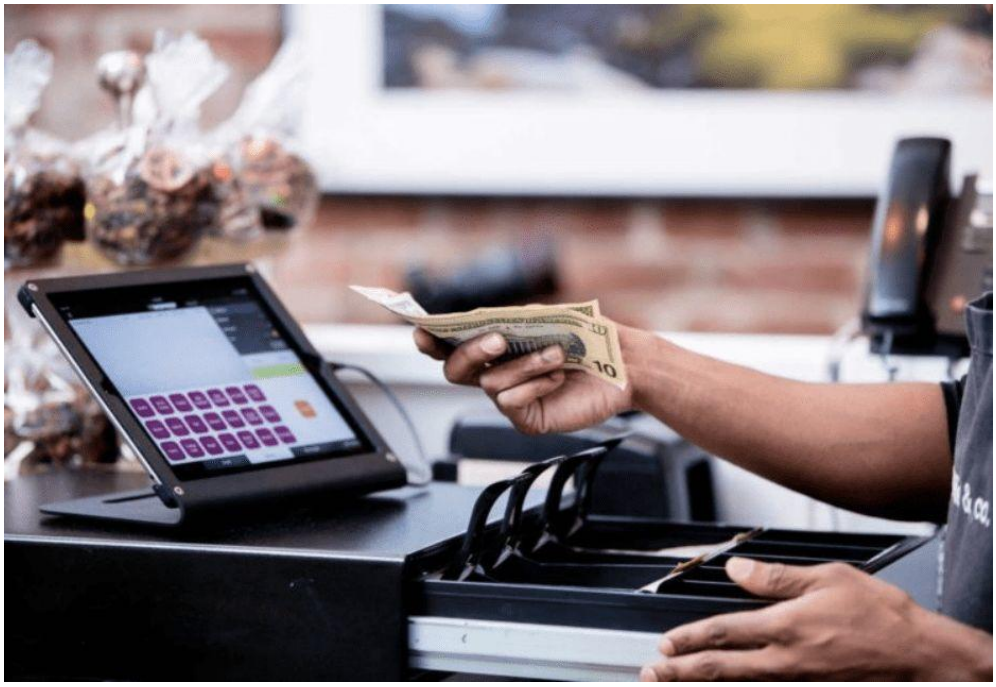
[加精]设计模式是什么鬼（访问者）

凸凹 设计模式 2019年1月3日 1.20K 0 2

众所周知，对于数据的封装我们通常会用到POJO类，它除了getter和setter之外是不包含任何业务逻辑的，也就是说它只对应一组数据并不包含任何功能。举个最常见的例子，比如数据库对应的实体类，一般我们不会在类里封装上业务逻辑，而是放在专门的Service类里去处理，也就是Service作为拜访者去访问实体类封装的数据。

现在假设有这么一个场景，我们有很多的实体数据封装类（各类食品）都要进行一段相同的业务处理（计算价格），而每个实体类对应着不同的业务逻辑（水果按斤卖，啤酒论瓶卖），但我们又不想每个类对应一个业务逻辑类（类太繁多），而是汇总到一处业务处理（结账台），那我们应该如何设计呢？



我们就以超市结账举例，首先是各种商品的实体类，包括糖、酒、和水果，它们都应该共享一些共通属性，那就先抽象出一个商品类吧。

```
1  public abstract class Product {
2
3      protected String name;// 品名
4      protected LocalDate producedDate;// 生产日期
5      protected float price;// 价格
6
7      public Product(String name, LocalDate producedDate, float price) {
8          this.name = name;
9          this.producedDate = producedDate;
10         this.price = price;
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     public LocalDate getProducedDate() {
22         return producedDate;
```

凸凹 官方

关注 私信

凸凹里歐，十余年以上开发经验，《设计
列文章作者，旨在用最通俗的方式诠释设

最新文章

- 设计模式是什么鬼（解释器）
- 设计模式是什么鬼（访问者）
- 设计模式是什么鬼（命令模式）
- 设计模式是什么鬼（建造者）
- 设计模式是什么鬼（抽象工厂）

极客快讯

- 红帽宣布从 Red Hat Enterprise Lin
  删除 MongoDB
  2019年1月19日 14
- 知名文件传输协议 SCP 被曝存在 35
  洞
  2019年1月18日 26
- Angular.js 1.7.6 发布
  2019年1月18日 58
- Dubbo 2.7.0 发布
  2019年1月18日 62
- Jenkins 2.160 发布
  2019年1月17日 98

```
    }

    public float getPrice() {
        return price;
    }

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

}
```

我们抽象出来的都是些最基本的商品属性，简单的数据封装，标准的POJO类，接下来我们把这些属性和方法都继承下来给具体商品类，它们依次是糖果、酒、和水果。

```
public class Candy extends Product {// 糖果类

    public Candy(String name, LocalDate producedDate, float price) {
        super(name, producedDate, price);
    }

}
```

```
public class Wine extends Product {// 酒类

    public Wine(String name, LocalDate producedDate, float price) {
        super(name, producedDate, price);
    }

}
```

```
public class Fruit extends Product {// 水果
    private float weight;

    public Fruit(String name, LocalDate producedDate, float price, float weight) {
        super(name, producedDate, price);
        this.weight = weight;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

基本没什么特别的，除了水果是论斤销售，所以我们加了个重量属性，仅此而已。接下来就是我们的结算业务逻辑了，超市规定即将过期的给予一定打折优惠，日常促销可以吸引更多顾客。

我们思考一下怎样设计，针对不同商品的折扣力度显然是不一样的，其实不止是打折，我们知道过期商品超市不准继续售卖，但这对于酒类商品又不存在过期问题。这个业务很明显是针对不同的类要有不同的逻辑反应了，那对于我们所访问的商品应该加以区分，用instanceof判断并分流？这显然太混乱了，代码里

```
1  public interface Visitor {// 访问者接口
2
3      public void visit(Candy candy);// 糖果重载方法
4
5      public void visit(Wine wine);// 酒类重载方法
6
7      public void visit(Fruit fruit);// 水果重载方法
8
9  }
```

三个重载方法会在响应不同的的商品类对象，这是一种功能上的多态性。下面来看具体的业务实现类，我们这里实现一个日常打折并计算最终价格的业务类DiscountVisitor。

```
1  public class DiscountVisitor implements Visitor {
2      private LocalDate billDate;
3
4      public DiscountVisitor(LocalDate billDate) {
5          this.billDate = billDate;
6          System.out.println("结算日期: " + billDate);
7      }
8
9      @Override
10     public void visit(Candy candy) {
11         System.out.println("=====糖果【" + candy.getName() + "】打折后价格=====");
12         float rate = 0;
13         long days = billDate.toEpochDay() - candy.getProducedDate().toEpochDay();
14         if (days > 180) {
15             System.out.println("超过半年过期糖果，请勿食用！");
16         } else {
17             rate = 0.9f;
18         }
19         float discountPrice = candy.getPrice() * rate;
20         System.out.println(NumberFormat.getCurrencyInstance().format(discountPrice));
21     }
22
23     @Override
24     public void visit(Wine wine) {
25         System.out.println("=====酒品【" + wine.getName() + "】无折扣价格=====");
26         System.out.println(NumberFormat.getCurrencyInstance().format(wine.getPrice()));
27     }
28
29     @Override
30     public void visit(Fruit fruit) {
31         System.out.println("=====水果【" + fruit.getName() + "】打折后价格=====");
32         float rate = 0;
33         long days = billDate.toEpochDay() - fruit.getProducedDate().toEpochDay();
34         if (days > 7) {
35             System.out.println("￥0.00元（超过一周过期水果，请勿食用！）");
36         } else if (days > 3) {
37             rate = 0.5f;
38         } else {
39             rate = 1;
40         }
41         float discountPrice = fruit.getPrice() * fruit.getWeight() * rate;
42         System.out.println(NumberFormat.getCurrencyInstance().format(discountPrice));
43     }
44
45 }
```

业务看上去也许有些复杂，其中构造方法传入初始化结单日期（第4行），糖果（第10行）的过期日设置为半年否则按9折出售，酒品（第24行）则没有过期限制，一律按原价出售，对于水果（第30行）有效期设置为一周，如果超过3天按半价出售，总之就是三种商品对应不同的计算逻辑。其实我们可以完全忽略业务实现，这里应该着重于模式的思考，让我们看看怎样客户端访问数据。

```
1  public class Client {
2      public static void main(String[] args) {
3          //小黑兔奶糖，生产日期：2018-10-1，原价：￥20.00
4          Candy candy = new Candy("小黑兔奶糖", LocalDate.of(2018, 10, 1), 20.00f);
5          Visitor discountVisitor = new DiscountVisitor(LocalDate.of(2019, 1, 1));
6          discountVisitor.visit(candy);
7          /*打印输出：
8              结算日期：2019-01-01
9              =====糖果【小黑兔奶糖】打折后价格=====
10             ￥18.00
11         */
12     }
13 }
```

品Product定义引用，让我们选购多件商品实验下。

```
1  public class Client {
2      public static void main(String[] args) {
3          // 三件商品加入购物车
4          List<Product> products = Arrays.asList(
5              new Candy("小黑兔奶糖", LocalDate.of(2018, 10, 1), 20.00f),
6              new Wine("猫泰白酒", LocalDate.of(2017, 1, 1), 1000.00f),
7              new Fruit("草莓", LocalDate.of(2018, 12, 26), 10.00f, 2.5f)
8          );
9
10         Visitor discountVisitor = new DiscountVisitor(LocalDate.of(2018, 1, 1));
11         // 迭代购物车轮流结算
12         for (Product product : products) {
13             discountVisitor.visit(product);// 此处报错
14         }
15
16     }
17 }
```

注意重点来了，我们顺利地加入购物车并迭代轮流结算每个产品，可是第13行会报错，编译器对泛化后的product很是茫然，这到底是糖还是酒？该调用哪个visit方法呢？很多朋友疑问为什么不能在运行时根据对象类型动态地派发给对应的重载方法？试想，如果我们新加一个蔬菜产品类Vegetable，但没有在Visitor里加入其重载方法visit(Vegetable vegetable)，那运行起来岂不是更糟糕？所以在设计期编译器提前就应该禁止此种情形通过编译。

难道我们设计思路错了？有没有办法把产品派发到相应的重载方法？答案是肯定的，这里涉及到一个新的概念，我们需要利用“双派发”（double dispatch）巧妙地绕过这个错误，既然访问者访问不了，我们从被访问者（产品资源）入手，来看代码，先定义一个接待者接口。

```
1  public interface Acceptable {
2      // 主动接受拜访者
3      public void accept(Visitor visitor);
4
5  }
```

可以看到这个“接待者”定义了一个接待方法，凡是“来访者”身份的都予以接受。我们先用糖果类实现这个接口，并主动接受来访者的拜访。

```
1  public class Candy extends Product implements Acceptable{// 糖果类
2
3      public Candy(String name, LocalDate producedDate, float price) {
4          super(name, producedDate, price);
5      }
6
7      @Override
8      public void accept(Visitor visitor) {
9          visitor.visit(this);// 把自己交给拜访者。
10     }
11
12 }
```

糖果类顺理成章地成为了“接待者”（其他品类雷同，此处忽略代码），并把自己（this）交给了来访者（第9行），这样绕来绕去起到什么作用呢？别急，我们先来看双派发到底是怎样实现的。

```
1  public class Client {
2      public static void main(String[] args) {
3          // 三件商品加入购物车
4          List<Acceptable> products = Arrays.asList(
5              new Candy("小黑兔奶糖", LocalDate.of(2018, 10, 1), 20.00f),
6              new Wine("猫泰白酒", LocalDate.of(2017, 1, 1), 1000.00f),
7              new Fruit("草莓", LocalDate.of(2018, 12, 26), 10.00f, 2.5f)
8          );
9
10         Visitor discountVisitor = new DiscountVisitor(LocalDate.of(2019, 1, 1));
11         // 迭代购物车轮流结算
12         for (Acceptable product : products) {
13             product.accept(discountVisitor);
14         }
15         /*打印输出：
16             结算日期：2019-01-01
17             =====糖果【小黑兔奶糖】打折后价格=====
```

```
22              ￥12.50
23          */
24      }
25  }
```

注意看第4行的购物车List<Product>已经被改为泛型Acceptable了，也就是说所有商品统统被泛化且当作“接待者”了，由于泛型化后的商品像是被打了包裹一样让拜访者无法识别品类，所以在迭代里面我们让这些商品对象主动去“接待”来访者（第13行）。这类似于警察（访问者）办案时嫌疑人（接待者）需主动接受调查并出示自己的身份证给警察，如此就可以基于个人信息查询前科并展开相关调查。

如此一来，在运行时的糖果自己是认识自己的，它就把自己递交给来访者，此时的this必然就属糖果类了，所以能得偿所愿地派发到Visitor的visit(Fruit fruit)重载方法，这样便实现了“双派发”，也就是说我们先派发给商品去主动接待，然后又把自己派发回给访问者，我不认识你，你告诉我你是谁。

终于，我们巧妙地用双派发解决了方法重载的多态派发问题，如虎添翼，访问者模式框架至此搭建竣工，之后再添加业务逻辑不必再改动数据实体类了，比如我们再增加一个针对六一儿童节打折业务，加大对糖果类、玩具类的打折力度，而不需要为每个POJO类添加对应打折方法，数据资源（实现接待者接口）与业务（实现访问者接口）被分离开来，且业务处理集中化、多态化、亦可扩展。纯粹的数据，不应该多才多艺。

本文作者：凸凹

Java设计模式(18) 设计模式(63)

上一篇：MySQL:主从复制

下一篇：高可用Redis服务架构分析与搭建

相关文章

设计模式六大原则（1）：单一职责原则

简说外观模式

23种设计模式介绍-外观模式

简说适配器模式

设计模式是什么鬼（观察者）

设计模式是什么鬼（桥接）

发表评论

评分 ☆ ☆ ☆ ☆ ☆

您的评论一针见血（必填，该内容可在后台设置）

昵称 邮箱

网址

发表评论