0. 前情提要

0.1 背景

在阿里加班加腻了的我，遂选择国企，作为人生的下一站。

众所周知，当今中国，不考虑某些不和谐的东西，最赚钱的只有两个行业，那就是IT，以及金融。

于是，聪明的我，既然已经干过 IT 了，那么接下来，当然要去干金融小……！——哦不，是在金融业干 IT。

于是，我回到了西安，选择了一家信托公司开始干前端组长（请不要理解错，不是那个干），这个还算有点前途的工作。

0.2 当时环境

这是一家信托公司，也是一家国企。

众所周知，大部分国企的技术体系，很一般。不过还好的是，从上面的大领导，到部门领导，都很有能力，也很有魄力。这也是为什么我来这家公司的原因。毕竟，来国企虽然虽然为了稳定，但也不想混日子嘛。

在我刚刚来到这家公司时，面对的是前任留下的烂摊子。具体来说，有以下这些：

1. 现有前端组员 3 人，优点是态度很好，缺点是技术水平一般，特别是代码习惯并不好；
2. 遗留前端项目一个，但问题很大，具体情况下面说；
3. 原本预期工期三个月的项目，已经 5 个月了，还没完成；
4. 后端提供的接口，变更频率很大，并且经常无告知变更，导致代码容易突然不能运行；
5. 其他，略；

特别要说一下这个前端工程相关的问题：

1. 所有后端接口都是跨域访问的，一方面后端是开跨域的，另一方面，当访问到后端接口，前端用的路径是绝对路径；
2. 有四个环境（开发/测试/预生产/生产），打包到不同环境需要执行四个不同的命令（记不记第一条例，绝对路径的接口url），也就是说打四种包；
3. 没有自动化部署（即 CI/CD），需要手动打包，然后复制代码，通过 ftp 上传到不同服务器。所以很容易误操作；
4. 其他基本没有。

基本上，只有烂摊子，没什么值得称道的地方。

唯一值得庆幸的是，领导的支持力度还可以，虽然只能招外包开发，但是招人的权利全权放给我，我说要，那就招。

0.3 上一篇博客

19 年 5 月，我写过一篇 8000 字的《大型项目前端架构浅谈》，当时在掘金曾经最高升到历史总榜第三的位置（累计阅读接近 12w，点赞 3500+）。但又有一些人反馈，觉得说的有点浅，缺少具体实现方法。

但那篇，即使谈的比较浅，也已经 8000 字了，这篇相对上一篇博客，细化谈了几点，我在写这句话的时候算了一下，大概也已经有 8000 字了。

所以不是我不想谈深一点，是想写深一点，每一点都可能有 8000 字以上，尽力了。

1. 解决遗留问题

1.1 定思路

在做事前，必须明确思路。几个原则：

1. 确保现有稳定;
2. 先做低成本高产出的事情;
3. 考虑长期成本尽可能低;

结合以上情况，我判断：

1. 最重要的是人的因素;
2. 其次是判断问题的轻重缓急，逐个解决;
3. 最后是在日常搭建优化前端技术体系，尽量通过工具和规范来提高大家代码质量。

虽然以上问题，我都可以一个人解决，并且做的很好，但是，我一个人写不完所有的项目。即使我能力再强，但随着团队扩大，边际效益必然越来越低。

1.2 了解团队成员

所以我先单独约每位同学聊天，了解其为人想法，以及个人情况，为接下来的行动做准备。

谈话过程中，我先了解他们想法、个人情况，基于其个人意愿，为他们的未来提出设想，并给出具体可行的发展方向。

谈话结果总体还可以，起码大家表面上来看，大家都是有想法有干劲的（最后验证，的确如此）。不用担心有人故意拖后腿。

【ok，人的问题解决】

1.3 轻重缓急分析

对已有问题进行分析判断：

问题	严重程度	修复成本
接口变更	高，严重影响进度	低，只要和后端协调好即可
缺少CI/CD	中	低，需要写个脚本自动从 gitlab 同步代码
脚手架质量低	中	中，对旧项目影响大
接口跨域	高	已有代码中，新工程低
组员代码不规范	高	高，需要时间
忘了列出来的	*	*

【现有问题明确】

1.4 明确架构师任务

前端架构师进行前端技术建设的核心目的，是为了提高开发效率，保证开发质量，为保障项目高质量按时交付。

同时兼顾考虑中长期研发实际情况，结合团队实际能力，为未来做技术储备，为业务发展提供更多的可能性。

于是，将自己的任务分为以下三类：

1. 基础架构设计：主要目的是从架构层面出发，通过流程化设计，规避常见问题，提高开发效率；
2. 工程化设计：与代码强相关，主要目的是提高代码质量，增强代码的长期可维护性，降低开发时间和成本；
3. 团队管理类：通过合理有效的团队管理，提高团队人效比，为未来项目研发、技术发展，进行人才储备、技术研发；

我要从以上三个角度切入，既能解决已有问题，又能为长期未来提供支持。

【方向明确】

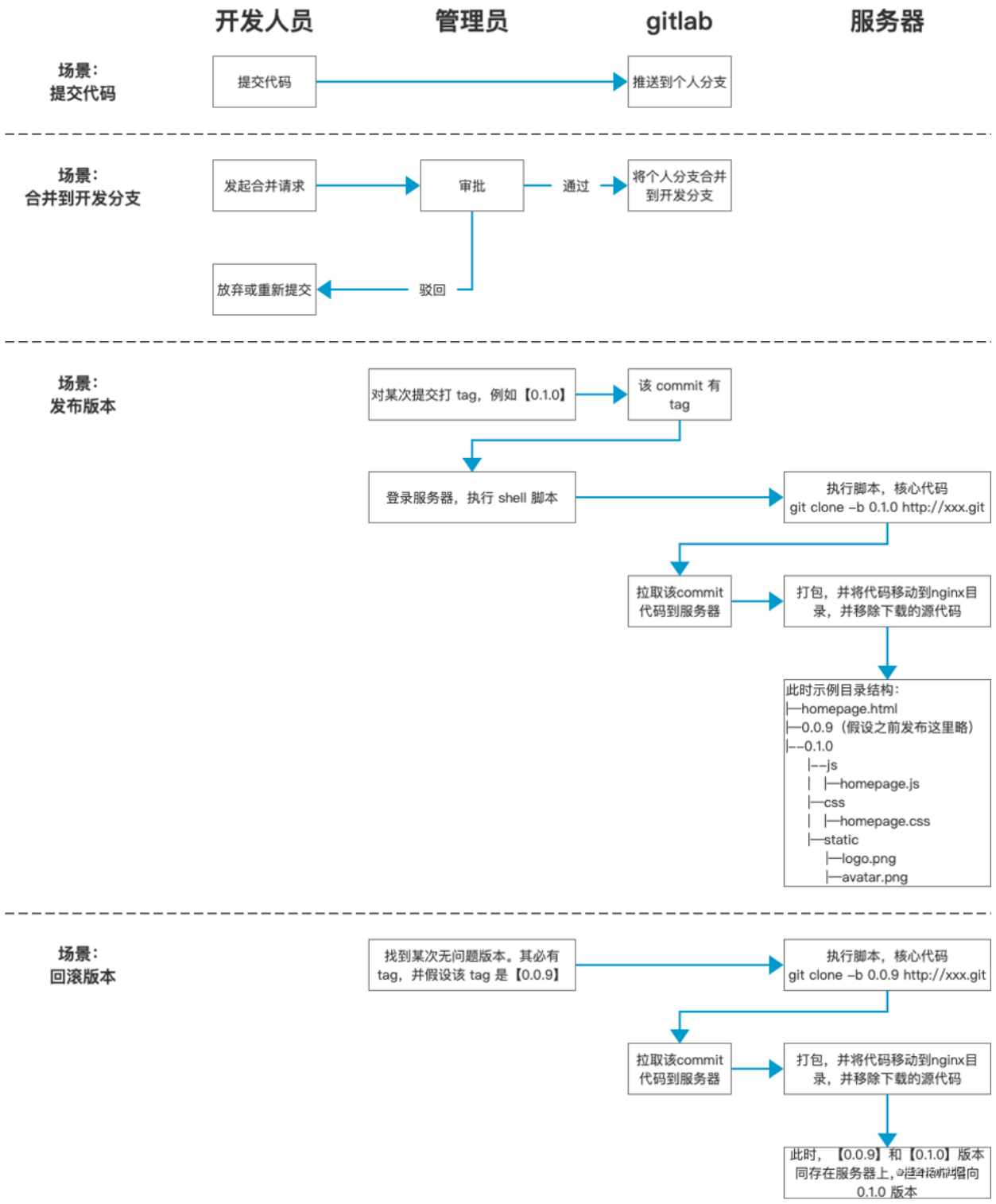
于是，开始干活，用了 4 个月，做了以下工作。

2. 基础架构设计

2.1 前端pipeline

基础架构的核心设计，是流程设计。好的流程可以高效率低成本的运行，同时可以通过流程化规避大量问题。

基本流程设计如下：



01.png

在这个流程中，代码被分成了三层：【个人代码——主分支代码——线上代码】，其中线上代码又分成了dev、qa、pro 三个版本。

开发人员只对个人代码拥有可控权，而无法直接影响改变主分支代码，当需要提交到主分支时，需要发起 merge 请求，并经过管理员 review 代码后，将其代码合并到主分支。

而主分支代码又和线上代码进行隔离，由管理员将指定版本代码发布到指定环境。无论是哪个dev、qa 还是 pro，都将直接从 gitlab 上拉取指定提交，然后打包发布。通过这种二次隔离，导致代码环境可控、干净、稳定。因此，无论在哪个环境，代码的发布都是幂等的，换句话说，是稳定可控可预期的。

通过以上流程，前端代码能被保证以高质量高稳定性的状态，运行在服务器端。

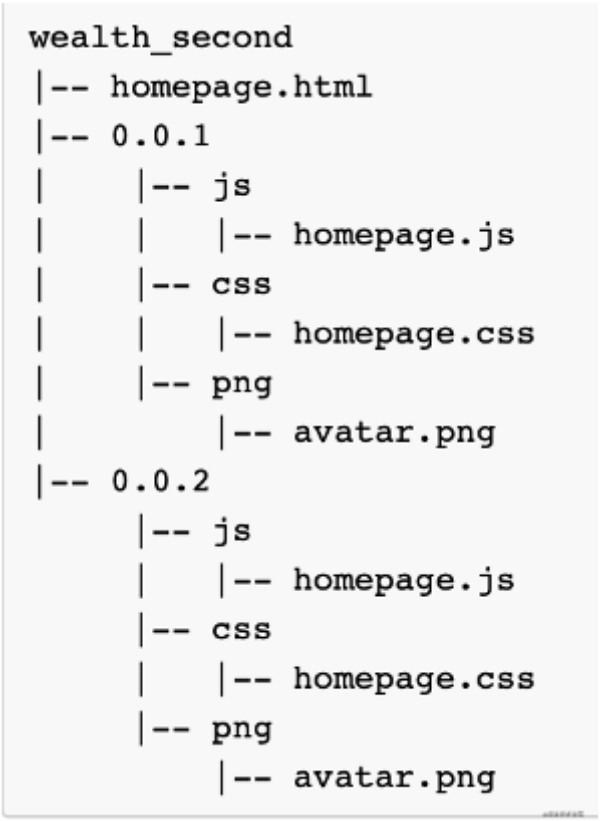
2.2 代码版本控制

传统前后端分离项目中，前端代码通常由前端自己独立部署，打包完成后，直接部署到指定目录下。但该流程有几个严重的问题：

1. 缓存问题：为了提高用户访问速度，降低加载资源带来的流量压力，我们通常会对前端静态资源做缓存处理。但缺点在于，当我们需要发布或者回滚版本时，不可避免的会有用户访问到缓存静态文件。
2. 版本发布、回滚速度慢：传统方案，每次需要发布、回滚都需要重新打包发布代码。这个流程一方面比较慢，另一方面，不一定能准确打包到具体的某一次提交（不准确）。

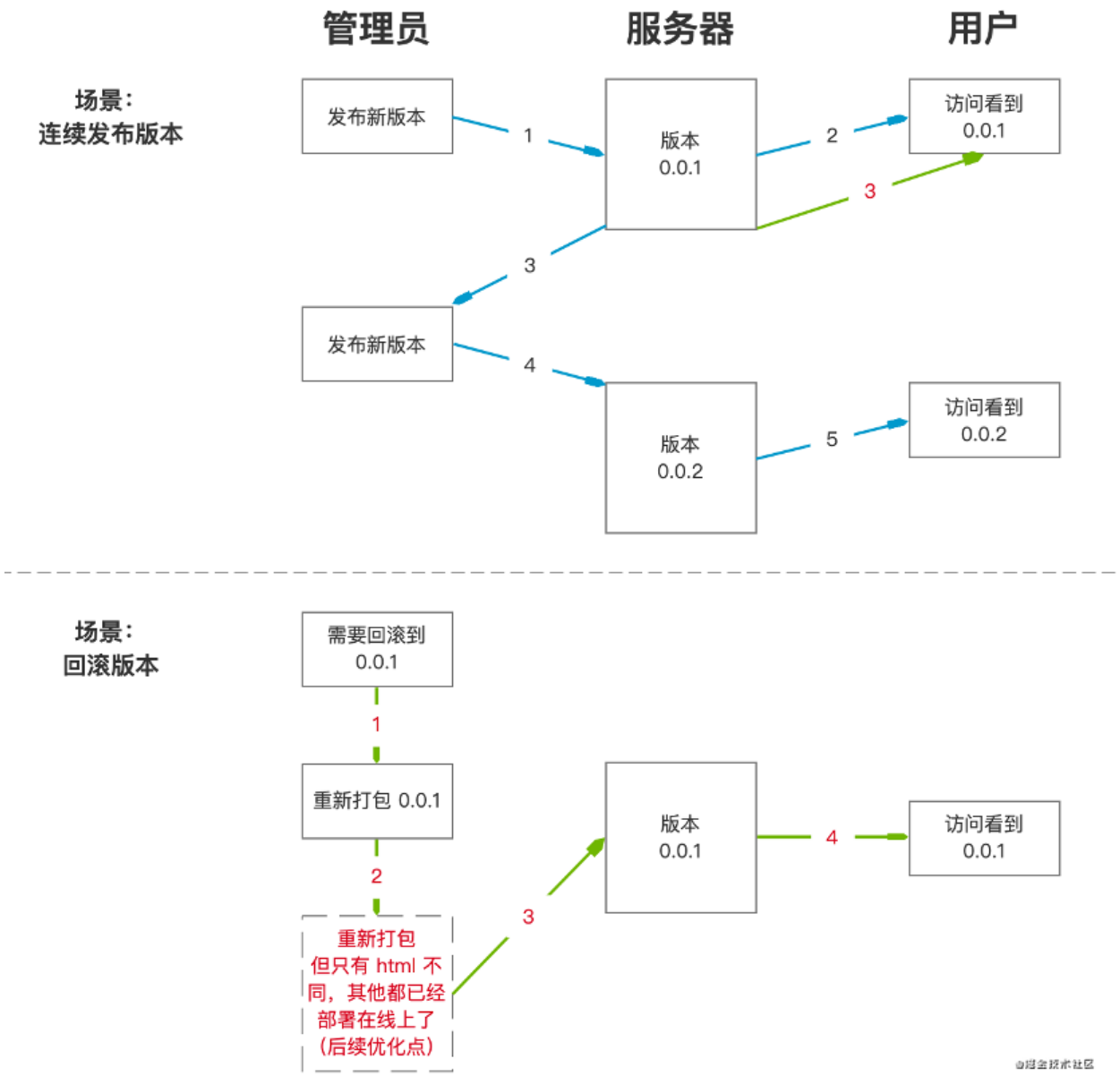
因此，根据实际情况，使用自研的前端 pipline 管理。核心有以下几点：

1. 代码版本管理：对于需要提交到线上服务器的代码，在 gitlab 对其添加 tag，只有添加正确 tag 的代码才会被打包发布到线上。该操作关键在于 css、js 等静态资源文件，以版本号作为文件夹目录，同时多版本并存线上。html 不做 cache，作为入口，引入不同版本。示例目录如下：



02.png

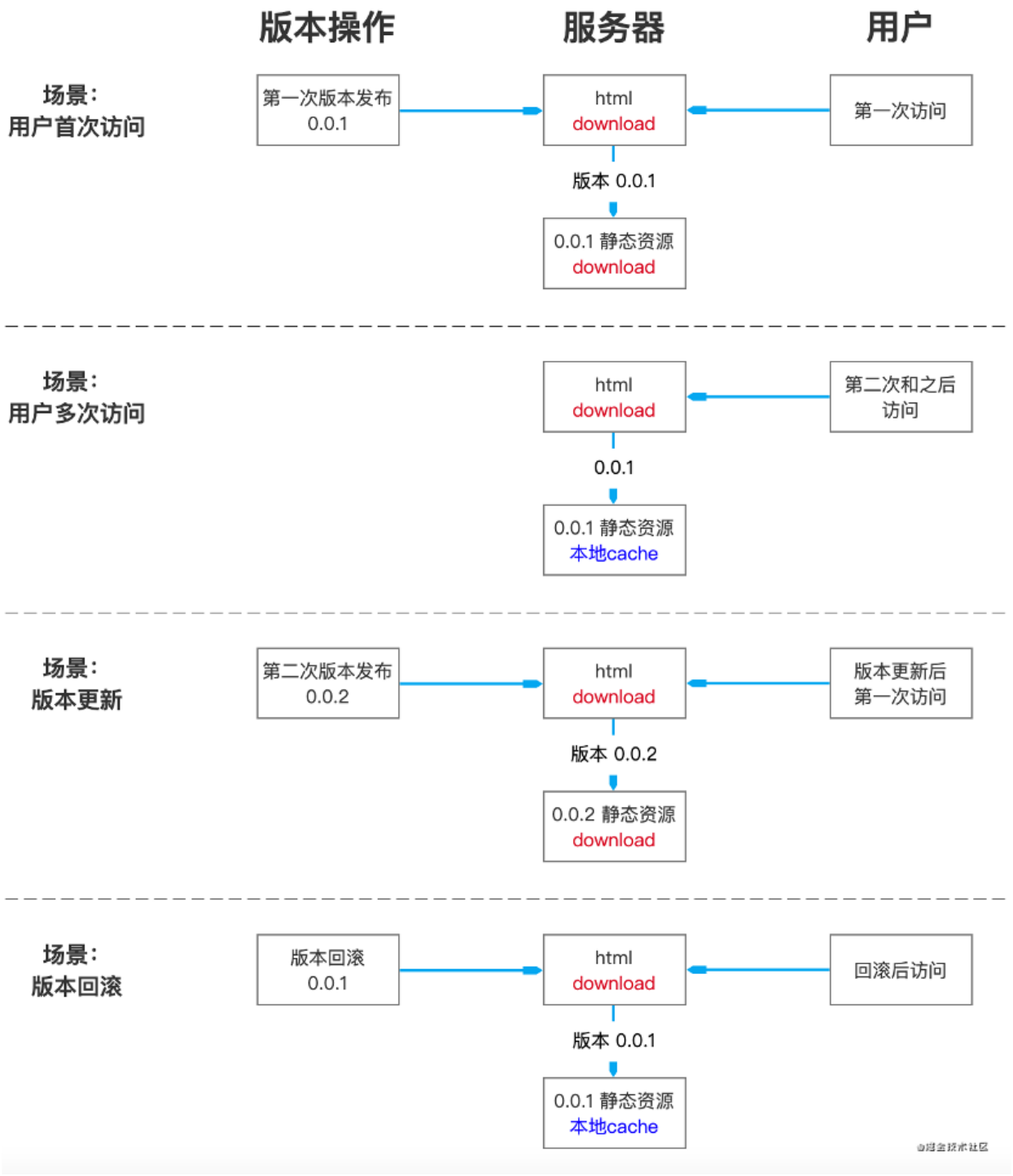
在 nginx 层，html 做无缓存处理，每次加载都加载最新的。其他静态资源做 cache，过期时间 15 天，发布流程见上图。切换版本时，具体如下：



03.png

2.多版本共存：由于静态文件本身资源大小比较小（由于做了公共静态文件抽离，所以其实更小），目前财富二期打包后只有 16 MB，未来二次优化后，将降低到 10 MB 左右。所以即使多版本同时放到服务器上，其占用也不多。

管控方法是作为入口的 html 文件是唯一的，通过修改 html 里，指向的静态资源文件的版本号，来指定用户当前访问的版本号。用户访问流程图如下：



04.png

可以从上图中看出，用户只会在第一次访问的时候，加载静态资源。其余时间访问的都是本地 cache，可以极大的提高用户访问速度和用户体验，降低服务器网络 IO 压力。

而html 不做缓存处理，又可以避免在新版本发布后，用户访问到的是旧版本页面，而导致业务处理出现异常。

同时，因为每个版本号都是和commit 强耦合的，所以无论哪个环境下，同一个版本号的代码，都必然是幂等相同的，不需要担心不同环境下代码表现不同的问题。

2.3 私有npm源

由于我们通常访问的是内网，因此，需要考虑以后内外网隔离的问题。因此，搭建私有 npm 源，提前从基础设计层面解决问题。私有源采用开源技术方案Verdaccio。优势有以下几点：

1. 轻量：verdaccio基于 Node.js 开发，占用资源少，运行速度快，配置简便，适合我们公司实际情况；
2. 配置难度低：可以快速配置用户组、权限、运行端口、存放地址等，足以满足一般场景；
3. 保证依赖版本稳定：相对于将依赖放到第三方远程源，放本地更稳定，一方面可以提高下载速度（因为部分第三方依赖连外网不稳定），另一方面，可以确保依赖始终存在（被缓存到本地私有 npm 源）；

通过以上三点组合，从架构层面上，解决了发布的线上代码不确定性问题，提高了前端代码的可靠性。

另外，我们前端组会将一些自己封装的 npm 包发布到自己的 npm 源，提供给不同工程使用，避免重复开发带来的研发浪费，以及开发人员能力不稳定带来的公共依赖不可靠的问题。

2.4 公共静态资源cache

除了业务代码以外，前端还会有一些公共静态资源，例如 Vue 的 js 文件、ElementUI的资源文件、Echarts 资源文件，以及一些图片文件等。

对于这些文件，是所有工程所共享的，假如这些文件分散在各个工程里，既没必要，也容易导致不同工程依赖文件不统一。另外，也会重复加载这些文件，浪费网络带宽和静态服务器存储空间，没有意义。

因此，设一个公共静态路径，通过在脚手架里进行配置，当需要加载这些资源时，直接去指定 nginx 静态服务器里加载，并做长时间 cache，可以提高访问效率和性能。

未来，将做一个专门的静态资源管理服务，用于将静态资源（主要是图片），传到静态资源服务，并进行 CDN 加速，然后前端工程通过直接引入图片 url 来使用这些资源。这样可以减轻自己的服务器的网络带宽消耗。

2.5 其他

以上都是已实现的，其他还有一些，但过于琐碎——其实主要是我写不动了，就不写了。

3. 工程化设计

3.1 定制 webpack 脚手架

为了低成本高效率的提高代码质量，基于公司实际情况，定制化开发了一个 webpack 脚手架，解决了以下问题：

1. 避免了无限制引入第三方库：无限制引入第三方库，是一种常被忽略，但却很容易带来问题的行为（例如 vue+jquery 等）。因为开发人员通常并不会阅读第三方库的源代码，不能保证其是安全可靠的。而定制脚手架，提供了所有开发中需要的依赖，并严控新依赖的引入，则确保整个工程是安全的。
2. 约束开发人员代码规范：开发最常见问题是代码规范混乱，各写各的，导致维护成本高企。而通过引入 eslint，强行规定代码风格，自动对不符合规范的代码报错，解决了这些问题。并且由于 IDE 支持通过应用 eslint 规则，自动格式化代码。也降低了对开发人员的约束成本，降低推行代码规范的难度。
3. 方便提供给其他开发使用标准的脚手架，并提供技术支持（比如可以统一大前端的技术栈）。

我使用的脚手架，是自己基于 webpack5 配置的 vue 脚手架，包含各种各样的配置。

3.2 集成调试控制台

考虑到项目的页面里，有大量的表单内容。为了方便开发测试联调，避免每次都需要手动输入很多表单，因此集成了专门的调试控制台。通过特殊操作打开调试控制台后，页面上会出现一个小浮窗，浮窗上有一些预置按钮，点击可以执行一些预定行为（比如填写表单）。

如下图，点击 test 按钮，表单将被自动填满：

05.png

通过集成调试控制台，有以下好处：

1. 减少重复性操作的低效率行为：自动填写表单是最常见的一种场景，也可以在需要无限下拉加载时，直接加载指定页码或增加每页加载数量，从而快速获取到某页数据，并且无需修改业务代码；

2. 更多的调试信息：相对于用户使用来说，开发时，有时候有必要需要在页面上显示更多信息，但若修改原有业务代码，则是不优雅的。因此通过调试控制台开启关闭是否显示这些额外信息。并且由于其是默认关闭的且本地独立的，因此也可以在线上调试的时候使用；

3. 线上调试：未来添加埋点系统后，当用户遇见异常情况而无法复现时，可以教用户打开调试控制台，然后点击按钮推送用户历史操作。之后可以在服务器端分析用户行为，查看问题出现的原因。

3.3 多页主从设计

针对公司实际业务需要——以工作台为核心平台，其他页面接入，并且后端是微服务设计。故设计思路为多工程多页面主从管理。具体来说，有以下几点：

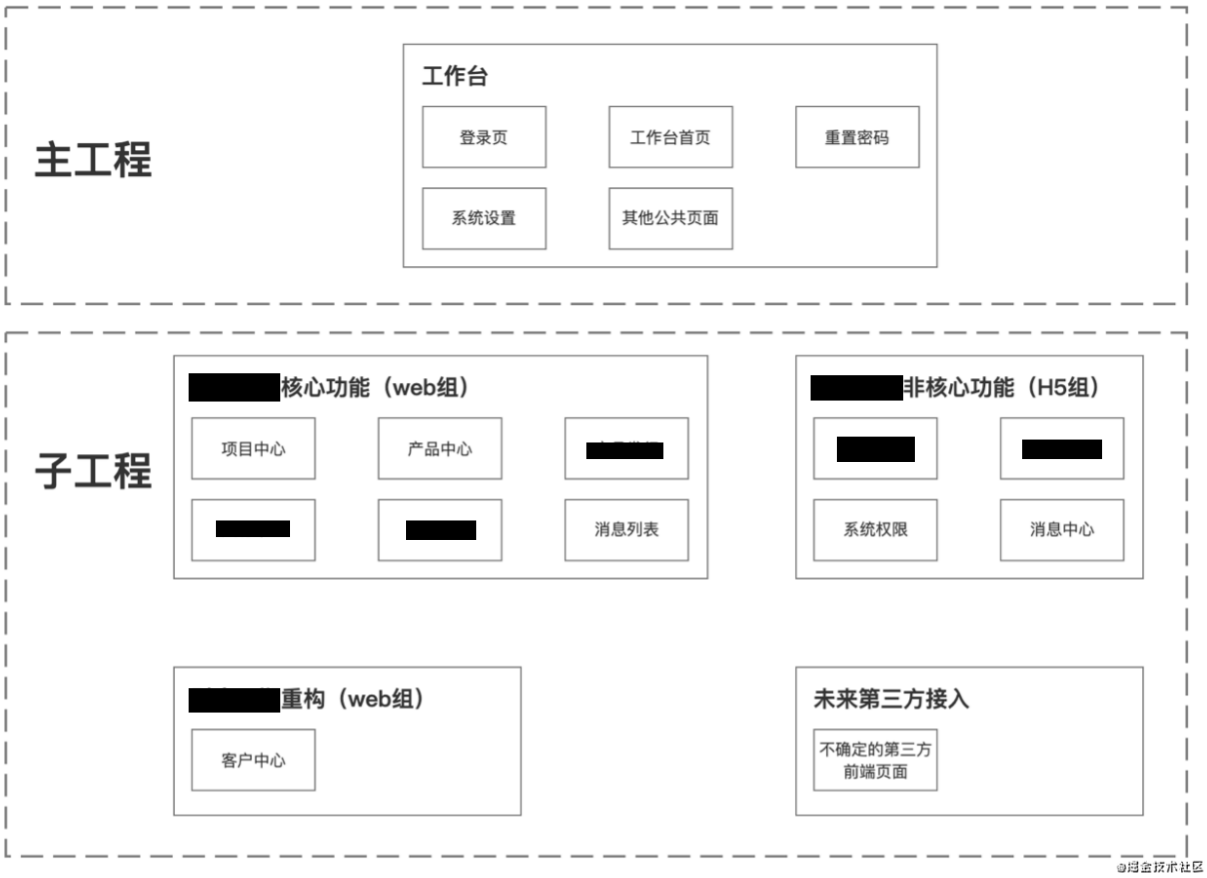
1. 多工程：考虑到我们未来将以工作台为核心，多系统接入的形式来部署。因此拆分前端工程，以工作台为核心入口，然后 story、开发人员、系统等，拆分为多个中小型前端工程。通过这样的拆分，降低了代码开发、维护难度，也减少不同story 开发之间协作成本；

2. 多页面：对于一个长期维护的项目来说，必须考虑到长期维护难度。因此，结合我之前踩坑、维护历史项目的经验，将前端页面设计为多页面模式。具体来说，各个story根据功能点拆分为不同的页面，页面之间跳转以本地缓存、浏览器参数等进行数据交换。通过这样的隔离，无论后续需求是，加入新页面新功能、重构单个页面，还是在新页面上进行新技术尝试，其成本都非常低；

3. 主从管理：介入的工程。其他页面需要在工作台打开的话，需要预先注册后，通过页面 id 来调用打开。这样可以通过集中控制，禁止第三方网页在工作台打开非法页面，提高系统安全性；

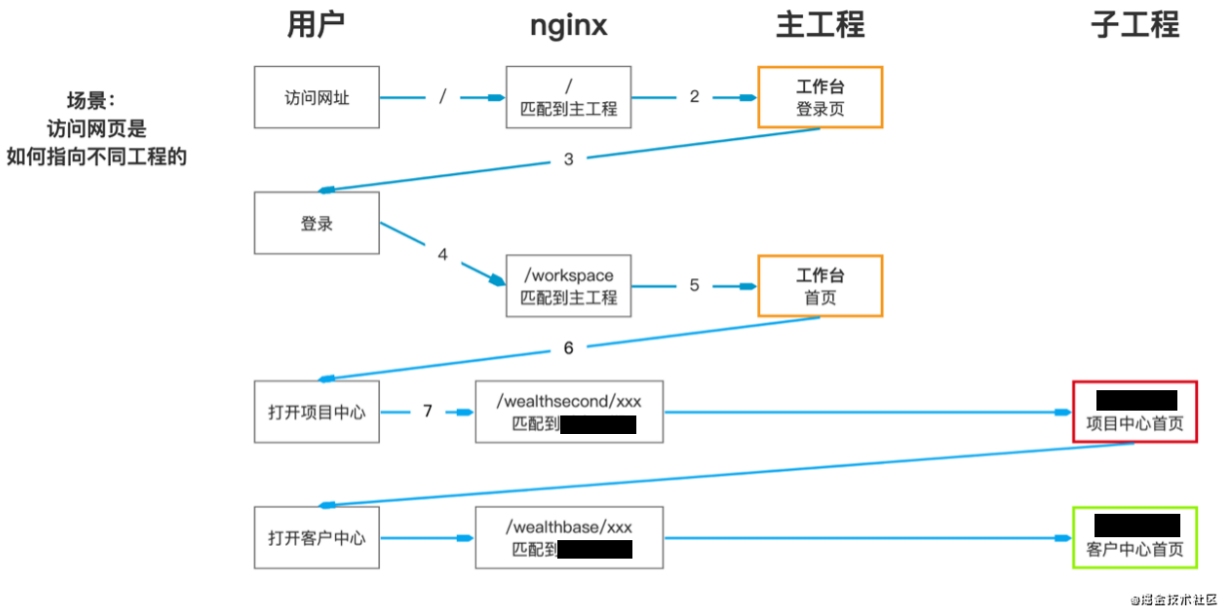
通过这样的设计，将传统大型前端工程，拆分为类似后端微服务的微前端工程，提高开发效率，特别是并行开发效率。也增强了权限管理，确保了代码的安全性。

具体设计结构如下图：



06.png

他们之间的交互流程如下图



07.png

无论背后如何跳转，但因为是通过 nginx 转发的，所以表面上，用户访问的都是同一个网址，不影响用户认知。

4. 团队管理

4.1 权限控制

这里的权限控制，主要是指代码权限控制，目的是确保代码安全，问题可控可避免可追溯。

具体管理举措有以下几条：

1. 保证依赖版本稳定：代码属于公司财产，因此基于 gitlab，对代码进行权限隔离，默认关闭所有访问权限，针对每个工程，按实际需要给开发赋予指定权限。可以避免因代码泄露的公司财产损失；
2. 提交权限：允许开发在自己分支上提交，但涉及到主分支的合并，必须提交 merge，然后在组长进行 code review 后，才能提交到主分支。通过这样的形式，主分支的代码必然是可控的，不会引入严重问题；
3. 发布权限：对于将要发布到 dev、qa、pro 的代码，收敛权限到组长，只允许组长进行发布。好处在于，一方面可以提供一个可靠可控版本的线上代码，另一方面，降低开发人员技能要求。

4.2 前后端接口对接管控

在去年十月十一月份时，前后端开发联调有一个严重问题，就是后端接口变动时，不会在事前事后通知相应前端开发、测试人员。开发效率非常低，并且会出现各种异常情况。

因此，通过梳理开发流程，强制要求后端提供接口文档前置，并且要求有一个可控稳定的接口文档，作为对接标准。

在十二月以后，因为接口临时变动带来的 bug 和开发、测试阻塞问题，粗略估计减少了 80%。

但实际情况中，还是会有一些接口文档和实际接口不符的情况发生，导致接口测试被迫重做，接下来需要设法减少此类问题。

4.3 人员管理

工作都是人在做，所以相对来说，把好人员关，有时候比做事更重要。因此为了确保人员的可靠性、稳定性，从以下几个角度入手：

1. 实战入职面试题：采用理论题+实际业务题结合的方式，确保入职人员真的懂业务开发，避免人员能力风险。在通过初步面试后，安排做一道代码量少，但有一定难度，考验解决问题能力，同时考察态度的笔试题，如果愿意做且能做出来的，根据目前经验来看，基本都比较可靠；

2. 导师帮带：在新人进来后，安排一个人带着他，答复常见问题，带着做一个 story。与由组长直接管理培训相比，更有利于新成员融入团队中；
3. 新人适应：新人常见问题是不明确自己的定位。因此，组长负责安排新成员的发展方向，并在新人入职的第一周，先安排其开发日报系统，了解脚手架，再安排其做基于现有页面的优化，帮助其了解不同人负责的 story；
4. 责任明确：明确团队里每个人的定位，并使其知晓。根据团队成员能力不同，态度不同，安排适合其的任务。举例来说，成亚旭多负责简单页面，以及自测串 story 等工作，马倩负责业务核心 story，王超负责需要一定技术创新的内容；

5. 未来计划

其实还做了一些其他的，但由于篇幅所限，就暂时不写了。（主要是写不动了，这些已经写了一周了啊~~）

1、静态图片资源服务器：

目前是把图片打包到工程里，但这种缺点是工程太重，其次是 nginx 服务器压力大。所以未来需要考虑做一个专门的静态资源服务器，上传后，自动传到 cdn，并返回 cdn 上图片链接。然后在页面里直接引入图片链接即可；

2、backend 层渲染 html：

这个是我未来核心工作重点。相对于目前的nginx 渲染 html，更好的选择是 backend 渲染 html。好处有以下几点：

1. 降低 nginx 服务器压力；
2. 版本发布，从需要手动执行脚本替换html，变为只需要更改配置就能变更前端代码版本；
3. 可以以中间件形式，嵌入一些公共 js、css，比如埋点 js 等；
4. 可以解决 csrf 问题；
5. 强约束网页入口；

缺点是，需要 backend 层开发配合，可能需要改造代码。

1. gitlab 集成 eslint：

相对于要求开发自己检测代码质量，即使继承了 eslint，效果也不是最好的。因此，最好的办法是将 eslint 集成到 CI/CD 中。在开发提交代码到 gitlab 时，自动执行 eslint 审查代码，如果审查失败，则自动打回；

1. 埋点系统：

我们常见问题是无法得知用户操作习惯，以及对报错情况无法捕获。因此，埋点系统是十分有必要的。嵌入埋点系统后，可以从数据层面统计系统的各方面信息，包括访问速度、访问时间段、访问人数、不同岗位人数访问的网页分布情况、

1. CSRF:

金融系统的安全性是非常重要的，除了常规的 防止 xss 注入之外，也需要防止 CSRF（跨站请求伪造）攻击。解决方法就是上 csrf-token 来避免。

终：

总的来说，前端相关的技术建设，已经做了一些，但未来还有更多需要去建设，特别是使其自动化、GUI 化、便利化、集中化。

例如自建 OA 系统，接入钉钉考勤，接入 gitlab，进行全流程自动化（一键发布、推送到服务器、检查可靠性、通知到相关负责人），线上出问题自动报警通知相关负责人。

而且我们最近 2 个月上了 k8s（本文是在开始之前写的，所以其实也只是截止于 k8s 之前的情况），更多东西，将在下面博客继续说吧（估计起码得鸽子半年）。

未来需要做的工作还很多，前端架构师的施展领域，其实也有很多。