

Item.java

```
public interface Item {  
    public String name();  
    public Packing packing();  
    public float price();  
}
```

Packing.java

```
public interface Packing {  
    public String pack();  
}
```

步骤 2

创建实现 Packing 接口的实体类。

Wrapper.java

```
public class Wrapper implements Packing {  
  
    @Override  
    public String pack() {  
        return "Wrapper";  
    }  
}
```

Bottle.java

```
public class Bottle implements Packing {  
  
    @Override  
    public String pack() {  
        return "Bottle";  
    }  
}
```

步骤 3

创建实现 Item 接口的抽象类，该类提供了默认的功能。

Burger.java

```
public abstract class Burger implements Item {  
  
    @Override  
    public Packing packing() {  
        return new Wrapper();  
    }  
  
    @Override  
    public abstract float price();  
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {  
  
    @Override  
    public Packing packing() {  
        return new Bottle();  
    }  
  
    @Override  
    public abstract float price();  
}
```

步骤 4

创建扩展了 Burger 和 ColdDrink 的实体类。

VegBurger.java

```
public class VegBurger extends Burger {  
  
    @Override  
    public float price() {  
        return 25.0f;  
    }  
}
```

```
@Override
public String name() {
    return "Veg Burger";
}
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

    @Override
    public float price() {
        return 50.5f;
    }

    @Override
    public String name() {
        return "Chicken Burger";
    }
}
```

Coke.java

```
public class Coke extends ColdDrink {

    @Override
    public float price() {
        return 30.0f;
    }

    @Override
    public String name() {
        return "Coke";
    }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

    @Override
    public float price() {
        return 35.0f;
    }

    @Override
    public String name() {
        return "Pepsi";
    }
}
```

步骤 5

创建一个 Meal 类，带有上面定义的 Item 对象。

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
    private List<Item> items = new ArrayList<Item>();

    public void addItem(Item item){
        items.add(item);
    }

    public float getCost(){
        float cost = 0.0f;
        for (Item item : items) {
            cost += item.price();
        }
        return cost;
    }

    public void showItems(){
        for (Item item : items) {
            System.out.print("Item : "+item.name());
        }
    }
}
```

```
        System.out.print(", Packing : "+item.packing().pack());
        System.out.println(", Price : "+item.price());
    }
}
}
```

步骤 6

创建一个 MealBuilder 类，实际的 builder 类负责创建 Meal 对象。

MealBuilder.java

```
public class MealBuilder {

    public Meal prepareVegMeal (){
        Meal meal = new Meal();
        meal.addItem(new VegBurger());
        meal.addItem(new Coke());
        return meal;
    }

    public Meal prepareNonVegMeal (){
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Pepsi());
        return meal;
    }
}
```

步骤 7

BuiderPatternDemo 使用 MealBuilder 来演示建造者模式（Builder Pattern）。

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
    public static void main(String[] args) {
        MealBuilder mealBuilder = new MealBuilder();

        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        vegMeal.showItems();
        System.out.println("Total Cost: " +vegMeal.getCost());

        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("\n\nNon-Veg Meal");
        nonVegMeal.showItems();
        System.out.println("Total Cost: " +nonVegMeal.getCost());
    }
}
```

步骤 8

执行程序，输出结果：

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

← 单例模式

原型模式 →



2 篇笔记



写笔记



建造者模式举例：去肯德基点餐，我们可以认为点餐就属于一个建造订单的过程。我们点餐的顺序是无关的，点什么东西也是没有要求的，可以单点，也可以点套餐，也可以套餐加单点，但是最后一定要点确认来完成订单。

```

public class OrderBuilder{
    private Burger mBurger;
    private Suit mSuit;

    //单点汉堡,num为数量
    public OrderBuilder burger(Burger burger , int num){
        mBurger = burger;
    }

    //点套餐，实际中套餐也可以点多份
    public OrderBuilder suit(Suit suit, int num){
        mSuit = suit;
    }

    //完成订单
    public Order build(){
        Order order = new Order();
        order.setBurger(mBurger);
        order.setSuit(mSuit);
        return order;
    }
}

```

另外适用于快速失败，在 build 时可以做校验，如果不满足必要条件，则可以直接抛出创建异常，在 OkHttpClient 中的 Request.Builder 中就是这样用的。

```

public Request build() {
    if (url == null) throw new IllegalStateException("url == null");
    return new Request(this);
}

```

例如订单要求价格至少达到 30 块：

```

//完成订单
public Order build(){
    Order order = new Order();
    order.setBurger(mBurger);
    order.setSuit(mSuit);
    if(order.getPrice() < 30){
        throw new BusinessException("订单金额未达到30元");
    }
    return order;
}

```

另外，在构建时如果有必传参数和可选参数，可以为 Builder 类添加构造函数来保证必传参数不会遗漏，例如在构建一个 http 请求时，url 是必传的：

```

public class RequestBuilder{
    private final String mUrl;
    private Map<String, String> mHeaders = new HashMap<String, String>();

    private RequestBuilder(String url){
        mUrl = url;
    }

    public static RequestBuilder newBuilder(String url){
        return new RequestBuilder(url);
    }

    public RequestBuilder addHeader(String key, String value){
        mHeaders.put(key, value);
    }

    public Request build(){
        Request request = new Request();
        request.setUrl(mUrl);
        request.setHeaders(mHeaders);
        return request;
    }
}

```



建造者模式，又称生成器模式：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

三个角色：**建造者**、**具体的建造者**、**监工**、**使用者**（严格来说不算）

- **建造者角色**：定义生成实例所需要的所有方法；
- **具体的建造者角色**：实现生成实例所需要的所有方法，并且定义获取最终生成实例的方法；
- **监工角色**：定义使用建造者角色中的方法来生成实例的方法；
- **使用者**：使用建造者模式。

注意：定义中“**将一个复杂的构建过程与其表示相分离**”，表示并不是由建造者负责一切，而是由监工负责控制（定义）一个复杂的构建过程，由各个不同的建造者分别负责实现构建过程中所用到的所有构建步骤。不然，就无法做到“使得同样的构建过程可以创建不同的表示”这一目标。

建造者角色：

```
public abstract class Builder {  
    public abstract void buildPart1();  
    public abstract void buildPart2();  
    public abstract void buildPart3();  
}
```

监工角色：

```
public class Director { // 将一个复杂的构建过程与其表示相分离  
    private Builder builder; // 针对接口编程，而不是针对实现编程  
    public Director(Builder builder) {  
        this.builder = builder;  
    }  
    public void setBuilder(Builder builder) {  
        this.builder = builder;  
    }  
  
    public void construct() { // 控制（定义）一个复杂的构建过程  
        builder.buildPart1();  
        for (int i = 0; i < 5; i++) { // 提示：如果想在运行过程中替换构建算法，可以考虑结合策略模式。  
            builder.buildPart2();  
        }  
        builder.buildPart3();  
    }  
}
```

具体的建造者角色：


```
/**  
 * 此处实现了建造纯文本文档的具体建造者。  
 * 可以考虑再实现一个建造HTML文档、XML文档，或者其它什么文档的具体建造者。  
 * 这样，就可以使得同样的构建过程可以创建不同的表示  
 */  
public class ConcreteBuilder1 extends Builder {  
    private StringBuffer buffer = new StringBuffer(); // 假设 buffer.toString() 就是最终生成的产品  
  
    @Override  
    public void buildPart1() { // 实现构建最终实例需要的所有方法  
        buffer.append("Builder1 : Part1\n");  
    }  
  
    @Override  
    public void buildPart2() {  
        buffer.append("Builder1 : Part2\n");  
    }  
  
    @Override  
    public void buildPart3() {  
        buffer.append("Builder1 : Part3\n");  
    }  
  
    public String getResult() { // 定义获取最终生成实例的方法  
        return buffer.toString();  
    }  
}
```

```
}

客户角色：

public class Client {
    public void testBuilderPattern() {
        ConcreteBuilder1 b1 = new ConcreteBuilder1();//建造者
        Director director = new Director(b1);//监工
        director.construct();//建造实例(监工负责监督，建造者实际建造)
        String result = b1.getResult();//获取最终生成结果
        System.out.printf("the result is :%n%s", result);
    }
}
```



jade 2个月前 (09-20)



Communicate Like A Team

Set Your Team Up For Successful Communication. Get Grammarly Business.

LEARN MORE



<div>在线实例</div> <ul style="list-style-type: none">· HTML 实例· CSS 实例· JavaScript 实例· Ajax 实例· jQuery 实例· XML 实例· Java 实例	<div>字符集&工具</div> <ul style="list-style-type: none">· HTML 字符集设置· HTML ASCII 字符集· HTML ISO-8859-1· HTML 实体符号· HTML 拾色器· JSON 格式化工具	<div>最新更新</div> <ul style="list-style-type: none">· JavaScript let ...· MySQL 运算符· C 语言的布尔类...· JS 中彻底删除 J...· 搞笑程序员表情...· Java 中 JSON ...· C++ const 关键...	<div>站点信息</div> <ul style="list-style-type: none">· 意见反馈· 免责声明· 关于我们· 文章归档
---	---	---	---

关注微信



Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1

反馈/建议