第三篇 设计篇

读者叙

在原始社会，人类作为灵长类生物天生具有更强的思考能力，开始学会钻木取火，开始制造狩猎的工具。人类通过不断思考、探索、实验，对各项工具开始具备了外形、材质的形象定义，也许这就是人类最初对工具的设计过程。

现在的世界，有了各种交通工具、电灯、网络。在很多人心目中这些东西是被制造出来的，但在制造之前，是需要根据社会的需求以及物理学所能支撑的技术范畴综合考量后，加以推敲、思考然后进行工艺、外形的设计，才会有制造的成果。

写程序依然如此，我们今天只是换了一种方式来制造世界上更多的虚拟内容，因此我认为设计的思路对每一位程序员来讲都是十分重要的，在本篇，将与大家交流一些设计的心得和思路，包含 Java 代码的设计、软件架构设计、产品设计、交互设计方面。

作为一个程序员需要了解的设计吗？

设计的思路让我们站在换一个角度看待技术和业务问题，或者说将这些问题的整体看得更加清楚，可以让我们逐步具备从上往下看待问题的思路。另外，通过设计，可以让一个团队对共同的目标达成共识，一个团队做的事情才能像一个人做的事情一样，有机会去达到在效率上的最大化。

设计篇会有那些设计知识？

程序员的设计可以说无处不在，从上层架构到最基础的代码结构设计，甚至于对复杂的代码还会继续分层设计。总之，设计的基本原则是让每一个层次的逻辑尽量简单清晰。按照宏观上分层的结构，本篇将分 3 个部分：

第 1 部分讲解代码及软件结构设计希望更加贴近程序员本身。

第 2 部分探讨产品设计的一些理念和思考，但我们不细谈产品经理应该做的事情。

第 3 部分探讨一些交互设计理念和思考，同样我们不详细探讨交互设计师做的工作。

Java 设计是谈设计模式吗？

胖哥也不太懂什么设计模式，^_^，我只是根据在工作中的一些经历，与大家一起探讨一些代码案例如何设计会更好一些，而且，你可能会有更好的方案哦！

第 1 章 代码及软件架构设计

章节简介

本章将从接口和抽象类开始与大家探讨软件设计的基础理论的一些细节，通过一些简单案例，探索于设计的意义和原理。

后续将以开源框架拓展、模块化、组件化、代码分层几个角度阐述框架搭建的意义和基础方法。

最后我们将阐述产品业务发展过程中必然面临的重构、拆分，其时机与方法基础。

1.1 接口及抽象类

接口和抽象类是耳熟能详的 Java 专业术语：

- Java WEB 代码会惯性去写接口和实现类。
- 大学的课本里面会反复提到。
- Java 提到多态、设计模式等会随处可见。
- 面试官通常会问这样的问题。

许多程序员这个时候会做一个“乖孩子”去遵循非常多在其中的许多条款，也会用“规范”二字去让更多人无条件使用。当然按照小胖的习惯，这本书里面肯定是非常规的手段来与你探讨这些事情，希望大家通过这些内容开启对有封装、继承、多态的原理的理解。

1.1.1 接口无处不在

接口的思想并非 Java 独有，Java 语言本身也借鉴于现实世界成熟思路进行模拟。现实生活中体现得较为明显的是制造业。例如生活中所熟知的一些事物：计算机、汽车、商品房建造等等。下面给大家逐步举几个例子来探讨一下：

计算机：

计算机的内部几大核心部件：**主板、CPU、内存、显卡**等，这些部件通常由不同的生产厂家来制造，但是又能粘合在一起，其原因是插口都有标准。随着业务的快速发展，这样做的好处被推广，逐步成为一种社会普遍的规范，每一个想参与这个行业的公司，都会按照指

定的规范去生产具体的部件，组装过程就会比较顺利了，而且这些公司之间也不需要太多会议上的沟通。

在现实社会中，类似的思路都是相通的，例如前文中提到的汽车，现在国内的许多“合资车”厂商根据进口发动机规格等信息开始批量生产其它部件，发动机进口后根据相应接口标准对接上即可。

管道：

现在我们从另一个角度来谈接口，联想一下生活中的所用到的水管，从水厂开始净化水后，经过大小管道层层分流达到各个家庭，大小管道的设计提前有了规格设计，那么在部署和施工过程中就会有精准地调度过程。不同规格的管道之间的连接需要有一些转换的过程，同规格的管道在拐弯处也需要转换地过程。

从这个角度，可以理解管道之间的适配是通过一种转换处理的方式来完成。

从现实世界回到程序中：

Java 语言中的接口利用是非常多的，只要你用心去看工作中的代码，就应该会发现很多设计上的技巧。我并没有大家的业务代码，因此无法与你的工作绝对相关，将会以一些较为熟知的通用组件来给大家讲解，希望能够帮助你去挖掘业务代码中的设计思路。

JDBC 标准的定义就是一种接口，语言的作者首先会了解到数据库的常用操作，因此定义了一套 java.sql.*的接口出来，这样可以让 Java 的程序设计者拿到驱动后，就会比较简单地使用 SQL 访问数据库，而不需要关心通信细节。但是语言的作者并不想涉及到各种数据库的通信细节，这样它会花费大量的时间和精力，因此它将细节由各个厂家去实现，各个厂家或第三方去保证稳定性。

在这个目录下还有连接池 DataSource 的接口定义，同样的，DataSource 的实现者、驱动研发者、基于连接池的框架（如 ibatis）、业务程序员就可以并行发展。其原因就是它们之间的衔接通过一种接口的方式来完成，而不会相互依赖导致串行化发展。

这样的案例也不仅仅局限于 Java 语言本身，例如 Spring 的事务管理器为了面对不同的事务场景，抽象了 PlatformTransactionManager 相关的方法，具体的 commit、rollback 等动作由具体的事务组件来完成，而 Spring 在这一块将更加专注于事务对于业务代码的切入、封装处理。

Java 中用到的例子还很多

- WEB 容器中划分的 Filter、Servlet、Listener 标准。
- Hadoop 中定义的 Map 和 Reduce 标准。

- Spring 的拦截器（interceptor）定义。
- …….

回到现实，程序员朋友们，我们有些时候跳出自己代码逻辑的视野，就会看到更高的维度的事情，而下面的事情会更加抽象，事情的全盘也会因此更加清晰明朗一些。正所谓“站得越高看得越远”。

1.1.2 抽象类是接口的“好基友”

上面提到接口，自然少不了它的一个好基友“抽象类”，因为经常会有一些面试官问“抽象类和接口的区别是什么”这样的问题。

从本质上来讲，这两者还谈不上啥区别。从语法上看，抽象类比起接口的区别在于可以编写一些实现方法，介于接口和实体类之间的一种类。这样的说法你难道不觉得这也太表面了吗？

在上一节描述的接口实例中，如果我们去读一读其中的源码会发现实现类中大多都会出现抽象类，这些抽象类通常扮演的角色是接口方法的通用逻辑实现的内容，由于它是抽象多个子类的通用逻辑，那么子类必然就有一些特殊的变化。

例如：一台固定品牌的汽车生产过程中会有一个大致一样的流程，但是当这台汽车到了控制台配置、颜色、发动机排量等信息的时候，会不太一样，那么工艺流程中就类似于公共父类定义了模板化的工艺步骤，或者说主体步骤是早已固定好的，但是工艺步骤中并不知道喷漆的颜色、发动机的排量、各项其它配置，将在对应的步骤中根据车辆需求决定，然后到了具体的步骤后，由根据需求来决定需要选用的“子组件”来完成对应的车辆配置工作。

在设计模式中通常叫做“模板方法”，从程序设计的角度来讲，它有这样一些好处：

- 抽象类中可以实现一些通用的组件方法，让代码变少维护成本低。--继承也可实现
- 抽象类更为清晰明确地定义该接口所需要总体逻辑步骤，公共的逻辑步骤由父类完成，非公共的组件将“**明确要求**”子类来完成。---明确要求这一点普通继承做不到。
- 子类的方法单纯清晰，要实现的方法已被明确定义（就先工艺流程中该步骤只做喷漆操作），整个代码成组件思路，这样一来，子类的方法在拓展的时候代码不至于与各种不同的业务绕在一起，导致可读性很低。

回到与接口的关系上来讲，抽象类更多是在实现层做了一些实体类无法完成的“明确定义子类工艺”的动作，和接口的关系更多是为了达到业务目的配合使用，谈不上什么区别可

言，如果你不介意，也可以用实体类来完成，未实现的方法都用空 Body 来完成，子类 Ovrride 该方法也一样可以实现。但这样的操作大家可以对比一下，写代码的过程通常不会是设计的思路，子类继承父类的方法也会很随机，最后会感觉子类不知道是什么的定义，可能是父亲类的一个随机补充。这样反过来我们才能看出抽象类的实用价值。

1.1.3 设计者角度思考问题

通过前两个小节的分析，抽象类和接口更多是为设计者思路提供了实现上的便利，即：设计者会从整体结构上把握体系，在乎主干而并非肉体本身。在主体结构存在的前提下，从上向下去考虑问题，当前这一层只关注这个抽象层次的思维方式并编写相应的逻辑，相关的抽象方法会由接口调用来完成。

这个道理有点像工厂的车间主任关心车间工人的头、技术负责人、机器整体情况，技术头会关注自己所涉及技术高级问题和带徒弟做事情，徒弟们通常会关注自己做事情的每一个细节。

在设计代码前，如果根据深入了解业务背景，进而不断细化，就逐步能得到许多业务对象、对象依赖、调用关系、动作、步骤、流程等元素，从而可以在代码设计之初将许多框架体系想得比较清楚。

在这个过程可以用一些伪代码来提升设计上的可读性（关键是要将问题讲清楚），也不用拘泥于一定要在整个产品或项目的前期才这样做---这样做似乎是给客户或老板看的，每一个相对复杂的模块都可以尝试这样去做，它可以将许多问题提前得更清楚一点---在很多时候它能指导很多大体方向的可行性及难度。

Java 所提供的设计思路天生似乎就是这样的，这使得相对之间的抽象层次很简单和清晰，大家分工协作但目标一致，使得每一个人的思路非常的清晰从而效率较高。这比较著名的就是 Java 的分层开发，关于这方面的内容，将在本章 1.6 小节与大家有更多的沟通。

既然是一种设计者的思路，因此希望每一位开发者在使用的时候理解它的存在价值，这样才有机会在较为合适的时机选择适当的设计思路。这样的分层体系会让很多非 Java 的开发者非常的困惑，尤其是阅读很多源代码时会感觉 Java 将许多简单的问题复杂化了。

从上述的结论来看，在阅读 Java 代码的时候如果懂设计思路的基础上先关注整体架构逐层阅读每一层次的抽象意义会容易多了，反过来看我们自己，如果编写的代码每一层的抽象意义不明确，通常来讲我们也就没有太明白分层的价值。

上面我们人云亦云，不知道读者朋友有没有一些感觉，如果还没有，来看看我们下面的一些小例子，再反过来看上面的概括。

1.2 抽象代码的小例子

在本节，将从一个非常简单的例子开始，逐步提升一些相对复杂的例子，希望大家能从代码中感受一些代码抽象的感觉，真正体会到一些与没有设计的代码之间的区别在那里。实际的案例也与真实的场景差距很大，我们在这里更多的是探讨一个思路和感觉。

1.2.1 星座代码的简单改变

下面这个星座的例子是一个小伙伴写的小案例，主要是需要写了一个关于根据“月份+日期”返回“星座”的简单函数，当时这位小伙伴提出的问题还并不是设计上的问题，而是：**“无法得到星座的返回值”**，这显然是代码本身有问题，下面来看看代码，**你能一眼看出代码那里有问题吗？**

代码清单 1-1 有问题的判定星座原始代码

```
public String getConstellationValue(Integer month, Integer day) {
    String Constellation = null;
    if ((month == 12 && day >= 22) || (month == 1 && day <= 20)) {
        Constellation = Constellation.Capricorn.getValue();
    } else if ((month == 1 && day >= 21) && (month == 2 && day <= 19)) {
        Constellation = Constellation.Aquarius.getValue();
    } else if ((month == 2 && day >= 20) && (month == 3 && day <= 20)) {
        Constellation = Constellation.Pisces.getValue();
    } else if ((month == 3 && day >= 21) && (month == 4 && day <= 20)) {
        Constellation = Constellation.Aries.getValue();
    } else if ((month == 4 && day >= 21) && (month == 5 && day <= 21)) {
        Constellation = Constellation.Taurus.getValue();
    } else if ((month == 5 && day >= 22) && (month == 6 && day <= 21)) {
        Constellation = Constellation.Gemini.getValue();
    } else if ((month == 6 && day >= 22) && (month == 7 && day <= 22)) {
        Constellation = Constellation.Cancer.getValue();
    } else if ((month == 7 && day >= 23) && (month == 8 && day <= 22)) {
        Constellation = Constellation.Leo.getValue();
    } else if ((month == 8 && day >= 23) && (month == 9 && day <= 22)) {
        Constellation = Constellation.Virgo.getValue();
    } else if ((month == 9 && day >= 23) && (month == 10 && day <= 23)) {
        Constellation = Constellation.Libra.getValue();
```

```
    } else if ((month == 10 && day >= 24) && (month == 11 && day <= 22)) {
        Constellation = Constellation.Scorpio.getValue();
    } else if ((month == 11 && day >= 23) && (month == 12 && day <= 21)) {
        Constellation = Constellation.Sagittarius.getValue();
    }
    return Constellation;
}
```

这段代码看似非常简单，但是这位小伙伴问这个问题的时候，大多数人都没有看出啥问题，因为大家心里都觉得是不是漏掉了某个时间段没写，而忽略了一些低级问题。

但是问题的根本原因往往很“菜”，就是从第一个 else if 逻辑开始，日期判定逻辑之间的“||”写成了“&&”，而且还是千篇一律的写法。这也许而是我们写代码最爱干的事情吧：“复制粘贴”。

这是一个比较低级的问题，但是通常会让我们很麻木，而最直接的解法是将这个符号换成“||”就解决问题了，但是如果这样的代码只有一两个逻辑错误其实是很难发现的，我们是否可以重新设计一下这段代码呢？设计的目的就是让我们通过一些方法让逻辑变得简单，让那些能“难倒高手的低级问题”尽量被扼杀在摇篮里，答案是肯定的，且听分解：

- 12 个星座，每年月份也是 12 个，每 1 个星座时间正好连续地横跨 2 个月
- 反过来看，每 1 个月也会出现 2 个星座，那么根据输入的月份就能确定到 2 个星座，且两个星座的顺序是可人为确立的。
- 每一个星座有结束日期，那么根据条件 2+输入日期就能决定 2 个星座中的具体那一个了。

既然输入日期与星座结束日期有关系，且星座是固定不变的，那么基于星座这个我们做一个基于结束日期的星座表格，然后基于这些枚举对象，建立月份维度的星座表格 *MONTH_ARRAY*（这个建立表格的过程，就是一个比与、或、非简单得多的编辑逻辑），**通过这个表格，就可以根据月份+日期得到其在表格中的坐标：**

代码清单 1-2 修改后的星座代码

```
public enum Constellation {
    //入口参数为星座结束的日期，不带月份
    Capricorn(20),
    Aquarius(19),
    Pisces(20),
    Aries(20),
    Taurus(21),
    Gemini(21),
```

```
    Cancer(22),
    Leo(22),
    Virgo(22),
    Libra(23),
    Scorpio(22),
    Sagittarius(21);
    //记录月份中的两个星座的"月中日"的分割点，也就是记录每个星座的"月中日"
    private final int endDay;

    private final static Constellation[][]MONTH_ARRAY = {
        {Capricorn , Aquarius} , //1月
        {Aquarius , Pisces} , //2月
        {Pisces , Aries} , //3月
        {Aries , Taurus} , //4月
        {Taurus , Gemini} , //5月
        {Gemini , Cancer} , //6月
        {Cancer , Leo} , //7月
        {Leo , Virgo} , //8月
        {Virgo , Libra} , //9月
        {Libra , Scorpio} , //10月
        {Scorpio , Sagittarius} , //11月
        {Sagittarius , Capricorn} , //12月
    };

    private Constellation(int endDay) {
        this.endDay = endDay;
    }

    public static Constellation fromValue(int month , int day) {
        Constellation[]tempArray = MONTH_ARRAY[month - 1];
        if(tempArray == null) return null;
        return day <= tempArray[0].endDay ? tempArray[0] : tempArray[1];
    }
}
```

代码分析：

这段代码似乎为了解决一个简单问题变得复杂了？小胖可不这么认为，有些时候叫前人种树后人乘凉，我们将逻辑封装后，变成了简单的表格填写和坐标定位，逻辑只有*fromValue(int month , int day)*里面的一点点，从设计来降低逻辑成本，同时也降低了低级问题出现的概率。

1.2.2 解压缩处理逻辑的封装

本节，我们以一个解压缩处理的业务为例，逐步引导简单的封装思路，请看业务描述：

- 每次输入一个 ZIP 压缩包文件。
- 压缩包内部有多层文件和文件夹，程序负责扫描其中所有的文件。
- 文件类型有 txt、jpg、csv 等格式，程序根据不同的文件类型做不同的处理。
- jpg 返回文件大小、文件名、文件的二进制内容。
- txt 返回文件大小、文件名、文件的字符串内容。
- csv 返回文件大小、文件名、文件字符串内容、文件行数、列数量。

按照常规的思路，我们通常会写下面这样一段代码：

代码清单 1-3 伪代码模拟

```
ZipFile zipFile = new ZipFile("xxx.zip");
try {
   Enumeration zipEntries = zipFile.getEntries();
   while(zipEntries.hasMoreElements()) {
       ZipEntry zipEntry = (ZipEntry)zipEntries.nextElement();
       String name = zipEntry.getName();
       if(zipEntry.isDirectory()) {
          ..目录递归向下处理
       }else if(endsWith(".txt") {
          …文本文件处理逻辑…
          …这里可能会写比较长的逻辑…
       }else if(…) {
          …其它文件处理逻辑…
          …这里可能会写比较长的逻辑…
       }
   }
} finally {
   zipFile.close();
}
```

这段代码的核心逻辑并没有太大的问题，但是这样会使得我们在写代码的时候，将细节与整体逻辑放在一起来编写，导致的结果就是代码逻辑非常复杂，代码维护起来会非常的麻烦，到一定的复杂度的时候可能没有太多人敢维护这个代码。

按照常理，抽象的第一步是**把不同文件的处理单独用一个方法来处理**，做这件事情也是没有任何语言上的区别的，因为逻辑上至少拆分开了，各个逻辑看起来就会变得简单了。但接下来我们会面临两个问题：

- 各个方法随着业务的发展可能会进一步变得更加复杂，因此还会拆分出子方法，这个类会因此变得混乱（刚才是方法比较混乱）。
- 这些方法的抽象并没有明确的定义入参和返回，也就是其作用意图并不是那么明确，更像是随机性的剥离，如果要后续清晰的维护需要非常依赖程序员的水平的认知。

此时我们认为可以将代码剥离为两层来看，父类定义处理文件的方法为抽象方法，定义统一的入口参数和返回值：

- **入口参数：**ZipEntry
- **返回值公共父类：**BaseChildFileDO

  int fileType

  String fileName

  long fileLength
- **返回 jpg 文件：**JPGChildFileDO extends BaseChildFileDO

  byte []content
- **返回文本文件：**TXTChildFileDO extends BaseChildFileDO

  String content
- **返回 CSV 文件：**CSVChildFileDO extends TXTChildFileDO

  int rows

  into columns

最终，我们将父亲类的代码进行封装如下：

代码清单 1-4 **伪代码**模拟

```
public abstract class ChildFileProcessor {//具体实现由子类来完成
  protected  abcstract BaseChildFileDO processChildFile(ZipEntry zipEntry);
}

public static ChildFileProcessor getChildFileProcessorByFileName(String fileName) {
   …可以用反射、Map 映射、枚举映射、也可以直接判定返回对应文件的处理实体类
}

public  void  processZipChildFiles(String fileName) {
    ZipFile zipFile = new ZipFile("xxx.zip");
    try {
       Enumeration zipEntries = zipFile.getEntries();
       while(zipEntries.hasMoreElements()) {
```

```
            ZipEntry zipEntry = (ZipEntry)zipEntries.nextElement();
            if(zipEntry.isDirectory()) {
               ..目录递归向下处理
            }else {
               String name = zipEntry.getName();
               BaseChildFileDO xxx = getChildFileProcessorByFileName(name).processChildFile(zipEntry);
               //进一步如何处理结果，可以再从另一个角度抽象
            }
         }
      } finally {
         zipFile.close();
      }
   }
```

这样拆分是不是有点过火呢？

如果单纯按照本节的案例，是没有必要做这么复杂的拆分的，这样拆分会增加很多子类，以后维护起来其实还会变得更加麻烦一些，这样会导致我们不会不会更加专注于业务发展本身，那为什么小胖还说要拆分呢？

基本的方法拆分是必须要做的，但是第 2 步做不做其实是看这个业务是不是足够大，如果每一类文件后续的处理都比较复杂，或者未来很可能会变得比较复杂，那么一个方法是无法承载的，可能对于一类文件的处理，继续再拆分出几个乃至十几个方法，那么将这些不同文件类型的处理方法都放在一起都比较难以管理了，这个时候拆分出子类完成是很有必要的，随着业务的发展子类还可能会继续拆分出多个模型出来。这正好应证一句话“合久必分、分久必合”，分分合合的目的是便于整体上统一维护和管理，保持高速的业务发展形态。

所以，拆分这个动作将是一个无底线的，是否需要根据业务场景和理解来决定，达到业务与技术发展的平衡，能够有理有据推导结论，切忌以自己的业务场景来要求别人。将这个事情反推到许多开源框架和公司级框架上，它们更多是帮你做了很多抽象，根据通用的业务特征和公司项目的整体业务特征进行抽象。有的公司甚至于将一些框架限制得很死，可能会导致你感觉业务很痛苦地在发展。软件框架只有灵活与拓展性特征，没有完美的软件框架。

1.2.3 登陆体系对接案例

如果上面两个案例属于瞎 YY 的话，现在给大家举一个大家都熟知的例子：登录。相信绝大部分 WEB 系统都会有登录这个环节，这有什么好讲的呢？且看业务描述：

- 该产品有 2 层登录体系；分别是第三方登录和本地二次登录。

- 该产品与第三方登录主要确定用户信息，但是为了产品的通用性，所以要灵活适应第三方登录。
- 二次登录是在第三方登录用户信息获取后进行资源验证的登录。

整体会话策略如下图所示：

图 1-1

数据结构的定义：

- 由于产品所管理的资源是明确的，所以资源类会话的内容是可以明确的。
- 用户会话虽然不明确内容是什么，但是可以事先定义一个用户应当具备的内容是什么：唯一标识符是必须的，其次 email、电话号码、角色等信息，可能有也可能没有，但我们可以事先定义，唯一标识符将是产品对外部对接的底线，也就是边界。

登录逻辑的定义：

- 二级会话的登录逻辑是可以事先编写的，因为数据结构和验证逻辑基本可以确定。所以在设计上，并不是我们要探讨的重点。
- 用户会话无法确定，可以事先用简单内部模拟用户来对接。
- 为了保持外部对接可插拔，我们希望动态指定登录逻辑代码执行。在本节中选择比较原始的 Filter 来完成，也就是动态指定 Filter。
- 思路上讲，可以通过 maven 打包动态指定 Filter，但是 Filter 通常是在 web.xml 下的，要动态调整要相对麻烦一些，因此可以对 Filter 进行一层封装。

Filter 封装的思路：

- 前置条件：使用 Spring MVC 作为 WEB 框架。
- 所有的实际对接的 Filter 实体不配置在 web.xml 里面，并且在类的定义上添加以下 2 个注解：

```
@Lazy
@Component(value = "XXXFilter")
```

- 不同环境的配置文件指定 XXXFilter 的名称，然后需要一个中转 Fitler 配置在 web.xml 当中。
- 中转 Filter 需要继承 DelegatingFilterProxy 类，重写方法：initFilterBean()，这样就可以拦截到启动 Filter 的过程，在这里只需要调用 super.setTargetBeanName('xxxFilter'); 就可以让 Spring 加载不同的 Filter 来完成。
- 但是这个 XXXFilter 在配置文件里面，所以要读到这个配置文件，要么我们自己写代码去读取配置文件，要么掌握 Spring 的加载原理接手这个参数。后者需要定义一个类继承 PropertyPlaceholderConfigurer 类，该初始化方法会传入 Properties 参数就是配置文件的内容了。简化代码如下：

代码清单 1-5

```
public class XXXPropertyPlaceholderConfigurer extends PropertyPlaceholderConfigurer {

    private Properties properties;

    @Override
    protected void processProperties(ConfigurableListableBeanFactory beanFactoryToProcess,
                                     Properties props)
            throws BeansException {
        properties = props;
        super.processProperties(beanFactoryToProcess , props);
    }

    public Properties getProperties() {
        return properties;
    }

    public void setProperties(Properties properties) {
        this.properties = properties;
    }

    public String getTargetBeanName() {
        return properties.getProperty("xxx_filter_bean_name");//属性文件的 key
    }
}

//动态 Filter 的代码大致为：
public class XXXDelegatingFilterProxy extends DelegatingFilterProxy {
```

```
    @Override
    public void initFilterBean() {
        XXXPropertyPlaceholderConfigurer propertyPlaceholderConfigurer
                        = SpringApplicationContext.getBeanByType(XXXPropertyPlaceholderConfigurer.class);
        String mainFilterTargetBeanName = propertyPlaceholderConfigurer.getTargetBeanName();
        if(StringUtils.isNotEmpty(mainFilterTargetBeanName)) {
            super.setTargetBeanName(mainFilterTargetBeanName.trim());
        }
    }
}
```

这段代码是一段比较初级的抽象，利用了 Spring 的框架结构，动态设置需要使用 Filter，按照这个方式，不同的第三方登录只需要按照字段映射到相应的用户唯一标识符、email 属性上就可以达到内部处理逻辑一致的目的。通过动态打包，可以做到不同用户登录体系的无缝插拔的处理。

但是新的问题来了：

- 产品随着业务的发展，第三方登录的接入的越来越多，各自有所不同。
- 发现第三方登录的逻辑基本是一致的，只是验证会话、Cookie、参数的内容不一样，虽然验证的细节有所不同，但是验证的整体逻辑步骤是一致的。
- 因此代码中出现很多类似的逻辑，甚至于是复制粘贴的逻辑。通过抽象也能发现即使不是复制粘贴的代码，最终要实现的主体逻辑步骤基本是一致的。

于是，我们想进一步抽象这个代码，也就是第三方登录的抽象类的核心逻辑步骤：

- 设置相关环境参数，决定后续的处理 ---子类实现
- 获取第三方登录标识符 ---子类通过 Cookie 或其它位置获取
- 获取第三方登录信息 ---子类调用对应的 API 或组件实现
- Cookie 与用户之间信息验证逻辑 ---抽象类实现
- 登录会话有效性验证 ---抽象类实现
- 第三方登录信息持久化 ---抽象类实现
- 会话失效跳转或 Ajax 处理逻辑 ---抽象类实现
- 非法登录处理返回逻辑 ---抽象类实现
- 成功登录或会话有效的处理逻辑 ---抽象类实现

按照上面的描述，这里面大多数步骤都可以由抽象类定义出来（顺序根据实际情况而定），而同时抽象类可以实现大部分的处理逻辑，只有具体会话标识符获取和第三方 API 验证才需

要子类来完成，这些也是抽象类所无法决定的事情。

这样一来，核心逻辑可以很直接地在抽象类中被描述和展现，而子类所需要实现的方法是提取对应第三方登录的标识符，调用第三方 API 验证的处理等相关的方法，因此**子类可以专注于某一类第三方验证完整过程**，整个类的用途也会比较明确，拓展某一类第三方验证的逻辑也很容易。

同时，这样的代码一旦梳理清楚后，处理逻辑有问题就在抽象类，第三方验证有问题就在具体的某一个子类里面，每一个类的改动都会相对清晰得多，不会相互干扰，就不像我们把不同层次的逻辑写到一块，改动的时候可能一动而引发全身。

1.2.4 小小总结

本小结是一些小模块的简单推导，不知道大家是否有一些代码设计上思路呢？例子不宜举太多，更多的案例和理解还是需要大家能够结合自己的代码和项目来进行推敲。

另外，上面的例子，大家也可以继续向前推敲，可能会遇到一些复杂的场景，例如 1.2.3 节经过两次推导后，可以很容易拓展第三方登录且代码冗余很少。但如果需要考虑同时在一个环境中兼容 2 种类型的第三方登录，且不同的登录来源处理逻辑会不同又该如何进一步抽象呢？

1.3 框架搭建

1.3.1 开源框架的扩展思路

开发一个软件产品，研发团队都会基于一系列的开源框架及组件架来完成，这些框架抽象了研发过程中诸多公共的元素，并定义好了接口或接入规范，让研发人员可以在一个体系下写代码，这个体系就像一个框子一样定义了整体代码的编写方式。

那么，将多个开源框架拼装起来就是一个软件架构了吗？显然不是，在很多时候，我们还需要框架本身和业务级别的扩展，这样让研发过程的效率得到提升，随着业务的变动不断提升框架的伸缩性来很好地支撑业务的快速发展。

在 1.2.3 节中，就提到了通过 Spring 的简单拓展和利用达到动态加载登录逻辑的目的，这算是一个简单的框架级的小拓展，下面我再举一些相对通用的案例：

- **前端交互**，前端交互中，有很多公共组件是开源界可以直接使用的，但如果需要在

效果上达到更好的业务效果，需要对开源控件进行源码修改或扩展。同时，前端也同样基于业务会封装许多通用 JS 函数，为产品中多个模块所复用，例如对 Ajax 调用后台服务的二次封装，可以定义请求中的：字符集、Method、返回类型解析等等，这样不用每一个 Ajax 调用都去关心这些内容了，另外，它还可以拓展出一层拦截器，在 Ajax 的请求前、返回正确数据、网络错误时，可以进行相应通用处理（部分前端框架本身提供了拦截器的功能也是类似的做法）。

- **WEB 输入输出，例如产品中需要封装统一 JSON 格式输出到页面，**假如在页面的数据输出外，还返回统一的信息：状态、Message、errorCode 等信息，如果我们去封装统一的 Response 组件的，就不需要每一段代码都要去封装这个格式了。
- **工作流，产品中出现大量的流程，**可以将流程中的关键性元素（环节、动作、路由等）进行抽象，让环节中的业务处理过程称为一个回调动作。当然现在有不少工作流本身就实现了类似的功能。
- **通用组件，假如产品中有大量的日期、字符串的“业务级”处理，**在开源的 commons 包里面是提供了一些列日期、字符串的处理方法，但是在具体的业务下它为够用，例如：代码中多个地方需要判定字符串是不是数字字符串；判定字符串是否有特殊字符。类似这样的判定都不是通用的技术层所能决断的，需要根据实际业务的进行抽象，为了研发人员可以方便地使用开源的组件和业务组件，要么完全抽象另一个字符串组件与开源组件区分开，要么兼容开源组件中大部分内容，并且这些事情在框架搭建之初最好就能达成共识。
- **连接池，产品需要访问非常多的数据库目标，**这些数据库目标是动态确定的，连接数据库的目标几千上万乃至上十万，开源连接池如果对每个目标数据库分配几个连接，那么就需要非常多的服务器来承载连接，我们更希望某些数据库在使用的时候能够分配并复用连接，不用的时候能够很好地回收，默认的开源连接池已经无法进行良好地在这种场景下支持了。我们需要自己来做一些连接池的扩展，要做这些事情，首先需要借鉴开源连接池的原理和思路，然后由两个思路来完成：1、由自己完整写一套动态分配和回收的连接池；2、基于开源连接池进行二次包装。不论选择那一种方式，都需要对原有的框架进行一些拓展才能让代码方便地跑起来。至于选择前者还是后者，根据业务复杂性而定，前者在研发阶段的代价更高，且需要长时间的验证，但对于大规模的数据库连接管理会更加持续；后者是比较容易接入的，且不需要太多的验证，但是后者的分配释放逻辑并不会完全由自身控制，相对

接入会比较重一些。

这些开源框架的拓展例子是取自 WEB 产品中较为常规的拓展思路，更多的具体框架和组件拓展，请读者朋友继续根据自己的产品继续深挖，例如你可能用到了：Quartz、MQ、JSCH 等等第三方或二方组件。

1.3.2. 业务模块化

开源框架不仅仅提供了一些便于研发的基础组件，也同时为研发在业务上的拓展提供了思路。业务级别的拓展更多是与具体的背景本身结合起来的，需要结合业务模型来设计出一个体系，一个高水准的研发或软件设计师需要清晰地认识到产品能看到的关键性核心模块和核心逻辑有那些？这些核心的板块都应当是需要被模块化的，就像上面的工作流、连接池一样，它们属于每个产品几乎都需要用的，产品内部也同样需要去抽象当前产品或相关产品的核心组件，便于复用、降低研发复杂度和 Bug 概率、研发可以更专注做好自己的业务、剥后的组件可以将自己的组件业务离做大做强。这些内容讲起来比较抽象，下面我们假想一个非常简单的交易平台：

- 任何人可以讲自己的商品发布出来卖。
- 浏览者可以收藏或直接下单。
- 可以在线完成支付和交易过程。

通过上面几个最基本的业务，我们认为应该将业务模块化剥离为：

- **用户体系**，关于用户的服务，应该包含对用户的注册、销户、修改信息、等级等信息，这也是绝大部分用户级产品都应该有的模块或子系统。
- **登录和退出**，属于后端服务，用于服务用户登录、退出、状态服务，所有组件都可以从中得知用户当前登录状态。
- **商品中心，**售卖者后台录入的商品，从前端页面到后台实现均应该由该产品完成，商品录入后，应该进入一个商品中心的，为外部提供商品详细信息和库存信息。复杂的系统可能将库存会单独剥离出来。
- **收藏，**该模块需要记录用户搜藏的商品，且与商品的变动进行定期或实时联动，提供用户查询自己的收藏信息。
- **下单交易，**该模块可以从用户浏览或收藏信息中提取购买的物件，计算价格和优惠信息，调用第三方支付系统完成支付过程，第三方组件的回调接口或轮询第三方支

付状态的处理也应当由该模块负责完成。

这是一个产品初步的划分，进而在模块上进行剥离，甚至于会拆分为子系统。这样做以后，模块之间的交叉是相对较少的，相互之间是以服务的方式来完成调用，内部的复杂性可以继续拓展，不会为其它模块所知，或者说相对其它模块是一个“黑盒”。与之对应的就是软件工程学的：“高内聚、低耦合”。

随着模块内部的复杂性越来越强，其可能会成为一个子系统，甚至于由其本身内部会抽象出非常多的基础组件、子系统、同级别兄弟系统。产品通常就是这样逐步地将业务体系做深、做大的。

大家还记得本章提到的“工艺流程”吗？在工艺流程中有非常多的步骤，每个步骤需要完成一个具体的业务模块，业务模块如果非常复杂，需要进一步细化模块内部的子模块。也是与本节对应的。

对于程序员来讲，模块化思路的前提是在懂得研发的基础之上，需要全局性去看待业务（相对全局），并且深度了解业务的背景和发展趋势，才能掌握从技术上如何去把控全局。在模块设计的过程中，从上到下逐步细化业务细则进行划分，了解模块之间的调用关系以及耦合度进行清晰地拆分处理，学会这一点，我们逐步就开始向架构的思维在买进了。

1.3.3 组件化思路

组件化的概念与模块化似乎有一些接近，相对来讲组件更像是脱离业务本身的组件，但又是基于业务背景编写出来的，组件也是为业务的编写提供便利的基础。使用的开源框架组件是基于通用业务编写的，我们在研发的过程中也会基于业务来编写一些组件。

组件化不一定是从头到尾编写的一个组件，很多时候也是基于对开源框架进行二次封装的成果，就像 1.3.1 节所提到的二次封装，再加以梳理就可以称其为组件。而组件化更像是明确目的、单一方向、稳定、成体系的相关服务 API 的组合。

如果说：**“组件是为业务进行服务的”**，那么是不是它是比模块更细的粒度？其实不然。

例如前文中所提到的“工艺流程”中的步骤有点类似于模块，具体的业务执行过程中可能会用到一些基础组件来完成，例如：汽车加工到某环节的时候，就需要对应的“机器人、电脑控制、组装部件等”来配合完成，其实这些东西就可以当成是公共组件，也是制造汽车所需要使用的公共组件。这些组件（机器人、电脑）他们自己的生产也会有单独生产流程，同样是一个很复杂的工艺流程，但是由于它已经被生产出来了，所以它的稳定性和复杂性不

是在这个业务中所需要去考虑的，在这个业务中它会被称之为“组件”。

从这个例子看来，组件是生产出来的，或者直接“买”来的。组件生产的时机也有点意思。如果是业务前期，通常只能抽象一个较为通用或范围较大的组件（除非是对原产品的重构升级），在具体的模块编写之前，如果涉及到一些相对复杂的业务，就可以开始梳理一些具体的组件了。业务逻辑梳理组件，模块复用组件，服务复用模块+组件，业务步骤使用服务来控制用户操作逻辑，如图所示：

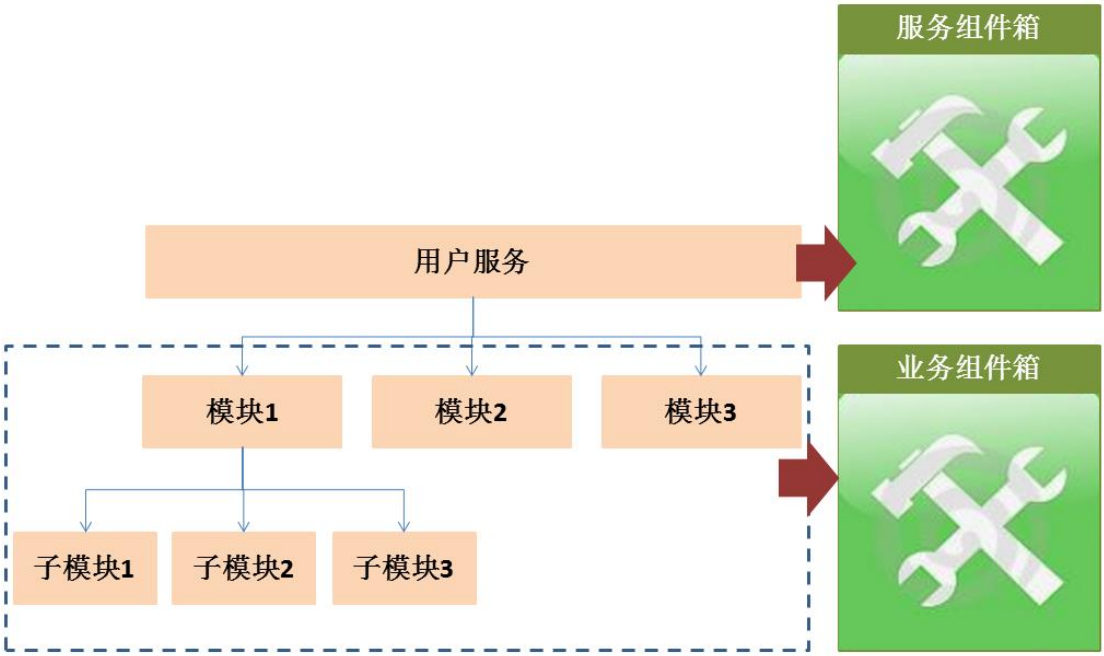

图 1-2 模块组件关系图

在软件研发中，开源组件数不胜数，从前端到后端的很多公共组件都已封装好，可直接使用。业务层的组件需要根据业务自身来决定，一般来讲，业务级别的组件按照类型划分常用的有：会话组件、MVC、任务调度组件、通信组件、连接池组件、工作流组件、安全组件、日志组件、监控等等。

如果是一些特定的业务，可能需要单独封装组件，例如某产品的业务上需要将一些超大文件切片为小文件、超小文件合并为相对较大文件的操作，那么在业务编写前应当封装相关的通用组件，便于后续的业务开发，初步地可以看出它应当提供这样一些接口：

- 输入 2 个或两个以上的文件地址进行合并，并返回文件的 meta 列表（即每个文件合并后在总文件中的偏移量）。
- 输入一个文件地址，并输出拆分的尺寸大小，拆分为多个文件分段。
- 输入合并后的文件名可以获取到所有合并的文件 meta 信息。

- 输入合并后的文件名和合并前的一个文件文件名，可以得到其偏移量、大小。
- 输入合并后的文件名和合并前的一个文件文件名，可以得到文件的内容。
- 输入拆分前的文件名，可以得到所有被拆分的子文件列表，并包含大小。
- 输入拆分的文件名，可以得到所有子文件的合并内容。
- 输入拆分的文件名+分段下标，可以得到具体分段的内容。

这里提供的接口是一个用于文件处理的，它所提供的功能与业务是不太相关的，但是它是基于业务来抽象的，这类似于：“现实生活中买因为要修门买了锤子和钉子回家，但是这个锤子和钉子还可以用来修家里别的东西”。通过这样封装后，可以得到一个针对于该业务较为完整的文件处理组件。当这种组件和业务进行剥离后，它们的代码编写方式将是完全不同的，组件将完全注重自身的正确性、稳定性，部分场景下还会考虑自身的容灾策略，业务将会认为组件的服务一定是可靠的，只需要根据业务的需要组件相应的方法来完成即可，它将更加专注于业务逻辑。因此业务和组件本身都更加具有逻辑的扩展性，或者说绕在一起的话即使没扩展之前就会相对复杂，要拓展业务和逻辑就会很困难、很容出问题。

综合来讲，如果你的业务代码中有大量判定状态、选择性的操作，这通常是需要二次封装的。例如，某业务为了在读多写少的业务中提升性能，将业务数据写入到 RDBMS、Redis 当中，读取时先从 Redis 读取，如果失效再从 RDBMS 读取再写入 Redis；这个看似简单的过程，如果将代码与业务绕在一起，就会存在很多问题。

这里封装的 API 并没有对大文件、小文件的判定逻辑，为什么？

封装组件的过程中切忌加入过多的业务因素在里面，这一层它一定要单纯地只做文件处理的，它也只关心文件合并和拆分的处理细节，即使内部有对文件大小的判定也只是对产品运行过程中的自我保护。业务上需要判定文件大小来决定合并还是拆分，不应该在这里被抽象，否则后续要调整判定逻辑，需要去修改组件才行，这会增加 Bug 出现的概率。

组件有大小之分吗？

组件可大可小，从 Java 代码角度来讲，可以细到具体的业务中的处理细节，也可以是核心引擎，所以，组件在于看问题的思路和封装的方法，不在乎大小。换一个角度，从宏观上将，随着业务体系的发展，整体业务会变得十分庞大，子系统会越来越多，很多子系统的复杂性是有可能胜过核心系统的，但是从总体上看，每个子系统也可以算是庞大业务体系下的一个组件。

封装组件的小技巧：

对于组件的封装，只要首次整理清楚它的目标，也就是组件的作用，然后不断梳理其细

节后再封装组件，是一个较为不错的切入方式。它是否完美并不是那么重要了，甚至于刚开始的时候部分方法的实现体可以先不实现（抛出 NotSupport 相关异常）。当我们发现业务的发展过程这些 API 已经无法满足需求的时候，可以再来重新梳理是否需要增加新的 API，或者基于这些 API 来做二次封装。

1.3.4 代码分层

前文中我们提到了模块化、组件化，从这两节中已经可以看到分层的影子，组件就是相对于业务更靠下面的一层，有些时候会把它叫做：“底层”，这个概念也是相对的。在 WEB 项目中通常会将后台代码分为多层，常见的有控制跳转层、业务层、数据访问层，如下图所示。

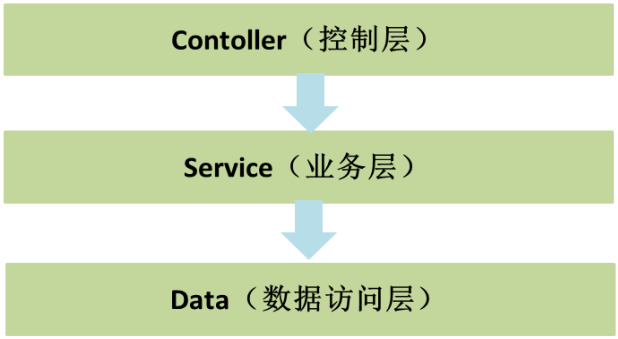

图 1-3 WEB 项目常见业务分层方式

分层的意义何在？

提到分层的意义，直接讲比较枯燥，不如反过来探讨这个问题：如果不分层会是什么样的。在较早写 JSP 的时候，代码几乎都是通过 JSP 里面的逻辑来完成的，这里面包含了页面的跳转判定、渲染逻辑、数据库访问、业务动作控制，放在一起感觉比较简单，但是代码非常难读，并且随着业务的发展，发现很多页面会有许多公用的代码，虽然可以通过 JSP 的 include 来达到一定的共享效果，但是这种复用随着模块的增多是相对比较难于维护的，如果产品需要横向或纵向拆分为子系统，将是更难的事情。这样的研发模式如果想后续扩展方便，取决于架构师和程序员的整体能力，这显然不同的公司做出来的水准千差万别，这也不是 Java 的理念。

因此有人开始抽象业务研发中的模型，标准的分层模型可以让所有产品的架构是一致：

- 数据层只做数据输入输出、数据细节转换处理等事宜，可以是数据库、接口访问、

文件系统。

- 业务层通过数据层对数据进行处理，自己并不关心数据处理过程，专注于根据数据的状态或结果进行相应的业务动作处理，也可以进行多个不同的数据访问 API 的调用，根据需要决定是否封装为事务，并将业务层的处理结果返回给控制层。
- 控制层拿到业务层的处理结果后，决定渲染的页面以及填充需要渲染的数据（这些数据也是来自于业务层），控制层也可以调用多个业务层的方法。

回过头来看，这个过程本身就是分层模块化的过程，通过分解层次达到复用的目的，每一个模块的每一个层次，如果有必要，可以进行剥离用于服务 API，尤其是业务层的代码将会越来越复杂。

分层代码编写的过程中，我们很容易犯一些小毛病（小胖自己也犯）：由于不清楚分层的意义，所以很多代码不清楚写在那一层，业务层的部分代码会写到数据层或控制层，而且这样写还会“上瘾”，因为觉得这样写会比较简单。但如果遇到模块复用的时候，就会发现很多代码还得重新来一遍。所以小胖认为，分层的意义理清楚，才能帮助我们去搞清楚层次之间的边界，当然即使再怎么理解层次的意义也一定会存在非常模糊的地方，但在理解意义的基础上这个边界会非常小，我们不必过分在乎了。

随着业务的复杂化，部分业务层的代码一定会变得非常地复杂，因为它是专注于业务逻辑的区域，业务的发展必然让业务层变得复杂，这个时候，就不要老想着三层结构了，需要将这个大的业务层打散成多个小的业务层，如下图所示：

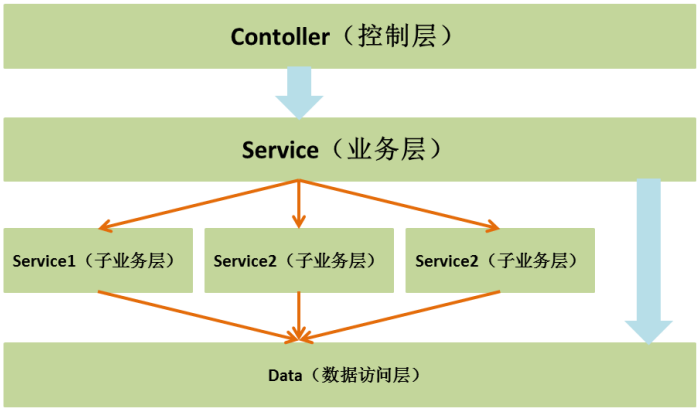

图 1-4 组件分层的拓展

控制层随着控制逻辑的复杂性变强，根据实际的业务，会由一个控制逻辑变成多个，但是它通常不再分层。

数据访问层随着访问的数据内容越来越丰富可能会开始按照类型的垂直划分，通常也不需要再多分层，唯一需要多分层的理由是在基础数据处理输入输出的基础上需要封

装一些通用的数据处理逻辑。

多层代码经过分层后，在进行业务垂直和水平上的业务拓展甚至于是子系统的拆分，都将会变得十分清晰。有人会提到这样分层代码会变得很重写，会一些十分简单交互代码，变得需要使用非常多的代码才能完成，并且对于代码编写人员不需要不断切换写代码的思路来完成（每一层的意义不同，需要用不同的思路来编写，否则就会出现前文中所提到的分层不明确的问题），这样可能会让程序员的效率降低，很切感觉非常辛苦但是没完成太多的业务目标，确实如此，因此，很多公司开始让程序员按照分层分工合作的方式来完成业务目标，让程序员专注于某一层来完成工作，这也算是一个解决问题的方法。但是这样的方法依然会有一些问题，这样的产品每一个业务点都需要每一个人你参与，每一个人都要较高的素质以及彼此之间配合默契，如果中间任何一个人跟不上节奏就会影响整体节奏；另外，这里面没有任何一个人可以深刻认识一个模块完整的从业务到后端的整体业务目标，以至于可能会失去一些主动对业务的思考，因为每一个层次都无法决定全局。所以，我们又会问一个新的问题：**在实际的产品中，是否可以没有中间的某一层呢？**

答案是肯定的，首先在技术上是没有任何难度的。其次，架构的目的之一是提升效率，所以在未来维护和效率上我们会做一些权衡，根据产品的业务特征来权衡。假如说产品中90%以上的代码业务层的代码都非常简单（可能经过组件封装后非常简单），并且你能够掌握未来较长时间这些模块都是比较简单的，那么它是可以与数据层合并的，针对不到 10%的业务单独分 3 层处理即可。

一些地方分 2 层，一些地方分 3 层，这样写会不会很乱呢?

小胖认为不会，我们的目的是达到可维护的目的，整体的代码架构依然是清晰的，在这个基础上效率可以得到提升其实就足以。高矮胖瘦在任何事情上都是存在的，只是我们内心的强迫因素在作怪而已，我们敞开一些内心的纠结去看这些事情其实就好了。话说回来，部分业务复杂性增强后，可能会超过 3 层呢?

同样的，每一层并不一定要用接口来完成，早期的分层框架之所以用接口也有技术实现方面的问题，而且接口是一种标准，因此我们习惯于用接口，但很多业务其实并没有那么容易标准化，业务在摸索阶段，业务量也非常庞大，变动会非常频繁，接口也不会稳定。在这种场景下，如果每个模块还需要去加一个接口，研发的效率会大大降低。这样的情况可以先不用接口，快速迭代业务，只要我们在编写代码的时候考虑到后续需要抽象即可，当某模块

的业务稳定后，再来抽象也不迟，如果真的遇到无法抽象的情况，那么就要考虑重构思路（下一节介绍），此时即使重构也不必过分担心代价，因为业务已非常清晰，可从上向下梳理业务，达到整体推进的目的。

小小总结：

分层是一种思路，MVC 的分层模型是一种参考，我们需要理解其意义和思想所在，才能对产品研发游刃有余。同时，分层的概念不仅仅局限于 WEB 项目中，在其它很多产品和组件的代码中均存在分层思路，这为我们自己去研发一种组件、非 WEB 类的产品时提供了思路。

同样的分层不仅仅在代码方面，经过系统的庞大化，分层也会出现在子系统之间，这一点我们放在本章 1.5 节拆分中来讲解。

1.4 重构

有不少人想要构建完美的产品，小胖也曾尝试过，但是事与愿违，很多社会的变化往往超出我们的想想，通过不断地在产品中摸索，小胖认为完美产品通常就是业务没有太多发展的产品了。

只有业务发展迅猛的产品才会让我们不断去修改产品，甚至于到了需要重新认识产品的需求、目标、业务价值、产品架构的阶段。此时，我们不得不去做一件事情“重构”，或者叫做自我革命，但是不重构就是想好好活着，但是也算是继续等死。

重构是需要我们重新梳理产品的体系，从上向下梳理分布、逻辑，而不仅仅针对问题改变问题，不断在产品中寻找某一个位置加一个 else if 逻辑或在原有判定逻辑上增加与或非。我曾经看到满篇的判定逻辑代码，甚至于代码中出现了大量的地区名称的判定逻辑，来保证产品的适配性，大家试想一下，这样的业务逻辑不断发展以后谁能理清楚其中的细节，谁又敢调整这部分代码呢？

改变自我对于曾经经历产品发展的人是一种痛苦，对于新人是一种惊喜，这个时候需要信人的胆量和老人对业务的严谨一起才能完成。话说回来，重构并不是好玩，我们也不能为了重构而重构，而且希望重构的影响面尽量小，毕竟重构会消耗较大的人力和物力才能完成，下面我讲解几种常见的重构场景。

1.4.1 快速化局部重构

快速局部化重构是我们日常工作中经常遇到的，也就是你遇到产品发展中遇到了什么样的问题，例如大量重复的判定逻辑、大量的字符串或数字处理等等，就应当考虑局部化重构了，只有这样产品的体系才能走得更久。局部化重构可以细化到一个方法里面，就像 1.2 节中的小例子一样（这里不再多举例），当你发现这样的问题，可以花 10-20 分钟便可以将问题处理好，而且随着局部重构越来越多，整体代码自然会变得干净整洁得方向发展。

局部化重构也可以抽象到组件，例如 1.3.3 节中所提到的组件（本节也不再举例），同样的你可以花上一些时间将组件封装好，便开始具备框架思维方式。

那么，是不是局部重构等价于封装组件和代码抽象呢？

是，但不完全是，其中的区别在于重构是业务发展到了一定程度，导致架构上已经开始阻碍业务的发展或开始产生未来的隐患，并且发现它可能会开始恶化，为了挽救这些事情而对整个框架体系的补充。而前文中所提到的组件更多是从产品发展之初，从整体架构体系或业务体系的角度来设计一系列的基础组件，或者说这是一个从上到下的设计体系，并非查漏补缺的过程。

局部化重构的度如何掌握？

有一定经验的程序员都会发现这样的问题，但是这类的问题的修改通常不会得到非技术类老板的赏识，因为这类改动只会让许多的隐患扼杀在摇篮里，不会带来大量实质性的变化，甚至于会减少未来的“优化”的机会，并且还会花费程序员的很多时间，还可能导致新的问题，所以在这个时候会不会去改动是看程序员的素质。

但是对于局部化重构，我们需要辩证一点看待，局部化的重构也会花费时间，但这不是我们不去做的理由，这好比一个赛车出发前任何一个部件都要检查好，这样才能保证发挥赛车的最大能力。反过来看，现实并不是那么完美，局部化重构永无止境，而且越加抽象的代码对于后续维护者需要学习的内容会更多，对于产品来讲，每一个程序员也应当考虑阶段性的胜利和成果，在这个中间我们需要选择一个平衡：局部化重构的条件是当问题有恶化倾向并可能会持续的时候选择，而且要做就一定要取得阶段性的胜利。

什么是阶段性的胜利，你可以设计一些指标，例如：彻底解决这类 Bug，业务代码量降低多少，业务中产生的 Bug 降低多少等等，如果做到了，那就是胜利。

1.4.2 模块化局部重构

模块化局部重构与快速局部重构类似，如果模块足够小，可以认为就是快速局部化重构。但当业务发展到较大的规模时，其设计就远远不止增加几个组件或调整一些代码了，对于业务层需要全面思考。

举个例子：运营商提供的电话卡开通其后台应该会有非常多的步骤，我们以计费为例子，最初的电话卡只是用于简单的打电话、接电话、短信，因此计费是十分简单的，可以是产品中的一个小模块，加上一些计算公式就可以完成了，但是随着业务的发展，计费开始变得十分复杂：

- 彩信计费
- 长途、漫游开始各种计费
- 跨运营商计费
- 网络费用
- 随着费用越来越复杂，运营商推出各种组合套餐让用户自由选择，每一种套餐收费标准完全不同，还有超出套餐范围外的计费

因此，计费系统开始变得十分复杂，如果用原来的计费入口来改造，代码会让业务本身变得十分复杂，对于业务来讲，只需要将套餐和相应的时间参数给予计费系统，计费系统便可以结算费用是最好的。

业务通过多年发展后，这些套餐已经比较明确，那么对于结算这一块的重构思路自然就变得比较清晰了，因此需要将该模块进行全面重构，才能达到目的。而全面重构后，该业务的复杂性可能已经不亚于一个子系统的时候，就会涉及到子系统的剥离拆分了，关于这一点在本章 1.5 节中继续。

1.4.3 全局重构

同上，全局重构是整体业务的变化已经不可控所致，或者说业务的发展已经由于架构的问题非常困难，这个时候，整体业务需要重新梳理，模块需要重新设计，分层需要重新设计，组件需要重新设计。

除此之外，产品的业务目标如果彻底改变（在现代快速发展的社会不是不可能的），或产品的定义重新改变，尤其是在原有产品定义的上层或中间添加层次，此时是产品必须要去

面临一定量重构。

通常全局重构，也会涉及到子系统的拆分，而在子系统已经拆分的系统中，全局重构也通常是子系统内部完成。即使如此，它也是一个非常痛苦的过程，这将推翻原有的思路，全面重构，对于业务复杂的系统来讲，这是我们不愿意看到的，所以，我们在日常多做局部化重构，必要时机对大模块进行，用“短平快”的方法不断迭代，让代码不断顺畅发展，可以让产品的发展更加具有延续性。

即使如此，在必要的时候，也一定要做，这个原因同样与前文中辩证看问题的方式是一样的，如果不做，可能活在当下，但几乎等同于是等死。

小小反思：

- 不因代码迟早要重构而在没有想清楚的时候开始写代码。
- 不因代码迟早要被优化或看不到短期效果而放弃日常的局部化重构。
- 不因需要快速要实现业务而不全部读懂原有代码直接上去寻找 else if 的位置。
- 不因业务过忙而不抽时间对自己的模块进行局部化重构。

这一切一切都是为了业务流畅地发展，更加长远地发展，也是为了我们这些技术人员对于技术的执着追求，这也是一个技术人员的自我尊严所在。

1.5 拆分

拆分，在前文中已经多次提到，相信读者朋友对于拆分有一定的理解，在模块化、组件化、代码分层等几个方面，都无时无刻都在阐述“拆分”的不同价值，而本节将是拆分的延续：“子系统拆分”。

在业务发展的过程中，我们不断通过局部优化、局部化重构、模块重构等方式在让产品不断清晰，顺畅地向前发展，但是各个模块的复杂性变得十分重，可能会导致以下问题：

- 部分模块变动频繁，模块发布相互影响
- 部分模块业务量变得十分庞大，需要剥离进行单独管理
- 业务的性质不同，以至于对性能的需求不同，希望剥离优化
- 部分模块希望不仅仅服务于产品内的业务，需要为更多的产品去服务

接下来就上面提到的这些场景进行举例说明，让大家能够感性认识拆分的时机和目的，我们在实际的产品中应当如何选择。

部分模块变动频繁，发布相互受到影响：

这类例子其实大家都遇到过，例如前端用户只要调整一下样式，可能会涉及到整个产品的发布，而其他模块用户的按钮点击可能也会在发布的时间点上失败几秒钟。

在前文中提到的运营商对电话卡结算模块，假如运营商从开通到最后结算是一个系统，假如运营商在调整业务的时候，对开通 CRM 及结算的改动是最大的，但是这两个模块的改动将频繁发布，而其他的模块可能根本没有改动而受到影响。

在这类场景下，可以开始考虑拆分，但是不一定非要拆分，为什么？模块间的发布要做到没有任何相互影响是不可能，不能因为有影响就拆分，类似于结算和 CRM 发布来讲，它的客户是运营商内部人员，且白天工作，所以发布放在晚上即使有影响也无所谓，所以更加适合的判定依据是对业务的影响是否需要拆分。但并不意味着发布频繁是不应该拆分，这里所证明的是它的算是一个拆分的理由，但还不够充分。

部分模块业务量变得十分庞大，需要剥离进行单独管理：

例如某公司提供的支付平台，最初的可能就是为了付款，绑定银行卡进行付款操作，这可能是一个对银行打交道的模块，提供交互通道。随着业务的发展，业务形态发生了诸多变化：

- 银行卡种类越来越多，需要打通各种各样的通道并需要进行通道选择。
- 利息，让用户将部分资金存入。
- 理财服务，理财服务公司以各种不同的形式对接。
- 借款（信用），用户存入资金借款给他人。
- 贷款，用户存入资金可以贷款给其他人。

这里面的业务还会继续变化，可以覆盖到用户生活的方方面面，但这一切注定，业务并不会像当初一个简单的银行通道那么简单，如果业务继续还在原产品中继续发展，会比较痛苦，因此在这个时机我们可以考虑拆分和剥离。

同样的，它还不是充分条件，只是构成拆分的因素之一，再复杂的模块，只要划分清楚也同样是可以管理的，要知道拆分后，如果没有足够多专业上的人，维护成本是更高的。

业务的性质不同，以至于对性能的需求不同，希望剥离优化：

在一个产品中，部分模块涉及到大量数据计算，部分模块涉及到大量的网络调用，部分模块是后台调度任务，部分模块涉及大量的用户请求，各自都有性能优化的方向以及 JVM 参数设置的调优方式，此时融入一个产品中会比较痛苦其发展。

此时我们希望剥离其中的模块成为单独的产品来提供服务，对产品内的性能、业务进行

单独的抽象和优化，例如处理数据的模块对物理内存、JVM 的 TLAB、CPU 数量会有特定的要求，对用户请求量大的产品对 WEB 服务器参数、线程数、连接池数量、阻塞队列、各类 Timeout、单个请求的性能等有特定的优化方式，它们所用到的组件也是完全不同的，所以，在这种情况下，是一种拆分较为充分的条件。

同样的，为什么只是较为充分呢，这取决于 2 个“度”因素，一个因素是在性能和业务的局部优化是否那么迫切，如果现阶段以及未来较长时间不是问题，那么拆分必要性不大；另外，拆分会带来子系统的维护成本，需要有足够的人将每一个子系统局部做强做大，才能达到拆分的意义。

部分模块希望不仅仅服务于产品内的业务，需要为更多的产品去服务

这个条件是一个绝对充分的拆分理由，因为它是一个绝对新的业务形态，为了让业务变得更加庞大，如果在原有产品里面继续拓展，会让原有产品变得完全看不懂是什么，或者说是一个大杂烩，那么自然也不会有业务的定义方向。

这样的拆分，会让被拆分的子产品拥有广阔的思考空间去拓展，如前文中所提到的支付平台如果拆分后就可以开始自己为其它公司提供支付的能力，还可以提供各种金融服务业务，反之它将受制于本身产品的业务限制，代码也会因为产品本身而变得不太通用。

这些子系统的拆分，将会在产品中以兄弟产品或相对不同的抽象层次出现在整个业务服务体系中，通用的产品还可以存在于多个业务体系中，这类似与数据库也算是一个几十年前全球业务中统一拆分出来的子系统，用于数据存储计算服务的子系统。这个时候，对于单独的业务服务体系，就应当有单独的业务体系架构图，可以清晰地认识到产品的服务关系和支撑的业务范围，这样做的目的是为后续架构图的布局提供参考，或者说从全局的角度去布局整个架构。

敢于这样做需要有一定战略布局思路，对于产品中的任何一个大模块甚至于不起眼的小模块，如果你具有足够的思维视野和眼光，就可以把它变成一个非常庞大的系统体系，相反，很多隐藏着非常重要的事情可能会因为我们目光短浅将事情做得很小。拥有一定的布局思维后，如果你有无限多的人来为你服务，你可以去做你任何想做的事情，但如果只有一定的人数，你会选择将什么剥离就是你对于业务的敏感度和对模块拆分后价值的判定能力。

1.6 小总

本章所提到的所有知识，都需要同学们对其有一定的辩证看法，并非某一种方式在任何

场景下都很好，也不仅仅是希望大家从中学到方法，我们更多是通过找到目的及案例学会分析场景和推敲的习惯，在业务、资源、矛盾、方法学之间我们需要去做一些决策，在有限地条件下去发挥最大的价值。

希望大家能够以自己产品中的实际例子来进行摸索思考，从业务、技术架构、服务、可用性、监控、部署、隔离等角度画一些图，也可以画一画复杂模块处理流程图，也可以画一画开源框架的架构图和处理流程图，然后不断推敲和思考发展方向及缺陷的解决方案，也不乏自己做实验进行推导，养成这个习惯，对整体的思维会越来越清晰，你便开始具备从设计角度思考问题的能力。