架构师

ARCHITECT

特刊

之架构漫谈

SPECIAL ISSUE

March,2016

扫这里
和百位架构师共同聊架构

InfoQueue

版权声明

InfoQ 中文站出品

架构漫谈

出版：极客邦控股（北京）有限公司

北京市朝阳区洛娃大厦 C 座 1607

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn

序

架构师是一个很特殊的群体，特别是在软件行业；同时架构师也是一个很令人向往的工作。在软件行业，架构师和工程师就类似于上帝，创建出形形色色的软件产品来服务于人类。要想当好这个角色，架构师自然也需要具备某种上帝的视角，来观察并表达这个世界。这样创造出来的软件，才是可以长大并与时俱进的，才能更好的满足人类的需求。

从事软件行业有些年头了，从这个行业学到了很多，也收获了很多。一直以来都是在获取，总觉得自己应该为这个行业做点什么。2016 年春节期间，发心把自己这么些年的思考和总结，用一个系列文章的方式比较系统的表达出来，以至诚恭敬之心，奉献给各位读者。愿这些文章能够令各位有所启发，并引发思考，在更好地认识架构、设计架构和写好代码等方面，起到一些微小的作用。

这些文章得以和各位见面，要感谢 InfoQ 的霍泰稳和郭蕾的支持和帮助。也谢谢内人方旻在家庭方面给予的支持，并不厌其烦的为我校对，纠正我的行文错误。

思及自身水平有限，文字功底也差，难免伤人慧命，深感惭愧和惶恐！望各位读者，鉴其愚诚，不吝慈悲指正！

王概凯 Kevin

2016.03.03

目录

1

什么是架构

本文是漫谈架构专栏的第一篇，作者将会通过类比的方式来介绍什么是架构以及为什么会产生架构。

缘起

一直以来，在软件行业，对于什么是架构，都有很多的争论，每个人都有自己的理解。甚至于很多架构师一说架构，就开始谈论什么应用架构、硬件架构、数据架构等等。我曾经也到处寻找过架构的定义，请教过很多人，结果发现，没有大家都认可的定义。套用一句关于 Big Data 流行的笑话，放在架构上也适用：

Architecture is like teenage sex，everybody talks about it，nobody really knows what is it。

事实上，架构在软件发明时的 N 多年以前，就已经存在了，这个词最早是跟随着建筑出现的。所以，我觉得有必要从源头开始，把架构这个概念先讨论清楚，只有这样，软件行业架构的讨论才有意义。

什么是架构？

架构的英文是 Architecture，在 Wikipedia 上，架构是这样定义的：

Architecture (Latin architectura, from the Greek ἀρχιτέκτων arkhitekton"architect", from ἀρχι- "chief" and τέκτων "builder") is both the process and the product of planning, designing, and constructing buildings and other physical structures。

从这个定义上看，架构好像是一个过程，也不是很清晰。为了讲清楚这个问题，我们先来看看为什么会产生架构。

为什么会产生架构？

想象一下，在最早期，每个人都完全独立生活，衣、食、住、行等等全部都自己搞定，整个人类都是独立的个体，不相往来。为了解决人类的延续的问题，自然而然就有男女群居出现，这个时候就出现了分工了，男

性和女性所做的事情就会有一定的分工，可是人每天生活的基本需求没有发生变化，还是衣食住行等生活必须品。

但是一旦多人分工配合作为生存的整体，力量就显得强大多了，所以也自然的形成了族群：有些人种田厉害，有些人制作工具厉害，有些地方适合产出粮食，有些地方适合产出棉花等，就自然形成了人的分群，地域的分群。当分工发生后，实际上每个人的生产力都得到了提高，因为做的都是每个人擅长的事情。

整个人群的生产力和抵抗环境的能力都得到了增强。为什么呢？因为每个人的能力和时间都是有限的，并且因为人的结构的限制，人同时只能专心做好一件事情，这样不得已就导致了分工的产生。**既然分工发生了，原来由一个人干生存所必需的所有的事情，就变成了很多不同分工的角色合作完成这些事情，这些人必须要通过某些机制合在一起**，让每个人完成生存所必需的事情，这实际上也导致了交易的发生（交易这部分就不在这里展开了，有机会再讨论）。

在每个人都必须自己完成所有生活必须品的生产的时候，是没有架构的（当然在个人来讲，同一时刻只能做有限的事情，在时间上还是可能会产生架构的）。一旦产生的分工，就把所有的事情，切分成由不同角色的

人来完成，最后再通过交易，使得每个个体都拥有生活必须品，而不需要每个个体做所有的事情，只需要每个个体做好自己擅长的事情，并具备一定的交易能力即可。

这实际上就形成了社会的架构。那么怎么定义架构呢？以上面这个例子为例，把一个整体（完成人类生存的所有工作）切分成不同的部分（分工），由不同角色来完成这些分工，并通过建立不同部分相互沟通的机制，使得这些部分能够有机的结合为一个整体，并完成这个整体所需要的所有活动，这就是架构。由以上的例子，也可以归纳出架构产生的动力：

1. 必须由人执行的工作（不需要人介入，就意味着不需要改造，也就不需要架构了）
2. 每个人的能力有限（每个人都有自己的强项，个人的产出受限于最短板，并且由于人的结构限制，同时只能专注于做好一件事情，比如虽然有两只眼睛，但是只能同时专注于一件事物，有两只手，无法同时做不同的事情。ps. 虽然有少部分人可以左手画圆右手画框，但是不是普遍现象）
3. 每个人的时间有限（为了减少时间的投入，必然会导致把工作分解出去，给擅长于这些工作的角色来完成，见 2，从而缩短时间）

4. 人对目标系统有更高的要求（如果满足于现状，也就不需要进行架构了）
5. 目标系统的复杂性使得单个人完成这个系统，满足条件 2，3（如果个人就可以完成系统的提高，也不需要别的人参与，也就不需要架构的涉及，只是工匠，并且一般这个工作对时间的要求也不迫切。当足够熟练之后，也会有一定的架构思考，但考虑更多的是如何提高质量，提高个人的时间效率）

有人可能会挑战说，如果一个人对目标系统进行分解，比如某人建一栋房子，自己采购材料，自己搭建，难道也不算架构嘛？**如果对于时间不敏感的话，是会出现这个情况的，但是在这种情况下，并不必然导致架构的发生。**如果有足够的自觉，以及足够的熟练的话，也会产生架构的思考，因为这样对于提高生产力是有帮助的，可以缩短建造的时间，并会提高房子的质量。事实上建筑的架构就是在长期进行这些活动后，积累下来的实践。

当这 5 个条件同时成立，一定会产生架构。从这个层面上来说，架构是人类发展过程中，由懵懵懂懂的，被动的去认识这个世界，变成主动的

去认识，并以更高的效率去改造这个世界的方法。以下我们再拿建筑来举例加强一下理解。

最开始人类是住在山洞里，住在树上的，主要是为了躲避其他猛兽的攻击，以及减少自然环境的变化，对人类生存的挑战。为了完成这些目标，人类开始学会在平地上用树木和树叶来建立隔离空间的设施，这就是建筑的开始。但是完全隔离也有很多坏处，慢慢就产生了门窗等设施。

建筑的本质就是从自然环境中，划出一块独占的空间，但是仍然能够通过门窗等和自然环境保持沟通。**这个时候架构就已经开始了**。对地球上的空间进行切分，并通过门窗，地基等，保持和地球以及空间的有机的沟通。当人类开始学会用火之后，茅棚里面自然而然慢慢就会被切分为两部分，一部分用来烧饭，一部分用来生活。当人的排泄慢慢移入到室内后，洗手间也就慢慢的出现了。这就是建筑内部的空间切分。

这个时候人们对建筑的需求也就慢慢的越来越多，空间的切分也会变成很多种，组合的方式也会有很多种，比如每个人住的房子，群居所产生的宗教性质的房子，集体活动的房子等等。这个时候人们就开始有意识的去设计房子，架构师就慢慢的出现了。一切都是为了满足人的越来越高的

需求，提升质量，减少时间，更有效率的切分空间，并且让空间之间更加有机的进行沟通。这就是建筑的架构以及建筑的架构的演变

总结一下，什么是架构，就是：

1. 根据要解决的问题，对目标系统的边界进行界定。
2. 并对目标系统按某个原则的进行切分。切分的原则，要便于不同的角色，对切分出来的部分，并行或串行开展工作，一般并行才能减少时间。并对这些切分出来的部分，设立沟通机制。
3. 根据 3，使得这些部分之间能够进行有机的联系，合并组装成为一个整体，完成目标系统的所有工作。

同样这个思考可以展开到其他的行业，比如企业的架构，国家的架构，组织架构，音乐架构，色彩架构，软件架构等等。套用三国演义的一句话，**合久必分，分久必合**。架构实际上就是指人们根据自己对世界的认识，为解决某个问题，主动地、有目的地去识别问题，并进行分解、合并，解决这个问题的实践活动。架构的产出物，自然就是对问题的分析，以及解决问题的方案：包括拆分的原则以及理由，沟通合并的原则以及理由，以及拆分，拆分出来的各个部分和合并所对应的角色和所需要的核心能力等。

2

认识概念是理解架构的基础

本文是漫谈架构专栏的第二篇，作者通过几个例子，讨论了一下认识概念的误区，如何有效的去认识概念，明白概念背后的含义，以及如何利用对概念的理解，快速的进行学习。

在第一章中，我们讨论了什么是架构。事实上，这些基础概念对于做架构是非常重要的，大部分人对于每天都习以为常的概念，都自以为明白了，但实际上都是下意识的，并不是主动的认识。比如说“什么是桌子？”，做培训的时候，我经常拿这个例子来问大家，回答千奇百怪。这实际上就导致了做架构的时候，不同角色的沟通会出很多问题，那么结果也就可想而知了。

如前一篇所说，**架构实际上解决的是人的问题**，而概念是人认识这个世界的基础，自然概念的认识就非常的重要。这篇文章尝试讨论一下，如何去认识概念。当然这篇不是语言学的文章，我这里所讨论的，和语言学可能不太一样，如果大家对语言学感兴趣，也可以去参考一下。

首先我要先声明一下，这一系列的文章，都是以人的认识为主体去讨论的，解决的都是人的问题，任何没有具体申明的部分，都隐含这一背景，以免大家误解。

概念也属于人认识这个世界并用来沟通的手段，包括“概念”这个概念，也是一样的。在古代，不叫“概念”，称之为“名相”。

何为相？

一般我们认为：看到一个东西，比方说杯子，“杯子”就是一个名字，指代的看到的东西就是相，就是事物的相状。我们一听到“杯子”这个词，脑海里就会浮现出一个杯子的形象。而“杯子”这个词，是用来指代的是这个相状的，叫做名。合起来就叫做“名相”。

可是当我们把杯子打碎了的时候，我们还会称这个碎了的东西叫杯子吗？ 肯定不会，一般会叫“碎瓦片”，如果我们把碎瓦片磨碎了呢，名字又变了，叫做“沙子”。这就奇怪了，同样一个东西，怎么会变出这么多的名字出来？

那究竟什么才是相？

实际上“相“表达的不是一个具体的东西，如上面所提的一个瓷器杯子，并不是指这个瓷器，而是这个瓷器所起的一个作用：一手可握，敞口（一般不超过底的大小，太大口就叫碗了），并且内部有一个空间可乘东西的

这么一个作用。并不是指这个瓷器本身。这也是为什么我们从电视上看到一个人拿杯子的时候，我们知道这个是杯子。但是实际上我们看到的都是光影而已。所以说相实际上代表的是这个作用，并不是具体的某个东西，而名是用来标识这个作用的，用来交流的。

为何需要这个作用？

这个作用其实是为了解决“人需要一个可单手持握，但是希望避免直接接触所盛物体”这个问题。

所以说，每个概念实际上所解决的，还是人遇到的某个特定的问题，我们把解决问题的解决方案，给定了一个名字，这个名字就是对应的某个特定的概念。对于概念这个词本身，为了统一指代这些名字，我们称起这类作用的名字称为“概念”。我们上次讨论的“架构”也是是同样的一个特定概念，这里不再详述。同样，什么是“建筑”？ “建筑”实际上解决的就是“人需要独占的空间，并还能够比较流畅的和外部世界沟通”的问题。

再拿前面的“桌子”来举例，什么叫“桌子”？ 很多人回答，四条腿，或者说有腿，有一个平面，等等，柜子不也是这样吗？为什么我们看到柜子，不会认为是桌子呢？即使我们放在柜子上吃饭，我们看到仍然会问，为什么在柜子上吃饭？ 不会叫桌子。如果明白了上面的道理，就很简单了，桌子实际上是为了解决人坐在椅子上，手还能够支撑在一个平面上继续开展活动的问题，一般会和椅子配对出现。坐在椅子上工作，对着柜子有一

个很严重的问题–不知道大家试过没有–就是腿无法展开的问题。当这么坐着超过半小时就知道是什么痛苦了。所以桌子的平面下方一定会有一个足够容纳膝部和小腿的空间，来解决这个问题。解决了这些问题的装置，才能称之为桌子。

类似也可以定义出来椅子，由此可见，桌子和椅子的高度也是有限定的，都是是解决人的问题，要符合人的身高：椅子的高度和深度，必须符合小腿和大腿的长度；椅背的高度要配合脊柱的高度；桌子的高度要配合小腿和脊柱的高度之和；成人和小孩的自然也就有区别了。这又变成生理学了，事实上要做好桌子和椅子，必须要理解人的生理结构，才能正确的理解桌子和椅子的概念。

同理，为何我们可以在不同的语言间进行翻译，是因为虽然语言不同，但是人类所面临的的问题是一样的，所使用的名不同而已。对于不同的动物之间的翻译也是同理。

关于抽象

在讨论桌子这个概念的过程中，很多人会提出抽象这个概念，认为定义桌子实际上就是抽象的一个过程。这里，我觉得有必要要澄清一下抽象这个概念，我认为这个里面有误解。我注意到，在做架构师的群体中，不谈抽象好像就不是一个合格的架构师。

抽象这个词代表的含义，实际上是把不同的概念的相似的部分合并在一起，形成一个新的概念。这个里面问题很多：首先“相似的部分”在不同的人看来，并不一定那么相似；其次，抽象之后形成的是一个新的概念，和原来那个概念并不一样，所解决的问题也不一样。所以我们不能用抽象来定义一个事物，抽象实际上是一个分类的过程，完全是另一码事。再举一个例子，杯子和容器，很多人认为容器是杯子的抽象，但是实际上杯子是杯子，容器是容器，它们所解决的问题是不一样的。当我们需要解决装东西的问题的时候，会说容器；当我们需要解决单手持握要装东西的时候，会说要一个杯子。

回过头来，根据架构的定义，要做好架构所首先必须具备的能力，就是能够正确的认识概念，能够发现概念背后所代表的问题，进而才能够认识目标领域所需要解决的问题，这样才能够为做好架构打好基础。事实上，这一能力，在任何一个领域都是适用的，比如我们如果想要学习一项新的技术，如 Hibernate、Spring、PhotoShop、WWW、Internet 等等，如果知道这些概念所要解决的问题，学习这些新的技术或者概念就会如虎添翼，快速的入手；学习一个新的领域，也会非常的快速有效；使用这些概念来解释问题，甚至发明新的概念都是很容易的事。为什么强调这个呢，因为做架构的时候，很多时候都是在一个新的领域解决问题，必须要快速进入并掌握这个领域，然后才能够正确的解决问题。

以上通过几个例子，讨论了一下认识概念的误区，如何有效的去认识概念，明白概念背后的含义，以及如何利用对概念的理解，快速的进行学习。掌握了这些原则，会有利于帮助我们在架构阶段，快速的识别和定位问题。

下一章我们会来讨论一下，如何快速的定位和识别问题，这是架构的起始。

3

如何做好架构之识别问题

本文是漫谈架构专栏的第三篇。接之前的第二篇，作者将会讨论如何做好架构，并根据实际经验分析如何找出实际工作中需要解决的问题。

按照之前架构的定义，做好架构首先需要做的就是识别出需要解决的问题。一般来说，如果把真正的问题找到，那么问题就已经解决了 80%了。这个能力基本上就决定了架构师的水平。

那么面对问题有哪些困难呢？

我们先看一则笑话。女主人公：老公，把袋子里的土豆切一半下锅。结果老公是把袋子里的每个土豆都削了一半，然后下锅。

当然很多人会说，这个是沟通问题，然后一笑了之。其实，出现这个现象是由于我们大部分时候过于关注解决问题，急于完成自己的工作，而不关心“真正的问题是什么”而造成的。当我们去解决一个问题的时候，一定要先把问题搞清楚。这也是我为什么要单独写一篇文章讲这个的原因。去看看软件开发工作者的时间分配也可以看出，大家大部分时间花在讨论解决方案和实现的细节上，基本都不会花时间去想“问题是什么”。或者即

使想了一点点，也是一闪而过，凭自己的直觉下判断。只有真正投入思考问题是什么的工程师，才可能会真正的成长为架构师

以这个笑话为例，看看在我们处理问题的时候，都会犯什么样的错误：

- 被告知要处理一个问题，但是交过来的实际上是一个解决方案，不是问题本身。
- 被告知要处理一个问题，直接通过直觉就有了一个解决方案，马上考虑解决方案如何落地，或者有几种解决方案，选哪个合适。

那么如何识别问题呢?

所有的概念基本都有一个很大的问题，就是缺乏主语。而我们大家都心照不宣的忽略这个主语，沟通的时候也都以为大家都懂得对方说的主语是谁，结果大家都一起犯错误。识别问题的一个最大的前提就是要搞清楚：是谁的问题。这个搞清楚了，问题的边界也就跟着确定了，再去讨论问题才有意义。

以上面切土豆的例子来分析：

1. 女主人提出一个问题，要切土豆下锅煮。
2. 男主人有一个问题，女主人交代了自己必须要完成的一个任务。

每个人都是优先处理自己的问题，自然就选择了 2，完成了这个任务。这也是大部分软件工程师处理的方式，以自己认为对的方式完成自己的问

题，没什么不对啊，也难怪能得到我们的共鸣。这个里面犯的错误是什么呢?

首先，女主人公提出的实际上是解决方案，而不是烧土豆这个问题本身。女主人当时执行这个解决方案可能有困难，就把执行解决方案作为一个任务，委托给了男主人。

其次，男主人得到了一个任务，尽心尽职地把这个任务完成了。

最后的结果是什么呢，每个人都做了很多工作，每个人都认为自己做的是对的，因此没有一个人对结果满意。因为真正的问题没有被发现，自然也就没有被解决，那么后续还得收拾残局，还要继续解决问题。事实上自己的工作并没有完成，反而更多了。把原因归结为沟通问题也是可以的，但对于解决问题似乎并没有太多的帮助。因为要改进沟通，这也是一个大问题。搞明白目标问题“是谁的问题，是什么问题”，当然也是需要沟通的。为了帮助自己更快的搞明白，首先要做的事是问正确的问题。架构师应该问的第一个正确的问题就是：目标问题是谁的问题。

当我们处理问题的时候，如果发现自己正在致力于把自己的工作完成，要马上警惕起来，因为这样下去会演变成没有 ownership 的工作态度。在面对概念的时候，也会不求甚解，最终会导致没有真正的理解概念。

作为软件工程师或者架构师，我们大部分时候是要去解决别人的问题，“别人”是谁，是值得好好思考的。在这个故事里面，男主人要解决的，

实际上是这个家庭晚餐需要吃土豆的问题，目标问题的主体实际上是这个家庭的成员。

明白了问题的主体，这个主体就自然会带来很多边界约束，比如土豆是要吃的，要给人吃的，而且还是要给自己的家人吃的。“切土豆下锅”这个问题，因为识别了问题的主体，自然而然的就附带了这么多的信息。**后续如何煮，是否放高压锅煮，放多少水，煮多长时间等等，就自然而然能够问出来其他问题来了，说不定还能够识别出来，女主人给的这个解决方案可能是有问题的**。这个时候才算是真正的明白了问题。可以想象，这样下去最后的结果一定是大家都满意的，因为真正的问题解决了。只有真正明白了是谁的问题，才能够真正地完成自己的任务，真正地把自己的问题解决掉，而不是反过来。

由上面的分析可以看出，找出问题的主体，是做架构的首要问题。这也是我一再强调的，我们要解决的问题，一定都是人的问题。更进一步，架构师要解决的，基本都是别人的问题，不是自己的问题。**再进一步，我们一定要明白，任何找上架构师的问题，绝对都不是真正的问题**。为什么呢？因为如果是真正的问题的话，提问题过来的人肯定都能够自己解决了，不需要找架构师。架构师都要有这个自觉：发现问题永远都比解决问题来的更加重要。

当问题的主体离架构师越远，就会让找出问题主体的过程越加困难，我们再举一个软件行业比较熟悉的例子：用户给产品经理提出要求，想要一把锤子。这是典型的拿解决方案作为问题的。真正的问题的主体是谁，是用户还是设计师还是施工队？如果产品经理当成是自己的问题，那么毫无疑问就给了锤子了。

我们需要识别：**用户究竟是二传手，还是问题的真正主体**。如果是设计师，那么问题的边界就变成了设计师的问题，如果是施工队，那么问题就变成了施工队的问题，如果是用户，那么就要看看用户到底有什么困难，绝对不是要一个锤子这么简单。这也说明了，问题的主体对问题的边界确定有多么的重要。

当明白了问题的主体，我们才可能真正的认识问题是什么。因为问题的主体是问题的隐含边界，边界不确定下来，问题就是不确定的。一旦确定了主体，剩下的就是去搞明白主体有哪些问题。这个就比较直接了，常用的方式就是直接面对主体进行访谈，深入到主体的工作生活当中，体验并感受这些问题，甚至通过数据的反馈来定位问题。这个大家就比较熟悉了，我就不展开了。

一般来说，**从问题暴露的点，一点点去溯源查找，一定会找出来谁的问题，以及是什么问题**。最坏情况就是当我们时间或者能力有限，实在是无法定位出是谁的问题的时候，比如系统出故障，也就意味着我们无法根

本解决问题。这时最好的办法就是去降低问题发生所带来的成本，尽量去隔离问题影响的范围。给我留出时间和空间去识别真正的问题。

总结一下，要正确的认识问题，需要问两个问题：

1. 这是谁的问题？
2. 有什么问题？

当得到的回答是支支吾吾的时候，我们就知道正确的方向在哪儿，以及需要做哪些事了。以我的经验，问题 1 会花比较多的时间，也是支支吾吾最多的地方，因为架构要解决的问题都是人的问题。但是一旦确定了答案，问题 2 就会变得非常容易。可以这样说，架构师的能力大部分会体现在问题 1 的识别上。

4

如何做好架构之架构切分

本文是漫谈架构专栏的第四篇，作者将会介绍架构的切分，并直戳切分的本质其实就是利益的调整。文中作者将会讨论为什么需要切分、切分的原则、切分与建模、切分的输出和组织架构等问题。欢迎阅读和反馈。

前一章已经讲了如何识别问题。在识别出是谁的问题之后，会发现，在大部分情况下，问题都迎刃而解，不需要做额外的动作。**很多时候问题的产生都是因为沟通的误解，或者主观上有很多不必要的利益诉求导致的**。但是总还有一部分确实是有问题的，需要做调整，那么就必须要有所动作，做相应的调整。这个调整就是架构的切分。

切分就是利益的调整

我们要非常的清楚，所有的切分调整，都是对相关人的利益的调整。为什么这么说呢，因为维护自己的利益，是每个人的本性，是在骨子里面的，我们不能逃避这一点。我们以第一篇文章里面的例子为例来做解释。

我们已经知道，**随着社会的发展，分工是必然的，为什么呢？**这个背后的动力就是每个人自己的利益。每个人都希望能够把自己的利益最大

化，比如：生活的更舒适，更轻松，更安全，占用并享有更多的东西。但是每个人的能力和时间都非常的有限，不可能什么都懂，所以自然需要舍掉一些自己不擅长的东西，用自己擅长的东西去换取别人擅长的东西。

对比一个人干所有的事情，结果就是大家都能够得到更多，当然也产生了一个互相依赖的社会，互相谁都离不开谁。这就是**自然而然而产生的架构切分，背后的原动力就是人们对自己利益的渴望。**人们对自己利益的渴望也是推动社会物质发展的原动力。

在这个模式下，比较有意思的是，每个人必须要舍掉自己的东西，才能够得到更多的东西。**有些人不愿意和别人进行交换，不想去依赖于别人，这些人的生活就很明显的差很多，也辛苦很多**，自然而然的就被社会淘汰了。如果需要在这个社会上立足，判断标准就变成了：如何给这个社会提供更好更有质量的服务。提供的更好更多的服务，自然就能够换取更多更好的生活必需品。实际上这就是我们做人的道理。

为什么需要切分

当人们认识到要主动的去切分一个系统的时候，毫无疑问，我们不能忘掉利益这个原动力。所有的切分决策都不能够违背这一点，这是大方向。结合前一篇“识别问题”，一旦确定了问题的主体，那么系统的利益相关人（用现代管理学语言叫：stakeholder）就确定了下来。所发现的问题，会有几种情况：

1. 某个或者某些利益相关人负载太重。
   A. 时间上的负载太重。
   B. 空间上的负载太重，本质上还是时间上的负载太重。
2. 某个或者某些利益相关人的权利和义务不对等。

切分的原则

情况 1 是切分的原因，情况 2 是切分不合理而导致的新的问题，最终还是要回到情况 1。对于情况 1，本质上都是时间上的负载。因为每个人的时间是有限的，怎么在有限的时间内做出更多的事情？那么只有把时间上连续的动作，切分成时间上可以并行的动作，在空间上横向扩展。所以切分就要有几个原则：

1. 必须在连续时间内发生的一个活动，不能切分。比如孕妇怀孕，必须要 10 月怀胎，不能够切成 10 个人一个月完成。
2. 切分出来的部分的负责人，对这个部分的权利和义务必须是对等的。比方说妈妈 10 月怀胎，妈妈有权利处置小孩的出生和抚养，同样也对小孩的出生和抚养负责。为什么必须是这样呢？因为如果权利和义务是不对等的话，会伤害每个个体的利益，分出来执行的效率会比没有分出来还要低，实际上也损害了整体的利益，这违背了提升整体利益的初衷。

3. 切分出来的部分，不应该超出一个自然人的负载。当然对于每个人的能力不同，负载能力也不一样，需要不断的根据实际情况调整，这实际上就是运营。
4. 切分是内部活动，内部无任怎么切，对整个系统的外部应该是透明的。如果因为切分导致整个系统解决的问题发生了变化，那么这个变化不属于架构的活动。当然很多时候当我们把问题分析的比较清楚的时候，整个系统的边界会进一步的完善，这就会形成螺旋式的进化。但这不属于架构所应该解决的问题。进化的发生，也会导致新的架构的切分。

原则 2 是确保我们不能违反人性，因为维护自己的利益，是每个人的本性。只有权利和义务对等才能做到这一点。从原则 2 的也可以推理，所有的架构分拆，都应该是形成树状的结果，不应该变成有向图，更不应该是无向图。很多人一谈架构，必谈分层，但是基本上都没意识到，是因为把一个整体分拆为了一棵树，因为有了树，才有层。

从某种意义上来说，**谈架构就是谈分层，似乎也没有错，但是还是知道为什么比较好**。从根节点下来，深度相同的是同一层。这个是数学概念，我就不展开了，感兴趣可以去复习一下数学。

同样我们看一个组织架构，也可以做一个粗略的判断，如果一个企业的组织架构出现了“图”，比方说多线汇报，一定是对 stakeholder 的利益分

析出现了问题，这就会导致问题 2 的发生。问题 2 一旦出现，我们必须马上要意识到，如果这个问题持续时间长，会极大的降低企业的运作效率，对相关 stakeholder 的利益都是非常不利的，同样对于企业的利益也是不利的。必须快速调整相关 stakeholder 的职责，使得企业的组织架构成为一个完美的树状，并且使得数的层数达到尽可能的低。只有平衡数可以比较好的达到这个效果。

当然如果某个节点的能力很强，也可以达到减小树的高度的结果。技术的提升，也是可以提升每个节点的能力，降低树的层数。很多管理学都在讨论如何降低组织架构的层数，使得管理能够扁平化，原因就在于此，这里就不展开讨论了。从这里也基本可以得出一个结论，一个好的组织的领导，一定也是一个很好的架构师。

切分与建模

实际上切分的过程就是建模的过程，每次对大问题的切分都会生成很多小问题，每个小问题就形成了不同的概念。这也是系列第二篇文章尝试表达的。这些不同的概念大部分时候人们自发的已经建好了，架构师更多的是要去理解这些概念，识别概念背后所代表的的人的利益。比如人类社会按照家庭进行延续，形成了家族，由于共享一片土地资源，慢慢形成了村庄，村庄联合体，不同地域结合，形成了国家。由于利益分配的原因，形成了政权。每次政权的更迭，都是利益重新分配的动力所决定的。

同样对于一个企业也是一样的，一开始一个人干所有的事情。当业务量逐渐变大，就超过了一个人能够处理容量，这些内容就会被分解出来，开始招聘人进来，把他们组合在一起，帮助处理企业的事务。整个企业的事务，就按照原则 2，分出来了很多新的概念：营销，售前，售中，售后，财务，HR 等等。企业的创始人的工作就变成了如何组合这些不同的概念完成企业的工作。如果业务再继续增大，这些分出来的部分还要继续分拆，仍然要按照原则 2 才能够让各方达到利益最大化。如果某个技术的提升，提高了某个角色的生产力，使得某个角色可以同时在承担更多的工作，就会导致职责的合并，降低树的层数。

切分的输出和组织架构

架构切分的输出实际上就是一个系统的模型，对于一个整体问题，有多少的相关方，每个相关方需要承担哪些权利和义务，不同的相关方是如何结合起来完成系统的整体任务的。有的时候是从上往下切（企业），有的时候是从下往上合并，有的时候两者皆有之（人类社会的发展）。而切分的结果最终都会体现在组织架构上，因为我们切分的实际上就是人的利益。

从这方面也可以看出，任何架构调整都会涉及到组织架构，千万不可轻视。同样，如果对于 stakeholder 的利益分析不够透彻，也会导致架构无法落地，因为没有入愿意去损坏自己的利益。一旦强制去执行，人心就

开始溃散。这个也不一定是坏事，只要满足原则 2 就能够很好的建立一个新的次序和新的利益关系，保持组织的良性发展，长久来看是对所有人的利益都有益的，虽然短期内有对某些既得利益者会有损害。

总结

1. 架构的切分的导火索是人的负载太重。
2. 架构的切分实际就是对 stakeholder 的利益进行切分或合并，使得每个 stakeholder 的权责是对等的，每个 stakeholder 可以为自己的利益负责。
3. 架构切分的最终结果都会体现在组织架构上，只有这样才能够让架构落地并推进。
4. 架构切分的结果一定是一个树状，这也是为什么会产生分层。层数越多沟通越多，效率越低，分层要越少越好。尽可能变成一颗平衡树，才能让整个系统的效率最大化。

5

什么是软件

本文是漫谈架构专栏的第五篇，作者将会从自己的认知角度再次反思什么是软件，文中作者探讨了软件发展火热的根本原因以及软件扮演的角色等问题。如前几天一位架构师所说，我们并不缺架构实践，而是缺少对于架构的反思，希望这系列文章能帮你重新理解架构，重新认识软件。

前面四章，把什么是架构，如何做好架构等必要的概念澄清了一下。这些概念对于在各种不同的领域都应该也是有用的，需要读者自行思考，并应用到自己所在的领域中。在这篇文章开始，我们用同样的思考，来看看软件是怎么回事，以及如何运用架构思维，更好的设计和实现软件。

冯诺依曼结构，图灵机，以模拟人为目标

软件的历史，实际上可以说是用机器模拟人的历史。不管大家（包括在这个历史过程中的参与者）有没有意识到，我们都有意无意的在计算机上模仿人类的行为。从冯诺依曼结构开始，程序逻辑开始脱离硬件，采用二进制编码。加上存储，配合输入输出，一个简化的大脑就出现了。图灵机则是模拟大脑的计算，用数学的方式把计算的过程定义了出来，著名的

邱奇-图灵论题：一切直觉上能行可计算的函数都可用图灵机计算，反之亦然。软硬件两者一结合，一个可编程的大脑出现了，这也是现在为什么我们把计算机叫做电脑。在硬件上编写出的程序，就是软件，是用来控制硬件的行为的。

成本为王

在初期，软件使用二进制编写的，从硬件到软件，成本都非常的高。随着半导体技术的进步，硬件的成本越来越低，性能越来越高，甚至出现了**摩尔定律**：当价格不变时，集成电路上可容纳的元器件数目，约每隔 18-24 个月增加一倍，性能提升一倍。软件方面，为了简化难度，开始采用汇编，进一步出现了类似于人类的语言的高级语言，比如 C/C++/Java 等，这使得人类可以用类似于人的语言来和计算机沟通。软件工程师慢慢越来越多，开发软件的成本也越来越低。计算机就好像是一个只需要电，不需要休息的人，可以无休无止的工作。

人们越来越愿意把原来只有人才能做的事情，交给计算机来做。结果就导致软件越来越丰富，能够做的事情也越来越多，成本也越来越低。可以这么说，成本是我们为什么采用软件的主要动力，可以节省大量的人员培训，减少雇员的数目。随着互联网的发展，人类社会也开始软件化了。原来必须实体店来进行售卖的，搬到互联网上，开店成本更低，并且能够接触到更多的人。想象一下，一个门店每天的人流达到百万级别是很恐怖

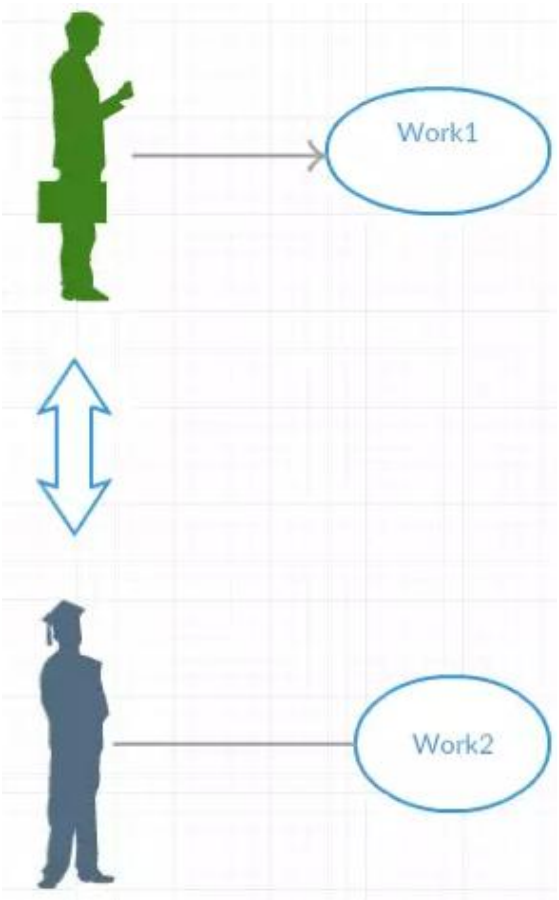
的，由实体空间大小来决定。但是在互联网上，访问量千万级别都不算什么。最终的结果就变成，每个人能够负担的工作越来越多，成本越来越低。这也是为什么软件这么热的原因。

软件扮演的角色

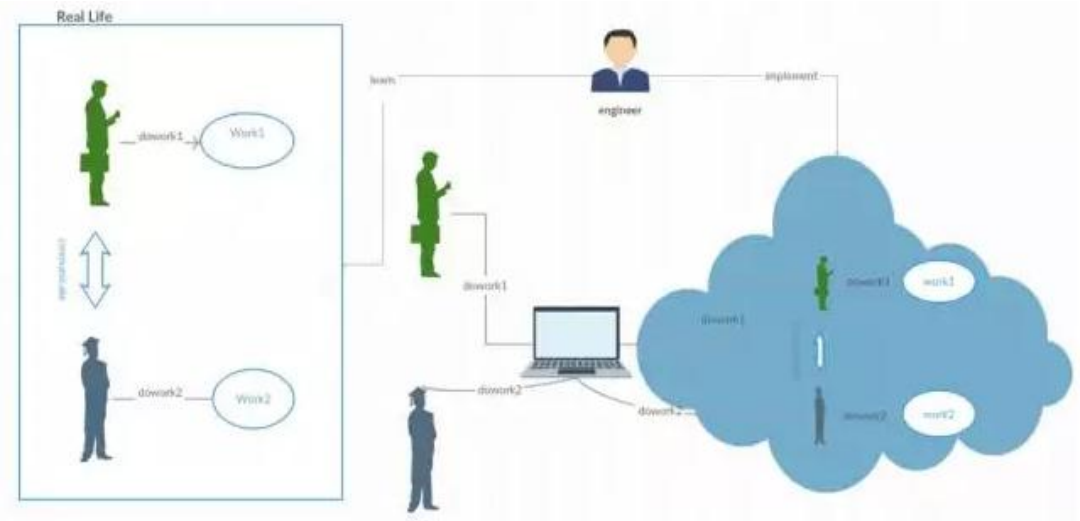
随着软件的规模的变大，做好一个软件也变得越来越难了。早期的程序员写程序，主要是为了帮助自己研究课题。这些程序员熟练了之后，提高了自己的生产力，并发现还可以帮助别人写程序，慢慢软件就变成了一个独立的行业。**程序从早期由一个人完成，也逐渐变成了由很多不同角色的人共同合作来完成**。以下讨论的前提，都是基于帮助别人写程序，多人合作的基础上的。结论对于单人为自己写程序也适用。

在没有软件之前，每个人干自己的工作，自行保存自己的工作结果。人们面对面或者通过电话等沟通，如下图所示。

有了软件之后，实际上，我们是把我们日常生活中所做的事情，包括我们自己本人都一起虚拟化到了计算机中。而人则演化成了，通过计算机的输入输出设备，控制计算机中的自己，来完成日常的工作，以及与其他人的沟通。也就是说，软件一直以来的动力，始终都是来模拟人和这个社会的。比如模拟大气运动（天气预报），模拟人类社会（互联网社交），模拟交易，包括现在正在流行的 VR，人工智能等等。模拟的对象越来越高级，难度越来越大。



不管如何发展，模拟人的所有行为都是一个大的趋势。也就是说，软件的主要目的，还是把人类的生活模拟化，提供更低成本，高效率的新的生活。从这个角度来看，软件主要依赖的还是人类的生活知识。软件更多的是扮演一个 cost center，这也是为什么会出现很多的软件代工。



软件开发的架构演变

软件工程师是实现这个模拟过程的关键人物，他必须先理解人是怎么在日常生活中完成工作的，才能够很好的把这些工作在计算机中模拟出来。可是软件工程师需要学习大量的计算机语言和计算机知识，还需要学习各行各业的专业知识。软件工程师本身的培养就比较难，同时行业知识也要靠时间的积累，这样就远远超出了软件工程师的能力了。**所以软件开发就开始有分工了，行业知识和业务的识别**，会交给 BA，系统的设计会交给架构师，设计的实现交给架构师，实现的检验交给测试，还有很多其他角色的配合。为了组织这些角色的工作，还有项目经理。这就把原来一个人的连续工作，拆分成了不同角色的人的连续配合，演化成了不同的软件开发的模式。然后慢慢演变出专门为别人开发软件的软件公司。

软件架构的出现

如同前面描述的架构的定义，软件架构的出现也是同样的。一开始是懵懵懂懂的去写软件，后来慢慢的就有意识的去切分，演变成了不同的架构。这个背后的动力也是一样的，就是**提升参与的人的利益**，降低成本。导火索也是软件工程师的任务太重，我们需要把很多工作拆分出来。拆分的原则也是一样的，如何让权责一致。同样，这个拆分也是需要组织架构的调整，来保证架构的落地。具体如何分拆，如何调整，我们将在另外一篇中着重讨论。

以上通过简单的描述计算机和软件的发展历史，阐明软件的本质，其实就是通过把人类的日常工作生活虚拟化，减少成本，提升单个人员的生产力，提升人类自己的利益。软件工程师的职责在这个浪潮中，不堪重负，自然而然就分拆为不同的角色，形成了一个独特的架构体系。这一切的背后，仍然是为了提升人类自己的利益，解决人类自己的问题。

6

软件架构到底是要解决什么问题

本文是漫谈架构专栏的第六篇，作者 Kevin 继续沿着前几篇文章的思路，探讨了软件架构为什么要有软件架构，进而再去解释什么是软件架构。这和最近网上疯传的黄金圆环（Why-How-What）思路非常贴合。

第五章简述了什么是软件。那么什么是软件架构呢？按照惯例，我们来看看是什么问题，是谁的问题。

要解决谁的问题？

如前所述，软件实际上就是把现实生活模拟到计算机中，并且软件是需要在计算机的硬件中运行起来的。要做到这一点需要解决两个问题：

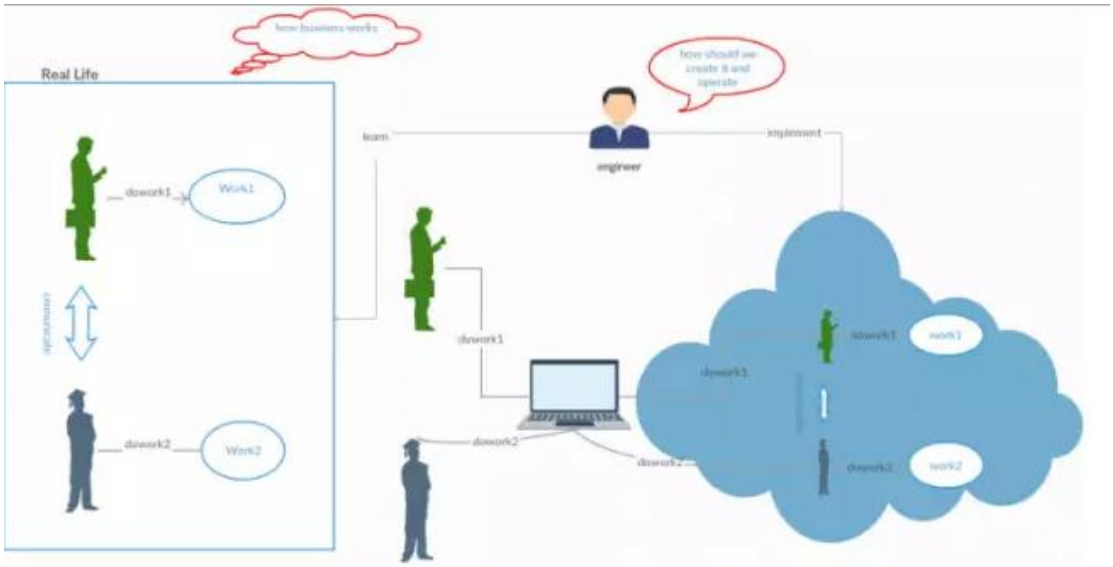
一、业务问题

具体的现实生活状态下，没有软件的时候，所解决的问题的主体是谁，解决的是什么问题，是如何解决，如何运作的？

二、计算机问题

1. 如何把现实生活用软件来模拟？

2. 模拟出来的软件，需要哪些硬件设施才能够满足要求？ 并且当访问量越来越大的时候，软件能否支持硬件慢慢长大，性能线性扩展？
3. 因为硬件是可能会失效的，软件如何在硬件失效的情况下，仍然能够保证可用性，让用户能够不中断的访问软件提供的服务？
4. 怎么收集软件产生的数据，为下一阶段的工作提供依据？



分别是谁的问题

1. 业务的 owner 需要提升业务的效率，降低业务的成本，这是动机。这个实际上就是业务的问题，所以一般软件开发的出发点就在这里。
2. 是软件工程师的问题，要解决业务 owner 把业务虚拟化的问题，并且要解决软件开发和运营的生命周期的问题。

分别有什么问题

1. 业务问题的本质，是业务所服务的对象的利益问题，明白了这个，就很容易搞清业务的概念和组织方式。再次强调一下，有了软件，可以降低业务的成本，没有软件的情况下，业务是一样跑的。如果只是为了跟风要用软件，说不定反而提高了成本，这个是采用软件之前首先要先搞清楚的。我们经常说软件和技术是业务的 enabler，实际就是把原来成本很高的降到到了很低的程度而已，并不是有了什么新的业务。另外软件也不是降低业务成本的唯一方式。
2. 为了能够让软件很好的跑起来，软件工程师必须理解业务所服务的对象，他们的利益所在，即业务问题。业务面对这些问题是如何分拆解决的？ 涉及到了哪些概念？ 这些概念分别解决了哪些哪些问题？ 我们不能自己按照自己的理解，用自己的一套概念体系来表述。如果这么做的话，会导致两个问题：
    - 业务无法和我们交流，因为他们无法明白我们所自己创建的概念，所以他们无法确认我们的理解是否正确。
    - 我们所表述的东西，并没有在实际生活中实践过，我们也不知道这些概念是否能够解决业务的问题。
3. 软件工程师还必须要考虑，用什么样的硬件把软件跑起来，怎样跑得好，跑得快，并且可以随着业务的流量逐渐的长大？

分析问题

对于 2，在有限的时间下，软件工程师毫无疑问无法一个人去完成这么多事情，那么我们需要把所做的事情列出来，进行分析。

一、虚拟化业务需要完成这些事情

1. 学习业务知识，认识业务所涉及的 stakeholders 的核心利益述求，以及业务是如何分拆满足这些利益述求，并通过怎样的组织架构完成整个组织的核心利益的，以及业务运作的流程，涉及到哪些概念，有哪些权利和责任等。
2. 通过对业务知识的学习，针对这些概念所对应的权利和责任以及组织架构，对业务进行建模，把并把建模的结果用编程语言实现。这是业务的模型，通常是现实生活中利益斗争的结果，是非常稳定的。
3. 学习业务所参与的 stakeholder 是如何和业务打交道，并完成每个人的权利和义务的，并通过编程语言，结合业务模型实现这些打交道的沟通通道。这部分是变化最频繁的，属于组合关系。明白了这一点，对后续的实现非常有帮助。
4. 如何把业务运行的结果，持久化，并通过合适的手段把持久化后的数据，在合适的时间合适的地点加载出来。这部分和基础设施有关，变化可能也会比较频繁。

二、代码如何运营，需要完成这些事情

1. 需要多少硬件设备来满足访问的需求？
2. 代码要分成多少个组件部署到哪些硬件设备上？
3. 这些代码如何通过硬件设备互相连接在一起？
4. 当业务流量增大到超过一台机器的容量时，软件能否支持通过部署到新增机器上的方式，扩大对业务的支撑？
5. 当某台或某些硬件设备失效时，软件是否仍然能够不影响用户的访问。
6. 软件运行产生的数据，能否支持提取出来并加以分析，为下一轮的业务决策提供依据。

三、如果分成不同的角色来完成这些事情，就需要一个组织架构来组织代码的编写和运营，需要做哪些事情

1. 完成一和二所列的这些事情，需要哪些角色参与？
2. 这些事情基本都需要顺序的发生，如何保证信息在不同角色的传递过程中不会有损失？ 或者说即使有损失，也能快速纠正？
3. 这些角色之间是如何协调，才能共同完成虚拟化业务的需求？

会生成哪些架构

如果业务足够简单，用户流量够小，时间要求也不急迫，那么一个人，一台机器就够了，这个时候一般不会去讨论架构的问题。当访问的流量越来越大，机器就会越来越多，代码的部署单元就会拆分的越来越多。

同样就会需要越来越多的人来完成拆分出来的越来越多的部署单元，甚至同一个部署单元也需要分拆为多人合作完成。但是我们需要注意到一点，整个的概念体系，或者说业务的建模不会有任何的变化，还是完成同样的这些事情。唯一的区别就是量越来越大，超过了单个人和单个机器的容量，不断地增长。这样就会导致以下的架构：

1. 当流量越来越大，我们就会发现，软件所部属的机器就会开始按照树状的结构开始分拆，就会形成硬件的部属架构。这就是为什么会形成部署的分层。
2. 为了把业务在软件中实现并落地，需要前端人员、业务代码人员、存储层等不同技巧的人同时工作，需要切分成代码的架构。这就是为什么会形成代码的分层，形成代码的架构。当然，当这些角色由一个人来完成的时候，不一定会有代码架构，往往会比较乱。
3. 当参与的人员越来越多，就会形成开发体系的组织架构。因为代码开发的过程是一个连续的过程，会用流程来吧不同的角色串联起来，这就是软件工程。

4. 为了完成业务的工作，需要识别出来业务架构和支撑业务的组织架构，以及业务运作的流程。这是被虚拟化的业务架构和组织架构，也需要体现在代码中，保持和现实生活中一致。

什么是软件架构

这就是软件比较复杂的地方，涉及到软件本身的业务体系，和所虚拟的业务体系。根据以上的分析，所生成的架构，究竟那些算是软件架构呢？

1. 软件因为流量增大而分拆成不同的运行单元，在不同的机器上部署所形成的架构，属于软件架构。
2. 每个运行单元为了让不同角色的人，比如前端，业务，数据存储等能够并行工作，所分成的代码架构，也属于软件架构。

所以当我们说软件架构的时候，我们一定要讲清楚，究竟说的是部署的架构，还是代码的架构。软件架构的落地，需要软件的组织架构和流程来保障，离开了这个，软件架构是一句空话。

另外很多人讲，架构是进化出来的。架构实际上是在量不断的增大，超过了单台服务器的容量，逐渐的分拆，同时导致超过单个人员的能力，工作人员不断的增多，工作内容不断的分拆形成的。这本身就是架构的意义所在。不管怎么分拆，所达到的目标没有任何变化，就是完成业务在计算机中的虚拟化。

7

架构师没有话语权，还架什么构

本文是漫谈架构专栏的第七篇，作者 Kevin 探讨了什么是架构师、成为架构师的前提条件、如何发现“是谁的问题”、架构师的权利和义务等话题。正如作者所说，**架构师必须是一个组织的领导人**，有权利调动这个组织的架构，才能够更好的发挥架构师的作用，更好的把利益的调整落到实处。

什么是架构师

在之前的几篇文章中，经常会提到架构师这个词。我们已经定义了什么叫架构，那怎么定义架构师呢，是不是做架构的就叫架构师了？ 没有这么简单，本篇尝试讨论一下这个问题。

架构师的前提条件

如果一个人在工作中，只是致力于完成自己的工作，以做好自己的工作为主要目标，那么最多只能成为一个工匠，无法成为一个架构师。因为

这个过程解决的还是自己的问题，并没有时间的压力，可以随意什么时候做完都可以。

当我们所做的工作是处于社会的分工的一环，**需要帮助别人解决问题，并且按时解决别人的问题成为我们自己的问题的时候，我们就有了时间压力，潜意识里会自然而然的有一种对时间的恐惧**。这个恐惧在潜意识里面会想方设法推动我们采用各种手段，以便及时的完成工作，换取报酬。甚至会加班加点，不择手段。

如果我们还生活在这个恐惧下面，是不可能成为架构师的。要成为架构师，必须要超越这个恐惧才能够看清楚，我们要解决的是别人的问题，不是自己完成工作的问题。因为仅仅是完成了自己的工作，也并不一定就解决了别人的问题。如果别人的问题没有解决–即使我们认为自己的工作完成了–我们的工作实际也没完成，因为我们工作是否完成，是别人说的算的，不是我们自己。

为什么会有这个对时间的恐惧和压力呢？这是因为我们把完成自己的工作当成了我们的最大利益。如果别人的问题没有真正的解决，必然会觉得付出的报酬不值得，我们的利益实际上是受损失了。这和我们所以为的恰恰相反，因为我们所能得到的工作只会越来越少，别人会越来越不愿意依赖于我们。

另一方面也说明，我们对自己所从事的工作，还没有足够的自信，我们解决自己的问题还有困难，才会这么在意，并恐惧。如果我们把完成别人工作当成自己的最大利益，这个对时间的恐惧自然就会消失，这个时候就自然而然的开窍了，就知道怎么去发现问题了。只有做到这一点，才能在自己所服务的领域建立起自信，成为一个合格的架构师。

其实刚开始一般是硬着头皮去克服对时间的恐惧和压力的，一点自信都没有。但只要做成功了一次（只要真的舍得这么去做了，想不成功也很难！），自信就开始慢慢建立起来了，这个时候就是我们开始慢慢变成架构师的时候。大家就当着上当一回，试试看。

如何发现“是谁的问题”

当我们真正专注于别人的问题的时候，我们自己的理想，抱负，对技术的追求都不算什么了。这些理想，抱负，对技术的最求，不就是要达到自己的利益吗？ 只有帮助别人解决了问题，这些理想，抱负，对技术的追求才可能实现，否则这些理想，抱负，对技术的追求有什么意义，能得到什么利益？

这个时候就会真正的开始思考，别人究竟有什么问题。其实也很简单，和我们自己面临的问题一样，别人的问题也都是如何获取更好更多的利益。我们自己想明白了这一点，自然也就能想明白别人的问题。这个时候就能够问出正确的问题：**如果问题不解决，究竟谁会有利益的损失？ 如**

果问题解决了，究竟谁会有收益，谁的收益最大？ 回答了这两个问题就找到了问题的主体。只回答一个是没有用的，因为很多时候这个世界的事情，权责是不对等的。明白了这两个问题，我们只要让事情权责对等起来，让每个人为自己的权利产生的结果负有义务，大部分时候我们自己根本就不需要做什么，问题就都解决了。这就是最高明的架构师。

架构师的权利和义务

架构师是要去平衡别人的利益，甚至会调整别人的利益的。**一旦架构师是全心全意的为别人的利益服务，自然而然的架构师就拥有了强有力的影响力，肯定会是一个 leader**。但是只是民意上的 leader 是没有用的，不能完全发挥架构师的能量。

架构师必须是一个组织的领导人，有权利调动这个组织的架构，才能够更好的发挥架构师的作用，更好的把利益的调整落到实处。所以**很多公司设了很多架构师的职位，但是并不具备调动组织架构的权利，那么这个架构师的职位一定是形同虚设**。架构师只能够通过建立某些流程来行使架构师的权利，比如强制架构 review，反而会造成很多内部不必要的冲突，最终都会导致这些流程流于形式，得不偿失。相信很多人都已经经历过这些，但似乎很少有人回去探讨这是为什么。

反过来，**具备架构师能力的组织领导人，一定是一个很好的领导，这个组织一定是很健康向上的，因为每个人的权利和义务就是比较均等的。**

并且这类领导对于组织成员权利和义务的对等状况会非常的敏感，会及时的调整组织架构，在问题发生之前就解决了。这样这个组织就会具备更好的抗压能力，能够更好的为这个组织的客户服务，这个组织的成员内心一定都是比较平衡的，每个人的能力都能够得到比较好的发展。当然读者可能又会说，这不是管理学的东西吗？ 是的，但也是架构的问题。所有架构的核心就是组织架构。或者也可以这样说，一个合格的组织领导人，一定必须是一个合格的架构师。

架构师的义务似乎不用说了，大家提的要求可能比我提的都高 – 当然是发现问题并且解决问题。架构师必须能够超越对时间的恐惧 –也就是说必须具备了一定程度的自信，哪怕是装的，去真正的发现问题的主体，识别真正的问题，并把这个行为变成为自己面对问题的第一反应。架构师还必须要明白，所给出的解决方案 – 架构的分拆、合并方案，只有让问题的主体的权责对等，才能够真正的解决别人的问题。一般明白了问题的主体，以及主体的利益所在，做到这一点也没有问题。

架构师和技术

很多人会问，特别是做软件行业的，架构师是不是需要学习技术，甚至是学习语言？ 如果一个架构师还有这个困扰—就如问这个问题的人，说明目前还不具备做架构师的能力，或者说还不具备对自己领域–哪怕是技术领域–的自信，更别谈业务领域了。

因为技术和语言，都是用来识别和解决所服务的主体的权责，保护并提升所服务的主体的权利的。特别对于软件领域来说，必须明白软件本身是怎么回事，解决什么问题，还要解决软件所服务的对象的领域本身是怎么回事，解决什么问题，这就要求更高了。语言和技术应该是随手拈来才对，对于架构师这些都是工具。学习技术和语言，如果明白了这些技术和语言要解决的是谁的问题，什么问题，学起来是非常快，非常容易的。

同样，采用哪个技术或者语言，只要某个技术或语言所解决的问题的主体，以及所解决的问题，和自己所面对的问题的主体和这个主体要解决的问题，这两者是匹配的，那么这个方案是成本是最低的，所采用的技术或者语言就是靠谱的。没有趁手的工具或语言的情况下，自己设计一个也不难，因为很清楚自己要什么。要不要自己做，无非是一个成本问题，也就是利益问题。并且从这个思路下去，选择的工具和语言肯定都是最简单的，成本是最低的。因为架构毕竟解决的还是人的利益问题，成本越低越好，这个成本当然是长期总体成本，不是眼前的短期成本。

8

从架构的角度看如何写好代码

本文是漫谈架构专栏的第八篇，作者 Kevin 举例介绍了如何写好代码。当我们有了好的架构，那就需要考虑如何将架构落地，而这个时候，代码就显得无比重要了！千万不要让代码成为架构扩展的瓶颈。文中作者提到了代码架构，细细品味吧。

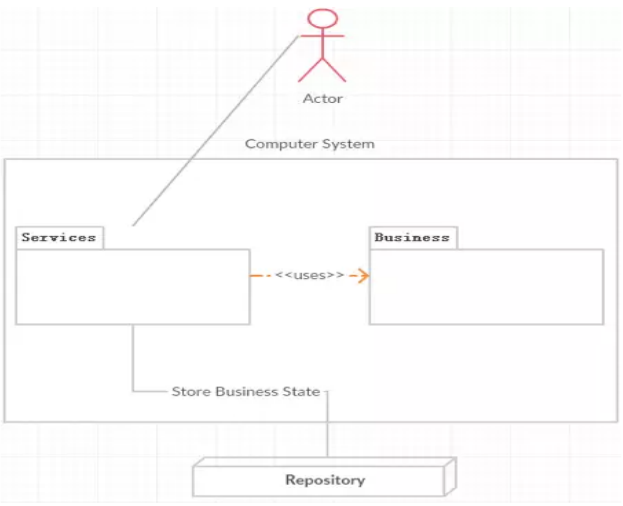
在第六章中，我们得出一个结论，软件架构实际上包括了：代码架构，以及承载代码运行的硬件部署架构。实际上，硬件部署架构最终还是由代码的架构来决定。因为代码架构不合理，是无法把一个运行单元分拆出多个来的，那么硬件架构能分拆的就非常的有限，整个系统最终很难长的更大。

所以我们经常会听说，**重写代码，推翻原有架构，重新设计等等说法，来说明架构的进化。这实际上就是当初为了完成任务，没有充分思考所带来的后果。**这也并不是架构进化的事情，而是个人对问题领域的逐渐深入理解的过程。所以有必要再讨论一下，代码的架构应该是怎样的。

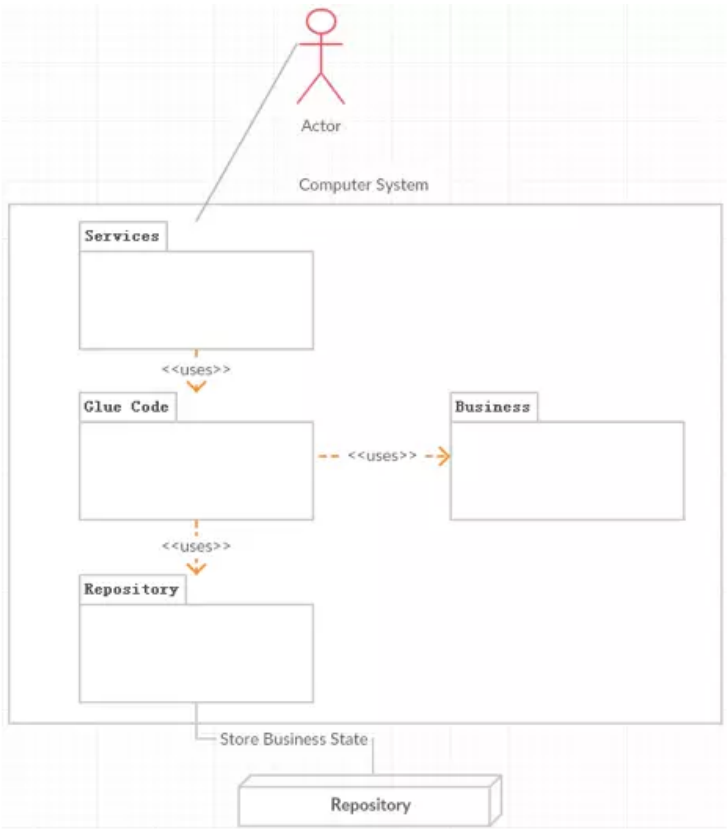
本文会在之前几篇文章的基础上，进一步探讨如何把架构的思考进行落地，细化到我们代码的实践当中，尽量不要让代码成为系统长大的瓶颈，降低架构分拆的成本。

在前面我们提到，**软件实际上是对现实生活的模拟，虚拟化**。这是一个非常重要的前提，直接决定了我们的代码应该分为几部分。结合每个部署单元所承担的责任，可以明确的拆分为两个不同的责任：

1. 表达业务逻辑的代码。很多人把这部分叫做 Domain Logic，或者叫 Domain Model。这部分实际是来源于生活的，必须保持和现实生活中的切分一致，并非人为的抽象而成。
2. 对用户提供访问并保存业务逻辑运行结果的代码。计算机的状态保存有一个缺陷，本机保留业务运行结果有很大的问题，一般都在外存储设备上保存，也便于扩展。所以单个部署单元的代码可以分为两个部分，如下图所示：



从这个图中可以看出，软件代码的相关利益人为运行时的访问人员和存储设备。而 service 的代码是最复杂的，需要服务于三方，代码人员的负担是最重的。为了把这三方的变化对 service 的影响降到最低，对于 service 还必须进一步的分拆为三个部分，让每一个部分都能够独立的变化，这样这三方的变化就不会产生连锁响应，降低成本。如下图所示：



这样，就划分成了几个责任：

1. Service 就专注于 user 的需求，并组合 Glue Code 提供的服务完成需求。

2. Glue Code 专注于组合 business 的调用，管理 Business 里面对象的生命周期，并且通过 Repository 保存或加载 Business 的状态
3. Business 专注于实现业务的核心模型。
4. Repository 专注于数据的保存，并和存储设备一一对应。

大家注意看，还是树形架构。并且左侧的主要需要计算机的相关理论知识，并且要直接面对用户的需求。右侧的更多的需要面对业务的核心。只要这几块的开发人员互相商量好了接口定义，这几个部分的开发就可以并行的进行，极大的提升开发的效率，缩短开发的时间。要做好这几部分，还需要注意，**逻辑只允许存在于 Business 中，Service、Glue Code、Repository 都不允许存在业务逻辑**。为什么呢？首先我们来看看什么叫业务逻辑。

什么叫业务逻辑

首先这个定义的前提是指软件代码中的逻辑，不是现实生活中的逻辑。**在软件代码中，不需缩进和计算的顺序调用，包括缩进的代码目的是 catch exception 的，都不算逻辑，除此以外都是逻辑**。以下用严格的顺序调用来指代这种代码。因为顺序调用是计算机的特性，由编译器来决定的，当然最本质的是因为我们计算的基础都是图灵机。在现实生活中，顺序调用也是逻辑，大家不要和我们这里说的业务逻辑相混淆。

为什么说除了 Business 代码中有逻辑以外，其他地方不能有逻辑呢？我们每个部分分别分析：

- 如果 service 里面不是严格的顺序调用，有很多分支，那么说明这个 service 做了两件或者两件以上的事情。必须把这个 service 分拆，确保每个 service 只做一件事情。因为如果不这么分拆的话，一旦这个 service 中的某各部分发生变动，其他的部分的执行必定会受影响。而确定到底有哪些影响的沟通成本非常高，其他相关利益方没有动力去配合，我们往往不会投入精力仔细评估。最后上线会出很多不可预料的问题，最终会导致损失用户的利益，并且肯定会导致返工，损坏自己的利益。如果是有计算的逻辑的话，比如受益计算，订单金额计算等，那么这部分应该是 Business 代码需要完成的，不能交给 service 代码来实现。
- Glue Code 里面如果不是严格的顺序调用，同理会和 service 一样遇到同样的问题。
- Repository 里面如果不是严格的顺序调用，包括存储访问的代码里面（比如 SQL），会导致逻辑进入到存储设备中。存储设备的主要目的是拿来存储的，一旦变成了逻辑计算的主体，就会导致存储设备无法通过增加机器的方式横向扩展长大。这个时候就没有架构了，

只能换性能更好的机器，这个叫 scale up。只有 scale out 才能算架构。

以上都会导致架构无法快速的横向扩展和分拆，并且增加了修改的成本，这些是不符合开发人员以及业务的利益的。

这么做的好处有哪些呢

1. Service、Glue Code、Repository 里面的代码是严格的顺序调用，那么这些代码只要做连通性测试即可，不需要单元测试。因为这些代码都需要和很多上下文打交道，很难做单元测试。这样才算是真正的组合。
2. Business 不访问任何上下文，不访问任何具体的设备，所以这部分代码是非常容易些单元测试的，并且单元测试必须 100%覆盖。因为其他地方没有业务逻辑，所以一旦有问题，就可以断定是 Model 的问题，单元测试肯定可以发现。如果单元测试没有发现问题，那么单元测试一定有问题。线上问题的模拟也就变得非常的简单，单元测试也能够得到进一步的补充。
3. Repository 很容易按照存储设备本身的最小访问粒度来完成工作，比如 DB，完全可以做到单表访问。因为这个时候存储设备只关心存取数据，完全和业务没有关系。做表的分拆也是非常容易的事情，存储设备通过增加机器就可以横向扩展长大。很多人会担心说，没

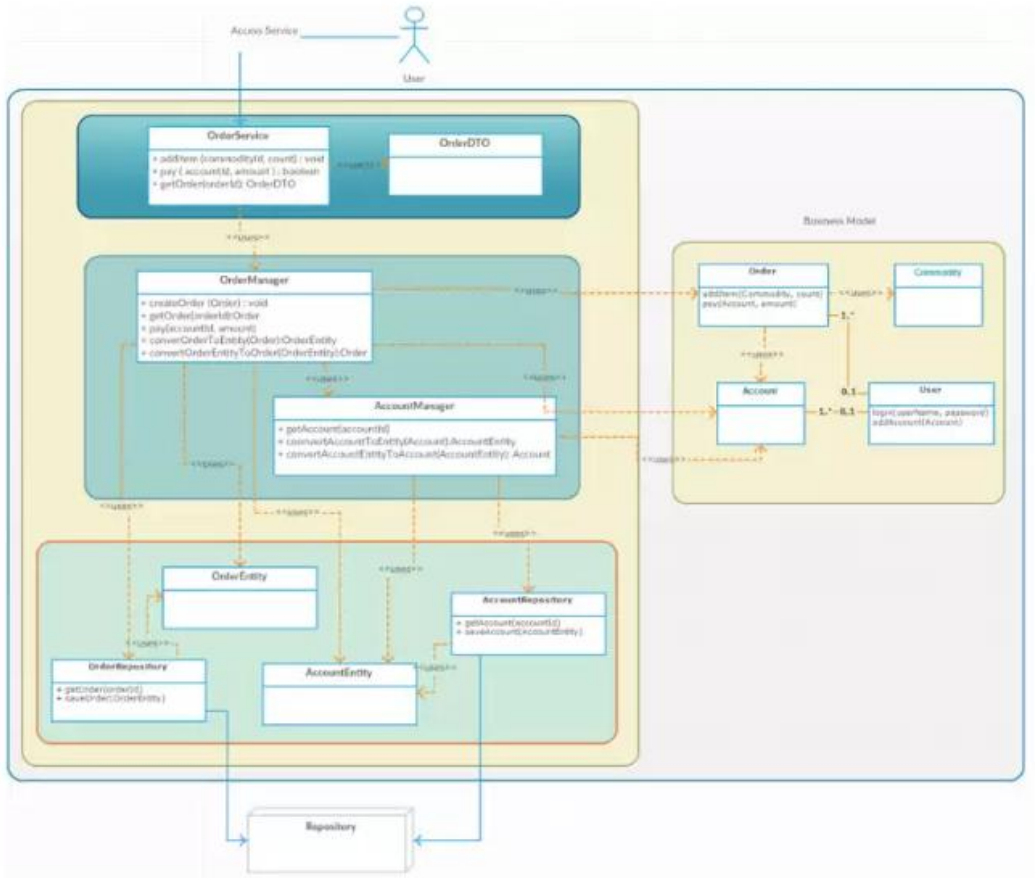
有了 join，访问 DB 的次数是不是更多了，会导致性能下降？ 按照现在网络的条件，网络访问和 Disk IO 访问的差距已经不大了，合理的设计下，多访问几次 DB 并不会导致这个问题。另外如果多台 DB 的话，还能通过并行加速访问。

4. 由于 Service、Glue Code、Repository 代码简单了，才可以让我们的开发人员投入更多的时间研究业务，毕竟这部分才是软件所真正服务的对象。

我们再来看一个实际的例子，如下图所示。

Manager 类实际就是 Glue Code。有几个注意点需要说明一下：

1. 不能把 Business Model 当做数据对象来处理，Model 关心的实际上是业务行为，数据只是是这些行为的结果。所以 Glue Code 需要把 Model 转换为 Entity，Entity 和存储设备里面的存储粒度一一对应。比如在 DB 中，每个 Entity 对应一张表，并且跟着表的变化而变化，这样就保证存储的变更不会影响 Model。同样 Service 和用户之间的数据交互，也是不会和 Model 之间相关的，确保用户的需求变化，不会影响到 Model。因为用户的需求变化是最频繁的，没有逻辑，可以让我快速的满足业务的需求。



2. 在 Service 这里，最好不要考虑代码重用。因为当多个不同的角色访问同一个接口，一旦某个角色的需求发生了变化，就会要求开发人员去修改。而这个修改往往会影响到其他的角色，需要这些角色一起配合来确定是否受影响，但是这些角色因为没有需求，往往不会配合。这样就给开发人员造成了很多不必要的沟通，成本是非常高的。最终都会导致线上 Bug，影响最终的用户。所以尽量给不同的角色不同的 Service，避免重用，降低沟通成本。很多人会说这样 Service 不就太多了吗？这样 Service 注册，查找等管理需求就出现

了，Service 治理中心就是来解决这个问题的。因为 Service 里面没有逻辑，所以开发和管理非常的简单，可以快速应对业务的变化。我们只有更快地变，更容易的变，才能更好地应对变。

3. Business Model 是必须要重用的，一旦发现重用出现问题，那么说明 Business Model 的识别出现了问题，这是一个我们要重新思考 Model 的信号。Business Model 必须是一个完美的树状，如果不是，也说明 Model 的识别出了问题。

在实际操作中，Service、Glue Code、Repository 不能有逻辑，实际上和很多人的观念是冲突的，认为这个根本做不到。做到这一点需要很多的学习成本，但是一定可以做得到。当发现做不到的时候，可以断定是业务的分析出了问题。比如不该合并的合并了，不该计算的计算了。这个问题一定有办法解决的，做不到都是理由，无非是想早点把自己的工作结束罢了。虽然刚开始会比较困难，一旦把这个观念变成自觉，开发的质量和效率马上就能高好几个级别。

我的游泳教练曾和我说过这些话，我至今记忆犹新：“**业余选手，越想从水里浮起来，就越想把头抬起来，身体反而沉下去。只有克服恐惧，把头往水里压下去，身体才能够从水里浮起来。真正专业的习惯往往是和我们日常的行为相反的**”。

我们真正想快速的完成代码工作，就要克服自己对时间的恐惧，真正的去研究业务的问题，相关 stakeholder 的利益，把这个变成我们的习惯。写代码的时候让该出现逻辑的地方出现逻辑，让不该出现的地方不能出现。一旦不该出现的地方出现了逻辑，那么要马上意识到，这个地方是一个坑，这个问题一定和业务的分析不透彻有关系。

很多人可能会把这个做法和 Martin Fowler 曾经提出过充血模型和贫血模型来比较，和 Domain Driven Design 来比较，其实没有必要。这个分拆完全是从软件所解决的问题，根据软件架构推导出来的，很多地方和两位前辈的观点是一致的，但是并不完全等同。

以上只是针对单一的 Service 部署单元的分析，扩展开去，对于其他的部署单元也是类似的。每个单元的下一级都可以认为是 Repository，每个单元的上一级都可以认为是 User。这些实践在我自己的项目中都有用到，非常的有效，迭代的速度非常的快。很多人担心 Business Model 建不好，其实没关系，刚开始可以粗糙一点，后续可以慢慢的完善。这个架构已经隔离好了每个部分的变化对其他部分的影响，变化成本都在可控的范围之内。

9

你理清技术、业务和架构之间的关系了吗

本文是漫谈架构专栏的第九篇，作者 Kevin 以钻木取火为切入点，深入介绍了技术、业务和架构之间的关系。正如作者所说，技术总是在人类解决对业务的要求不断提高的情况下产生，目的也是为了获取更大更好的利益。

某天和朋友吃饭正好聊到这个话题。作为架构师或者做技术的人，在开发软件时，我们基本上就是在**扮演上帝的角色**：我们不但要创建出一个个的程序，还要让这些程序能够脱离我们在硬件上独立运行，以便为这个程序所服务的群体提供服务。当这个程序出现问题甚至 bug 的时候，我们还得扮演牧师的角色去修复这些问题。这不正是一个程序的社会吗？ 和人类社会的演变何其相似！那么我们自然也能够拿人类演变的历史来指导软件开发工作，以避免再经历一次像人类演变发展那么痛苦的过程了。由此我们也可以看出，架构师和程序员们都在扮演着多么重要的角色，如果还在解决自己的问题，怎么扮演好上帝这个角色？

在软件设计开发的过程中经常会看到，很多所谓的架构讨论实际上只是在讨论某种技术。在很多人的概念里面，架构和技术实际上是等同的。

学会了几种技术，就认为自己是架构师了，**甚至是学习的技术越多，就觉得自己的水平越高**。这样实际上是对自己很不负责任的。要知道任何技术都是为了解决某种问题而存在的，学会了技术，并不代表自己能够解决问题，这一点非常的重要。学会的技术的多少，所带来的差别只是自己解决问题的手段多了罢了。但是手段多了就一定是好事吗？ 很多时候，学习的技术越多，越不知道采用哪种技术好，所谓“乱花渐欲迷人眼”。

还有另一种很普遍的观点：技术人普遍看不起业务，认为技术更高端，而业务太低端，并且业务往往喜欢给技术挖坑。业务则觉得技术眼光高，但是实际解决不了问题，总是理解有偏差，但是又无可奈何，因为自己不会。

本篇文章尝试从这里入手，分析一下这三个概念到底有什么关系，我们应该**怎么处理业务、技术还有架构的关系**。

什么是技术

当我们一无所有，或者什么都不会的时候，这个时候实际上是没有技术的。就好比人类在最早期，什么都得用自己的双手来干活。一旦我们在日常生活中无意间发现某些规律的时候，我们就可以通过创造条件，让这个规律重复的发生。通过人为创造条件，让指定的规律按照人类的意愿发生，这就是技术。比如取火，最早人类只能靠打雷等自然现象产生火。

取火其实就是一个业务目标，要解决的是人类自己的问题，这就是业务，实际就是人类的利益。这个时候人类没有生火的技术，只能靠不断的加木材，保持火不熄灭。后来人们发现了钻木取火：只要用一个干的木棍，在另一个干木表面快速的转动，就可以生火。这个办法让人类可以自行创造火源，就产生了钻木取火的技术。

但是双手快速转动木棍钻木取火，并不是所有人都能够做得到的，需要很多力量和速度，对人的要求太高。为了解决快速转动的问题，就有人采用弓弦来提升木棍转动的速度。

也就是说：

1. 业务目标是为了取火，钻木取火这个技术的出现解决了这个问题。
2. 钻木取火的效率不高，影响了业务（取火）的效率，就有了进一步改进的动机，改进转动木棍的方式，产生了弓弦转动木棍的技术。

技术与架构，以及与业务之间的关系

技术总是在人类解决对业务的要求不断提高的情况下产生，目的也是为了获取更大更好的利益。所以：

1. 技术是为了解决业务的问题而产生的，没有了业务，技术就没有了存在的前提。
2. 有了更好的技术，效率更差的技术，就会慢慢的被淘汰，消失，一切都遵从人类的利益诉求–也就是业务。有人会问，不用钻木取火了，但是弓弦加速转动木棍还可以用啊？ 没错，因为弓弦转动木棍这个技术，不是来生火的，是用来加速木棍转动的，所解决的问题不一样。但是两种不同的技术，合理结合起来，会更好更有效率的解决业务问题。

所以技术与技术之间，有两种关系：

1. 在解决同一个业务问题的前提下，更高效，更低成本的技术，会淘汰低效，高成本的技术。这是人类利益诉求所决定的。
2. 一般刚开始解决根本问题的技术（钻木取火）的效率是比较低的，只是把不可能变成了可能（从这一点上来说，技术才是业务的enabler）。然后就会有提高效率的需求出现，要求改进这个技术。这个技术的低效率部分就会被其他人（或者技术发明人自己）加以改进，这部分就会形成新的技术。

当关系 2 发生的时候，这个地方必定会形成一个切分，新技术会通过某种方式和原有的技术连接在一起形成一个整体，让这个新的技术可以和原有技术共同工作，使得原有的技术可以用更高的效率解决问题。因为要解决的主要问题（生火）并没有发生改变，分拆所形成的是一个树状的结构。

按照前面的架构定义，这个时候其实已经产生了架构。也就是说，一般是先有技术，才会有架构。这些其他技术（弓弦拉动木棍），是从直接解决问题的初始主要技术中分拆出来形成的，并通过树状结构和主要技术（钻木取火）组合在一起。在解决主要问题（生火）之后，再开始逐渐的分拆为更为细粒度的技术（弓弦转木棍）。

而这个细粒度的技术（弓弦转动木棍）往往不会和业务的主要目标（生火）发生直接的关系。不同的技术，通过树状结构，组合在一起，形成了一个完整的架构解决方案，共同完成业务的目标。这就是技术，业务和架构之间的关系。很多人把这个过程称为架构的进化，我更愿意把这个过程称为技术的进步所导致的新的架构分拆，因为这个过程内在的动力，更多的是来自技术对解决业务问题的解决。

技术人员和业务人员的关系

为什么技术人员总是和业务人员发生冲突呢？ 这是因为技术人员很多时候关心的技术，和业务的主要目标往往不是直接对应的，业务也是负

责某一部分的业务，也不是和业务的主要目标直接对应的，都是树的分支节点（上文已经解释了为何会发生这种情况）。只有直接解决业务问题的那个技术（或业务）–树的根节点–会和业务直接相关。所以一旦产生冲突，一般必须两个根节点（一般都是领导）碰面才能解决问题，就是这个原因–他们都知道业务主要目标。这也是为什么下层无法理解上层，而上层都喜欢下军令状，要求下层执行。人只有尽量去理解上层的问题才能做下层的分拆。

在软件行业，这个根节点技术就是软件。这也是为什么架构师要认识什么叫软件，软件解决谁的问题，什么问题，软件本身又是怎么分拆的，才能够更好的组合不同的技术，完成业务的目标。而软件里面和业务直接相关的，只有 Business Domain 这一部分。

用人来打比方，Business Domain 相当于人的大脑，而 Service，Repository，Glue Code 等部分所采用的技术，全部都是计算机自己领域的技术，都是为了能够让程序跑起来，相当于人的四肢。我们大部分开发人员的工作主要专注于四肢部分。我们真正应该投入的是大脑部分。因为大脑能够决定四肢长什么样，而不是反过来。很多架构师、技术人员主要专注于计算机相关的技术，忽略了业务本身，甚至看不起业务，这也是为什么技术总是和业务冲突的原因。

架构师应该承担起解决业务问题的这个角色来，专注于 Business Domain 和软件本身的架构，让技术人员致力于为业务在计算机中跑起来而努力。只有把这两者很好的结合起来，才能更好地完成业务的目标，才会让软件更好地服务于大家。最终一定会得到一个很好的软件架构，令软件开发团队和业务部门都能够很好地开展工作并降低成本。

重新发明轮子

当现有已经存在很多技术，而这些技术却和我们所要解决的问题并不是那么直接对应的时候，我们就需要有意识的组织和识别不同的技术，来实现业务的目标。这个时候组织的方式有很多种，其中成本最低的方法就是按照要达成的目的和当前的问题，从上到下进行架构分拆。分拆出来的更细粒度的问题，分解到不同的人来进行解决，就形成了业务架构和组织架构。解决这些问题就需要组合很多不同的技术，那么应该采用哪些技术？还是自己重头实现一个？ 自己实现一个—这就是很多人所谓的重新发明轮子。以下试着分析一下：

1. 当技术所提供的能力远远超过需要解决的问题时，往往掌握技术和维护技术会成为瓶颈。因为越复杂的技术，成本越高。如果自己实现一个仅仅是解决当前问题的方案，可能成本反而更低。这也是为什么很多大型的互联网公司不断地开源出来自己的技术的原因。而这些技术对于我们来说是否适用？他们原本是用来解决谁的问题

的？什么问题？如果不清楚这些，就冒然采用，可能会导致更高的成本。

2. 当技术所提供的能力和我们所要解决的问题部分匹配时，还是要看成本。比如当我们需要一个锤子的时候，手边正好没有，但是却有一只高跟鞋，勉强也可以替代锤子。但是长期来看，这么用不划算，因为高跟鞋的价格比锤子高很多，耐用性差很多，维护成本也高很多。

所以，准确识别采用什么技术的能力，也是架构师所要具备的能力之一。考虑的主要因素也是长期的成本和收益。