前端基建

> 一切基建源于业务，并服务于业务

本文基于常规**中后台业务**，描述从 0 到 1 构建前端基建体系的那些事。本文仅从全局的角度讲解基建过程会遇到什么，有哪些处理方式，对于细节不会有深入的讲解。对于个别需要详细说明的部分，会分文章补充讲解。

你将获得：

- 对前端业务全链路流程有一个全面、清晰的了解；
- 前端基建全貌及相关方案；

适合人群：

- 爱折腾；
- 有技术团队管理与规划的需求；
- 不限前端工程师；

注意：基建与业务关系密切，本文不可能完全适用于读者公司实际场景，仅供参考，切勿生搬硬套。

1 背景

首先，我们来了解一下，什么是基建、基建的意义以及基建落地可能存在的问题与挑战。

1.1 什么是基建

> 以提效为目的，服务于业务的建设称之为基建，可作为团队技术沉淀是否丰富的评判标准。

基建通常包括但不限于：

- **团队规范：** 团队标准化共识，标准化规范是团队高效协作的必要前提；
- **研发流程：** 业务研发链路，标准化流程直接影响上下游的协作和效率；
- **基础资产：** 工具链、物料库（组件、区块、模板等）、团队 DSL 沉淀等；
- **工程管理：** 项目全生命周期的低成本管控，如：项目创建、本地环境配置、打包部署；
- **安全防控：** 三方包依赖安全、代码合规性检查、安全风险检测等防控机制；
- **质量保障：** 自测 checkList、单测、UI 自动化测试、链路自动化测试；
- **性能检测：** 通过自动化、工具化的方式发现页面性能瓶颈，提供优化建议；
- **统计监控：** 埋点方案、数据采集、数据分析、线上异常监控等。

1.2 基建的意义

> 对于每个团队来说，肯定是业务优先，但高效的业务处理离不开基建的支撑。

基建意义归结如下：

- 解决业务问题
- 研发效能提效
- 团队梯队建设
- 团队影响力提升

1.3 问题与挑战

> 基建不是一个团队的口号，落地确实会遇到不少问题

包括但不限于：

- 如何找到适合做基建的人？
- 如何合理地做基建规划？
- 如何在业务开发中找到基建的机会，有方法论吗？
- 如何衡量基建带来的价值？
- 如何合理处理基建与业务的时间、人力管理？
- 如何提升团队基建氛围？
- 如何在公司层面推广基建？

希望在接下来的内容中，你能找到想要的答案。

1.4 目标

结合以上内容，初步拟定基建目标如下：

- [x] 基建人才梯队管理方案
- [x] 常规业务场景的基建规划方案
- [x] 基建的反馈机制
- [x] 基建的内部完善机制
- [x] 基建的内外合作机制

2 选人方案

> 基建首先要解决的是人的问题，团队有没有适合做基建的人，该如何挑选？

专业技能：

- 不设限语言，全栈最优；
- 不设限领域，眼界开阔；
- 扎实的前端基础知识；
- 组件化、工程化、架构，Devops 等；

综合方面：

- 基础夯实、技术攻坚、学习能力强；
- 拥抱业务、关注细节、执行力强；
- 责任心、心态开放、有同理心；
- 爱折腾、善于总结与分享；

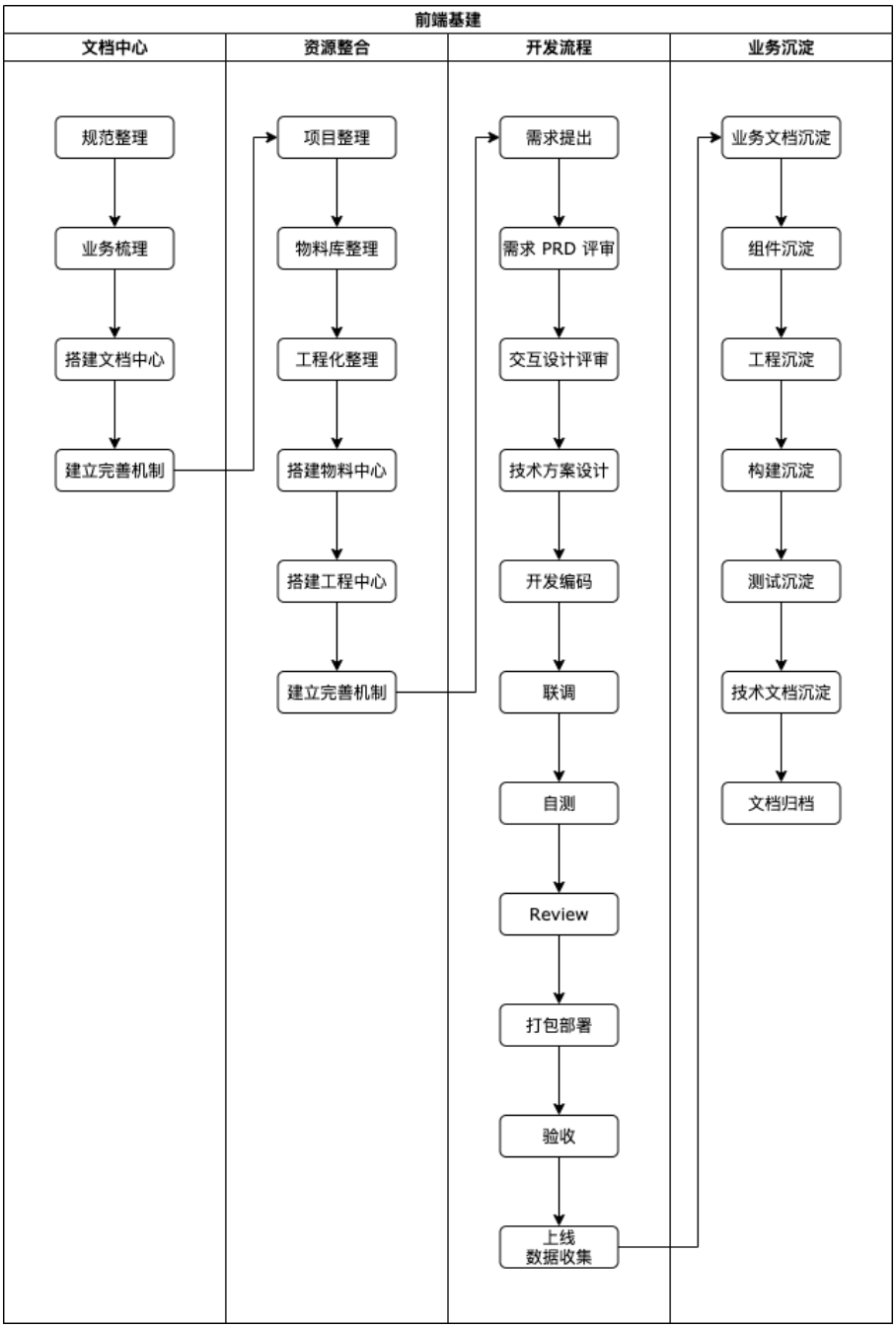
挑选机制：

- 以周为维度的 CodeReview；
- 以月为维度的技术分享；
- 以季为维度的职责调岗；

3 基建规划

> 基建会随着业务的增长变得愈发重要，可视团队情况逐步进行。

常规业务流程下的基建过程：



我们从以下四个部分来描述基建的建设过程：

- 文档中心
- 资源整合
- 开发流程
- 业务沉淀

3.1 文档中心

> 文档中心的搭建要优于其他基建，并在完成业务过程中，不断完善文档中心内容。

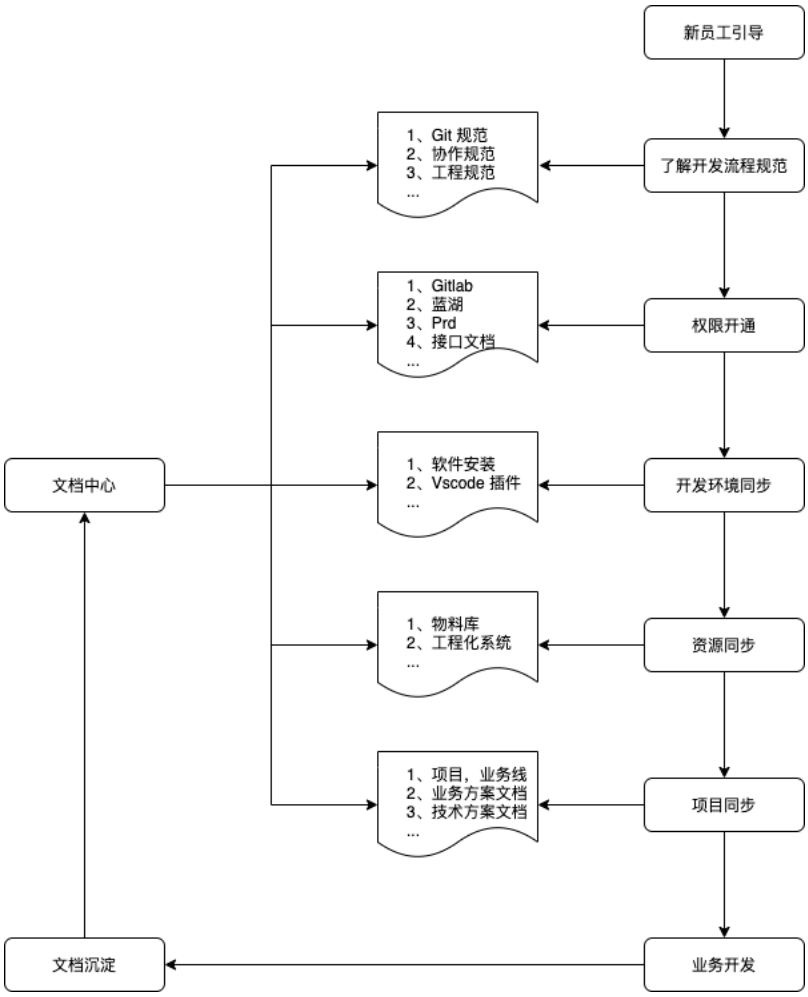
文档中心可作为所有基建资源的唯一入口，如：团队规范、物料中心、工程中心等。

对于刚搭建文档中心的公司，可对现有开发、业务规范进行整理，最终通过以下方式建立线上文档中心：

- 语雀
- Vuepress
- Docusaurus

推荐 Docusaurus（或 Vuepress） 结合版本管理的方式维护前端团队相关文档。

此外，我们可以从新员工引导的角度去思考整个文档中心的内容建设：



文档中心完善机制（以 Docusaurus + Gitlab 为例）：

- 新增与更新文档采用 Gitlab 提交 PR 的形式审核；
- 结合 Gitlab webhook 或者 shell 脚本命令完成部署操作；
- 部署环境搭建视公司情况而定；

3.2 资源整合

> 文档中心搭建完成后，可对公司整体业务进行梳理。

主要从以下维度出发：

- 项目技术栈规整
- 项目依赖规整
- 项目技术债规整
- 物料库资源规整
- 工程化相关规整

规整的目的在于：

- 资源复用
- 问题收集
- 统一标准

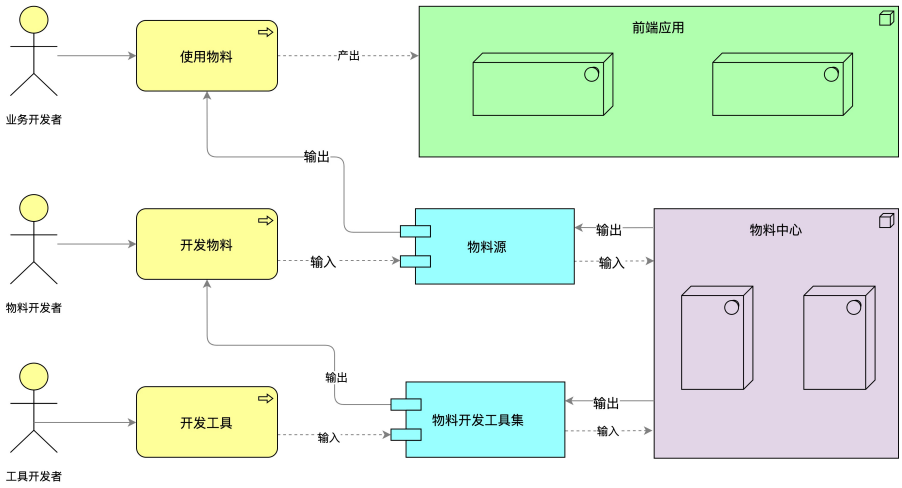
3.3 物料中心

> 物料可用于标准化页面元素组成。

包含但不限于以下内容：

- 基础组件：对 DOM 元素及交互逻辑高内聚的封装，外部可通过传递 Props 的方式对其进行定制；
- 业务组件：由基础组件根据业务需要组合而成；
- 区块：由基础、业务组件组合而成的代码片段，不对外提供配置属性，仅通过拷贝代码的形式放到项目中，通常区块内部还包含事件处理、状态管理、数据请求等逻辑；
- 模板：对区块进行组合，就形成了特定的模板；

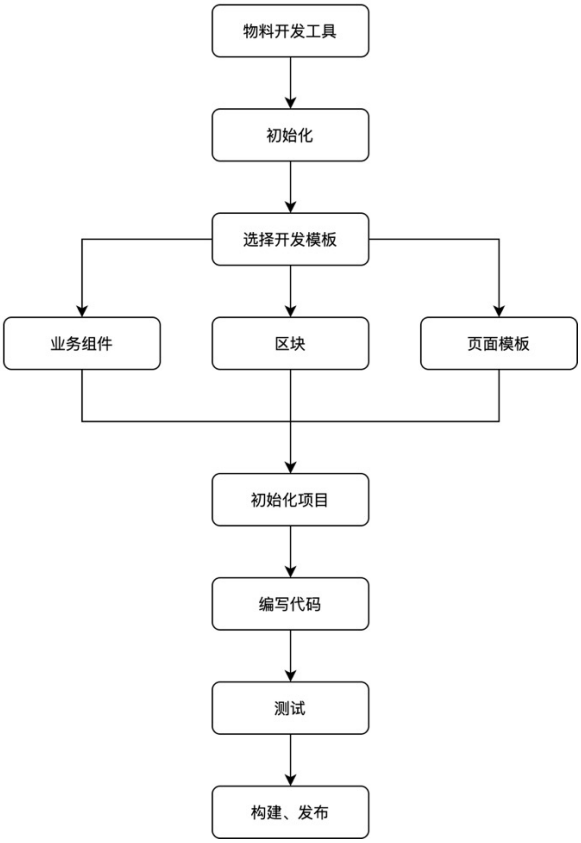
物料用户关系图：



搭建物料中心的意义在于：

- 研发提效
- 提高项目可维护性
- 视觉与交互标准化

物料开发流程：



物料开发工具可以是基于 Node 的命令行工具，视公司具体情况采用不同的模式开发物料，最终沉淀出来的内容有：

- 物料开发脚手架
- 物料统一打包配置
- 物料统一发布配置
- 物料平台，可统一使用 Storybook 作为文档输出载体

社区可参考的物料中心：

- 羚珑
- 飞冰
- Material Center
- antd

3.3.1 组件库

> 组件库是前端领域一个重要的技术单元，为效率、质量、体验服务

- 效率：抽象业务研发中 UI 共同点，避免重复；
- 质量：通过测试和 UI 走查，质量有保证；
- 体验：交互统一，表现一致；

组件库类型：

- PC 端：面向用户和商户的大多都是较为独立的产品，标准化意义其实不大，通常采用社区标准组件库，如 Antd、Element-UI；
- 移动端：组件库定制需求比较多，条件允许可从零搭建；

建设步骤：

- 了解业内相关设计规范；
  - ant design
  - material design
  - atomic design
- 了解 styled system，知道如何从全局角度去设计组件库的主题定制系统；
- 与设计部门制定视觉组件标准，共同创建视觉组件库；
- 与 UED 侧制定完善的组件新增流程，在业务前端研发侧有强同步的约束；
- 从团队情况出发确定技术选型、文档方案、包管理及迭代策略等；

推荐阅读：

- 如何打造一个高效的前端组件库
- 开源的 React Native 组件库
- 开源 React Native 组件库 beeshell 2.0 发布

3.4 工程中心

> 什么是前端工程化？本质上就是将前端开发流程，标准化、规范化、工具化、自动化、简单化。通过规范和工具来提高前端应用质量及开发效率

包括但不限：

- 开发规范
- 脚手架
- 构建工具
- Mock 服务
- 单元测试
- 项目部署
- 可用性建设

3.4.2 开发规范

视团队情况，可以考虑如下开发规范：

- 目录结构
- 命名规则
- 开发工具规范（Eslint、prettier、editorconfig）
- Git commit 规范

业内规范：

- 京东凹凸实验室代码规范
- clean-code-javascript
- Airbnb 团队
- 百度 fex 团队

3.4.3 脚手架

> 脚手架用于快速生成新项目的目录模板，并集成一系列体系化工具的安装，能够提升前端开发人员的效率，减少 copy 操作

常见脚手架：

- vue cli
- create-react-app
- parcel
- yeoman

通常团队需要定制化适合团队开发的脚手架工具，涉及的常用工具库如下：

- childr_process：用于执行 shell 命令
- commander：用于处理控制台命令
- inquirer：用于控制台问答
- semver：用于版本检测提示
- fs-extra：用于 fs 操作询问
- execa：用于执行终端命令
- chalk：用于控制台颜色设置

脚手架可以采用 monorepo 的方式拆分项目配置项粒度，可参考 jslib-base

3.4.4 构建工具

> 构建工具可以让我们更好地自动化处理包括（es6 转换，css、js 压缩，less、sass 的转换等），让我们不再需要手动地去重复做这些事情，解放开发人员的双手，更好地聚焦到业务上的开发，构建本质上就是将代码“串”起来，然后压缩并混淆，最终构建出目标代码文件。

- 基于任务运行:
  - gulp
  - grunt
- 基于模块化打包
  - webpack
  - parcel
  - rollup
  - vite
  - browserify
- 整合型
  - yeoman

3.4.5 Mock 服务

> Mock 指是解决前端在完成页面搭建后，此时需要联调后端接口时，后端接口尚未开发完成，还无法联调时前端可以先按照事先与后端约束好的数据结构来模拟接口数据来走完开发（ 一般是通过后端接

> 口文档比如 Swagger ），实现真正意义上的前后端分离。

主要方式包括但不限于：

- 数据拦截型：通过模拟 Http 请求对相应匹配的接口进行真实请求拦截，返回模拟的数据，如：mockjs；
- json-server：是一个 Node 模块，通过运行 Express 服务器，可以直接把一个 json 文件作为一个具备全 RESTful 风格的 API,并支持跨域、jsonp、路由订制等功能的 web 服务器，也可结合 mockjs；
- 可视化接口管理平台：rap、easy-mock，Yapi；

自研接口平台可考虑：

- 标准化的接口文档
- 提供给前端使用的标准化假数据
- 提供给前端的 TS 类型定义
- 提供给后端使用的单测

3.4.6 单元测试

> 单元测试是工程化中用来确保项目质量及代码质量的一个环节，虽然测试并不能直接地减少 bug，但是可以减少因为反复修改过程中新生成的 bug，因为当你修改代码时，很容易忽略之前设定的一些逻辑，导致系统出现故障。

准备工作：

- 需要先选定一个单元测试框架：jest、Mocha、Karma 等；
- 制定测试规则；
- 约束团队单元测试覆盖率最小值：比如函数覆盖率达到 80%，那么如果每次自动化测试达不到这个条件，项目就发布失败，直到完成目标条件；

遵循规则:

- 假设：如：`test('formatTime() 默认格式，返回时间格式是否正常', () => {})` 指定完成测试所要达成的条件；
- 当：所要执行的操作，如：`date.formatTime(1586934316925)` 执行这个函数的测试；
- 那么：得到的结果，既获得断言 如：`expect(date.formatTime(1586934316925,'yyyy.MM.dd')).toBe('2020.04.15');`

3.4.7 项目部署

前端工程化项目部署涉及到的几个主流工具：

- jenkins：一个可扩展的自动化服务器，可以用作简单的 CI 服务器，具有自动化构建、测试和部署等功能；
- docker：虚拟环境容器,可以将环境、代码、配置文件等一并打包到这个容器中,最后发布应用；
- npm： Node.js 官方提供的包管理工具，主要用来管理项目依赖，发布
- nginx: 可以作为 Web 服务器，也可以作为负载均衡服务器，具备高性能、高并发连接；
- pm2：node 进程管理工具，可以利用它来简化很多 node 应用管中繁琐任务

3.4.8 可用性建设

> 从业务后台服务往上，一直到用户界面，一切都是前端服务，这里面一切用户可能遇到的问题都是前端可用性的范畴。

前端服务可用性包含三个部分：

- 前端代码可用性（测试质量、线上异常）
- 静态资源服务可用性（NGINX、Node、CDN）
- 网络链路可用性（DNS 劫持、网络性能）

可用性的保障分为三个阶段：

- 事前
- 事中
- 事后

保障手段分为三个大类：

- 软的：是指用 “人” 来保障的部分：
    - 流程保障
    - 规范保障
    - 测试保障
- 硬的：是指用 “工程工具” 来保障的部分：
    - 静态代码检查
    - 单测
    - Web 自动化测试
    - 持续集成
    - 线上前端异常监控
    - 业务异常监控
    - 前端服务异常监控
    - 网络异常监控
- 根源的：是整个可用性保障的核心，是指通过 “技术选型” 来让系统更健壮，这里面有两个核心点：
    - 技术选型要简单稳健（技术上避免被动，应具备源码维护能力）
    - 避免出现核心链路上的可用性短板

3.5 私有 npm 仓库

> 在物料中心与工程中心雏形完成后，可以考虑私有 npm 仓库 的搭建。

公司私有 npm 仓库对比公有的优势在于：

- 稳定性
- 私密性
- 安全性

社区 npm 源：

- npm

- cnpm
- taobao

搭建方案：

- cnpmjs.org

基础环境：

- Linux 服务器
- node 环境
- 数据库(Mysql)
- nginx

扩展功能：

- 进程管理（PM2）
- 私有包存储上云（nfs 模块规范）

推荐阅读：

- 分分钟教会你搭建企业级的 npm 私有仓库

3.6 GitLab CI/CD

> GitLab CI/CD 是 GitLab 内置的持续集成/持续交付产品。

GitLab CI/CD 由 CI 和 CD 两部分组成：

- GitLab CI 用于在一个共享仓库内整合团队开发的代码。开发者通过 MR 提交代码，在合并代码前 MR 会触发一个流水线，构建、测试、验证新产生的代码。
- GitLab CD 通过结构化的部署流水线来确保 CI 环节验证通过的代码的交付。

GitLab CI/CD 主要有四个核心概念分别为：

- Pipelines（流水线）是持续集成、持续交付、持续部署的最上层概念；
- Stages（阶段）Stages 串行执行，同一个 Stage 中的 Jobs 并行执行；
- Jobs（任务）是流水线最基础单元，由运行器（Runners）执行，一个 stage 下的所有 jobs 执行成功，则继续，否则失败，流水线终止执行；
- Runners（运行器）是一个轻量的、高度可拓展的用于运行 job 的代理；

GitLab CI/CD 有以下触发方式：

- 代码变更触发（Push/PullRequest）
- 手动页面触发
- 定时触发
- API 触发

运行一个 GitLab CI/CD 流水线的必要条件：

- 在仓库根目录创建一个 `.gitlab-ci.yml` 文件；
- 有可运行的运行器；

`.gitlab-ci.yml` 文件是用于定义流水线的 YAML 文件，主要定义：

- Jobs 的结构和执行顺序；
- 运行器在特定情况下如何运行；
- YAML 语法;

运行器类型：

- 通用运行器，适用于所有 groups 和 projects；
- 组运行器，适用于某个 group 下的所有 projects 和子 group；
- 特定运行器，只适用于特定 project；

推荐阅读：GitLab CI/CD 分析与实践

3.7 monorepo

在项目初期还有一个需要提前设计的就是项目组织方式，下面介绍目前比较推荐的方案：monorepo 策略

> monorepo 是一种将多个项目代码存储在一个仓库里的软件开发策略（"mono" 来源于希腊语 μόνος 意味单个的，而 "repo"，显而易见地，是 repository 的缩写）。

monorepo 优势：

- **代码重用将变得非常容易**：由于所有的项目代码都集中于一个代码仓库，我们将很容易抽离出各个项目共用的业务组件或工具，并通过 TypeScript，Lerna 或其他工具进行代码内引用；
- **依赖管理将变得非常简单**：同理，由于项目之间的引用路径内化在同一个仓库之中，我们很容易追踪当某个项目的代码修改后，会影响到其他哪些项目。通过使用一些工具，我们将很容易地做到版本依赖管理和版本号自动升级；
- **代码重构将变得非常便捷**：想想究竟是什么在阻止您进行代码重构，很多时候，原因来自于「不确定性」，您不确定对某个项目的修改是否对于其他项目而言是「致命的」，出于对未知的恐惧，您会倾向于不重构代码，这将导致整个项目代码的腐烂度会以惊人的速度增长。而在 monorepo 策略的指导下，您能够明确知道您的代码的影响范围，并且能够对被影响的项目可以进行统一的测试，这会鼓励您不断优化代码；
- **它倡导了一种开放，透明，共享的组织文化，这有利于开发者成长，代码质量的提升**：在 monorepo 策略下，每个开发者都被鼓励去查看，修改他人的代码（只要有必要），同时，也会激起开发者维护代码，和编写单元测试的责任心（毕竟朋友来访之前，我们从不介意自己的房子究竟有多乱），这将会形成一种良性的技术氛围，从而保障整个组织的代码质量。

monorepo 劣势：

- **项目粒度的权限管理变得非常复杂**：无论是 Git 还是其他 VCS 系统，在支持 monorepo 策略中项目粒度的权限管理上都没有令人满意的方案，这意味着 A 部门的 a 项目若是不想被 B 部门的开发者看到就很难了。（好在我们可以将 monorepo 策略实践在「项目级」这个层次上，这才是我们这篇文章的主题，我们后面会再次明确它）；
- **新员工的学习成本变高**：不同于一个项目一个代码仓库这种模式下，组织新人只要熟悉特定代码仓库下的代码逻辑，在 monorepo 策略下，新人可能不得不花更多精力来理清各个代码仓库之间的相互逻辑，当然这个成本可以通过新人文档的方式来解决，但维护文档的新鲜又需要消耗额外的人力；

- **对于公司级别的 monorepo 策略而言，需要专门的 VFS 系统，自动重构工具的支持**：设想一下 Google 这样的企业是如何将十亿行的代码存储在一个仓库之中的？开发人员每次拉取代码需要等待多久？各个项目代码之间又如何实现权限管理，敏捷发布？任何简单的策略乘以足够的规模量级都会产生一个奇迹（不管是好是坏），对于中小企业而言，如果没有像 Google，Facebook 这样雄厚的人力资源，把所有项目代码放在同一个仓库里这个美好的愿望就只能是个空中楼阁。

如何取舍：

从逻辑上确定项目与项目之间的关联性，然后把相关联的项目整合在同一个仓库下，通常情况下，我们不会有太多相互关联的项目，这意味着我们能够免费得到 monorepo 策略的所有好处，并且可以拒绝支付大型 monorepo 架构的利息。

使用 monorepo 策略后，收益最大的两点是：

- 避免重复安装包，因此减少了磁盘空间的占用，并降低了构建时间；
- 内部代码可以彼此相互引用；

实践方案：

- 锁定环境：volta 是一个 JavaScript 工具管理器，可以让我们轻松地在项目中锁定 node，npm 和 yarn 的版本。另外还有一个诱人的特性：当您项目的 CLI 工具与全局 CLI 工具不一致时，Volta 可以做到在项目根目录下自动识别，切换到项目指定的版本。

- 复用 packages：workspace；

    - 调整目录结构，将相互关联的项目放置在同一个目录，推荐命名为 packages；
    - 在项目根目录里的 `package.json` 文件中，设置 workspaces 属性，属性值为之前创建的目录；
    - 同样，在 `package.json` 文件中，设置 private 属性为 true（为了避免我们误操作将仓库发布）；

- 统一配置：合并同类项 - Eslint，Typescript 与 Babel（避免在多个子项目中放置重复的 eslintrc，tsconfig 等配置文件）

- 统一命令脚本：scripty 允许将脚本命令定义在文件中，并在 `package.json` 文件中直接通过文件名来引用。这使我们可以实现如下目的：

    - 子项目间复用脚本命令；
    - 像写代码一样编写脚本命令，无论它有多复杂，而在调用时，像调用函数一样调用；

- 统一包管理：Lerna；

- 格式化 commit 信息；

推荐阅读：All in one：项目级 monorepo 策略最佳实践

3.8 Nginx

> Nginx 可以作为 Web、负载均衡服务器，具备高性能、高并发连接等。

负载均衡：

当一个应用单位时间内访问量激增，服务器的带宽及性能受到影响，影响大到自身承受能力时，服务器就会宕机奔溃，为了防止这种现象发生，以及实现更好的用户体验，我们可以通过配置 Nginx 负载均衡的方式来分担服务器压力。

负载均衡的几种常用的方式：

- 轮询
- 权重 weight
- 响应时间来分配

```
// nginx.config
upstream backserver {
    server 192.168.0.1;
    server 192.168.0.2;
    fair;
}

server {
    listen 80;
    server_name localhost;
    location / {
      proxy_pass  http://backserver;
    }
}
```

健康检查：

Nginx 自带 ngx_http_upstream_module（健康检测模块）本质上服务器心跳的检查，通过定期轮询向集群里的服务器发送健康检查请求,来检查集群中是否有服务器处于异常状态。

```
upstream backserver{
    server 192.168.0.1  max_fails=1 fail_timeout=40s;
    server 192.168.0.2  max_fails=1 fail_timeout=40s;
}

server {
    listen 80;
    server_name localhost;
    location / {
      proxy_pass http://backend;
    }
}
```

反向代理：

> 反向代理指的是，当一个客户端发送的请求,想要访问服务器上的内容，但将被该请求先发送到一个代理服务器 proxy,这个代理服务器（Nginx）将把请求代理到和自己属于同一个局域网下的内部服务器

> 上,而用户通过客户端真正想获得的内容就存储在这些内部服务器上，此时 Nginx 代理服务器承担的角色就是一个中间人，起到分配和沟通的作用

反向代理的优势主要有以下两点：

- 防火墙作用：当你的应用不想直接暴露给客户端（也就是客户端无法直接通过请求访问真正的服务器，只能通过 Nginx），通过 nginx 过滤掉没有权限或者非法的请求，来保障内部服务器的安全
- 负载均衡：本质上负载均衡就是反向代理的一种应用场景，可以通过 nginx 将接收到的客户端请求"均匀地"分配到这个集群中所有的服务器上(具体看负载均衡方式),从而实现服务器压力的负载均衡

```
// nginx.config
server  {
  listen 80;
  server_name localhost;
  location / {
    proxy_pass http://127.0.0.1:8000; (upstream)
  }
}
```

Https 配置：

> Nginx 常用来配置 Https 认证，主要有两个步骤：签署第三方可信任的 SSL 证书 和 配置 HTTPS

- 签署第三方可信任的 SSL
- Nginx 配置 https

```
server {
   #ssl参数
   listen               443 ssl; //监听443端口，因为443端口是https的默认端口。80
为http的默认端口
   server_name          example.com;
   #证书文件
   ssl_certificate      example.com.crt;
   #私钥文件
   ssl_certificate_key example.com.key;
}
```

- IP 白名单
- 白名单配置

适配 PC 与移动环境

```
server {
 location / {
        //移动、pc设备agent获取
        if ($http_user_agent ~* '(Android|webOS|iPhone)') {
            set $mobile_request '1';
```

```
        }
        if ($mobile_request = '1') {
            rewrite ^.+ http://m.baidu.com;
        }
    }
}
```

配置 gzip

```
server{
    gzip on; //启动
    gzip_buffers 32 4K;
    gzip_comp_level 6; //压缩级别，1-10，数字越大压缩的越好
    gzip_min_length 100; //不压缩临界值，大于100的才压缩，一般不用改
    gzip_types application/javascript text/css text/xml;
    gzip_disable "MSIE [1-6]\."; // IE6对Gzip不友好，对Gzip
    gzip_vary on;
}
```

Nginx 配置跨域请求：

> 当出现 403 跨域错误的时候，还有 No 'Access-Control-Allow-Origin' header is present on the requested resource 报错等，需要给 Nginx 服务器配置响应的 header 参数

```
location / {
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Methods 'GET, POST, OPTIONS';
    add_header Access-Control-Allow-Headers 'DNT,X-Mx-ReqToken,Keep-
Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-
Type,Authorization';

    if ($request_method = 'OPTIONS') {
        return 204;
    }
}
```

如何使用 Nginx：

- 如何启动 `sudo nginx`；
- 修改`nginx.conf` 配置 (具体看你配置位置) `vim /usr/local/etc/nginx/nginx.conf`；
- 检查语法是否正常 `sudo nginx -t`；
- 重启 `nginx sudo nginx -s reload`；
- 创建软链接(便于管理多应用 nginx)；

推荐阅读：

- 前端 Nginx 那些事
- SSL 证书介绍

3.9 BFF

> BFF 顾名思义就是 Backend For Frontend,用中文解释就是服务于前端的后端。

在平时业务开发中，服务端提供给前端的数据可能不会按照前端需要的方式进行编排或过滤。

针对这种情况，我们可以使用 BFF 将一些前端逻辑转移到中间层，中间层就是 BFF。当前端请求一些数据时，它将调用 BFF 中的 API。

BFF 发挥的作用：

用户体验适配层和 API 聚合层 : 主要负责快速跟进 UI 迭代，对后 端接口服务进行组合、处理，对数据进行：裁剪、格式化、聚合等

在 BFF 层下面是各种后端微服务，在 BFF 上层则是各种前端应用（多端应用），向下调用后端为服务，向上给客户端提供接口服务，后端为 BFF 层的前端提供的的 RPC 接口， BFF 层则直接调用服务端 RPC 接口拿到数据，按需加工数据，来完成整个 BFF 的闭环（以 Node+GraphQL 技术栈为主）

BFF 将执行以下操作：

- 调用相关的微服务 API 并获取所需数据；
- 根据前端展现来处理数据；
- 将格式化后的数据发送到前端；

BFF 有助于简化数据展示，并为前端提供一个目的明确的接口。

BFF 的优点：

- **关注点分离**——前端需求将与后端关注点分离，便于维护。
- **更容易维护和修改 API**——客户端应用程序对 API 结构了解较少，这将使其对 API 中的更改更有弹性。
- **更好的前端错误处理**——大部分时间，服务器错误对前端用户是没有意义的。BFF 可以映射出需要显示给用户的错误，而不是直接返回服务器错误，这将改善用户体验。
- **多种设备类型可以并行调用后端**——当浏览器向 BFF 发出请求时，移动设备也可以这样做。这将有助于更快地获得相应服务的响应。
- **更好的安全性**——某些敏感信息可以被隐藏，并且在向前端返回响应时可以忽略不必要的数据。这种抽象将使攻击者更难以应用程序为目标。
- **共享组件的团队所有权**——应用程序的不同部分可以由不同的团队轻松处理。前端团队可以共享客户端应用程序及其底层资源消耗层的所有权，从而提高开发速度。

BFF 的痛点：

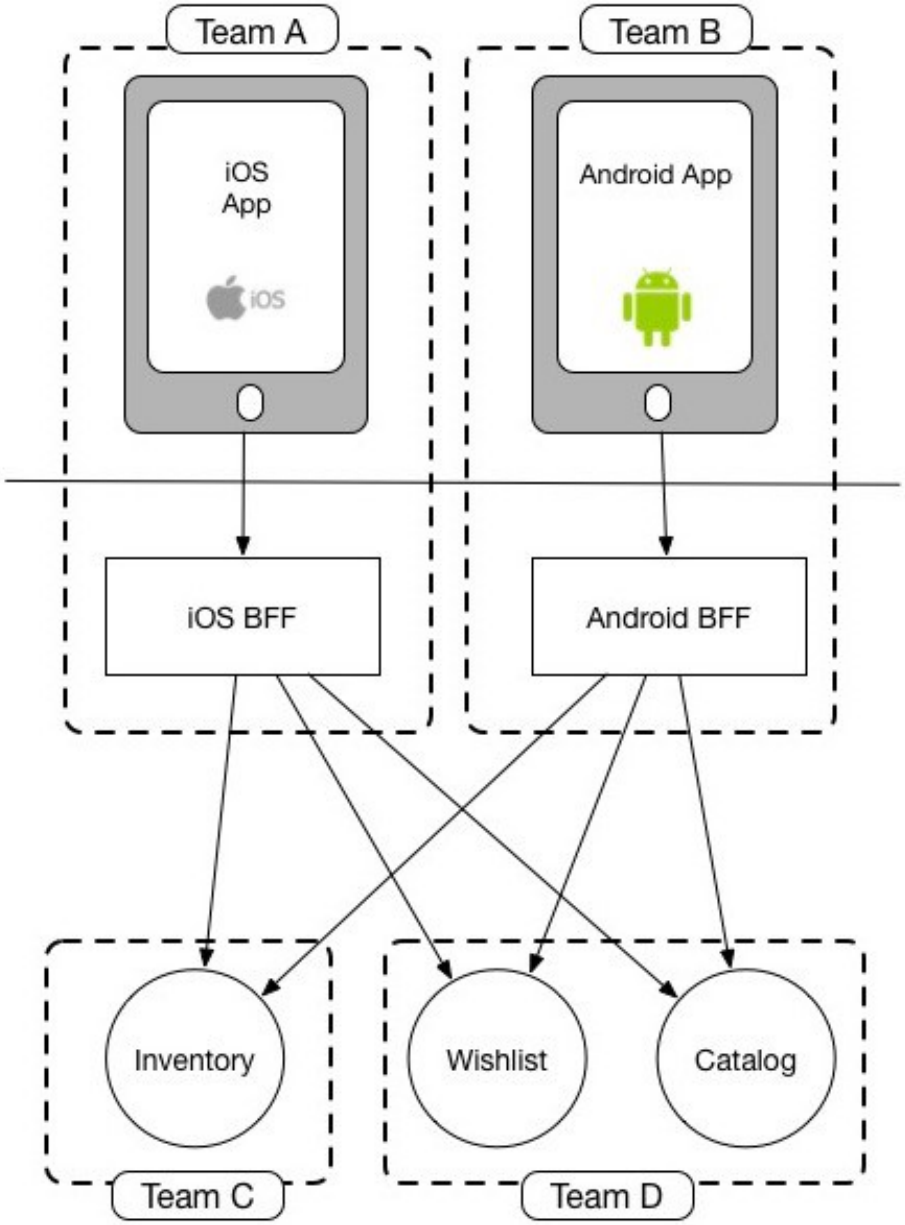
- **重复开发**：每个设备开发一个 BFF 应用，也会面临一些重复开发的问题展示，增加开发成本；
- **维护问题**：需要维护各种 BFF 应用。以往前端也不需要关心并发，现在并发压力却集中到了 BFF 上；
- **链路复杂**：流程变得繁琐，BFF 引入后，要同时走前端、服务端的研发流程，多端发布、互相依赖，导致流程繁琐；
- **浪费资源**: BFF 层多了，资源占用就成了问题，会浪费资源，除非有弹性伸缩扩容；

一些要遵循的最佳做法:

- **避免使用自包含的大而全的 API 实现 BFF**——你的自包含 API 应该位于微服务层。大多数开发人员忘记了这一点，也开始在 BFF 中实现服务级别 API。你应该记住，BFF 是客户端和服务之间的转换层。当数据从服务端 API 返回时，其目的是将其转换为客户端应用程序指定的数据类型。

- **避免 BFF 逻辑重复**——需要注意的一个关键点是，单个 BFF 应该满足特定的用户体验，而不是设备类型。例如，大多数时候，所有移动设备（iOS、Android 等）共享相同的用户体验。在这种情况下，所有这些操作系统的一个 BFF 就足够了。iOS 不需要单独的 BFF，Android 也不需要单独的 BFF。
- **避免过度依赖 BFF**——BFF 只是一个转换层。是的，它也为应用程序提供了一定程度的安全性。但是，你不应该过分依赖它。你的 API 层和前端层应该负责所有的功能和安全方面，而不管是否存在 BFF。因为 BFF 只是填补一个空白，而不是向应用程序添加任何功能或服务。

下图显示了团队划分 BFF 的例子：



推荐阅读：

- BFF 模式：微服务前端数据加载的最佳实践?
- 你学 BFF 和 Serverless 了吗

3.10 serverless

传统 BFF 痛点（可通过 serverless 解决）：

- 包括解决前端需要关心应用的负载均衡、备份冗灾、监控报警等一些列运维部署的操作
- 如何统一管理和运维，提高发布速度、降低运维成本

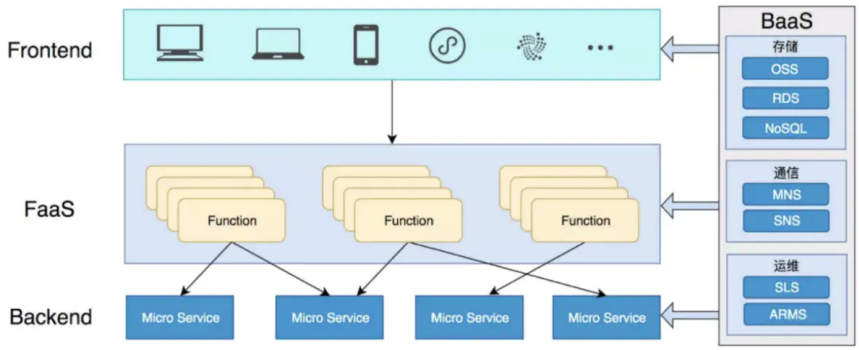
> 我们可以将 Serverless 拆解为 server 和 less 两个单词，从字面上推断词意即为“少服务器的，亦或是无服务器的，弱化后端和运维概念,当前比较成熟的 Serverless 云产品主要有 Amazon Lambda、Google Cloud Function、Azure Function、AliCloud Function Compute、Tencent CloudBase 等

什么是 Serverless?

> Serverless = Faas (Function as a service) + Baas (Backend as a service)

FaaS（Function-as-a-Service）是服务商提供一个平台、提供给用户开发、运行管理这些函数的功能，而无需搭建和维护基础框架，是一种事件驱动由消息触发的函数服务。

BaaS（Backend-as-a-Service）后端即服务，包含了后端服务组件，它是基于 API 的第三方服务，用于实现应用程序中的核心功能，包含常用的数据库、对象存储、消息队列、日志服务等等。



Serverless 的优势：

- 环境统一: 不需要搭建服务端环境, 保持各个机器环境一致 Serverless 的机制天然可复制；
- 按需计费: 我们只在代码运行的时候付费，没有未使用资源浪费的问题；
- 丰富的 SDK: 有完善的配套服务、如云数据库、云存储、云消息队列、云音视频和云 AI 服务等；
- 弹性伸缩: 不需要预估流量、关心资源利用率、备份容灾、扩容机器、可以根据流量动态提前峰值流量；

Serverless 的缺点：

- 云厂商强绑定：它们常常会和厂商的其他云产品相绑定，如对象存储、消息队列等，意味你需要同时开通其他的服务，迁移成本高，如果想迁移基本原有的逻辑不可服用，kennel 需要重构；
- 不适合长时间任务：云函数平台会限制函数执行时间，如阿里云 Function Compute 最大执行时长为 10 min；
- 冷启动时间：函数运行时，执行容器和环境需要一定的时间，对 HTTP 请求来讲，可能会带来响应时延的增加；
- 调试与测试：开发者需要不断调整代码，打印日志，并提交到函数平台运行测试，会带来开发成本和产生费用

Serverless 带来的其实是前端研发模式上的颠覆。相对以往纯前端研发的方式，Serverless 屏蔽底层基础设施的复杂度，后台能力通过 FaaS 平台化，我们不再需要关注运维、部署的细节，开发难度得到了简化，前端开发群体的边界就得以拓宽，能够参与到业务逻辑的开发当中，更加贴近和理解业务，做更有价值的输出。

Serverless 的应用场景：

- 场景 1: 负载有波峰波谷

- 场景 2: 定时任务（报表统计等）
- 场景 3: 小程序开发（云开发）

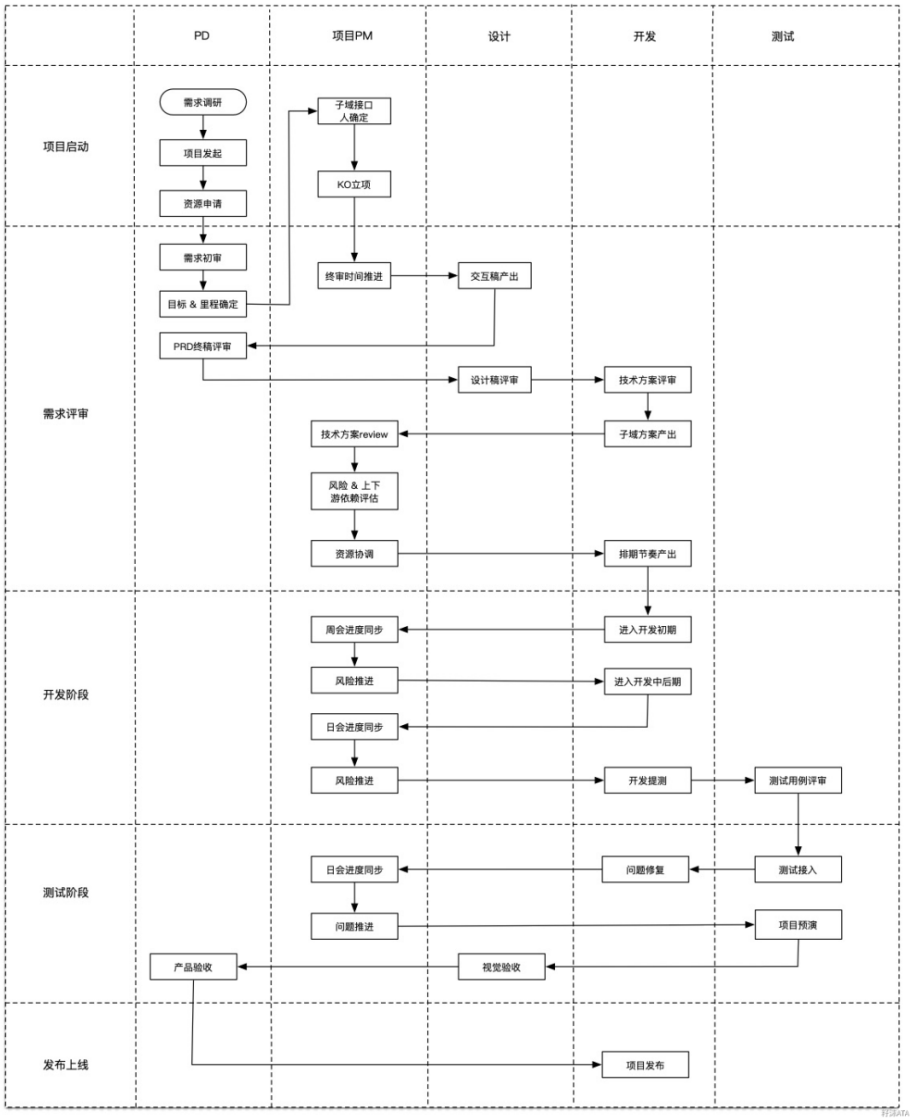
推荐阅读：

- “云”端的语雀：用 JavaScript 全栈打造商业级应用
- Serverless 架构应用开发指南
- Serverless 掀起新的前端技术变革
- 写给前端工程师的 Serverless 入门
- 我们是如何从前端技术进化到体验科技的?

3.11 跨团队协作项目

> 跨团队协作是指在给指定时间约束规范内，不同部门与部门之间、个人与个人之间的协调与配合完成一项明确目标的独立的工作任务。

项目流程图:



推荐阅读：

- 如何做好一个跨团队协作项目

3.12 稳定性监控平台

> 监控 -- 安全生产的第一战线，报警的有效覆盖率、线上问题的发现能力以及如何快速定位问题是监控的核心能力。

安全生产整体目标 1-5-10，1 分钟发现问题、5 分钟定位问题、10 分钟修复问题。

3.12.9 问题发现

多数故障未及时发现，核心问题不是报警有效性的问题，大部分来自于被动通知，线上反馈、舆情或者客诉等，从问题分析来看存在以下几个问题：

- 业务未接入监控，安全意识缺乏以及基础设施并不完备，页面需要手动引入监控 SDK，很多业务在线上完全处于裸奔状态；
- 核心指标未订阅，多数页面引入了监控但是没有订阅报警或者指标订阅不完整，导致线上问题没有第一时间发现；
- 监控指标不完整，从传统前端视角，可能只会关注页面运行时的异常，比如 jserror、接口报错运行等，但是从页面整个运行过程来看，很多监控指标是缺失的，比如 cdn 节点异常、页面加载白屏、页面执行 crash 等问题。

3.12.10 快速恢复

一个完整的开发流程包含： 开发 -> 发布 -> 线上验证回归流程。

如果问题已经发布上线，按照发布流程很难做到 10 分钟恢复，对于问题的核心解法，需要聚焦到页面发布之前的开发阶段和发布阶段，主要涉及两点：

- **发布之前**：完整的自动化测试流程，例如在发布之前对资源异常、JSError 等做检测拦截
- **发布过程**：页面发布过程具有可灰度、可监控、可回滚的完整过程

3.12.11 整体方案

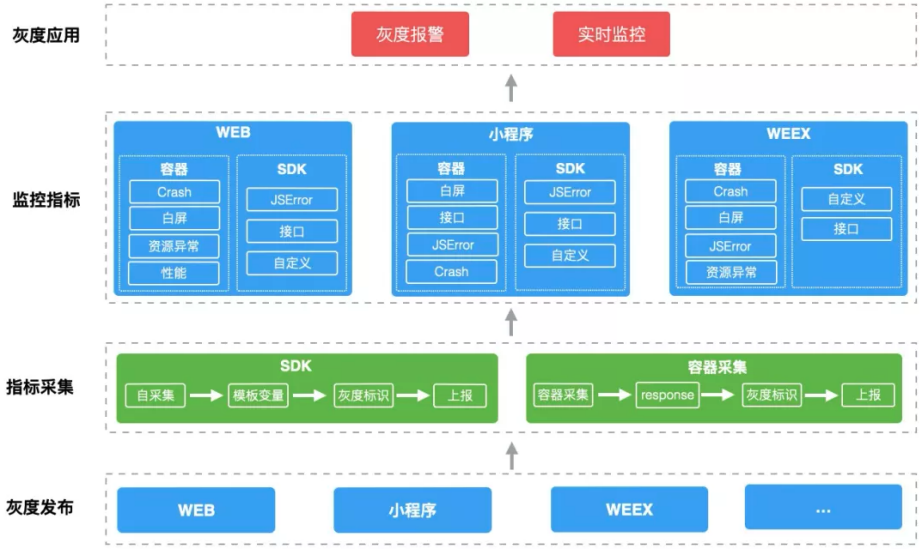
围绕着 1-5-10 安全生产整体目标，针对“问题发现、快速恢复”的问题解决方案如下：

- 接入覆盖：我们需从两个维度来解决接入覆盖率的问题，完善基础设施和业务治理推动的角度；
- 指标统计：在度量开发过程当中，需要各种维度来统计指标数据，比如团队、时间等；
- 度量&红黑榜：度量目标帮助业务、团队做辅助决策，基于页面的原始数据，从原始数据 -> 数据分析 -> 指标度量 -> 业务决策 ，可建立一套指标度量模型，帮助业务快速发现问题；

灰度监控：

快速恢复的核心是需要更快发现问题，更快的对变更进行回滚。历史故障数据，80%左右的线上问题是由变更导致的。但很多故障不是缺少监控导致的，而是新的版本变更引起的问题在整体的错误量不明显，在整体日志层没有区分，导致看到的表面现像是正常的，而忽略了问题。

对于变更的过程的监控 【灰度监控】，需要标识异常的日志是新的版本带来的，用来对比线上版本的监控区分出新版本的错误比例增长以及新引发的问题，解决方案如下：



推荐阅读：

- 百亿业务流量-如何做好稳定性监控

3.13 中后台搭建 PaaS 平台

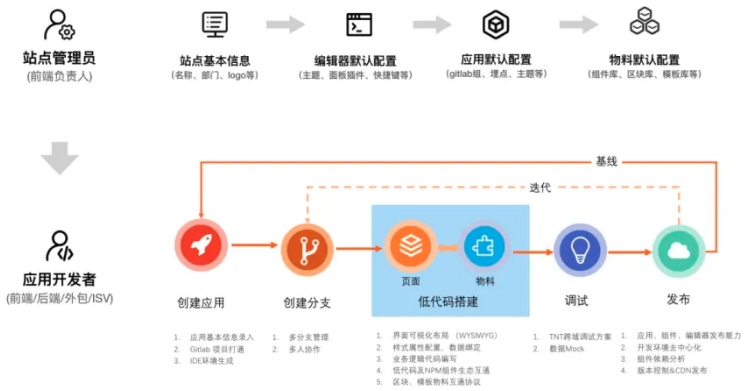
> 当公司的业务达到一定体量的时候，可推行可视化建站方向。

以中后台搭建平台为例，按面向角色的不同大致分为：

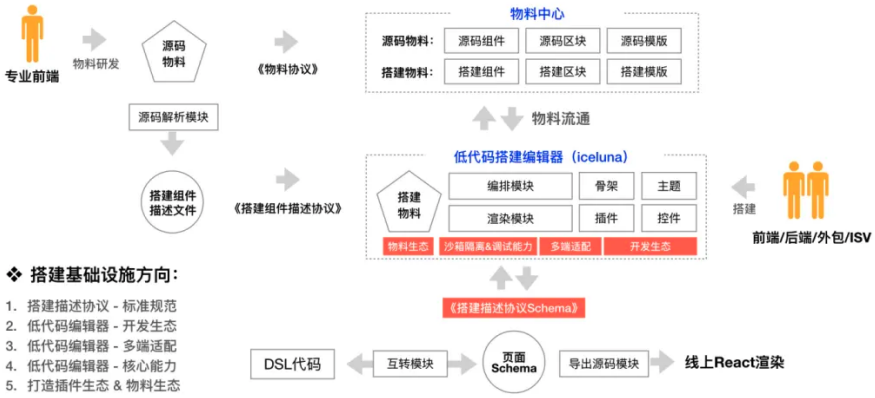
- 面向运营：可视化配置（No-code）的方式进行完整页面搭建，如营销活动页面搭建；
- 面向研发：低代码开发（Low-code）的方式；

搭建平台架构设计：

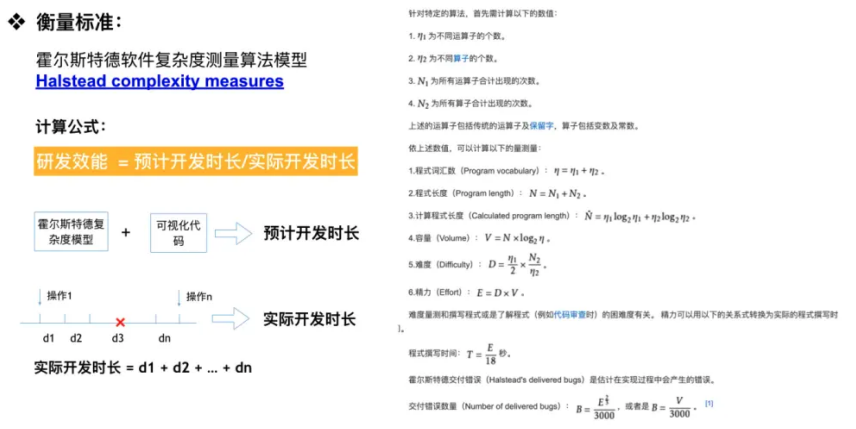
搭建流程：



搭建设施：



搭建效能衡量体系：



推荐阅读：

- 设计实现中后台搭建 PaaS 平台

3.14 微前端

> 微前端出现在我们的视线的次数越来越多，因为 to B 的发展越来越迅猛，导致中后台应用需求激增，如何将多项目集合成一个 web 主体就成为一个问题

> 微前端本质是是一种项目架构方案，是为了解决前端项目太过庞大，导致项目管理维护难、团队协作乱、升级迭代困难、技术栈不统一等等问题，有点类似微服务的概念，是将微服务理念扩展到前端开发的一种应用

微前端的落地方式：

- iFrame
- 路由分发方式：指通过路由将不同业务拆分的子项目，结合反向代理的方式实现（如使用：Nginx）
- Single-SPA
- qiankun

iFrame 局限性：

- 子项目需调整，需要隐藏自身页面中的导航（公共区域）
- iFrame 嵌入的视图控制难，有局限性
- 刷新无法保存记录，也就意味着当浏览器刷新状态将消失，后退返回无效
- iframe 阻塞主页面加载

Single-SPA 的优点：

- 各项目独立开发、部署、迭代，互不影响效率高
- 开发团队可以选择自己的技术并及时更新技术栈。
- 相互之间的依赖性大大降低
- 有利于 CI/CD,更快的交付产品

推荐阅读：

- 微前端那些事

4 辅助工具

- 语雀
- draw.io
- carbon 代码美化工具
- Mdnice markdown 美化
- shields
- Regexr 正则可视化工具
- Canluse 兼容性查询
- SwitchHosts host 管理工具
- nvm
- nrm