Java 设计模式之模板方法模式（十三）

一、前言

上篇[《Java 设计模式之代理模式（十二）》] 为 Java 设计模式中结构型模式的最后一章，今天开始介绍 Java 设计模式中的行为型模式的第一种模式--模板方法模式。

二、简单介绍

2.1 定义

模板方法（ Template Method）模式是行为模式之一，它把具有特定步骤算法中的某些必要的处理委让给抽象方法，通过子类继承对抽象方法的不同实现改变整个算法的行为。

2.2 应用场景

1. 具有统一的操作步骤或操作过程。
2. 具有不同的操作细节。

三、实现方式

我们以在银行办理业务为例，张三和李四去银行办理业务。张三办理存钱业务，李四办理取钱业务。

实体类：

```
public class People {

    private String name;

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

    public void queueUp() {
        System.out.println(this.name + "排队取号等候");
    }

    public void service(String type) {
        System.out.println(this.name + "办理" + type + "业务");
    }

    public void evaluate() {
        System.out.println(this.name + "反馈评分");
    }
}
```

客户端：

```
public class Client {

    public static void main(String[] args) {
        People p1 = new People("张三");
        p1.queueUp();
        p1.service("存钱");
        p1.evaluate();

        System.out.println("==============");

        People p2 = new People("李四");
        p2.queueUp();
        p2.service("取钱");
        p2.evaluate();
    }
}
```

打印结果：

```
张三排队取号等候
张三办理存钱业务
张三反馈评分
==============
李四排队取号等候
李四办理取钱业务
李四反馈评分
```

在银行办理业务需要 3 个步骤，即 “排队取号” -> “办理业务” -> “反馈评分”，我们发现 “排队取号” 和 “反馈评分” 的操作（逻辑代码）基本是一致的，只有客户办理的业务是有所差异的。但是，在客户端中，每个人办理业务需要调用 3 次方法，不够简便。

通过分析，现在的案例符合运用模板方法模式的两个场景：相同步骤（都需要调用 queueUp、service 和 evaluate）和不同操作细节（service）。

正如上文描述的，我们可以讲公共的代码（queueUp 和 evaluate）放到父类中，同时，由于父类中有部分不能复用的方法（service ），将其实现的责任延迟到子类中进行即可。

实体类：

```
public abstract class People {

    private String name;

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

    private void queueUp() {
        System.out.println(this.name + "排队取号等候");
    }

    protected abstract void service();

    private void evaluate() {
        System.out.println(this.name + "反馈评分");
    }

    public void work() {
        this.queueUp();
        this.service();
        this.evaluate();
    }

    public String getName() {
        return name;
    }
}

class ZhangSan extends People {

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

    @Override
    protected void service() {
        System.out.println(this.getName() + "办理存钱业务");
    }
}
```

```
class LiSi extends People {

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

    @Override
    protected void service() {
        System.out.println(this.getName() + "办理取钱业务");
    }
}
```

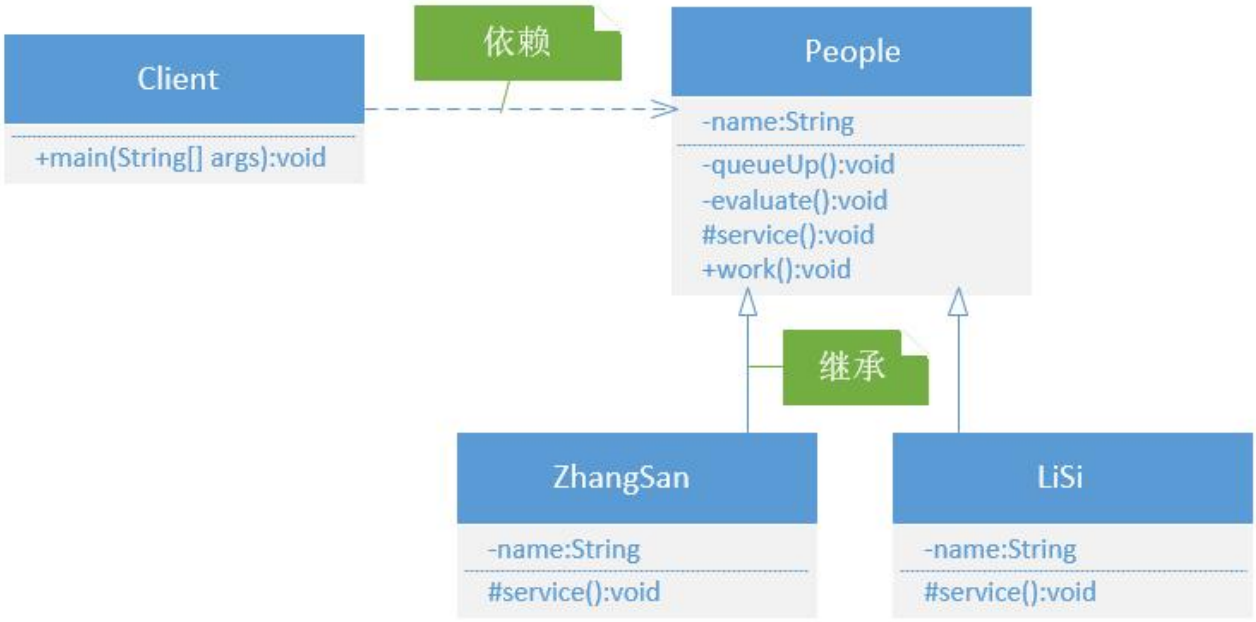
其中，work 方法就是一个模板方法，该方法不管方法实现，只负责调用方法顺序。子类继承 service 方法实现不同的业务需求。

客户端：

```
public class Client {

    public static void main(String[] args) {
        People p1 = new ZhangSan("张三");
        p1.work();

        System.out.println("===========");

        People p2 = new LiSi("李四");
        p2.work();
    }
}
```

运行结果与上文的一致。现在，客户端代码精简许多。

UML 类图表示如下：


Client
+main(String[] args):void
依赖
People
-name:String
-queueUp():void
-evaluate():void
#service():void
+work():void
继承
ZhangSan
-name:String
#service():void
LiSi
-name:String
#service():void