促进软件开发领域知识与创新的传播

架构师 ARCHITECT

热点 | Hot

英特尔、谷歌和AWS回应CPU安全事件

推荐文章 | Article

开发者需要知道的软件架构五件事

阿里盒马领域驱动设计实践

观点 | Opinion

今天我们还需要关注DDD吗？

专题 | Topic

如何规范公司所有应用分层

Geekbang 极客邦科技 | InfoQ

2018年2月

CONTENTS / 目录

架构师

2018 年 2 月刊

本期主编 蔡芳芳

流程编辑 丁晓昀

发行人　霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

卷首语

工业为什么需要人工智能?

作者 天泽智云首席技术官 刘宗长

虽然大数据和人工智能接连在安防、金融、电商等领域中获得突破，但似乎在工业一直难以落地。相比于已经厮杀成为一片红海的金融和消费等领域，工业似乎是一片还未被开发的处女地，成为许多公司竞相追逐的新目标。在过去的几个月中我们看到工业人工智能的赛道上正在变得热闹起来，入场的玩家们既包括了阿里、腾讯等互联网巨头，也有 GE、西门子、三一等老牌工业企业，还有像天泽智云这样的初创型企业。

工业为什么需要人工智能？一个重要的原因是人的知识产生和利用效率已经难以满足生产系统的要求，依靠人的知识和经验去驱动生产系统已经达到了生产力的边界，难以使其以最优的效率运行和协同。受人的经验和知识的限制，以人的决策为驱动生产系统中有很大一部分的价值并没有被释放出来。这主要体现在对知识获取的速度、能力的深度、应用的规模化三个方面的瓶颈。因此，第四次科技革命所要解决的核心问题主要包括:

1. 提升知识作为核心生产要素的生产力边界，使知识的产生、利用、和传承过程中的效率化和规模化，从而获得本质的提升。

2. 重新优化生产组织要素的价值链关系，使得整个产业链中的各个环节围绕最终用户的价值交付，以高效的协同方式组织生产活动。

明白这个目标对于我们去认知现在层出不穷的技术概念有很大的帮助，人工智能、大数据、物联网、工业互联网、云制造等新技术的诞生，本质都是为了提升知识生产力要素和价值交付的效率。

然而人工智能在工业领域的落地必然会充满着挑战，正因为其门槛较高，才使工业成为人工智能迟迟难以大规模进入的行业。这些挑战首先来源于对能力的要求，工业人工智能需要“工业领域知识”、“智能算法”和“软件平台技术”三方面的综合能力，而且其能够实现的价值瓶颈取决于最薄弱的那一部分。像 GE 和三一这样的老牌工业企业，虽然拥有丰富的领域知识，但是缺乏吸引和使用能够开发智能算法和软件平台人才的环境。而对阿里和腾讯这样的互联网巨头而言，工业场景也超出其理解的范围。

另一方面，我们不能用互联网所认知的人工智能来理解工业智能。在算法层面，首先需要消除算法中的 Surprise（意外），神经网络看似无所不能，但是如果它们甚至不能自我解释，我们真的可以依靠它们吗？尽管在神经网络的可解释性研究已经取得了一些成果，例如从深度神经网络提取基于树的规则（extraction of tree-based rules from deep networks）、卷积层可视化、特征贡献度、InterpretNet 等，但是机器学习的预测从不确定性到相对可控的确定性仍然还有很长的一段路要走。

人工智能真正开始被规模化地应用于工业系统中，进而实现工业智能系统，至少需要做到以下 4 个 S：

- Standard(标准化)：即如何与现有工业系统的标准化体系相结合，包括方法论、建模过程、数据质量、模型评价、容错机制、基于预测的操作规程、不确定性管理等各方面的标准化。
- Systematic(体系化)：在技术层级和应用层级方面的体系化，需要建立一套protocol 体系，明确工业智能在部件级、设备级、系统级和社区级等不同层级中的任务边界及相互的接口。

- Sustainable(稳定可持续)：与人工智能预测的可解释性和结果的确定性相似，如何能够做到同一组数据和同一个模型，不同的人来训练得到的结果都要是一样的，否则怎么做到制造系统的标准化和一致性管理呢?
- Streamline (连续流)：实现连续流一直是制造系统的追求的理想状态，而连续流的瓶颈受制于效率最低的一个环节。这决定了人工智能在工业系统中单点的提升很难换来最终产出的巨大飞跃，需要以工业系统的价值流为视角去优化协同效率和提升单点瓶颈。

要解决工业智能落地的挑战不是任何一个企业能够完成的，需要我们各自贡献自己的力量共同建立一个更大的生态。我们欣喜地看到在过去几个月中关于工业智能的联盟和组织正在悄然兴起，工业大数据和工业互联网联盟峰会上也第一次设置了工业人工智能论坛。希望有越来越多的企业、组织和个人加入到工业智能领域，共同为工业企业提供更好的 AI 产品和服务，也为中国工业的价值转型探索一条光明之路。

7折优惠

联席主席

方国伟
平安科技
CTO兼总架构师

褚霸（余锋）
阿里巴巴
研究员

出品人与分享嘉宾

丁宇（叔同）
阿里巴巴
2017天猫
双11技术大队长

金晓军
阿里巴巴
高级专家

肖世广
腾讯
QQ技术运营总监

陈功
腾讯
微信广告引擎负责人

张贺
南京大学
软件学院教授

Yunong Xiao
Netflix
Principal Software Engineer

杨钦民
贝聊
研发总监

孟晓桥
Pinterest
监控组经理

Bin Xu
Facebook
Software Engineer Manager

大沙
阿里巴巴
计算平台事业部
高级技术专家

段亦韬
网易有道
首席科学家

扫码
快速了解
大会嘉宾

目前7折报名中，欢迎以目前最优惠价格报名ArchSummit

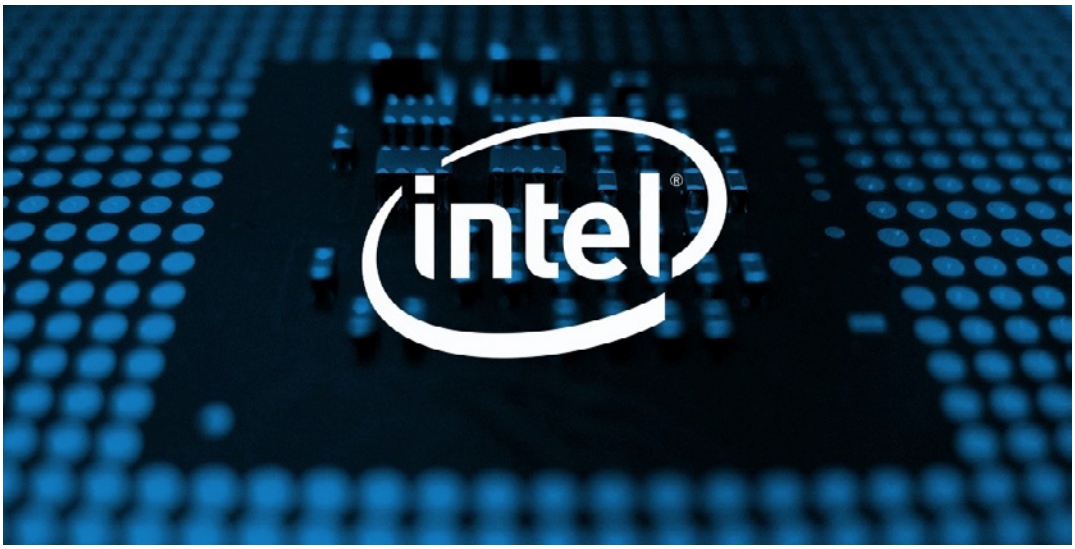
如果在报名过程中遇到任何问题，可联系大会主办方，欢迎咨询！

微信：aschina666　电话：010-84780850

ArchSummit全球架构师峰会深圳站 | 大会演讲：7月6-7日，深度培训：7月8-9日

英特尔、谷歌和 AWS 回应 CPU 安全事件：AMD 和 ARM 也有问题，影响巨大

作者 陈思



英特尔方面解释称，其已经向 ARM、AMD 以及其它多家操作系统供应商报告了此项漏洞。该方同时强调，该漏洞最初是由谷歌公司的 Zero 项目安全小组所发现，这一说法也得到了谷歌方面的证实。

英特尔公司指出，其将发布微代码更新以解决此项问题，而随时间推移这些修复补丁中的一部分亦将被引入硬件当中。

据英特尔所说，本来他们是计划下周当软件和固件都比较完善的时候向用户披露这个信息的，但是由于媒体的报道不准确，引起舆论哗然才让其不得不提前发布声明。在回应时，Intel 表示，这些漏洞不存在被攻击、修改或删除数据的可能，还表示 媒体报道此“漏洞”或“缺陷”是英特尔的产

品独有的问题这一说法是不准确的。基于目前为止的分析，许多类型的计算设备（不同供应商的处理器和操作系统）也很容易受到这些攻击。

微软公司拒绝就此事发表评论，不过预计微软也将推出自己的对应补丁。此外，AWS、谷歌公司也发布了自己的报告（见下文）。

意味着什么：就目前来讲，我们可以看到各大主要芯片与操作系统供应商已经意识到问题的存在，并努力发布修复程序。第一款补丁很可能被纳入微软的新一轮补丁周二发布。目前还不清楚这批补丁会影响到多少不同类型的软件与 CPU 架构，也不明确具体将给 PC 性能造成怎样的影响。

但是，补丁带来的影响会不会比安全问题本身更令人难以接受？根据初步报道，取决于具体任务与处理器型号的不同，PC 用户可能因此遭遇5% 到 30% 之间的性能衰减”。

英特尔方面似乎 坚信普通用户不会受到任何负面影响。其在一份声明中指出：“与一部分报道相反，任何潜在的性能影响都将取决于实际工作负载。而对普通计算机用户而言，其不会产生任何显著的影响，且性能衰减将随时间推移而得到缓解。”

英特尔公司进一步作出解释，即“具体取决于实际工作负载”，但客户仍然无法借此对影响作出准确的判断。

高管们表示，英特尔公司在研究这项漏洞的影响时，并没有将客户 PC 与数据中心服务器区分开来。事实上，普通用户所使用的应用程序可能遭受 0% 到 2% 的性能影响，但调整依赖应用程序与操作系统间交互的综合性工作负载则可能遭遇最高 30% 的性能衰减。

我们可以做点什么？

英特尔公司给出的建议是补丁、补丁、还是补丁。相信大家已经对这样的陈词滥调非常熟悉：“请咨询您的操作系统供应商或系统制造商，并尽快应用所有可用更新。遵循良好的安全措施以防范恶意软件，这将有助于预防可能的恶意利用活动直到更新工作全面完成。”

机器学习会受到影响吗？

据部分专家分析，这取决于集成模型，对模型每个部分的调用都可能产生影响。此外，生成一个新程序将产生新的开销，内存释放 / 分配算法将更复杂，而 CPU 处理器将成为深度学习 GPU 图像加载的瓶颈。

谷歌回应：Chrome 浏览器将受影响，请按操作更新

谷歌公司则报告称，其 Chrome 浏览器将受到影响，但用户可以采取两步走解决战略：首先，确保您的浏览器更新至版本 63; 其次，可以启用一项名为站点隔离的可选功能，从而将各站点隔离在独立的地址空间内。此可选项目可在 chrome://flags/#enable-site-per-process 当中开启。最后，计划于今年 1 月 23 日发布的 Chrome 64 将能够保护用户免受旁路攻击的侵扰。

受影响设备完整清单如下。

以下未明确列出的所有 Google 产品都不需要用户进行操作。

Android

- 最新安全更新的设备受到保护。此外，目前我们还未发现此漏洞成功再复制，即未经 ARM Android 设备的授权泄露信息。
- 最新安全更新的 Nexus 和 Pixel 设备受到保护。

Google Apps / G Suite（Gmail，日历，云端硬盘，网站等）：

- 无需额外的用户或客户操作。

谷歌浏览器：

- 一些用户或客户需要采取措施。

Google Chrome 操作系统（例如 Chromebook）：

- 需要一些额外的用户或客户操作。
- Google 云端平台
- Google App 引擎：无需额外操作。
- Google 计算引擎：需要一些额外操作。

- Google Kubernetes 引擎：需要一些额外操作。
- Google Cloud Dataflow：需要一些额外操作。
- Google Cloud Dataproc：需要一些额外操作。
- 所有其他 Google 云产品和服务：无需其他操作。
- Google 主页 / Chromecast：无需额外操作。
- Google Wifi / OnHub：无需额外的用户操作。

具体操作参考。

亚马逊回应：此 bug 已存在 20 余年，发布新版 Linux 系统

昨日，亚马逊也作出回应，称这个 bug 其实是 Intel、AMD 等现代处理器存在了 20 多年的问题，目前亚马逊 MC2 fleet 只有一小部分产品是可以抵御攻击的，在未来的几小时之内将会被修复。

Amazon Linux 存储库中提供了更新的 Amazon Linux 内核。2018 年 1 月 3 日下午 10:45（GMT）之后使用默认的 Amazon Linux 配置启动将自动包含更新的软件包。Amazon Linux AMI 用户可以运行以下命令，确保收到更新的软件包。

旁路分析利用是什么？

根据英特尔方面的说法，攻击者可以通过观察高权限内存中的内容利用名为“推测性执行”的 CPU 技术规避必要的权限级别约束。如此一来，攻击者将能够访问其正常情况下无法访问的数据。不过英特尔公司表示其无法对数据内容进行删除或者修改。

事实上，英特尔与研究人员们发现了 三种漏洞利用变种，分别为“边界检查旁路”、“分支目标注入”以及“流氓数据加载”，三者在具体攻击手段方面略有不同。但通过操作系统更新，三种问题都能够得到有效缓解。

英特尔公司工程技术负责人 Steve Smith 报告了这一重要发现，并补充称目前还没有观察到对该项漏洞的任何实际利用行为。他同时否认了此项漏洞属于缺陷的说法，亦不承认其出现是为了专门针对英特尔。在本次

电话会议上，Smith 向投资者们强调“处理器的运作方式仍然严格遵循我们的设计预期。”

Smith 同时表示，这一发现敦促全球各硬件制造商以“负责任的方式”应对此项漏洞。

这是一个影响到全行业的问题

前文提到的各家企业已经计划在未来一周内陆续发布相关补丁。而英特尔方面则表示，其提前作出评论的原因在于“目前存在诸多不够准确的媒体报道”——但需要强调的是，芯片巨头的声明当中并没有对这些报道作出任何否定。英特尔只是首先向媒体发布了一份声明，而后即召开电话会议。

英特尔公司指出，“英特尔与其它技术企业已经意识到用于描述软件分析方法的新型安全研究成果一旦出于恶意目的而遭利用，可能导致从计算设备上收集敏感数据。”

英特尔公司认为，此项漏洞与其它类型架构同样存在密切关联——这里所指的肯定是 AMD（尽管其否认自家芯片会因此受到影响）以及作为大多数智能手机核心的 ARM。除此之外，另有几家厂商以及操作系统供应商的产品“易受影响”。

微软公司的一位发言人在接受邮件采访时解释称，“我们已经意识到这一影响整个行业的问题，并一直与芯片制造商开展密切合作，旨在开发并测试缓解措施以保护我们的客户。我们正在为云服务部署缓解措施，同时亦发布了对应安全更新，希望保护 Windows 客户免受英特尔、ARM 以及 AMD 支持型硬件芯片内相关漏洞的影响。”

AMD 方面否认其处理器产品受到影响，并在采访中强调称目前 AMD 处理器所面临的风险“几乎为零”。

英特尔则试图解释其为何没有主动站出来曝光问题——芯片巨头宣称，就在此次消息传出之时，其几乎已经完成了漏洞的内部解决。英特尔公司指出，“我们致力于通过负责任的方式披露安全问题，也正因为如此，

英特尔与其它供应商才计划在下周公布相关软件与固件更新时正式披露这一问题。”

英特尔回应存疑，网友不买账

虽然官方做出了回应，但是网友们似乎并不买账，大多数网友对这一回应都持怀疑的态度，认为英特尔不够诚实，这样的做法是自欺欺人。知名社交网站 Reddit 上有关英特尔的讨论同样在如火如荼进行着，除了疯狂 diss 英特尔之外，也有网友给出了自己的分析。

我们摘录了部分网友的讨论内容

网友观点一：对于有独立数据中心的企业来说可能没太大影响，因为他们不会受到这个漏洞带来的安全问题影响，因此肯定会在没有内核补丁的情况下继续运行他们的数据中心。但对于学术研究人员来说，这可能（会？）确实是一个更大的问题，因为他们通常不管理自己的内核，而是云上运行计算任务。不过 CPU 一般很少成为性能瓶颈，现在大多数 BLAS 密集型计算是在 GPU 上完成的，数据集通常被完全加载到 RAM 中，而在现代库中不会经常产生新的线程。

网友观点二：这几乎肯定会导致产品召回，并最终从 Linux 内核中删除该内核补丁。

网友观点三：几乎所有在过去 20 年中生产的英特尔处理器都受到了影响，许多目前还有人在使用的处理器很可能早已停产，并且没办法快速生产出可以兼容的主板。英特尔可能会召回最近的一款处理器，但是全面召回受影响的处理器将需要数年时间，甚至可能导致公司破产。所以用户只能继续忍受这个内核补丁，直到购买一个新的处理器，由于这个问题补丁导致的性能下降，可能得比计划更早去购买新处理器。也许英特尔会给因为这个原因购买新芯片的客户适当的折扣，以恢复客户对他们的信任。

网友观点四：或许是时候该买点 AMD 的股票了。

网友观点五：或者卖掉 Intel 的股票。（英特尔的 CEO 在 12 月中下

旬卖掉了大量英特尔的股票）

网友观点六：他不仅卖掉了一堆股票，而且还卖掉了所有可以卖出的股票（附例说 CEO 必须拥有 25 万股股票——于是他把 25 万股以外的股票全卖掉了 ……）

网友观点七：这可能就是为什么微软发布了一个通知，说 Azure 中的一些虚拟机必须先重启，否则将会在 1 月 10 日自动重启。 当然，这可能只是标准的维护，因为它不像他们发布大量的信息。补充：这下有趣了，微软刚刚发出了警告，强制要求所有还没重新部署的人立即重新部署。看来是因为 Project Zero 刚刚发布的消息。

网友观点八：我不是这方面的专家，但是这听起来像是跳过了数据结构的边界检查——检查肯定会花费更多时间和资源，如果你确信检查是不必要的，那么直接省略这个步骤可以节省时钟周期。

这听起来像是和推测执行有关。如果你正在推测性地执行一条指令，那么你最终可能会舍弃它的结果，所以执行这个指令花费的代价越少越好。也许英特尔认为他们可以在推测执行的时候跳过 priv 检查，如果最终发现确实需要这条指令，就在实际执行结果之前执行指令。 然而，也许事实证明，这种推测性执行打开了一些通过后门获取数据的方法，比如通过缓存、定时等等，如果异常提前被发现，这个问题就不会暴露了。

网友观点九：这个修复补丁对于性能到底有多大影响会根据具体的计算任务而有所不同，但不可能全部是 34 %。这个补丁确实会损害内核的上下文切换性能。如果你的应用程序一直在做大量的系统调用，这对你的性能影响可能会非常大。然而，直到我们真的看到实际的案例，才能真正了解影响到底有多大。

网友观点十：这里有两篇文章，他们在集成并启用了该修复补丁的 Windows 10 Insider 上进行了测试，测试结果显示，普通用户其实无需担心。在合成、实际工作负载以及游戏中的 CPU 性能基本未受影响，基准测试中的差异都还在误差范围内。可能唯一有影响的是使用了 NVMe 驱动的用户，比如 960 Pro 这种速度非常快的驱动器，性能损失约 5 %，对大多

数人来说可以忽略不计。而像基于 NVME 和 SATA 的 SSD，性能损失几乎为 0%，因为这些驱动器的速度没有快到会被影响到。

- 7700K + 1080 Ti：
- 3930K + 1080 Ti：

网友观点十一：欸等等，AMD 也被报告“不安全”？ 哦，哦，哦，是修复补丁把 AMD 视为不安全，而不是 AMD 实际上不安全。补充：Bug 不会影响 AMD。 但显然，他们正在向所有架构都推送补丁，而不只是英特尔，这又反过来影响了 AMD 的性能。也许在经过全面测试之后，AMD 会回滚到打补丁之前的版本。补充吐槽：简直是天才，当你出了问题，就强制让别人也出问题。

网友观点十二：免责声明：我是一名英特尔的工程师，但是我的发言仅代表我个人意见，而且我没有直接参与这项调查。

我已经阅读了谷歌的 Meltdown 论文，可以分享一些想法。我也可以回答大家的问题，但显然我不会分享任何英特尔内部消息。

攻击主要利用了推测性执行（speculative execution），并依赖于缓存方攻击。其实并不是一个真正意义上的“bug”，因为推测性执行就是按照设计工作的，也就是说它不会影响寄存器或内存（除非已提交），但可能会影响缓存。因此，这种攻击会导致 CPU 首先进入某种故障或陷阱，然后按照程序的顺序执行一些访问特权内存的指令。这些指令永远不会退出，因为陷阱会导致他们中断，但可能会推测性地执行，这就可能影响缓存。然后这个攻击会使用一个旁道攻击（side channel attack）来获取缓存的内容，于是就可能在里面发现特权数据。

基本上任何支持推测性执行的硬件都可能被攻击。在操作系统层面可以解决大部分问题，当然从硬件层面上也有很多修复的方法。但是目前这个缺陷的严重程度，与乱序引擎的深度和其他结构的大小基本上是直接相关的。非常糟糕的是，这基本上意味着一个更快的内核就会更脆弱，更容易受到影响。谷歌的这篇文章称，他们发现所有的英特尔、AMD 和 ARM 都存在问题，但他们无法对 AMD 或者 ARM 进行攻击，这可能是因

为 AMD 和 ARM 没有那么复杂或者乱序引擎的深度更小（虽然这看起来似乎 ARM 存在另一个相关的问题）。所以英特尔暴露的问题最为严重，因为英特尔的乱序引擎有更高的性能。

Spectre 攻击要少见得多，但对所有 CPU 供应商的影响是一样的。风险较小是因为攻击必须针对一个特定的过程，但是相应的修复起来也很困难。

开发者需要知道的有关软件架构的五件事

作者 Simon Brown 译者 薛命灯

2010 年，我写了一篇叫作“Are You a Software Architect?”的文章，探讨了软件开发者与软件架构师之间的区别，以及如何从一名软件开发者转成一名架构师。8 年过去了，软件行业也在发展，但开发团队仍然面临着类似的问题，特别是与软件架构有关的问题。这些问题比以往任何时候都要来得突出，因为我们现在构建的系统越来越趋于分布式化，开发团队也越来越分布式化。为了解开这些迷思，开发者需要了解以下五个与软件架构有关的事实。

1. 软件架构不只是前期的“大设计”

传统的观点认为，软件架构就是在前期进行“大设计”，然后通过瀑布

模型进行交付，架构团队要确保软件的每一个元素在进行编码之前都要考虑妥当。2001 年，“敏捷开发宣言”建议我们“拥抱变化而不是遵循计划”，但这个观点后来却被误读成不应该制定任何计划。结果就是，有些开发团队直接从原先的“大设计”变成了零设计。这两种极端的行为都愚蠢至极，实际上，在某个时候，你会发现前期的设计并非开发出完美软件的必要因素。前期的设计应该只是一个起点，或是作为团队前进方向的指引。

在进行软件设计时需要做出一些设计决策。在谈及软件架构和软件设计之间的区别这个问题时，Grady Booch 说，“架构代表了重要的决策，决策的重要程度通过变更成本来衡量”。换言之，就是看在后续进行变更时那个决策需要付出更大的成本。所以，好的前期设计就是要充分理解什么是“重要的决策”。这些决策通常与技术选型和结构（也就是分解策略、模块化、功能边界等）有关。如果开发的是一个单体系统，那么选择何种编程语言可能就变得尤为重要。如果采用的是微服务架构，那么使用何种编程语言就变得不那么重要，但需要考虑其他方面的因素。类似的，如果采用了六边形架构，虽然可以将业务逻辑与技术选型解耦开来，但仍然需要做出其他方面的权衡。

所以说，前期设计就是要了解影响软件成型的重要决策，而不是具体的技术细节，比如数据库的某个列要设置多大的长度。在现实当中，我会让团队真正去了解他们将要做什么、如何去做以及他们已经设计好的东西是否可行。可以让他们识别出最高优先级的任务，如果有必要可以写出代码。总而言之，前期设计就是一个叠加成功几率的过程。

2. 每个开发团队都需要进行软件架构

上述的内容适用于每一个开发团队，从一个单人团队到数百人的分布式团队。设置起点和方向其实就是要建立技术领导力。如果做不到这一点，就可能出现混乱：结构混乱的代码库（典型的“大泥球”），难以理解，难以维护，质量不达标，如性能、伸缩性或安全性。简而言之，任何一个开发团队都需要技术领导力。

3. 软件架构师要会写代码、指导他人以及参与协作

在大部分人看来，软件架构师就是给开发团队下达指令的人，就像接力赛中跑第一棒的人。但事实不是这样的，现代的架构师喜欢编码、指导他人并参与协作。我所遇见的好架构师也都是好的开发者，他们仍然喜欢编码，最起码他们并不想放弃编码工作。因为变化太快，人们很容易就与技术失之交臂。但我认为软件架构师应该是“建筑大师”，在必要的时候他们仍然可以写代码。作为团队的一份子，编码会让架构师的工作变得更容易一些，因为编码有助于架构师理解系统，而且团队的其他成员会真正把架构师当成是同事。

需要注意的是，软件架构师不一定要指定某个人来担任。刚开始可以这样做，但其实也可以由多个人共同承当这个角色。 但要注意，建立协作式的技术领导力并不是件轻而易举的事。软技能本来就不是很轻松就能获得的。我曾经组建 2 到 5 个人的架构师小组，让他们来设计软件系统，但他们在与技术和设计决策方面无法达成共识。在极端情况下，个性冲突会导致小组解散。关键是要了解你的团队，并确保应用了恰到好处的技术领导力。

4. 使用 UML 不是必需的

传统的软件架构通常包含大量的 UML 模型图，试图充分展现软件系统的每一个细节。可惜的是，建模和 UML 在很大程度上与前期“大设计”相耦合，而这些技术在近几年已经过时了。现在完全不懂 UML 的软件开发团队比率在逐步上升。

现如今，大部分人只是“在白板上画几个方块和线条”，并将其作为沟通想法的手段。在过去几年，我参与过的软件架构小组画过大量这样的图表，我把它们拍成照片，有好几个 G。我敢说，从行业角度来讲，我们已经散失了软件架构的沟通能力。我见过不计其数的图表，它们有些是随机着色方块和线条的组合，模糊不清，难以辨认。如果一个团队无法就软件

架构进行沟通，那么就无法设置起点和建立技术领导力。

我建议使用C4模型来进行软件架构方面的沟通——上下文（Context）、容器（Containers）、组件（Components）和代码（Code）。其本质是创建一系列结构化、可伸缩的图表来描述软件系统。为每一个软件系统创建一个上下文图表，用于描述软件系统与真实世界之间的关系。然后放大系统边界，让内部的容器突显出来——容器就是可部署、可运行的实体，比如运行在浏览器上的单页应用、服务器端的 Web 应用、微服务、数据库实例，等等。如果有必要，可以再将每个容器放大，让容器内部的组件突显出来。最后，你也可以放大组件，让代码级别的元素（类、接口、函数、对象等）也突显出来。C4 模型独立于具体的表示方法，虽然我倾向于使用简单的“方块和线条”，但使用 UML 来表示也是可以的。

c4model.com 提供了更多的信息、视频、示例图表和工具链接。如果你的团队正纠结于软件架构沟通方面的问题，那么可以看看这些资料。

5. 好的软件架构是敏捷的

现在仍然存在一种误解，认为“架构”和“敏捷”之间是一种竞争和冲突的关系。但其实不是的。相反，好的软件架构也是敏捷的，它有助于应对业务变更，不管是需求变更、业务流程变更还是混合变更。当然，什么才是“好的架构”仍然有待商榷，但我认为，好的架构与好的模块化息息相关。如果你曾经有过在“大泥球”上进行变更的痛苦经历，那么你就会知道，好的代码库结构（好的模块化）是多么的重要。

现如今，很多团队都存在一个很大的问题，他们采用了一种叫作“架构中立的设计”，George Fairbanks 在他的“Just Enough Software Architecture”一书中对此进行了描述。换句话说，他们采用了一种不需要考虑权衡因素的架构风格。在现实中，开发团队使用微服务架构代替单体代码库。但实际上，他们有可能是创建出了一种“分布式的大泥球”，可见软件设计和分解过程是多么的重要，不管是要构建一个单体还是一个微服务架构的系统。敏捷和好的软件架构不是那么容易就能实现的，它需要一

些精巧的设计，也需要作出一些权衡。这也再次说明为什么设置起点和建立技术领导力是如此的重要。

作者简介

Simon Brown 是一个独立的软件架构顾问，同时是“Software Architecture for Developers”一书的作者。他还是C4软件架构模型的创立者。Simon 是国际软件开发大会的演讲常客，并周游世界，帮助企业促进软件架构方面的工作。

阿里盒马领域驱动设计实践

作者 张群辉

前言

设计是把双刃剑，没有最好的，也没有更好的，而是条条大路到杭州。同时不设计和过度设计都是有问题的，恰到好处的设计才是我们追求的极致。

DDD（Domain-Driven Design，领域驱动设计）只是一个流派，谈不上压倒性优势，更不是完美无缺。 我更想跟大家分享的是我们是否关注设计本身，不管什么流派的设计，有设计就是好的。

从我看到的代码上来讲，阿里集团内部大部分代码都不属于 DDD 类

型，有设计的也不多，更多的像“面条代码”，从端上一条线杀到数据库完成一个操作，仅有的一些设计集中在数据库上。我们依靠强大的测试保证了软件的外部质量（向苦逼的测试们致敬），而内部质量在紧张的项目周期中屡屡得不到重视，陷入日复一日的技术负债中。

一直想写点什么唤起大家的设计意识，但不知道写点什么合适。去年转到盒马，有了更多的机会写代码，可以从无到有去构建一个系统。盒马跟集团大多数业务不同，盒马的业务更面向 B 端，从供应到配送链条，整体性很强，关系复杂，不整理清楚，谁也搞不明白发生什么了。所以这里设计很重要，不设计的代码今天不死也是拖到明天去死，不管我们在盒马待多久，不能给未来的兄弟挖坑啊。在我负责的模块里，我们完整地应用了 DDD 的方式去完成整个系统，其中有我们自己的思考和改变，在这里我想给大家分享一下，他山之石可以攻玉，大家可以借鉴。

领域模型探讨

1. 领域模型设计：基于数据库 vs 基于对象

设计上我们通常从两种维度入手：

- Data Modeling: 通过数据抽象系统关系，也就是数据库设计
- Object Modeling: 通过面向对象方式抽象系统关系，也就是面向对象设计大部分架构师都是从 Data Modeling 开始设计软件系统，少部分人通过 Object Modeling 方式开始设计软件系统。这两种建模方式并不互相冲突，都很重要，但从哪个方向开始设计，对系统最终形态有很大的区别。

Data Model

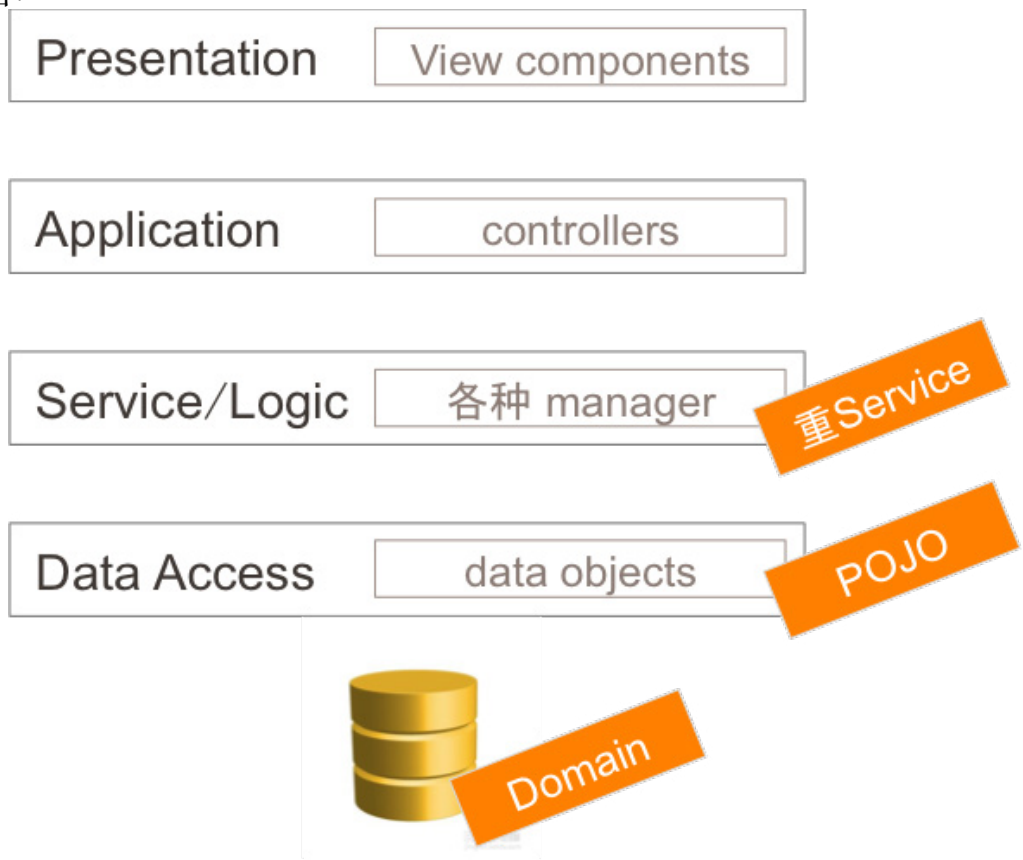
领域模型（在这里叫数据模型）对所有软件从业者来讲都不是一个陌生的名词，一个软件产品的内在质量好坏可能被领域模型清晰与否所决定，好的领域模型可以让产品结构清楚、修改更方便、演进成本更低。

在一个开发团队里，架构师很重要，他决定了软件结构，这个结构决

定了软件未来的可读性、可扩展性和可演进性。通常来说架构师设计领域模型，开发人员基于这个领域模型进行开发。“领域模型”是个潮流名词，如果拉回到 10 几年前，这个模型我们叫“数据字典”，说白了，领域模型就是数据库设计。

架构师们在需求讨论的过程中不停地演进更新这个数据字典，有些设计师会把这些字典写成 SQL 语句，这些语句形成了产品 / 项目数据库的发育史，就像人类胚胎发育：一个细胞（一个表），多个细胞（多个表），长出尾巴（设计有问题），又把尾巴缩掉（更新设计），最后哇哇落地（上线）。

传统项目中，架构师交给开发的一般是一本厚厚的概要设计文档，里面除了密密麻麻的文字就是分好了域的数据库表设计。言下之意：数据库设计是根本，一切开发围绕着这本数据字典展开，形成类似于下边的架构图：



在 service 层通过我们非常喜欢的 manager 去 manage 大部分的逻辑，POJO（后文失血模型会讲到）作为数据在 manager 手（上帝之手）里不停地变换和组合，service 层在这里是一个巨大的加工工厂（很重的一层），围绕着数据库这份 DNA，完成业务逻辑。

举个不恰当的例子：假如有父亲和儿子这两个表，生成的 POJO 应该是：

```
public class Father{…}
public class Son{
     private String fatherId;//son 表里有 fatherId 作为 Father 表 id
外键
    public String getFatherId(){
        return fatherId;
    }
    ……
}
```

这时候儿子犯了点什么错，老爸非常不爽地扇了儿子一个耳光，老爸手疼，儿子脸疼。Manager 通常这么做：

```
public class SomeManager{
    public void fatherSlapSon(Father father, Son son){
        // 如果逻辑上说不通，大家忍忍
        father.setPainOnHand();
         son.setPainOnFace();// 假设 painOnHand, painOnFace 都是数据库
字段
    }
}
```

这里，manager 充当了上帝的角色，扇个耳光都得他老人家帮忙。

Object Model

2004 年，Eric Evans 发表了《Domain-Driven Design – Tackling Complexity in the Heart of Software》（领域驱动设计），简称 Evans DDD，先在这里给大家推荐这本书，书里对领域驱动做了开创性的理论阐述。

在聊到 DDD 的时候，我经常会做一个假设：假设你的机器内存无限大，永远不宕机，在这个前提下，我们是不需要持久化数据的，也就是我们可以不需要数据库，那么你将会怎么设计你的软件？这就是我们说的 Persistence Ignorance：持久化无关设计。

没了数据库，领域模型就要基于程序本身来设计了，热爱设计模式的同学们可以在这里大显身手。在面向过程、面向函数、面向对象的编程语言中，面向对象无疑是领域建模最佳方式。

类与表有点像，但不少人认为表和类就是对应的，行 row 和对象 object 就是对应的，我个人强烈不认同这种等同关系，这种认知直接导致了软件设计变得没有意义。

类和表有以下几个显著区别，这些区别对领域建模的表达丰富度有显著的差别，有了封装、继承和多态，我们对领域模型的表达要生动得多，对 SOLID 原则的遵守也会严谨很多：

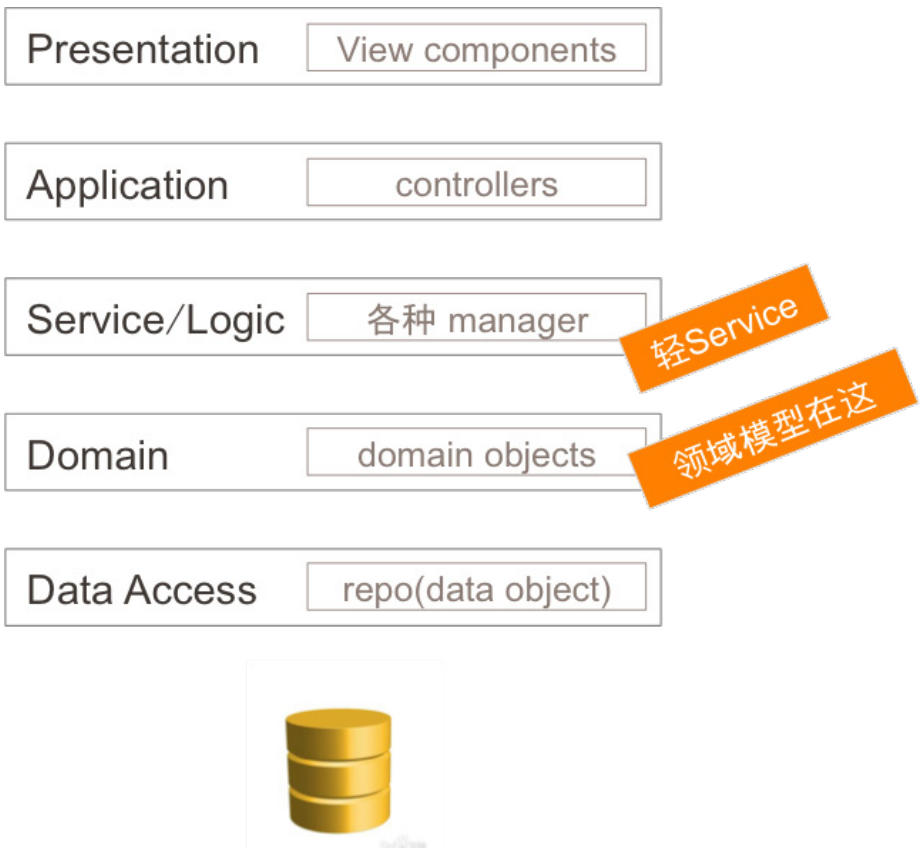
- 引用：关系数据库表表示多对多的关系是用第三张表来实现，这个领域模型表示不具象化， 业务同学看不懂。
- 封装：类可以设计方法，数据并不能完整地表达领域模型，数据表可以知道一个人的三维，但并不知道“一个人是可以跑的”。
- 继承、多态：类可以多态，数据上无法识别人与猪除了三维数据还有行为的区别，数据表不知道“一个人跑起来和一头猪跑起来是不一样的”。

再看看老子生气扇儿子的例子：

```
public class Father{
    // 教训儿子是自己的事情，并不需要别人帮忙，上帝也不行
    public void slapSon(Son son){
        this.setPainOnHand();
        son.setPainOnFace();
    }
}
```

根据这个思路，慢慢地，我们在面向对象的世界里设计了栩栩如生的

领域模型，service 层就是基于这些模型做的业务操作（它变薄了，很多动作交给了 domain objects 去处理）：领域模型并不完成业务，每个 domain object 都是完成属于自己应有的行为（single responsibility），就如同人跑这个动作，person.run 是一个与业务无关的行为，但这个时候 manager 或者 service 在调用 some person.run 的时候可以完成 100 米比赛这个业务，也可以完成跑去送外卖这个业务。这样的话形成了类似于下边的架构图：

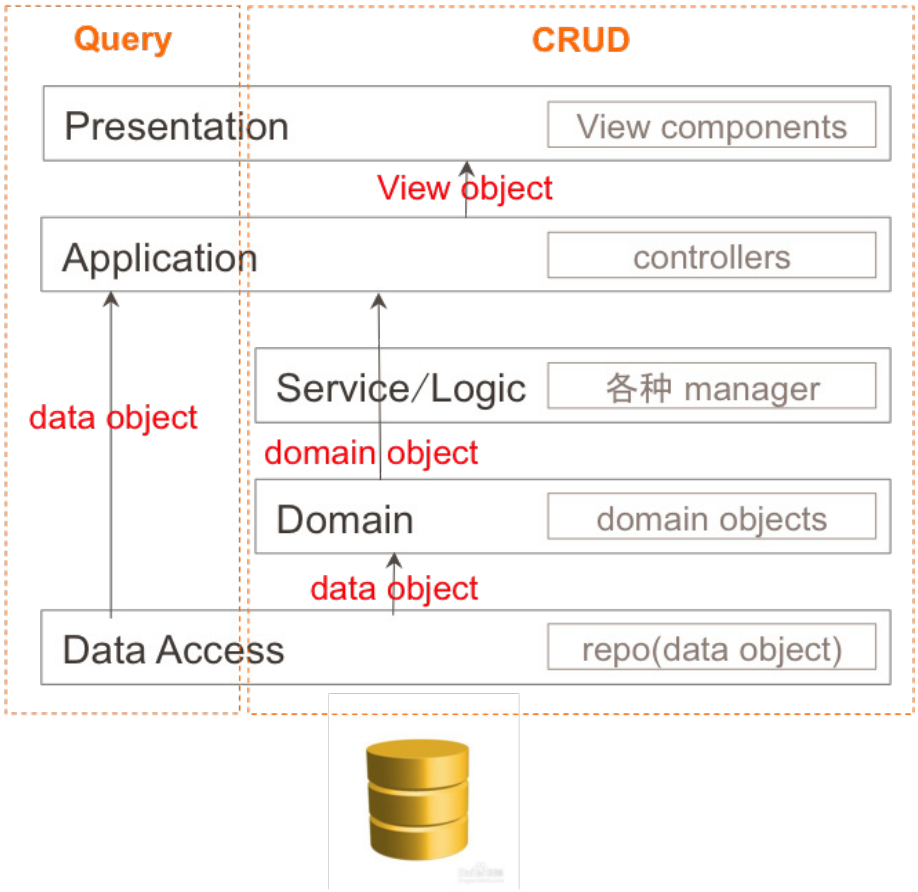


我们回到刚才的假设，现在把假设去掉，没有谁的机器是内存无限大，永远不宕机的，那么我们需要数据库，但数据库的职责不再承载领域模型这个沉重的包袱了，数据库回归 persistence 的本质，完成以下两个事情：

- 存：将对象数据持久化到存储介质中。
- 取：高效地把数据查询返回到内存中。

由于不再承载领域建模这个特性，数据库的设计可以变得天马行空，任何可以加速存储和搜索的手段都可以用上，我们可以用 column 数据库，

可以用 document 数据库，可以设计非常精巧的中间表去完成大数据的查询。总之数据库设计要做的事情就是尽可能高效存取，而不是完美表达领域模型（此言论有点反动，大家看看就好），这样我们再看看架构图：



这里我想跟大家强调的是：

- 领域模型是用于领域操作的，当然也可以用于查询（read），不过这个查询是有代价的。在这个前提下，一个 aggregate 可能内含了若干数据，这些数据除了类似于 getById 这种方式，不适用多样化查询（query），领域驱动设计也不是为多样化查询设计的。
- 查询是基于数据库的，所有的复杂变态查询其实都应该绕过 Domain 层，直接与数据库打交道。
- 再精简一下：领域操作 ->objects，数据查询 ->table rows

2. 领域模型：失血、贫血、充血

失血、贫血、充血和胀血模型应该是老马提出的（此老马非马老师，是 Martin Fowler），讲述的是基于领域模型的丰满程度下如何定义一个模型，有点像：瘦、中等、健壮和胖。胀血（胖）模型太胖，在这里我们不做讨论。

失血模型：基于数据库的领域设计方式其实就是典型的失血模型，以 Java 为例，POJO 只有简单的基于 field 的 setter、getter 方法，POJO 之间的关系隐藏在对象的某些 ID 里，由外面的 manager 解释，比如 son.fatherId，Son 并不知道他跟 Father 有关系，但 manager 会通过 son.fatherId 得到一个 Father。

贫血模型：儿子不知道自己的父亲是谁是不对的，不能每次都通过中间机构（Manager）验 DNA(son.fatherId) 来找爸爸，领域模型可以更丰富一点，给 son 这个类修改一下：

```
public class Son{
    private Father father;
    public Father getFather(){return this.father;}
}
```

Son 这个类变得丰富起来了，但还有一个小小的不方便，就是通过 Father 无法获得 Son，爸爸怎么可以不知道儿子是谁？这样我们再给 Father 添加这个属性：

```
public class Father{
    private Son son;
    private Son getSon(){return this.son;}
}
```

现在看着两个类就丰满多了，这也就是我们要说的贫血模型，在这个模型下家庭还算完美，父子相认。然而仔细研究这两个类我们会发现一点问题：通常一个 object 是通过一个 repository（数据库查询），或者 factory（内存新建）得到的：

```
Son someSon = sonRepo.getById(12345);
```

这个方法可以将一个 son object 从数据库里取出来，为了构建完整的

son 对象，sonRepo 里需要一个 fatherRepo 来构建一个 father 去赋值 son.father。而 fatherRepo 在构建一个完整 father 的时候又需要 sonRepo 去构建一个 son 来赋值 father.son。这形成了一个无向有环圈，这个循环调用问题是可以解决的，但为了解决这个问题，领域模型会变得有些恶心和将就。有向无环才是我们的设计目标，为了防止这个循环调用，我们是否可以在 Father 和 Son 这两个类里省略掉一个引用？修改一下 Father 这个类：

```
public class Father{
    //private Son son; 删除这个引用
    private SonRepository sonRepo;// 添加一个 Son 的 repo
    private getSon(){return sonRepo.getByFatherId(this.id);}
}
```

这样在构造 Father 的时候就不会再构造一个 Son 了，但代价是我们在 Father 这个类里引入了一个 SonRepository，也就是我们在一个 domain 对象里引用了一个持久化操作，这就是我们说的充血模型。

充血模型：充血模型的存在让 domain object 失去了血统的纯正性，他不再是一个纯的内存对象，这个对象里埋藏了一个对数据库的操作，这对测试是不友好的，我们不应该在做快速单元测试的时候连接数据库，这个问题我们稍后来讲。为保证模型的完整性，充血模型在有些情况下是必然存在的，比如在一个盒马门店里可以售卖好几千个商品，每个商品有好几百个属性。如果我在构建一个店的时候把所有商品都拿出来，这个效率就太差了：

```
public class Shop{
    //private List<Product> products; 这个商品列表在构建时太大了
    private ProductRepository productRepo;
    public List<Product> getProducts(){
        //return this.products;
        return productRepo.getShopProducts(this.id);
    }
}
```

3. 领域模型：依赖注入

简单说一说依赖注入：

- 依赖注入在 runtime 是一个 singleton 对象，只有在 spring 扫描范围内的对象（@Component）才能通过 annotation（@Autowired）用上依赖注入，通过 new 出来的对象是无法通过 annotation 得到注入的。
- 个人推荐构造器依赖注入，这种情况下测试友好，对象构造完整性好，显式地告诉你必须 mock/stub 哪个对象。

说完依赖注入我们再看刚才的充血模型：

```
public class Father{
    private SonRepository sonRepo;
    private Son getSon(){return sonRepo.getByFatherId(this.id);}
    public Father(SonRepository sonRepo){this.sonRepo = sonRepo;}
}
```

新建一个 Father 的时候需要赋值一个 SonRepository，这显然在写代码的时候是非常让人恼火的，那么我们是否可以通过依赖注入的方式把 SonRepository 注入进去呢？ Father 在这里不可能是一个 singleton 对象，它可能在两个场景下被 new 出来：新建、查询，从 Father 的构造过程，SonRepository 是无法注入的。这时工厂模式就显示出其意义了（很多人认为工厂模式就是一个摆设）：

```
@Component
public class FatherFactory{
    private SonRepository sonRepo;
    @Autowired
    public FatherFactory(SonRepository sonRepo){}
    public Father createFather(){
        return new Father(sonRepo);
    }
}
```

由于 FatheFactory 是系统生成的 singleton 对象，SonRepository 自然可以注入到 Factory 里，newFather 方法隐藏了这个注入的 sonRepo，这样

new 一个 Father 对象就变干净了。

4. 领域模型：测试友好

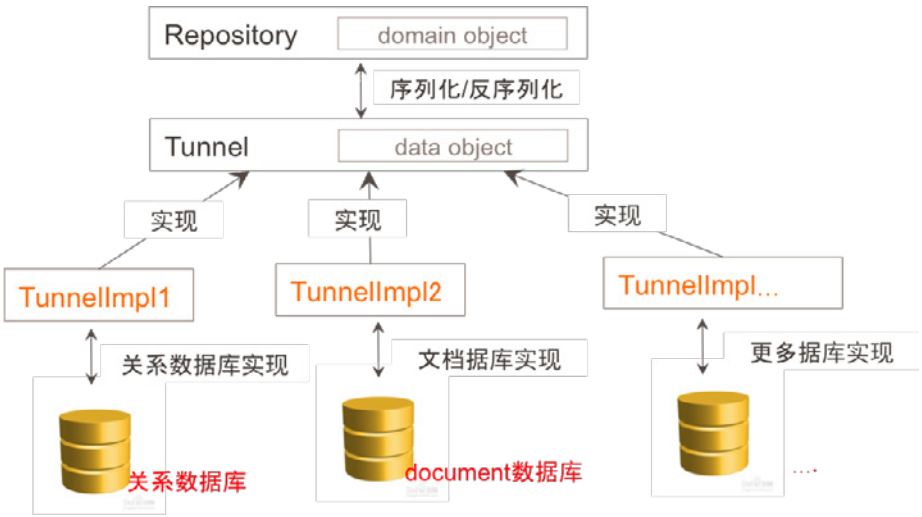
失血模型和贫血模型是天然测试友好的（其实失血模型也没啥好测试的），因为他们都是纯内存对象。但实际应用中充血模型是存在的，要不就是把 domain 对象拆散，变得稍微不那么优雅（当然可以，贫血和充血的战争从来就没有断过）。那么在充血模型下，对象里带上了 persisitence 特性，这就对数据库有了依赖，mock/stub 掉这些依赖是高效单元化测试的基本要求，我们再看 Father 这个例子：

```
public class Father{
    private SonRepository sonRepo;//=new SonRepository() 这里不能构造
    private getSon(){return sonRepo.getByFatherId(this.id);}
    // 放到构造函数里
    public Father(SonRepository sonRepo){this.sonRepo = sonRepo;}
}
```

把 SonRepository 放到构造函数的意义就是为了测试的友好性，通过 mock/stub 这个 Repository，单元测试就可以顺利完成。

5. 领域模型：盒马模式下 repository 的实现方式

按照 object domain 的思路，领域模型存在于内存对象里，这些对象最终都要落到数据库，由于摆脱了领域模型的束缚，数据库设计是灵活多

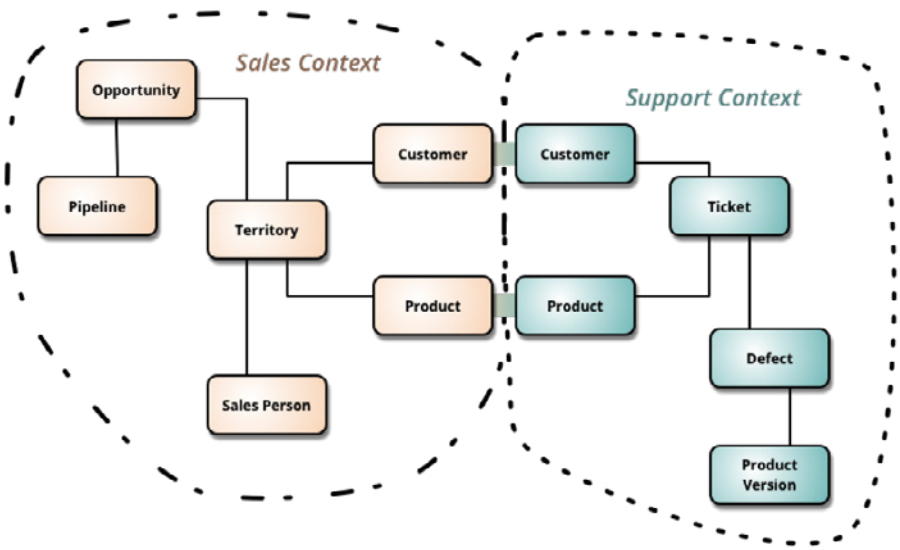


变的。在盒马，domain object 是怎么进入到数据库的呢。

在盒马，我们设计了 Tunnel 这个独特的接口，通过这个接口我们可以实现对 domain 对象在不同类型数据库的存取。Repository 并没有直接进行持久化工作，而是将 domain 对象转换成 POJO 交给 Tunnel 去做持久化工作，Tunnel 具体可以在任何包实现，这样，部署上，domain 领域模型（domain objects+repositories）和持久化 (Tunnels) 完全的分开，domain 包成为了单纯的内存对象集。

6. 领域模型：部署架构

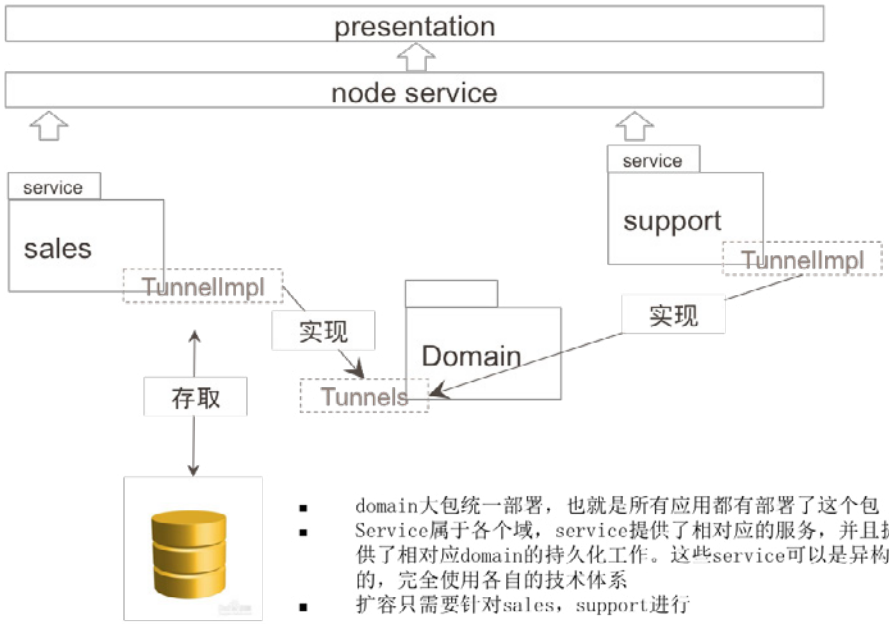
盒马业务具有很强的整体性：从供应商采购，到商品快递到用户手上，对象之间关系是比较明确的，原则上可以采用一个大而全的领域模型，也可以运用 boundedContext 方式拆分子域，并在交接处处理好数据传送，这里引用老马的一幅图：



我个人倾向于大 domain 的做法，我倾向（所以实际情况不是这样的）的部署结构如下图所示。

结语

盒马在架构设计上还在做更多的探索，在 2B+ 互联网的崭新业务模



式下，有很多可以深入探讨的细节。DDD 在盒马已经迈出了坚实的第一步，并且在业务扩展性和系统稳定性上经受了实战的考验。基于互联网分布式的工作流引擎（Noble），完全互联网的图形绘制引擎（Ivy）都在精心打磨中，期待在未来的就几个月里，盒马工程师们给大家奉献更多的设计作品。

作者简介

张群辉，阿里盒马架构总监。10 多年技术及管理实战经验，前阿里基础机构事业部工程效率总监，长期在一线指导大型复杂系统的架构设计。DevOps、微服务架构及领域驱动设计国内最早的实践者一员。崇尚实践出真知，一直奋斗在技术一线。

今天我们还需要关注 DDD 吗?

作者 雨多田光

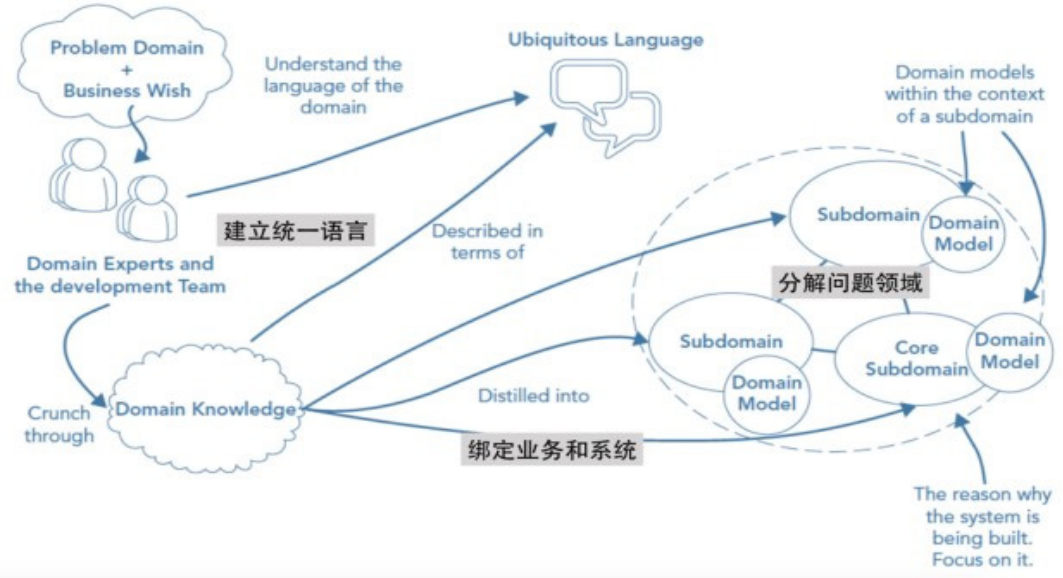
12 月 8 日，ThoughtWorks 举办了第一届 DDD（Domain-Driven Design，领域驱动设计）中国峰会，会上 DDD 领域的实践者分享了他们对于 DDD 的理解与应用。DDD 是什么？我们认为它是比系统架构更高层次的概念，它是一种设计思想，很多时下流行的架构模式都是在这种思想影响下产生的，像最近备受关注的 CQRS、微服务其实都存在 DDD 思想的影子；同时，DDD 还在软件行业的其它方方面面影响不断。但是 DDD 本身相关内容并没有多少人去传播、去分享，这不禁让我们想问：我们需要去关注 DDD 吗?

DDD 究竟意味着什么？在历史的进程中 DDD 的发展是怎么样的?

它未来的发展前景如何？带着一系列问题，我们在会场采访了 DDD 资深专家、Event Storming 之父 Alberto Brandolini 与 ThoughtWorks 资深咨询师王威。

DDD 是什么？

要讨论需不需要关注 DDD，首要的是先了解 DDD 是什么。Alberto 认为，DDD 是一种在面向高度复杂的软件系统时，关于如何去建模的方法论，“它的关键点是根据系统的复杂程度，建立合适的模型”。在 Alberto 当天的演讲“Why DDD Maters Today？”中，他提到“在一个系统中，没有一个人能完全掌握系统的全貌”，在多人参与的系统中，DDD 正是可以通过在不同角色之间进行协作，使参与者达成统一认知，对齐系统设计与程序实际所服务的业务领域。



图片源链接

具体来讲，DDD 方法论在系统建模过程中，可以为团队中的各个角色提供一套统一语言，避免组件划分过程中的边界错位，完成领域图预演、需求分析、架构模型、代码模型、测试等工作。“统一语言”概念在 DDD

中极为重要，在一个系统的构建过程中，往往业务人员关注的是业务架构，而技术人员则关注系统架构的表述方式；在将业务架构映射到系统架构的时候，需要经过一层“翻译”工作；而业务只要发生变化，就会影响到系统，系统就要重新对业务进行翻译，这会使工作变得复杂、低效。在 DDD 中，使用一个统一语言，可以直接将业务架构与系统架构绑定，不需要进一步去翻译，从而增强系统对业务的响应速度。

“领域驱动设计”中的“领域”一词指的是要实现的软件系统所要解决的实际问题所处的整个领域范围，它不仅包括系统架构的相关问题，还涉及到系统所支持的业务等内容，但它是与具体的开发技术无关的。也就是说 DDD 关注的是要构建的系统中，关于所要解决的问题的业务、流程和数据等内容是如何工作的，在这些东西理清之后，DDD 去构建出一个模型，接着再去选择具体的实现技术。DDD 强调的是解耦具体实现技术，所以它可以迅速梳理核心业务逻辑。

DDD 并不是直接给你建议某一个系统架构，它的执行结果是呈现一个方案，可以从这个构建出的模型中决定你去用什么技术来实现什么样的架构，进而来完成一个系统的设计。技术在这个过程中是被选择的，备选的各种技术只是像一个列表一样摆在眼前，它要根据你的领域需求来选择，比如“选择采用微服务架构”。

也正是因为它关注的是领域，而不是具体技术，所以 DDD 其实不仅可以应用在软件系统的开发中，也可以在其它领域，诸如测试体系的建设、公司的管理、需求变更的跟进和项目的管理。

总结起来，DDD 的一个生命周期是这样的：在设计和实现一个系统的时候，这个系统所要处理问题的领域专家和开发人员以一套统一语言进行协作，共同完成该领域模型的构建，在这个过程中，业务架构和系统架构等问题都得到了解决，之后将领域模型中关于系统架构的主体映射为实现代码，完成系统的实现落地。而用什么方式去做领域模型的构建，方法是多样的，Alberto 自己就为此发明了 Event Storming（事件风暴），并成为了一种经典的 DDD 落地模式。

Alberto 补充到：“为了方便理解，可以类比精益创业（Lean Startup），在我看来它是与 DDD 同个层次的概念，它也是一种能够通过快速对业务进行分析，快速去建模，支撑业务的模式。”

从微服务到 DDD

DDD 自 2004 年出现以来，其核心概念基本没发生什么变化，但是这些年来，DDD 整体的传播与实践都在向好的方向发展着，Alberto 认为有几个时间点使他印象深刻：

2003 年，Eric Evans 发布了影响深远的《Domain-Driven Design: Tackling Complexity in the Heart of Software》（领域驱动设计：软件核心复杂性应对之道）一书，DDD 问世，开始受到关注；

2007 年，Alberto 开始接触 DDD，他听了 Eric 的演讲，这让他很震撼，因为他在这之中了解到 DDD 对于处理界限上下文（BoundedContext）的思路很有价值。于是他开始深入了解 DDD；

到了 2011 年，这是 Alberto 认为对 DDD 的发展至关重要的一年，这一年是 DDD 思想大爆发的时期。在这一年中，社区中聚集了一些 DDD 的思想领袖（Thought Leader），Eric Evans 自不必说，还有像《Implementing Domain-Driven Design》（实现领域驱动）的作者 Vaughn Vernon、微服务巨匠 Martin Fowler 等人，他们聚到了一起，纷纷抛出自己的想法。大家突然之间发现，原来 DDD 可以做的事情有很多，它可以用来做 CQRS，可以用来做测试体系的建设，也可以用来做项目的管理……DDD 的应用场景变得多了起来。同时，Alberto 自己在 Event Storming 方面的想法获得成功，很多专家在演讲中引用了他的这种 DDD 建模方式。他认为这一年非常有价值，业界大牛们就 DDD 展开了深入的交流。

而从国内来看，王威认为 2014 年微服务的兴起是 DDD 的一个重要里程碑。不可否认，很多人是因为微服务才了解 DDD 的。在听说了微服务架构之后，人们觉得采用微服务架构会让系统开发与运维管理变得简单高效，同时实现的系统会更加合理，更加高可用、高性能，但是当他们实

际去做微服务架构的时候，有不少人会发现自己做得并不好，没法取得人们“吹捧”的那些效果，“就算用了微服务架构也不能解决他们的问题，反而带来很多开发与运维上的负担”。于是他们去咨询、去找方法，最后发现其实是自己划分微服务的方法出错了，这个时候才知道人们在谈论微服务的时候，其实都没有讲到一个点：应该用 DDD 的思想去指导微服务的实践。

是的，关于微服务架构怎么做，网上已经有很多相关理论和实践分享了，但是很少有人会说这个东西需要在 DDD 思想的指导下去做，在微服务的实践过程中，如果一开始就用 DDD 进行了全局模型设计，那么业务拆分、代码解耦等环节在实际架构建设时都是水到渠成的。而因为 DDD 使用统一语言来进行建模，这种高效建模、团队内部沟通无障碍和快速响应业务变化的特点会让微服务架构的实现更加简易。许多人盲目地去做微服务，如果在那之前他们先了解了 DDD，那么在 DDD 的指导下，微服务或许又会有另一番美景；另一方面，许多人虽然不知道 DDD，但是他们在系统构建的过程中，思想其实或多或少都会与 DDD 相符，那么，如果能够提前去了解 DDD，“从 DDD 到微服务，而不是从微服务到 DDD”，全面而系统地从头到尾以 DDD 的思想来操作，就能进一步降低微服务架构过程中行差踏错的可能性。

当然，人们谈论微服务而不涉及 DDD，可能还有另一种情况，就是他们实际上就是在 DDD 的指导下完成了微服务架构，但是“由于在建模的过程中，核心领域就是公司的核心资产，公司一般是不会把这个东西拿出来分享的”，王威解释到。很容易理解，像大型电信、金融企业，他们的业务核心模型肯定是属于公司机密，不会对外分享的。这其实也就是如今虽然 DDD 已经可以被应用在各种业务场景下，但是我们很难看到 DDD 实践案例的一个重要原因。

不管从国外还是国内来看，目前 DDD 主要还是停留在社区层面，但是就像 Alberto 说的“去参加演讲，今年看到的是这些人，明年看到的基本还是这些人”，虽然社区仍然不大，参与者的忠诚度却是很高的。如今，

国外有比较知名的 DDD Europe、DDD Exchange 等会议，国内像这次也举办了“DDD 中国峰会”，随着对 DDD 的研究、实践与传播，“这个圈子正在变得越来越大，我们相信更多的 DDD 实践将会被分享”。

看到这些变化，Alberto 与王威都认为 DDD 迎来了发展的最佳时机。越来越多人关注 DDD，而且出现了更多的企业去使用 DDD 的优势做业务，这使得目前 DDD 的境况不会变得更糟，但是 Alberto 提出了他的担忧：“我害怕 DDD 会不会最后变得像敏捷（Agile）那样。”王威进一步解释：“敏捷一开始其实是很好的，它的原则非常理想，大家对它的实践也非常广，但是目前来看敏捷，会发现每个人的实践都不同，大家对它的认知可能有分歧，甚至有些实践背离了敏捷的初衷。”两位都不希望 DDD 将来会发生类似的情况。

我们需要关注 DDD 吗？

DDD 的思想在它问世的这十几年间已经深深地影响了软件行业的架构理论和各种实践发展，像 CQRS 架构、依赖反转、洋葱圈架构、EDA 事件驱动架构和微服务架构等都能找到 DDD 的影子。DDD 甚至影响了软件行业中的各个方面，比如在本次 DDD 会议上还可以看到，有的讲师从测试体系的建设、公司的管理、需求变更的跟进和项目的管理等方面分享了 DDD 的应用。

通常来说，DDD 适用于任何规模、任何性质的公司，这是一种通用的、具有指导意义的方法论，因此它可以在各个业务场景里发挥作用。

王威认为，DDD 作为一种方法论，我们更应该关注的是它能够帮助团队针对业务达成一个统一的认知。在这个业务变化节奏相当快的时代，系统架构是必须不断演进的，而 DDD 在这个过程中因为构建了团队的统一语言，使得在对业务的快速响应方面表现得出类拔萃，这是它的精髓，不管什么领域，只要有这种业务快速响应的需求，那么 DDD 都是适用的。以往想要启动一个系统架构设计的时候，需要 3、4 个月的时间去咨询，等待调研报告，但是如果以 DDD 的方式，那么 2、3 周就可以出一

份 Roadmap。王威说:“客户更看重的是 DDD 对整个业务领域统一的认知，以及对业务响应变化的能力，DDD 快速启动的能力。”Alberto 补充说到:“所有领域的企业都可以采用 DDD，DDD 应用最大的障碍就是去承认原先的架构并不是那么完善的。”

另一方面，在一些特定的领域，比如需要有人参与指导、进行技术交流，以及需要大量人力协作而容易导致秩序混乱的设计工作，DDD 因为其关注业务问题域的特性，可以使得执行效果更好。

“而具体到创业公司，他们因为需要使用成本更低的方式去打入市场，所以使用 DDD 会让这个过程更加顺利。”Alberto 认为这是创业公司的切入点。

所以回过头来看：我们需要关注 DDD 吗？不言而喻。Alberto 与王威在这个问题上都回答的特别坚定：Sure，需要！

谈及如何去关注 DDD，Alberto 说他的经验是积极参与到本地社区中去，他认为这是一个很有效的方式，社区可以提供很多信息。同时他觉得看书并不是很好的方式，与其一个人看书学习，不如找一个人一起学习，确保你掌握了学到的东西。

而为 DDD 实践者构建一个社区，让关注 DDD 的人有一个交流平台，也正是此次 DDD 中国峰会的价值所在，举办方想以此在国内第一次正式地告诉软件设计从业者：DDD 是在我们这个业务高度不确定的时代，解决业务问题，适应业务变化时需要采用的架构思想。

受访嘉宾介绍

Alberto Brandolini： 全球软件巨匠，Event Storming 之父。

王威：ThoughtWorks China 咨询团队软件架构和企业转型咨询师。

中小型研发团队架构实践：如何规范公司所有应用分层？

作者 张辉清

一、写在前面

应用分层这件事情看起来很简单，但每个程序员都有自己的一套，哪怕是初学者。如何让一家公司的几百个应用采用统一的分层结构，并得到大部分程序员的认同呢？这可不是件简单的事情，接下来以我们真实案例与大家一起探讨，先问大家两个技术问题：

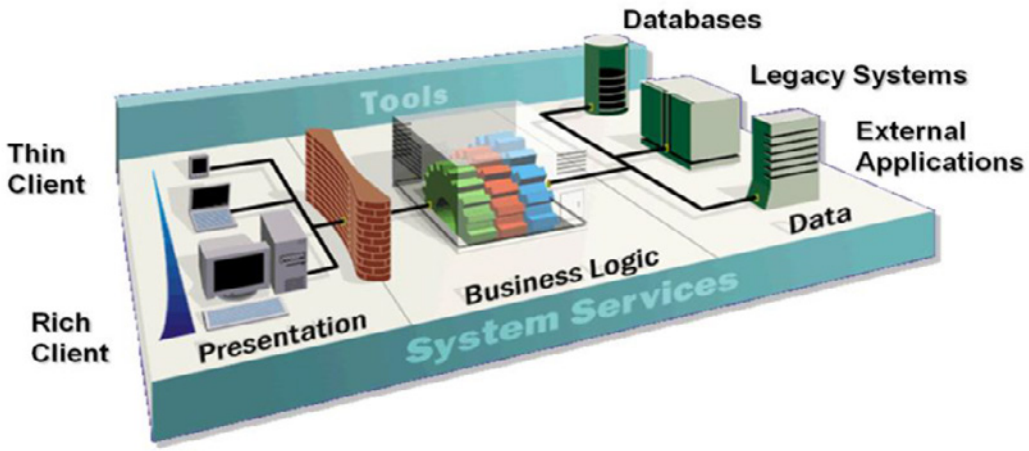
服务的调用代码你觉得放到哪一层好呢？

- A 表现层
- B 业务逻辑层

- C 数据层
- D 公共层

如何组织好 VO(View Object 视图对象)、BO(Business Object 业务对象)、DO(Data Object 数据对象)、DTO(Data Transfer Object 数据传输对象)呢?

不同的人会有不同的答案，所以要统一公司应用分层，以减少开发维护学习成本。统一应用分层要可大可小、简单易用、支持多种场景，我们采用 IPO 方式：I 是 Input、O 是 Output、P 是 Process，一进一出一处理。应用系统的本质是机器，是处理设备，一进一出一处理。



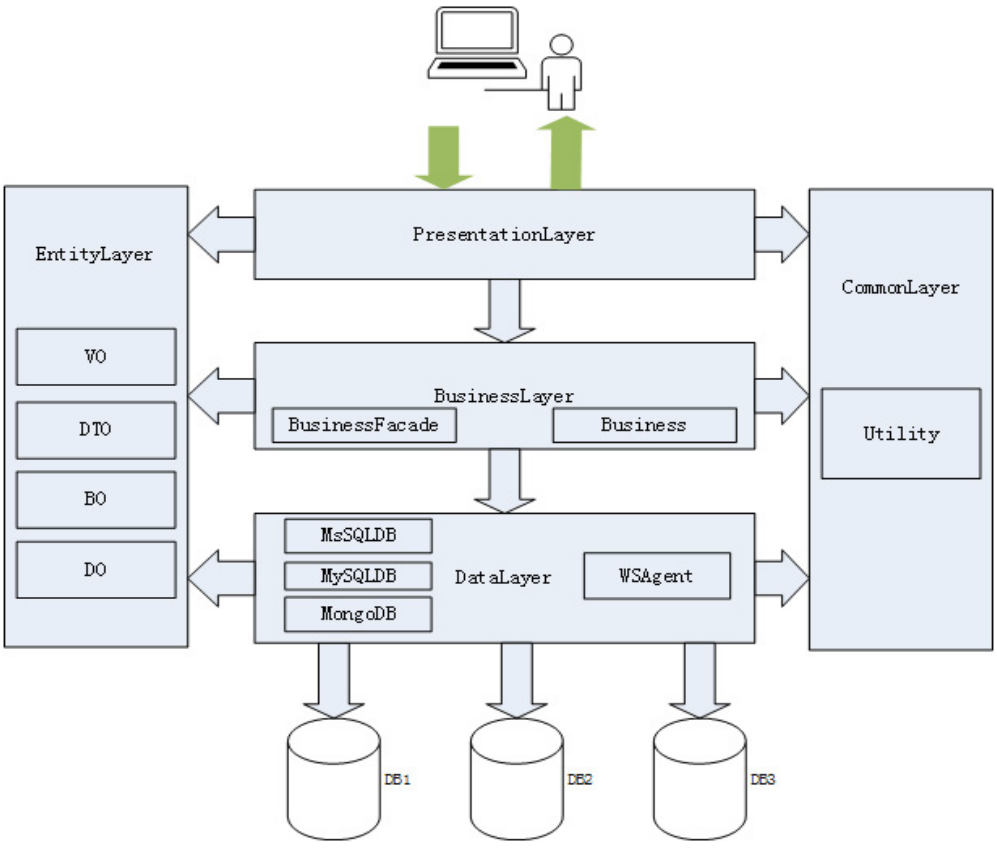
IPO 原理图

二、统一逻辑架构

职责说明:

- 文件夹分层法：应用分层采用文件夹方式的优点是可大可小、简单易用、统一规范，可以包括 5 个项目，也可以包括 50 个项目，以满足所有业务应用的多种不同场景;
- 调用规约：在开发过程中，需要遵循分层架构的约束，禁止跨层次的调用;

- 下层为上层服务：以用户为中心，以目标为导向。上层（业务逻辑层）需要什么，下层（数据访问层）提供什么，而不是下层（数据访问层）有什么，就向上层（业务逻辑层）提供什么；
- 实体层规约：DO 是数据表对象，不是数据访问层对象，不是只能给数据访问层使用；DTO 是网络传输对象，不是表现层对象，不是只能给表现层使用；BO 是内存计算逻辑对象，不是业务逻辑层对象，不是只能给业务逻辑层使用 。如果仅限定在本层访问，则导致单个应用内大量没有价值的对象转换。以用户为中心来设计实体类，可以减少无价值重复对象和无用转换；
- U 型访问：下行时表现层是 Input，业务逻辑层是 Process，数据访问层是 Output。上行时数据访问层是 Input，业务逻辑层是 Process，表现层就 Output。

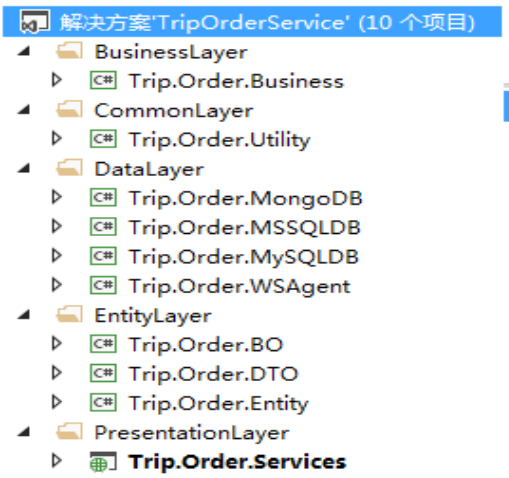


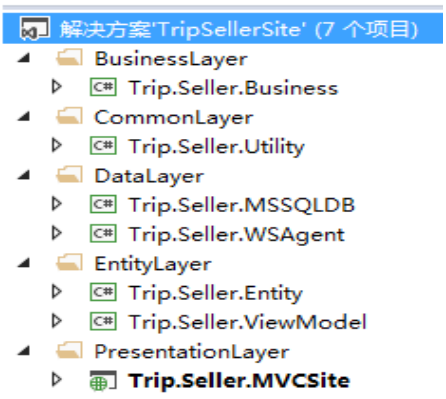
层英文名	中文名	说明	
PresentationLayer	表现层文件夹	上层向用户提供服务，负责视图展示。项目类型包括WebSite、WebForm、MVC、WCF、WebService等。	
BusinessLayer	业务逻辑层文件夹	中间逻辑处理，负责应用系统的业务逻辑的处理。	
DataLayer	数据访问层文件夹	下层调用服务，负责数据资源提供方如数据库、SOA、OpenAPI的交互。	
EntityLayer	实体层文件夹	VO：View Object视图对象； DTO：Data Transfer Object数据传输对象； BO：Business Object业务对象； DO：Data Object数据对象； 在实际项目中，为简化设计可进行裁剪，BO和DO为可选，DTO属于服务项目类型，VO属于网站项目类型，也不会同时存在。	
CommonLayer	公共层文件夹	工具类库，负责提供应用系统中常用的操作。	
TestLayer	测试层文件夹	单元测试（可选），负责对其它类库的自动化单元测试。	

统一应用分层的逻辑架构图

三、我们的具体规范

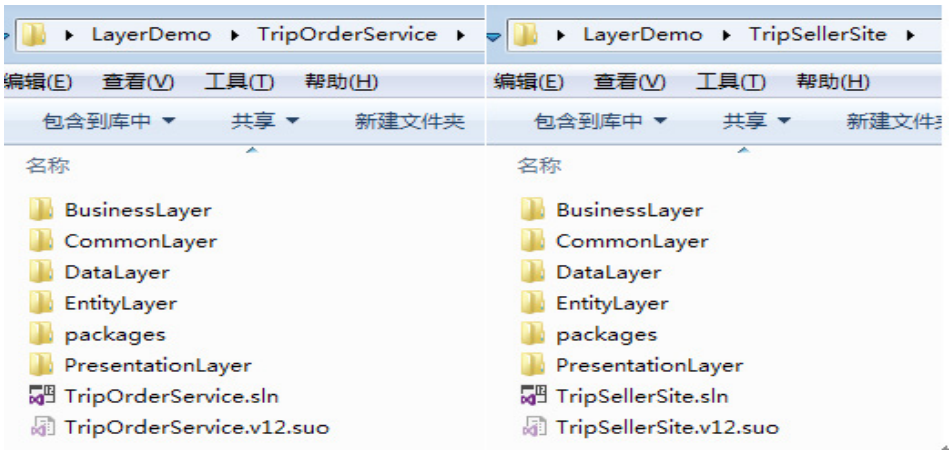
此规范我们用了四年，牵涉几百个应用，200 多个研发人员，是一个成功的实践。接下来就借用本文提供下载的 TripOrderService、TripSellerMVCSite 这两个 Demo 来进行具体规范的说明，以下是截图：





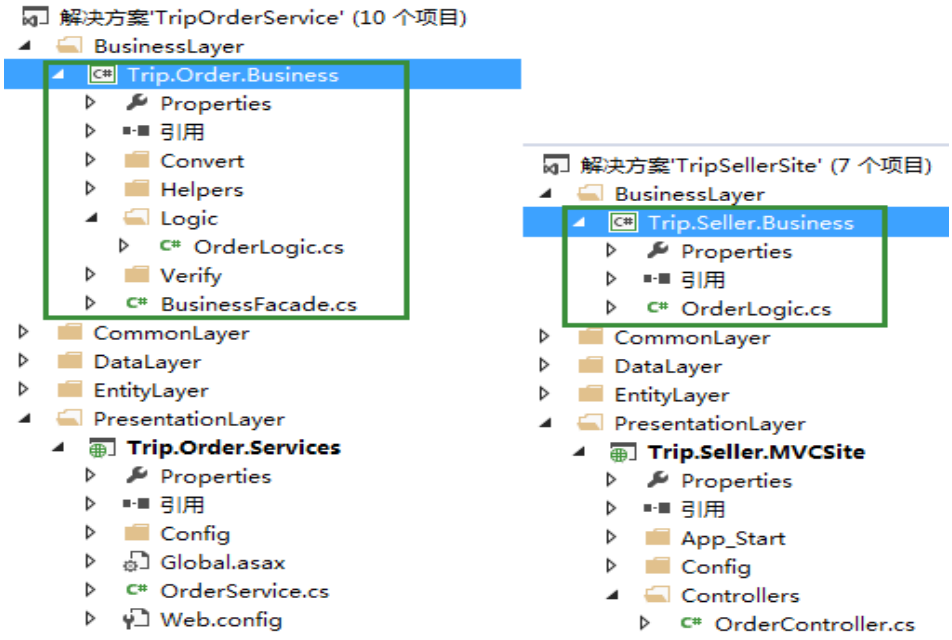
3.1、项目命名规则

项目命名规则：{产品线英文名全称}.{子系统英文名全称 + 应用



名 }.{ 项目职责英文名全称 }，如：Trip.Seller.DTO。

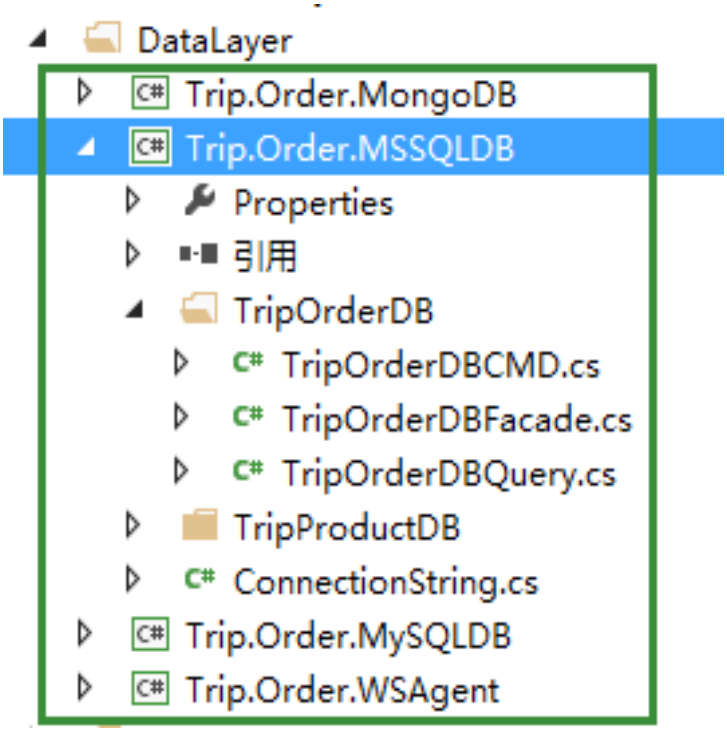
3.2、业务逻辑层的项目规范



规范说明：

1. 项目名的命名规则：{产品线英文名全称}.{子系统英文名全称 + 应用名}.xxxBusiness，如上图的 Trip.Order.Business。
2. 类名以 Logic 结尾，如上图的 OrderLogic.cs。

3.3、数据操作项目规范

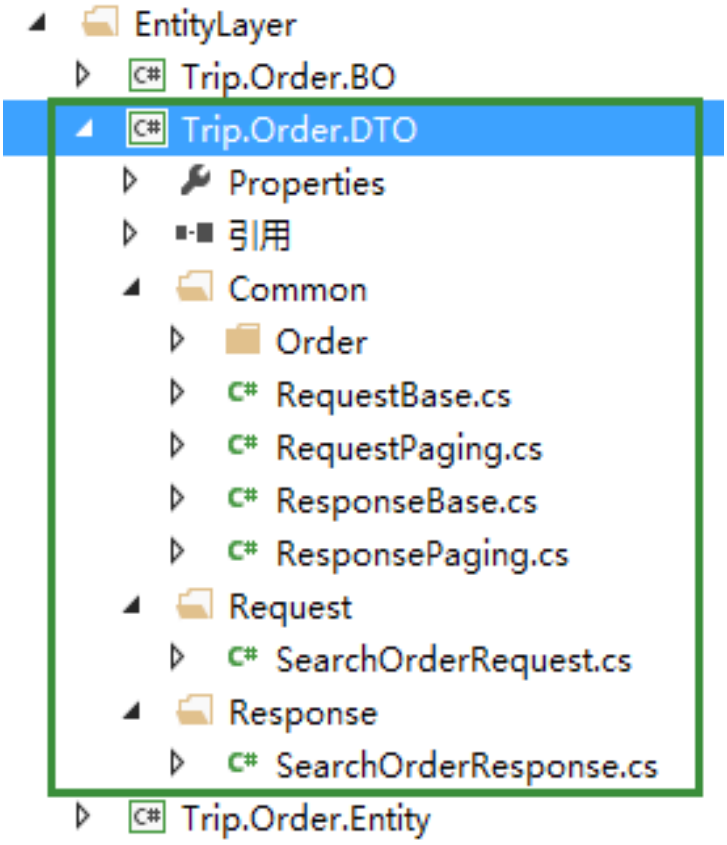


规范说明：

1. 各数据操作项目名根据使用什么数据库进行分类，然后以 DB 为结尾，具体命名规则是：{产品线英文名全称}.{子系统英文名全称 + 应用名}.{使用什么数据库}DB，如上图的 Trip.Seller.MSSQLDB。
2. 如果涉及到多个数据库访问的，那么数据操作项目下的类文件需要按数据库名称（以 DB 为结尾）创建文件夹分开，如上图的 TripOrderDB 文件夹。
3. 建议在应用中使用 SQL 语句，不使用存储过程。在数据库中不新增存储过程，但旧的存储过程可以继续使用和修改。
4. 分页建议使用数据库（如 SQLServer）的最新特性进行分页，并将每个分页 SQL 直接写到应用中。

3.4、实体类项目规范

数据传输对象 DTO 规范

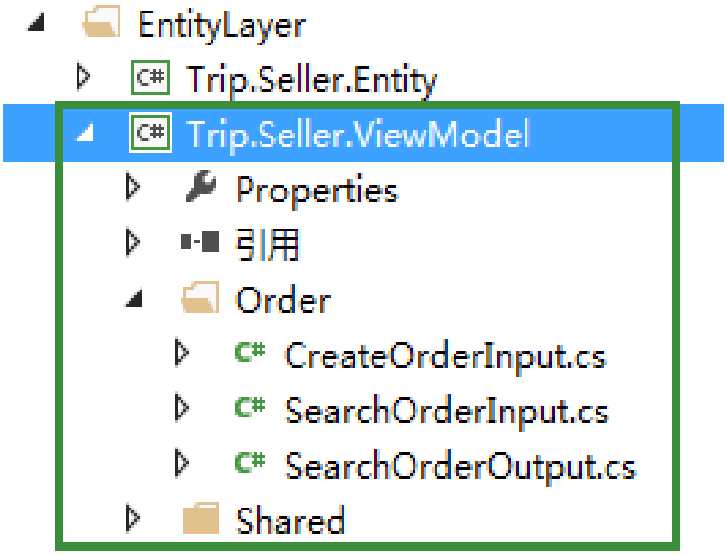


规范说明：

1. DTO 项目命名规则：{产品线英文名全称}.{子系统英文名全称 + 应用名}.DTO，如上图的 Trip.Order.DTO。
2. 请求参数 DTO 实体类、响应 DTO 实体类存放规范以及其命名规则：
   - 请求参数 DTO 实体类放在 Request 文件夹下，且命名规则为：以 Request 结尾，如上图的 SearchOrderRequest.cs。
   - 响应 DTO 实体类放在 Response 文件夹下，且命名规则为：以 Response 结尾，如上图的 SearchOrderResponse.cs。
   - 如果请求参数 DTO 实体类或响应 DTO 实体类的属性中有对象或枚举，那么这些对象所属的类、枚举放在 DTO 项目的 Common 文件夹下。
3. 如果请求参数 DTO 实体类、响应 DTO 实体类有基类要继承，那

么建议为基类取名为 RequestBase.cs、ResponseBase.cs。且这些基类直接放在 DTO 项目的 Common 文件夹下。

视图对象 VO 规范



规范说明：

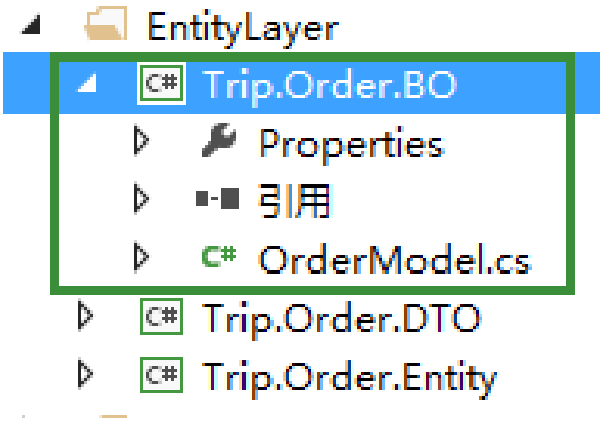
1. VO 项目命名规则：{产品线英文名全称}.{子系统英文名全称 + 应用名}.ViewModel，如上图的 Trip.Seller.ViewModel。
2. 各 VO 实体类，我们用 Controller 名作为文件夹名进行分开，如上图的 Order 文件夹。
3. VO 实体类名的命名建议：
   - 请求参数 VO 实体类以 Input/Form/Query 结尾，如上图的 SearchOrderInput.cs。
   - 响应 VO 实体类以 Output/List/Result 结尾，如上图的 SearchOrderOutput.cs。

业务对象 BO 规范（可选）

BO 实体类名以 Model 为结尾：

规范说明：

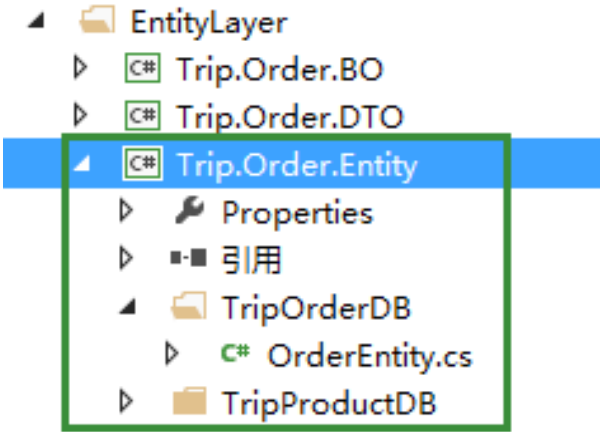
1. BO 项目命名规则：{产品线英文名全称}.{子系统英文名全称 + 应



用名}.BO，如上图的 Trip.Order.BO;

2. 以 Model 结尾，如上图的 OrderModel.cs；
3. 为了简化设计，BO 项目为可选，可在 DO 项目里建文件夹。

数据对象 DO 规范（可选）



规范说明：

1. DO 项目命名规则：{产品线英文名全称}.{子系统英文名全称 + 应用名}.Entity，如上图的 Trip.Seller.Entity；
2. 如果涉及到多个数据库访问的，那么需要按数据库名称（以 DB 为结尾）创建文件夹分开，如上图的 TripOrderDB 文件夹；
3. 表名 +Entity 结尾，如上图的 OrderEntity.cs；
4. DO 是数据表对象，供单表 CURD 操作。对于多表查询请求对象和返回对象，可定义新对象或使用现有对象（DTO/BO）来完成。

3.5、数据库连接配置规范

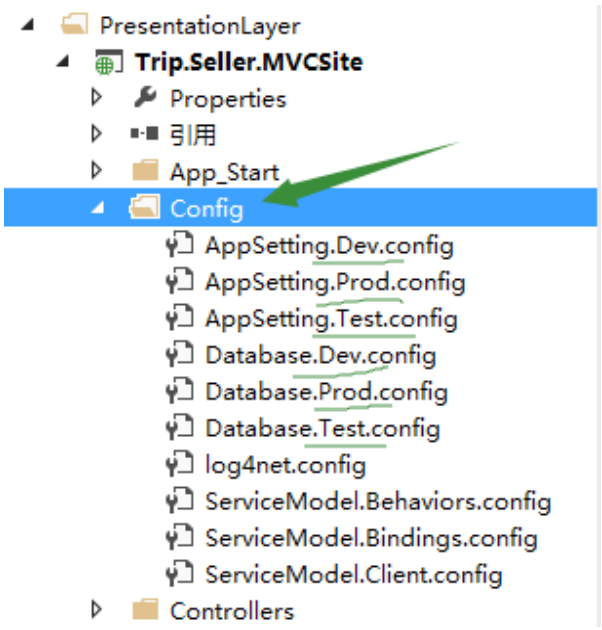
Database.Dev.config

```
<?xml version="1.0" encoding="utf-8"?>
<connectionStrings>
  <add name="TripOrderDB_SELECT" connectionString="M074cB4ekCz/kIFnIbx7PgM4K79PfGCn77Yag5dGhk7J1EcEpFTL05bX2ryVAOvJLPPSfU4eECDvvSKjIZWS
   +NgkAx5JFAOamVpnCaAJYrCSQWo9MoZkjCX4QTcDy2o8ASCotkoMdDKOPAL0aFACeqLp2k0WKA4s" providerName="System.Data.SqlClient"/>
  <add name="TripOrderDB_INSERT" connectionString="M074cB4ekCz/kIFnIbx7PgM4K79PfGCn77Yag5dGhk7J1EcEpFTL05bX2ryVAOvJLPPSfU4eECDvvSKjIZWS
   +NgkAx5JFAOamVpnCaAJYrCSQWo9MoZkjCX4QTcDy2o8ASCotkoMdDKOPAL0aFACeqLp2k0WKA4s" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

规范说明：

1. 数据库连接的配置必须读写分离。
2. 数据库连接字符串建议加密处理。
3. 数据库连接配置名的命名规则：{以 DB 为结尾的数据库名称}_ 读写类型，如：TripOrderDB_SELECT、TripOrderDB_INSERT。

3.6、配置文件方面的规范



规范说明：

1. 所有配置文件（除 Web.config 文件外）都必须放到 Config 文件夹下。

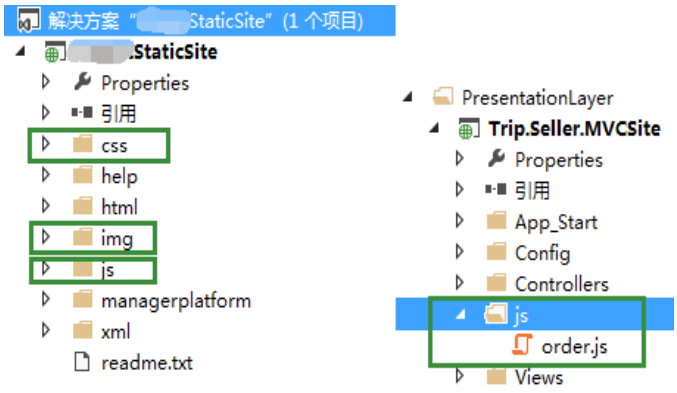
Web.config

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  有关如何配置 ASP.NET 应用程序的详细信息，请访问
  http://go.microsoft.com/fwlink/?LinkId=301880
  -->
<configuration>
  <appSettings configSource="Config\AppSetting.Dev.config" />
  <connectionStrings configSource="Config\Database.Dev.config" />

  <system.serviceModel>
    <bindings configSource="Config\ServiceModel.Bindings.config" />
    <behaviors configSource="Config\ServiceModel.Behaviors.config" />
    <client configSource="Config\ServiceModel.Client.config" />
  </system.serviceModel>
```

2. 所有配置文件（除 Web.config 文件外）按不同环境区分开，具体命名规则是：{功能模块英文名}.{环境英文简称名}.config，其中本地环境的英文简称名是 Dev，测试环境的英文简称名是 Test，正式环境的英文简称名是 Prod，如上图的 AppSetting.Dev.config。
3. 保持 Web.config 配置文件的干净，只留环境设置节点。

3.7、静态资源文件方面的规范



规范说明：

1. 公共的静态资源文件（css、js、image 等）放在另外的静态站点中，统一由前端进行开发和维护。一般，css 文件放在 css 文件夹下，js 文件放在 js 文件夹下，image 图片文件放在 img 文件夹下。
2. 与某项业务有关的 js 文件可以放到各自业务项目的表现层 PresentationLayer 下，以方便开发人员调试，js 文件可放在项目的 js 文件夹下。
3. 静态资源文件必须使用版本号管理，以防更新后由于客户端浏览器缓存而导致站点使用的依然是旧版本的静态资源文件：

```
<script src="~/js/order.js?v=@AppSetting.StaticFileVersion"></
script>
```

四、写在最后

4.1、问题回答

问：服务的调用代码应该放到哪一层呢？A 表现层、B 业务逻辑层、C 数据层、D 公共层。

我们的规范是统一放到数据资源访问层即 C。上层提供服务，下层调用服务，中间处理业务逻辑。

问：如何组织好 VO(View Object 视图对象)、BO(Business Object 业务对象)、DO(Data Object 数据对象)、DTO(Data Transfer Object 数据传输对象) 呢？

通常有两种做法，限定访问范围和不限定访问范围，实际项目中可根据需要选择、折中或裁剪。我们使用后者，将 EntityLayer 作为通用对象放到左侧，具体可参考实体层规约：“DO 是数据表对象，不是数据访问层对象，不是只能给数据访问层使用；DTO 是网络传输对象，不是表现层对象，不是只能给表现层使用；BO 是内存计算逻辑对象，不是业务逻辑层对象，不是只能给业务逻辑层使用 。如果仅限定在本层访问，则导致单个应用内大量没有价值的对象转换。以用户为中心来设计实体类，可以减少无价值重复对象和无用转换。”

问：应用分层范例代码的编写需要注意些什么？

应用分层范例的代码要想写好，非常不容易，很容易引起争议，很难让所有人满意。我们在具体实践时遵循以下几点：

应用分层范例的主要价值是明确层的职责和交互，每个层的职责是什么，哪些要干，哪些不要干，以及层与层之间依赖和交互；

- 私人定制：减少通用帮助类的编写，如果每一个应用中有大量相同的帮助类，这在架构层面上是有问题。在我们的几百个线上应用中，尽管减少通用的代码，包括分页帮助类、数据库帮助类、

缓存帮助类、MQ 帮助类、日志帮助类、AOP 帮助类、线程帮助类。业务应用的重点是为业务服务，每一个应用都是特别的，都需要私人定制，极少有通用的代码，如果有，那么应该由框架或组件专门解决；

- 少即是多：应用的场景多，参考人员多，每个人想法不同，牵涉的时间长，所以尽量只做大家都认同的规范、正确的事情，要自底向上、要减少有争议的代码范例，否则一个错误将会放大百倍、一个有争议的规范将会很难推行。
- 追求简单：代码编写可分为三个层次，简单、复杂、简单。第一简单是不知道的简单，第二个复杂是知道后的复杂，第三个简单是知道后有取舍的简单。范例代码要追求简单，既可轻松扩展支持复杂场景，又要简单到初级程序员也能操作。
- 内聚大于解耦：内聚是什么，内聚是部门内有共同的目标，然后大家紧密合作。解耦是什么，解耦是部门间各自职责明确，然后减少不必要的连接。一个应用如同一个部门，应有一个共同的目标和职责，然后大家紧密合作。

换句话说，应用内部应减少不必要契约接口（如同公司间才签约合同），减少不必要的依赖注入实现，减少不必要且代价过大的解耦。一切以简单实用为主，以应用价值输出、应用的目标（接口或界面）为导向。

4.2、Demo 下载

LayerDemo 下载地址链接。

作者介绍

张辉清，10 多年的 IT 老兵，先后担任携程架构师、古大集团首席架构、中青易游 CTO 等职务，主导过两家公司的技术架构升级改造工作。现关注架构与工程效率，技术与业务的匹配与融合，技术价值与创新。

一次 Serverless 架构改造实践：基因样本比对

作者 高 远

Serverless 是一种新兴的无服务器架构，使用它，开发者只需专注于代码，无需关心运维、资源交付或者部署。本文将从代码的角度，通过改造一个 Python 应用来帮助读者从侧面理解 Serverless，让应用继承 Serverless 架构的优点。

现有资源：

- 一个成熟的基因对比算法（Python实现，运行一次的时间花费为 2 秒）
- 2020 个基因样本文件（每个文件的大小为 2M，可以直接作为算法的输入）

- 一台 8 核心云主机

基因检测服务

我们使用上面的资源来对比两个人的基因样本并 print 对比结果（如：有直系血缘关系的概率）。

我们构造目录结构如下：

```
.
├── relation.py
└── samples
    ├── one.sample
    └── two.sample
```

relations.py 代码如下：

```
import sys

def relationship_algorithm(human_sample_one, human_sample_two):
    # it's a secret
    return result
if __name__ == '__main__':
    length = len(sys.argv)
     # sys.argv is a list, the first element always be the script's
name
    if length != 3:
        sys.stderr.write('Need two samples')
    else:
        # read the first sample
        with open(sys.argv[1], 'r') as sample_one:
            sample_one_list = sample_one.readlines()
        # read the second sample
        with open(sys.argv[2], 'r') as sample_two:
            sample_two_list = sample_two.readlines()
        # run the algorithm
           print relationship_algirithm(sample_one_list, sample_two_
list)
```

使用方法如下：

```
python relation.py ./samples/one.sample ./samples/two.sample
0.054
```

流程比较简单，从本地磁盘读取两个代表基因序列的文件，经过算法计算，最后返回结果。

我们接到了如下业务需求

假设有 2000 人寻找自己的孩子，20 人寻找自己的父亲：

首先，收集唾液样本经过专业仪器分析后，然后生成样本文件并上传到我们的主机上，一共 2020 个样本文件，最后我们需要运行上面的算法：

```
2000 * 20 = 40000（次）
```

才可完成需求，我们计算一下总花费的时间：

```
40000（次） * 2（秒）= 80000 （秒）
80000（秒）/ 60.0 / 60.0 ≈ 22.2（小时）
```

串行需要花费 22 小时才能算完，太慢了，不过我们的机器是 8 核心的，开 8 个进程一起算：

```
22.2 / 8 ≈ 2.76（小时）
```

也要快 3 个小时，还是太慢，假设 8 核算力已经到极限了，接下来如何优化呢？

一种 Serverless 产品：UGC

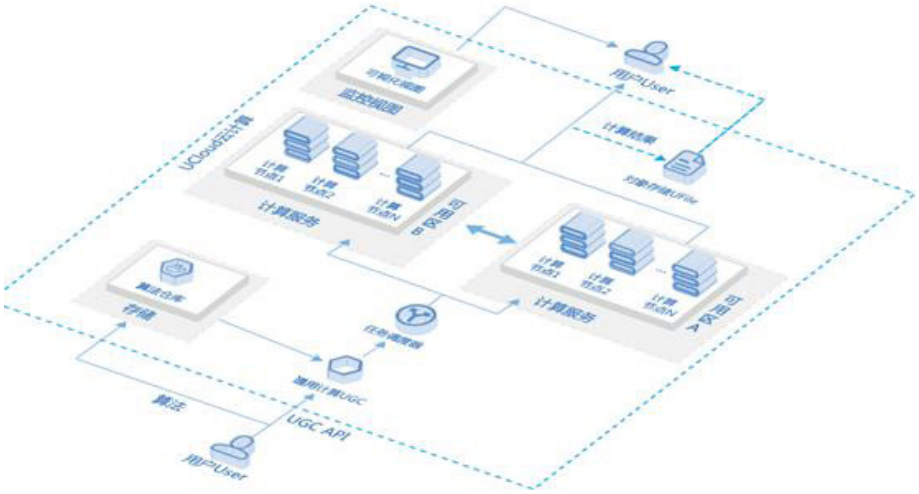
与 AWS 的 lambda 不同，UGC 允许你将计算密集型算法封装为 Docker Image （后文统称为「算法镜像」），只需将算法镜像 push 到指定的算法仓库中，UGC 会将算法镜像预先 pull 到一部分计算节点上，当你使用以下两种形式：

算法镜像的名字和一些验证信息通过 querystring 的形式 （例如：http://api.ugc.service.ucloud.cn?ImageName=relation&Token=!Q@W#E）

- 算法镜像所需的数据通过 HTTP body 的形式

特别构造的 HTTP 请求发送到 UGC 的 API 服务时，「任务调度器」

会帮你挑选已经 pull 成功算法镜像的节点，并将请求调度过去，然后启动此算法镜像「容器」将此请求的 HTTP body 以标准输入 stdin 的形式传到容器中，经过算法计算，再把算法的标准输出 stdout 和标准错误 stderr 打成一个 tar 包，以 HTTP body 的形式返回给你，你只需要把返回的 body 当做 tar 包来解压即可得到本次算法运行的结果。



UGC 产品架构图

讲了这么多，这个产品使你可以把密集的计算放到了数万的计算节点上，而不是我们小小的 8 核心机器，有数万核心可供使用，那么如何使用如此海量的计算资源呢，程序需要小小的改造一下。

针对此 Serverless 架构的改造

两部分：

改造算法中源数据从「文件输入」改为「标准输入」，输出改为「标准输出」

开发客户端构造 HTTP 请求，并提高并发

1. 改造算法输入输出

① 改造输入为 stdin

```
cat ./samples/one.sample ./samples/two.sample | python relation.py
```

这样把内容通过管道交给 relation.py 的 stdin，然后在 relation.py 中通

过以下方式拿到：

```
import sys
mystdin = sys.stdin.read()
# 这里的 mystdin 包含 ./samples/one.sample ./samples/two.sample 的全部内容，无分隔，实际使用可以自己设定分隔符来拆分
```

② 将算法的输出数据写入 stdout

```
# 把标准输入拆分为两个 sample
sample_one, sample_two = separate(mystdin)
 # 改造算法的输出为 stdout
def relationship_algorithm(sample_one, sample_two)
    # 改造前
    return result
    # 改造后
    sys.stdout.write(result)
```

到此就改造完了，很快吧。

2. 客户端与并发

刚才我们改造了算法镜像的逻辑（任务的执行），现在我们来看一下任务的提交：

构造 HTTP 请求并读取返回结果。

```
imageName = 'cn-bj2.ugchub.service.ucloud.cn/testbucket/relationship:0.1'
token = tokenManager.getToken() # SDK 有现成的

# summitTask 构造 HTTP 请求并将镜像的 stdout 打成 tar 包返回
response = submitTask(imageName, token, data)
```

它也支持异步请求。

之前提到，此 Serverless 产品会将算法的标准输出打成 tar 包放到 HTTP body 中返回给客户端，所以我们准备此解包函数：

```
import tarfile
import io
def untar(data):
```

```
    tar = tarfile.open(fileobj=io.BytesIO(data))
    for member in tar.getmembers():
        f = tar.extractfile(member)
        with open('result.txt', 'a') as resultf:
            strs = f.read()
            resultf.write(strs)
```

解开 tar 包，并将结果写入 result.txt 文件。

假设我们 2200 个样本文件的绝对路径列表可以通过 get_sample_list 方法拿到。

```
sample_2000_list, sample_20_list = get_sample_list()
```

计算 2000 个样本与 20 个样本的笛卡尔积，我们可以直接使用 itertools.product。

```
import itertools
all = list(itertools.product(sample_2000_list, sample_20_list))
 assert len(all) == 40000
```

结合上面的代码段，我们封装一个方法：

```
def worker(two_file_tuple):
    sample_one_dir, sample_two_dir = two_file_tuple
    with open(sample_one_dir) as onef:
        one_data = onef.read()
    with open(samle_two_dir) as twof:
        two_data = twof.read()
    data = one_data + two_data
    response = summitTask(imageName, token, data)
    untar(response)
```

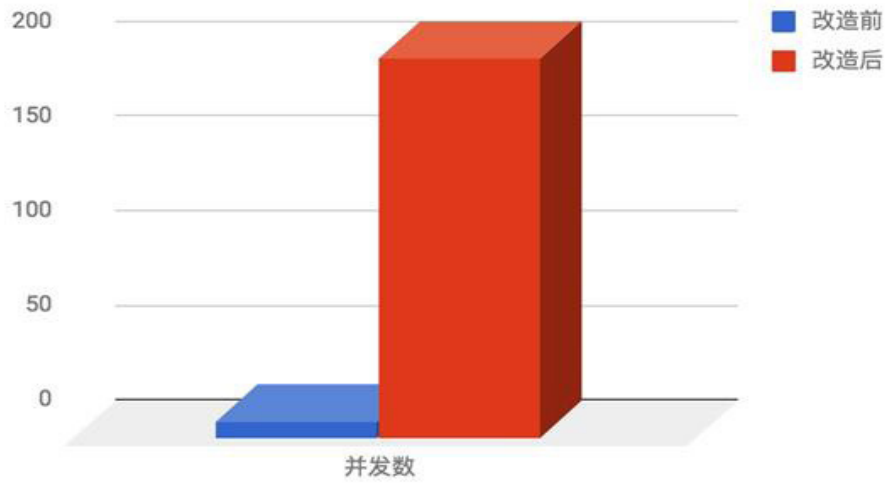
因为构造 HTTP 请求提交是 I/O 密集型而非计算密集型，所以我们使用协程池处理是非常高效的：

```
import gevent.pool
import gevent.monkey
gevent.monkey.patch_all() # 猴子补丁
pool = gevent.pool.Pool(200)
pool.map(worker, all)
```

只是提交任务 200，并发很轻松。

全部改造完成，我们来简单分析一下：

之前是 8 个进程跑计算密集型算法，现在我们把计算密集型算法放到了 Serverless 产品中，因为客户端是 I/O 密集型的，单机使用协程可以开很高的并发，我们不贪心，按 200 并发来算：



并发对比图

进阶阅读：上面这种情况下带宽反而有可能成为瓶颈，我们可以使用 gzip 来压缩 HTTP body，这是一个计算比较密集的操作，为了 8 核心算力的高效利用，可以将样本数据分为 8 份，启动 8 个进程，进程中再使用协程去提交任务就好了。

```
40000 * 2 = 80000（秒）
80000 / 200 = 400（秒）
```

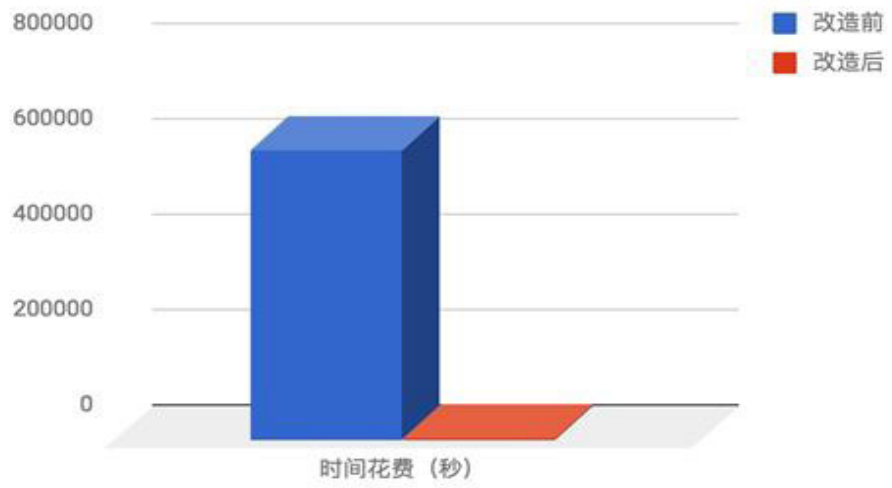
也就是说进行一组检测只需要 400 秒，从之前的 22 小时提高到 400 秒，成果斐然，图表更直观。

而且算力瓶颈还远未达到，任务提交的并发数还可以提升，再给我们一台机器提交任务，便可以缩短到 200 秒，4 台 100 秒，8 台 50 秒 ...

最重要的是改造后的架构还继承了 Serverless 架构的优点：

- 免运维
- 高可用

- 按需付费
- 发布简单



花费时间对比图

1. 免运维 -- 因为你没有服务器了……

2. 高可用 -- Serverless 服务一般依托云计算的强大基础设施，任何模块都不会只有单点，都尽可能做到跨可用区，或者跨交换机容灾，而且本次使用的服务有一个有趣的机制：同一个任务，你提交一次，会被多个节点执行，如果一个计算节点挂了，其他节点还可以正常返回，哪个先执行完，先返回哪个。

3. 按需付费 -- 文中说每个算法执行一次花费单核心 CPU 时间 2 秒，我们直接算一下花费。

```
2000 * 20 * 2 = 80000（秒）
80000 / 60 / 60 = 22.22（小时）
22.22（小时） * 0.09（元） * 1（核心） ~= 2（元）
每核时 0.09 元（即单核心 CPU 时间 1 小时，计费 9 分钱）
```

发布简单 -- 因为使用 Docker 作为载体，所以它是语言无关的，而且发布也很快，代码写好直接上传镜像就好了，至于灰度，客户端 imageName 指定不同版本即可区分不同代码了。

不同的 Serverless 产品可能有不同的改造方法，作为工程师，本人比

较喜欢这种方式，改造成本低，灵活性高，你觉得呢？欢迎展开讨论。

作者简介

高远，拥有多年 DevOps 经验， UCloud 实验室自动化运维运营平台负责人，干净、易懂代码的践行者，希望为计算机科学的发展尽一份力。（知乎 ID：临书）

[北京站] 2018

主办方 Geekbang. InfoQ

QCon
全球软件开发大会

北京·国际会议中心　演讲：2018年4月20-22日　培训：2018年4月23-24日

重磅技术大咖


Jun Rao
Confluent
联合创始人，Kafka作者之一


Katharina Probst
Netflix　工程总监


Georges Saab
Oracle
Java平台事业群VP


俞军
滴滴
产品高级副总裁


Bruce Eckel
《Java编程思想》作者


Neal Ford
ThoughtWorks　总监，
《卓有成效的程序员》作者


Mads Torgersen
微软　C#编程语言Program Manager

精彩案例 先睹为快


《浅谈前端交互的基础设施的建设》
Speaker：程劭非（寒冬）
淘宝 高级技术专家


《深度学习在广告投放中的应用》
Speaker：苏函晶
腾讯　MIG开放平台部商业化模型负责人


《Netflix的可靠性工程》
Speaker：Katharina Probst
Netflix　工程总监


《基于Kotlin协程实现异步编程》
Speaker：Roman Elizarov
JetBrains　工程师，Kotlin开发团队成员


《人工智能技术在金融行业应用探索》
Speaker：李闯
中国金融认证中心　机器学习实验室高级研究员


《分布式计算系统的性能优化》
Speaker：张建伟
百度　技术经理


《Prometheus监控系统最佳实践与常见陷阱》
Speaker：Julius Volz
Prometheus　监控系统创始人之一


《深入Apache Spark流计算引擎：Structured Streaming》
Speaker：朱诗雄
Databricks 软件开发工程师，Apache Spark PMC和Committer


《Google Translate助力自然语言理解》
Speaker：田野
Google研究院　机器学习工程师


《GraalVM及其生态系统》
Speaker：郑雨迪
Oracle Labs　高级研究员


《深入浅出概率编程》
Speaker：程显峰
火币网　CTO


《万物皆向量——双十一淘宝首页个性化推荐背后的秘密》
Speaker：黄丕培（灵培）
阿里巴巴　高级算法专家


《跳一跳的前世今生——小游戏开发经验分享》
Speaker：徐嘉键
腾讯　跳一跳游戏技术负责人


《使用开源分布式存储系统Alluxio来有效的分离计算与存储》
Speaker：富羽鹏
Alluxio　创始成员&资深架构师

8折 优惠报名中，立减1360元
团　购　享　受　更　多　优　惠

访问官网获取更多前沿技术趋势　2018.qconbeijing.com

如有任何问题，欢迎咨询
电话：15110019061，微信：qcon-0410

InfoQ 中文站　2018迷你书

架构师 月刊
2018年1月

本期主要内容：Werner Vogels 脑中的未来世界 @AWS re:Invent 2017；阿里巴巴正式开源其自研容器技术 Pouch；从 CQS 到 CQRS；为什么说 Service Mesh 是微服务发展到今天的必然产物？中小型研发团队架构实践：电商如何做企业总体架构？

深度学习器：TensorFlow程序设计

本书详细介绍了 TensorFlow 程序设计中的几个关键技术。

AI前线特刊：AI领域2017进展总结

AI 前线在 2018 年之初为各位读者奉上这样一本迷你书，涵盖了来自全球 AI 和大数据领域技术专家的年终总结与趋势解读，同时还有世界知名技术大厂的年终技术总结与趋势预测。

架构师特刊 范式大学

构建商业 AI 能力的五大要素；判别 AI 改造企业的 70 个指标；用最小成本 验证 AI 可行性；企业技术人员如何向人工智能靠拢？人工智能的下一个技术风口与商业风口。