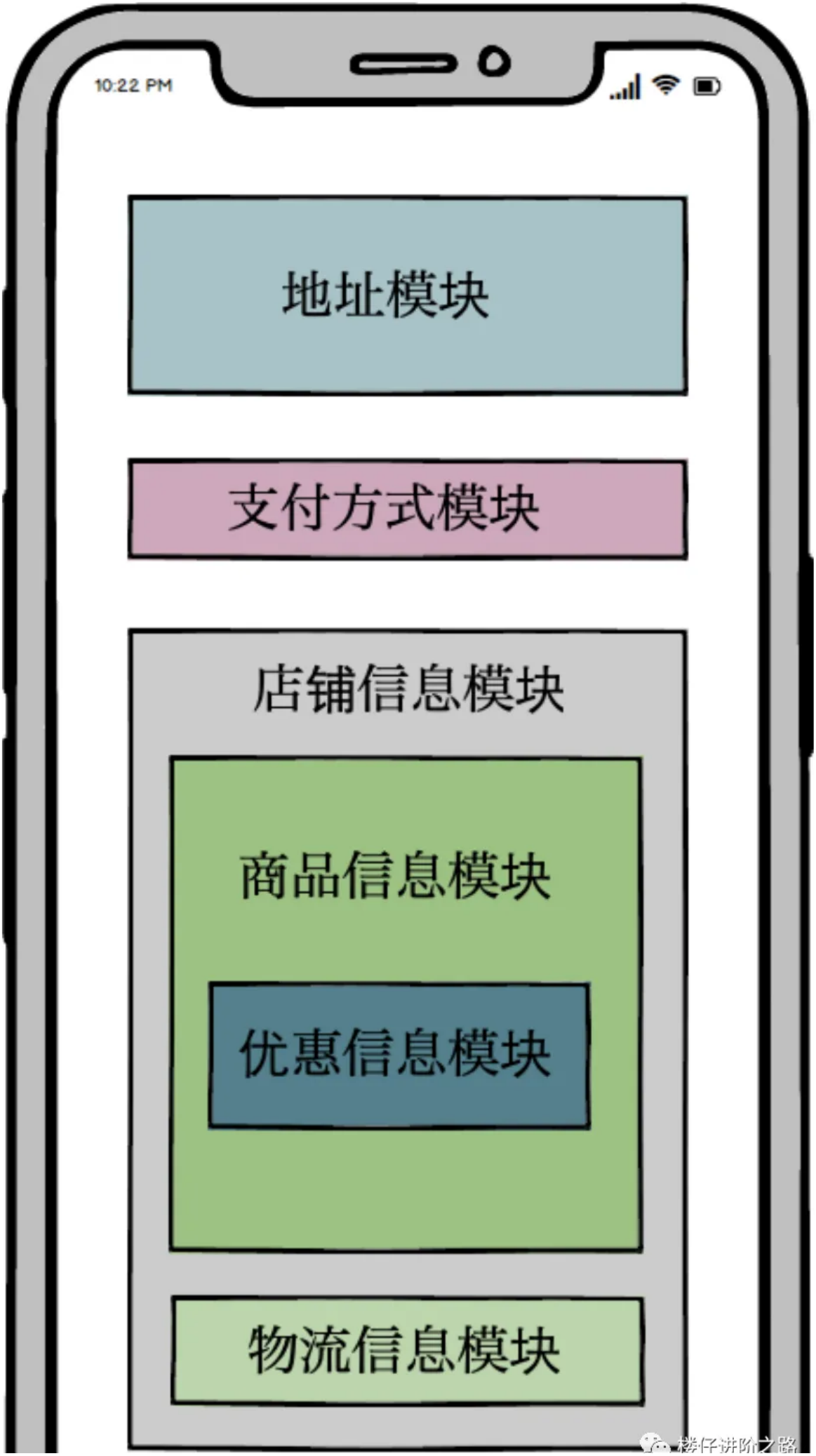
北京朝阳区四环到五环之间演示地址

¥199.00

提交订单

楼仔进阶之路

当然，我们也可以用组合模式，将购物车抽象成下面的树状结构（还有很多模块，仅列举一部分）：


10:22 PM
地址模块
支付方式模块
店铺信息模块
商品信息模块
优惠信息模块
物流信息模块

代码我就补贴了，核心实现就是**通过组合模式将购物车的对象按照“总-分”关系组合在一起**，最后**执行购物车的Process()方法，就可以调用所有对象的Process()操作**，从而完成每个模块对自身业务的逻辑处理。

6、单例模式

> 确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

6.1、懒汉模式

在类加载时，不创建实例，因此类加载速度快，但运行时获取对象的速度慢，代码如下：

```
public class penguin_懒汉 {
    // 可见性
    private static volatile penguin_懒汉 m_penguin =
null;
    // 避免通过new初始化对象
    private void penguin_懒汉() {}
    public void beating() {
        System.out.println("打豆豆");
    };
    public static penguin_懒汉 getInstance() {
        // 减少锁的竞争
        if (null == m_penguin) {
            // 加锁，初始化
            synchronized(penguin_懒汉.class) {
                // 防止重复初始化
                if (null == m_penguin) {
                    m_penguin = new penguin_懒汉();
                }
            }
        }
```

```
        return m_penguin;
    }
}
```

懒汉模式实现要点

- 单例使用volatile修饰;
- 单例实例化时，要用synchronized 进行同步处理;
- 双重null判断。

6.2、饿汉模式

在**类加载时就完成了初始化**，所以类加载较慢，但获取对象的速度快，代码如下:

```
public class penguin_饿汉 {

    private static penguin_饿汉 m_penguin = new
penguin_饿汉();
    private void penguin_饿汉() {}
    public static penguin_饿汉 getInstance() {
        return m_penguin;
    }
}
```

> 适用场景

单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，**如多个模块使用同一个数据源连接对象**等等。如:

- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
- 有状态的工具类对象。
- 频繁访问数据库或文件的对象。

7、代理模式

> 为其他对象提供一种代理以控制这个对象的访问。

涉及角色及说明：

> Subject（抽象主题类）：接口或者抽象类，**声明**真实主题与代理的共同**接口方法**。
>
> RealSubject（真实主题类）：也叫做被代理类或被委托类，**定义了代理所表示的真实对象，负责具体业务逻辑的执行。**
>
> Proxy（代理类）：也叫委托类，**持有对真实主题类的引用**，在其所实现的接口方法中调用真实主题类中相应的接口方法执行。
>
> Client（客户端类）：使用代理模式的地方，客户端可以通过代理类间接的调用真实主题类的方法。

7.1、静态代理

> 创建抽象主题

```
public interface penguin {
    public void beating();
}
```

> 创建真实主题

```
public class littlePenguin implements penguin {

    @Override
    public void beating() {
        System.out.println("打豆豆");
    }
}
```

> 创建静态代理类

```
public class penguinProxy {

    private penguin m_penguin;

    public penguinProxy(penguin p) {
        this.m_penguin = p;
    }

    public void beating() {
        System.out.println("打豆豆前");
        m_penguin.beating();
        System.out.println("打豆豆后");
    }
}
```

> 客户端使用

客户端传入不同的接口实现类

```
public static void main(String args[]) {
    penguin penguin1 = new littlePenguin();
    penguinProxy proxy = new penguinProxy(penguin1);
    proxy.beating();
}
```

7.2、动态代理 todo