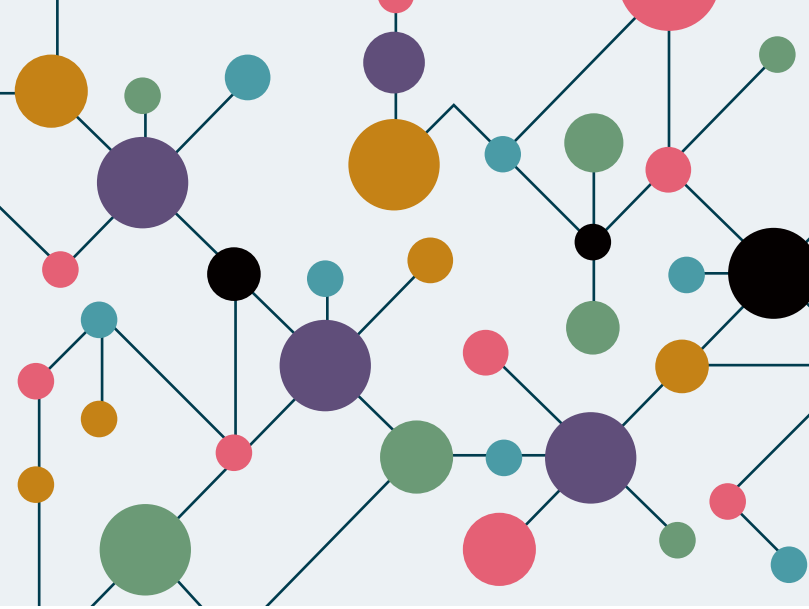


换个角度认识软件

林宁 著

/thoughtworks

为什么我们跟别人说不明白?

在软件开发过程中，比较难的一件事就是如何表达需求、方案、问题，甚至有时候日常沟通也会出现“驴头不对马嘴”的沟通窘境。

之所以会出现这类表达问题，一部分原因是我们对逻辑的理解不同。大多数有经验的开发者、系统分析师都具备一定的辩证思维和方法，要说谁没有逻辑，这件事情很难说得过去。如果每个人都是用自己的思维方式和“逻辑”，让沟通过程变得非常困难。令我疑惑的是，每个人都相信逻辑是很重要的，但几乎没有文章讨论过在软件设计和开发过程中如何使用现代逻辑学，以及解决诡辩的问题。

这里讨论一些能在软件工程中使用的逻辑学基础知识，尤其是概念相关的内容在业务分析、领域建模和架构设计中都可能会被用到。

1 当你提到用户时，你是指什么?

我曾经参与过一个物联网系统的设计，其中大家经常会提到一个词“设备”，但是“设备”这个词在不同的开发者眼里有不同的概念，为此，讨论“设备”这个词花费了不少的功夫，最终依然没有定义清楚。

有些开发者认为“设备”是现实中看得见摸得着的物品，另外的开发者将服务器上用于映射物理设备的实例也叫做“设备”。于是，他们在沟通时经常会出现对于“设备”的理解不一致导致的混乱。

另外一个例子是“用户”这个概念。在不同的场景（上下文）下，“用户”这个概念可以是使用软件的大活人，也可以是数据库中的一条记录,也可以是服务中的用户对象，有时候也将用户服务类叫做用户。

这样就非常混乱，不仅无法沟通，而且还导致开发者对系

统的认知也变得困难，很多东西处于混沌状态。

在咨询的工作中，我发现非常有意思的是，将软件中的概念一一定义清楚，整个系统的设计工作差不多就完成了。所以设计软件的过程和现实中人们相互交流非常类似。英国哲学家维特根斯坦把人们交流的过程叫做“语言的游戏”，当我们描述事物的时候实际上就是将有清晰边界的元素贴上标签，这个标签就是我们说的概念。

朴素的概念是来源于个人背景和理解，因此概念难以统一。正是因此不同语言之间准确地翻译也不太可能[1]，不同文化背景难以找到合适的概念互相映射。后来哲学家认识到人们认识概念是由一些更为基础的属性构成的，那可以认为概念就是由属性组成的。比如“人”这个概念，有四肢、直立行走的行为、皮肤光滑等属性。

这些基本的属性又是一些更基本的概念，如果我们对这些基本的概念达成共识，那么我们就有机会对概念进行统一。类似于面向对象语言 Java 中的类，类有各种属性，这些

1 翻译是否真的可以被实现是一个经常被讨论的观点，参考：严春友 . 翻译之可能与不可能——对翻译问题的哲学思考 .

属性最终都可以通过 8 种基本的数据结构描述。

因此属性是认识概念非常重要的一方面。属性包含了事物自身的性质、行为，比如黑白、高矮、是否能飞行、是否独立行走。事物除了自身的性质外，还与其他事物发生一定的关系，比如大于、相等、对称、属于等。事物的性质、行为以及和其他事物的关系，统称为事物的属性。

通过属性就能找到概念的边界。 具有相同属性的概念是同一个概念,即使它们的名称不同也不应该分为不同的概念。例如，土豆和马铃薯都是同一个概念。如果意识不到属性对概念的影响，则会出现生活中的命名错误，例如小熊猫并不是小的熊猫，而是单独的一种动物。

2 尝试精确定义概念

一个概念可以具有多种表达方法，对于软件设计来说，我们可以用自然语言描述概念。也可以通过定义一个类来描述，并在程序运行时实例化这个概念。通过数学或者数理逻辑，我们可以使用集合来描述一个概念。

比如“商品”这个概念，可以通过不同的方法表达。

自然语言中，商品是指可以通过货币或者其他物品交易的物品，可以是自然实体，也可以是虚拟物品。这是社会经济中对商品的描述，商品具有一个核心属性就是价格，有价格意味着可以交易。

自然语言（Natural Language）就是人类讲的语言，它是人类自然发展中自然形成的，比如汉语、英语。

这类语言不是经过特别设计的，而是通过自然进化的。它的特点是**语法规则只是一种规律**，并非需要严格遵守的规则，这种语言含有大量的推测，以及对话者本身的认知背景（比如东西方不同的文化背景形成了大量的哩语）。认知背景赋予了词汇、概念的不同含义，比如，豆腐脑这个词，不说东西方差异，就是国内南北都会有争议。

著名的白马非马争论在于自然语言的不确定性：

- 从概念上说，白马这个概念不是马这个概念，所以白马非马。
- 从谓词（“是”这个谓词）逻辑来说，白马这个概念代表的事物集合属于马这个概念代表的事物集合。

> 所以白马是马（白马属于马，但是白马这个概念不是马这个概念）。

正因为如此，才会产生大量的诡辩，让交流效率降低。

在逻辑学中，形式语言开始发挥作用。形式语言（Formal Language）是指用精确的数学或机器可处理的公式定义的语言。例如数学家用的数字和运算符号、化学家用的分子式等，以及编程语言中的一些符号（Token）。计算机编程也是一种形式语言，是专门用来表达计算过程的形式语言，以操作计算机。

形式语言需要严格遵守语法规则，例如 1+1=2 是数学中一种形式语言。

总之，知道形式语言和自然语言之间的区别，可以避免无意义的争论。**软件工程师就是一个对现实业务形式化的工作岗位，将需求这种自然语言转变为代码这种形式语言。**

正因为如此，需求和沟通的矛盾不可能避免，除非提出需求的人也使用形式语言，那么软件工程师的价值也就没有了。

使用形式语言可以精确地定义一个概念，并使用精确的语义规则和计算机沟通，这就是软件工程师编写软件的过程。如果通过计算机语言来描述一个概念，其实就是面向对象中的一个类，这里定义商品有两个属性名称和价格：

```
public class Item {
  private String name;
  private BigDecimal price;
}
```

如果用集合的枚举法来表述商品就是：

```
Item {name,price}
```

计算机语言和数学语言是一种形式化的语言，可以精确地描述一个概念，但是自然语言只能通过模糊地给出概念的描述。自然语言翻译成计算机语言的不确定性，带来了无休无止的争吵，但这也是软件设计者的主要工作。

3 对象应该定义多少属性?

正是因为自然语言的这种模糊性，为了更加具体地描述一个概念。哲学上概念的共识是，概念有两个基本的逻辑特征，即内涵和外延。概念反映对象的特有属性或者本质属性,同时也反映了具有这种特有属性或者本质属性的对象，因而概念有表达属性的范围。

例如商品这个概念的内涵是“能进行交换的产品”，本质属性是能进行交换，从本质上区别产品。它的外延就是投入市场能卖的所有事物。

对概念外延的清晰描述对我们设计软件产品的定位非常有帮助，我们购买软件服务无非两种情况，生活娱乐使用，或者工作使用。马克思社会经济学精妙的描述为生产资料、生活资料。这其中的逻辑完全不同，按照生活资料的逻辑设计一款生产资料的产品注定要走弯路。

概念的内涵和外延在一定条件下或者上下文中被确定的，这取决于参与人的共识。严格锁定概念的内涵和外延，能帮助我们讨论问题和改进软件模型。随意修改内涵和外延这是典型的偷换概念和诡辩。

概念的内涵和外延是一个此消彼长的兄弟。当内涵扩大时，外延就会缩小，概念就会变得越具体。当内涵缩小时，外延就会扩大，反映的事物就会越多。

这在面向对象软件建模中的影响非常明显。对象特有属性或者本质属性越少，那么这个对象能被复用的场景越多，也就是内涵越小。反之，特有属性越多，能被复用的情况就越少了。软件建模过程中随意修改概念往往意识不到，但是每一次属性的添加和移除都带来概念的内涵和外延发生变化。

非常典型的一个例子发生在订单模型中。一般来说，我们会把支付单和订单分开设计，订单的概念中没有支付这个行为，但有时候觉得支付单的存在过于复杂，会将支付单的属性放到订单中，这个时候订单的内涵和外延变了。

内涵和外延发生变化但是设计人员没有意识到，会使用同一个词语。一旦使用同一个词语就会产生二义性，二义性的存在对软件建模是致命性打击。比如用户维护的地址、地址库中的地址、订单中的地址，这三个“地址”虽然名字相同，但是内涵和外延不同。

意识不到概念的内涵和外延，是无法设计出逻辑良好的软件模型的。

4 如何为变量命名?

变量命名和缓存失效是编程中最让人头疼的两件事。

变量命名实际上就是为一个概念进行定义。定义是揭示概念内涵和外延的逻辑方法，而一个准确的定义需要反映出对象的本质属性或特有属性。在下定义时，存在两个常见的困难：

1. 不知道如何使用良好的逻辑方法进行定义。
2. 对业务概念或者领域不熟悉。

对于第一个困难，逻辑学有一些很好的下定义方法，可以根据概念的属性、内涵和外延来进行。

属加种差定义法。 这种下定义的方法通俗来说就是先把某一个概念放到另一个更广泛的概念中，逻辑学中将这个大的概念叫做“属概念”，小的概念叫做“种概念”。从这个属概念中找到一个相邻的种概念，进行比较，找出差异

化本质属性，就是“种差”。比如，对数学的定义，数学首先是一门学科，和物理学处于同类，它的本质属性是研究空间形式和数量关系。于是可以得到数学这个概念定义：

数学是一种研究现实世界的空间形式和数量关系的学科。

用这种方法给订单、支付单、物流单下一个定义：

订单是一种反映用户对商品购买行为的凭据。属概念是“凭据”，种差是“反映用户对商品购买行为”。

支付单是一种反映用户完成某一次支付行为的凭据。属概念是“凭据”，种差是“用户完成某一次支付行为”。

物流单是一种反映管理员完成某一次发货行为的凭据。属概念是“凭据”，种差是“管理员完成某一次发货行为”。

在逻辑中可以参考下面的公式：

被定义的概念 = 种差 + 属概念

对于第二个痛点，这不是软件建模能解决的问题，需要充

分和领域专家讨论，获取足够的业务知识。人们对概念的定义或者认识是随着对事物的认识不断加深而变化的。一个完全对某个领域没有基本认识的软件工程师很难做出合理的软件建模，例如银行、交易所、财会等领域的软件需要大量的行业知识。

我们做消费者业务的互联网开发时，往往因为和我们的生活相关，所以这种感受并不明显。当做行业软件时，领域知识对软件模型的影响甚至是决定性的。

5 避免无意义的争论

如果需要建立逻辑思维，还需要一些逻辑规律。逻辑学的三个基本规律可以让沟通更加准确，避免无意义的争论，减少逻辑矛盾，让讨论有所产出。这三个重要的规律是：同一律、矛盾律、排中律。

同一律

在同一段论述（命题和推理）中使用的概念含义不变，这个规律就是同一律。形式化的表述是 A → A。同一律描述的是在一段论述中，需要保持概念的稳定，否则会带来谬误。

在辩论赛中可以利用这个规律，赢取辩论。

比如论题是“网络会让人的生活更美好吗？”，两个主要的论点是：

- 网络让人们的生活更方便。
- 网络让人们沉溺于虚拟世界。

假如我们选择的论点是“网络让人们的生活更方便”。在辩论赛中，我们陈述了“没有网络非常不方便”，反方被诱导描述了“打电话、写信也可以让人生活很美好，不一定需要网络，且不会像网络一样容易沉溺在虚拟世界中”。这刚好落入我们的逻辑陷阱。我们指出，邮政、电话网络也是网络的一种，对方的逻辑不攻自破。

这属于典型的“偷换概念”，我们偷换了“计算机网络”和“网络”这两个概念。

矛盾律

矛盾律应用得更为普遍，几乎所有人都能认识到矛盾律。它的含义是，在一段论述中，互相否定的思想不能同时为真。形式化的描述是：“A 不能是非 A”。

矛盾律这个词的来源就是很有名的“矛和盾”的典故，出自《韩非子 • 难势》中。说有一个楚人卖矛和盾，牛吹得过大，说自己的盾在天底下没有矛能刺破，然后又说自己的矛，天底下的盾没有不能穿透的。前后矛盾是一个众所周知的逻辑规律，但是并不是一开始马上就能看出来，需要多推理几步才能看出来。即使如此，在同一个上下文中，出现了矛盾的逻辑论述也被认为是不可信的。

具有矛盾的论述有时候又被称为悖论。尤其是宗教领域充满了大量的悖论，例如，是否存在一个万能的神，做一件自己不能完成的事情。

矛盾律的用处可以驳斥不合理的论断，也可以用于反证法。在软件开发过程中，我们时常遇到这种情况，需要在开发过程中才能发现矛盾。我们常常会接到一些充满矛盾的需求，下面举一个例子。

“在多用户空间系统中，用户可以被禁用并且可以加入多个空间。用户所属空间的管理员有权禁用用户。即使用户被某个空间管理员禁用，仍然可以使用其他空间的功能。”

上面提出了一个矛盾的业务需求：空间管理员可以修改用户自身的禁用状态,但是又要求空间之间保持隔离。实际上，需求提出者并没有理清用户本身和空间成员之间的关系。在需求评审过程中，我们经常会发现这种矛盾，并通过解决矛盾来改进业务需求。

排中律

排中律是逻辑规律中最难理解的一个规律。它的表述是：同一个思维过程中，两个互相否定的思想必然有一个是真的。用形式化的表述就是：“A 或者非 A”。

排中律的意义在于，明确分析问题的时候不能含糊其辞，从中骑墙。比如有人讨论：人是不是动物。不能最终得到一个人既是动物又不是动物，这种讨论是没有意义的。

比如在一次技术会议中，需要选择使用的数据库，只能使用一种数据库。如果采用了 MySQL 就不能说没有采用 MySQL。

排中律看起来好像没有意义，却是一项重要的逻辑原则，让讨论最终有结论，而不是处于似是而非的中间状态。

用模型理解编程语言和面向对象

1 模型是错误的

模型这个词常常会听到，通常出现在某个 PPT 或者一篇商业评论中。社会和经济学中的模型往往比较朴素，金字塔、V 型图、四象限会以各种形式出现在不同场合中；软件工程师的模型会更加形式化，UML、E-R 图等，能用较为精确的形式语言描述；数学模型就更加精确，马尔可夫、蒙特卡洛等模型可以用数学语言描述。

广义来说这些都叫模型，甚至是你随手在白板上画的一个用来解释当前程序结构的图形，通过这种方式表达思维框

架。哲学家库恩将这种思维框架叫做范式，也就是模型。维基百科将广义的模型定义为：

> “用一个较为简单的东西来代表另一个东西，这个简单的东西被叫做模型。”

我们天生就有用简单的东西代表另外一个东西的能力，比如幼儿园数数用的竹签，学习物理时的刚体、真空中的球形鸡，都是模型。通俗来说模型就是经验的抽象集合，平时听到的谚语、公式、定理本质上都是一种模型。

为了理解模型，斯科特 • 佩奇在 《模型思维》一书中给出了模型的几个特征：

1. **模型是简化的。**正是因为我们要认识的事物非常复杂，因此需要通过简化找出最一般的规律，才能一语中的。“天圆地方”学说就是最简单的古人认识世界的模型之一；毛主席的“阶级划分论” 简单、直接地指出旧中国的社会状态。
2. **模型是逻辑的。**例如用金字塔原理描述社会阶层，每层的定义是明确的而非模糊的，数学模型能用数学符号系统和公式描述，模型中的元素能用一种逻

辑关系做到自洽。

3. **模型是错误的。**因为模型是一种抽象，所有的模型都是错误的，只能在一个方面反映事物的特征。场景变了，模型就需要修正，连牛顿、爱因斯坦的定律都没能逃脱这个规律[1]。好的模型能在尽可能简单的情况下较好地拟合事物，完全匹配现实的模型就不再满足简化特征了。

为了建立和利用模型，模型思考有几个层次：

1. **数据。**我们能直接观察到的现实情况，比如下雨了，并从观察到的情况中采样，转换成具体的数字，得到某个地区某年的降雨量。
2. **信息。**信息是数据中反映出能被人类理解的内容，比如通过降雨量判断一个地区是否属于干旱地区。
3. **知识。**知识是对信息的处理方式，比如我们利用信息，将信息中的一般规律找出来，建立模型。比如某地区降雨量和年度呈现一定相关性，建立一个周期性降雨模型。

---

1 经典力学在高速、微观场景不适用；相对论在粒子物理学不适用。

4. **智慧。**面对不同情况需要使用不同的模型和修正模型的能力，并能用它指导实践，比如根据周期性降雨模型修建水利设施。

我们可以尝试用这种方式来看待原本很困难的知识，比如去简化复杂问题，并理解它。通过模型思维来看待软件开发，我们会发现，**软件从设计到开发的过程就是各种模型的转换。**

2 软件工程中的模型

我整理了一个图表，说明了一款软件从商业探索开始，到编译成可交付的软件整个过程中可能会用到的模型。

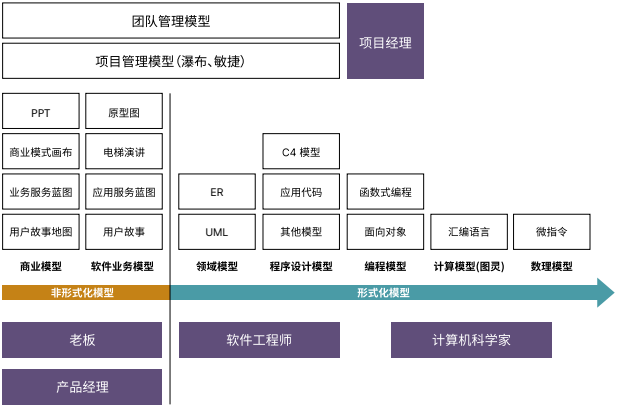


图 1 软件工程中的模型

我将模型分为形式化和非形式化两种。形式化的模型是精确描述的模型，例如表达领域模型的 UML、ER 图，而非形式化的模型是一些非精确描述的模型，主要用来做商业、业务探索。

对于应用开发的软件工程师来说，核心的问题并非如何编写代码，而是如何将非形式化的业务输入（模型）进行合理抽象、设计，并转换为形式化的过程。

某种程度上来说，通过高级语言编写的代码也是一种模型。在多年以前，计算机科学家们认为编写 Java 代码的人不算程序员，可以由业务人员直接编写业务软件。由于软件工程中非形式化和形式化之间存在一个巨大的鸿沟，编程就是模型的形式化过程，从这个角度看能深刻分析业务并获得良好抽象结果的程序员具有竞争力，即便我们有了 ChatGPT，分析业务和建模的工作也不会被它替代。

在非形式化模型这一步，实际上又存在两种模型。一种是描述软件背后的生意，即使不使用计算机系统参与到业务中，该如何完成交易，并让企业获得利润，我把它叫做商业模型。另一种是描述软件的操作和交互的模型，关注参

与的用户、流程和业务规则，我把它叫做软件业务模型。

除此之外，还有一些模型和计算机工作原理相关的模型，这是计算机科学家的工作，对于普通开发者来说可以加深对计算机的理解。例如，符合人类的认知的编程语言（面向对象、函数式）背后的模型，面向对象可以看作一种模型，还有一些计算机科学基础的模型：冯诺依曼结构、图灵模型、布尔逻辑（数理模型）。

将这些模型串起来，能够提高对软件工程的理解，以及每个部分背后的逻辑，明白这些模型背后的目标后可以更加从容地应对各种问题。限于篇幅本章讨论计算机科学中的模型，关于商业、业务分析、架构的模型将在后续讨论。

3 计算机科学中的模型

如今，计算机已经发展到一种近乎魔法般的存在。对于非计算机专业毕业的从业者而言，理解计算机和编程语言的原理以及面向对象思想变得困难。然而，我们可以寻找一些模型或比喻来帮助我们理解与计算机相关的内容。

从算盘、计算机到电脑

从算盘到计算机，人类走过了漫长的历史。计算机发展的转折点往往都是一些大师提出关键模型的时期，了解这些模型可以帮我们更好理解计算机世界。

新一代的软件工程师已经不再关注计算机是如何工作的了，他们把计算机当作一种可以通过编程语言对话的“生物”来看待了。我曾被问到过，我们日常使用的“电脑”为何被称作计算机，它和计算看似毫无关系。

要回答这个问题需要将图灵和冯诺依曼模型两个计算机科学基础模型清晰地分开。

算盘会被经常和计算机一起提到，算盘是人力驱动的一种计算机，算珠的状态可以看作寄存器。对中国人来说理解图灵机非常简单，我们可以使用算盘来类比。当算盘归零后，算盘的状态为初始状态，每一次拨动算珠就是一个指令，当所有的指令下发完成，算盘上最终状态就是计算结果。指令序列就是算法，算盘就是一个状态机。

在算盘之后的时代，还有计算尺，甚至手摇计算机。手摇

式计算机算是一种半自动的计算机，我国科研人员曾使用它进行原子弹的计算工作。

把“计算”这两个字与计算机联系在一起的功劳应归于图灵。1937 年，图灵发表了一篇论文，阐述了可计算性的概念，并提出了计算机的抽象模型。在论文《论可计算数及其在判定问题中的应用》中，图灵提出了著名的理论计算机抽象模型——“图灵机”。

它描述了这样一种机器：一个虚拟的机器，由一条无限长的纸带和读写头组成。纸带上分布有连续的格子，并能被移动、读写。机器能读取一个指令序列，指令能对格子纸带进行移动和读写。和算盘的逻辑一样，机器每执行一个指令，纸带的状态就发生了变化，最终完成计算。

图灵模型只是描述了一步一步地完成计算任务，这种机器称不上“电脑”。让一堆“沙子”具备通灵般能力的人是冯 • 诺依曼。现代的计算机实际上是一个死循环，可以类比为冲程发动机，才让计算机看起来有了生命。这才有了我们说熟悉的“电脑”。

ENIAC 是公认第一个满足图灵模型的计算电子计算机，

ENIAC 通过纸带编写程序，并拨动开关执行和获得结果。冯 • 诺依曼在比 ENIAC 更先进的计算机项目 EDVAC 中描述了另外一种模型，他认为程序本质上也是一种数据，将指令和数据共同存放到内存中，这些指令中存在特殊的跳转指令，让程序周而复始地运行。

存储程序模型构建了一个能自我运行计算模型，构成了一个系统。处理器和内存之间使用总线连接，用来给这个系统提供输入的设备叫做外设，每一次指令循环可以访问一次外设传入的信号，这就是中断。

想象一台由继电器组成的计算机，如果每一次执行指令计算机会发出“嗒” 的声音，图灵模型就是程序开始运行后线性地嗒嗒嗒……嗒嗒停”。冯 • 诺依曼的模型就是上电后 嗒嗒嗒嗒嗒……中断……嗒嗒嗒嗒嗒”，并反复循环。冯 • 诺依曼让计算机永不停息，并产生交互效果。

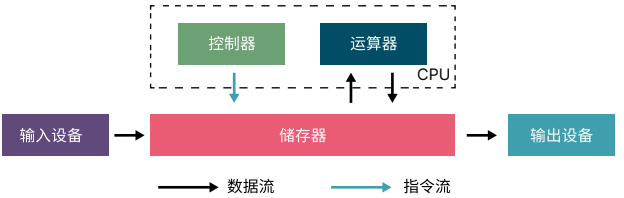


图 2 冯 · 诺依曼简化模型

总的来说，图灵模型将解决问题的过程抽象为算法和数据，冯 • 诺依曼进一步抽象了算法也是数据，进而让计算机无限循环下去。至于怎么通过门电路实现基本运算（全加器）可以参考《编码：隐匿在计算机软硬件背后的语言》一书。

通过自动推理模型理解编程语言

各种各样的编程语言层出不穷，由于工作的需要，我们会接触不同的编程语言。如何能理解编程语言的本质是什么呢？我尝试找一些模型简化对编程语言的理解。先用矛盾论分析一下编程语言解决的什么矛盾：

计算机只能识别机器指令和人类难以使用机器指令解决具体问题之间的矛盾。

所以人类设计出来各种各样符合人类习惯（各不相同）的

方式编写程序，这些编写程序的模型就是高级语言。要使用自己定义的语法规则来写程序，就需要一个转换器，能将符合人类习惯的语法进行转换，这就是编译器。

一门新的语言需要满足几个条件：

1. 新定义的语法必须是形式化的。
2. 新定义的语法能方便地转换。
3. 人们能接受这种语法编写程序。

所以编译器是一个自动推理机，只要能被推理的形式化语言都可以作为输入。除了自然语言无法实现之外，无论用中文、表情包、符号、图形都能作为一种编程语言的形式。

编译的过程有：语法分析、词法分析、语义分析、中间代码和优化、目标代码。大师通过编译过程学习如何实现编译器，普通工程师可以反过来用这个过程理解一门新的语言。

我尝试将编译过程中的环节找到一个现实中的类比来理解编译器，将其类比为人类阅读法律文书（法律文件是最贴

近形式化的自然文本）。

阶段	编译器	类比
词法分析	扫描，识别代码 Token，将关键字、变量、操作符提取出来	处理调查材料，案件人员、行为等要素
语法解析	将 Token 组织为一棵树 (AST) 用于推理	将人员和行为映射成图谱，用于形式逻辑推理
语义分析	处理上下文相关的信息	识别行为发生的动机、背景，提取上下文信息
中间代码	上面三步是前端，中间代码是为了多平台代码生成用	整理为卷宗
目标代码	根据不同的平台进行代码生成	输出到报纸、网站等媒体

我尝试找到一些通俗的模型理解编译过程，https://craftinginterpreters.com/a-map-of-the-territory.html 这个网站介绍了一个清晰的编译过程。

理解编译器后再学编程语言就清晰很多，比如语法（Grammar）有三个层次：

1. 词法（Lexical）：决定哪些表达式、关键字是否合法。
2. 句法（Syntax）：决定一个句子是否合法，比如流程语句。
3. 语义（Grammar）：决定一段代码的组织结构是否合法，函数、类、闭包等规则。

Lexical 和 Syntax 往往可以看成一体，Grammar 不太一样，在一些编译器中 Syntax 和 Grammar 的错误提示都不太一样。所以可以这样看一门语言：Syntax 是类 C 的还是非类 C 的，Grammar 上是面向对象的还是面向过程的，是否支持闭包这类上下文追溯的能力。

理解推理模型可以用来帮助学习编程语言，比如 TypeScript 可以编译成 JavaScript，很多时候我们不需

要特别学习 TypeScript，将小段 TypeScript 代码编译一下，看看生成的 JavaScript 是什么就行了。

面向对象模型

有了自动推理机，可以将人们定义的语法转换成机器代码的语法规则。让我们有了方法、变量、条件、循环等这些概念，可以大大简化编程的心智负担。

面向过程的语言依然还是图灵模型解决问题的思路：有限的有序指令序列。只不过这里的指令从机器语言、汇编代码换成了容易理解的表达式而已，面向过程的编程语言和机器代码在认知上没有本质区别。

组织面向过程的程序，这部分工作的心智负担需要高水平的程序员来完成，将现实中的业务分解成有限的有序指令序列。分解任务成为指令序列的过程就是编程，它要求程序员既要像人一样思考现实又要像机器一样思考。像机器一样思考需要最聪明的人来完成才行，好的程序员可不好找。

能不能想办法利用推理机，再进一步，让程序员按照人类

一样思考事物，写出符合人类语义的代码，然后再翻译成目标代码呢？回答这个问题就需要先回答另外一个问题，符合人类认知的思考方式的语言是怎么回事。

人类需要通过概念来进行交流，给一撮物质一个标签，这个标签就是概念。将一堆便签夹起来再打上标签，就是抽象概念。不同的语言、不同文化背景的人无法交流就是因为使用了不同的标签系统，甚至也有可能标签贴错了的情况，导致认知无法对齐。

要理解面向对象编程，需要到生活中去观察小孩玩泥巴的情景。他们用泥巴创造出一个城堡，而泥土就好像计算机世界中的数据。将泥土组织成有清晰边界的物品就是对象。我们为了描述这类对象，就给它们起个名字以便交流。类可以对应现实中的一个概念，但很多面向对象编程的书籍并没有明确说明这一点。

可以把现实和面向对象中的元素对比一下，建立一个理解面向对象的模型。

现实	人类语言	比喻	面向对象
一类物质	概念	标签	类
不存在实物	抽象概念	给一组标签打包再打个标签	抽象类
一个有清晰边界的物体	实体	用陶土制作了一个杯子	对象的实例化
一个有行动的人	人	拿起了这个杯子	调用者
符合条件的人	契约	有手就能拿起杯子	接口

所以面向对象编程是建立在非常好的心智模型上的，只不过这个模型对于不熟悉西方哲学的人来说过于抽象。对象、实体、类、行为，这些面向对象中的内容和概念早已经被哲学家讨论过数千年，但是在中文的语境中并不常见。

人是通过语言思考的，我们不遗余力地使用自然语言描述事物，面向对象是计算机语言和自然语言的一座桥梁，这座桥梁由哲学连接。对象这个词在不同的领域都被用到，

而且不是巧合：

1. 哲学中的对象概念。
2. 数学（范畴学）中的对象概念。
3. 语言中的宾语。

维特根斯坦的《逻辑哲学论》中对对象、类的阐述和面向对象极为相似，不过这本书非常晦涩。通俗来说：

对象是人认识世界的基本单位，对象由实体和正在发生的事构成。

也就是说对象不是一成不变的，可以由“造物主”自由地设计和组合。当我们在开发一款 XXX 管理系统时，被管理的“物品”被模拟为一个静态的物品，就能看作一个对象。假设我们正在开发仓储管理系统，极端的面向对象者会告诉你将行为放到“货物”这类实体中，这样看起来更加像面向对象的风格，但是他们背离了面向对象的初衷。

虚拟的世界里，静态的对象需要由动态的对象处理构成了一组主客体关系。而对于“上帝”来说，它们都是对象。

熟悉 Java 的程序员可以这样理解，Spring 中的 Bean 是一种对象，在应用启动时就被初始化了，就像上帝造出亚当开始干活儿。而从数据库中提取出来的实体，就像是从仓库中提取出来的“物品”。

如果开发一款游戏，对象貌似都是有生命的。但是对于普通的管理系统来说，真正需要设计的是“货物管理者”、“收银员”这类对象，而“货物”这类实体就应该让它们安安静静地躺在那里。

使用面向对象越久，越会下意识地使用面向对象思考现实，面向对象是程序员进入哲学世界的启蒙课。

理解软件背后的生意

1 需求变化的原因：软件价值金字塔

需求变化是软件工程师最难以容忍的一件事，为了做好软件设计，不得不猜测未来需求的变化方向。猜中了就是“正交分解”，猜不中就是冗余设计。

那么需求变化背后的逻辑是什么呢？

首先我们不得不承认，从客观上讲软件它是有区别于硬件的，为什么叫软件，因为它本身就是能改的，并且修改的成本低于硬件。硬件涉及电路设计、制版、开模等流程，在开发的过程当中，需求变化会带来巨大的成本。这是为

什么软件能够提高效率的原因，因为通过软件搭建在通用计算机平台上，能够很快做出业务应用和实现。通用软件的出现，软件开发和硬件开发分离是信息社会的一个关键节点，所以软件被发明出来就是为了容易改。

但是事物总是矛盾的，容易改的软件相比硬件降低了成本，但软件容易修改的能力被滥用后，给软件工程也带来负担。

对于软件来说，修改并不是没有成本的，只是相对硬件而言小了许多。对软件工程师来说，业务的变化往往会带来困扰，因为它会让软件的架构设计和模型的建立变得非常复杂。

但并不是所有的软件需求变化，我们都不能接受。对于一些软件的交互和界面 UI 样式等这些细节上面的修改不影响主体的业务变化，这种修改是没有任何问题的。我们说的软件需求变化带来的困扰指的是在软件开发过程中随意变更软件的逻辑，让软件的整体性和逻辑性受到了破坏，这是我们不喜欢，不能接受的软件修改的方式之一。

对于专业的产品经理来说，软件的修改是非常谨慎的，因为修改一个地方，可能会影响其他地方。

那么在软件开发和迭代的过程当中，我们可能难以意识到一个小小的修改会影响整个开发、测试、上线等不同的环节，造成项目的延期，这是开发团队人员不喜欢软件被修改的根本原因。

那么怎么应对软件需求的变化呢？

软件价值金字塔模型

如果我们对软件的需求进行分层，我们可以把软件所存在的价值分为 4 层。

最底层是软件所存在的业务价值，或者是通俗来说它是前面提到的“软件的生意”。我们在构建一个点餐软件、构建一个电商软件、构建一个物流软件，那么软件帮我们做的事情就是取代原来传统商业活动中人需要做的事情。提高这些行为的效率，为社会创造更多的价值，这些软件背后的需求就是我们的生意。业务价值，可以看做软件的灵魂。

那么在这层软件需求之上的，是我们软件的架构。软件的架构承载了生意或者商业模式，可以看做软件的骨骼。比如说电商里面就有订单等这些关键的模型，或者一些惯用模式。类比起来就相当于我们人体的一个骨架或者建筑物

的一些承重结构。

还有一些是软件的一些具体的逻辑细节，比如说约束电商系统确认收货时间是多少天。软件这些业务规则，就像人体的血肉一样，丰富了软件。业务规则填充了软件的一些交互逻辑细节，让软件工程师在不修改主体结构的情况下去，改这些逻辑细节，有时候并不是非常困难的，软件工程师对于这类需求的变化也是乐于接受的。

还有一种软件的需求，就是软件的交互方式和 UI 样式，这些就好像动物的皮肤。不具备特别的功能性，而是负责软件的美观性。这些需求的变化，修改成本也是非常低的。

所以我们总结一下软件的价值可以分为 4 层：

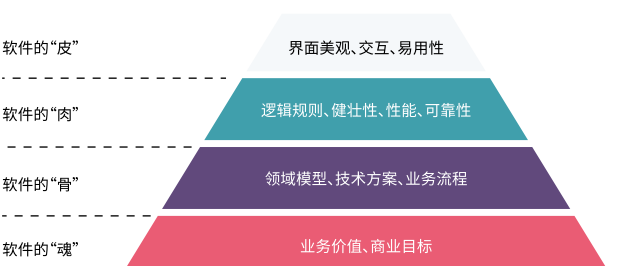


图 1 软件价值金字塔模型

当我们软件的业务架构和业务价值发生翻天覆地的变化时，修改这个软件的难度，会呈指数上升，不亚于重新设计一个软件。

对于创业公司来说，他们的业务架构和生意，或者说它的商业模式还不确定，还在探索当中。对于这样的业务来说，他们的需求几乎每天都会发生变化，因为他们的生意会变，一旦生意会变，“上层建筑就会变化”。

对于成熟的公司来说，软件是这个公司业务流程的沉淀，业务流程可能不会发生特别大的变化，比如说银行、保险或者财会，这些特定的业务流程基本上已经形成了行业的规范或者标准。他们的变化情况不会特别大，那么软件的架构也就不容易受到破坏，重大的业务需求变化就会非常少。

理解软件背后的生意

软件价值模型给了软件工程师两点启示。

一个能够在市场上存活的软件，一定有它背后的业务逻辑和业务价值。那么我们从底层出发，找到了一个软件的业

务价值，也就是它的生意，我们就可以快速地理解软件的架构。

其次，我们可以真正地挖掘出业务分析师或产品经理希望的业务。基于软件价值模型，软件背后的逻辑和生意总是存在的，但是产品经理不一定能够用自己的语言或合适的方式讲给软件工程师。

对于软件工程师来说，只有两个选择。要么给自己的软件的架构设计提供足够的灵活性，这也是很多软件设计思想提倡的。但它背后的代价很明显，我们需要留出 “冗余设计”，在特定的环境下，软件的竞争力被削弱。一个有灵活或者弹性的软件架构，背后是付出一定的代价，但往往我们没有意识到这一点。

另外一个选择就是真正地理解软件背后的生意，通过软件价值模型的启示从变化中找到不变。因此我们不得不将视野从软件本身返回到软件承载的业务上，为了理解这些业务我们又需要追溯回到这些业务服务的商业目标中。

那么软件工程师理解软件的路径为：理解商业→理解业务→理解软件产品和信息系统。

2 理解商业模式

如果我们理解了软件背后的生意，可以更加从容地设计软件。更为重要的是，和需求提出者的交流更加容易，除非需求提出者也并不熟悉正在设计的软件背后承载的商业目标。

分析一个企业的商业模型方法非常多，下面介绍比较常见也比较简单的方法——商业模式画布。

顾名思义，商业模式画布是一种描述企业商业模式的模型，最早来源于亚历山大 • 奥斯特瓦德的《商业模式新生代》一书。其主要的思想是，商业模式不应该由几百页的商业策划书来描述，而是应该由一页纸就能清晰地呈现。根据思维经济性原则，无法清晰表述的商业模式其价值也值得怀疑。

商业模式画布，包含 9 个模块，可以呈现在一张画布上。如下图所示：

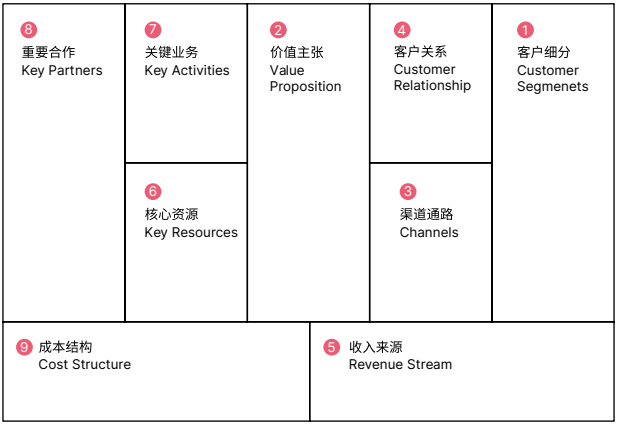


图 2 商业模式画布

编写商业模式画布实际上是需要回答 9 个问题，弄明白至少这 9 个问题，才能知道对未来相关的商业设想是否靠谱。如果投资人看这份商业模式画布，才能快速知道这笔生意是否能赚钱。

使用商业模式画布来研究商业模式的案例非常多，这些案例的研究材料容易找，我以拼多多为例并结合 IT 视角来看商业模式对信息系统的影响。

很多人可能和我一样对拼多多有一些疑惑，为什么在电商

格局已经充分竞争后，依然还有崛起的机会？我们不妨用商业画布来分析一下。

1. 客户细分（CS，Customer Segments）

拼多多的客户是什么？

如果不加以区分客户和用户，我们很容易得到拼多多的客户是普通的消费者。实际上从财务的角度，如果消费者的每一笔消费都算在拼多多的收入中，那么拼多多需要支付巨额的增值税，消费者不是拼多多的客户。

拼多多的业务旨在帮助小微企业、农户和个人快速开设店铺，并从中获得佣金。因此，在客户这一侧和淘宝网的初期发展阶段相比，拼多多并没有太大的差别。在某种程度上，由于天猫的存在和战略因素，阿里电商在这个领域相当薄弱。

2. 价值主张（VP，Value Propositions）

关于价值主张这部分我一直比较疑惑，拼多多到底能提供什么新的价值？

在一份名为《“电商黑马”拼多多的商业模式探析》[1]的报告中，提到了拼多多价值主张为“免去诸多中间环节，实现 C2M 模式，提供物有所值的商品和互动式购物体验的 “新电子商务”平台。C2M 为（Customer-to-Manufacturer，用户直连制造）但是这个模式并不新鲜，戴尔、玫琳凯等直销公司都是这种模式。

一些分析者将拼多多的模式总结为物找人。通过拼单的方式，先定义物品，再通过社交媒体找到需要的目标群体。让“社会化消费”发挥作用。

从价值主张上说，拼多多的价值和其他主流、非主流电商的差异并不大。

3. 渠道通路（CH，Channels）

在价值主张上，各种电商平台差距非常小，无非都是“消除中间商，降低流通成本”。但是在细分领域，渠道通路的竞争非常明显，甚至有些电商平台将自己的电商属性隐藏了起来。

---

1 邱柳方 .“电商黑马”拼多多的商业模式探析 [J]. 国际商务财会 ,2021(11):34-37+41.

例如，以社交抹茶美妆、小红书、Keep 这些产品的电商属性非常弱，实际上是通过社交渠道强化了电商的渠道能力。拼多多的渠道是建立在一种病毒营销的模式上的，俗称“人传人”。

4. 客户关系（CR，Customer Relationships）

拼多多的店铺分为了几类[2]，不过最终还是可以分为专业类（企业或个体工商户店）和普通类。专业类的客户为具有一定资本的经销商，需要缴纳保证金以及工商登记材料，普通类的无需保证金和工商材料即可开店，而正是普通类占据了主要的店铺类型。

5. 收入来源（RS，Revenue Streams）

根据财报显示（2021 年数据），拼多多的收入来源为在线市场服务和少量的自营商品销售，财务来源并没有特殊的地方，主要还是来源于店铺佣金。

6. 核心资源（KR，Key Resources）

在商业模式和收入来源都没有特殊的情况下，拼多多的核心资源是什么呢？在一些商业分析中，将拼多多的核心资

2 参考拼多多商家入驻说明 https://ims.pinduoduo.com/qualifications

源归结为用户流量。截至 2021 年第一个季度结束，拼多多年活跃买家数达 8.238 亿[3]，那么这些买家是从哪里来的呢?

除了前面说的“人传人”的基础上，拼多多借助了微信渠道，可以说这是拼多多的核心资源。

7. 关键业务（KA，Key Activities）

拼多多的关键业务是市场活动和供应链管理。

8. 重要合作（KP，Key Partnership）

拼多多的合作伙伴有：腾讯微信、物流企业、电视媒体。将商家排除在外的原因是，商家已经作为了客户存在。

换句话说，商家是赚消费者的钱，拼多多是赚商家的钱。由腾讯微信提供渠道，通过特有的病毒营销获得用户流量，并将流量转化为商家的客源，可以看做是微信的用户群体资源在电商领域的变现。

9. 成本结构（CR，Cost Structure）

拼多多的成本结构主要是市场推广费用，其次是管理费用

3 数据来源 https://view.inews.qq.com/k/20220527A0AG7300

和研发费用。

根据商业模式画布分析，拼多多的商业模式主要是以独特的营销推广为基础，为小微企业和个体农商户促成交易。在交易渠道上借助了微信腾出的渠道真空（微信渠道对淘宝不开放，拼多多和京东无太多竞争关系，腾讯为拼多多的第二大股东）。从其营收结构主要为在线市场佣金收入反映了这一点，成本结构上以营销费用为主也进一步佐证。

3 理解业务

通过商业模式画布可以理解企业的商业模式，弄明白在企业的业务中谁是客户，收入从哪里来，合作伙伴是谁等。不过，商业模式画布没有将企业的内部运转结构打开，一个企业需要运转起来，需要各个部分之间的通力合作，并和用户产生交互。

要明白地表达企业内部各方的合作情况，业务服务蓝图可以帮上忙。不过请注意在使用服务蓝图时，存在一些争议。例如，是否应该将 IT 系统参与到服务蓝图中表达？这里存在两种流派和方法：一种是使用两张图来表达，这样能看清楚企业引入 IT 系统前后的变化，区分为业务服务蓝图和

应用服务蓝图；另外一种流派是将其绘制到一张图上，统称为服务蓝图。

由于在这里区分了商业模型和业务模型以及加入 IT 系统之后的形态，应用服务蓝图更多地是关注待分析的 IT 信息系统。

业务服务蓝图本质上是一种流程图，表达商业中各个参与的主体（参与业务的各个部门可以看做业务主体）之间的往来，通过多个泳道来表达参与的业务主体。服务蓝图在“服务设计”这个概念下可以看做是用户旅程的延伸。在服务蓝图中，不仅包含水平方向的客户服务过程，还包括垂直方向各个业务主体之间的合作关系，描述服务前、中、后台构成的全景图。

以蔬菜配送为案例，我们来看下服务蓝图的应用。我还原了一个真实的小本买卖——某批发市场的食材配送公司的业务形态。

餐厅老板往往（或者他的员工）需要自己整理一些食材清单，然后通过电话下订单给某家配送公司的客户经理，客户经理生成食材订单后，构成简易合同，随即让仓配部进行配送。好在我虚拟的这家配送公司自建了物流，出库和配送是一

个部门，否则还需要新的契约来满足配货出库和物流之间的关系。

当餐厅收到货物后，食材配送公司一般会出具一张清单，餐厅清点完成后需要签字盖章。这张单据往往会被用来作为处理纠纷的关键单据，而纠纷的发生会比想象中多非常多，可以说商业就是处理纠纷的艺术。

在具有固定合作关系的商业主体之间，往往都不会结算现款，都有一定的结算周期，通过结算单来完成结算，随之进行支付，某些情况下还会出现抵扣。

4 理解软件产品和信息系统

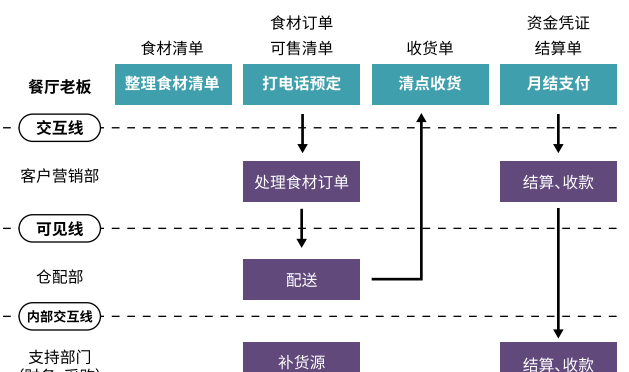


图 3 业务服务蓝图模板

如果理解了一个企业的商业模式，以及支持了支持商业模式的业务，再来看构建在两者之上的信息系统或者软件就容易很多。

我们可以做一个思维实验，一家主营食材配送的企业，它的客户是餐厅老板，公司的主要业务为每日清晨为各个餐厅配送食材。毫无疑问在现代化的社会，信息系统必然是存在的。这家公司使用了微信作为渠道，建立了小程序、H5 应用建立了食材订购的应用，同时又为承担配送工作的员工开发了具有送货、打单功能的安卓原生 APP，以及财务核算的 Web 应用。

假定在某天系统故障了，但是配送的工作不能停下来，这是事关商誉的事情。如果因为一次无理由的断供，会导致相关的餐厅无法营业，营业中断带来的损失远远超过当天货物的价值。于是，公司领导无论如何都需要想办法将食材送到客户手中。在信息系统无法使用时，它们可能的做法是，从数据库导出备份的数据，打印出来，人工通知到客户。我们会发现，对于这类软件，完全可以使用纸和笔进行延续之前的业务。

这种思维实验，也是也是在软件设计时常用的方法。当业

务复杂，产品经理或者业务人员无法描述清楚，我们可以将“电”断掉，思考如何通过纸和笔来完成软件设计。

断电法，可以将系统中晦涩难懂的概念在现实中找到可以被理解的物品。比如，用户这个概念比较抽象。餐厅老板或者经理可以作为用户在配送平台上下单，如果断电了，那么用户在现实中是什么呢？可能是食材配送老板大脑中的一段记忆，也可能是写在笔记本上的一段记录。

可能会发生这样的场景：

> 烤鸭店张老板老板需要预定 100 斤大葱，打电话给食材配送的王老板。
> 张老板：老王啊（电话接通后）。
> 王老板：原来是老张啊，今天需要什么货呢。（根据电话号码、声音定位到这个用户）
> 张老板：我需要 100 斤大葱，上午帮我送过来。（下单、填写配送时间）
> 王老板：收到等下我就去装（确认订单，备货）
> ……

如果用现实中的行为来扮演 IT 系统的逻辑，可以降低认知

难度，更容易理解业务。前面的业务服务蓝图可以看做是断电后的纸笔推演，接下来我们需要将 IT 系统引入整个商业体系。

对于行业软件来说，软件技术本身并不复杂，它更像是一个“机器人”（软件），难在如何教会这个机器人像专业人士一样工作（领域知识）。

有软件参与的服务蓝图，我们可以叫做应用服务蓝图。意味着，我们找到了“电子帮手”，电子帮手会参与到业务体系中。

在应用服务蓝图中，我们可以将 IT 系统看做新的主体，这个主体可以帮助业务完成更高效的工作。在前面食材配送的例子中，微食材（这里设定出来的产品名称）会参与到业务中，作为虚拟的主体存在。

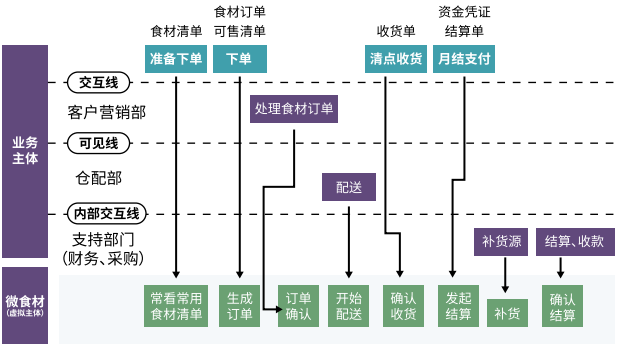

食材清单
食材订单
可售清单
收货单
资金凭证
结算单
准备下单
下单
清点收货
月结支付
业务主体
交互线
客户营销部
可见线
处理食材订单
仓配部
配送
内部交互线
支持部门
(财务、采购)
补货源
结算、收款
微食材
(虚拟主体)
常看常用食材清单
生成订单
订单确认
开始配送
确认收货
发起结算
补货
确认结算

图 4 应用服务蓝图

透过领域建模看软件的骨相

通过对软件业务模型的分析，对业务领域进行建模就能获得软件的逻辑结构。获得了软件的逻辑结构就能进一步指导服务划分、数据库设计、API 设计等技术实践。没有领域模型设计的软件，工程师往往会过多地关注到技术问题上，而忽视了产品设计和业务的目标。

领域建模对于商业软件来说是非常重要的一环，也是工程师消化行业领域知识的重要方法。

1 认识领域驱动设计

领域驱动设计 (DDD)[1] 是 Eric Evans 提出的一种软件设计方法和思想，主要解决业务系统的设计和建模。DDD 有大量难以理解的概念，尤其是翻译的原因，某些词汇非常晦涩，例如：模型、限界上下文、聚合、实体、值对象等。

实际上 DDD 的概念和逻辑本身并不复杂，很多概念和名词是为了解决一些特定的问题才引入的，并和面向对象思想兼容，可以说 DDD 也是面向对象思想中的一个子集。如果遵从奥卡姆剃刀的原则，“如无必要，勿增实体”，我们先把 DDD 这些概念丢开，从一个案例出发，在必要的时候将这些概念引入。

从纸和笔思考 IT 系统的工作逻辑

让我真正对计算机软件和建模有了更深入的认识是在一家餐厅吃饭的时候。数年以前，我还在一家创业公司负责餐饮软件的服务器端的开发工作，因为工作的原因，外出就餐时常都会对餐厅的点餐系统仔细观察，以便于改进我们自己产品的设计。

1 参考图书：《领域驱动设计——软件核心复杂性应对之道》
https://book.douban.com/subject/26819666

一次偶然的情况，我们就餐的餐厅停电了，所幸是在白天，对我们的就餐并没有什么影响。我突然很好奇这家店，在收银系统无法工作的情况下怎么让业务继续运转，因此我饶有趣味地等待服务员来接受我们的点单。

故事的发展并没有超出预期，服务员拿了纸和笔，顺利地完成了点餐，并将复写纸复写的底单麻溜地撕下来交给了后厨。

我这时候才回过神来，软件工程师并没有创造新的东西，只不过是数字世界的砖瓦工，计算机系统中合乎逻辑的过程，停电后人肉使用纸和笔一样合乎逻辑。

合乎现实世界的逻辑和规则，使用鼠标和键盘代替纸和笔，就是软件设计的基本逻辑。如果我们只是关注于对数据库的增、删、改、查（CRUD），实际上没有对业务进行正确地识别，这是导致代码组织混乱的根本原因。

会计、餐饮、购物、人员管理、仓储，这些都是各个领域实实在在发生的事情，分析业务逻辑，从中找出固定的模式，抽象成计算机系统中对象并存储。这就是 DDD 和面向对

象思想中软件开发的一般过程。

你可能会想，我们平时不就是这样做的吗？

现实是，我们往往马上关注到数据库的设计上，想当然地设计出一些数据库表，然后着手于界面、网络请求、如何操作数据库上，业务逻辑被封装到一个叫做 Service 对象上或散落在其他地方，然后基于对数据的操作。

图 1 面向数据库的编程方法

一般来说这种方法也没有大的问题，甚至工作得很好，Martin Fowler 将这种方法称作为事务脚本（Transaction Script）。还有其他的设计模式，将用户界面、业务逻辑、数据存储作为一个“模块”，可以实现用户拖拽就可以实现简单的编程，.net、VF 曾经提供过这种设计模式，这种设计模式叫做 SMART UI。

这种模式有一些好处。

- 非常直观，开发人员学习完编程基础知识和数据库 CRUD 操作之后就可以开发。
- 效率高，能短时间完成应用开发。
- 模块之间非常独立。

麻烦在于，当业务复杂后，这种模式会带来一些问题。

虽然最终都是对数据库的修改，但是中间存在大量的业务逻辑，并没有得到良好的封装。客人退菜，并不是将订单中的菜品移除这么简单。需要将订单的总额重新计算，以及需要通知后厨尝试撤回在做的菜。

由于没有真正的“订单”对象负责执行相关的业务逻辑，初学者程序员擅自修改数据片段，破坏了整体业务逻辑。Service 上的一个方法直接修改了数据库，业务逻辑的完整性完全取决于程序员对系统的了解。

我们在各个餐厅交流的时候发现，这不是一个 IT 系统的问题。在某些餐厅，所有的服务员都可以收银，即使用纸和笔，

但收银员划掉菜品后没有更新小计，另外的服务员结账时会出现错误。因此，餐厅约定修改菜品后必须更新订单总价。

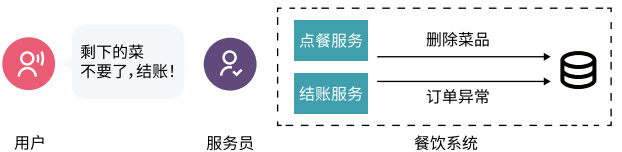


图 2 一致性问题

我们吸收这个业务逻辑到 IT 系统中来，并意识到系统中这里有一些隐藏的模型：

- 订单。
- 菜品。

我们决定，抽象出订单、菜品的对象，菜品不应该被直接修改，而是通过订单才能修改，无论任何情况，菜品的状态变化都通过订单来完成。

复杂系统的状态被清晰地定义出来了， Service 承担处理各个应用场景的差异，模型对象处理一致的业务逻辑。

在接触 Eric Evans 的 DDD 概念之前，我们没有找到这种开发模式的名字，暂时称作为朴素模型驱动开发。

图 3 朴素模型驱动开发

模型和领域模型

从上面的例子中，模型是能够表达系统业务逻辑和状态的对象。

我们知道要想做好一个可持续维护的 IT 系统，实际上需要对业务进行充分的抽象，找出这些隐藏的模型，并搬到系统中来。如果发生在餐厅的所有事物，都要能在系统中找到对应的对象，那么这个系统的业务逻辑就非常完备。

现实世界中的业务逻辑，在 IT 系统业务分析时，适合某个行业和领域相关的，所以又叫做领域。

领域，指特定行业或者场景下的业务逻辑。

DDD 中的模型是指反映 IT 系统的业务逻辑和状态的对象，是从具体业务（领域）中提取出来的，因此又叫做领域模型。

通过对实际业务出发，而非马上关注数据库、程序设计。通过识别出固定的模式，并将这些业务逻辑的承载者抽象到一个模型上。这个模型负责处理业务逻辑，并表达当前的系统状态。这个过程就是领域驱动设计。

我从这里面学到了什么呢？

我们做的计算机系统，实际上是替代了现实世界中的一些操作。按照面向对象设计的话，我们的系统是一个电子餐厅。现实餐厅中的实体，应该对应到我们的系统中去，用于承载业务，例如收银员、顾客、厨师、餐桌、菜品，这些虚拟的实体表达了系统的状态，在某种程度上就能指代系统，这就是模型，如果找到了这些元素，就很容易设计出软件。

后来，如果我什么业务逻辑想不清楚，我就会把电断掉，假装自己是服务员，用纸和笔走一遍业务流程。

分析业务，设计领域模型，编写代码。这就是领域驱动设计的基本过程。领域模型设计好后（通俗来说，领域模型就是一些 UML 类图），可以指导后续的工作：

- 指导数据库设计。
- 指导模块分包和代码设计。
- 指导 RESTful API 设计。
- 指导事务策略
- 指导权限设计。
- 指导微服务划分（有必要的情况）。

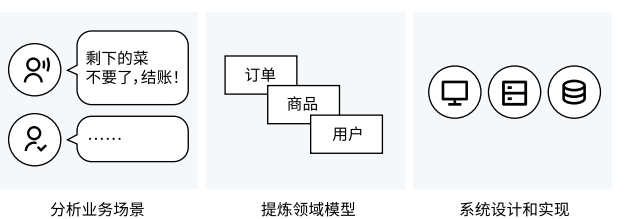


图 4 软件设计过程

在我们之前的例子中，收银员需要负责处理收银的操作，同时表达这个餐厅有收银员这样的一个状态。收银员收到

钱并记录到账本中，账本负责处理记录钱的业务逻辑，同时表达系统中有多少钱的状态。

技术和业务复杂度

我们进行业务系统开发时，大多数人都会认同一个观点：将业务和模型设计清楚之后，开发起来会容易很多。

但是实际开发过程中，我们既要分析业务，也要处理一些技术细节，例如：如何响应表单提交、如何存储到数据库、事务该怎么处理等。

使用领域驱动设计还有一个好处，我们可以通过隔离这些技术细节，先进行业务逻辑建模，然后再完成技术实现，因为业务模型已经建立，技术细节无非就是响应用户操作和持久化模型。

我们可以把系统复杂的问题分为两类：

- 技术复杂度。软件设计中和技术实现相关的问题，例如处理用户输入，持久化模型，处理网络通信等。
- 业务复杂度。软件设计中和业务逻辑相关的问题，

例如为订单添加商品，需要计算订单总价，应用折扣规则等。

图 5 技术复杂度和业务复杂度

当我们分析业务并建模时，不要关注技术实现，因为会带来极大的干扰。和上一章聊到的断电法理解业务一样，就是在这个过程把“电”断掉，技术复杂度中的用户交互想象成人工交谈，持久化想象成用纸和笔记录。

DDD 还强调，业务建模应该充分和业务专家在一起，不应该只是实现软件的工程师自嗨。业务专家是一个虚拟的角色，有可能是一线业务人员、项目经理或者软件工程师。

由于和业务专家一起完成建模，因此尽量不要选用非常专业的绘图工具和使用技术语言。可以看出 DDD 只是一种

建模思想，并没有规定使用的具体工具。我这里使用 PPT 的线条和形状，用 E-R 的方式表达领域模型，如果大家都很熟悉 UML 也是可以的。甚至实际工作中，我们大量使用便利贴和白板完成建模工作，好处是一屋子的人方便参与到共创的工作坊中来。

这个建模过程可以是技术人员和业务专家一起讨论出来，也可以使用“事件风暴”这类工作坊的方式完成。这个过程非常重要，DDD 把这个过程称作协作设计。通过协作设计，我们得到了领域模型（这里以简单的图表表示，也可以用 UML）。

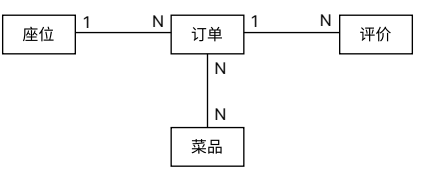


图 6 领域模型 v1

上图是我们通过业务分析得到的一个非常基本的领域模型，我们的点餐系统中，会有座位、订单、菜品、评价几个模型。一个座位可以关联多个订单，每个订单可以有多个菜品和

评价。同时，菜品也会被不同的订单使用。

上下文、二义性、统一语言

我们用领域模型驱动的方式开发软件系统，相对于事务脚本的方式，已经容易和清晰很多了，但还是有一些问题。

有一天，市场告诉我们，这个系统会有一个逻辑问题。就是系统中菜品被删除，订单也不能查看。在我们之前的认知里面，订单和菜品是一个多对多的关系，菜品都不存在了，这个订单还有什么用。

这里的菜品存在了致命的二义性！！！这里的菜品实际上有两个含义：

- 在订单中，表达这个消费项的记录，也就是订单项。例如，5 号桌消费了鱼香肉丝一份。
- 在菜品管理中，价格为 30 元的鱼香肉丝，包含菜单图片、文字描述，以及折扣信息。

菜品管理中的菜品下架后，不应该产生新的订单，同时也不应该对订单中的菜品造成任何影响。这些问题是因为，技术专家和业务专家的语言没有统一， DDD 一书提到了这个问题，统一语言是实现良好领域模型的前提，因此应

该“大声地建模”。我在参与这个过程目睹过大量有意义的争吵，正是这些争吵让领域模型变得越来越清晰。

这个过程叫做统一语言。

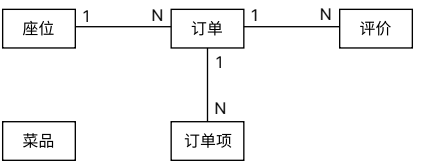


图 7 领域模型 v2

和现实生活中一样，产生二义性的原因是我们的对话发生在不同的上下文中，我们在谈一个概念必须在确定的上下文中才有意义。在不同的场景下，即使使用的词汇相同，但是业务逻辑本质都是不同的。想象一下，发生在《武林外传》中同福客栈的几段对话。

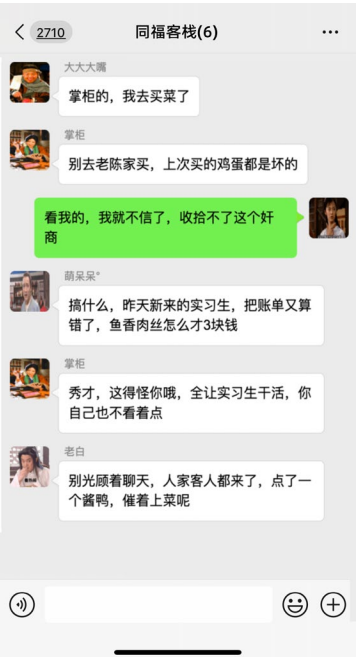


图 8 关于上下文的对话

这段对话中实际上有三个上下文，这里的“菜”这个词出现了三次，但是实际上业务含义完全不同。

- 大嘴说去买菜，这里的菜应该被理解为食材，如果

掌柜对这个菜进行管理，应该具有采购者、名称、采购商家、采购价等属性。

- 秀才说实习生把账单中的菜算错了价格，秀才需要对账单进行管理，这里的菜应该指的账单科目，现实中一般是会计科目。
- 老白的客人点了一只酱鸭，但老白关注的是订单下的订单项。订单项包含价格、数量、小计、折扣等属性信息。

实际上，还有一个隐藏的模型——上架中商品。掌柜需要添加菜品到菜单中，客人才能点，这个商品就是我们平时一般概念上的商品。

我们把语言再次统一，得到新的模型。

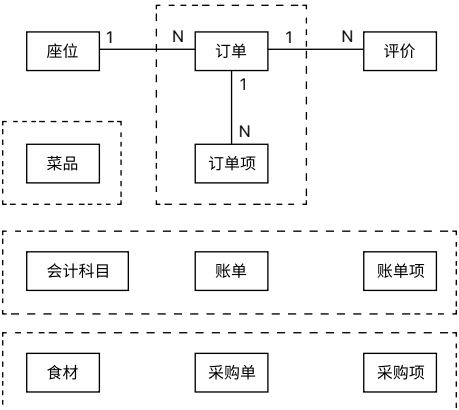


图 9 领域模型 v3

在被虚线框起来的四个区域中，我们都可以使用“菜品”这个词汇（尽量不要这么做），但是大家都知道“菜品”具有不同的含义。这些区域被称为上下文。当然，上下文不仅仅是由二义性决定的，也可能是由完全不相关的概念产生的。例如，在订单和座位之间实际上并没有强烈的关联，当我们讨论座位时，我们完全在谈论其他事情，因此座位应该被视为单独的上下文。

识别上下文的边界是 DDD 中最难的一部分，同时上下文

边界是由业务变化动态变化的，我们把识别出边界的上下文叫做限界上下文（Bounded Context）。限界上下文是一个非常有用的工具，限界上下文可以帮助我们识别出模型的边界，并做适当的拆分。

限界上下文的识别难以有一个明确的准则。上下文的边界非常模糊，需要有经验的工程师并充分讨论才能得到一个好的设计。同时需要注意，限界上下文的划分没有对错，只有是否合适。跨限界上下文之间模型的关联有本质的不同，我们用虚线标出，后面会详细讨论这种区别。

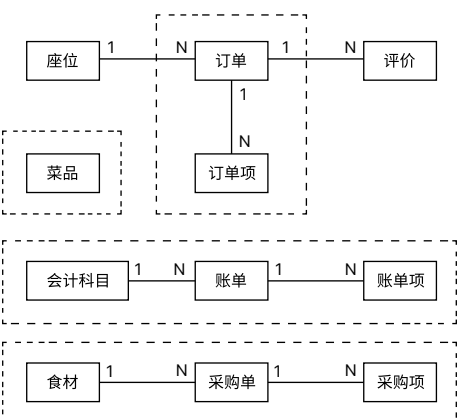


图 10 领域模型 v4

使用上下文之后，带来另外一个收获。模型之间本质上没有多对多关系，如果有，说明存在一个隐含的成员关系，这个关系没有被充分分析出来，对后期的开发会造成非常大的困扰。

聚合根、实体、值对象

上面的模型，尤其是解决二义性这个问题之后，已经能在实际开发中很好地使用了。不过还是会有一些问题没有解决，实际开发中，每种模型的身份可能不太一样，订单项必须依赖订单的存在而存在，如果能在领域模型图中体现出来就更好了。

举个例子来说，当我们删除订单时，订单项应该一起删除，订单项的存在必须依赖于订单的存在。这样业务逻辑是一致的和完整的，游离的订单项对我们来说没有意义，除非有特殊的业务需求存在。

为了解决这个问题，对待模型就不再一视同仁了。我们将相关性极强的领域模型放到一起考虑，数据的一致性必须解决，同时生命周期也需要保持同步，我们把这个集合叫做聚合。

聚合中需要选择一个代表负责和全局通信，类似于一个部门的接口人，这样就能确保数据保持一致。我们把这个模型叫做聚合根，聚合根充当一组领域模型领航员的角色。当一个聚合业务足够简单时，聚合有可能只由一个模型组成，这个模型就是聚合根，常见的就是配置、日志相关的。

在聚合中，无论是否是聚合根，对于有自己的身份（ID）的模型，我们都可以叫做实体。

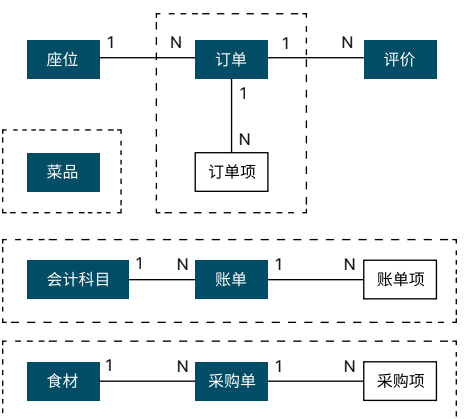


图 11 领域模型 v5

我们把这个图完善一下，聚合之间也是用虚线连接，为聚合根标上更深一点的颜色。识别聚合根需要一些技巧。

- 聚合根本质上也是实体，同属于领域模型，用于承载业务逻辑和系统状态。
- 实体的生命周期依附于聚合根，聚合根删除实体应该也需要被删除，保持系统一致性，避免游离的脏数据。
- 聚合根负责和其他聚合通信，因此聚合根往往具有一个全局唯一标识。例如，订单有订单 ID 和订单号，订单号为全局业务标识，订单 ID 为聚合内关联使用。聚合外使用订单号进行关联应用。

还有一类特殊的模型，这类模型只负责承载一组字段值的表达，没有自己的身份。在我们饭店的例子中，如果需要对账单支持多国货币，我们将纯数字的 price 字段修为 Price 类型。

```
public class Price {
    private String currency;
    private BigDecimal value;
    public Price(String currency, BigDecimal value) {
        this.currency = currency;
        this.value = value;
```

```
    }
}
```

价格这个模型，没有自己的生命周期，一旦被创建出来就无须修改，因为修改就改变了这个值本身。所以我们会给这类的对象一个构造方法，然后去除掉所有的 setter 方法。

我们把没有自己生命周期的模型，仅用来呈现多个字段的值的模型和对象，称作为**值对象**。

值对象一开始不是很容易理解，但是理解之后会让系统设计非常清晰。“地址”是一个显著的值对象。当订单发货后，

实体	值对象
有 ID 标识	无 ID 标识
有自己的生命周期	一经创建就不要修改
可以对实体进行管理	使用新的值对象替换
使用 ID 进行相等性比较	使用属性进行相等性比较

地址中的某一个属性不应该被单独修改，因为被修改之后这个“地址”就不再是刚刚那个“地址”，判断地址是否相同我们会使用它的具体值：省、市、地、街道等。

最简单的理解，值对象就是“属性包”，就是一些自己定义的通用拓展类型，持久化时展开到数据库表或者存为 JSON 字符串。

值对象是相对于实体而言的，对比如下。

另外值得一提的是，一个模型被作为值对象还是实体看待不是一成不变的，某些情况下需要作为实体设计，但是在另外的条件下却最好作为值对象设计。

地址，在一个大型系统充满了二义性。

- 作为订单中的收货地址时，无须进行管理，只需要表达街道、门牌号等信息，应该作为值对象设计。为了避免歧义，可以重新命名为收货地址。
- 在进行系统地理位置信息管理时，其具有自身的生命周期，因此应将其作为实体进行设计，并将其重命名为“系统地址”。

- 用户可以根据 ID 管理添加的自定义地址。这些地址应该视为实体，并重命名为“用户地址”。

我们使用浅色表达值对象以便区别于聚合根和实体，更新后的模型图如下：

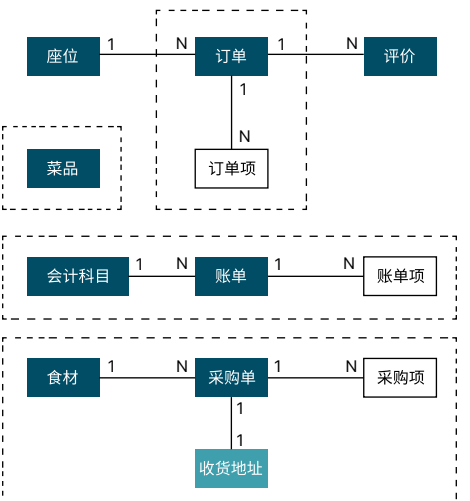


图 12 领域模型 v6

虽然我们使用 E-R 的方式描述模型和模型之间的关系，但是这个 E-R 图使用了颜色（如果是黑白印刷的纸质版可能看不到具体的颜色，可以自行体会）、虚线，已经

和传统的 E-R 图大不相同，把这种图暂时叫做 CE-R 图（Classified Entity Relationship）。DDD 没有规定如何画图，你可以使用其他任何画图的方法表达领域模型，如果需要严谨一点可以采用 UML 的类图绘制（推荐使用 UML 绘制领域模型）。

2 建模方法元模型

在《领域驱动设计》一书中，Eric 阐述了领域驱动设计的重要性和一些基本实践，但并没有给出具体的建模过程方法。这为架构师提供了很大的发挥空间，可以使用各种建模方法，例如事件风暴、四色原型等。

于是有一些朋友会产生疑惑，这些建模方法背后的逻辑是什么呢，它们有没有什么共通之处？这里和大家一起探讨软件建模过程方法的基本逻辑，以及如何设计一套简单的建模过程。

目前进行领域建模方法使用得最多的是事件风暴。事件风暴[2]的发明人是 Alberto Brandolini，它来源于 Gamestorming，通过工作坊的方式将领域专家和技术专

2 Event storming 网站 https://www.eventstorming.com/

家拉到一起，进行建模。事件风暴非常有意思的地方在于，它先从事件开始分析，捕获事件。然后分析引发事件的行为和执行者，从这些信息中寻找领域模型，最终进一步展开。

Event Storming 的逻辑是什么？为什么需要先从事件开始分析？这是事件风暴工作坊中遇到过最多的问题。

我带着这些问题请教了很多专家，甚至发送了邮件给 Alberto Brandolini，有幸得到回复。根据 Alberto Brandolini 的理解，他认为系统中事件是一种容易寻找到的元素，通过头脑风暴，容易打开局面，仅此而已。

带着同样的问题，我分析了几种建模方法。

系统词汇法（OOA）

系统词汇法就是面向对象分析方法。这种面向对象建模的方法比较原始和直接，直接通过经验提取领域模型，就是简单的面向对象分析方法。其操作过程简化如下：

1. 首先，从需求陈述中找出所有的名词，将它们作为“类—对象”的初步候选者。去掉不正确和不必要

的对象（不相关的、外部的和模糊的对象），作出合理地抽象。

2. 为上一步的模型作出定义，构建数据字典，描述对象的范围、成员和使用场景。
3. 聚合，把业务一致性强的对象聚合到一起。
4. 使用合适的关联方式设计对象之间的关系。

系统词汇法建模的优点和缺点都比较明显。优点是没有过多的建模过程，对于简单的系统有经验的架构师马上就能观察出合适模型。相应地，缺点也很明确，没有对业务充分分析，直接得到模型，容易错误理解业务和过度设计模型。

用例分析法

用例模型是一种需求分析模型，是需求分析后的一种输出物，通过对用例再加工可以得到我们的领域模型。1992 年，Jacobson 中提出了用例的概念和可视化的表示方法用例图。

用例（UseCase）是对一个活动者使用系统的一项功能时所进行的交互过程的一个文字描述。

用例由参与者、关系、用例三个基本元素构成，用例图的基本表示方法如下：

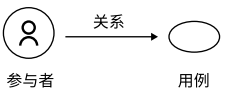


图 13 用例图

通过用例图来提取领域模型的方法如下：

1. 梳理用例的用词，统一用例中所有的概念，避免混淆。
2. 从用例中提取出名词，作为备选模型，这个时候不区分对象或者属性。
3. 找动词，通过动词和用例关系分析模型之间的关联关系，比如：用户结账用例，会触发累积积分的用例，说明用户账户和积分有关联。
4. 对名词进行抽象、展开，把用例中作为属性的名词归纳到对象中，补充为完整模型。

由于用例图从不同的参与者出发，因此非常适合表达业务行为，可以避免错误的复用。在很长一段时间里，很多软

件架构师都依赖用例图来建立模型。用例分析法的特点是不容易遗漏，但缺点是由于名词的二义性，往往会设计出一些过度复用的模型。

四色建模法

四色建模法的思路和用例略有不同，它的理念是：

> “任何业务事件都会以某种数据的形式留下足迹”。

四色建模法其实是以数据驱动，通过挑选一些关键数据（类似于办事过程中的存根），来还原整个业务流程。然后基于这个线索，找出时标性对象（moment-interval）、实体（party/place/thing）、角色（role）、描述对象（description）。

1. 以满足业务运营的需要为原则，寻找需要追溯的业务事件。
2. 基于这些业务事件发生的存根，建立时标性对象，比如订单 → 发货单 → 提货单等。
3. 基于时标性对象反推相应的实体，比如订单 → 商品，发货单 → 货物和发货员。

4. 最后把描述的信息放入描述对象，附着在需要补充的对象上。
5. 梳理为最终的模型。

四色建模法由 Peter Coad 提出[3]，其实并不是一种非常主流的建模方式，其原因为存根和时标性对象在很多业务系统中并不容易找到。

事件风暴

事件风暴相对其他的建模方法非常独特，所以放到最后来说。简单来说，它的思路是：

> “事件是系统状态变化的关键帧”。

事件是比较容易找到的，它的建模过程有点逆向思维。

1. 寻找事件。事件（Event）是系统状态发生的某种客观现象，事件格式参考“XXX 已 YYY”，比如“订单已创建”。

---

3 关于四色建模法来源见文章 https://www.infoq.cn/article/xh-four-color-modeling

2. 寻找命令和执行者。命令可以类比于 UML 分析中的业务用例，是某个场景中领域事件的触发动作，执行者是命令的发生者。
3. 寻找模型。为了在这个阶段保持和业务专家的良好沟通，寻找“领域名词”。
4. 设计聚合。对领域名词进行建模，获得模型的组合、关系等信息。
5. 划分限界上下文。对模型进行划分，在战略上将模型分为多个上下文。

事件风暴在获得模型的深刻性上具有优势，但是在操作上更为困难。另外由于它不从用例出发，和四色建模一样，可能有一些遗漏，所以对工作坊的主持人要求较高。

元模型

元模型是关于模型的模型，可以用于建立建模方法的模型。在计算机领域中，关于元模型的研究资料和书籍较少，因为它涉及到更高的抽象层次，难以理解。在有限的资料中，《本体元建模理论与方法及其应用》一书介绍了如何建立软件建模的元模型。

通过对这些建模方法进行分析，发现他们有一些共同特点。都是围绕着参与者、行为、事件、名词这几个元素展开的，通过对这些方法的总结，我们可以尝试建立一个简单的建模方法元模型，为建模方法的改进提供依据。

其实，面向对象中的模型是现实世界在计算机系统中的一种比喻，类似的比喻还有函数式等其他编程范式。对于现实世界的分析，我们可以使用认识论建立一个非常简单的模型。

主体 + 行为 + 客体 = 现象

主体：主体是有认识能力和实践能力的人，或者是在社会实践中认识世界、改造世界的人。

客体：客体是实践和认识活动所指向的对象，是存在于主体之外的客观事物。

	系统词汇法(OOA)	用例分析法	四色建模法	事件风暴
主体	-	参与者	角色	执行者

	系统词汇法(OOA)	用例分析法	四色建模法	事件风暴
行为	-	用例关系		命令
客体	名词，模型	名词，模型	时标性对象、实体、描述对象	领域名词、模型
现象	-	-	业务事件	事件

在认识论中，每一个客观现象的出现，都可以使用主体、客体来分析。找到导致这个客观现象的行为背后的主体、客体，就能清晰地描述事件，也更容易看到问题的本质。从认识论的角度出发，建模的过程就是找到确定的客体作为模型的过程。

基于元模型把 4 种建模方法实例化一下：

从这个表中我们可以看出，系统词汇法的建模线索不够清晰，直接获得模型，没有从业务行为中抽取的过程。

而事件风暴可以这样理解：

执行者作为业务主体，在系统中发出了一个命令作为业务行为，对模型的状态发生了改变，最终导致了事件的发生。事件风暴是从事件、命令和执行者为线索推导出模型，整个过程更加完整。

换个角度看架构

什么是软件架构？通俗来说，软件架构就是将软件中合适的组件放到合适的地方，这就让软件架构变成了一种混合大量经验的艺术行为。

架构是什么？在维基百科中，软件架构的定义是：

软件架构是关于软件整体结构和组件的抽象描述，用于指导大型软件系统各个方面的设计。软件架构包括软件组件、组件之间的关系、组件特性以及组件间关系的特性。

那么我们可以解构一下，软件架构的内涵就是：

1. 组成软件结构的元素。
2. 结构之间的关系。

除此之外，还需要有一些原则来指导具体的实施，类似于施工规范和标准。在设计软件架构时，会做出大量的决策才能得出最终的元素和关系。然而，我们不需要马上作出所有细致入微的决策，只需要先作出后续难以调整的决策即可。正如维特根斯坦所说“世界是一切发生的事情”，我们也可以说架构是所有决策完成后的事情。

于是我们可以将架构描述为：

软件架构 = 架构中的元素 + 关系 + 实现原则和技术规范。

软件架构的过程 = 一切决策之和。

架构中的元素有可能用不同的视角来表达的，也有可能具有层级结构，于是我们往往通过图形化的方式来描述和表达，也就成了传说中的“PPT 工程师”。

1 复杂性管理

从复杂和混乱的信息中找到重点才能定义出元素以及找到关系。架构是一个非常玄学的领域，它不像编写一个确定的中间件一样，有明确的输入条件和输出。架构充满了权衡、取舍和纠结，其原因就在于复杂性问题。复杂的信息越多，系统熵越多，没有能量输入时，系统逐步趋向复杂、无序状态。

《人月神话》里面提到两个概念：本质复杂度和偶然复杂度。本质复杂度是在解决问题时，无法避免的事；偶然复杂度是做事方法不对，人为引入的复杂性。

架构师就是与复杂性对抗的人。

本质复杂度无法避免，而导致偶然复杂度上升的原因有：

1. 在软件项目中，随着参与人数的增加，沟通节点也会增多，从而导致信息节点的增加，偶然复杂度也越来越高。
2. 信息噪音过多，就像信息论描述的那样，当噪音太大真实的声音就被掩盖了。

我们必须能找到办法对抗复杂，我们能用到的工具有：

1. 获得信息“地图”，通过模型过滤掉多余的信息。
2. 分而治之，将复杂度隔离到局部，让局部的复杂性减弱，将复杂性进行分层。

PPT 是复杂信息的索引

在架构设计中，过滤掉多余信息的方法就是做 PPT 或者画图。

PPT 的真正作用是复杂性管理，这也是微软幻灯片软件 PowerPoint 的寓意来源。据说 Gaskins 在给 PowerPoint 起名字时，最初在洗澡时迸发的灵感，想到的名字中包含了这个词。而且在不知情的情况下，他的同事 Glenn Hobin 在机场的海报中，一眼看到了 PowerPoint 这个被强调的词。

一套架构用的汇报材料，可能就是某个复杂系统一份极好的索引。全景是什么、边界在哪里、骨干的业务逻辑是什么，都需要体现在幻灯片中。

所以做好幻灯片的前提是深刻吃透业务、产品、技术方案，而且还需要具备非常强的表现能力，把方案清晰地表现出

来。幻灯片图表并非只是展示，更像是一个索引来描述复杂的系统。

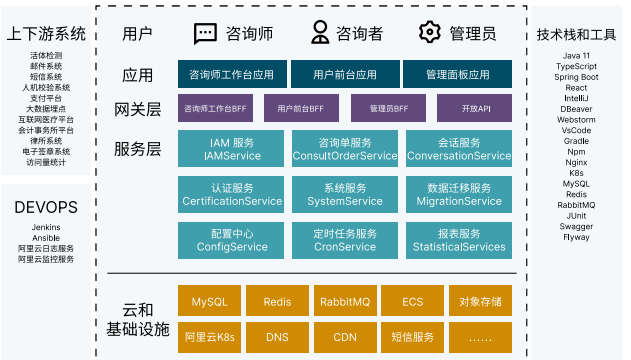


图 1 虚构的架构全景图

优秀的架构师、咨询师都能作出好的幻灯片，有时候甚至幻灯片就是一种很好的概念模型，这样想就应该能把幻灯片重视起来。没有好的思维结构，就做不出幻灯片，想到的不一定能表达出来，所以幻灯片做得好的人具有特别的强的思维能力。

简化和克制的图才是真正有用的图，因为保留的有效信息更为突出，将庞大无比的系统简化到几张 A4 能够被打印出来的纸时，它的复杂性才能真正被驾驭。

分而治之

分而治之不能降低复杂性，只能隔离复杂性。而分而治之可以通过不同的维度来进行，主要分解的方向有三个：

1. 水平分解。对系统进行分层，下层对上层透明，上层的开发者无需关心下层的变化。
2. 垂直分解。对系统进行按模块（上下文）切割，上下文之间不需要关心彼此，只有在有互相依赖的情况下，才需要了解对方。
3. 按时间分解。对系统的实现分步骤完成，根据迭代演进，制定产品、架构路线图（RoadMap）。

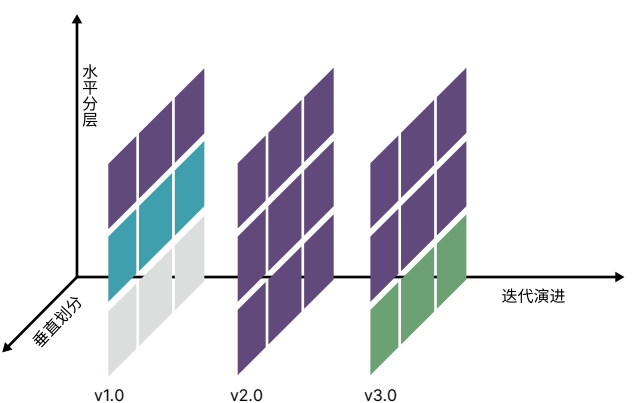


图 2 架构维度

注意区分广为流传的 AFK 拓展立方体，AFK 拓展立方体更像是对系统容量的分解，而不是对复杂度的拆解。

对于系统设计来说，每一个版本我们可能都会进行垂直、水平分解得到一张平面的架构图，在持续演进的状态下不断更新。

这就印证了我们前面说的架构的过程是一切决策之和。架构决策的影响有时候远远被我们低估，有时候我们的决策是基于上一次决策之后的结果作出的，而不是最初的问题，这就让架构问题变得更加复杂。

2 系统水平分层

分层的目的是水平隔离复杂性，那么怎么定义“一层”呢？由于对具体分层的定义非常模糊，导致了我们实际上分了很多层，但是却觉得没几层。

架构分层的主客体分析

互联网通信依赖的网络协议 TCP/IP 是一个非常经典的分层模型，因为全球网络是一个经典的分布式系统，实际上

我们无论在设计哪种形态的分布式架构时，都可以参考网络协议的设计思想。

我们在学习 TCP/IP 或者 OSI 分层网络时会使用一个常见的“邮差”比喻，来形象地描述网络协议的原理，其中就体现了分层的思想。

Montreal 需要寄送一封信，她在信的结尾写上了自己的名字作为落款，然后通过邮局将其寄出。邮局进一步包装并贴上邮局的标签，将信件发送到运输公司。运输公司将其装箱，并通过不同的交通工具将其递送到目标站点，并发送到目标邮局，也就是他们在目的地的客户那里。最终，邮局将信件发送到收信人手中。

我们将整个过程看作三层：

1. 用户层：也就是收信人、发信人收发信件的过程。
2. 邮局层：邮局的工作人员处理邮件的过程。
3. 运输层：物流公司通过不同的交通工具运输货物的过程。

有时候，我们仅仅通过行为来描述分层很难说清楚分层是什么，比如邮局和物流公司的分层在某些情况下可能说不明白。我们可以通过另外一个视角来看待这个问题。

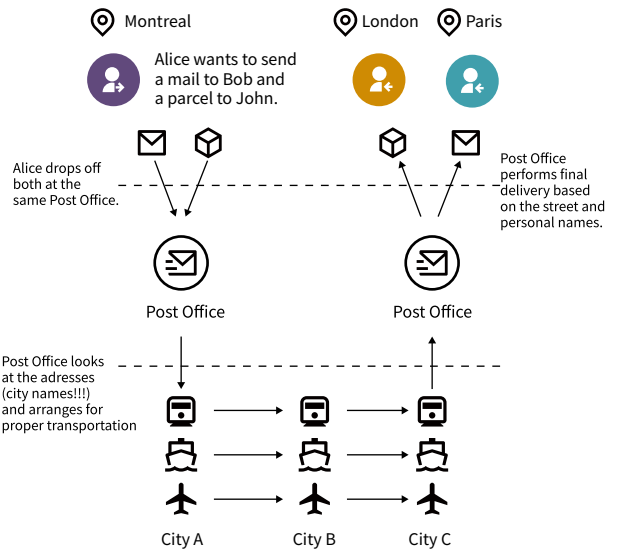


图 3 邮差比喻

图片来源：https://www.eecs.yorku.ca/course_archive/2010-11/F/3213/CSE3213_03_LayeredArchitecture_F2010.pdf

任何一个行为都能找到它的操作者以及身份，也就是行为的主体，也能找到操作的内容，也就是行为的客体。我们可以通过分析主体、行为、客体三个要素来辨析分层之间的关系。这样让分层更加明确。如果能在该层找到明确的主体对象、客体对象，以及说明其关系，我们就能将其说清楚。

我们用一张表格来划分，并将其表述更加精确：

分层	主体	行为	客体
用户层	收信人、发信人	收发信件的过程	原始信件
揽收层	邮局、揽收点	揽收寄件，并打包的过程	包装后信件
运输层	物流公司	运输货物，装箱运输的过程	物流箱

通过主体的明确和客体的明确，主体之间的职责会清晰地浮现出来，主体的权责更加清晰，我们细心分析也会发现这种分层也是社会化分工的体现。主体的性质是截然不同

的，邮局、揽收点作为法律主体时，一般不是以自然人的性质出现的。另外物流公司这类主体往往也需要额外的资质、营业许可，侧面说明了分层的要素。

这是现实中的分层思想，那么在软件中是不是这样的呢？假设以后端业务系统的经典三层结构，我们来看下它的分层主客体分析：

分层	主体	行为	客体
Controller 层	Controller	处理业务场景	Request /Response
Service 层	Service	处理通用能力	Model
Repository 层	Repository	处理数据持久化	数据 /SQL

用主客体来分析，MVC 模型如果没有 Service 时，只能算两层，因为 Model 只是客体（忽略 Model 和其属性之间的主客关系），构不成完整的一层。Service、Repository 层都有对应的主客体关系，能够说清楚它的权

责关系。

如果按照网络协议的分层设计，下层是不需要知道上层的信息或者知识的，也就是说理想的情况下 Repository 层的客体应该是无差别的数据才对。所以我们可以看到 JPA 这类 ORM 工具接收了两类参数：数据体 + 领域模型的类型信息。当我们无法实现出无差别的 Repository 层时，才不得不使用持久化对象这类概念。

所以这里总结下对分层的理解：

1. 分层是主体权责的让渡，通过分层演化出更多主体，实现分工。
2. 下层需要尽可能地提供无差别的能力给到上层，让上层对下层保持透明。

那么通过辨析主客体的关系，就能提高代码的表达力，尤其是在命名上。所以对客体起名的关键在于定义这个客体的概念，使用拟物的方式起名。对主体的起名需要定义它的职责，使用拟人的方式起名。

这样就能通过类似“主谓宾补”（主语：服务对象，谓语：

方法，宾语：参数，补语：返回值）的方式编写代码，让我们在编写业务代码时思绪流畅。

3 系统垂直划分

服务划分的目的是垂直分解复杂性。这里的“垂直”指的是在某一层内的垂直。因此，不同层之间垂直划分的粒度可能并不相同。

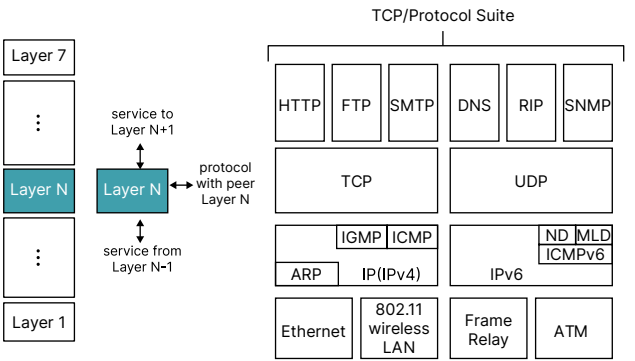


图 4 垂直分层

图片来源：https://www.eecs.yorku.ca/course_archive/2010-11/F/3213/CSE3213_03_LayeredArchitecture_F2010.pdf

在很多系统的垂直划分时最大的误区是穿透了分层，想象一下我们有一套自己的通讯协议，这套通讯设备同时具备了应用层、网络层、传输层、数据链路层，那么这套通讯协议就很难被归纳到 TCP/IP 协议簇中了。

垂直划分的权责问题

实际上水平分层比垂直分层要简单很多，因为我们很容易根据工作的性质识别到他们边界。比如，网关、业务服务、数据库中间件，很容易就知道他们的分层关系。

我们怎么找到垂直划分的边界呢?

技术类的垂直划分实际上比较简单的，比如接入层，如果有两种物联网设备接入协议，我们很容易将其根据协议类型划分开。这是因为计算机科学家在这些领域有充分的解决方案。

但是业务服务的垂直划分就非常麻烦了，特别是没有经历过沉淀的创新类软件系统。以企业通讯软件为例，企业通讯录、群组、用户这几个概念若即若离，无论是划分开、还是合并到一起都会有不少的麻烦，有时候甚至没有完美

（或者有些架构师称作干净）的解决方案。

我们会发现，垂直划分和水平划分的特点可以被归纳出来，这便于我们对系统进行设计。

划分方式	特点	示例
水平划分	性质具有明显的不同	领域层、网关
垂直划分	性质类似但是职责范围不同	用户服务、会议服务

下面这张图为互联网收银系统的分层架构，水平的方向使用了同样的背景色，他们的性质基本类似。假设这个系统以非常理想的方式设计，接入层为不同的网络接入方式，它取决于应用场景，它的垂直划分非常容易。

但是对于应用层来说，如何清晰地界定哪些属于应用，需要对产品设计有非常深刻的理解，以及和产品经理达成共识。对于领域层来说，如何找出相对独立的能力单元也不是那么容易（当我们把领域逻辑和应用逻辑分开后，领域层的垂直划分相对简单一些）。

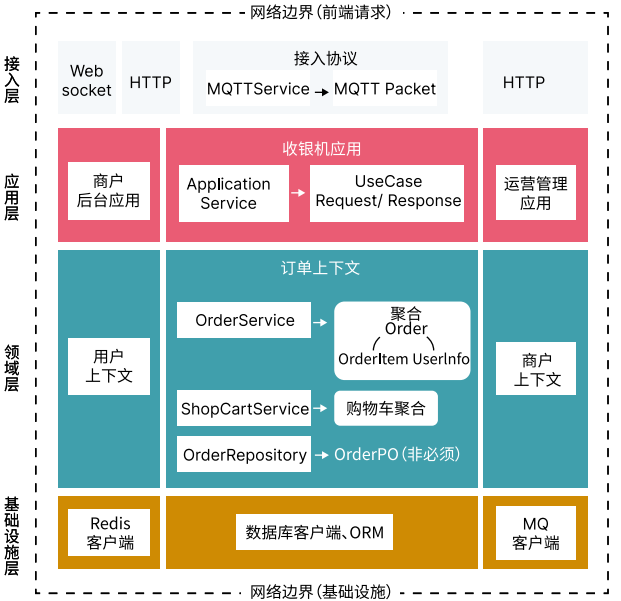


图 5 完整示例

那么对于业务相关的服务来说，我们有什么线索可以进行垂直划分吗？对于应用层的服务来说，我们可以主要以使用该应用的业务主体来划分。比如在餐饮系统中，我们可能会有下面几个主体使用该系统：终端用户（店员）、商户、系统管理员、第三方 API 调用者，在应用和服务分离部分

我们已经详细讨论过这类问题，应用层的划分比较容易。

那么领域层呢？领域层的微服务之间大部分情况下是平等的。由于领域服务和系统状态（有自己的数据库）相关性比较强，这些状态可以通过模型（实体）来表达。这也是为什么我们通常说的微服务划分，实际上是说的领域微服务，它们的划分和上下文划分可以一一对应。所以领域服务的划分，是根据系统所处理的客体来划分的（这是为什么我们划分领域服务需要先找模型，因为模型一般是作为客体出现），这是一个比较好的线索。

这里总结下应用层和领域层划分线索的区别，以及辨析权责关系：

分层类型	划分方式	权限	职责
应用层	参考业务主体为线索来划分	访问领域层、基础设施层的服务能力，无权修改系统的状态	编排领域层，为业务主体提供个性化场景

分层类型	划分方式	权限	职责
领域层	参考业务客体（领域模型）的分类来划分	修改系统状态的能力，无权干涉应用场景	提供上下文内对系统状态的管理职责

当权责关系被定义清楚后，开发团队在开发时能减少沟通的成本，但是并不意味着应用层和领域层的鸿沟。对于规模非常大的系统来说，让领域层持有所有的系统状态会变得过重，也可以考虑让应用层持有一些局部的领域逻辑。

把团队看作分布式系统

团队管理是一件非常困难的事情，尤其是在认知能力强的群体中尤其如此（和直觉相悖，认知能力越强的团队越不好管理）。历史告诉我们，缺乏组织的人类群体没有任何战斗力，且在社会化生产的过程中效率非常低下。

在一些公司中，管理问题时时刻刻存在，要么靠管理者的本能管理，要么就是管理混乱，或者是靠经验性的管理框架来进行管理。在 IT 团队中这种现象尤其比较明显，因为往往技术管理者更关注技术本身而非管理。

有意思的是，管理能不能也用 “技术的语言”来表述呢？

其实是可以的，作为一个分布式系统的爱好者，慢慢发现分布式系统和团队管理有一些共通之处，能用这些发现解决一些问题。

1 团队管理和分布式系统

团队管理是社会学讨论的问题，分布式系统是计算机中的概念。它们能有什么关系呢？在开始写作前，我在和同事聊到这部分内容的想法，同事笑道：你这个想法非常有意思，但是你可能只是强行将它们联系到一起。

这两个概念甚至都不在一个学科，一个是文科，而一个算工科的内容。但是，**世界是非常有意思的，跨学科的碰撞往往能发挥意想不到的作用。**

在《分布式计算——原理、算法与系统》[1] 这本书的开篇提到，“分布式系统是一组相互独立的实体构成的集合，这些实体相互协作可以解决任何单独的实体所不能解决的问题”。作者认为，分布式系统在宇宙之初就存在了，从蜂群、微生物系统、甚至由人体细胞构成的各种系统，这

1 参考图书：《分布式计算——原理，算法与系统》
https://book.douban.com/subject/10785422

些都是分布式系统。

团队是一个能独立承担一定功能和职责的人类群体，那么也应该是一个分布式系统，符合分布式系统的一些基本理论。

接下来我们会聊到分布式系统的两种模型，分别代表两种典型的团队形态，也代表不同的计算模型：

1. 主从调度模型。在微观状态适用，当团队人数不多，能被直接调度到，可以看作微观团队系统。
2. 反馈调节（市场）模型。在宏观状态下适用，当团队规模大到不可能被直接管理的时候，只能通过宏观调节机制来做宏观调控。

大部分情况下，我们不会用到反馈调节模型，但是当我们仔细观察和分析大型企业的工作机制时，就能发现端倪。这部分的内容，这里不过多讨论。

2 主从调度模型

让我们先看下主从调度模型的基本形态是什么样的，它能指导我们加深对团队的认识吗？这种系统由两个主要的角色构成：Dispatcher 和 Worker，这是主从调度模型的基本逻辑。

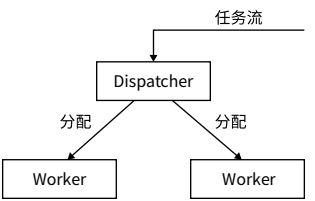


图 1 基本模型

回顾一下计算机系统中的这两个角色。基于负载均衡的无状态服务集群，负载均衡器充当了 Dispatcher 的角色，普通的服务器充当了 Worker 的角色；基于主从的 CI 构建系统 Jenkins，它的 Master 节点就是 Dispatcher 角色，负责处理任务调度，Slave 节点用于执行任务构建。

在这种模型下，我们发现如果 Master 节点用来跑具体的任务，会挤压它的调度能力， Master 节点崩溃整个系统也不可用了。

我们回归到团队管理中来，一名团队的 Leader 如果每天关注在自己具体的工作上，让 Worker 角色的工作挤占了 Dispatcher 角色的工作，整个团队会开始混乱。在好的情况下，团队中会有其他成员自发地弥补这部分工作，就有点类似于人体被切除某些器官后发生的代偿行为。然而，团队并不总是有这么好的运气，如果没有人来承担 Dispatcher 的工作时，整个系统就陷入混乱。

所以我们需要将模型做一些简单的修正，一名 Leader 不仅需要作为 Dispatcher 的角色还需要作为 Worker 完成某些具体的任务。这也更反映现实，一名技术经理不仅需要分配工作或者任务，往往还需要完成一部分的开发工作。

承担一定的 Worker 职责是有好处的，如果不熟悉具体的工作是什么，很难真正地承担 Dispatcher 的职责，分解的任务不具备可执行性。

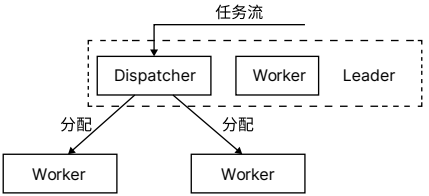


图 2 Leader 的职能

在基本模型中，也可以找到 Dispatcher 和 Worker 的主客体关系，这样更能辨明他们的职责关系。

对于 Dispatcher 来说，主体是他自己，客体是被调度的 Worker 以及调度工作；对于 Worker 来说，主体是 Worker 自己，但是客体是具体业务工作。

在主从模型中，Dispatcher 带动 Worker 的能力是有限的，因此为了让系统规模能进一步扩大，多级主从模型就是有必要的了。

根据经验，IT 团队由于工作性质的原因直接管理的人数最多能到 10 人，也就是常常说的两个披萨就能吃饱的团队规模，如果人数再多就需要增加系统层级通过间接管理的方式进行管理。

间接管理就形成了多层调度模型，也会产生中间节点。中间节点在上层的的角色就是 Worker，在下层就是 Dispatcher。

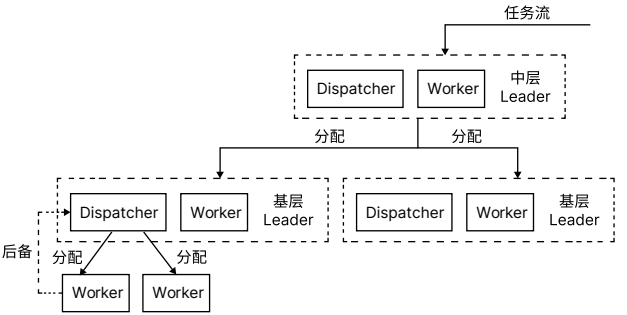


图 3 多层结构

在多层模型中，我们可以将系统看作一个由多个子系统组合的系统。为了度量系统的健康状态，需要引入两个概念：

1. 团队宽度。一个团队的宽度是指每层能够直接调度到的最大节点数量。对于不同的 Dispatcher 由于工作性质和能力不同直接调度的数量可能不同。所以团队宽度又可以分为最大宽度、最小宽度、平均宽度。
2. 团队深度。一个团队的深度是指信息从团队顶层节点传递到最终端节点的层数。信息在传递的过程中会失真、变形，这也是为什么越来越多的公司追求扁平化（扁平化也不等于就是好）的原因。和团队宽度类似，每一个终端节点向上的路径深度也不一

样，也可以分为最大深度、最小深度、平均深度。

通过这些概念我们可以建立起一些指标，定量的管理模式。

当我们认识到团队满足这样一个基本模型后，可以通过模型识别到团队管理中的问题，也就能针对性地优化团队管理，将有问题的团队带出泥潭。

无领导小组

这种情况发生在私人关系非常好的团队，领导交给几个平级的团队成员一些任务，但是没有经过任务拆分，且没有说明谁为这件事负责。

有时候这种场景是刻意为之，比如在应届生招聘时，会通过群体面试的方式设计无领导小组进行面试。目的是，通过竞争、选举、冲突寻找具有领导能力的潜在调度者。

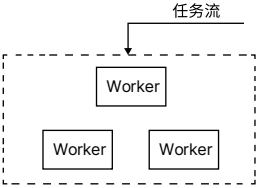


图 4 摩擦 - 无领导小组

多任务流团队

主从调度模型中，当一个 Dispatcher 的能力不能满足团队需要时，能否增加多个 Dispatcher？

答案否定的，在分布式系统中，避免这样的模型：多个承担有 Worker 角色的 Dispatcher 构成系统，它会带来状态的一致性问题。在团队管理中，Dispatcher 的负载不会太大，但是需要保证一致性。在一个团队中出现 2 个 PM 会是一个灾难，然而这种场景在各个公司反复上演。

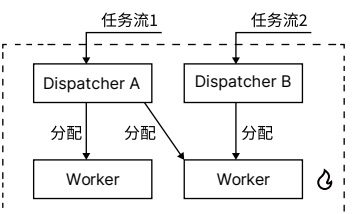


图 5 摩擦 - 多任务流团队

这种情况发生在团队出现 2 个实时调度者，当执行者需要接收多个调度者的任务时，会发生下面问题：

1. 任务逻辑矛盾。

2. 任务计划被打乱。
3. 超出正常工作时间能消化的工作量。

跨级指挥

跨级指挥和多调度者类似，对于执行者而言，无法区分哪些任务优先级更高。同时，跨级指挥造成执行者的直接领导者受到影响，对团队的效率产生影响。

跨级指挥的出现，往往导致中间层指挥体系失效的情况。

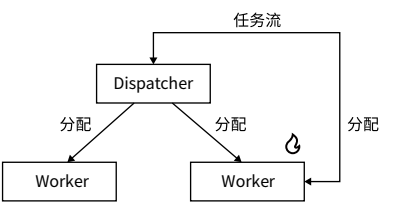


图 6 摩擦 - 跨级指挥

无上升通道的团队

在这种团队中，即使工作非常久也无法从执行者成为调度者。无上升通道的团队是一个僵化的团队，意味着团队规模和业务没有增长，也没有足够的人员流动。

在这种团队中执行者和调度者都没有足够的主观能动性。

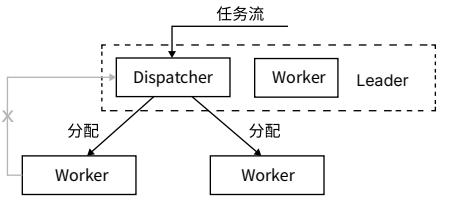


图 7 摩擦 - 无上升通道的团队

另外一种情况造成无上升通道问题的原因是，高层对团队彻底失去信心，决定从外部或者其他团队空降 Dispatcher 从而对抱有升职的现有团队成员造成打击。

“傀儡” Leader

在这种团队中，看似团队具有良好的结构，实际上 Leader 如果因为某些原因没有起到调度者的作用，也会让团队的任务传递出现问题。

和无领导小组不同的是，团队中存在名义上的调度者，阻挡了其他成员补齐这个位置的动力。但民间自发出现调度者时，会造成民间调度者和名义上调度者之间的冲突。

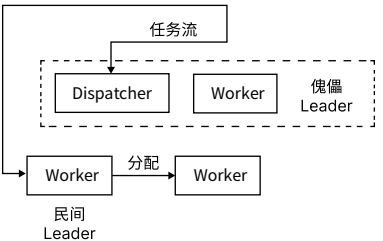


图 8 摩擦 - 傀儡 Leader

激励失效

激励不仅仅是金钱，还包括很多内容。如果激励出现问题，无法传递、或者不合适地传递到团队成员，会造成严重的问题。

在多级的团队系统中，每一层都需要存在相应的激励。激励体系需要建立一个正反馈，符合“劳者多得”而不是“能者多劳”的局面。

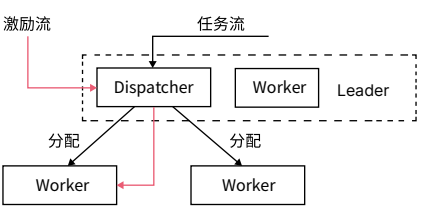


图 9 摩擦 - 激励失效

反馈失灵

这类团队高层无法知道基层的运行情况，无法作出合理的决策，盲目下发政策，造成基层工作无法展开。

反馈失灵虽然不会短期影响团队运行状态，但是系统会持续性恶化。

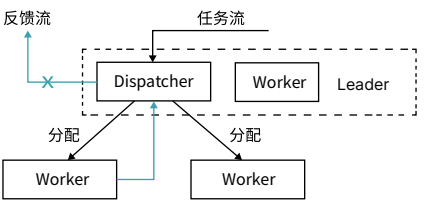


图 10 摩擦 - 反馈失灵

主从调度模型的特点和局限性

主从调度模型具有一些明显的特点，在不同的场景下具有一些局限性，当我们了解到局限性带来的相关影响时，就需要作出干预和修正，让系统回归到正常状态。

首先，主从调度模型是有状态的。系统的调度状态在 Dispatcher 上，如果 Dispatcher 上的调度信息丢失，可

能会造成系统的任务调度混乱。

这时很多人可能会说，在微服务系统中，我们的业务服务是没有状态的，是谁来调度的呢？无状态并不是真的无状态，而是状态被隔离了。例如，负载均衡器 - 服务器 - 数据库模型中，状态被隔离到负载均衡器、数据库中，服务器可以做到没有状态，系统状态由负载均衡器和数据库承担。

客户端负载均衡这种方式是不是没有状态？

其实也是有的，即使是通过 HASH 算法直接匹配到目标服务器，通过算法和计算规则实现客户端自己调度，实际上这个规则是数学规律帮我们提供了状态。

这种模型是有中心的，有一些看起来无中心化的系统，实际是由选举机制自动完成中心化的选举，慢慢磨合出真正的领导者。

其次，Worker 的主动性会受到抑制。主从调度模型有点类似于计划经济，如果调度的层级过深就会出现积极性和

效率问题，而这一点正好是市场模型所解决的问题。

主从调度模型的风险大部分来源于 Dispatcher，如果没有建立良好的后备机制和做好知识传递，当 Dispatcher 出现问题后（离职、生病），系统会处于短暂停顿状态。

另外这种系统中，由于竞争由上层决定，会导致腐败和潜规则，带来不良的影响。由于竞争由上层的决定，因此基层的声音被忽视，基层领导者只需要讨好上级，Worker 的诉求可能会被忽视。当然，反过来看，人不是纯粹的机器，有一定自主性，这一点非常重要。

3 市场模型

主从调度模型看起来很完善了，但是却不能描述一些特殊的场景，因此我们需要另外一种模型：市场模型，它是通过反馈调节来完成的。这个系统由 3 种基本元素构成：玩家、市场、调节者，以及一个隐藏的元素：庄家所构成。

这种系统出现在层级较为扁平的公司，各个团队相对独立和灵活，对于巨型公司的上层结构也符合这个模型。对于

市场经济为主体的国家来说，整个经济体就是这个模型，所以我借用了市场这个词。

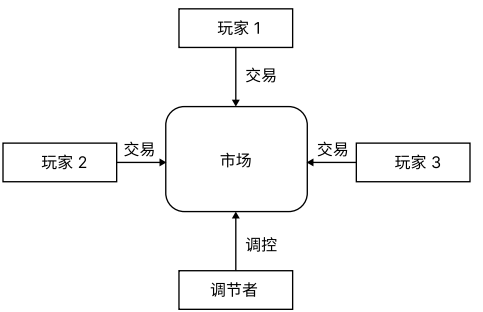


图 11 市场模型

在分布式的计算机系统中，这种模型比较少见，一些弹性扩容的系统可以看作这种模型的简单实现。这是因为计算机科学基础决定了的，计算机科学建立在离散数学上，我们使用的计算模型为图灵模型，图灵模型是一种确定的计算模型（可计算性）。反馈调节模型不是一种确定的计算模型，目前的超计算（Hyper computation ）就是在研究如何在计算机中应用这类模型。

在这个模型中，Dispatcher 被市场代替了，市场可以认为

是一个无形的手，这个手是全体玩家构成的。这种模型是真正的去中心化模型，在生活中如果能细心一点，会到处发现这种模型的影子：生物圈、股市、人体内分泌系统等。

回到团队管理来看，我们可以把市场看作一个大的公司，每个玩家就是一个团队，这些团队可以找其他团队合作，但是都要在市场上来竞争；对于国家而言，这些玩家就是企业。如果我们把场景聚焦到大型企业来看，每个团队都需要在这个企业的生态链中寻求一席之地，和上下游的合作关系就是交易的过程。

这种系统具有和主从调度模型不同的逻辑，很多性质甚至是违反直觉的。

系统具有自我调节能力，且是无中心化的，调节者不是必需的，只要市场在就不会崩溃。由于没有中心，调节的效率非常高、平滑且精细。但是在一定时间后，由于马太效应的积累，会出现庄家，庄家会控制市场，让市场失去平衡。

另外一方面，玩家具有主动性，市场上出现新需求时，玩家会立即参与，参与者的积极性高。但是在没有明确监管机制的环境下，会快速出现欺骗等不正当竞争行为。

作者简介

林宁

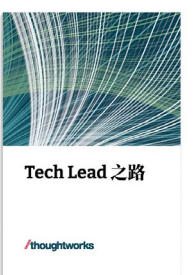
网名“少个分号”

Thoughtworks 软件工程师和 Tech Lead

专注于 Java 分布式系统和中间件的开发、领域驱动设计（DDD）的咨询工作。服务过国内外通信、保险、银行、餐饮等行业的企业，为国内知名企业提供过 DDD 和微服务的咨询服务。Thoughtworks 内部 DDD 社区发起人，维护有公众账号“DDD 和微服务”。

个人网站

更多洞见电子书



更多电子书下载

Thoughtworks 洞见公众号

Thoughtworks 洞见网站

/thoughtworks