前面我们说过，设计模式的核心操作是去观察你整个逻辑里面的**变与不变**，然后将变与不变分离，达到使变化的部分灵活、不变的地方稳定的目的。我们本节就来验证一下这个思想。

先来说说构造器

在介绍工厂模式之前，为了辅助大家的理解，我想先在这儿给大家介绍一下构造器模式。

别看这个名字很吓人（其实设计模式里每个名字好像都挺吓人的哈哈哈），这玩意儿你几乎天天用（所以咱们不单独给它开小节），不信你来看看：

有一天你写了个公司员工信息录入系统，这个系统开发阶段用户只有你自己，想怎么玩怎么玩。于是在创建“自己”这个唯一的用户的时候，你可以这么写：

```
const liLei = {
    name: '李雷',
    age: 25,
    career: 'coder',
}
```

有一天你的同桌韩梅梅突然说：“李雷，让我瞅瞅你的系统做得咋样了，我也想被录进去”。你说好，不就多一个人的事情吗，于是代码里手动多了一个韩梅梅：

```
const liLei = {
    name: '李雷',
    age: 25,
    career: 'coder',
}

const hanMeiMei = {
    name: '韩梅梅',
    age: 24,
    career: 'product manager'
}
```

又过了两天你老板过来了，说李雷，系统今天提测了，先把部门的 500 人录入看看功能。李雷心想，500 个对象字面量，要死要死，还好我有**构造函数**。于是李雷写出了一个可以自动创建用户的 User 函数：

```
function User(name , age, career) {
    this.name = name
    this.age = age
    this.career = career
}
```

楼上个这 User，就是一个**构造器**。此处我们采用了 ES5 构造函数的写法，因为 ES6 中的 class 其实本质上还是函数，class 语法只是语法糖，构造函数，才是它的真面目。

接下来要做的事情，就是让程序自动地去读取数据库里面一行行的员工信息，然后把拿到的姓名、年龄等字段塞进User函数里，进行一个简单的调用：

```
const user = new User(name, age, career)
```

从此李雷再也不用手写字面量。

像 User 这样当新建对象的内存被分配后，用来初始化该对象的特殊函数，就叫做构造器。在 JavaScript 中，我们使用构造函数去初始化对象，就是应用了**构造器模式**。这个模式太简单了，简单到我这一通讲对很多同学来说其实并不必要，大家都是学过 JavaScript 基础的人，都知道怎么 new 一个对象。但是我们洋洋洒洒这么一段的目的，并不是为了带大家复习构造函数本身的用法，而是希望大家去思考开篇我们提到的问题：

在创建一个user过程中，谁变了，谁不变？

很明显，变的是每个user的姓名、年龄、工种这些值，这是用户的**个性**，不变的是每个员工都具备姓名、年龄、工种这些属性，这是用户的**共性**。

那么构造器做了什么？

构造器是不是将 name、age、career 赋值给对象的过程封装，确保了每个对象都具备这些属性，确保了**共性**的不变，同时将 name、age、career 各自的取值操作开放，确保了**个性**的灵活？

如果在使用构造器模式的时候，我们本质上是去抽象了每个对象实例的变与不变。那么使用工厂模式时，我们要做的就是去抽象不同构造函数（类）之间的变与不变。

简单工厂模式

咱们先不说简单工厂模式定义是啥，咱们先来看李雷的新需求：

老板说这个系统录入的信息也太简单了，程序员和产品经理之间的区别一个简单的 `career` 字段怎么能说得清？我要求这个系统具备**给不同工种分配职责说明**的功能。也就是说，要给每个工种的用户加上一个个性化的字段，来描述他们的工作内容。

完了，这下员工的共性被拆离了。还好有构造器，李雷心想不就是多写个构造器的事儿吗，我写：

```
function Coder(name , age) {
    this.name = name
    this.age = age
    this.career = 'coder'
    this.work = ['写代码','写系分', '修Bug']
}
function ProductManager(name, age) {
    this.name = name
    this.age = age
    this.career = 'product manager'
    this.work = ['订会议室', '写PRD', '催更']
}
```

现在我们有两个类（后面可能还会有更多的类），麻烦的事情来了：难道我每从数据库拿到一条数据，都要人工判断一下这个员工的工种，然后手动给它分配构造器吗？不行，这也是一个“变”，我们把这个“变”交给一个函数去处理：

```
function Factory(name, age, career) {
    switch(career) {
        case 'coder':
            return new Coder(name, age)
            break
        case 'product manager':
            return new ProductManager(name, age)
            break
        ...
}
```

看起来是好一些了，至少我们不用操心构造函数的分配问题了。但是大家注意我在 switch 的末尾写了个省略号，这个省略号比较恐怖。看着这个省略号，李雷哭了，他想到：整个公司上下有数十个工种，难道我要手写数十个类、数十行 switch 吗？

当然不！回到我们最初的问题：大家仔细想想，在楼上这两段并不那么好的代码里，**变的是什么？不变的又是什么？**

Coder 和 ProductManager 两个工种的员工，是不是仍然存在都拥有 name、age、career、work 这四个属性这样的共性？它们之间的区别，在于每个字段取值的不同，以及 work 字段需要随 career 字段取值的不同而改变。这样一来，我们是不是对共性封装得不够彻底？那么相应地，共性与个性是不是分离得也不够彻底？ 现在我们把相同的逻辑封装回User类里，然后把这个承载了共性的 User 类和个性化的逻辑判断写入同一个函数：

```
function User(name , age, career, work) {
    this.name = name
    this.age = age
    this.career = career
    this.work = work
}

function Factory(name, age, career) {
    let work
    switch(career) {
        case 'coder':
            work =  ['写代码','写系分', '修Bug']
            break
        case 'product manager':
            work = ['订会议室', '写PRD', '催更']
            break
        case 'boss':
            work = ['喝茶', '看报', '见客户']
        case 'xxx':
            // 其它工种的职责分配
            ...

    return new User(name, age, career, work)
}
```

这样一来，我们要做事情是不是简单太多了？不用自己时刻想着我拿到的这组数据是什么工种、我应该怎么给它分配构造函数，更不用手写无数个构造函数——Factory已经帮我们做完了一切，而我们只需要像以前一样**无脑传参**就可以了！

现在我们一起来总结一下什么是工厂模式：工厂模式其实就是**将创建对象的过程单独封装**。它很像我们去餐馆点菜：比如说点一份西红柿炒蛋，我们不用关心西红柿怎么切、怎么打鸡蛋这些菜品制作过程中的问题，我们只关心摆上桌那道菜。在工厂模式里，我传参这个过程就是点菜，工厂函数里面运转的逻辑就相当于炒菜的厨师和上桌的服务员做掉的那部分工作——这部分工作我们同样不用关心，我们只要能拿到工厂交付给我们的实例结果就行了。

总结一下：工厂模式的目的，就是为了实现**无脑传参**，就是为了爽！

小结

工厂模式的简单之处，在于它的概念相对好理解：将创建对象的过程单独封装，这样的操作就是工厂模式。同时它的应用场景也非常容易识别：有构造函数的地方，我们就应该想到简单工厂；在写了大量构造函数、调用了大量的 new、自觉非常不爽的情况下，我们就应该思考是不是可以掏出工厂模式重构我们的代码了。

但工厂模式可不止这一种表达形式。本节我们可以看到，构造器解决的是多个对象实例的问题，简单工厂解决的是多个类的问题。那么当复杂度从多个类共存上升到多个工厂共存时又该怎么处理呢？在下个小节，我们一起来看看这个问题。