项目总结和架构师技能提升

项目总结

思维导图版

https://www.processon.com/view/link/637f182e5653bb3a8420f9af

文字版

电商网站架构设计和需求分析

- 电商业务概述
- 设计电商系统的核心流程
- 根据流程来划分模块
- 电商系统设计核心要点
- 面试题:你的项目为什么要从单体变化到微服务
  - 单体服务的优点缺点
  - 微服务架构的优缺点
- 五期商城项目的架构
  - 项目结构
    - 代码结构
    - Database
    - 课程关键 Table
  - 项目环境

- 代码管理和项目发布
- 中间件和基础设施
- 项目中将学习的技术问题

- 项目架构图
- 项目学习必备知识清单

- 需求分析指引
  - 一、需求分析定义
  - 二、软件需求分析目标
  - 三、软件需求分析原则
  - 四、软件需求分析内容
  - 五、软件需求分析过程
  - 六、软件需求评估方法
  - 七、需求分析优先级的方法
  - 八、如何确定软件需求
  - 九、整理需求
  - 十、需求不明确带来的影响
  - 十一、需求分析推荐书籍

快速梳理电商后台管理系统核心功能

- 电商管理后台 MyBatis-plus 多数据源切换实战
- 自定义实现基于 MyBatis-plus 的逆向工程

前端架构串讲及电商支撑服务

- 电商前端重要功能
- 电商登录用户状态同步机制
- 分布式任务调度系统 XXL-JOB 部署及使用

电商项目微服务架构拆分实战

- 电商项目运行环境介绍
- 微服务拆分时机和策略分析
- 电商项目微服务技术栈选型以及常用组件的接入
- 微服务全链路灰度解决方案实战

自动化部署方案设计及日志收集方案

- Devops 与 CI\CD 的优化方案
- 基于 Jenkins 和 GitLab 部署基础自动化运维体系
- 熟悉 SpringBoot 常见的 Maven 打包部署方式。
- 实战部署基于 FileBeat+Logstash+ES 实现的典型分布式日志收集体系

电商项目分布式 ID 服务实战

- 唯一 ID 需求分析
- UUID

- Snowflake 算法实现
- 数据库生成
- 美团 Leaf 实现
  - Leaf-segment 数据库方案
  - 双 buffer 优化
  - Leaf-snowflake 方案-解决时钟问题
- tulingmall-unqid 实现

微服务网关整合 OAuth2.0 授权中心

- 电商项目统一认证业务场景分析与演示
- 电商项目搭建授权中心设计思路和实战
- 接入网关服务实现统一认证
- 未来授权服务 Spring Authorization Server 实战

订单系统的设计与海量数据处理实战

- 订单系统业务分析
- 订单系统的核心功能和数据表
- 订单重复下单问题
- 订单 ABA 问题和解决
- 读写分离与分库分表
- 商城订单服务的实现

- MySQL 应对海量数据
-
  - 归档历史数据
  - 商城历史订单服务的实现

订单接入支付宝流程实战

- 熟悉支付宝支付能力接入方式。
- 实现支付宝沙箱环境应用接入。
- 电商项目订单系统接入支付宝，实现第三方支付。
- 使用 RocketMQ 优化电商项目订单支付流程

下单链路分布式事务 Seata&MQ 可靠消息实战

- 分布式事务在电商项目中的应用
- 常见的分布式事务解决方案选型
- Seata 实现下单冻结库存场景的分布式事务
- 分库分表场景下 Seata 如何使用
- 基于 MQ 可靠消息如何实现分布式事务
- Rocketmq 事务消息实战

电商项目高并发缓存实战

- 缓存为什么不仅仅是 Redis
  - 客户端缓存

        - 网络缓存
        - 服务端缓存
    - 缓存的数据一致性
        - 工程实践
        - 缓存更新的设计模式
    - 商城项目的缓存实践
        - 基础实现
        - 缓存预热
        - 数据一致性
        - 双缓存
    - 大厂如何利用 Redis 应对海量和高可用、高并发

电商项目数据高可用架构设计与实现

- 大型企业如何实现 MySQL 到 Redis 的同步
- 实现跨系统的数据同步
    - 缓存不命中
    - 使用 Binlog 实时更新 Redis 缓存
    - Canal 详解
- 5 步法实现不停机更换数据库
- 安全实现数据备份与恢复

高并发秒杀系统的设计与实现

- 秒杀系统的挑战
- 秒杀系统设计
- 常见的秒杀系统架构
- 商城秒杀系统架构
  - 秒杀系统的隔离
  - OpenResty
  - 秒杀系统的商详页
    - 静态化
    - 商详页的库存获取
- 秒杀的前期流量管控
  - 预约系统设计
  - 预约系统优化
- 秒杀的事中流量管控
  - 削峰
    - 验证码和问答题
    - 消息队列
  - 限流
    - Nginx 限流
    - 应用/服务层限流

- 分层过滤

- 限购、秒杀的库存与降级、热点
  - 限购
  - 库存扣减
    - 数据库方案
    - 分布式锁方案
    - 商城库存扣减的实现
    - 服务降级
- 防刷、 风控和容灾处理

核心订单链路兜底方案之限流&熔断降级实战

- 高并发场景下如何保证系统稳定运行
- 微服务网关常见限流方案分析
- 网关整合 Sentinel 之 Route&API 维度限流实战
- 生产环境接入 Sentinel 规则持久化实战
- 秒杀链路接入 Sentinel 限流实战
- 下单链路接入 Sentinel 熔断降级实战
- 电商系统自适应保护方案实战

大型网站高并发的读、写实践

- 高并发读写问题场景
- 高并发读

        - 策略 1:加缓存/读副本
            - 方案 1：本地缓存或集中式缓存
            - 方案 2：MySQL 的 Master/Slave
            - 方案 3: CDN/静态文件加速/动静分离
        - 策略 2:并发读
            - 方案 1：异步 RPC
            - 方案 2： 冗余请求
        - 策略 3:重写轻读
            - 方案 1： 微博 Feeds 流的实现
            - 方案 2： 多表的关联查询： 宽表与搜索引擎
    - 高并发写
        - 策略 1：数据分片
            - 数据库的分库分表、Redis Cluster、ES 的分布式索引
        - 策略 2：异步化
            - 案例 1：短信验证码注册或登录
            - 案例 2：电商的订单系统拆单
            - 案例 3：广告计费系统
            - 案例 4：写内存 + Write-Ahead 日志

        - 策略 3：批量写
            - 案例 1：广告计费系统的合并扣费
            - 案例 2：小事务合并机制
    - RockDB 详解

海量数据存储与查询最佳实践

- 海量数据存储的技术选型
    - 技术选型时应该考虑哪些因素
    - 在线业务系统如何选择存储产品
    - 常见问题：如何存储前端埋点之类的海量数据
    - 转变思想：根据查询选择存储系统
    - 京东的物流数据存储分析
- 商品系统的存储架构设计
    - 商品系统需要保存哪些数据
    - 如何存储商品的基本信息
    - 如何保存商品参数
    - 使用对象存储保存商品图片和视频
    - 商品介绍静态化
- 购物车系统的存储架构实践
    - 如何设计“暂存购物车”的存储

- 用户购物车的存储

业务架构之道——架构师技能提升

技术领域的知识往往是“硬”知识、“硬”技能，1 就 1，2 就是 2。但作为一个架构师仅仅掌握这些“硬”知识、“硬”技能还不够，“**业务驱动技术**”，但是业务领域更多是很多“软”性的、抽象的技能。一旦一个东西呈“软”性的，大家虽然知道这类东西存在，但又难于表述。

业务架构呈现很明显的这个特征，当问一个程序员或架构师什么是业务架构的时候，他们通常都知道一个大概，但好像又难于描述，就像是“只能意会不能言传”。我们本章就来看看业务和技术的融合。

业务意识

互联网时代有个岗位——产品经理。而在互联网大规模发展起来之前，软件行业通常称为“需求分析师”。作为一个技术人员，通常认为需求分析是产品经理或需求分析师的职责，自己只需做好技术就行。

但技术不是无源之水，一旦离开业务纯粹地谈技术，就失去了驱动技术发展的根本要素。另一方面，研发部门的人力资源和时间资源是有限的，而业务需求是无限的，要用有限的资源应对无限的需求，必然存在需求的取舍问题，而这种取舍往往也会影响系统的架构设计。

对于一个技术人员，不需要像产品经理或需求分析师一样对需求了如指掌，但具有良好的业务意识却是做业务架构的基本条件。

什么叫业务意识？

(1）需求来自何处？如果是一个 C 端的互联网产品，需求可能来自用户反馈或用户调研；如果是一个 B 端的项目，需求可能直接来自客户；还有可能需求来自对业务数据的分析挖掘，从数据中发现了某些问题需要解决；更有可能，需求来自老板的决定。

有时，需求来自何处、技术为谁而做，往往和公司的基因、盈利模式紧密挂钩，公司本身决定了需求从什么地方来。

(2)真需求还是伪需求。技术人员经常会听到要开发一个某某功能、一个某某系统，但“功能”和“系统”并不是需求。需求是要解决的“问题”，而问题一定是系统所要面对的用户问题或客户问题，功能或系统只是解决问题的一种答案而已。

很多原因都会导致伪需求，比如老板的决定、面向 KPI 的需求，这些都比较容易看到。

但是当发生一个事件时，第一个人 A 看到事件的全过程，掌握的信息量按 100 分计算。当 A 向 B 描述事件过程时，受其记忆力、表达力、主观故意等原因的限制，最多能把整个事件信息的 90%描述出来，A 讲出去的只是 90 分。而听者 B 同样受到注意力、理解力、主观故意等因素的影响，是不可能完全理解 A 所陈述的所有信息的，不同程度地产生丢失忽略或误解情况，则事件真相经过 A 讲述给 B 的传递过程，信息量可能只剩下 85 分了。

依此类推，当 B 再向 C 转述时，与 A 向 B 传递的一样，信息会不同程度地再次丢失或误解，事件的真实信息可能只剩 80 分了……在这个过程中，真实的信息量衰减得越来越厉害，每增加一个中间环节，被加入的误解信息量越来越多。这个问题是在沟通、传播过程中最普遍的。

一个需求被用户或客户提出来，可能经过总监、组长、产品经理层层传导，等传到了技术人员，可能已经不是最初的需求，最后做出来的东西往往不是对方真正想要的。

所以，作为一个技术人员，当从产品经理接到需求的时候，一定要回溯，明确需求是在什么背景下提出的，究竟要解决用户的什么问题。

(3)产品手段 vs.技术手段。对于业务问题，产品经理会考虑用产品的手段解决，技术人员会考虑用技术的手段解决。用哪个更好？

有一个搜索引擎的输入框增长的案例：最初做搜索引擎的时候，研究人员发现，如果用户搜索时多输入几个字，搜索结果就会准确得多。那么，有没有什么方法能提示用户多输入几个字呢？有人想到能否做一个智慧化的问答系统，引导使用者提出较长的问题呢？

但是，这个方案的可行性会遇到许多挑战。也有人想到，能否主动告诉用户请尽量输入更长的句子，或根据使用者的输入词主动建议更长的搜索词呢？但是这样似乎又会干扰用户。最终有一位技术人员想到了一个最简单也最有效的点子：把搜索框的长度延长一半。结果，当用户看到搜索框比较长时，输入更多的字词的可能性更大。这就是一个典型的用产品手段解决用户问题的案例。

再比如某个 UGC 社区首页的个性化推荐，用户可以一页页地往下翻，翻几百页、几千页,但实际上很少有用户有这个精力。对于一个推荐系统，可能只需要保证前 1000 条数据是精准推荐的，后面的选择自然排序。

这相当于把一个无限数据规模的排序问题缩小为固定有限长度的排序问题。这也是用产品手段解决技术问题的案例。

再比如很多的后台报表统计类系统，为了降低技术的实现难度，采用分钟级、甚至小时级或天级别的报表，而不是实时统计。这也是典型的通过产品的设计解决技术问题的案例。

(4）需求的优先级。人力资源和时间资源是有限的，一个需求是很重要，还是不那么重要;是很紧急，还是可以从缓，需要有清晰的认识。系统的架构是被需求驱动着一步步迭代、升级

什么叫作一个“业务”

既然谈“业务架构”，首先要从一个技术人的视角来看，什么可以被称为一个“业务”"。比如：

案例 1：美团点评公司，做团购、外卖、餐饮、生鲜、休闲娱乐、丽人、结婚、亲子、配送、酒店旅游、出行……请问，这家公司有几个“业务”？

如果以 2019 年公布的最新的组织架构来看，这家公司主要有四大业务：

1）到店（包括餐饮、团购、休闲娱乐、丽人、结婚、亲子……)。

2）大零售（外卖、配送、生鲜)。

3）酒店旅游。

4）出行（打车)。

但如果以 2018 年前的组织架构来看，这家公司有三大业务：

1）餐饮类（团购、外卖)。

2）综合类(休闲娱乐、丽人、结婚、亲子……)。

3）酒店旅游。

为什么要这样的变化呢？

案例 2：某电商平台有 B2C、C2B、C2C、海淘和海外。

这是五个业务，还是一个业务，或是三个业务?

第一种分法，认为这是一个业务，产品、技术、运营各一套技术架构，支撑不同的玩法而已。

第二种分法，认为是三种业务，国内、海淘、海外三个团队，只是账号体系、技术基础实施共用而已。

第三种分法，认为是五个不同的业务，五个团队各做各的。同第二种一样，某些基础设施共用。

案例 3:把案例 2 进一步细化

电商的“供应链”是否是一个业务？前端的“搜索”，是否是一个“业务”？

从上面的案例可以看出，一个内容是否算作一个“业务”往往与公司的长期战略、盈利模式、发展阶段、组织架构密切相关，并没有一个标准的划分方式。但抛开这些差异性，一个内容能称为一个“业务”，往往具有一个特点，就是“闭环”。

什么叫闭环？

• 团队闭环：有自己的产品、技术、运营和销售，联合作战。

• 产品闭环：从内容的生成到消费，整条链路把控。

• 商业闭环：具备了自负盈亏的能力(即使短期没有，长期也是向这个发展方向）

• 纵向闭环：某个垂直领域涵盖从前到后。

• 横向闭环：平台模式，横向覆盖某个横切面。

同时闭环可大可小。

• 小闭环:：一个部门内部的某项内容有独立的产品、技术、运营团队，独立运作。

• 大闭环：事业群、事业部级别，公司高层战略来决定的。

• 更大的闭环：产业上下游，构建完整的生态体系。

“业务架构”的双重含义

前面的案例讨论的“业务”，其实对应的是“业务架构”一词的字面意思，也就是“业务的架构”。这通常关乎大的战略，主要是从商业角度去看，公司高层领导决定的。

但对于技术人员来说，讨论业务架构的时候则是另外一个意思:“支撑业务的技术架构”。注意，这里的落脚点在技术上，是从技术的视角看业务应该如何划分。

业务架构既关乎组织架构，也关乎技术架构，业务架构、组织架构和技术架构三者之间的关系是怎么样的呢？

(1）先说业务架构的第一重意思。从理论上讲，合理的团队的组织架构应该是根据业务的发展来决定的。不同的公司在不同的发展阶段会根据业务的发展情况，将壮大的业务拆分，萎缩的业务合并。拆分到一定的时候又合并，组织架构相应地跟着调整，相应的技术团队跟着整合，技术架构自然也会跟着变化。

这种变化规律在半个世纪前就已经被提出，也就是“康威定律”：一个组织产生的系统设计等同于组织之内、组织之间的沟通结构。这也意味着：如果组织架构不合理，则会约束业务架构，也约束技术架构的发展。而组织架构的调整涉及部门利益的重新分配，所以往往也只能由高层来推动。

(2）业务架构的第二重意思。“支持业务的技术架构”,业务架构和技术架构会相互作用、相互影响。举个例子，对于广告业务，如果认为 CPC（效果广告）、CPM（展示广告）、CPT（按时间段计费）是三个业务，

可能会各自设计三套技术架构方案，并让三个团队去做；但如果认为是一个业务，会思考这三者之间哪些是共用的，哪些又是个性化的，尽可能把三者通过一个技术架构支撑，让一个团队去做。

这种技术的思考会反过来影响业务，重新思考团队的组织方式，团队的组织方式又会变,接下来又会影响业务的发展方式。

既然“业务架构”指的是“支撑业务的技术架构”。那既牵扯业务，又牵扯技术，二者究竟如何区分?又如何融合?

“业务架构”与“技术架构”的区分

之所以要谈“分”，是因为经常遇到的情况是：明明是业务问题，却想用技术手段解决；明明是技术问题，由于技术无法实现，反过来将就业务；可能既不是业务问题，也不是技术问题,而是组织架构问题，是部门利益问题，是公司的盈利模式问题。

一般技术架构要关注的一系列问题:

1）你的系统是在线系统还是离线系统？

2）如果是在线系统，需要拆分成多少个服务？每个服务的 QPS 是多少，需要部署多少台机器?

3）运行方式是多线程，多进程？还是线程同步机制，进程同步机制？

4）如果是离线系统？有多少个后台任务？任务是单机，还是集群调度？

5）对应的数据库的表设计？是否有分库分表？

6）数据库的高可用？

7）服务接口的 API 设计？

8）是否用了缓存？缓存数据结构是怎样的？缓存数据更新机制？缓存的高可用？

9）是否用了消息中间件？消息的消费策略？

10）是否有限流、降级、熔断措施？

11）监控、报警机制？

12）服务之间的数据一致性如何保证？比如分布式事务。……

通过上面一系列问题可以看到，技术架构涉及的都是“系统"“服务”“接口”“表”“机器”“缓存”这样技术性很强的词语。这些是开发人员直接可以通过写代码实现的，很务实，没有虚的内容在里面。

把上面这些内容梳理一下，归类后就变成了我们经常挂在嘴边的各种架构词汇：

1）物理架构（物理部署图);

2）运行架构（多线程、多进程);

3）数据架构（存储系统的选择、数据库表的 schema);

4）应用架构（系统的微服务划分);……

从具体技术到抽象技术，再到业务，所用的词汇会越来越抽象，越来越偏向各自的领域，则在沟通和表达过程中，产生歧义的概率就越大。

实践中只有时刻意识到我们面对的是业务问题，还是技术问题，或是其他的更高层次的问题，才可能在一个正确的层面上去解决。

通过分析，我们知道了“技术架构”究竟代表什么，这也为我们提供了一个参照系。“业务架构”不是“技术架构”，是“技术架构”外面的东西。

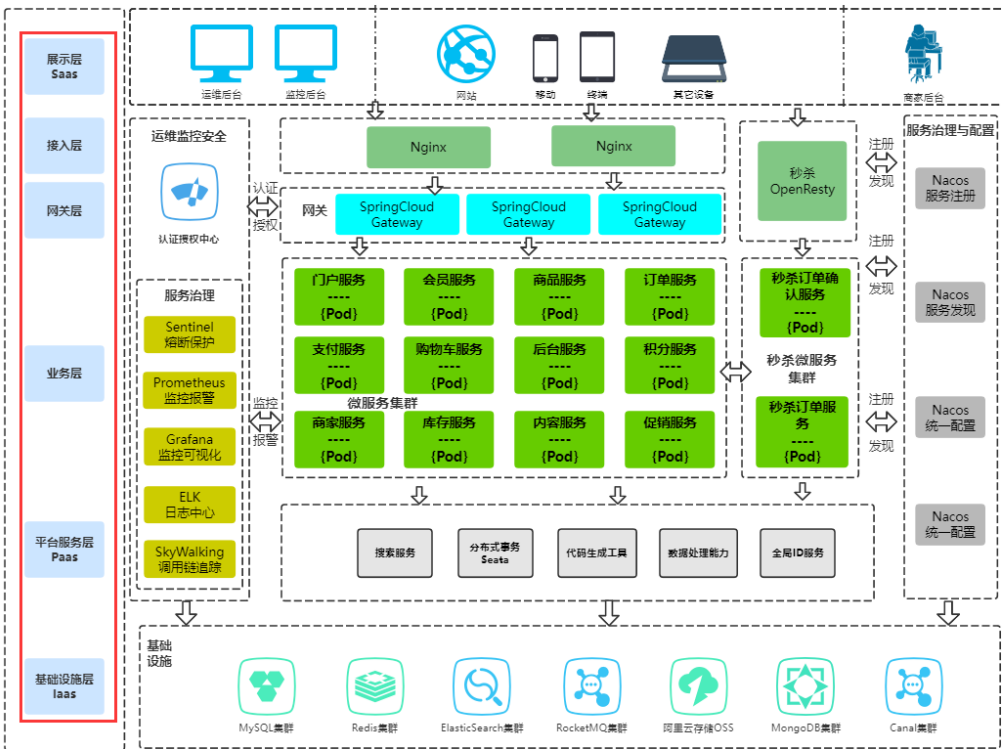
业务架构思维

警惕“伪”分层

对于分层架构，我们都不陌生。无论业务架构，还是技术架构；无论做 C 端业务，还是 B 端业务；无论做服务器，还是客户端，所有人都会用到。

因为分层其实不光是一个技术词汇，而是一个通用的思维方式。

比如我们商城系统的架构分层图：



但这只是一个图纸上的示意图，实际的代码、系统是否能按分层架构严格执行呢？如果把所有系统之间的调用关系都梳理出来把依赖关系图画出来，不一定会是这样一个分层结构，很可能是一个网状结构了。产生了伪”分层。

比如：

(1）底层调用上层。比如某个基础服务调用上层的业务服务，怎么解决呢？

办法 1:要思考业务逻辑是否放错了地方?或者业务逻辑是否需要一分为二，一部分放在业务服务，一部分放在基础服务。也就避免了底层调用上层。

办法 2:DIP（依赖反转)。底层定义接口，上层实现，而不是底层直接调用上层。

(2）同层之间，服务之间各种双向调用。比如业务服务 1、业务服务 2、业务服务 3 之间都是双向调用。

这时就要思考，业务服务 1、业务服务 2、业务服务 3 之间的职责分配是否有问题，出现了服务之间的紧耦合？

是否应该有一个公共的服务，让公共服务和业务服务 1、业务服务 2、业务服务 3 交互，而三个业务服务之间相互独立?

(3）层之间没有隔离，参数层层透传，一直穿透到最底层，导致底层系统经常变动。例如，App 一直发版本，为了实现兼容，服务器端会根据不同的版本调用不同的函数。

比如客户端要支撑各式各样的业务，因此肯定有类似 businessType 用于区分不同业务，或者说区分同一个业务的不同业务场景。businessType 字段一旦透传到最底层的基础服务，在基础服务里面都能看到 if busessType = XXX 这样的代码，就是典型的上层业务多样性透传到了最底层。

这种情况下，虽然是严格分了层，层层调用，但“底层服务”已经不是底层服务，因为每一次业务变动都会导致从上到下跟着一起改。

(4）聚合层特别多，为了满足客户端需求，各种拼装。遇到这种情况，往往意味着业务服务层太薄，纯粹从技术角度拆分了业务。而不是从业务角度让服务成为一个完整的闭环，或者说一个领域。

一个优秀的分层架构应该具有的典型特征如下。

1)越底层的系统越单一、越简单、越固化；越上层的系统花样越多、越容易变化。要做到这一点，需要层与层之间有很好的隔离和抽象。

2）做到了上面一点，则要做到层与层之间严格地遵守上层调用下层的准则。

边界思维

在所有的架构思维模式中，如果说最终只能留下一种思维模式，那就是边界思维。腾讯公司前 CTO 张志东曾说过“优雅的接口，龌龊的实现”，可以说是边界思维最好的昵释。

在技术领域,“封装”“面向接口的编程”等技术，也都是边界思维的体现。只要一个系统对外的接口是简洁、优雅的，即使系统内部混乱，也不会影响到外界其他系统。相当于把混乱的逻辑约束在一个小范围内，而不会扩散到所有系统。

边界思维是一种通用的思维方式，从小到大来，边界思维在不同层面的均有体现。

1.对象层面（SOLID 原则)

在面向对象的五大原则中，第一个原则 S 就是单一职责原则（The Single ResponsibilityPrinciple)。通俗地讲，就是一个函数、一个类、一个模块只做一件事。不要把不同的职责杂糅在一起,这就是边界思维的一种体现。

2．接口层面

对于开发人员来说，做一个系统往往先想到的是如何实现。而利用边界思维，首先想到的不是如何实现，而是把系统当作一个黑盒，看系统对外提供的接口是什么。接口也就是系统的边界。接口定义了系统可以支持什么、不支持什么。所以，接口的设计往往比接口的实现更重要！站在使用者的角度来看，并不在意接口如何实现，而更在意接口的定义是否清晰，使用是否方便。具体来说，就是：接口的输入、输出参数分别是什么？哪些参数可选，哪些必选？如果输入参数很多，为什么不是分成多个接口？设计策略是什么？接口是否幂等？各种异常场景接口的返回结果都是什么?

3.产品层面

除了技术，产品同样需要有边界思维。对于产品，常说的一句话是:内部实现很复杂，用户界面很简单。把复杂留给自己，把简单留给用户！尤其现在的 AI 产品，更是把这句话发挥到了极致。AI 算法本身很复杂，但对用户来说，使用却越来越“傻瓜化”，以前还有图形界面，现在直接对着系统说句话，它就明白了。

4．组织架构层面

组织的各个部门之间如果没有非常清晰的边界，就会导致该自己做的事情不做，互相推诿、踢皮球；不该自己做的事情抢着做，你争我夺。最后整个体系权责不分，做事效率低下，还容易产生各种问题。

回到系统，不管用哪种分析方法和设计方法，最终必须保证每个系统有清晰的边界，各自分工清晰。无论谁要了解这个系统，他不用看这个系统是怎么实现的，只要看系统的接口，就能知道系统支持什么，不支持什么。

边界思维的重点在于“约束”，是一个“负方法”的思维方式。什么意思呢?比如要看一个开源软件的功能，要看的不是它能做什么，而是它不能做什么!“不能做什么”决定了系统的边界，或者说它的“极限”。

做架构尤其如此，架构强调的不是系统能支持什么，而是系统的“约束”是什么，不管是业务约束，还是技术约束。没有“约束”，就没有架构。一个设计或系统，如果“无所不能”，则意味着“一无所能”。

系统化思维

与边界思维相对应的是系统化思维。哲学中有一句话:事物之间的普遍联系。通俗的说法就是:不能头痛医头，脚痛医脚。头痛的时候，可能原因不在头上，而是身体其他部位出了问题引发的头痛。

比如现在有一个系统 A 和系统 B，应用边界思维，在两个系统之间定义了接口。

但随着业务的发展，发现每次来新的需求，两个系统都要跟着一起改，之间的接口也不够用，要么增加新接口，要么为之前的接口增加新参数。原因可能是在最初设计的时候，接口定义得不合理；也可能是这两个系统的逻辑本身就耦合得很紧。应该把系统 A 和系统 B 重新放在一起整体考虑，然后一致对外提供统一的接口，对内，系统 A 和系统 B 就是一个系统的两个联系紧密的部分。这就是系统化思维的一种体现，把不同的“东西”串在一起考虑，而不是割裂后分开来看。

系统化思维的另外一个体现，就是遇到事情要刨根问底。如果遇到了问题 A，经过分析，是原因 1 导致的;原因 1 又是如何产生的，是原因 2 导致的;原因 2 又是如何产生的，是原因 3 导致的……如此追到最后，直至事物的本质。这点在物理学中叫作“第一性原理”，在哲学上叫作“道”。无论是遇到技术问题，还是产品问题、业务问题，都可以利用这种思维方式。这个倒追的过程会让你探究到事物与事物之间的普遍联系。

比如一个电商系统中商品库存。

站在 C 端来看：用户下单，要减库存；用户发生客退，要加库存；

站在 B 端供应商角度来看：采购，要加库存；退供，要减库存；

站在内部商务和物流人员角度来看：调拨，一个仓库减库存，另一个仓库加库存。

无论 C 端的下单、客退，还是 B 端的采购、退供，还是内部的调拨，都是很复杂的业务流程，对应的是不同的团队开发的不同系统。单独去看每一个业务的每一个系统，都没有问题，但要系统性地把这五大类业务串在一起来看，可能库存的账目是对不齐的。

这就是为什么电商系统里库存往往是单独的微服务，库存不仅仅是一个数字<SKU,数量>，然后对这个数字进行加加减减，逻辑很简单。但实际问题在于库存不是一个简单的数字，而是一个复杂的数据模型，有自己单独的领域。

利益相关者分析

做一个系统与做一个产品一样，首先要了解用户是谁。在架构里面称为利益相关者。下面随便举几个例子，来说明什么是利益相关者:

例 1:哪几类人在用微信？

C 端普通用户;支付收款个人商家;支付收款接入商和开发商;游戏开发商;广告投放商家;订阅号作者;服务号开发者;小程序开发者;

所以说微信是一个平台，一个超级平台、游戏平台、电商平台、广告平台、媒体平台。

例 2:哪几类人在用电商系统?

C 端用户;B 端卖家、供应商;

供应商 ERP（注意:利益相关者不一定是人，也可以是一个外部系统);公司内部人员:包括采购人员、运营人员、客服人员、仓储人员、物流人员、关务人员(如果做海外贸易)、财务人员。

例 3:电商系统里面的支付系统

把例 2 的范围缩小，只讨论里面的支付部分，有哪几类利益相关者呢?

用户;商家;银行;第 3 方支付平台;财务系统;

为什么谈业务架构，要先谈“利益相关者”呢?

(1)利益相关者其实是从“外部”来看系统。把系统当作一个黑盒子，看为哪几类人服务。这其实也就定义了整个系统的边界，定义了整个系统做什么和不做什么。

(2）前面说到一个词“业务”，业务具有“闭环”的特点。而利益相关者就是一个最好的看待业务的视角。

(3)每个利益相关者代表了一个视角。站在 C 端用户的视角和 B 端商家的视角上，对系统有不同的看法。系统很复杂，无法从一个角度全面认识，每个视角都是盲人摸象，只看到系统的一部分。

(4）利益相关者往往也对应了一种业务划分、系统划分。根据不同的利益相关者,可以划分成不同的系统和业务。

所以，当谈到系统的时候，首先要确定的是系统为哪几类人服务，同哪几个外部系统交互，也就确定了系统的边界。

非功能性需求

软件有功能性需求和非功能性需求之分。

其实软件的非功能性需求有很多，不同类型的软件的侧重点也有差别。互联网比较关注的有：

(1）并发性。对于 C 端的系统，大家首先关注的是系统能抵抗多大的流量。说通俗点，是可以承受多少人同时访问。常用的衡量指标是 TPS 和 QPS，平均响应时间/最大响应时间，并发数。

(2）可用性。从服务角度来说，一个服务不可用有两层意思:

• 机器宕机，不能对外提供服务，直接抛错;

• 机器虽然没有宕机,但是超时。这涉及“性能”问题。

(3）一致性。比如数据库的参照完整性、事务、缓存与数据库数据同步、多备份数据一致性问题。

(4）稳定性和可靠性。

“稳定性”指系统没有任何未定义的行为，体现在如下几方面:

• 所有的 if-else 语句里面,没有不处理的分支;

• 所有的 API 调用，每种异常返回值都有处理;

• 考虑内存、磁盘的上限;

• 系统不会时不时冒出一个问题出来;

• 出了问题,有很好的日志监控,能快速修复;

• 系统的 QPS 不会有抖动（除非业务有变化，比如大促），是一条平滑的

• 发布新版本,有回滚方案;

• 新系统上线,灰度平滑切换; •

• 通过压力测试;

可以看出,“稳定性”或“可靠性”的涉及范围很广，很能反映一个工程师的素养。

(5)可维护性。与可维护性密切相关的是“可理解性”,或者说“代码可读性”。体现在如下几个方面:

• 系统架构设计简单，接口简洁，表数据关系清晰;

• 老人离职,新人接手，无须很长时间就能厘清代码逻辑;

• 系统功能不耦合,改一个地方不会牵动全身;

• 系统某些模块即使时间久远,也有人能厘清内部逻辑;

(6）可扩展性。体现在如下几方面:

• 来了一个新需求，伴随一些新功能，可以在现有系统上灵活扩展

• 没有地方写死，可以灵活配置;

• 容易变化的逻辑没有散落在各个系统里面，不需要多个地方跟着一起改。

(7）可重用性。体现在如下几方面:

开发新的需求，旧的功能模块拿过来可以直接用。

通常来讲，对于 C 端应用，会更关注高并发和高可用，然后有的业务（比如支付）对一致可重用性,有了这些特性要求非常高；对于 B 端业务，会更关注系统的可维护性、可扩展性性，系统才能不拖业务的后腿，可以快速地支撑各种各样的复杂业务需求的开发。

学会抽象

什么是抽象？

(1)语言中的抽象。上学的时候，语文老师经常让学生读完一篇文章后概括其“中心思想”。一篇文章 800～1000 字，概括之后，中心思想也就两三句话，这其实就是“抽象”。

在著名的语言学书籍《语言学的邀请》中，讲到了一个理论:语言的抽象阶梯。书中举了这样一个例子:

我们在一个农庄里看到了一头牛，脑海里浮现的是牛的三维立体形象，有它的体形、动作，观察仔细一点甚至可以从牛的眼泪里想象出一些它当时的“心情”。这时，农场的主人走过来跟你介绍说，它叫阿花，这下你脑子里就自动把阿花的名字和刚刚看到的影像给对应起来了。这时你去找你朋友，跟他说，我刚刚看到了阿花。你的朋友估计会莫名其妙，会问哪个“阿花”。这个名字根本无法给你朋友任何信息，只是一个符号而已，但是当你自己说到阿花，却会自动浮现出那头牛的形象。

这时你很着急，因为你的朋友不理解你在说什么，你就会就和他说阿花是一头牛。说到牛,你朋友就会自动浮现出他经验里牛的形象，并且说不就是一头牛吗，有啥稀奇的。这时，你就会进一步向他描述，这是一头母牛，而且自己刚刚和它对视的时候，发现牛的眼睛里有泪光,所以觉得这头牛不一般。所以，同样是牛这个名字，你脑海里感知的牛和朋友脑海里的牛是相当不一样的。如果你朋友没有见过阿花，他就不会把你看到的牛和阿花联系在一起。

所以说，语言只是对现实中我们所注意到的事物特征的一种抽象。每

抽象化的过程，这种过程忽略了现实中事物的许多特征。这种抽象一方面给我们提供了交流的便利，虽然那个朋友听到牛不会想到阿花的泪光，但起码知道牛的其他一些基本特征，比如四足、长角。另一方面，如果没有注意到语言的这种抽象过程，就会误以为我们通过语言认识到了真实的世界，从而容易陷入语言的牢笼里。

继续说阿花，经过你的描述，你的朋友终于相信你，阿花相比于其他牛很不一般。这激发他对于农庄的兴趣，他问到那个农庄还有什么家畜。家畜相比于牛来说又是更高一个层级的抽象，指的不仅是牛了。

这时你和他说，农庄里除了牛，还有一群鸡在扒食，还有几头猪在拱土。你继续说道，特别有意思的是那个农庄并不大，所以鸡、牛、猪都在一起，农场主并没有把它们分开

你朋友说，那你估计这个农庄有多少资产啊?你打算给他多少?你说，除了这些家畜外，农庄里还有两间房，而且自己也特别喜欢阿花，不打算还价了。终于，你买下了这个农庄，阿花包括其他家禽以及房屋，就成了你资产的一部分，而资产则组成了你所拥有的财富。

从阿花、牛、家畜、农庄资产、资产、财富，就组成了一个抽象阶梯，抽象程度越来越高，而其中组成部分的细节特征显示得也越来越少，到后来其真实特性也就完全不提了。

所以抽象的过程也是一个总结、分门别类的过程。

对于人类，无论是沟通，还是理解事物，我们都在不断地做“抽象”，也就是做“简化”,因为我们的大脑和能量不足以装载和处理现实世界如此巨大的信息量。

(2）计算机中的抽象

• 存储的抽象:关系型数据库,表格。

现实世界中的数据各式各样,但到了计算机中,有一种东西叫关系型数据库。通俗地讲，就是一张张的表格，然后表格之间通过主外键关联起来。

这其实就是一种抽象，把现实世界中花样繁多的数据形式进行规整,最终变成了一张张的表格。

• 计算的抽象是顺序、选择、循环。再复杂的算法，逻辑计算到了计算机里面，最终都会变成顺序、选择、循环三种语句。

现实的逻辑很复杂，但计算机里面的逻辑只有三种。

• 面向对象的方法学:父类与子类、继承。在面向对象的方法学里面，提取共性形成父类，提高代码复用性，这是抽象。

• 面向接口的方法学。

把面向对象的方法再往前推进一步，就是所谓的“面向接口的方法学”。接口是交互双方的一种协议，也是对交互细节的一种抽象。

怎么做抽象

(1）分解：找出差异和共性。要做抽象，首先要做的是分解。只有分解，才知道两个事物间的差异和共性。

举个简单的例子：牛和马的区别在哪？共性在哪？

首先要做的，肯定是把牛和马各自分解成很多特征，然后对这些特征逐个比较，看差异和共性在哪。

(2）归纳：造词。找到了共性和差异，把共性的部分总结成一个新的东西，造一个新词来表达“共性”，就是归纳，也是抽象。

所以抽象的过程往往也是一个“造词”的过程。

抽象带来的问题

抽象的好处就是找出共性、简化事物，但抽象也会造成问题:

(1）抽象造成意义模糊。越抽象的东西往往越“虚”，最后就变成“空洞的大话”，华而不实。

不同人对“虚”的东西理解都不一样,大家在沟通时往往不在同一个频道中，牛头不对马嘴。

（2）抽象错误：地基不稳。没有做分解就分析，会把一个非原子性、容易变化的东西抽象出来，作为整个系统的基础。

(3）抽象造成关键特征丢失。把事物的某个重要的关键特征抽象掉了，会导致对事物的认知偏差。

(4）抽象过度。抽象是为了提供灵活性和扩展性。但如果业务在某一方面变化的可能性很小，则可能压根不需要抽象。

抽象是人类思维认知的一个基本能力，在现实生活和各种学科中都会遇到。具体到计算机的软件架构里，就是分析和分解各种概念、实体、系统，然后又造出一些新的概念、框架的过程。

从架构到技术管理

能力模型

我们需要把技术之外的东西也纳入进来,看看一个全面的技术人员的能力模型应该是什么样子。

格局

假如我们去一个陌生的城市旅游，首先需要的是一张“地图”。这张地图定义了城市的“边界”，也定义了城市的所有地方。通过地图，我们会对这个城市有一个“全局的了解”。

而这种“全局的视野”不只架构需要，换作其他职位、其他行业，也同样需要。作为产品经理，需要对产品有全局视野;作为运营人员、市场人员，需要运营、市场相关的全局视野;做技术，需要技术相关的全局视野。

说了这么多，可能还是比较“虚”，到底什么是全局视野？

比如现在负责开发一个新系统，可能需要理解下面这些关系到“大局的问题”:

• 系统的定位是什么？它能创造哪些核心价值？

•开发这个系统的背景是什么？为什么以前不做，现在要做？是因为业务发展到了一定规模？还是开发资源现在有多余，没事可干?

•系统在整个组织架构中处于什么位置？与这个系统关联的其他系统目前处于什么状况？

• 产品经理如何看待这个系统？技术负责人/CTO 如何看得这个系统？

•这个系统的需求处于比较确定清晰的状态，还是有很大的模棱两可的空间，核心点大家有没有想清楚?

• 这个系统所用的技术体系是比较老，还是最新的?

• 对于业界类似的系统，别的公司是如何做的?

上面这些问题并没有标准答案，每个系统都不一样，但是一个有大局观、有“格局”的人，在做一件事情之前，都会问问自己，会对所做的事情有一个“全局把握”，风险如何？挑战在哪？提前都有心理准备。

“格局”是有层次的，国家总理在“国家”的层次思考，公司 CEO 在行业思考，业务线负责人在其负责的“业务”层次思考，技术负责人可能主要在“技术层次”思考,产品负责人在“产品层次”思考，程序员在“代码”层次思考。

不同层次的人聚焦的范围不一样。如果能把自己的“范围”往外扩大一圈，这对自己做“本职工作”会很有益处。而这，就是“格局”。

技术历史

“格局”是从空间的角度看待问题，那么“历史观”就是从时间的角度看待问题。任何一种技术，都不是凭空想象出来的，它一定是要解决某个特定问题而产生的。

这个特定问题一定有它的历史背景：因为之前的技术在解决这个特定问题时不够好或有其他副作用，所以才发明了这个新技术。所以，看待一个技术或方法论，需要把它放到“历史长河”中去，看它在历史中处于什么位置。

抽象能力

“抽象能力”已经有所说明，这个词听起来很“虚”。可作为架构师，就是需要这种“务虚思维”。

抽象也是一个“层次”结构，从最底层到最上层，在不同工作阶段需要在不同的抽象层级中思考。

很多写代码的人都习惯“自底向上”的思维方式。当讨论需求的时候，他首先想到的是这个需求如何实现，而不是这个需求本身是否合理？这个需求与其他需求有何关联关系。

这种过早考虑“实现细节”的思考方式会让我们“只见树木，不见森林”，最终淹没在错综复杂的各种细节之中，因层次混乱，往往把握不住重点。

同样是前面的例子，假如做一个新的系统，从“抽象”到“细节”，应该考虑:

- 每个需求的合理性?
- 这个系统的领域模型是什么样的?
- 这个系统应该在旧的上面改造?还是应该另起炉灶?
- 这个系统可以分成几期，分期实施?
- 这个系统要拆分成几个子系统? • 每个子系统又要拆分出多少个模块?
- 系统的表设计?API 接口设计?系统之间的消息传输如何实现?

从上到下，是一个逐级细化的过程，并目进入到下一级之后，上一级可能又会退回去修改。

深入思考的能力

深入思考能力主要指“技术”的深度。关于“广度”，在“格局”层面已经包含。

“深度”并不是要在所有领域都很深。人一生的精力是有限的，我们不可能深入掌握所有技术领域，但至少需要对于一个领域非常精通。

拥有这种深度并不代表要胜任当前的工作必须达到这种要求，而是要养成这种“深入思考的习惯”，当我们在思考其他问题时也会带着这种“习惯”。

由于技术一直在更新换代，当面对一个新技术的时候，如果具有深入思考的能力和习惯,对新技术的理解往往也会更透彻。

同时，“深度”会让你对“技术风险”有更加清醒的认知。在做一个项目的时候，可能会前发现里面潜在的“坑”，而不是等到实施了才发现问题，被动解决。

落地能力

落地能力就是通常所说的“执行力”,取决于很多因素。首先,架构方案必须是能够落地的而不是只能停留在 PPT 上面。对于一个技术管理者，需要跟踪从架构设计到架构落地的完整程，在落地过程中发现问题，实时修正，才可能真正做到“理论”与“实际”的统一。

然后是项目管理的问题，需要对项目中任何可能存在的不确定因素、阻碍项目进度的因素进行把控。

在这些不确定因素里面，很多是“人”的因素，而不是“技术”的因素。再复杂的系统,都是由“人”开发出来的。而人多了之后，与“人”相关的问题会自然产生:沟通不充分、组织混乱、职责不清。

作为一个技术管理者，需要意识到这些问题的存在，然后在各种障碍之下找到一条路线,去达成业务和团队目标。

影响力的塑造

初入职场时，大家都很青涩，能力也相差无几。然而几年过后，有的人升成了负责人，管理更大的事情;有的人还在原地踏步，没有多大提升。影响一个人职业生涯的因素有很多，有公司业务、运气、人/团队、个人和领导性格合拍等。这些方面多数超出了一个技术人的掌控范围。

作为一个技术人，应如何务实地塑造的影响力呢？。运气是很难掌控，但是塑造影响力是任何一个人只要用心做都可以做到的。

关键时候能顶上

在一个组织开展业务的过程中,必然会有一些比较“关键”的事件:

- 某个问题困扰了团队一个星期，也没有人能搞定;
- 某个成员离职，他负责的模块没有人接手;
- 某位用户反映的问题像牛皮癣一样，总是时不时发生,无法根治;
- 某个需求发生工期延误，到了快上线的时候却上不了;

如此种种，有的人的解决办法是“能避开就避开”，有的人解决办法是主动迎上、“死磕”,不解决誓不罢休……

打工思维和老板思维

如果是打工思维，安排的事干完了其他一概不管。只管好自己的一亩三分地，技术做完了，产品、运营、业务发展一概不关心。产品体验好不好，业务发展前景如何，与自己无关。

如果是老板思维，会想:

- 这个产品的价值究竟在哪?
- 这个产品有什么问题，如何改进?
- 团队的协作流程有什么问题，如何改进?
- 技术架构有什么问题，如何改进?
- 某些用户投诉一直没解决，如何处理?

空杯心态

术业有专攻。水平再高的人，也只是在某一个领域很强换一个其他的领域，可能什么都不懂。

技术、产品、运营、测试、运维……每个领域都有自己的门道。单说技术，也有前端、后端、架构、算法、数据……每个子领域也都有很深的门道。

在任何时候，我们需要意识到自己的“无知”。只有意识到自己是有“局限的”，才可能不断去听取别人的意见，不断改进自己的工作方法,提升自己的专业能力和视野。否则就会一直待在自己的舒适区里,刚愎自用。

持续改进

世界上从来没有能做到 100 分的事情。产品也好，业务流程也好,技术架构也好,项目管理也好，运营也好……只要想“鸡蛋挑骨头”，总可以找出要改进的地方。

所以要有“批判性思考”的习惯。不能觉得“差不多”就可以，要追求极致，其实有很多事情要做。

在考试时会体会到这样一个道理:从不及格到 60 分,很容易;从 60 分做到 80 分，难一点;从 80 分做到 95 分，很难;从 95 分到 100 分，每增加 1 分，难上加难。

做事情和考试一样，有的人选择做很多事情，但每个事情都只是及格;有的人选择做一个事情，不断向 100 分靠近。

建言献策

接上面的问题，如果有“批判性思考”的能力，能看到一个组织存在的各种问题，并想出应对的解放办法。然后多和同事、领导沟通这些事情，无论对于个人成长还是组织，都是一个正向作用。

说了这么多，最后换位思考一下:如果我们看到公司某个同事在关键时候能顶上，做事追求极致，思考总是很全面，对业务的了解总是比其他技术人员要多，总是很虚心地接受意见，时不时地给公司提出自己的建议。是不是觉得这个人很靠谱，觉得这个人适合带团队。

团队能力提升

在“修身”就是“齐家”，也就是如何带领团队打仗。

不确定性与风险把控

技术管理的首要任务是项目管理。就是如何带领一个团队完成一次次的产品迭代，一个个的项目开发。这里涉及的东西很多也很复杂，包括研发、测试、运维、产品、项目管理、数据分析……不同类型的项目、不同的公司文化，在这件事情的做法上都会有差异。

但无论这些差异如何，对于项目管理，有一个关键问题要面对:“不确定性”问题。

从人的认知来讲，做任何事情，思路都是从一个“朦胧”到逐步“清晰”的过程，项目的进展也是一个从思路、到方案、到落地的细化过程。

在这个过程中，不可避免地存在各种“灰度”，或者说“不确定性”。而项目管理就是要提前防范各种“不确定性”，并采取相应措施，让整个团队、项目克服重重干扰，成功到达终点。

有哪些“不确定性”呢?

需求的不确定性

在做产品或项目时，产品经理、老板或其他相关人员都会有很多“想法”。有些想法很成熟、逻辑严密、很有系统性;而有些想法还不成熟，需要进一步优化;也有些想法，纯粹是头脑风暴，想想而已。

而由于各种外部条件，比如工期的约束、绩效的追逐、领导的压力.…....很可能项目在一个想法没有完全想清楚的情况下就开始了实施。

这就是一个重大的“不确定性”。遇到这种情况，作为技术负责人，需要和产品经理、相关业务方、上级领导等进行广泛的沟通，最终在这个事情上达成“共识”：到底哪些是东西清晰的，我们可以开始做,哪些还需要进一步思考和细化。

技术的不确定性

在做一个新项目时，可能会遇到技术选型的问题，团队中的成员尚未掌握某个框架、开源库或对接的第三方开放 API 等。对于这种情况，必须在项目早期做尽可能多的调研和测试。对于引入的技术框架，哪些特性可以支持、哪些不能支持;对于技术选型，不同方案的优缺点都是什么。

尤其是一些关键的技术细节，如果在前期不调研，等到中后期才发现某个框架无法支持或有问题,可能对整个的技术架构和项目进度造成严重影响。

人员的不确定性

例如一个系统耦合度高，有一个关键模块的开发人员突然离职，新成员又对项目不熟悉、然后慢慢摸熟、上路，等最后项目完工时，离预定工期已经差了一大截。

对于这种情况，一种应对策略就是:不要把项目最核心的部分让一个人开发维护，导致别人无法插手。要分摊风险，在技术的架构设计层面，保证整个系统耦合性不能太高，根据团队成员的水平，每个人都可以承担一块东西。这样即便某个人离职，也有相应的人可以补上。

组织的不确定性

公司越大，业务越复杂，部门越多。随便做一个项目，都可能与好几个业务部门打交道。这些部门可能还在异地，平时只能即时通信，或者远程电话沟通。

对于这种情况，在项目前期必须要做尽可能多的沟通，调研对方提供的业务能力，哪些目前有,哪些还在开发中，哪些还没有开发。

在充分沟通的基础上，和对方敲定排期表，不定期地同步进度，保证对方的进度和自己在一个节奏上。

历史遗留问题

一般当一个新人进入一家公司，除了创业型公司，很少会一上来就能做一个新项目。首先是接手前人留下的老项目，在此基础上进行迭代升级。

有些老项目的技术架构很清晰，文档清楚，业务清楚，还有对项目熟悉的其他同事;也有些遗留项目欠了很多技术债，之前的开发人员也走了，业务人员很多都熟悉。

对于这种情况，需要对项目进行完整的梳理:从产品到技术，找各个接口人沟通，可能经过了两三个月，才对整个系统有了一个全局的把控。

总之，要有“风险把控”的意识，在项目早期努力地想出各种各样的“不确定性”，未雨绸缪。

以价值为中心的管理

作为一个程序员，特别是有技术追求的程序员，经常关注的是：技术水平有多么高，多么复杂，多么酷炫。可当被问到做的东西有什么“价值”时，往往很难说清楚。

技术的价值到底是什么?

我们都知道 GitHub 网站上有很多的开源项目,如何衡量这些项目的价值大小呢?下面有一些考虑因素:

- 以技术复杂度衡量?
- 以代码行数衡量?
- 以技术的先进性衡量?
- 以创新性衡量?

对这些这些项目的关键指标是：有多少人使用了这个开源项目。即使这个项目的代码量很少，功能也很简单，但如果很多组织、个人都在用，说明它就是有巨大价值的。

第一个层次

程序员最熟悉且经常谈论的:系统有多少个业务模块，功能多么强大，采用了多少新技术,采用了某个先进的算法。

第二个层次

在所做的所有工作中，最核心的是采取了哪种措施?最终可能会抽象出一到两个。再追问一下,这一到两个大的技术改进有什么价值，通常都会追问到软件的各个非功能性需求：

(1）可重用性。做了某个 Jar 包、组件、服务，别人不再需要重复造轮子。

(2）可扩展性。来了一个新的需求，只需要配置一下或做很简单的代码开发即可实现，不需要改动很多系统。

(3）可维护性。整个系统解耦做得很好，代码也很整洁。叠加功能或找人接手都比较容易。

(4）高性能。用户体验很好，所有请求都在 100ms 内返回。

(5）高并发。能支持千万到亿级的用户并发访问。

(6）稳定性。系统时不时出问题、宕机，已经把这些问题都解决了，还增加了监控，出问题会立即报警。

(7）高可靠。做了灾备方案，即使某个机器宕机，系统也不受影响。(8）一致性。做到了强一致性,极大地提高了业务体验。

第三个层次

所做的系统为公司带来了什么业务价值:

- 极大提升了用户体验?因此促进了用户增长?提高了用户的活跃度?
- 为公司增加了收入?
- 降低了公司的研发成本?
- 提升了公司的运维效率?
- 为公司开辟了一个新的市场?

第四个层次

站在公司的角度来看，公司是一个在市场经济中追求利润最大化的组织。从这个角度来看，技术也好，产品也好，运营销售也好，最终目的都是要增加公司的利润，即使短期不盈利，长期也是要盈利的。而增加利润，要么“开源”，要么“节流”。所以做的任何东西的价值，基本都会被归结到从这个层次去评判。当然，还有一类是“战略性投入”的项目，虽然它本身不直接挣钱或挣钱很少，但是为了支撑其他盈利的核心业务而能发挥重要作用。

在这四个层次之外，当然还可能会涉及研究性质的技术、技术的普惠性、技术对整个社会的促进作用等,这已经超出了某个业务的技术范畴。目前国内的互联网公司这个方面比较少，国外的 Google 等做的比较好。

以“价值”为中心的管理，会让人避免陷入“无效忙碌”的状态:整个团队天天忙得不亦乐乎，做各种功能，解决各种问题，但回过头来想想，到底有多少东西是有“价值”的?

团队培养

有些技术团队的负责人水平很高，解决问题迅速，但团队成员技术平平，遇到问题都需要负责人亲自上阵解决、累个半死，团队整体非常低效，成员得不到成长，这是典型的缺乏团队培养的思维和意识的案例。

团队的培养包括很多个方面,常见有如下几方面:

技术能力

要培养人，首先得“识人”。只有清楚地知道团队成员的技术能力层次，才能针对不同层次的人设置不同的培训内容。对于技术人员，可以粗略地分为几个层次。

初级：

全部是“面条式”代码，超长类、超长函数，各种晦涩难懂的 if-else。写出来的代码时常出问题，且长时间定位不到问题,对写的功能无法完全掌控

中级：

能熟练地完成各种功能开发，问题少，出现问题能快速解决，代码模块化程度比较高，系统稳定，有业务运维的意识功底深厚，能解决各种开发中的“疑难杂症”

高级：

熟悉业务,能根据业务设计出合理的技术方案

资深：

对技术和业务都有深刻的思考，能对大规模、跨团队的复杂系统进行很好的架构设计。

对每个层级来说，

对于初级人员，需要时常做代码评审，需要读《数据结构与算法》《代码整洁之道》之类的书籍，培养代码思维。

对于中级人员，要培养系统设计能力。

对于高级人员，虽然有系统设计能力，但不够深入,缺乏完善的方法论。

对于资深人员，就解决问题来讲，技术已不是问题，需要发展的是业务能力，成为某个领域的技术领军人物。

对于一个技术团队来说，绝大多数都处于前三个层次，在技术上还有很多的上升空间。在这种情况下，需要在完成业务需求的同时，让团队成员的技术水平不断提高。

可以不定期地举行技术培训、技术分享，在整个团队中形成一个较好的技术的文化氛围,形成一个人带人、人帮人的协作氛围。

独立意识

独立非常的关键，无论对于任何级别的人，都需要独立。所谓独立，就是能掌控事情。交给一个功能开发，能独自把功能做得很好;

交给一个模块，能把模块快速开发完，运行稳定;

交给一个系统，能把系统从设计，到编码、上线，完整地接住;

交给一个项目，能带领一个小团队从需求开始一直到上线完成整个项目，不需要上级操心,按时按质地交付。

做到这一步，意味着团队的每个人在自己所处的层次都是可“托付”的，否则就会频繁出现“补位”。组长干组员的活，经理干组长的活，总监干经理的活，副总裁干总监的活……层层错位,最后整个组织缺乏“顶层设计”。

思维能力

当团队成员遇到问题时，很多人的解决办法是，告诉他问题的解决办法，然后让他把这个问题解决好。如果只做到这一步，则没有起到培养的作用。

对于一个团队来说，解决项目中遇到的问题只达到了及格分数。更需要在解决项目问题之上升华一层,也就是培养思维能力。

思维能力的培养只能靠平时，在面对一个个的问题时，通过一次次的讨论来言传身教。面对问题要刨根问底，深挖问题的背景，掌握解决问题的办法背后的技术原理，研究是否有更好的解决办法……如此一来，思维能力慢慢就会提高。

每个人在职场上工作，都是要养家糊口的。站在团队成员的角度去想下:如果跟着你干，能

力没什么提升，薪资待遇也没什么增长，公司业务前景也看不到，为什么会跟着干呢?

所以，作为一个管理者，要多去赋能他人、成就他人，做项目只是一个过程，最终是要打造一个极具战斗力的团队。有了这样的团队，可以在公司发展的不同阶段自如地切换到不同的业务。