# 设计模式

- 设计模式（Design pattern）代表了**最佳的实践**，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的**解决方案。**这些解决方案是众多软件开发人员经过相当长的**一段时间的试验和错误总结出来的。**
- 设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计**经验的总结。**使用设计模式是为了**重用代码、让代码更容易被他人理解、保证代码可靠性。**项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

**开闭原则**

由Bertrand Meyer提出的开闭原则（Open Closed Principle）是指，软件应该对扩展开放，而对修改关闭。这里的意思是在**增加新功能的时候，能不改代码就尽量不要改**，如果只增加代码就完成了新功能，那是最好的。

一共有 23 种设计模式。这些模式可以分为三大类：

创建型模式（Creational Patterns）：核心思想是要把对象的创建和使用相分离，这样使得两者能相对独立。

结构型模式（Structural Patterns）：结构型模式主要涉及**如何组合各种对象**以便获得更好、更灵活的结构。虽然面向对象的继承机制提供了最基本的子类扩展父类的功能，但结构型模式不仅仅简单地使用继承，而更多地通过**组合与运行期的动态组合**来实现更灵活的功能。

行为型模式（Behavioral Patterns）：主要涉及算法和对象间的职责分配。通过使用对象组合，行为型模式可以描述一组对象应该如何协作来完成一个整体任务。

这次课主要讲解下列：

**创建型模式**：**单例模式、简单工厂、工厂方法模式、抽象工厂模式、生成器模式(建造者模式)**

**结构型模式**：**装饰器模式**

**行为型模式**：**策略模式、观察者模式**

# 单例模式

**一个类只有一个实例**

## 注意：

- 1、单例类只能有一个实例。
- 2、单例类必须**自己创建自己**的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

## 例子

- 1、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。
- 2、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

## 优点

- 1、在内存里只有一个实例，减少了内存的开销。
- 2、避免对资源的多重占用（比如多个程序写文件操作）。

## 几种实现方式

**此处在AS中只给出双重校验锁的Java和Kotlin代码**

**Kotlin的其他实现方式请见 这里**

下面给出Java的几种方式

### 1、懒汉式，线程不安全

可能有多个线程同时进入了 if (instance == null) ,就**会实例化多次**

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

### 2、懒汉式，线程安全

必须加锁 synchronized 才能保证单例，但加锁**会影响效率。**当1个线程进getInstance之后,其他的线程只能等待,降低了性能

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

### 3、饿汉式

类加载时就初始化，没有用到的时候也存在，**浪费了内存**

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

### 4、双重校验锁（DCL，即 double-checked locking)

最好的一种。这种方式采用双锁机制，**安全且在多线程情况下能保持高性能。**

当两个并发线程(thread1和thread2)访问synchronized代码块时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞，必须等待当前线程执行完这个代码块以后才能执行该代码块。

这里的volatile也必不可少，volatile关键字可以防止jvm**指令重排**优化，

**什么是指令重排？ 更加详细的内容请见** https://blog.csdn.net/blueheart20/article/details/52117761/

> 在计算机执行指令的顺序在经过程序编译器编译之后形成的指令序列，一般而言，这个指令序列是会输出确定的结果；以确保每一次的执行都有确定的结果。但是，一般情况下，CPU和编译器为了提升程序执行的效率，会按照一定的规则允许进行指令优化，在某些情况下，这种优化会带来一些执行的逻辑问题，主要的原因是代码逻辑之间是存在一定的先后顺序，在并发执行情况下，会发生二义性，即按照不同的执行逻辑，会得到不同的结果信息。

> 因为 singleton = new Singleton() 这句话可以分为三步：
>
> 1. 为 singleton 分配内存空间；
> 2. 初始化 singleton；
> 3. 将 singleton 指向分配的内存空间。
>
> 但是由于JVM具有指令重排的特性，执行顺序有可能变成 1-3-2。 指令重排在单线程下不会出现问题，但是在多线程下会导致一个线程获得一个未初始化的实例。例如：线程T1执行了1和3，此时T2调用 getInstance() 后发现 singleton 不为空，因此返回 singleton， 但是此时的 singleton 还没有被初始化。
>
> 使用 volatile 会禁止JVM指令重排，从而保证在多线程下也能正常执行。

volatile关键字的第二个作用，保证变量在多线程运行时的可见性： **关于内存模型详细请见这里和这里**

> 在 JDK1.2 之前，Java的内存模型实现总是从主存（即共享内存）读取变量，是不需要进行特别的注意的。而在当前 的 Java 内存模型下，线程可以把变量保存本地内存（比如机器的寄存器）中，而不是直接在主存中进行读写。这就 可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数 据的不一致。 要解决这个问题，就需要把变量声明为 volatile，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行 读取。

**java代码**

```
public class Singleton {
    private volatile static Singleton instance;
    private Singleton (){}
```

```
    public static Singleton getInstance() {
        if (instance == null) {                 //(1)
            synchronized (Singleton.class) {
                if (instance == null) {         //(2)
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

如果只有（1）处的判空，没有（2）处的判空，则当有两个线程进入if后，其中一个线程进入synchronized代码块，产生了一个实例，执行完了这个代码块；然后下一个线程进入synchronized代码块，**会出现又产生一个实例的问题。**

如果只有（2）处的判空，没有（1）处的判空，当有两个线程时，则若已经有了实例，某一个线程进入synchronized代码块，if语句判断为假，执行完synchronized代码块；之后另一个线程再进入。**后面进入的线程在刚才一直在等待前一个线程执行完synchronized代码块，浪费时间**，降低了代码效率。如果1、2处判空都有，则当已经有了实例是，直接第一个判断为假，节省了时间，提高了效率。

**Kotlin代码，用委托，延迟属性 Lazy**

```
class KtSingleton {
    companion object {
        val instance: KtSingleton by lazy(mode =
LazyThreadSafetyMode.SYNCHRONIZED) {
            //利用LazyThreadSafetyMode.SYNCHRONIZED参数来保证线程安全
            KtSingleton()
        }
    }
}
```

# 观察者模式

当一个对象被修改时，则会自动通知依赖它的对象，自动更新。

## 实例：

- 西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。
- 服务器发送推送新闻，手机客户端接收新闻。
- 参考**袁兵**学长讲设计模式时的例子：气象站的气象数据发生改变的时候,客户端也要对这种变化作出反应。比如这里的客户端就是手机,需要手机能够实时刷新当前的天气状况。可以有两种方式来实现
    1. 由客户端不断的向服务器请求数据,当服务器的数据发生变化之后,客户端也就能够知道这种变化（轮询）（缺点：一直会占用网络，**浪费客户端和服务器的性能**）
    2. 由服务器主动向客户端推送消息,当数据发生改变的时候,服务器将这种变化推送到客户端（推送，在数据发生变化之后再通知客户端数据发生了变化）（优点：比轮询更**节省资源**）

## 使用场景：

- 一个对象的改变将导致其他一个或多个对象也发生改变。
- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。（MVVM中，Model把数据给ViewModel，ViewModel把数据给View）

## 观察者模式有两个部分：

**1.可观察对象** `Observable`（例如上例中的服务器，气象站，菩萨）它在数据改变的时候通知观察者

**2.观察者** `Observer`（例如上例中的客户端，老乌龟）当 `Observable` 中的数据发生改变后会回调其中的方法来通知 `Observer` 数据发生了改变。

①我们在 `Observer` 中持有一个 `Observable` 对象,在通知更新时不传参数,而是在 `Observer` 中通过 `Observable` 对象来获取数据。

②也可以把数据作为参数直接传递给 Observer , Observer 就不需要持有一个 Observable 对象 此处不再赘述

## *详细见代码*

### 介绍几种，一种是通用最基本写法（Java和Kotlin各一份），一种是利用Kotlin委托的写法，一种是MVVM里的View观察ViewModel里的数据(若要演示，见DaysMatter代码)

**最基本的写法**

见AS代码

**Kotlin用委托方式：**

Delegates.observable() 函数接受两个参数: 第一个是初始化值, 第二个是属性值变化事件的响应器(handler)。在属性赋值后会执行事件的响应器(handler)，它有三个参数：被赋值的属性、旧值和新值：

```
var news: String by Delegates.observable("昨天的新闻") {
    property, old, new ->
    Log.d("property name is ${property.name}", "旧值：$old -> 新值：$new")
}
```

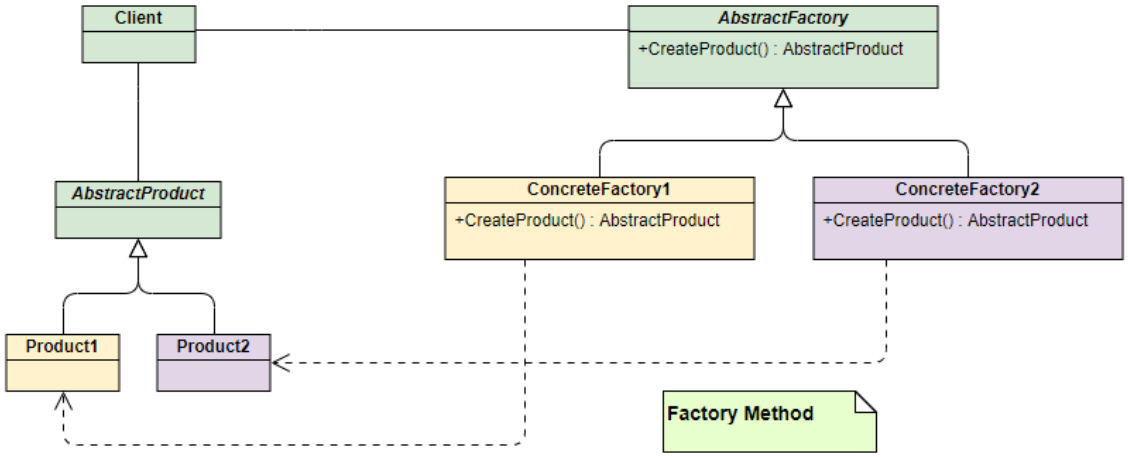
**MVVM里的View观察ViewModel里的数据**

```
//var mldTodayOnHistoryList: MutableLiveData<List<TodayOnHistoryItem>>
mainViewModel.mldTodayOnHistoryList?.observe(
    mActivity,
    //通过对象表达式实现一个匿名内部类的对象
    object : Observer<List<TodayOnHistoryItem>> {
        override fun onChanged(t: List<TodayOnHistoryItem>) {
            //do something...
        }
    })
```

# 工厂模式

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。

定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。和简单工厂模式中工厂负责生产所有产品相比，工厂方法模式将生成具体产品的任务分发给具体的产品工厂，每一个工厂负责一个指定的产品




## 例子：

需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。

## 使用场景：

用户知道它所要创建对象的类(产品类)，但是不关心如何创建的时候。

## 加深理解：

工厂模式也就是鼠标工厂是个父类，有生产鼠标这个接口。戴尔鼠标工厂，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，例如若创建戴尔鼠标工厂，则后续直接调用 `鼠标工厂.生产鼠标()` 即可生产戴尔鼠标。

## 优点：

1、一个调用者想创建一个对象，只要知道其名称就可以了，不关心是如何创建的。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，不需要知道是怎么造出来的，调用者只关心产品的接口。

## 注意事项：

作为一种创建类模式，在任何需要生成**复杂对象**的地方，都可以**使用工厂方法模式。**有一点需要注意的地方就是复杂对象适合使用工厂模式，而**简单的对象**，特别是只需要通过 new 就可以完成创建的对象，**无需使用工厂模式。如果使用工厂模式**，就需要引入一个工厂类，这就会**增加系统的复杂度。**

## 例示代码

### 例示1（AS中无例示1）

使用生产手机的例子来讲解该模式。

**Phone**：手机标准规范(AbstractProduct)

```
public interface Phone {
    void make();
}
```

**MiPhone类**：制造小米手机（Product1）

```
public class MiPhone implements Phone {
    public MiPhone() {
        this.make();
    }
    @Override
    public void make() {
        System.out.println("make xiaomi phone!");
    }
}
```

**IPhone类**：制造苹果手机（Product2）

```
public class IPhone implements Phone {
    public IPhone() {
        this.make();
    }
    @Override
    public void make() {
        System.out.println("make iphone!");
    }
}
```

**AbstractFactory**：生产不同产品的工厂的抽象类

```
public interface AbstractFactory {
    Phone makePhone();
}
```

**XiaoMiFactory类**：生产小米手机的工厂（ConcreteFactory1）

```
public class XiaoMiFactory implements AbstractFactory{
    @Override
    public Phone makePhone() {
        return new MiPhone();
    }
}
```

**AppleFactory类**：生产苹果手机的工厂（ConcreteFactory2）

```
public class AppleFactory implements AbstractFactory {
    @Override
    public Phone makePhone() {
        return new IPhone();
    }
}
```

```
public class Demo {
    public static void main(String[] arg) {
        AbstractFactory miFactory = new XiaoMiFactory();
        AbstractFactory appleFactory = new AppleFactory();
        miFactory.makePhone();            // make xiaomi phone!
        appleFactory.makePhone();        // make iphone!
    }
}
```

## 例示2

**此处只提供Kotlin代码**

```
interface Product2 {
    fun getPrice(): Int
    fun getName(): String
}
```

```
class Product2A: Product2 {
    override fun getPrice(): Int {
        return 100
    }

    override fun getName(): String {
        return "Product2A"
    }
}
```

```
class Product2B: Product2 {
    override fun getPrice(): Int {
        return 200
    }

    override fun getName(): String {
        return "Product2B"
    }
}
```

```
class Product2C: Product2 {
    override fun getPrice(): Int {
        return 300
    }

    override fun getName(): String {
        return "Product2C"
    }
}
```

```
interface Factory2 {
    fun createProduct(): Product2
}
```

```
class Factory2A : Factory2{
    override fun createProduct(): Product2 {
        return Product2A()
    }
}
```

```
class Factory2B : Factory2{
    override fun createProduct(): Product2 {
        return Product2B()
    }
}
```

```
class Factory2C : Factory2{
    override fun createProduct(): Product2 {
        return Product2C()
    }
}
```

```
val factoryA = Factory2A()
val productA: Product2 = factoryA.createProduct()
Log.d("factorymethod---", "name: ${productA.getName()}; price:
${productA.getPrice()}")
val factoryB = Factory2B()
val productB: Product2 = factoryB.createProduct()
Log.d("factorymethod---", "name: ${productB.getName()}; price:
${productB.getPrice()}")
val factoryC = Factory2C()
val productC: Product2 = factoryC.createProduct()
Log.d("factorymethod---", "name: ${productC.getName()}; price:
${productC.getPrice()}")
```

## 输出：

```
2020-07-10 15:31:45.321 29194-29194/com.redrock.designpattern D/factorymethod--
-: name: Product2A; price: 100
2020-07-10 15:31:45.321 29194-29194/com.redrock.designpattern D/factorymethod--
-: name: Product2B; price: 200
2020-07-10 15:31:45.321 29194-29194/com.redrock.designpattern D/factorymethod--
-: name: Product2C; price: 300
```

当需要**增加一个新产品Product2D,只需要新建对应的Factory2D来实现生产功能即可,对原有的代码没有任何影响，符合开放封闭原则**，但是由于**每增加一个产品，都需要新增对应的生产工厂**，导致**增加额外**的开发**工作量。**

# 生成器模式(建造者模式)

使用多个简单的对象一步一步构建成一个复杂的对象。一个 Builder 类会一步一步构造最终的对象。

**当一个类的构造函数参数个数超过4个，而且这些参数有些是可选的参数，考虑使用构造者模式。**

## 解决的问题

当一个类的构造函数参数超过4个，而且这些参数有些是可选的时，我们通常有两种办法来构建它的对象。 例如我们现在有如下一个类计算机类`Computer`，其中CPU与RAM是必填参数，而其他3个是可选参数，那么我们如何构造这个类的实例呢,通常有两种常用的方式：

```
public class Computer {
    private String cpu;//必须
    private String ram;//必须
    private int usbCount;//可选
    private String keyboard;//可选
    private String display;//可选
}
```

第一：折叠构造函数模式（telescoping constructor pattern ），这个我们经常用,如下代码所示

```
public class Computer {
     ...
    public Computer(String cpu, String ram) {
        this(cpu, ram, 0);
    }
    public Computer(String cpu, String ram, int usbCount) {
        this(cpu, ram, usbCount, "罗技键盘");
    }
    public Computer(String cpu, String ram, int usbCount, String keyboard) {
        this(cpu, ram, usbCount, keyboard, "三星显示器");
    }
    public Computer(String cpu, String ram, int usbCount, String keyboard,
String display) {
        this.cpu = cpu;
        this.ram = ram;
        this.usbCount = usbCount;
        this.keyboard = keyboard;
        this.display = display;
    }
}
```

第二种：Javabean 模式，如下所示

```
public class Computer {
        ...

    public String getCpu() {
        return cpu;
    }
    public void setCpu(String cpu) {
        this.cpu = cpu;
    }
    public String getRam() {
        return ram;
    }
    public void setRam(String ram) {
        this.ram = ram;
    }
    public int getUsbCount() {
```

```
        return usbCount;
    }
...
}
```

那么这两种方式有什么弊端呢？ 第一种主要是使用及阅读不方便。第二种方式在构建过程中对象的状态容易发生变化，造成错误。因为这个类中的属性是分步设置的，所以就容易出错。

为了解决这两个问题，builder模式就横空出世了。

## 实例：

- 1、去快餐店，汉堡、可乐、薯条、炸鸡等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。
- 2、图片处理框架Glide，网络请求框架Retrofit等都使用了此模式。
- ```
  mRetrofit = new Retrofit.Builder()
          .baseUrl("https://v1.alapi.cn/") //设置网络请求的Url地址
          .addConverterFactory(GsonConverterFactory.create()) //设置数据解析器
          .build();
  ```
- ```
  Glide.with(context).load(item.data?.header?.icon).into(ivAvatar)
  ```
- 3、Android中的AlertDialog对话框的构建过程就是建造者模式的典型应用

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("标题");
builder.setMessage("内容");
builder.setPositiveButton("确定", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {}
});
builder.setNegativeButton("取消", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {}
});
builder.show();
```

## 例示代码

### 例示1（AS中无例示1）

```
public class Computer {
    private final String cpu;//必须
    private final String ram;//必须
    private final int usbCount;//可选
    private final String keyboard;//可选
    private final String display;//可选

    private Computer(Builder builder){
        this.cpu=builder.cpu;
```

```
        this.ram=builder.ram;
        this.usbCount=builder.usbCount;
        this.keyboard=builder.keyboard;
        this.display=builder.display;
    }
    public static class Builder{
        private String cpu;//必须
        private String ram;//必须
        private int usbCount;//可选
        private String keyboard;//可选
        private String display;//可选

        public Builder(String cpu,String ram){
            this.cpu=cpu;
            this.ram=ram;
        }

        public Builder setUsbCount(int usbCount) {
            this.usbCount = usbCount;
            return this;
        }
        public Builder setKeyboard(String keyboard) {
            this.keyboard = keyboard;
            return this;
        }
        public Builder setDisplay(String display) {
            this.display = display;
            return this;
        }
        public Computer build(){
            return new Computer(this);
        }
    }
}
```

## 如何使用

在客户端使用链式调用，一步一步的把对象构建出来。

```
Computer computer=new Computer.Builder("因特尔","三星")
                .setDisplay("三星24寸")
                .setKeyboard("罗技")
                .setUsbCount(2)
                .build();
```

## 例示2

### 只提供Kotlin代码

创建一个表示食物条目和食物包装的接口。

```
interface Item {
    fun name(): String
    fun packing(): Packing
    fun price(): Float
}
```

```
interface Packing {
    fun pack(): String
}
```

创建实现 Packing 接口的实体类。

```
class Wrapper :Packing{
    override fun pack(): String {
        return "Wrapper"
    }
}
```

```
class Bottle :Packing{
    override fun pack(): String {
        return "Bottle"
    }
}
```

创建实现 Item 接口的抽象类，该类提供了默认的功能。

```
abstract class Burger :Item{
    override fun packing(): Packing {
        return Wrapper()
    }
}
```

```
abstract class ColdDrink :Item{
    override fun packing(): Packing {
        return Bottle()
    }
}
```

创建扩展了 Burger 和 ColdDrink 的实体类。

```
class VegBurger : Burger() {
    override fun name(): String {
        return "Veg Burger"
    }

    override fun price(): Float {
        return 25.0f
    }
}
```

```
class ChickenBurger : Burger() {
    override fun name(): String {
        return "Chicken Burger"
    }

    override fun price(): Float {
        return 50.5f
    }
}
```

```
class Coke :ColdDrink(){
    override fun name(): String {
        return "Coke"
    }

    override fun price(): Float {
        return 30.0f
    }
}
```

```
class Pepsi :ColdDrink(){
    override fun name(): String {
        return "Pepsi"
    }

    override fun price(): Float {
        return 35.0f
    }
}
```

创建一个 Meal 类，带有上面定义的 Item 对象。

```
class Meal {
    val items: MutableList<Item> = ArrayList()

    fun addItem(item: Item) {
        items.add(item)
    }

    fun getCost(): Float {
        var cost: Float = 0.0f
        for (item: Item in items) {
            cost += item.price()
```

```
        }
        return cost
    }

    fun showItems() {
        for (i in items.indices) {
            Log.d("builder---", "showItems: name: ${items[i].name()}, Packing:
${items[i].packing().pack()}, Price: ${items[i].price()}")
        }
    }
}
```

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

```
class MealBuilder {
    fun prepareVegMeal(): Meal {
        val meal = Meal()
        meal.addItem(VegBurger())
        meal.addItem(Coke())
        return meal
    }

    fun prepareNonVegMeal(): Meal {
        val meal = Meal()
        meal.addItem(ChickenBurger())
        meal.addItem(Pepsi())
        return meal
    }
}
```

```
val mealBuilder = MealBuilder()

val vegMeal: Meal = mealBuilder.prepareVegMeal()
vegMeal.showItems()
Log.d("builder---", "Total Cost: " + vegMeal.getCost())

val nonVegMeal: Meal = mealBuilder.prepareNonVegMeal()
nonVegMeal.showItems()
Log.d("builder---", "Total Cost: " + nonVegMeal.getCost())
```

**输出：**

```
2020-07-10 17:30:36.913 19946-19946/com.redrock.designpattern D/builder---:
showItems: name: Veg Burger, Packing: Wrapper, Price: 25.0
2020-07-10 17:30:36.913 19946-19946/com.redrock.designpattern D/builder---:
showItems: name: Coke, Packing: Bottle, Price: 30.0
2020-07-10 17:30:36.913 19946-19946/com.redrock.designpattern D/builder---:
Total Cost: 55.0
2020-07-10 17:30:36.913 19946-19946/com.redrock.designpattern D/builder---:
showItems: name: Chicken Burger, Packing: Wrapper, Price: 50.5
2020-07-10 17:30:36.913 19946-19946/com.redrock.designpattern D/builder---:
showItems: name: Pepsi, Packing: Bottle, Price: 35.0
2020-07-10 17:30:36.913 19946-19946/com.redrock.designpattern D/builder---:
Total Cost: 85.5
```

## 写在最后

设计模式就**是经验，是思想**，是为了方便开发的。个人认为设计模式最重要的是**思想**，如果是为了追求**“设计模式”**而使用设计模式，而不是为了方便开发，那么我认为意义不大。如果有任何错误，还请指出，谢谢。