抽象工厂这块知识，对入行以来一直写纯 JavaScript 的同学可能不太友好——因为抽象工厂在很长一段时间里，都被认为是 Java/C++ 这类语言的专利。

Java/C++ 的特性是什么？它们是**强类型的静态语言**。用这些语言创建对象时，我们需要时刻关注类型之间的解耦，以便该对象日后可以表现出多态性。但 JavaScript，作为一种弱类型的语言，它具有天然的多态性，好像压根不需要考虑类型耦合问题。而目前的 JavaScript 语法里，也确实不支持抽象类的直接实现，我们只能凭借模拟去还原抽象类。因此有一种言论认为，对于前端来说，抽象工厂就是**鸡肋**。

抽象工厂模式的学习价值、以及为何被布局在小册 No.2 这个位置背后的思量，我会在文末给大家一五一十地捋清楚。但现在，各位先答应我，**不要跳读**，不要看见“抽象”俩字儿就跑 —— 鸡肋不鸡肋，学明白了才有发言权。

一个不简单的简单工厂引发的命案

在实际的业务中，我们往往面对的复杂度并非数个类、一个工厂可以解决，而是需要动用多个工厂。

我们继续看上个小节举出的例子，简单工厂函数最后长这样：

```
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)
}
```

乍一看没什么问题，但是经不起推敲呀。首先映入眼帘的 Bug，是我们把 Boss 这个角色和普通员工塞进了一个工厂。大家知道，Boss 和基层员工在职能上差别还是挺大的，具体在员工系统里怎么表现呢？首先他的权限就跟咱们不一样。有一些系统，比如员工绩效评估的打分入口，就只有 Boss 点得进去，对不对？除此之外还有许多操作，是只有管理层可以执行的，因此我们需要对这个群体的对象进行单独的逻辑处理。

怎么办？去修改 Factory 的函数体、增加管理层相关的判断和处理逻辑吗？单从功能实现上来说，没问题。但这么做其实是在挖坑——因为公司不仅仅只有这两类人，除此之外还有外包同学、还有保安，他们的权限、职能都存在着质的差别。如果延续这个思路，每考虑到一个新的员工群体，就回去修改一次 Factory 的函数体，这样做糟糕透了——首先，是**Factory会变得异常庞大**，庞大到你每次添加的时候都不敢下手，生怕自己万一写出一个Bug，就会导致整个Factory的崩坏，进而摧毁整个系统；其次，**你坑死了你的队友**：Factory 的逻辑过于繁杂和混乱，没人敢维护它；最后，你还**连带坑了隔壁的测试同学**：你每次新加一个工种，他都不得不对整个Factory 的逻辑进行回归——谁让你的改变是在 Factory 内部原地发生的呢！这一切悲剧的根源只有一个——**没有遵守开放封闭原则**。

我们再复习一下开放封闭原则的内容：对拓展开放，对修改封闭。说得更准确点，**软件实体（类、模块、函数）可以扩展，但是不可修改**。楼上这波操作错就错在我们不是在拓展，而是在疯狂地修改。

抽象工厂模式

上面这段可能仍有部分同学觉得抽象，也没关系。这里咱们先不急着理解透彻这个干巴巴的概念，先来看这么一个示例：

大家知道一部智能手机的基本组成是操作系统（Operating System，我们下面缩写作 OS）和硬件（HardWare）组成。所以说如果我要开一个山寨手机工厂，那我这个工厂里必须是既准备好了操作系统，也准备好了硬件，才能实现手机的**量产**。考虑到操作系统和硬件这两样东西背后也存在不同的厂商，而我现在**并不知道我下一个生产线到底具体想生产一台什么样的手机**，我只知道手机必须有这两部分组成，所以我先来一个抽象类来**约定住这台手机的基本组成**：

```
class MobilePhoneFactory {
    // 提供操作系统的接口
    createOS(){
        throw new Error("抽象工厂方法不允许直接调用，你需要将我重写！");
    }
    // 提供硬件的接口
    createHardWare(){
        throw new Error("抽象工厂方法不允许直接调用，你需要将我重写！");
    }
}
```

楼上这个类，除了约定手机流水线的通用能力之外，啥也不干。如果你尝试让它干点啥，比如 new 一个 `MobilePhoneFactory` 实例，并尝试调用它的实例方法。它还会给你报错，提醒你“我不是让你拿去new一个实例的，我就是个定规矩的”。在抽象工厂模式里，楼上这个类就是我们食物链顶端最大的 `Boss--AbstractFactory` （抽象工厂）。

抽象工厂不干活，具体工厂（ConcreteFactory）来干活！当我们明确了生产方案，明确某一条手机生产流水线具体要生产什么样的手机了之后，就可以化抽象为具体，比如我现在想要一个专门生产 Android 系统 + 高通硬件的手机的生产线，我给这类手机型号起名叫 FakeStar，那我就可以为 FakeStar 定制一个具体工厂：

```
// 具体工厂继承自抽象工厂
class FakeStarFactory extends MobilePhoneFactory {
    createOS() {
        // 提供安卓系统实例
        return new AndroidOS()
    }
    createHardWare() {
        // 提供高通硬件实例
        return new QualcommHardWare()
    }
}
```

这里我们在提供安卓系统的时候，调用了两个构造函数：AndroidOS 和 QualcommHardWare，它们分别用于生成具体的操作系统和硬件实例。像这种被我们拿来用于 new 出具体对象的类，叫做具体产品类（ConcreteProduct）。具体产品类往往不会孤立存在，不同的具体产品类往往有着共同的功能，比如安卓系统类和苹果系统类，它们都是操作系统，都有着可以**操控手机硬件系统**这样一个最基本的功能。因此我们可以用一个**抽象产品（AbstractProduct）类**来声明这一类产品应该具有的基本功能（众：什么抽象产品？？？要这些玩意儿干啥？老夫写代码就是一把梭，为啥不让我老老实实一个一个写具体类？？？大家稍安勿躁，先把例子看完，下文会有解释）

```
// 定义操作系统这类产品的抽象产品类
class OS {
    controlHardWare() {
        throw new Error('抽象产品方法不允许直接调用，你需要将我重写！');
```

```
    }
}

// 定义具体操作系统的具体产品类
class AndroidOS extends OS {
    controlHardWare() {
        console.log('我会用安卓的方式去操作硬件')
    }
}

class AppleOS extends OS {
    controlHardWare() {
        console.log('我会用🤚的方式去操作硬件')
    }
}
...
```

硬件类产品同理：

```
// 定义手机硬件这类产品的抽象产品类
class HardWare {
    // 手机硬件的共性方法，这里提取了“根据命令运转”这个共性
    operateByOrder() {
        throw new Error('抽象产品方法不允许直接调用，你需要将我重写！');
    }
}

// 定义具体硬件的具体产品类
class QualcommHardWare extends HardWare {
    operateByOrder() {
        console.log('我会用高通的方式去运转')
    }
}

class MiWare extends HardWare {
    operateByOrder() {
        console.log('我会用小米的方式去运转')
    }
}
...
```

好了，如此一来，当我们需要生产一台FakeStar手机时，我们只需要这样做：

```
// 这是我的手机
const myPhone = new FakeStarFactory()
// 让它拥有操作系统
const myOS = myPhone.createOS()
// 让它拥有硬件
const myHardWare = myPhone.createHardWare()
// 启动操作系统(输出‘我会用安卓的方式去操作硬件’)
myOS.controlHardWare()
// 唤醒硬件(输出‘我会用高通的方式去运转’)
myHardWare.operateByOrder()
```

关键的时刻来了——假如有一天，FakeStar过气了，我们需要产出一款新机投入市场，这时候怎么办？我们是不是**不需要对抽象工厂MobilePhoneFactory做任何修改**，只需要拓展它的种类：

```
class newStarFactory extends MobilePhoneFactory {
    createOS() {
        // 操作系统实现代码
    }
    createHardWare() {
        // 硬件实现代码
    }
}
```

这么个操作，**对原有的系统不会造成任何潜在影响** 所谓的“对拓展开放，对修改封闭”就这么圆满实现了。前面我们之所以要实现**抽象产品类**，也是同样的道理。

总结

大家现在回头对比一下抽象工厂和简单工厂的思路，思考一下：它们之间有哪些异同？

它们的共同点，在于都**尝试去分离一个系统中变与不变的部分。**它们的不同在于**场景的复杂度。**在简单工厂的使用场景里，处理的对象是类，并且是一些非常好对付的类——它们的共性容易抽离，同时因为逻辑本身比较简单，故而不苛求代码可扩展性。抽象工厂本质上处理的其实也是类，但是是一帮非常棘手、繁杂的类，这些类中不仅能划分出门派，还能划分出等级，同时存在着千变万化的扩展可能性——这使得我们必须对**共性**作更特别的处理、使用抽象类去降低扩展的成本，同时需要对类的性质作划分，于是有了这样的四个关键角色：

- **抽象工厂（抽象类，它不能被用于生成具体实例）：** 用于声明最终目标产品的共性。在一个系统里，抽象工厂可以有多个（大家可以想象我们的手机厂后来被一个更大的厂收购了，这个厂里除了手机抽象类，还有平板、游戏机抽象类等等），每一个抽象工厂对应的这一类的产品，被称为“产品族”。
- **具体工厂（用于生成产品族里的一个具体的产品）：** 继承自抽象工厂、实现了抽象工厂里声明的那些方法，用于创建具体的产品的类。
- **抽象产品（抽象类，它不能被用于生成具体实例）：** 上面我们看到，具体工厂里实现的接口，会依赖一些类，这些类对应到各种各样的具体的细粒度产品（比如操作系统、硬件等），这些具体产品类的共性各自抽离，便对应到了各自的抽象产品类。
- **具体产品（用于生成产品族里的一个具体的产品所依赖的更细粒度的产品）：** 比如我们上文中具体的一种操作系统、或具体的一种硬件等。

抽象工厂模式的定义，是**围绕一个超级工厂创建其他工厂。**本节内容对一些工作年限不多的同学来说可能不太友好，但抽象工厂目前来说在JS世界里也应用得并不广泛，所以大家不必拘泥于细节，只需留意以下三点：

1. 学会用 ES6 模拟 JAVA 中的抽象类；
2. 了解抽象工厂模式中四个角色的定位与作用；
3. 对“开放封闭原则”形成自己的理解，知道它好在哪，知道执行它的必要性。

如果能对这三点有所掌握，那么这一节的目的就达到了，最难搞、最难受的抽象工厂也就告一段落了。

最后，再跟大家谈谈学习

现在我们回到开篇抛出的那个问题——抽象工厂对于各位而言的价值是什么？这么一个看似鸡肋、其实也确实不怎么常用的一个设计模式，凭什么值得我们花这么大力气去理解它？原因有三：

其一： 开篇我们说过，**前端工程师首先是软件工程师。**只会写 JavaScript、只理解 JavaScript、只通过 JavaScript 去理解软件世界，是一件可怕的事情，它会窄化你的技术视野——因为 JavaScript 只是编程语言中的一个分支，准确地说，它是一个后辈。虽说它确实很流行，但它还不够强大（正是因为不够强大，所以在演化发展的过程中必然需要借鉴其它优秀语言的优秀特性，也会渐渐遇到其它语言的应用场景，不信大家看看 ES6789 都做了什么，再看看遍地开花的 TypeScript）。

但写这本小册并不是为了把大家指去学 Java/C++，而是为了以最小的时间成本帮大家去理解设计模式的套路和原则。比起要求大家为了这个设计模式去理解强类型语言、去理解强类型语言里的应用场景，我更希望能在这儿用 JavaScript 把这个东西给说清楚，把那些关键的设计模式概念在这儿给大家引出来——哪怕你当下用到它的场景还不是那么多（相信以当下前端语言和前端应用的发展速度和发展趋势来看，它会有用的：））。

其二： 在大家今后的职业生涯里，可能会不止一次地遇到服务端/客户端出身、或者单纯对受试者知识广度有疯狂执念的各种不同背景不同脑回路的面试官。在他们的世界里，不知道抽象工厂就像不知道 `this` 一样恐怖：）。所以，**要学。**

其三： 也是最重要的一点。前面我们说过，设计模式的“术”说到底是在佐证它的“道”。充分理解了设计原则后，设计模式纵有 1w 种也难不倒大家。**抽象工厂是佐证“开放封闭原则”的良好素材**，通过本节的学习，相信大家会对这个抽象的概念有更加具体和感性的认知。在后面的章节中，“开放封闭”作为各位的老朋友，会被反复提及。有了本节的平稳过渡，相信大家在后续的学习中可以真正做到心中有数、游刃有余。

说了这么多，无非是想传达给大家一个学习态度：**不要小看那些看似“无用”的知识。**

技术，尤其是前端技术，它的更新迭代速度是非常快的。仅仅因为“这个技术点我现在用不到”而推开摆在眼前的知识，是一种非常糟糕的学习方法——它会极大地限制你的能力和你职业生涯的可能性。举个例子，React 新版本推出的 Fiber 架构现在很火，很多同学认为这是个特别新潮的玩意儿——它新吗？新个屁！作为一种架构模式，它在软件领域早就有过不同姿势的生产实践了，React 并不是 Fiber 的发明者，而是 Fiber 的使用者和受益者。

同理，包括 ES2015 刚出来的时候，有同学说这个也没见过、那个也要重新学，累死了累死了，学不动了想转行...哎，其实它们都是软件世界里存在了很久很久的模式和知识点啊同学们。试想如果这份知识曾经摆在你面前的时候你没有拒绝它，此刻你的学习成本又该低了多少呢？

设计模式之外的东西，我们点到即止，剩下的就看大家的悟性和造化了。 接下来，我们一起看点更好玩的东西~

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）