大型项目前端架构浅谈

update：2019.06.17更新2.4文章链接

目录：

1、综合

我在2年之前，写过一篇中小型项目的前端架构浅谈。随着能力的上升，以及在阿里巴巴工作的经验，是时候写一篇大型项目的前端架构分析了。

本篇文章不会更多侧重于具体技术实现，而是尝试从更高角度出发，分析为什么要这么做，这些设计能解决什么问题，成本和收益如何。

由于作者能力有限，可能会有所缺漏或者部分错误，欢迎读者指出。

1.1、适用场景：

本篇文章，适用于单个/多个大型项目、拥有超过10个以上的前端开发的场景。

前端项目的规模不同，成本收益比也会有所差别。通常来说，人员越多、项目复杂度越高，那么收益/成本的比值越大。

对于人数较少、项目简单的开发团队，可能有部分措施不适用，因此应该根据具体情况来选用。

1.2、核心思想：

【1】解决问题：前端架构的设计，应是用于解决已存在或者未来可能发生的技术问题，增加项目的可管理性、稳定性、可扩展性。

【2】人效比：对于需要额外开发工作量的事务（本文中存在一些需要一定开发量的内容），我们在决定是否去做的时候，应该考虑到两个要素：第一个是花费的人力成本，第二个是未来可能节约的时间和金钱、避免的项目风险与资损、提高对业务的支撑能力以带来在业务上可衡量的更高的价值、以及其他价值。

【3】定性和定量：架构里设计的内容，一定要有是可衡量的意义的，最好是可以定量的——即可以衡量带来的收益或减少的成本，至少是可以定性的——即虽然无法用数字阐述收益，但我们可以明确这个是有意义的，例如增加安全性降低风险。

【4】数据敏感：专门写这一条强调数据作为依据的重要性。当我们需要说服其他部门/上级管理者，以推动我们设计的内容时，只有数据——特别是跟钱有关的数据，才是最有说服力的证明。

由于篇幅所限，本文很难直接给出定量的值，因此建议架构设计者，先确保项目中设计使用2.7里的埋点系统，根据埋点系统获取的数据，对项目效果进行定量分析，并以此写成PPT和其他部门/上级管理者进行协调。

1.3、切入角度：

分为基础层和应用层。

基础层偏基础设施建设，与业务相关性较低。

应用层更贴近用户，用于解决某一个问题。

部分两个都沾边的，根据经验划分到其中一个。

1.4、其他

由于已经谈到架构层级，因此很多内容，并不仅仅只属于前端领域，有很多内容是复合领域（前端、后端、运维、测试），因此需要负责架构的人，技术栈足够全面，对未来发展有足够的前瞻性。

文章的内容结构为：【项目】—>【解决的问题和带来的好处】—>【项目的实际意义】

2、基础层设计

2.1、自建Gitlab

这个是基础的基础了。本不应该提的，不过考虑到我最近面试的几家公司，有的公司（人数并不少）并没有使用Gitlab，因此专门提一下，并且使用这个的难度非常低。强烈建议使用Gitlab进行版本管理，自建Gitlab难度并不大，方便管理，包括代码管理、权限管理、提交日志查询，以及联动一些第三方插件。

意义：

公司代码是公司的重要资产，使用自建Gitlab可以有效保护公司资产。

2.2、版本管理

版本管理的几个关键点：

- 发布后分支锁死，不可再更改：指当例如0.0.1版本成功发布后，不可再更改0.0.1分支上的代码，否则可能会导致版本管理混乱。
- 全自动流程发布；指应避免开发者提交后，手动编译打包等操作，换句话说，开发人员发布后，将自动发布到预发布/生产环境。开发人员不和相关环境直接接触。实现这个需要参考下面的2.3。
- 多版本并存；指当例如发布0.0.2版本后，0.0.1版本的代码应仍保存在线上（例如CDN），这样当出现线上bug时，方便快速回滚到上一个版本。

意义：

提高项目的可控性。

2.3、自动编译发布Jenkins

这个工具用于在代码发布后，执行一系列流程，例如自动编译打包合并，然后再从Gitlab发布到CDN或者静态资源服务器。

使用这个工具，可以让一般研发人员不关心代码传到Gitlab后会发生什么事情，只需要专心于开发就可以了。

意义：

让研发人员专心于研发，和环境、运维等事情脱钩。

2.4、纯前端版本发布

纯前端版本发布分为两步：

- 前端发布到生产环境——此时可以通过外网链接加正确的版本号访问到新版本的代码，但页面上的资源还是旧版本；
- 前端通过配置工具（或者是直接更新html文件），将html中引入的资源，改为新版本。

解决的问题是：当前端需要发布新版本时，可以不依赖于后端（根据实际情况，也可以不依赖于运维）。毕竟有很多需求并不需要后端介入，单纯改个前端版本后就要后端发布一次，显然是一件非常麻烦的事情。

这个需要专门的工具，用于配置版本发布，我最近就在写这个。

意义：

提高发布效率，降低发布带来的人员时间损耗（这些都是钱），也可以在前端版本回滚的时候，速度更快。

文章链接：

juejin.im/post/684490…

2.5、统一脚手架

适用场景：有比较多独立中小项目。好处：

- 可以减少开发人员配置脚手架带来的时间损耗（特殊功能可以fork脚手架后再自行定制）；
- 统一项目结构，方便管理，也降低项目交接时带来的需要熟悉项目的时间；
- 方便统一技术栈，可以预先引入固定的组件库；

意义：

提高开发人员在多个项目之间的快速切换能力，提高项目可维护性，统一公司技术栈，避免因为环境不同导致奇怪的问题。

2.6、Node中间层

适用场景：需要SEO且前端使用React、vue，或前端介入后端逻辑，直接读取后端服务或者数据库的情况。

- SEO：仁者见仁智者见智，虽然很多公司已经不做了，但通常认为，还是有一定意义的（特别是需要搜索引擎引流的时候），因此React或者Vue的同构是必须的。并且同构还可以降低首页白屏时间；
- 前端读取后端服务/数据库：好处是提高前端的开发效率和对业务的支持能力，缺点是可能导致P0级故障。

意义：

让前端可以侵入后端领域，质的提升对业务的支持能力。

2.7、埋点系统

强烈推荐前端做自己的埋点系统。这个不同于后端的日志系统。

前端埋点系统的好处：

- 记录每个页面的访问量（日周月年的UV、PV）；
- 记录每个功能的使用量；
- 捕捉报错情况；
- 图表化显示，方便给其他部门展示；

埋点系统是前端高度介入业务，把握业务发展情况的一把利剑，通过这个系统，我们可以比后端更深刻的把握用户的习惯，以及给产品经理、运营等人员提供准确的数据依据。当有了数据后，前端人员就可以针对性的优化功能、布局、页面交互逻辑、用户使用流程。

埋点系统应和业务解耦，开发人员使用时注册，然后在项目中引入。然后在埋点系统里查看相关数据（例如以小时、日、周、月、年为周期查看）[原创水印-作者：零零水(王冬)，微信：qq20004604]。

意义：

数据是money，数据是公司的生命线，数据是最好的武器。

2.8、监控和报警系统

监控和报警系统应基于埋点系统而建立，在如以下场景时[原创水印-作者：零零水(王冬)，微信：qq20004604]触发：

- 当访问量有比较大的变化（比如日PV/UV只有之前20%以下）时，自动触发报警，发送邮件到相关人员邮箱；
- 比如报错量大幅度上升（比如200%或更高），则触发报警；
- 当一段时间内没有任何访问量（不符合之前的情况），则触发报警；
- 每过一段时间，自动汇总访问者/报错触发者的相关信息（例如系统、浏览器版本等）；

建设这个系统的好处在于，提前发现一些不容易发现的bug（需要埋点做的比较扎实）。有一些线上bug，因为用户环境特殊，导致无法被开发人员和测试人员发现。但其中一部分bug又因为不涉及资金，并不会导致资损（因此也不会被后端的监控系统所发现），这样的bug非常容易影响项目里某个链路的正常使用。

意义：

提高项目的稳定性，提高对业务的把控能力。降低bug数，降低资损的可能性，提前发现某些功能的bug（在工单到来之前）。

2.9、安全管理

前端的安全管理，通常要依赖于后端，至于只跟单纯有关系的例如dom.innerHTML=’xxx ’这种太基础，就不提了。

安全管理的很难从架构设计上完全避免，但还是有一定解决方案的，常见安全问题如下：

- XSS注入：对用户输入的内容，需要转码（大部分时候要server端来处理，偶尔也需要前端处理），禁止使用eval函数；
- https：这个显然是必须的，好处非常多；
- CSRF：要求server端加入CSRF的处理方法（至少在关键页面加入）；

意义：

减少安全漏洞，避免用户受到损失，避免遭遇恶意攻击，增加系统的稳定性和安全性。

2.10、Eslint

Eslint的好处很多，强烈推荐使用：

- 降低低级bug（例如拼写问题）出现的概率；
- 增加代码的可维护性，可阅读性；
- 硬性统一代码风格，团队协作起来时更轻松；

总的来说，eslint推荐直接配置到脚手架之中，对我们提高代码的可维护性的帮助会很大。可以考虑在上传到gitlab时，硬性要求eslint校验，通过的才允许上传。

意义：

提高代码的可维护性，降低团队协作的成本。

2.11、灰度发布

灰度发布是大型项目在发布时的常见方法，指在发布版本时，初始情况下，只允许小比例（比如1~5%比例的用户使用），若出现问题时，可以快速回滚使用老版本，适用于主链路和访问量极大的页面。

好处有以下几点：

- 生产环境比开发环境复杂，灰度发布时可以在生产环境小范围尝试观察新版本是否可以正常运行，即使出问题，也可以控制损失。
- 对于大版本更新，可以先灰度一部分，观察埋点效果和用户反馈（即所谓的抢先试用版）。假如效果并不好，那么回滚到老版本也可以及时止损；
- 当我们需要验证某些想法或问题的时候，可以先灰度一部分，快速验证效果如何，然后查漏补缺或者针对性优化；

灰度发布通常分为多个阶段：【1】1%；【2】5~~10%；【3】30~~50%；【4】全量推送（100%）。灰度发布一定要允许配置某些IP/账号访问时，可以直接访问到灰度版本。

意义：

降低风险，提高发布灵活度。

2.12、前后端分离

这个并不是指常见的前后端分离，而是指在分配前后端管控的领域。

中小项目常见的情况是后端只提供接口和让某个url指向某个html，前端负责html、css、js等静态资源。

但大型项目并不建议这么做，建议前端负责除html以外的静态资源，而html交给后端处理，理由有很多：

- 后端进行渲染，方便统一插入一些代码和资源，例如埋点js，监控js，国际化文本资源，页面标识符等。这些通常是后端通过调用某些服务直接写入的；
- 当页面需要统一的头尾时（参考淘宝里我的淘宝页面），前端不应该关注这些跟当前页面无关的东西；
- 某些东西，如果通过html来管理，那么耦合度太高了，违背了解耦和分离的原则；
- 前端版本发布在后端引入某种功能模块后，可以从单独的页面控制前端发布内容，比更新html更方便，也利于灰度发布；

意义：

更规范的进行页面管理，降低页面和功能的耦合度，减少复杂页面的环境配置时间。

2.13、Mock

Mock也是常见前端系统之一，用于解决在后端接口未好时，生成返回的数据。

我个人不太建议开发一个专门的系统来Mock，更好的Mock手法是直接嵌入到脚手架之中。思路如下：

- 当在开发环境下，访问链接通常是localhost:8000/index.html，此时加入后缀 ?debug=true；
- 封装好的异步请求在发现当前链接有以上标志时，认为是测试环境，访问/userinfo 时，不去读取线上的数据（因为也读取不到），去本地环境读取src/test_ajax/userinfo.json，并将内容返回给用户；
- 异步请求正常拿到数据，在页面中显示[原创水印-作者：零零水(王冬)，微信：qq20004604]；
- 当线上接口可以获取到数据后，从network里找到返回的数据，放入/src/test_ajax/userinfo.json中，此时再次本地调试的话，相当于使用的是线上的真实数据。

复制代码

这种处理，可以降低mock的复杂度，随时更改mock时返回的数据，比单独开发一个mock系统性价比更高。

意义：

在前后端并行开发时，降低沟通交流成本，方便开发完毕后直接对接。

2.14、定期备份

备份是常被忽略的一件事情，但当我们遇见毁灭性场景时，缺少备份带来的损失是非常大的，常见场景：

- 服务器损坏，导致存在该服务器上的内容全部完蛋；
- 触发某致命bug或者错误操作（例如rm -f），导致文件和数据全部消失；
- 数据库出现错误操作或出现问题，导致用户数据、公司资产遭受严重损失；

总的来说，没人想遇见这样的场景，但我们必须考虑这种极端情况的发生，因此需要从架构层面解决这个问题。常见方法是定期备份、多机备份、容灾系统建设等。

意义：

避免在遭遇极端场景时，给公司带来不可估量的损失。

3、应用层设计

3.1、多页和单页

除了特殊场景，通常推荐使用多页架构。理由如下：

- 多页项目，页面和页面之间是独立的，不存在交互，因此当一个页面需要单独重构时，不会影响其他页面，对于有长期历史的项目来说，可维护性、可重构性要高很多；
- 多页项目的缺点是不同页面切换时，会有一个白屏时间，但通常来说，这个时间并不长，大部分现有大公司的线上网页，都是这样的，因此认为是可以接受的；
- 多页项目可以单次只更新一个页面的版本，而单页项目如果其中一个功能模块要更新（特别是公共组件更新），很容易让所有页面都需要更新版本；
- 多页项目的版本控制更简单，如果需要页面拆分，调整部分页面的使用流程，难度也会更低；
- 灰度发布更友好；

之前面试的一家，采用了单页的形式，之前因为种种原因，同时采用了ng和react。由于项目历史也比较久（3年以上），结果导致目前继续维护更新的难度很大，即使想部分重构，也很麻烦。

意义：

降低长期项目迭代维护的难度，

3.2、以应用为单位划分前端项目

在项目比较大的时候，将所有页面的前端文件放入到同一个代码仓库里，我之前参与过一家企业的前端项目开发，发现其就是这么做的。根据使用经验来看[原创水印-作者：零零水(王冬)，微信：qq20004604]，存在很多问题：

- 会极大的增加代码的维护难度；
- 项目会变得很丑陋；
- 不方便权限管理，容易造成页面误更改或代码泄密；
- 任何人都有权利改任何他能看到的页面（在合并代码的时候，管理人员并不能确定他本次修改的页面是否是需求里他应该改的页面）；
- 发布成本高，即使改一个页面，也需要发布所有资源；

因此，我们应该避免这种现象的发生，个人推荐以应用为单位进行开发、发布。所谓应用即指一个业务涉及到的前后端代码，好处很多：

- 方便进行管理，当某个业务有需求变更时，可以只给研发人员该业务前端应用的developer权限；
- 在需要发布某业务时，只需要发布该业务的所属应用即可；

意义：

规范项目，增加代码的安全性，降低项目维护成本。

3.3、基础组件库的建设

这个蛮基础的，对于组件库的建设，不建议研发人员较少时去做这件事情，专职前端开发人数少于10人时，建议使用比较靠谱的第三方UI库，例如Antd，这样性价比更高。

设计基础组件库的前提，是要求统一技术栈，这样才能最大化基础组件库的效益。组件库建议以使用以下参考标准：

- 使用ts；
- 可扩展性强；
- 适用程度高；
- 文档清楚详细；
- 版本隔离，小版本优化加功能，大改需要大版本更新；
- 和UI协调统一，要求UI交互参与进来；

总的来说，建设起来后，利大于弊，但是需要专人维护，因此还是有一定成本的。

意义：

统一不同/相同产品线之间的风格，给用户更好的体验，减少单次开发中写UI组件时浪费的时间和人力，提高开发效率。

3.4、技术栈统一

前端有三大主流框架，还有兼容性最强jQuery，以及各种第三方库，UI框架。因此项目需求如果复杂一些，很容易形成一个大杂烩。因此前端的技术栈必须统一，具体来说，建议实现以下举措：

- 三大框架选型其一，团队水平一般推荐Vue、水平较好推荐React，对外项目选React或者ng；
- 需要兼容IE8或更老版本时，建议使用jQuery；
- 组件库自建或者统一选择一个固定的第三方；
- 一些特殊第三方库统一使用一个版本，例如需要使用地图时，固定使用高德或百度或腾讯地图；

- 基础设施建设应避免重复造轮子，所有团队尽量共用，并有专门的前端平_台负责统一这些东西，对于特殊需求，可以新建，但应当有说服力；

总的来说，技术栈统一的好处很多，可以有效提高开发效率，降低重复造轮子产生的成本。

意义：

方便招人，简化团队成员培养成本，以及提高项目的可持续性。

3.5、浏览器兼容

常见的问题是IE6、7、8，以及部分小众浏览器（PC和手机）产生的奇怪问题。因此应该考虑统一解决方案，避免bug的重复产生。常见解决方案有：

- 配置postcss，让某些css增加兼容性前缀；
- 写一个wepback的loader，处理某些特殊场景；
- 规范团队代码，使用更稳定的写法（例如移动端避免使用fixed进行布局）；
- 对常见问题、疑难问题，总结解决方案并团队共享；
- 建议或引导用户使用高版本浏览器（比如chrome）；

意义：

避免浏览器环境产生的bug，以及排查此类bug所浪费的大量时间。

3.6、内容平_台建设

为了提高公司内部的沟通效率，总结经验，以及保密原因。应建设一个内部论坛+博客站点。其具备的好处如下：

- 可以记录公司的历史；
- 研发同学之间分享经验；
- 总结转载一些外界比较精品的文章，提高大家的眼界；
- 增加公司内部同学的交流，有利于公司的团队和文化建设；
- 对某些技术问题可以进行讨论，减少因没有达成共识带来的沟通损耗；

众所周知，大型互联网公司通常都有这样一个内部论坛和博客站点。其降低了公司的沟通和交流成本，也增加了公司的技术积累。

意义：

博客增强技术积累，论坛增强公司内部沟通能力。

3.7、权限管理平_台

当公司内部人员较多时，应有一个专门的平_台，来管理、规范用户的权限以及可访问内容[原创水印-作者：零零水(王冬)，微信：qq20004604]。权限管理平_台有几个

特点：

- 必然和Server端天然高耦合度，因此需要有专门的控制模块负责处理权限问题（负责Server端开发处理，或者前端通过中间层例如Node层介入处理）；
- 自动化流程控制，即用户创建、申请、审批、离职自动删除，都应该是由系统推进并提醒相关人士，必要时应能触发报警；
- 权限应有时效性，减少永久性权限的产生；
- 审批流程应清晰可见，每一阶段流程应具体明确；
- 应与公司流程紧密结合，并且提高可修改性，方便公司后期进行流程优化；

意义：

使得公司内部流程正规化、信息化。

3.8、登录系统设计（单点登录）

当公司内部业务线比较复杂但相互之间的耦合度比较高时，我们应该考虑设计添加单点登录系统。具体来说，用户在一处登录，即可以在任何页面访问，登出时，也同样在任何页面都失去登录状态。SSO的好处很多：

- 增强用户体验；
- 打通了不同业务系统之间的用户数据；
- 方便统一管理用户；
- 有利于引流；
- 降低开发系统的成本（不需要每个业务都开发一次登录系统和用户状态控制）；

总的来说，大中型web应用，SSO可以带来很多好处，缺点却很少。

意义：

用户体验增强，打通不同业务之间的间隔，降低开发成本和用户管理成本。

3.9、CDN

前端资源的加载速度是衡量用户体验的重要指标之一。而现实中，因为种种因素，用户在加载页面资源时，会受到很多限制。因此上CDN是非常有意义的，好处如下：

- 用户来自不同地区，加入CDN可以使用户访问资源时，访问离自己比较近的CDN服务器，降低访问延迟；
- 降低服务器带宽使用成本；
- 支持视频、静态资源、大文件、小文件、直播等多种业务场景；
- 消除跨运营商造成的网络速度较慢的问题；

- 降低DDOS攻击造成的对网站的影响；

CDN是一种比较成熟的技术，各大云平_台都有提供CDN服务，价格也不贵，因此CDN的性价比很高。

意义：

增加用户访问速度，降低网络延迟，带宽优化，减少服务器负载，增强对攻击的抵抗能力。

3.10、负载均衡

目前来看，负载均衡通常使用Nginx比较多，以前也有使用Apache。当遇见大型项目的时候，负载均衡和分布式几乎是必须的。负载均衡有以下好处：

- 降低单台server的压力，提高业务承载能力；
- 方便应对峰值流量，扩容方便（如举办某些活动时）；
- 增强业务的可用性、扩展性、稳定性；

负载均衡已经是蛮常见的技术了，好处不用多说，很容易理解。

意义：

增强业务的可用性、扩展性、稳定性，可以支持更多用户的访问。

3.11、多端共用一套接口

目前常见场景是一个业务，同时有PC页面和H5页面，由于业务是一样的，因此应避免同一个业务有多套接口分别适用于PC和H5端。[原创の水印-作者：零零水(王冬)，QQ：20004604]因此解决方案如下：

- 后端提供的接口，应该同时包含PC和H5的数据（即单独对一个存在亢余数据）；
- 接口应当稳定，即当业务变更时，应尽量采取追加数据的形式；
- 只有在单独一端需要特殊业务流程时，设计单端独有接口；

多端共用接口，是减少开发工作量，并且提高业务可维护性的重要解决方案。

意义：

降低开发工作量，增强可维护性。

4、总结

由于各个公司具体情况不同，项目也具有特殊性，因此以上设计不可强行套入，应根据自己公司规模、项目进展、人员数量等，先添加比较重要的功能和设计。并需要考虑到长期项目的可维护性和发展需要，对部分基础设施进行提前研发设计。

篇幅所限，因此无法面面俱到，只提了一些我认为比较重要的架构层面需要考虑的内容，欢迎大家补充。大家如果有自己的看法，欢迎回复，或者添加我的微信qq20004604（昵称：零零水）进行讨论。

最后问一下，西安有没有不加班，并且需要前端架构师的公司，请联系我。