当我们谈论前端架构时，我们到底在谈论什么？

前端架构漫谈

下笔的时候有点后悔开了一个这么大的话题，“架构”这词对于程序员来说是略显神圣的，而且其涵盖范围和意义也是非常广泛，可能每个人每个团队对于它的理解也是不同的，所以今天不深入探讨它本身，而是着重讲讲我们在这方面的一些实践。

架构到底是什么？前端架构又是什么？

我们先看维基百科对软件架构的定义。

> 软件架构是一个系统的草图。软件架构描述的对象是直接构成系统的抽象组件。各个组件之间的连接则明确和相对细致地描述组件之间的通讯。在实现阶段，这些抽象组件被细化为实际的组件，比如具体某个类或者对象。在面向对象领域中，组件之间的连接通常用接口来实现。

传统架构的理解和前端架构的理解略有不同，这个稍后讨论，我们先看看传统意义上对于软件架构的定义。这两句话里可以总结出几个核心名词出来：抽象、解耦、组合，而架构的实际工作，其实就是对这些架构方法和实际场景的梳理把握。

传统意义上的架构师，在实际项目层面，高级些的负责整个系统的整体分解服务分层设计等，而中级的架构师，则负责其中某些模块的“系统分析”。在项目产出上看，分别是架构图 和 系统分析图。架构图体现的是整个大型服务包含的模块，及其运行关系。而系统分析，则是每个服务内部的具体逻辑以及与外部服务接驳的方式。

软件工程师在拿到这些分析之后，就可以用框架将其按照架构师的逻辑来实现，这种工作方式，可以保证大型软件的系统合理解耦，并且合理实现，而对于软件工程师这一环来说，他们无需关心整个系统是如何运行的，只需要按照系统分析设计来实现自己部分的逻辑，将大型软件工程化。

这里就引申出了 “前端架构” 的重点。

其实，前端在整个软件工程中扮演的只是其中的一部分，它的定位较为特殊，不是独立的子系统，却又跨域于整个系统之间，而且最重要的特点是它的内部极为分散。这就造就了我们无法用传统的软件架构来定义“前端架构”这个词汇。实际上，通常所说的“前端架构”在整体软件工程中扮演的是架构之下，代码之上的一个层面，它关注的不是整个系

统的解耦和组合，而是横向的面的开展效率和一致性。（这里排除掉了非常复杂的SPA应用的场景）

很多前端同学其实对于“架构”这个词非常的困惑，我觉得没有必要去斟字酌句，非要将它做成一个固定的职位或者职责。在项目层面，甚至某个页面层面，遇到复杂的交互和数据逻辑，你做一个抽象，你抽离组件，你设计组件的参数和内部状态，这些是不是架构？当然是！它是一个微小系统的内部架构。前端是一个和服务端工程完全相悖的领域，服务端可能从整体来看就是一整个系统，然后抽象，然后分层，最后组合，到了细化的软件层面的时候，就是一些固定的组合逻辑。而对于前端来说，没有整体的概念，一个公司的前端，必然是分散于各个业务各个系统之间的，虽然这些业务系统最后可能也是一个整体，但是对于前端来说，他们就是分散的，反而在每个前端系统内部，又有一套软件工程的思想，像我刚才说的，针对一个页面进行解耦抽象组合。所以，我们总结前端架构的第一层含义：某个系统内部的逻辑抽象和组合。

我们继续观察前端系统的特点，前面也提到了，前端的系统是分散的，这个分散不光是最终实现上的分散，甚至连刚才提到的抽象组合也是分散的，甚至在团队上也是分散的，这样分散的局面，如何变得可控从而让整个前端开发变成一个工程式的工作？这就引申出了前端界最重要的一个词汇：工程化。

实际上，这就是我们要总结的前端架构第二层含义：中立于各个系统又植根于各个系统中的前端工程化。

工程化的核心关注点是什么？可控，效率！事实上接下来讲的一切都是围绕这两点，由“可控”，我们分化出“开发框架”、“开发规范”，甚至是“脚手架”的一些开发约束，还有诸多“开发流程”和“开发工具”的保障，诸如Review机制和Eslint检查、线上错误报警等。由效率，我们分化出“组件库”复用跨业务的组件，“脚手架”将整个流程封装进几个固定的命令，“mock”系统快速模拟数据等。

架构师的工作方式和职责

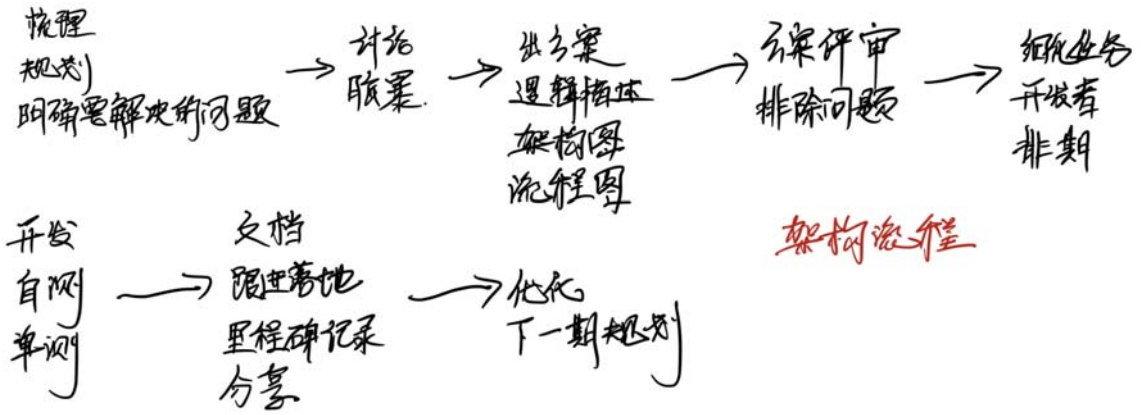
针对上述的对架构的定义，我们来看看具体在日常中如何开展架构工作。通常对于一个成熟的前端团队来说，建立一个专门的架构师小组来说是一个比较合理的结构。这个小组可以是虚拟的，也可以是专门的脱离于项目之外的，特别是业务开发较多时其实工程化需要做的事情还是很多的。

再聊聊架构师的工作方式，首先要抛弃传统的工作思想，之前在业务团队的时候，很多事情都是有专门的人来安排（诸如PM，PD），然后一些技术上的事情则往往草草产出，在成立架构组织后这两个方面都会发生很大的变化。

1. 脱离于业务之外后，对自我规划、自我任务和时间管理的要求变高了，要有非常强的自我管理意识。

2. 技术的产出要有严格的流程，因为你产出的是通用方案，要保证技术方案的质量，这时候需要有一套流程，从发现问题到调研到初步方案到评审确认可行性到详细架

构系统分析到开始编码到推广到文档。之前一整个项目组一起做的流程，现在可能都落到了你自己的头上，看似繁琐但是却是必要的，因为你是一个专业的架构师。



这里我特别想强调下这种思维的转换，程序员通常比较厌恶这种流程上的事情，喜欢自己捣鼓研究敲代码，殊不知其实对于程序员来说最简单的事情其实就是敲代码，如果你一直想敲代码不想做设计/规划/推广，那绝对是在精神上偷懒。而有挑战的事情是什么呢？是设计，设计数据结构，设计组件，设计解决方案。更有挑战的则是将这些设计做完美，做通用，并落地。

所以，做架构绝非只是一直在做有意思的事情，底层的调研，代码的实现只是其中的一部分，一个很重要的自我衡量的标准就是工作时间中最多只有20%是在写代码，而且越少越好，正如上面所说，你的工作是设计落地完善的通用方案，解决特定的问题，而不是玩新技术给团队挖坑。

转换了工作角色之后，我们再来聊聊架构的职责。

1. 宏观上，把控整个团队的技术选型和技术栈，技术发展方向。这看似是一次性的工作，但是却需要持续优化的。例如推动整个公司向Vue或者React转换，推进ReactNative的实施，需要架构师对这些技术栈有深入的了解，能够正确的权衡选型，评估风险，并且给出切实可落地的方案。

2. 各个技术栈的技术体系。业务开发的技术体系，包含脚手架，开发框架，开发规范，组件库，配套工具等。细说起来其实是个挺庞大的事情，每一部分都可以展开成一个话题。例如脚手架，其实是在管理规范或者简化一个业务项目从创建到开发到调试到测试到发布的整个过程（通常会做诸多定制）。例如开发框架，抛开底层的mvvm框架，上层需要做封装，将一些难以理解的概念或者写法繁琐的东西封装起来，同时糅合一些强制的开发规范，还有就是通过框架层规避开发中可能会出问题的风险点（例如数据类型转换之类）。这里提到的每个点可能都是个大工程，而且可能会有好几套体系，例如我司前端的 Vue 体系，客户端的 React Native 体系，服务端界面的 React 体系等。其中有些体系甚至要做到让不是前端的开发去写前端，稍后我会介绍，其封装、规范、工具需要做到简单强大的程度可想而知。

3. 统一环境管理，开发发布流程制定等。制定公司统一的前端开发方式/流程，中间可能会需要一些工具来提高开发效率，推进这些工具如何融入流程被大家接受等。还有前端开发需要的测试、预发、线上环境资源管理的方式、权限等。例如在我们团队中，RN的发布集成上线过程其实非常复杂（为了做版本锁定）但是业务开发

需要做的事情可能就一两步而且非常简单，这种维护方式转化成最终集成结果，把复杂的方案包装成极简的使用方式，这也是架构的重要职责之一。

4. 提供特殊解决方案，例如服务端渲染，可能原理不难，但是需要有专人将其构建成低入侵、高性能、高可靠、统一的服务集群，业务可以非常低成本的接入，并且不需要关系其运维、可用性、性能等问题。其他例如 数据统计打点、跨端调用等，都需要做一些统一的封装处理，让业务方方便使用。

5. 提供一些特殊的工具和系统，例如性能收集，错误收集，mock系统，在线调试，可视化编辑，短链管理等。

6. 提炼业务中除了UI组件之外公用的部分业务，独立维护，我们称之为业务SDK，例如跟金融相关的钱包业务，数据业务，聊天业务，全景业务等，都会作为独立的业务系统服务于其他业务。

搜车前端开发体系

前端开发体系

得益于快速迭代的业务，搜车的前端技术栈统一是Vue2.0，从移动端到PC端，从B端业务到C端业务到营销业务，所以整个体系都围绕Vue2.0展开。大概罗列一下体系中的组成。

- 脚手架，定制自官方，糅合项目管理规范、版本管理规范、配置管理、开发发布流程等。

- 开发框架和规范，目前这块做的比较弱，后续准备做一些深度的封装，做好分层的约束，甚至是请求方式、数据映射等，主要是减少业务项目中的不确定性风险，同时简化开发过程。

- 组件库，包括两端的UI组件库，功能库，还有一些独立于组件库之外的组件（例如图片引用库，会自动根据环境切换webp，并且能优化图片大小，自动实现延迟加载）。在这方面我们投入的精力也比较多，我们有组件准入规范（规范设计师的设计）、组件开发规范（规范组件的开发过程，确保质量和一致性）见附图。

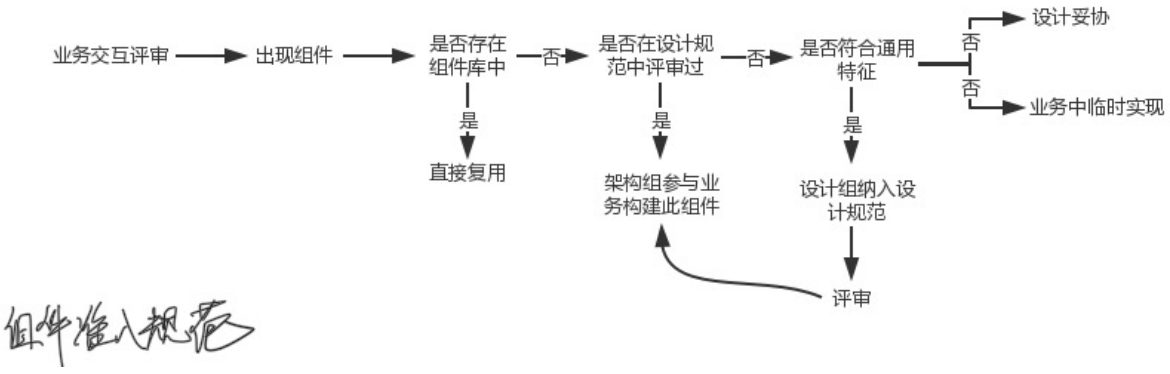
- 开发工具，mock系统（自动从后端接口定义生成随机mock数据，Easy Mock），错误收集系统（所有端如有必要均用Sentry收集），外部链接管理（一个短链服务）。

- 跨端调用，Tower.js + 客户端SDK，可以在所有app内使用各种Native功能，例如照片，视频，用户信息，截图，分享，视图栈管理等等。

- 服务端渲染方案，可以低成本接入任意项目，有专门的监控运维性能优化。

- 静态资源托管服务，基于Nodejs的一个支持部分动态功能的静态资源服务，有一部分服务则会直接通过nginx托管。另外静态资源服务的发布方式，权限管理等相关

的脚本和系统。



React Native 开发体系

团队的RN开发体系最重要的目标是实现让客户端无障碍开发RN业务，事实上，目前整个团队的RN业务大部分是客户端同学在开发，所以在工程化方面更具挑战，需要让不懂React更不懂mobx不擅长flexbox布局，甚至不懂JS的客户端开发快速上手，照葫芦画瓢就可以开发业务，然后也保留所有灵活性。

需要完成这个目标，有几个核心点：

1. 封装跨平台差异，这是RN被人诟病的问题之一，我们对一个UI组件的差异做了自己的封装，封装主要是两部分，一部分是Native组件，自己实现了一些跨平台的轻量级和业务级组件，例如 toast，时间选择，图片选择等；另一部分是在RN JS代码层面封装，例如最常见的navigation的封装，scrollview的封装等。

2. 组件库，除了常见的组件库（主题色，表单，tabbar，tablist等）之外，还有一些将原来的组件复杂度做封装的包装组件，让大家用组件的时候减少困惑和各种兼容问题。

3. 脚手架，这个脚手架比前端的脚手架复杂的多，除了对官方做了深度定制之外（直接改源码），还做了很多开发流程上的功能。深度定制例如拆包逻辑，会分析整个依赖树，剔除公用代码。开发流程，从项目开始创建，到调试方式（扫码离线调

试）到发布时，编译然后上传包，自动升级版本，打tag，请求node服务器记录版本和包信息，最终用js bundle编译成android的maven包和ios的pod包，供客户端开发在工程中直接依赖。整个过程对开发者来说非常隐蔽，只是几个固定的命令，但是背后做了很多事情。

4. 开发框架。因为客户端开发对整个React的项目的组成和原理并不了解，要让他们写出健壮的应用，需要将项目分层和每层的定义和组合方式固定下来，这个事情就是通过开发框架去约束，例如只能在pages下面定义页面（继承自basePage，然后有一些固定的定义页面属性的方式），只能在components里定义子组件，stores下面定义viewModel，models下定义model实体，包括router都是自动管理的（根据pages扫描生成），结合mobx，整个应用的开发模式非常固定，结构也很简单清晰。后续还打算做一些更深层次的封装，参考一些服务端框架的封装思想。

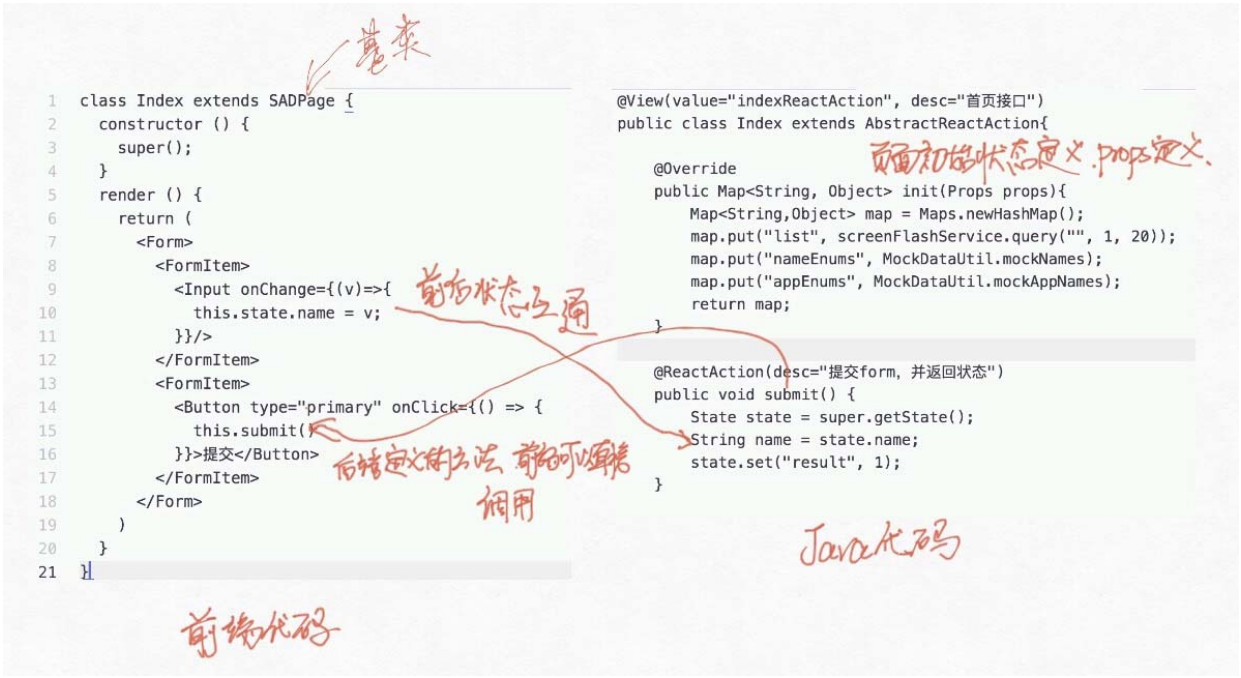
5. 开发规范/模版项目/示例项目，这些东西也非常重要，在项目初始化的时候，项目中就会存在很多代码，其中每一步代码基本都有详细注释，也会有两个示例页面，注释中会有详细的文档链接，新人利用这个模版项目，可以快速理解各个部分的含义。另外项目中还会代又一个example的文件夹，其中演示了一些常见场景的数据定义和页面的写法。加上大量的规范文档，对于初学者非常友好。

6. 功能的封装，将页面跳转，网络请求，缓存，全局事件等操作封装成功能库，简化使用，做各种预处理等。

7. 严格控制热更新能力，严格的RN版本和app版本依赖管理，事实上下个月我们准备将RN的热更新完全关闭，将RN的业务完全当作原生业务来依赖和集成，这样有两个好处，一是控制风险（强制走发版流程，以及版本锁定），二是降低native开发集成发布RN业务时候的困惑。

后台系统页面开发方案

我们团队所有的不面向第三方用户的系统都是Java服务端开发，为了达成这个目标，需要给Java服务端选型一个封装度足够高的UI库（Ant-design），然后要给他们赋予一套用传统Java代码+JSP模板渲染的方式来开发React。这就是这套方案的初衷，内部成为SAD，事实上是 React for Java。



具体细节不细讲了，就是利用前后端不断的同步状态来达到数据共享。效果就是，开发将React组件完全当做是模板引擎来用，数据的定义和操作逻辑，甚至action逻辑，都是在Java端完成的。

1. 提供整套开发框架，前端的封装+后端的封装，就是几个基类+一些注解，还有一些错误处理，表单验证的封装。
2. 提供一套拖拽生成jsx代码的可视化工具，事实上大部分页面都不需要自己写jsx，完全可以用工具拖出来。
3. 提供ant-design组件的二级封装和业务组件封装，尽量减少开发需要写的jsx代码。
4. 大量的常见场景处理demo供开发参考。

NodeJS开发体系

NodeJS除了给自己提供的基础体系以支撑业务之外，还需要给团队输出一些工具和解决方案。简单介绍下。

1. 自己的Node 开发框架，约定各种分层接口定义校验，自产的ORM，还有一些内网协议适配，队列系统，定时任务等。
2. 前端的一些通用解决方案，例如服务端渲染，无痛接入前端业务，做好运维报警性能优化等底层。还有短链这种体系，为前端开放出去的链接做分组数据管理等。

结语

一个好的架构师必然有自己的一套工作习惯，清楚地知道自己的职责，知道自己应该关注的重点，利用自己的经验和方法来控制风险、简化开发、提高效率，希望大家都能成

为牛逼的架构师，当然这一切都要先基于对技术的广度和深度的深挖。有问题请在chat讨论时提出。