深圳技术大学
SHENZHEN TECHNOLOGY UNIVERSITY

本科毕业论文（设计）

题目： 前端监控平台及其 SDK 的设计与实现

姓　　名	王维
学　　院	大数据与互联网学院
专　　业	计算机科学与技术
学　　号	202240292065
指导教师	郑俊虹
职　　称	实验师
提交日期	2024 年 5 月 10 日

深圳技术大学本科毕业论文（设计）诚信声明

本人郑重声明：所呈交的毕业论文（设计），题目《前端监控平台及其SDK的设计与实现》是本人在指导教师的指导下，独立进行研究工作所取得的成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式注明。除此之外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。本人完全意识到本声明的法律结果。

毕业论文（设计）作者签名：王维

日期：2024年5月10日

目录

前端监控平台及其 SDK 的设计与实现

【摘要】随着信息技术的飞速发展，Web 应用在现代社会的各个领域中扮演着越来越关键的角色，用户基数迅速增长，应用范围也日益扩大。这种快速的发展使得 Web 应用的性能和稳定性直接影响到企业的用户留存率和流量状况，一旦出现故障，可能会导致显著的用户流失和业务影响。因此，确保 Web 应用在真实运行环境中的可靠性和效率，变得尤为重要。在此背景下，前端监控系统能够实时监控 Web 应用的运行状况，及时分析和定位系统异常，从而快速响应和处理潜在的问题。本文提供了前端监控平台的基本思路并且给出了一套具有较强可行性和实践性的前端监控开发方案。在 SDK 部分，主要工作包括错误数据、性能数据以及资源加载数据的采集和上报。这些数据的准确采集为后续的分析提供了基础。平台部分则专注于对这些收集到的数据进行深入分析和直观展示，使开发者能够迅速识别并定位潜在问题。通过实时监控和数据分析，该平台能够有效提升应用性能，优化用户体验，并增强应用的稳定性。总体而言，通过本文提出的前端监控平台，开发团队可以更有效地管理和优化 Web 应用，从而减少故障率，提升用户满意度，保障业务连续性。

【关键词】前端监控；Web 应用；系统稳定性；用户体验

Design and implementation of front-end monitoring platform and its SDK

【**Abstract**】With the rapid development of information technology, Web applications play an increasingly critical role across various domains of modern society, experiencing rapid growth in user base and expansion in scope. This swift growth means that the performance and stability of Web applications directly impact an enterprise's user retention rates and traffic conditions. Any failures can lead to significant user attrition and business repercussions. Therefore, ensuring the reliability and efficiency of Web applications in real operational environments has become particularly crucial. In this context, front-end monitoring systems are able to continuously monitor the operational status of Web applications, promptly analyze and pinpoint system anomalies, thereby quickly responding to and addressing potential issues. This paper provides a basic concept of a front-end monitoring platform and offers a feasible and practical front-end monitoring development scheme. In the SDK part, the primary tasks include the collection and reporting of error data, performance data, and resource loading data. The accurate collection of these data lays the foundation for subsequent analysis. The platform part focuses on in-depth analysis and intuitive display of these collected data, enabling developers to quickly identify and locate potential problems. Through real-time monitoring and data analysis, the platform can effectively enhance application performance, optimize user experience, and strengthen application stability. Overall, the front-end monitoring platform proposed in this paper enables development teams to more effectively manage and optimize Web applications, thereby reducing failure rates, enhancing user satisfaction, and ensuring business continuity.

【**Key words**】Front end monitoring； Web applications； system stability； User experience

1. 绪论

本章主要介绍了前端监控平台的研究背景、研究 Web 前端监控的意义以及国内外研究现状。其次简要介绍前端监控平台在实际生产中解决的问题。

1.1 研究工作的背景与意义

前端监控在当今互联网时代的前端开发中扮演着至关重要的角色，其意义和重要性不容忽视。随着互联网应用的日益复杂和用户需求的不断增长，前端监控作为一种技术手段，为开发人员提供了关键的数据和洞察，有助于优化用户体验、提升应用性能、保障系统稳定性，进而实现持续的技术提升和业务发展。

首先，前端监控对于前端开发的意义在于实现对用户体验的全面把控。作为用户接触应用的第一入口，前端界面的流畅度、交互性以及加载速度直接影响用户对产品的感知。通过监控前端性能指标如页面加载时间、交互响应速度、错误率等，开发团队可以全面了解用户在不同环境下的真实体验，及时发现和解决存在的问题，从而提升用户满意度和忠诚度。

其次，前端监控也为持续优化和迭代提供了数据支持。通过收集用户行为数据、页面加载数据等，开发团队可以进行数据分析和挖掘，发现用户习惯、行为模式、热点等信息，为产品的优化和升级提供指导。基于监控数据分析，团队可以有针对性地进行功能改进、性能优化，不断提升产品质量和竞争力。

总的来说，前端监控对于前端开发具有重要意义。它不仅能够帮助开发团队及时发现和解决问题，提升用户体验和应用稳定性，更能够为持续优化和创新提供数据支持，推动产品不断发展。因此，在当前互联网发展的背景下，设计与实现一套高效可靠的前端监控平台及其 SDK 显得尤为重要，将有助于提升前端开发的效率和质量，为用户带来更优秀的应用体验。

1.2 国内外研究现状

在国内，许多公司已经开始关注 Web 前端异常监控的重要性，并着手研究相关的技术方案或平台。这些公司致力于通过监控前端页面的异常情况，及时发现和解决潜在的问题，以提高用户体验和系统稳定性。在国外，也有一些典型的

技术方案或平台用于 Web 前端异常监控。这些方案通常包括实时监测、错误报告、性能分析等功能，能够帮助开发人员快速定位和解决问题，提升应用程序的可靠性和性能。

国内一些互联网公司也推出了自家的监控平台，例如阿里云 ARMS，腾讯云前端性能监控（RUM）。国外比较典型的有 Sentry，Google Analytics[1]，接下来对这些平台及其涉及到的技术做简要介绍与分析。

1) 阿里云 ARMS

阿里云的 ARMS 是一个云原生的可观测平台，它涵盖了前端监控、应用监控和云拨测等多个模块。这个平台的设计宗旨是为了适应现代复杂的云原生环境，它能够覆盖从浏览器、小程序、APP 到分布式应用和容器等各种不同的可观测环境和场景。ARMS 能够对整个应用程序的性能进行监控，从前端到后端，无论是在浏览器还是服务器上运行的代码。通过集成的跟踪系统，ARMS 可以帮助开发者追踪请求在不同系统和服务之间的流动，从而快速定位问题。通过智能化的监控和诊断，ARMS 可以显著提高监控的效率，减少运维工作量。

然而，ARMS 也存在一些不足点。首先，作为一款综合性的监控平台，其配置和设置可能相对复杂，对于初学者来说可能需要一定的学习成本。其次，虽然 ARMS 提供了全面的监控功能，但对于某些特定场景或特殊需求，可能需要额外的定制化开发。最后，作为云服务产品，其价格可能会随着使用规模的扩大而增加，这可能会对中小企业造成一定的经济压力。

2) 腾讯云前端性能监控（RUM）

腾讯云的前端性能监控（RUM）是一个针对 Web 和小程序领域的综合性解决方案，旨在优化用户体验并提高页面性能。通过在用户浏览器中植入轻量级的监控脚本，RUM 能够实时监控页面加载速度、接口调用效率以及内容分发网络（CDN）的性能表现。它特别注重真实用户监控（RUM），提供反映实际使用情况的数据，以区别于传统的实验室监控数据。收集到的性能数据被发送至云端服务器进行深入处理和分析，随后生成直观的报告和优化建议。此外，RUM 还具备警报系统功能，能够在性能指标低于预设阈值时及时通知开发团队，以便快速

响应和处理潜在的性能问题。总体而言，腾讯云的前端性能监控（RUM）通过其全面的功能和技术手段，为开发者提供了一套有效的工具集，帮助其监测、分析和优化前端性能，确保最终用户获得流畅且高效的体验。

3) Google Analytics

Google Analytics 是一个强大的网站和应用分析工具，它通过在网页或应用中加入跟踪代码来收集详细的用户互动数据。这些数据包括访问来源、用户行为、页面浏览量、停留时间等关键指标。随后，Google Analytics 利用先进的数据处理技术对这些信息进行汇总、分析和可视化处理，以直观的报表形式呈现给使用者，帮助他们理解用户如何与网站或应用互动。此外，它还提供个性化首页、移动应用访问、效果衡量和报告等功能，以及强大的 API 支持，让用户可以根据自己的需求定制数据分析和集成。通过 Measurement Protocol，Google Analytics 能够接受更广泛的数据类型，从而为用户营销策略的优化和品牌形象的提升提供了坚实的数据支撑。简而言之，Google Analytics 通过其丰富的功能和技术手段，为网站和应用提供了一个全面的数据收集、分析与优化平台。

4) Sentry

Sentry 是一个强大的应用错误跟踪系统，它专注于监控和报告软件应用中的错误。其核心功能是提供实时的错误报告和分析，当线上应用程序出现 bug 时，Sentry 能够立即发现并通知相关的责任人员。通过详细的错误报告和上下文信息，Sentry 帮助开发者快速定位问题。此外，Sentry 支持几乎所有主流的开发语言和平台，并且提供了 SaaS 版本，免费版支持每天 5000 个事件。在前端监控方面，Sentry 能够帮助监控到 Vue 中的错误和异常。总的来说，Sentry 为开发者提供了一个强大的工具，以便更好地理解和解决应用中的问题，从而提高软件的稳定性和用户体验。

当前的第三方前端监控系统在实际应用中展现出了一些明显的局限性。首先，这些系统往往是为了满足特定企业的需求而设计的，因此它们与企业的业务逻辑紧密相连，导致了高度的业务耦合。

其次，目前这些前端监控系统大多数并未开放源代码，这意味着它们的内部

工作机制对于外界来说是一个黑盒。这种情况在一定程度上限制了整个社区对这些系统的理解和改进，也减少了它们在 Web 前端异常监控与报警领域的研究和应用价值。开源是推动技术进步的重要动力，通过共享代码，开发者可以相互学习，共同改进系统，但在目前的状况下，这种协作和共享的机会被大大限制了。

此外，由于缺乏透明度，这些系统很难被其他企业或开发者完全信任和接受。在没有能力深入理解系统工作原理的情况下，用户可能会对系统提供的监控数据的准确性和可靠性持有保留态度。

综上所述，尽管现有的第三方前端监控系统在一定程度上满足了部分企业的需求，但它们的业务耦合性、代码入侵性强以及缺乏开源的问题，限制了它们在 Web 前端异常监控与报警领域的广泛应用和发展。为了更好地推动这一领域的研究和创新，有必要考虑开发更加开放、通用、易于集成的前端监控解决方案。

1.3 本文结构安排

第一章：绪论。本章主要介绍了前端监控平台的开发背景，阐述其在现代网络服务中的重要性，并分析其对提升服务质量、保障系统可靠性的贡献。随后，分析国内外典型的前端监控平台，评估这些平台的功能特点。

第二章：相关技术与理论介绍。为了有效地实现前端监控，需要了解和应用一系列技术和理论，本章主要介绍在实现前端监控过程中涉及的关键技术。

第三章：需求分析。本章主要对前端监控 SDK 和平台进行需求分析，包括功能性需求和非功能性需求。通过需求分析，我们能够清晰地识别出项目的目标和约束条件，这对于后续的开发工作至关重要。它帮助我们理清开发思路，制定合理的开发计划，确保开发工作能够按照既定目标高效推进。

第四章：监控 SDK。本章主要介绍 SDK 的实现，针对每个模块每个功能的实现思路。以及涉及到的相关技术，最后给出伪代码加以说明参考。

第五章：监控平台。本章主要介绍监控平台的实现，主要内容包括用户应用模块，异常数据模块，页面性能模块，map 映射文件模块，页面访问模块。每个部分使用文字描述加界面截图的方式阐述。

第六章：总结与展望。对本文主要内容做简要总结，以及分析有待优化和提升的地方。

2. 相关技术与理论介绍

2.1 相关理论介绍

2.1.1 Performance API

Performance API[2]是 Web 性能 API 的一部分，它提供了一个可以让开发者测量他们网页和应用性能的方式。这个 API 允许开发者获取到与性能相关的信息，例如页面加载时间、解析 DOM 所需的时间、资源加载时间等。通过这些信息，开发者可以识别和解决性能瓶颈，从而优化用户体验。

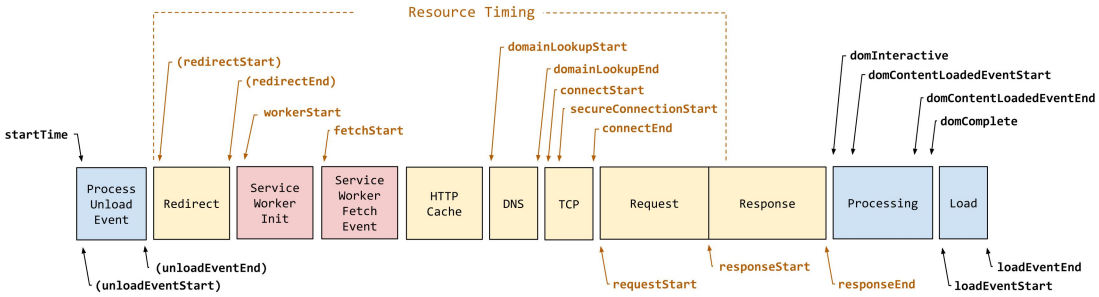

图 2-1 浏览器文档加载过程各节点

Performance API 包含了许多子接口。Resource Timing API 可以获取和分析应用资源加载的详细网络计时数据，应用程序可以使用时间度量标准来确定加载特定资源所需要的时间。Navigation Timing API 提供了可用于衡量一个网站性能的数据，如图 2-1 所示，与用于相同目的的其他基于 JavaScript 的机制不同，该 API 可以提供更有用和更准确的端到端延迟数据。User Timing API 接口允许开发者在浏览器性能时间线中创建针对特定应用的时间戳，可用于自行测量操作耗时。

在 Performance 中，所有的性能指标都使用 performance entry 对象来表示，其中 name 属性表示该性能条目的名称。entryType 表示该条目的类型，可能是与导航相关的性能指标，可能是与资源加载相关的性能指标，也有可能是用户自定义的性能指标。duration 表示该事件持续的时间。startTime 表示该指标上报的时间。

PerformanceObserver 用于监测性能度量事件，在浏览器的性能时间轴记录新的 performance entry 的时候将会被通知。可以借助这个 API 采集各种性能数据。

2.1.2 核心 Web 指标

核心 Web 指标（Core Web Vitals）[3]是 Google 推出的一组特定的页面性能指标，旨在帮助开发者量化用户体验的质量。这些指标关注于网页的加载性能、互动性和视觉稳定性，是衡量网站整体用户体验的重要工具。核心 Web 指标包括以下三个主要指标:

Largest Contentful Paint (LCP)：最大内容绘制（LCP）衡量的是在视口中最大内容元素（如图片、视频或大的文本块）渲染完成的时间。这个指标反映了页面内容对用户可见的速度，一个好的 LCP 应该在 2.5 秒内完成。

First Input Delay (FID)：首次输入延迟（FID）衡量的是用户第一次与页面交互（例如点击链接、按钮等）到浏览器能够响应该交互的时间。这个指标衡量的是用户互动的响应性，理想的 FID 应该小于 100 毫秒。

Cumulative Layout Shift (CLS)：累积布局偏移（CLS）衡量的是页面加载期间所有意外布局偏移的总和。布局偏移发生在页面上的元素在没有用户输入的情况下突然改变位置。这个指标评估的是页面的视觉稳定性，一个好的 CLS 评分应该小于 0.1。

核心 Web 指标的目标是提供一套清晰、统一的标准，让开发者能够理解和改进网页的用户体验。通过优化这些指标，可以显著提升网站的性能，提高用户满意度，同时也有助于改善网站的搜索引擎排名。

2.2 开发技术

2.2.1 Node.js

Node.js[4]是一个开源且跨平台的 JavaScript 运行时环境，它允许开发者在服务器端运行 JavaScript 代码。Node.js 是基于 Chrome V8 JavaScript 引擎构建的，这

意味着它能够提供非常高的执行速度。Node.js 的设计哲学是提供一种简单、轻量级的方式来构建可伸缩的网络应用。

Node.js 使用事件驱动和非阻塞 I/O 模型来处理多个连接，这使得它在处理高并发请求时非常高效。这种模型意味着 Node.js 服务器在等待 I/O 操作完成时不会被阻塞，而能够处理其他请求，从而提高了吞吐量和性能。

Node.js 不仅仅可以开发服务端应用，在前端工程化中扮演着基础和核心的角色，原因在于它不仅使得 JavaScript 能够在服务器端运行，而且提供了强大的工具和生态系统，支持前端开发的自动化、模块化和组件化等现代开发实践。因此，Node.js 可以被视为现代前端开发工程化的基础和关键组成部分。

2.2.2 Egg

Egg 是基于 Node.js 的 Web 后端框架，Egg 的核心设计理念是“约定优于配置”，它通过一套统一的约定来减少团队内部的沟通成本和学习成本，使得开发者可以更加高效地协作。尽管 Egg 强调约定，但它并不牺牲扩展性。Egg 提供了高度的可扩展性，允许团队根据自己的需求定制框架。因其简约的设计理念和高扩展性，因此选择 Egg 作为后续的后端开发框架。

2.2.3 Monorepo

Monorepo（单一仓库）是一种代码管理策略，它指的是在一个单一的版本控制仓库中维护多个项目或代码库。这些项目可能包括不同的库、应用程序、服务等，它们可以相互独立，也可以有依赖关系。Monorepo 与多仓库（Multi-repo）策略相对，后者是指为每个项目或代码库使用单独的版本控制仓库。

在 Monorepo 中，所有项目都共享同一个版本管理系统，代码回滚，历史查看都更加方便。不同项目之前共享和复用代码也更加方便，因为所有代码都在一个仓库中。得益于 Monorepo 策略，不同的项目可以使用统一的构建、测试和部署流程。

Pnpm 内置了对 Monorepo 的支持，即工作空间（Workspace），可以通过 wordspace 协议引用同一个工作空间内的 packages，这使得多包协同开发变得简

单，无需发布依赖即可本地调试。发布时，Pnpm 会根据 Semver 版本范围规范替换 worksapce 依赖，这样开发时既可以使用本地 workspace 中的依赖，包的使用者也可以像常规的包那样使用。

2.2.4 Vue

Vue.js[5]是一款流行的开源 JavaScript 框架，用于构建用户界面和单页应用程序（SPA）。Vue 是由尤雨溪（Evan You）于 2014 年创建的，其设计目标是通过尽可能简单的 API 提供响应式的数据绑定和组合的视图组件。Vue 的核心库专注于视图层，易于学习且与其他库或已有项目集成，同时也完全能够为复杂的单页应用提供驱动。

Vue 是一个 MVVM（Model-View-ViewModel）框架，这种模式通过分离应用的逻辑表示和呈现层来提高代码的可维护性和可扩展性。MVVM 主要由三部分组成：Model（模型）、View（视图）和 ViewModel（视图模型）。MVVM 的核心优势在于其双向数据绑定特性，这意味着 ViewModel 中的数据更改会自动反映在 View 上，同样地，View 中的更改也会自动同步到 ViewModel 中。这减少了手动操作 DOM 的需求，使得开发者可以更专注于业务逻辑的实现。此外，由于 View 和 Model 是松耦合的，这也使得单元测试和代码的维护变得更加容易。

3. 需求分析

需求分析能够明确开发目标，避免需求偏差，是软件开发过程中不可或缺的一部分，本节主要对监控 SDK 及其平台进行需求分析，列出基本需求与开发过程中的注意点。

3.1 监控 SDK 需求分析

本小节阐述了 SDK 的基本需求，从 SDK 整体流程到设计思路再到大致功能。其中 3.1.1～3.1.3 为非功能性需求，3.1.4～3.1.6 为功能性需求。捋清了 SDK 开发思路，确定了基本技术方案。

3.1.1 SDK 整体流程

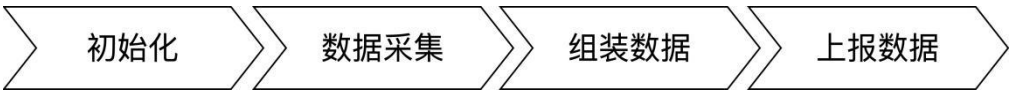

图 3-1 SDK 流程

监控需要做的事情如图 3-1 所示，类似一条流水线。初始化部分负责初始化用户传入的整体配置，初始化各个监控程序，使监控准备就绪。数据采集部分负责在应用运行的过程中采集各种数据，如运行时错误，页面性能指标。组装数据部分负责将原始数据需要包装成特定的数据格式，还需要增加上下文信息，这是上报前的最后一步。最后根据上报策略将数据上报。

为了保证 SDK 可扩展性与维护性，需要将上述流程进行抽象解耦，图 3-2 抽象出五种角色 Monitor，Builder，Sender，ConfigManager，Client。

Monitor 采集器负责采集各种原始数据，Monitor 有若干个，每个 Monitor 对应一个功能，比如采集 JS 错误是一个 Monitor，采集 HTTP 错误又是一个 Monitor。

Builder 数据构建器负责将原始数据包装成特定数据格式的数据，同时添加上下文，比如对于一个 JS 错误而言，上报前需要附加上浏览器类型，浏览器版本，当前时间，等等。

Sender 发送器负责发送逻辑，在其内部维护一个缓存队列，按照一定的队列长度或者缓存时间来上报数据，同时会将开放一些配置项来自定义队列长度和缓存时间，也对外暴露立即上报和清空队列的方法。

ConfigManager 配置管理器，负责管理配置逻辑，提供一套默认配置，也支持用户传入配置。在配置完成和变化时发出通知，所以 ConfigManager 应该支持订阅。

Client 实例主体负责串联 Monitor、Builder、Sender、ConfigManager，构建起整个流程，同时提供生命周期以扩展 SDK 功能。

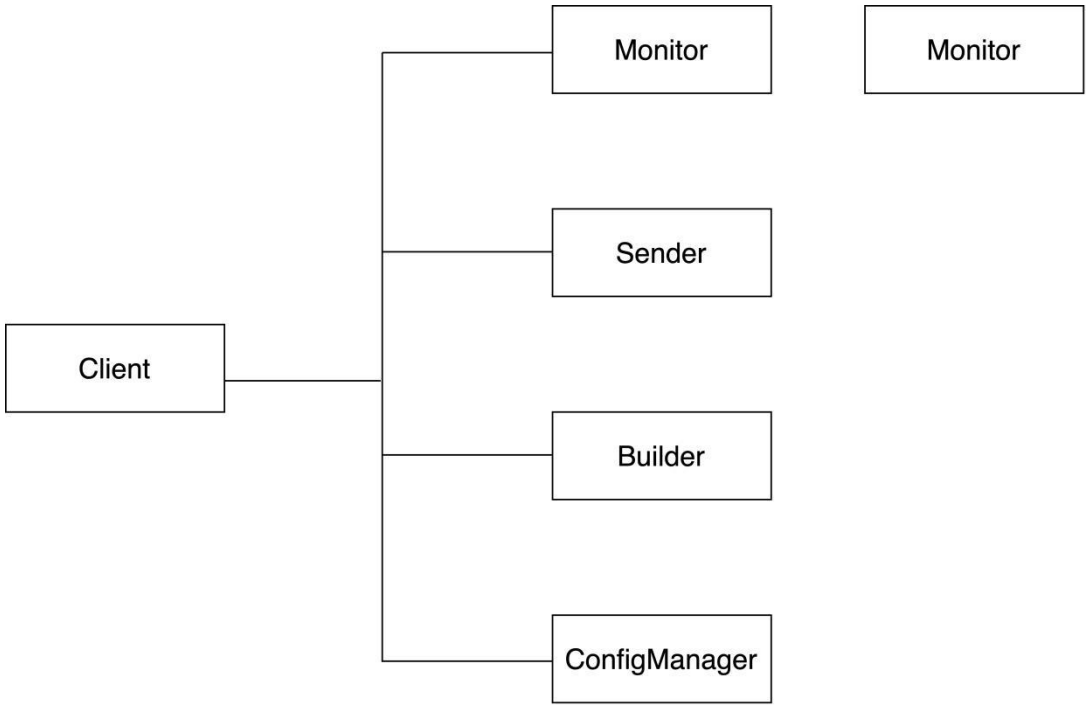

图 3-2 SDK 逻辑解耦

角色之间足够抽象，相互独立，比如 Monitor 只负责采集数据，并不知道数据最终格式，如何上报，Builder 只负责组装数据，交给 Client，之后就是 Sender 负责上报数据。这样的抽象对 SDK 本身的扩展性是极其友好的，同时也降低了开发成本。

3.1.2 SDK 生命周期

在上述流程中，希望在不同的阶段执行各种操作，这些操作大致分为 SDK 内部功能和用户自定义操作，用户自定义操作在 SDK 开发过程中是未知的，对于某些内部功能也不希望与其他代码耦合。所以，要求 SDK 内核提供完善规范的生命周期来应对这两种需求，向外部开放一些介入 SDK 流程的接口。这样在保持内核精简的情况下，也不失扩展性。

按照是否影响 SDK 工作流程可以将生命周期分为处理类生命周期与回调类生命周期。处理类生命周期执行后返回特定含义的值，决定是否继续执行之后的流程，比如在上报前这个生命周期中，如果返回了 false 将阻止上报。回调类生命周期只执行，不影响流程。

3.1.3 插件化

插件化架构又称微核架构，指的是软件的内核相对较小，主要功能和业务逻辑都通过插件实现。插件化能提高软件可扩展性，保持内核稳定，降低程序复杂度[6]。对于一个监控 SDK 来说，插件化也是必不可少的，它可以带来以下好处。

自行扩展，SDK 基本功能完成后，面临新的需求，比如需要增加一类新数据的采集，这时可以单独写一个插件来满足这个需求，而不需要迭代 SDK 本身，这种不修改源代码完成需求的方式也符合开放封闭原则。在构建 SDK 流程中，插件化也可以起到解耦逻辑的作用，比如可以将 Monitor 作为插件接入主体程序中。

按需加载，SDK 默认会集成大部分功能，但有时用户并不需要那么多功能，如果部分功能都以插件的形式提供，那么这些功能就是可插拔的，用户可以自行选择是否添加该功能。

3.1.4 错误采集

错误采集是打造前端监控系统最关键的第一步。采集的准确性，是平台不漏报不误报的核心，日志信息的完整性，直接影响开发者能否获取到关键错误代码来解决问题。 对于前端而言，一般的监控指标都会包含 页面性能、JS 异常，

资源加载异常；而前端页面通常还需要请求后端服务器数据，因此需要把 API 请求失败也监控起来，从而覆盖页面访问的全过程。

3.1.5 性能监控

如果页面加载速度过慢，用户可能会在页面完全加载之前离开，这会导致网站的跳出率升高。高跳出率不仅影响用户体验，还可能对业务目标产生负面影响，比如降低转化率。所以性能监控是必须的，持续监控和预警页面性能的状况，并且在发现瓶颈的时候指导优化工作。

3.1.6 数据上报

数据上报是最后一环，也是最重要的一环。上报的准确率和稳定性关乎数据完整性。需要设计一套合理完善的上报流程和技术方案，在不影响业务逻辑的前提下，保证数据的完整性，不错报不漏报。

3.2 监控平台需求分析

3.2.1 数据存储与处理

在前端监控平台中，数据存储和处理与分析是确保平台有效运作的关键环节。首先，数据存储需要采用高性能、可扩展的数据库系统，通过精心设计的数据模型和优化的存储结构，保证大数据量的高速读写及查询效率。同时，实施定期的数据备份和恢复策略，确保数据的安全性和可恢复性。

SDK 直接上报的数据存在数据量大，动辄几兆、十几兆；没有分类，同一类型的错误只是时间维度不同，没必要都持久化；可能存在一些非法数据不利于数据聚合，也给服务器造成了很大的压力。所以对于原始数据需要进行数据清洗，首先去除一些无用信息，非法数据，对数据进行聚合，降低数据复杂性。

此外，整个数据处理流程严格遵守数据保护法律法规，对敏感数据进行脱敏处理，保护用户隐私。总之，通过高效的数据存储和深入的数据处理与分析，前端监控平台能够提供实时监控、即时警报和深度洞察，帮助优化网站性能和提升用户体验。

3.2.2 数据可视化

数据可视化[7]是使用图表、图形或地图等可视元素来表示数据的过程。该过程将大量复杂的数值数据转化为更易于处理的可视化表示。数据可视化工具可自动提高视觉交流过程的准确性并提供详细信息。在监控平台中，数据可视化主要展示的内容有：页面性能数据，错误数据，页面访问数据。还应具备基于时间筛选数据可视化的功能。

4. 监控 SDK

本章主要介绍 SDK 的实现，首先按照上节给出的 SDK 整体流程，完成整体流程的开发。接下来，按照功能模块逐个阐述，阐明模块的工作职责，对应地，会给出各个模块的关键代码，重要代码会重点分析其原理。

4.1 ConfigManager 配置管理

ConfigManager 提供默认配置并且接收用户配置，与默认配置进行深度合并。对必要配置进行非空检查及合法性校验。配置可能在后续被更改，所以提供监听接口，方便在配置更改时发出通知，广播最新的配置。

表 4-1 ConfigManager 部分伪代码

ConfigManager 部分伪代码

```
export default class ConfigManager {
  // 使用默认配置初始化
  private config: IGlobalConfig = cloneDeep(DEFAULT_CONFIG);
  private events: Subscribe[] = [];
  /**
   * 更新配置
   * @param customConfig 配置对象
   * @param isOverwrite 是否覆盖默认配置，默认是递归合并配置
   */
  set(customConfig: Partial<IGlobalConfig>) {
    this.config = merge(this.config, customConfig);
    for (let i = 0; i < this.events.length; i += 1) {
      // 通知所有的监听方
      const notice = this.events[i];
      try {
        notice(this.config);
      } catch {}
    }
  }

  checkConfig() {
    // 省略一系列的检查
    ......
    return true or false;
  }
```

```
26.
27.   /**
28.    * 获取配置项
29.    * @param path 目标值的路径
30.    * @returns 路径对应的值
31.    */
32.   get<T extends DeepKeys<IGlobalConfig>>(path: T): DeepType<
  IGlobalConfig, T> {
33.     return get(this.config, path, get(DEFAULT_CONFIG, path))
  ;
34.   }
35.   // 监听配置变化
36.   onChange(fn: Subscribe) {
37.     this.events.push(fn);
38.   }
39. }
```

表 4-1 展示了 ConfigManager 部分伪代码，其中 set 方法将用户传入的配置与初始化默认配置进行深度合并，可以尽可能地让用户少传入参数，同时将配置的变化通知到监听方。checkConfig 则是进行配置检查，例如检查上报地址是否合法，必传参数是否空白等等。另外参数的检查并不会在对象实例化时进行，而是由 Client 在初始化整个程序时检查配置。某些配置是嵌套的对象结构，为了方便获取，提供 get 方法支持以字符串路径的方式读取某个配置。

4.2 Builder 数据组装

Builder 内需要将原始数据附加上用户配置的基本信息和环境相关的数据，比如浏览器类型，操作系统类型。

表 4-2 解析操作系统类型伪代码

解析操作系统类型伪代码

```
1.  /**
2.   * 获取用户操作系统信息
3.   * @returns 操作系统类型及版本字符串
4.   */
5.  export function getOS() {
6.    const userAgent = window.navigator.userAgent;
7.    let os = 'Unknown';
8.    let version = 'Unknown';
```

```
9.
10.   // 匹配 Windows 操作系统及版本
11.   const windowsRegex = /Windows\sNT\s(\d+\.\d+)/;
12.   if (windowsRegex.test(userAgent)) {
13.     os = 'Windows';
14.     const result = userAgent.match(windowsRegex);
15.     version = result ? result[1] : '';
16.   } else if (/Mac OS X (\d+[._]\d+[._]\d+)/.test(userAgent))
    {
17.     // 匹配 MacOS 操作系统及版本
18.     os = 'MacOS';
19.     const result = userAgent.match(/Mac OS X (\d+[._]\d+[._]
  \d+)/);
20.     version = result ? result[1] : '';
21.   } else if (/Linux/i.test(userAgent)) {
22.     // 匹配 Linux 操作系统
23.     ......
24.   } else if (/Android (\d+\.\d+)/.test(userAgent)) {
25.     // 匹配 Android 操作系统及版本
26.     ......
27.   } else if (/iPhone|iPad|iPod/.test(userAgent)) {
28.     // 匹配 iOS 操作系统及版本
29.     ......
30.   }
31.
32.   return `${os} ${version}`;
33. }
```

表 4-3 浏览器类型，设备信息伪代码

浏览器类型，设备信息伪代码

```
1.  /**
2.   * 获取用户浏览器信息
3.   * @returns 操作浏览器类型及版本字符串
4.   */
5.  export function getBrowser() {
6.    const userAgent = window.navigator.userAgent;
7.    let browser = 'Unknown';
8.    let version = '';
9.
10.   if (/Firefox/i.test(userAgent)) {
11.     browser = 'Firefox';
12.   } else if (/Chrome/i.test(userAgent)) {
```

```
13.     browser = 'Chrome';
14.  }
15.   // 省略其他浏览器的匹配代码
16.   ....
17.   // 获取浏览器版本
18.   const versionMatch = userAgent.match(/(Firefox|Chrome|Safa
  ri|Opera|Edge|MSIE|Trident)\/([\d.]+)/);
19.   if (versionMatch && versionMatch.length >= 3) {
20.     version = versionMatch[2];
21.     browser += ` ${version}`;
22.   }
23.
24.   return browser;
25. }
26.
27. /**
28.  * 获取屏幕分辨率信息
29.  * @returns 屏幕分辨率信息
30.  */
31. export function getScreenResolution() {
32.   return `${window.screen.width}x${window.screen.height}`;
33. }
34.
35. /**
36.  * 获取设备类型
37.  * @returns 设备类型字符串
38.  */
39. export function getDeviceType() {
40.   const userAgent = window.navigator.userAgent;
41.   if (/Mobile|Android|iPhone|iPad|iPod|Windows Phone/i.test(
  userAgent)) {
42.     return 'Mobile';
43.   } if (/Tablet|iPad/i.test(userAgent)) {
44.     return 'Tablet';
45.   }
46.   return 'Desktop';
47. }
```

表 4-2 及 4-3 所示的代码将获取各种信息的逻辑以函数的方式封装，之后在 Builder 中只需要将这些数据与原始数据进行拼装即可。

表 4-4 Builder 伪代码

Builder 伪代码

```
1.  export default class Builder {
2.    private appInfo: AppInfo;
3.
4.    private env: EnvContext;
5.    // 接收用户配置信息，与环境数据
6.    constructor(appInfo: AppInfo, env: EnvContext) {
7.      this.appInfo = appInfo;
8.      this.env = env;
9.    }
10.   // build 组装数据
11.   build(category: any, context: any) {
12.     const structure = {
13.       ...this.appInfo,
14.       env: this.env,
15.       context,
16.       timestamp: new Date().getTime(),
17.       category,
18.     };
19.
20.     return structure;
21.   }
22. }
```

表 4-4 展示了 Builder 的伪代码实现，在实例化时，传入用户提供的有关于应用的信息以及环境数据，之后通过调用 build 方法返回组装好的数据。

4.3 Sender 发送器

Sender 负责数据的上报，首先接收各个监控插件通过 Client 向 Sender 发送待上报数据。内部对上报的数据分成立即上报和延迟上报，提供两种队列 waitQueue 和 unloadQueue。waitQueue 中的数据采取定时定量上报，unloadQueue 中的数据则需要等到页面隐藏或卸载时上报，其整体时序图如图 4-1 所示。

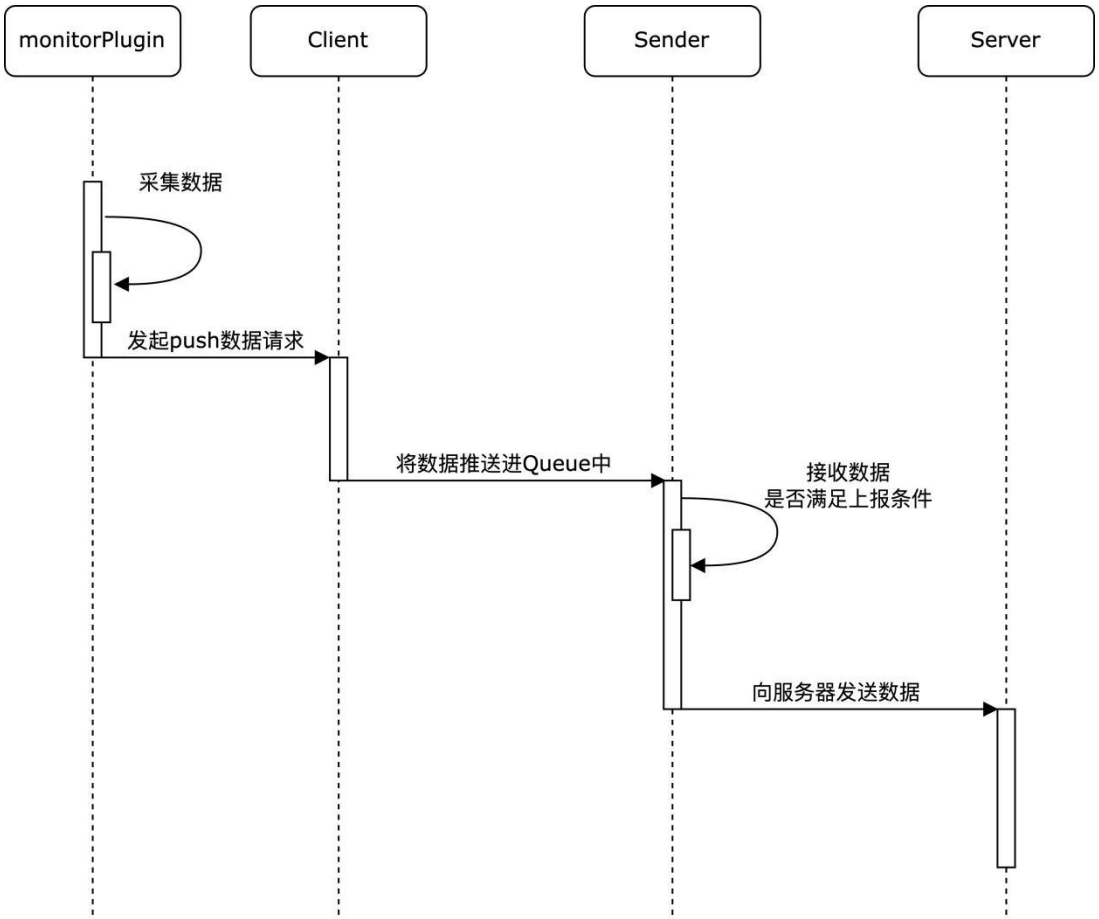

图 4-1 上报时序图

上报方式有两种，一种是通过<img/>元素设置其 src 属性，达到发送数据的目的。这样有一些缺点，大部分浏览器会延迟卸载文档以加载图像，使得下一个页面出现地更慢。第二种是采用 sendBeacon，数据是通过 HTTP POST 请求发送的，它最初要解决的问题就是向服务器发送统计数据，在浏览器有机会时异步地向服务器发送数据，同时不会对下一个导航造成影响。另外，考虑到有些浏览器不支持 sendBeacon，所以在实现中采取降级上报。

unloadQueue 中的数据需要等到用户离开页面或者直接关闭页面时发送到服务器，要在代码中描述这个时间点需要借助 Page Lifecycle 规范，Page Lifecycle 是 W3C 提出的新规范，它旨在为浏览器提供管理页面生命周期的能力，进而管理 Web 应用的内存、CPU、电池，网络等关键资源，浏览器会更积极的优化系统资源。现代浏览器基本支持 Page Lifecycle，并且允许开发者响应这些状态之间的转换，页面生命周期中的状态以及转换事件如图 4-2 所示。

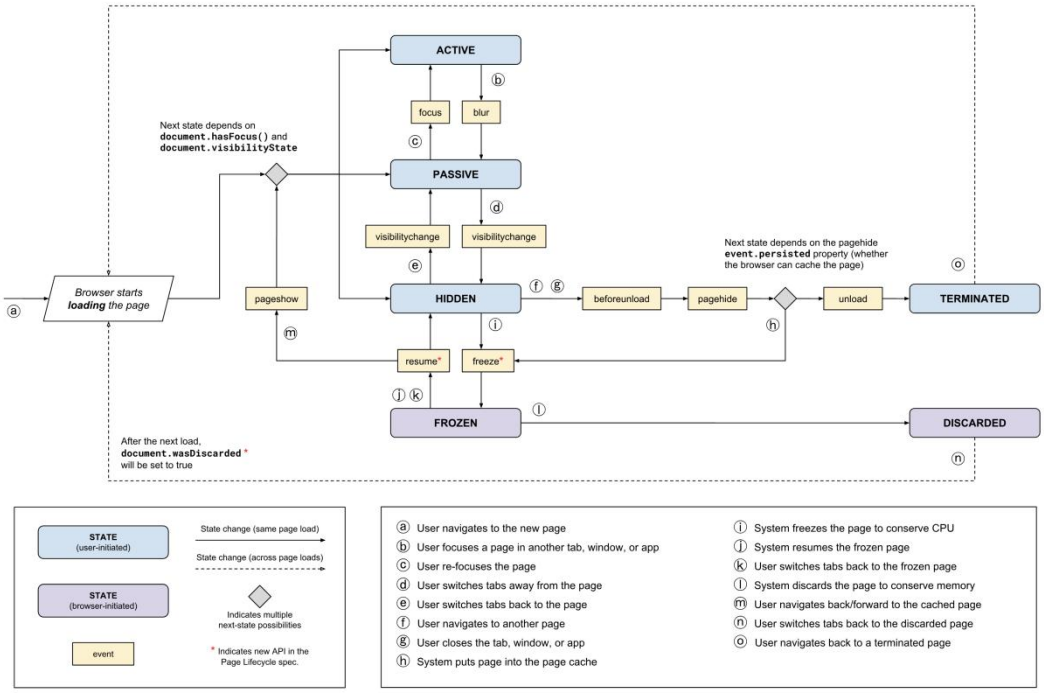

图 4-2 Page Lifecycle 状态转换图

从图中可以看出与页面结束相关的状态有 HIDDEN（隐藏）、TERMINATED（终止）、FROZEN（冻结）、DISCARDED（废弃），并且后面三种状态一定会先经过 HIDDEN 状态，下表是对这几种状态的描述。

表 4-5 Page Lifecycle 部分页面状态

状态	描述
HIDDEN	页面不可见 document.visibilityState === 'hidden'且不被冻结。
FROZEN	该状态浏览器会挂起任务队列中可冻结任务的执行，使用 setTimeout、setTimeinterval、fetch 注册的回掉不会被执行，正在执行的任务可能会被执行，但有可能会被限制。这样做的目的是为了节约系统 CPU、内存、电量。
TERMINATED	该状态表示浏览器已卸载页面并回收了资源占用，不会有新的任务执行，已运行的长任务可能会被清除。
DISCARDED	该状态表示浏览器已经卸载页面并且释放页面所占用的资源，任何任务，回调都无法执行。页面虽然被卸载，但 Tab 页的标签名和 favicon 对用户仍然可见，下次进入该页面可能需要重新刷新。

从表 4-5 可知，最适合发送采集数据的状态是 HIDDEN，其他状态都有可能造成数据丢失。与 HIDDEN 状态相关的事件有发生状态转换前的 visibitilityChange 和状态转换后的 pagehide，beforeunload，unload，图 4-3 说明了这些事件对用户卸载页面行为和不同浏览器之间的支持兼容性。

		pagehide	beforeunload	unload	visibitilityChange visible -> hidden
Chrome desktop	Close visible tab	y	y	y	n (bug)
	Close background tab	y	y	y	y
	Navigate away	y	y	y	n (bug)
Chrome mobile	Home-button, swipe away in app switcher	n	n	n	y
	Task-switch, swipe away	n	n	n	y
	Navigate away	y	y	y	n (bug)
Firefox desktop	Close visible tab	y	n	y	y
	Close background tab	y	n	y	y
	Navigate away	y	y	y	y
Firefox mobile	Home-button, swipe away in app switcher	n	n	n	y
	Task-switch, swipe away	n	n	n	y
	Navigate away	y	y	y	y
Safari desktop	Close visible tab	n	y	n	n (bug)
	Close background tab	n	y	n	y
	Navigate away	y	y	n	n (bug)
Safari mobile	Home-button, swipe away in app switcher	n	n	n	y
	Task-switch, swipe away	n	n	n	y
	Navigate away	y	n	n	n (bug)
Edge desktop	Close visible tab	y	y	y	n (bug)
	Close background tab	y	y	y	y
	Navigate away	y	y	y	n (bug)

图 4-3 页面生命周期事件兼容性报告

从这份兼容性报告可以得知 pagehide、beforeunload、unload 事件在移动端浏览器中许多关闭页面的交互下都不会触发。相比之下，visibitilityChange 事件更加可靠，但存在部分浏览器在部分交互下不支持的情况，所以采取 pagehide 作为回退方案。表 4-6 封装的 onHidden 函数，可以作为工具方法在页面不可见或卸载时执行逻辑。

表 4-6 onHidden 函数代码

onHidden 函数代码

```
1.  export const onHidden = (cb: OnHiddenCallback) => {
2.    const onHiddenOrPageHide = (event: Event) => {
3.      if (event.type === 'pagehide' || document.visibilityStat
  e === 'hidden') {
4.        cb(event);
```

```
5.       }
6.     };
7.     window.addEventListener('visibilitychange', onHiddenOrPage
  Hide, true);
8.     // 有些浏览器不支持 visibilitychange
9.     window.addEventListener('pagehide', onHiddenOrPageHide, tr
  ue);
10. };
```

表 4-7 Sender 模块代码

Sender 模块代码

```
1.  export default class Sender {
2.    private dsn: string;
3.    private waitQueue: string[];
4.    private unloadQueue: string[];
5.
6.    constructor(dsn: string) {
7.      this.dsn = dsn;
8.      this.waitQueue = [];
9.      this.unloadQueue = [];
10.     // 页面不可见时，发送 unloadQueue 队列中的数据
11.     onHidden(() => {
12.       while (this.unloadQueue.length) {
13.         this.send(this.unloadQueue.shift() as string);
14.       }
15.     });
16.   }
17.   push(data: string) {
18.     this.waitQueue.push(data);
19.     if (this.waitQueue.length >= 10) {
20.       while (this.waitQueue.length) {
21.         this.send(this.waitQueue.shift() as string);
22.       }
23.     }
24.   }
25.   pushUnload(data: string) {
26.     this.unloadQueue.push(data);
27.   }
28.   // 利用图片发送数据
29.   sendImg(data: string) {
30.     const img = new Image();
31.     img.src = `${this.dsn}?data=${encodeURIComponent(data)}`;
```

```
  }
  // 利用 Beacon 发送数据
  sendBeacon(data: string) {
    navigator.sendBeacon(this.dsn, JSON.stringify({ data }));
  }
  // 降级上报
  send(data: string) {
    if ('sendBeacon' in navigator) {
      this.sendBeacon(data);
    } else {
      this.sendImg(data);
    }
  }
}

```

在表 4-7 所示的 Sender 模块伪代码中先实现了图片上报和 Beacon 上报，此外提供了 push 方法和 pushUnlaod 方法向 waitQueue 队列和 unloadQueue 队列添加待发送的数据。上报的时机完全是由 Sender 内部决定的，调用方无需关心。

4.4 程序主体与插件化

上面主要阐述了 ConfigManager、Builder、Sender 的实现，这一小节会阐述如何将这些模块串联在一起形成一个完整的主体，实现插件化来扩展主体程序。

首先需要先定义插件协议，表明插件需要实现的基本 API，如表 4-8 所示。

表 4-8 插件类型定义

插件类型定义

```
export interface PluginInstance {
  name: string;
  setup(client: Client): void
  destory?: () => void
  loaded?: () => void
  [key: string]: any
}

export interface PluginWrapper<T> {
  (options: T): PluginInstance
}
```

上面代码约束了每个插件必须包含名称 name，实现一个 setup 方法用于插件自身的初始化，在这个方法中可以获取到程序主体并且得到程序主体暴露出来的能力，帮助插件更好地接入整个流程中。PluginWrapper 类型则是允许插件以函数的形式提供，考虑到插件本身可能需要从外部接收参数。

插件化在核心类上实现，具体实现对插件的管理，表 4-9 中的代码提供一个 use 方法安装插件。另外，提供一个 destory 方法在实例主体销毁时消除插件带来的副作用。

表 4-9 Core 伪代码

Core 伪代码

```
1.  export default class Core implements CoreType {
2.    plugins: Map<string, PluginInstance>;
3.    constructor() {
4.      this.plugins = new Map();
5.    }
6.    use(plugin: PluginInstance) {
7.      // 不能重复安装插件
8.      if (this.plugins.has(plugin.name)) {
9.        return;
10.     }
11.     this.plugins.set(plugin.name, plugin);
12.     try {
13.       // 初始化插件
14.       plugin.setup(this);
15.     } catch (err) {
16.       console.error(`plugin '${plugin.name}' occurred error,
  ${err}`);
17.     }
18.   }
19.   // 销毁副作用
20.   destory() {
21.     this.plugins.forEach((plugin) => {
22.       if (plugin.destory) {
23.           plugin.destory();
24.       }
25.     });
26.   }
27. }
```

Client 主体继承 Core 类，Core 类主要负责实现一些通用核心的功能，例如上面的插件化。Client 类主要负责串联之前实现的模块，并且提供暴露给插件的能力。

表 4-10 Client 主体程序

Client 主体程序

```
1.  export class Client extends Core implements ClientType {
2.    constructor(config: Partial<IGlobalConfig>) {
3.      super();
4.      this.eventBus = emitter;
5.      // 初始化 ConfigManager
6.      this.configManager = new ConfigManager();
7.      // 设置配置
8.      this.configManager.set(config);
9.      // 初始化 sender
10.     this.sender = new Sender(this.configManager.get('dsn'));
11.     // 初始化 builder
12.     this.builder = new Builder({
13.       appId: getConfig('appId'),
14.       appName: getConfig('appName'),
15.       appVersion: getConfig('appVersion'),
16.       uid: getConfig('uid'),
17.     }, getUserEnv());
18.
19.     this.initPush();
20.   }
21.
22.   private initPush() {
23.     // 通过 emitter 发布订阅模式提供 push 待发送数据的能力
24.     emitter.on('waitQueue', (data) => {
25.       const buildData = this.builder.build(data.category,
  data.context);
26.       this.sender.push(JSON.stringify(buildData));
27.     });
28.     emitter.on('unloadQueue', (data) => {
29.       const buildData = this.builder.build(data.category,
  data.context);
30.       this.sender.pushUnload(JSON.stringify(buildData));
31.     });
32.   }
33.
```

```
34.   /**
35.    * 启动，安装插件
36.    */
37.   start() {
38.     this.use(jsError);
39.     this.use(promiseError);
40.     this.use(rsError);
41.     ......
42.   }
43. }
```

在构造函数中，对 Builder，Sender，ConfigManager 进行实例化，方便后续调用他们各自提供的能力，在 start 方法中初始化各种插件，代表程序主体开始启动运行。使用 use 方法注册插件时，会自动将当前实例传入插件，这样插件可以获得一些程序主体的能力。但是有些能力是模块提供的，插件使用起来，可能不是很方便。故在 Client 中，对 Sender 模块中的 push 功能做一些封装，并且通过订阅发布器向插件暴露，如表 4-10 中的代码 22 至 32 行所示。

4.5 功能插件

上面已经使用代码实现了监控 SDK 的整体流程，包括配置管理，数据组装，数据发送。以及为了 SDK 得到扩展性而实现的插件化，这一节将借助插件机制来实现具体的数据采集功能。

4.5.1 性能监控插件

性能数据可分为两种，一种是以技术为中心的性能指标，另外一种是以用户为中心的性能指标[8]。用户为中心的性能指标是指用户可以直接感受到的相关性能指标，例如网页多快可以加载所有的视觉元素并将其渲染到屏幕上，页面上的元素是否会以用户意想不到的方式对用户的交互造成干扰，这些性能指标大多通过 PerformanceObserver 来获取，将其获取过程封装方便后续使用，如表 4-11 所示。

表 4-11 observe 指标获取方法

observe 指标获取方法

```
1.  const observe =
2.  (type: string, callback: PerformanceEntryHandler):
3.  PerformanceObserver | undefined => {
4.    // 类型合规，就返回 observe
5.     if (PerformanceObserver.supportedEntryTypes?.includes(type
  )) {
6.       const ob: PerformanceObserver = new PerformanceObserver(

7.         (l) => callback(l.getEntries()));
8.
9.       ob.observe({ type, buffered: true });
10.      return ob;
11.   }
12.   return undefined;
13. };
14.
```

observe 方法接收 type 参数表示获取的指标类型，callback 回调函数用于在浏览器报告该性能指标时接收性能条目 PerformanceEntry。方法首先检查传入的 type 是否被支持以确保正确接收性能条目，如果性能条目被支持，则实例化 PerformanceObserver 对象并观测对应的 type。

以技术为中心的性能指标是指用户无法直接感知到的延迟，但对于技术人员来说，采集其有意义的延迟时间能够从精确数据的角度对网站性能有一个定义，并提供优化方向。Navigation Timing API 提供关于页面导航各个阶段的延迟耗时，与其他基于 JavaScript 的机制不同，该 API 可以提供可以更有用和更准确的端到端延迟数据[9]。同时，可以衡量之前难以获取的数据，如卸载前一个页面的时间，在域名解析上的时间，等等。更多有意义的数据结合表 4-12 查看。

表 4-12 关键时间点计算方式

上报字段	描述	计算公式
TTI（Time to Interact）	首次可交互时间	domInteractive - fetchStart
Ready	HTML 加载完成时间，即 DOM Ready 时间。	domContentLoadEventEnd - fetchStart

Load	页面完全加载时间	loadEventStart - fetchStart
FirstByte	首包时间	responseStart - domainLookupStart
DNS	DNS 查询耗时	domainLookupEnd - domainLookupStart
TCP	TCP 连接耗时	connectEnd - connectStart
TTFB（Time to First Byte）	请求响应耗时	responseStart - requestStart
Trans	内容传输耗时	responseEnd - responseStart
DOM	DOM 解析耗时	domInteractive - responseEnd
Res	资源加载耗时	loadEventStart - domContentLoadedEventEnd
SSL	SSL 安全连接耗时	connectEnd - secureConnectionStart

表 4-13 展示的代码是根据上述计算公式来计算各个性能指标，篇幅原因省略了大部分的类同代码。

表 4-13 以技术为中心的指标获取

以技术为中心的指标获取

```
1.  export const getNavigationTiming = (): MPerformanceNavigatio
  nTiming => {
2.     const resolveNavigationTiming = (entry: PerformanceNavigat
  ionTiming):
3.       MPerformanceNavigationTiming => {
4.       const {
5.         domInteractive,
6.         fetchStart,
7.         // 其他时间节点
8.         ......
9.       } = entry;
10.
11.      return {
12.        TTI: domInteractive - fetchStart,
```

```
13.       // 其他指标
14.       ......
15.     };
16.   };
17.   const navigation = performance.getEntriesByType('navigatio
  n')[0]
18.   return resolveNavigationTiming(navigation as PerformanceNa
  vigationTiming);
19. };
```

以用户为中心的性能指标主要衡量 FP、FCP、LCP、FID、CLS，主要通过上述实现的 observe 方法获取这些指标，它们的获取方式一样，表 4-14 以获取 LCP 为例来说明。

表 4-14 获取 LCP 伪代码

获取 LCP 伪代码

```
1.  function getLCP(store: MetricsStore) {
2.    return new Promise((resolve) => {
3.      observe('largest-contentful-paint', (entryList) => {
4.        const entry = entryList[entryList.length - 1];
5.        if (entry) {
6.          resolve(entry);
7.        }
8.      });
9.    });
10. }
```

到目前，已经将各种指标的获取方法封装完毕，需要把这些零散的功能全部组合在插件内部，使用主体程序暴露的能力，表 4-15 所示代码将性能监控这部分功能集成进整体的监控流程中。

表 4-15 性能监控插件伪代码

性能监控插件伪代码

```
1.  const performancePlugin: PluginInstance = {
2.    name: 'plugin-performance',
3.    setup(client) {
4.      // 获取并上报以技术为中心的性能指标
5.      client.eventBus.emit('waitQueue', {
6.        context: getNavigationTiming(),
```

```
7.         category: TransportCategory.LOAD_SPEED,
8.       });
9.
10.      // 获取并上报以用户为中心的性能指标
11.      getVitals(metrics).then((data) => {
12.        client.eventBus.emit('waitQueue', {
13.          context: data,
14.          category: TransportCategory.PERF,
15.        });
16.      });
17.    },
18. };
19.
20. export default performancePlugin;
```

插件被初始化时直接向 waitQueue 队列中推送待发送的数据。getVitals 方法获取所有以用户为中心的指标，在接收所有指标完成后上报。这样保证所有指标都能被采集到。

4.5.2 计算 FMP

FMP（First Meaningful Paint）是指网页渲染主要内容的时间点，并且这个时间点能为用户提供有意义的信息，也就是说用户能够感知到页面已经渲染出有效内容[10]。这与上一节介绍的 FP，FCP 以及 LCP 是有区别的，FP 和 FCP 主要关注页面开始发生变化的时间点，LCP 主要衡量页面内最大元素的加载速度。但这些时间基本都不能代表用户看到页面主要内容的时间，所以需要引入 FMP 指标，也称为首屏时间。图 4-4 展示了页面加载过程对应的指标时间，直观地看到 FMP 是向用户完全展示所有内容的时间点。

图 4-4 页面可视化加载过程

但由于 FMP 指标具有主观性，比较难从技术角度确定哪些内容对用户来说是“有意义的”，同时 FMP 与其他指标有重叠的部分。鉴于此，FMP 指标并没有被规范化，浏览器也并未实现相关 API 来直接获取该指标，所以需要开发者自行研究算法来测量。虽然自行测量 FMP 可能存在误差，但是在实践中 FMP 仍然对改善用户体验具有重要意义。

本节介绍两种 FMP 算法，一种是基于权重计算，根据页面元素权重，计算权重最高的元素渲染时间，另外一种是基于趋势计算，认为页面中元素增量最大的时间点是主要内容的渲染时间。

4.5.2.1.基于 DOM 权重的方法

基于权重计算的算法分为两个阶段，打标记阶段和反推计算阶段。打标记阶段主要利用 MutationObserver 观测整个 DOM Tree，DOM Tree 发生变化时，取到发生变化的一组 DOM 节点，为这组 DOM 添加标记同时基于标记缓存当前时间作为这组 DOM 的加载时间，方便之后通过 DOM 节点读取其加载时间，其流程如图 4-5 所示。

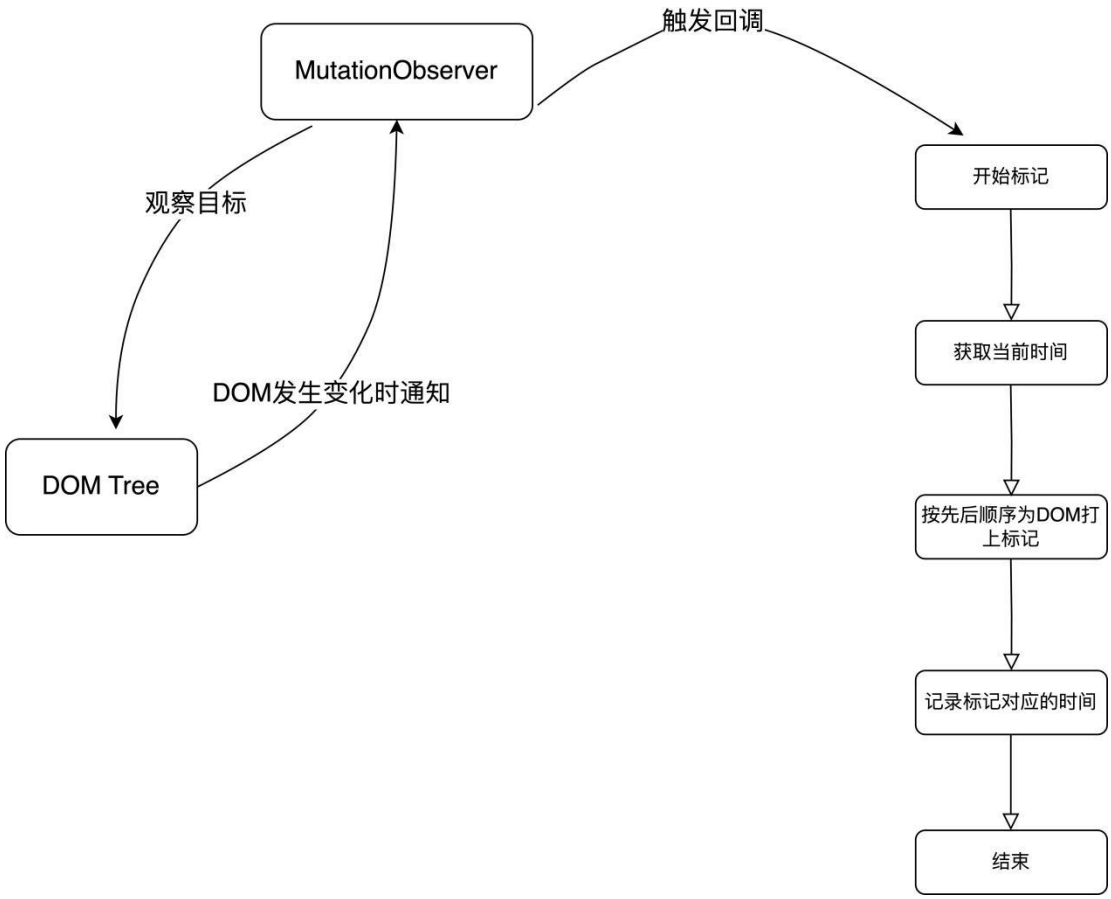

图 4-5 打标记流程图

前面的步骤在页面元素不断增加时，已经为所有 DOM 都打上了标记。当触发 Load 事件时，表示页面所有 DOM 已经解析并且渲染完成。此时应该进行权重计算，下面规定一些计算原则。

1. 权重计算之前应该先规定不同类型的元素的初始权重，形成一张权重表，比如一般认为视频控件比图片更有可能成为主要内容，那么视频控件的初始权重就应该大于图片。其他元素都可看成普通 DOM。

2. 每个元素权重的计算方式为：宽度*高度*初始权重。

3. 对于资源类型元素取其从网络中响应结束的时间点作为 FMP，其他元素取其加载完成事件作为 FMP。

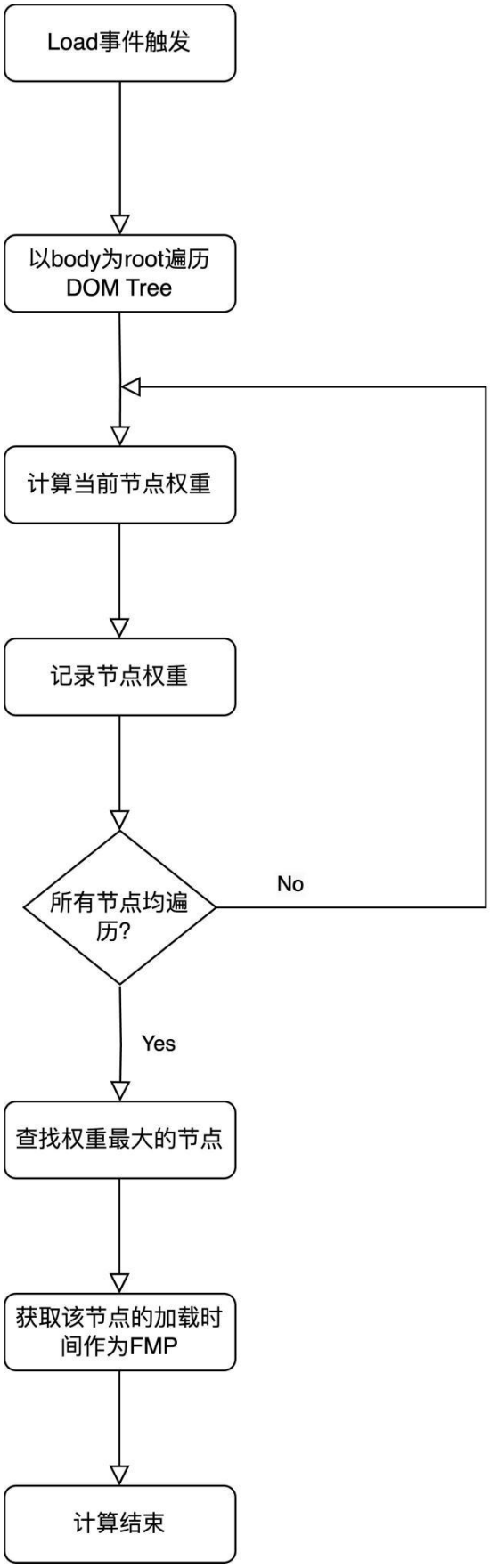
Load事件触发
以body为root遍历
DOM Tree
计算当前节点权重
记录节点权重
所有节点均遍
历?
No
Yes
查找权重最大的节点
获取该节点的加载时
间作为FMP
计算结束

图 4-6 FMP 计算过程

图 4-6 为计算流程图，首先遍历整颗 DOM 树，对每个节点计算其权重，最终找到权重最大的节点，获取其加载完成的时间作为 FMP，表 4-15 是该算法的伪代码。

表 4-15 FMP 计算伪代码

FMP 计算伪代码

```
let tagNumber = 0  // 节点标记
let stack = []  // 存储 tag 对应节点的加载时间
const domObserver = new MutationObserver(callback)
domObserver.observe(document.body, {
  subtree: true,
  childList: true
})
function callback() {
  // 记录下元素被渲染的时间 tagNumber -> time
  stack[++tagNumber] = performance.now()
  // 为元素打上标记  tagNumber -> node
  patchTag(tagNumber)
}
// 根据提前约定的元素标签权重计算节点最终的权重
function computeWeight(node) {
  // 获取元素尺寸
  let { width, height } = getSize(node)
  if (isOutside(node)) {
    return 0
  }
  const wts = TAG_WEIGTH_MAP[node.tagName]
  const weight = width * height * wts
  return {
    weight,
    tagNumber: node.getAttr('tagNumber'),
    tagName: node.tagName,
    node
  }
}
// 获取权重最大的元素
function getCoreNode(root) {
  const weightList = nodeTraversal(node)  // 递归遍历元素获取权重列表
  return getMaxWeight(weightList)
}
```

```
function getFMP() {
  const coreNode = getCoreNode(document.body)
  let fmp = -1
  let {
    tagName,
    tagNumber,
    node
  } = coreNode
  // 如果是图片和视频则获取其加载完成后的时间，否则直接获取打标记阶段记录的时间
  if (tagName === 'IMG' || tagName === "VDEIO") {
    fmp = getResourceLoaded(node)
  } else {
    fmp = stack[tagNumber]
  }
  return fmp
}
window.addEventListener('load', () => getFMP(document.body))

```

4.5.2.2.基于 DOM 变化趋势的方法

随着页面的加载，浏览器逐步将布局对象添加到布局树中。图 4-7 展示了某页面布局对象与时间的关系。

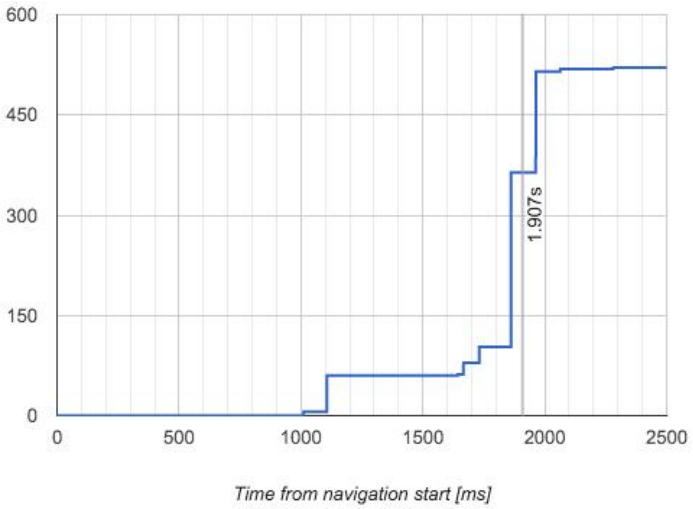

图 4-7 布局对象变化图表

图中的时间相对于浏览器导航开始的时间，可以看出在 1.907s 时的布局对象增量最大，结合图 4-8 可以总结出布局对象的变化与页面实际视觉变化的关系。

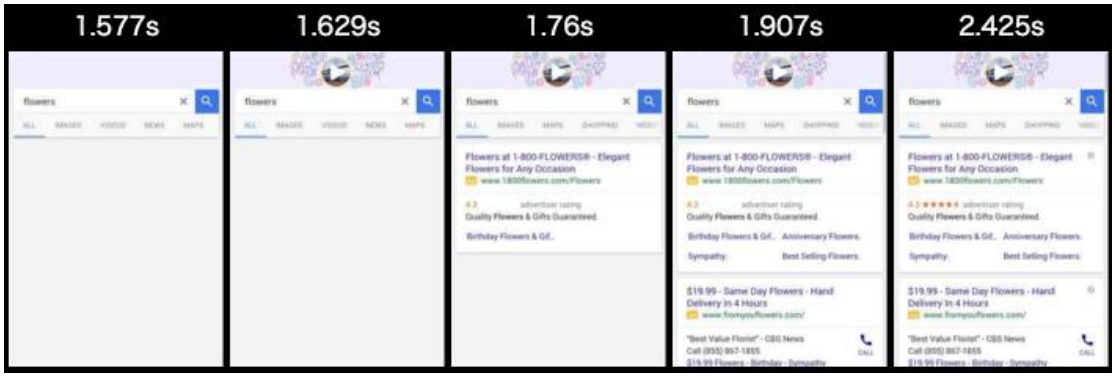

图 4-8 可视化加载过程

1. 1.57s 仅仅只渲染了页面 header 部分，此时，大约有 60 个布局对象在布局树中，接近 LCP 的时间点。

2. 在 1.83s 附近，整个顶部渲染完成，此时大约有 103 个布局对象。

3. 1.907s 展示了大量搜索结果，从产品的角度来看，主要内容已呈现，且满足用户信息需求，所以这个时间点可以看成 FMP。从布局对象变化趋势来看，这个时间点布局对象增量最大、最剧烈。

综上所述，考虑到浏览器绘制时间，可以认为 FMP = 布局对象变化最大的下一次绘制时间。

图 4-9 布局对象变化伪趋势

但上述计算方法可能并不完善，如图 4-9 所示，在 6.407s 与 24.25s 分别发生了一次较大的布局对象变化，后者要更大一些，如果认为 24.25s 是 FMP 这会产生极大的误差。从页面加载可视化过程来看，21.787s 至 24.25s 用户所看到的页面的变化并没有太大差别，反观 5.78s 至 6.047s 这段时间，在 6.047s 页面渲染出了之前没有的网页主要内容，所以 6.047s 才是真正意义上的 FMP。

造成这种误差的原因是因为布局对象的变化并不能够直接反应页面视觉变化，可能是因为被添加到布局树中的布局对象被渲染在了首屏之外，用户无法直接看到。另外也需要考虑 CSS 的影响，当元素的 visibility 属性设置为 hidden 时，元素不可见，但仍然占据空间，有高度和宽度，元素的 display 属性被设置为 none 时元素不可见，不占据空间。所以在计算 FMP 时需要排除不可见元素的影响，可认为 FMP = 可见区域内布局对象变化最大的下一次绘制时间。

具体的算法实现需要考虑如何衡量布局对象的变化剧烈程度。一般来说，如果 DOM 树的层级越深，子节点数目越多，这棵树就越复杂，越“茂密”。可以采取打分制对树打分，任意一颗 DOM 树的分数为子树的总分+1+0.5*当前节点的深度，对其子树做同样运算，递归处理。

在对一个树的计分过程中需要注意排除不可见元素的影响，如遇到不可见元素则其得分为 0，同时进行剪枝，防止不必要的计算，因为如果一个元素不可见则其所有子元素通常也不可见。判断元素是否在首屏视口内可见，需要获取元素的位置信息，可能导致强制同步布局，影响性能。可以借助 requestAnimationFrame 这个 API 在下一次绘制前执行计算过程，最大限度避免强制同步布局。同时可以减少获取元素位置信息的频率，一般如果子树总分大于 0，说明子树在视口范围内可见，则父元素同样在视口范围内可见，这种情况不需要获取父元素的位置。

表 4-16 计算 DOM 树分数伪代码

计算 DOM 树分数伪代码

```
1.  function computeDOMLayoutScore(root, depth, isSiblingExists)
    {
2.   if (!root) return 0
3.   const children = root.children
4.   // 所有子树分数
5.   let childrenScore = 0
6.   // 兄弟节点分数
7.   let prevChildrenScore = 0
8.   for (let i = 0; i < children.length; i++) {
9.    // 递归计算每个子树分数
10.   const score = computeDOMLayoutScore(children[i], depth + 1
    , prevChildrenScore > 0)
11.   prevChildrenScore = score
12.   childrenScore += score
13.  }
14.  let isVisible = true
15.  // 如果需要检查位置
16.  const isPositionCheck = childrenScore <= 0 && !isSiblingExi
    sts
17.  if (isPositionCheck) {
18.   // 不可见或者不在首屏内的元素对 FMP 没有影响
19.   if (isHidden(root)) {
20.    isVisible = false
21.   }
22.  }
23.  // 如果不可见，则得分为 0
24.  if (isVisible) return 0
25.
26.  return childrenScore + 1 + 0.5 * depth
```

```
27. }
```

表 4-16 的代码描述了如何计算某一时刻 DOM 树的分数，还需要借助 MutationObserver 在 DOM 结构发生变化时计算 DOM 树分数，并记录当前时间和分数，最终按照时间将分数排序，找出分数增量最大的时间节点即为 FMP。

4.5.3 错误数据采集插件

错误采集插件采集的范围为：JS 运行时错误、Promise 错误，资源加载错误，http 请求错误，请求跨域错误[11]。语法错误会在编译阶段被发现，所以不考虑语法错误的监控，下面简单介绍各种错误的采集方法，表 4-17、4-18、4-19、4-20 是各种错误的数据格式。

1.JS 运行时错误：监听全局 error 事件，可以捕获到错误类型，错误堆栈，错误发生的行列信息。

2.Promise 错误：监听全局 unhandledrejection 事件，但无法获取错误发生的代码行列信息。

3.资源加载错误：与 JS 运行时错误监听同一个全局事件，不同的是，资源加载错误只能在错误捕获阶段被采集，同时需要判断事件对象的 target 属性是否存在 src 或者 href，来与 JS 运行时错误区分开来。

4.http 请求错误：发送 http 请求通常使用 XMLHttpRequest 对象或者 fetch 函数，不存在有全局的错误监听事件，所以只能通过重写这两种 API，将错误采集的逻辑嵌入重写后的 API。

表 4-17 JS 运行时错误数据格式

JS 运行时错误数据格式

```
1.  {
2.    title  // 页面标题
3.    errorType  // 错误类型
4.    timestamp // 发生错误的时间戳
5.    url  // 发生错误页面的路径
6.    mechanism  // 捕获到错误的径
7.    errorUid // 错误的标识码
```

```
8.    filename // 发生错误的代码文件
9.    message  // 具体错误信息
10.   type // js 错误类型 类似 TypeError SyntaxError
11.   stack // 错误堆栈
12. }
```

表 4-18 Promise 错误数据格式

Promise 错误数据格式

```
1.  {
2.    title  // 页面标题
3.    errorType  // 错误类型
4.    timestamp // 发生错误的时间戳
5.    url  // 发生错误页面的路径
6.    mechanism  // 捕获到错误的途径
7.    errorUid // 错误的标识码
8.    reason   // promise 被拒绝的原因
9.  }
```

表 4-19 资源请求错误数据格式

资源请求错误数据格式

```
1.  {
2.    type  // 何种类型的资源错误
3.    tagName // 标签
4.    url  // 资源 url
5.    triggerTime  // 发生错误时间
6.    pageUrl  // 页面 url
7.    pageTitle // 页面标题
8.  }
```

表 4-20 http 请求错误数据格式

http 请求错误数据格式

```
1.  {
2.    method // 请求方法
3.    url  // 请求的 url
4.    body   // 请求实体
5.    requestTime  // 发起请求的时间
6.    responseTime  // 响应时间
7.    status  // http 请求状态
8.    statusText  // http 请求状态短语
9.    response // 响应实体
```

```
10. }
```

上表给出了各个错误最终的错误格式，插件需要做的就是根据采集方法，获取到对应的错误信息，转换成预定义的错误格式。

5. 监控平台

5.1 开发环境

监控平台使用的开发环境，开发工具如表 5-1 所示。

表 5-1 开发环境

环境	名称
IDE	Visual Studio Code v1.88.0
服务器环境	Ubuntu v20.04、Nginx v1.24.0
服务器运行平台	Node.js v20.10.0
数据库	mysql v8.2.0
包管理工具	npm v10.2.3
构建工具	vite v5.2.0
浏览器	Chrome 123.0.6312.87

5.2 获取数据

要对监控数据进行操作第一步要先获取数据，一般 SDK 通过图片上报的数据会停留在 nginx[12]访问日志（access_log）中，默认日志格式包含访客的 IP 地址、用户名称、请求时间、请求行信息、HTTP 状态码等信息。另外也可以通过 log_format 指令自定义日志格式。表 5-2 是精简之后，SDK 上报的一条日志。

表 5-2 原始 access_log 日志

原始 access_log 日志
14.120.115.181 - - [31/Mar/2024:15:53:28 +0800] "GET /assets/icon-6-e749ec25.png?data=%7B%22appId%22%3A%22mGGeXsyR%22%2C%22appName%22%3A%22%22%2C%22appVersion%22%3A%22%22%2C%22uid%22%3A%2288888%22%2C%22env%22%3A%7B%22browser%22%3A%22Chrome%20123.0.0.0%22%2C%22os%22%3A%22MacOS%2010_15_7%22%2C%22deviceType%22%3A%22Desktop%22%2C%22screen%22%3A%221470x956%22%7D%2C%22context%22%3A

%7B%22title%22%3A%22Vite%20App%22%2C%22errorType%22%3A%22js-error%22%2C%22mechanism%22%3A%22onerror%22%2C%22message%22%3A%22jjj%20is%20not%20defined%22%2C%22url%22%3A%22http%3A%2F%2Flocalhost%3A4001%2F%2F%22%2C%22timestamp%22%3A1711871608036%2C%22filename%22%3A%22http%3A%2F%2Flocal HTTP/1.1" 200 5969 "http://localhost:4001/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36" "-"

由于数据通过 url 参数携带，必须对其中的特殊字符进行 base64 编码，上述日志是原始编码后的数据，对其解码可看到 SDK 原始上报的数据，如表 5-3 所示。

表 5-3 解码后的日志

解码后的日志
14.120.115.181--[31/Mar/2024:15:53:28+0800] '/assets/icon-6-e749ec25.png?data={"appId":"mGGeXsyR","appName":"","appVersion":"","uid":"88888","env":{"browser":"Chrome123.0.0.0","os":"MacOS10_15_7","deviceType":"Desktop","screen":"1470x956"},"context":{"title":"ViteApp","errorType":"js-error","mechanism":"onerror","message":"jjjisnotdefined","url":"http://localhost:4001//","timestamp":1711871608036,"filename":"http://localhost:4001/main.js?t=1711869843323","stack":[{"filename":"http://localhost:4001/main.js?t=1711869843323","functionName":"b","lineNumber":17,"columnNumber":15},{"filename":"http://localhost:4001/main.js?t=1711869843323","functionName":"a","lineNumber":14,"columnNumber":3},{"filename":"http://localhost:4001/main.js?t=1711869843323","functionName":"triggerJSError","lineNumber":32,"columnNumber":3},{"filename":"http://localhost:4001/main.js?t=1711869843323","functionName":"HTMLButtonElement.<anonymous>","lineNumber":40,"columnNumber":3}],"errorUid":"anMtZXJyb3ItVW5jYXVnaHQlMjBSZWZlcmVuY2VFcnJvciUzQSUyMGpqaiUyMGlzJTIwbm90JTIwZGVmaW5lZC1odHRwJTNBJTJGJTJGbG9jYWxob3N0JTNBNDAwMSUyRm1haW4uanMlM0Z0JTNEMTcxMTg2OTg0MzMyMw==","type":"ReferenceError"},"timestamp":1711871608036,"category":"error"}' HTTP/1.1" 200 5969 "http://localhost:4001/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36" "-"

解码之后可以看到 SDK 上报的原始数据，随着时间的推移，日志量会非常庞大，不可能手动获取日志信息，所以必须有一个自动化的过程来获取日志，解析日志最后将日志以特点的结构持久化在数据库中，方便后续处理。数据量小的

情况下可以监控 access_log 日志文件的变化，然后将日志清洗入库。如果数据体量较大，存在多台日志服务器，这种方式很难保证数据不会丢失，需要一套更加完善处理机制。

对于多台日志服务器可以使用消息系统来作为日志的临时存储和传递介质，一个消息系统负责将数据从一个应用传递到另外一个应用，应用只需关注于数据，无需关注数据在两个或多个应用间是如何传递的[13]。分布式消息传递基于可靠的消息队列，在客户端应用和消息系统之间异步传递消息。Kafka[14]是一个分布式流处理平台和消息队列系统，广泛用于构建实时数据管道和流式应用程序。它是一个基于发布-订阅的消息系统，生产者和消费者都有其特定的主题（topic），数据可以被多个消费者消费，但消费者只能消费订阅主题产生的数据。

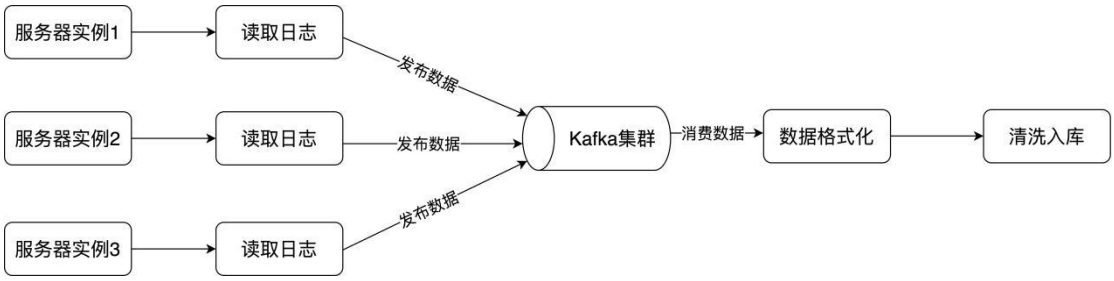

图 5-1 日志系统

图 5-1 展示了日志系统，日志先到达日志服务器，通过脚本读取日志添加到特定主题中。消费者消费特定主题中的数据，并按照时间年、月、日、小时、分钟以文件目录的形式归档这样做的好处是保证单个日志文件不会过大，同时方便检索。

考虑到集群实现复杂性，本文基于 node 定时任务定时读取日志文件，图 5-2 为过程示意图，表 5-4 为日志读取和处理的代码。

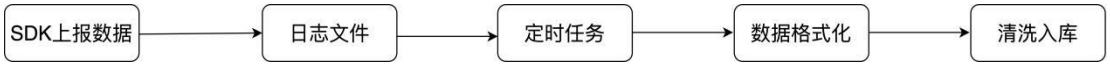

图 5-2 日志处理过程

表 5-4 日志读取伪代码

日志读取伪代码

```
const fs = require('fs');
const path = require('path');
const moment = require('moment');
const cron = require('node-cron');

// 定义日志文件路径和处理间隔时间（单位：毫秒）
const logFilePath = '/path/to/log/file';
const interval = '* * * * *'; // 每分钟处理一次

// 定义日志文件的目录结构
const logDir = '/path/to/log/dir';
const logFormat = 'YYYY-MM-DD-HH-mm';

// 定义日志处理函数
function processLog() {
  // 读取日志文件内容
  const logContent = fs.readFileSync(logFilePath, 'utf8');

  // 对日志进行预处理格式化
  const formattedLog = preprocessLog(logContent);

  // 将格式化后的日志按照年月日小时分钟归档到对应的目录下
  const logTime = moment();
  const logSubDir = logTime.format(logFormat);
  const logSubDirPath = path.join(logDir, logSubDir);
  if (!fs.existsSync(logSubDirPath)) {
    fs.mkdirSync(logSubDirPath);
  }
  const logFileName = `${logTime.valueOf()}.log`;
  const logFilePath = path.join(logSubDirPath, logFileName);

  fs.writeFileSync(logFilePath, formattedLog);

  // 清除已读取的日志
  fs.truncateSync(logFilePath, 0);
}

// 定义日志预处理函数
function preprocessLog(logContent) {
  // TODO: 根据实际需求实现日志预处理逻辑
```

```
40.   return logContent;
41. }
42.
43. // 定时执行日志处理函数
44. cron.schedule(interval, function run() {
45.   processLog();
46. });
47.
```

5.3 监控平台的实现

5.3.1 用户应用模块实现

本模块实现了用户登陆与注册，应用新建与删除。应用是监控数据的基本单位，一个用户可以创建多个应用。

图 5-3 用户界面

图 5-3 展示了用户登录与注册页面，用户在成功登录后，采用合 cookie 和 token 的方法来维持用户的登录状态[15]。具体来说，cookie 在这里扮演了一个重要的角色，它作为 token 的传输工具，负责在用户的浏览器和我们的服务器之间传递 token。每当用户登录成功后，系统会生成一个独特的 token，这个 token 是对用户身份的一种验证。然后，我们将这个 token 存储在 cookie 中，并将 cookie 发送回用户的浏览器。用户的浏览器会将这个 cookie 保存下来，这样，每次用户发起请求时，浏览器都会自动将这个 cookie 附加到请求中，从而将 token 传递给我们的服务器[16]。当服务器接收到用户的请求时，它首先会检查请求中是否包含

了 cookie。如果请求中包含了 cookie，服务器就会从中提取出 token，并对这个 token 进行校验。通过这种方式，服务器可以确定用户是否已经登录，以及用户的身份信息。

图 5-4 新建应用

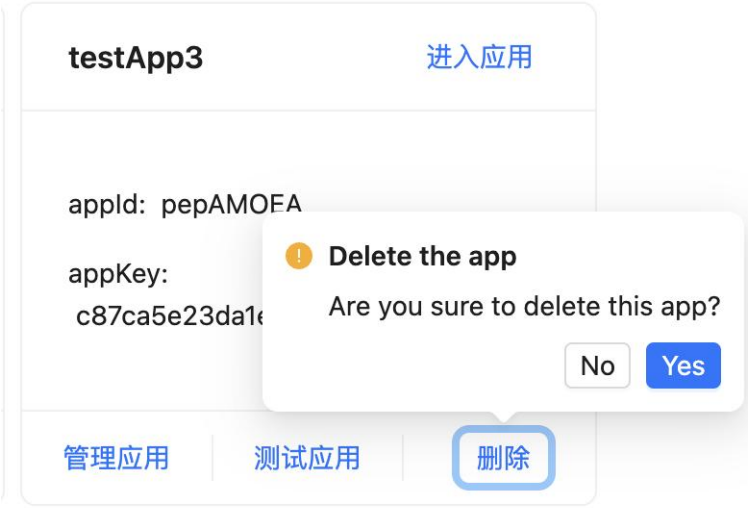

图 5-5 删除应用

如图 5-4 所示，新建应用之后系统会生成如图 5-5 所示的 appId 和 appKey，系统会自动生成两个重要的凭证，即应用 ID（appId）和应用密钥（appKey）。这两个凭证在后续的数据上报过程中起着至关重要的作用。应用 ID（appId）是一个独特的标识符，确保了每个应用程序都有一个独一无二的身份，确保只有你的应用程序可以向系统报告数据。应用密钥（appKey）是与应用 ID 配对的另一个重要凭证。它是一个安全的、随机生成的字符串，用于确保数据的安全性和完整性。

5.3.2 异常数据模块实现

本模块的主要功能是将异常数据进行记录和管理，图 5-7 和图 5-8 展示了运行时错误的详细信息。具体来说，它首先会将检测到的异常数据收集起来，然后以列表的形式进行整理和存储。这样做的好处是，用户可以一目了然地看到所有的异常数据，方便他们进行查看和管理。

如图 5-6 所示，本模块还提供了数据可视化的功能。通过图表的形式，用户可以轻松地看到数据的整体情况和变化趋势。这种直观的方式，可以帮助用户更好地理解和分析数据，从而做出更合理的决策。提供了时间查询功能。用户可以根据自己的需要，选择特定的时间段进行查询。这样，用户就可以在大量的数据中，快速找到自己需要的信息，大大提高了工作效率。

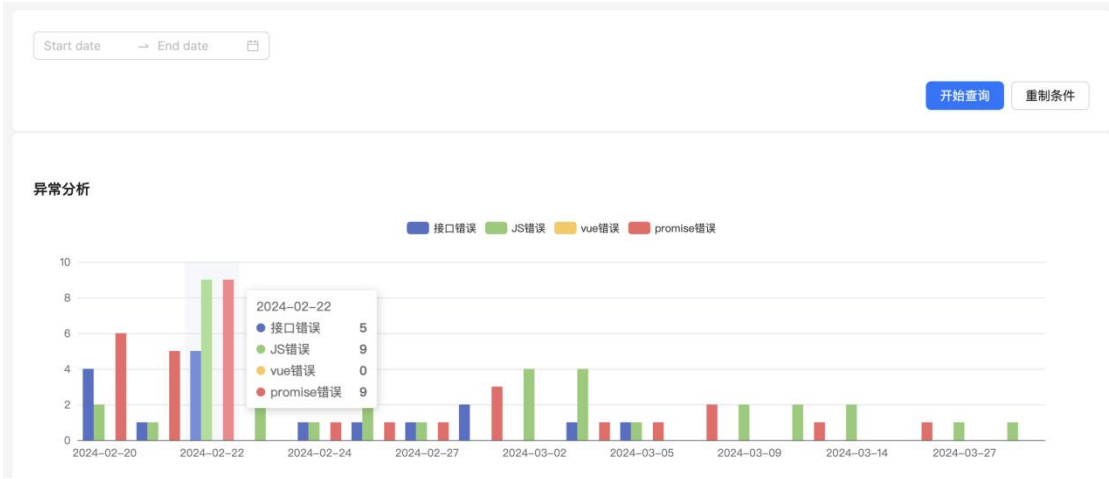

图 5-6 异常数据可视化

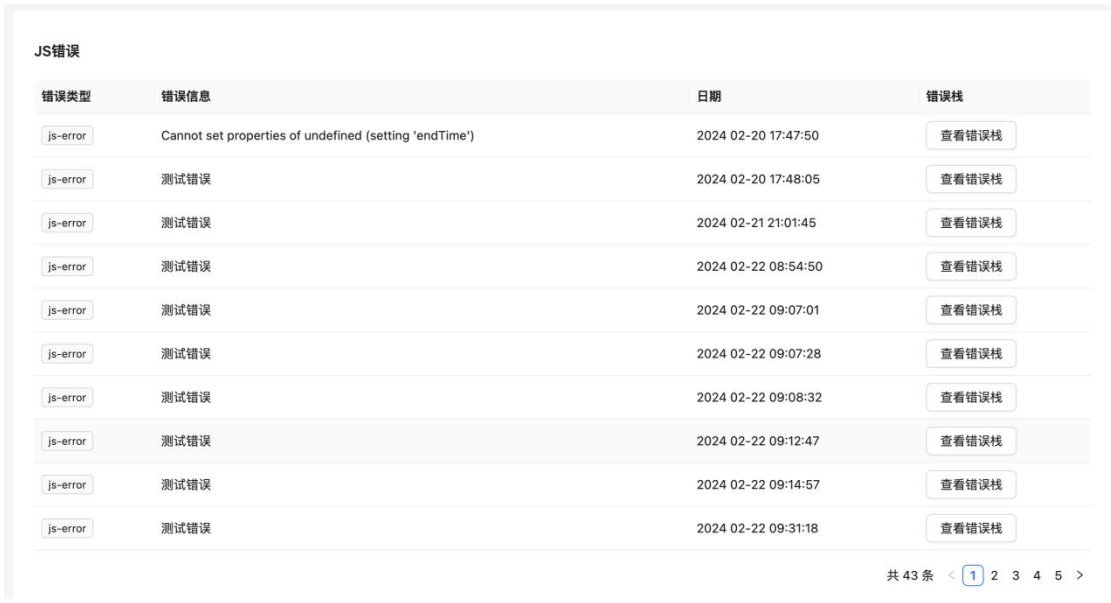

图 5-7 异常数据列表

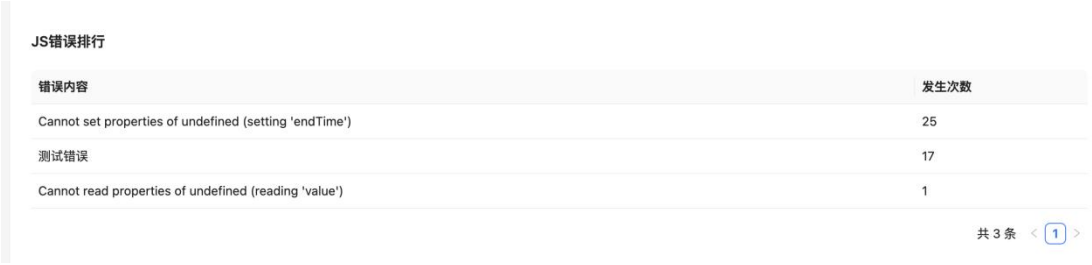

图 5-8 错误排行

5.3.3 页面性能模块实现

本模块将采集到的页面性能指标（CLS、FID、LCP）通过可视化的方式展示出来，如图 5-9 所示。统计时，使用均值来衡量性能可能会受到极端值的影响，这可能会扭曲整体的性能评估[17]。为了解决这个问题，可以采取分位数来统计量化。分位数是一种统计量，用来表示在一组数据中有多少比例的数值小于或等于这个数值。具体来说，第 P%百分位数是指将一组数据从小到大排列后，位于 P%位置的数值。这意味着在这个数值之下的数据占总数据的 P%。比如 50%的用户 FID 延迟都在 100ms 内，那么 100ms 就是 FID 的 50 分位数。Google 推荐使用 75 分位数来衡量以上指标。

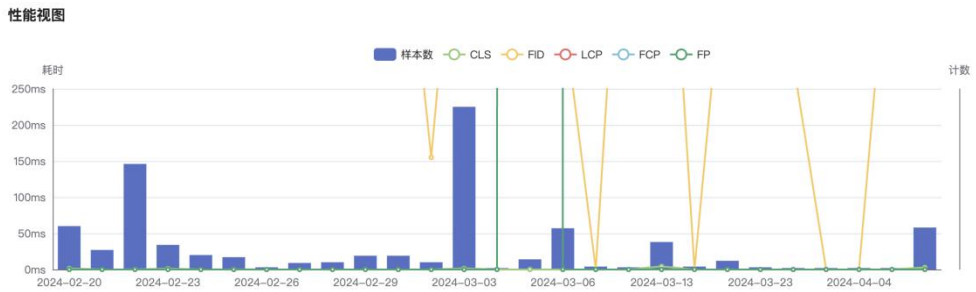

图 5-9 页面性能指标可视化

5.3.4 map 映射文件模块实现

一般来说，前端生产环境部署的代码是经过混淆与压缩的，混淆是一种使代码难以阅读和理解的过程，而压缩则是减少代码体积的方法[18]。这两种技术通常会将整个代码文件压缩成一行，当在生产环境中出现问题时，由于所有的代码都被压缩在一行内，这会导致 SDK 采集到的异常数据中的行列数信息变得不准确。这是因为，原本在源代码中分布在不同行的代码，在压缩后都位于同一行，因此，当出现错误时，错误发生的实际位置与采集到的行列数之间会存在偏差。

图 5-10 压缩后报错信息

图 5-11 未压缩报错信息

SDK 通常运行在生产环境下的，采集到的错误信息类似图 5-10，为了让开发者能够定位源码错误位置，希望采集到如图 5-11 所示的准确错误信息。需要借助 source map 映射文件[19]，source map 是一种映射关系，它将压缩后的代码与原始源代码进行对应，使得开发者可以通过查看压缩后的报错信息，快速定位到源代码中的错误位置。通过使用 source map 映射文件，我们可以将报错信息中的文件名和行号还原为原始源代码中的对应位置，从而帮助开发者更加准确地定位和修复错误，本系统可以上传 source map 文件，从而帮助开发者查看异常发生的

真实位置。

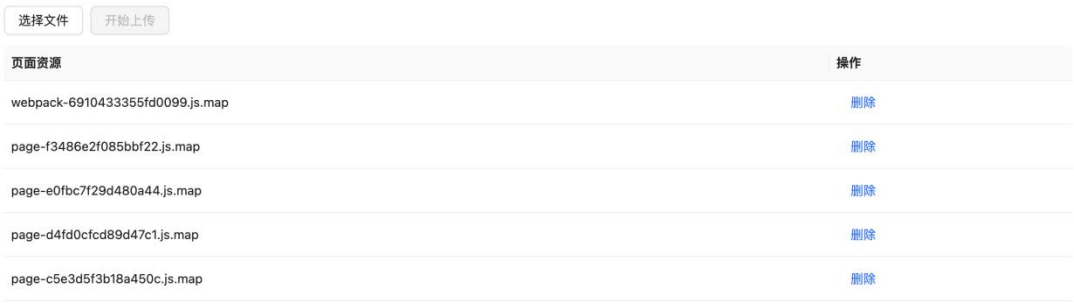

图 5-12 map 文件管理界面

用户在图 5-12 的 map 文件管理界面上传了 map 文件后，在查看具体错误堆栈时，后台会找到对应的 map 文件，自动进行还原，如图 5-13 所示。

图 5-13 错误还原界面

5.3.5 页面访问模块实现

PV 代表了用户对网站中每个网页的访问次数。每当用户在浏览器中加载一个网页，PV 就会增加一次。这个指标可以帮助网站管理员了解用户对网站内容的感兴趣程度，以及用户的访问行为模式[20]。例如，如果一个网页的 PV 数很高，可能意味着该网页的内容比较受欢迎，吸引了用户多次访问或者长时间停留。相反，如果 PV 数较低，则可能需要进一步分析原因，是否是内容不够吸引人，或者是用户体验方面存在问题。图 5-14 将每日页面访问量可视化，图 5-15 展示访问量最多的部分页面。

图 5-14 PV 可视化

访问页面 TOP 视图

页面 URL	页面访问量
http://www.weiwei8848.com:3000/appList	53
http://www.weiwei8848.com:3000/WXfvhDdQ/testApp	48
http://www.weiwei8848.com:3000/WXfvhDdQ/errorProfiler	44
http://weiwei8848.com:3000/appList	38
http://www.weiwei8848.com:3000/WXfvhDdQ/performance	37
http://www.weiwei8848.com:3000/WXfvhDdQ/resource	36
http://www.weiwei8848.com:3000/WXfvhDdQ/pv	36
http://www.weiwei8848.com:3000/WXfvhDdQ/mapManager	32
http://weiwei8848.com:3000/WXfvhDdQ/performance	21
http://weiwei8848.com:3000/	20

共 62 条 < 1 2 3 4 5 6 7 > 10 / page

图 5-15 页面访问量排行

6. 总结与展望

6.1 全文总结

本文的核心内容聚焦于前端监控 SDK 及其与平台整合的基础实现。在 SDK 的开发部分，我们首先对 SDK 的设计思路进行了细致的梳理，以确保其结构清晰、易于理解和维护。在设计过程中，我们采用了模块化的开发方法，这意味着 SDK 的各个组成部分都是独立的模块，每个模块负责不同的功能，这样的设计使得 SDK 更加灵活和可扩展。同时，我们还引入了插件化的扩展机制，这允许开发者根据项目的具体需求，方便地添加或移除特定的功能模块，从而使得 SDK 不仅能够适应当前的开发需求，还能够轻松应对未来可能出现的新需求，确保了其长期的可用性和前瞻性。在确立了 SDK 的设计理念和开发模式之后，我们对 SDK 的功能需求进行了深入的分析。这一步骤是为了确保 SDK 能够满足前端监控的关键需求，具备必要的核心功能。

SDK 实现部分，简单阐述了各个模块及插件的功能以及实现思路，最后给出部分伪代码加以说明，引入了两种 FMP 计算方式，一种是基于 DOM 权重，一种是基于 DOM 变化趋势。这两种计算方式各有优点。基于 DOM 权重的计算方式可以更准确地反映 DOM 节点的重要性，而基于 DOM 变化趋势的计算方式可以更好地捕捉到 DOM 节点的变化。

然而，这两种计算方式各有缺陷。基于 DOM 权重的计算方式可能会忽视一些权重较低的但重要的 DOM 节点，而基于 DOM 变化趋势的计算方式可能会受到噪声的影响。

6.2 未来展望

本文实现的 SDK 与监控平台实现了最基本的功能，满足一般使用。但仍存在一些问题有待优化。

首先，数据压缩与传输的安全性是一个重要的优化方向。在数据传输过程中，为了保证数据的完整性和安全性，需要对数据进行有效的压缩和加密[21]。目前，虽然已经有一定的机制来保护数据，但随着网络攻击手段的不断升级，我们需要进一步强化数据的安全性，例如采用更高级的加密算法，或者实现更为复杂的数据验证机制。

其次，异常分析模块的增强也是提升系统性能的关键。异常分析模块负责识别和处理系统中的异常行为，这对于维护系统稳定性至关重要。当前的异常分析模块可能还无法完全覆盖所有的异常情况，或者在处理某些复杂异常时效率不高。因此，我们需要对异常分析模块进行深入的优化，提高其识别的准确性和处理的效率，以便能够更快地响应并解决潜在的问题。

再者，FMP 计算算法的复杂度优化也是一个有待解决的问题。对于监控系统来说，快速准确地计算出 FMP 值是非常重要的。如果计算算法过于复杂，将会增加系统的计算负担，从而影响到整体的时间开销。因此，我们需要对现有的 FMP 计算算法进行优化，寻找更加高效的算法，以减少不必要的计算量，尽可能地降低时间开销。

参考文献

[1] Farney T. Getting the best Google analytics data for your library[J]. Library Technology Reports, 2016, 52(7): 5-8.

[2] Hamon D. Comparing In-Browser Methods of Measuring Resource Load Times[J]. W3c Workshop on Web Performance, 2012.

[3] Vasilijević V, Kojić N, Vugdelija N. A new approach in quantifying user experience in web-oriented applications[C]. 4th International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture–ITEMA, 2020: 9-16.

[4] 张钊源,刘晓瑜,鞠玉霞. Node.js 后端技术初探[J]. 中小企业管理与科技,2020(22):193-194.

[5] 唐斌斌,叶奕. Vuejs 在前端开发应用中的性能影响研究[J]. 电子制作,2020(10):49-50,59.

[6] 皮彬睿, et al. "一种可扩展插件化体系架构的软件系统设计方法.", CN113157335A. 2021.

[7] 温丽梅,梁国豪,韦统边,等. 数据可视化研究[J]. 信息技术与信息化,2022(5):164-167. DOI:10.3969/j.issn.1672-9528.2022.05.041.

[8] 康长安. "基于前端的 Web 性能优化." 电脑知识与技术：学术交流 006(2011):007.

[9] 班晏子婧,李晖,陈梅,等.面向 Web 应用的智能监控系统的设计与实现[J].智能计算机与应用, 2022, 12(9):7.

[10] 杨晨. Web 前端异常响应监控与 WRG 生成系统的设计与实现[D]. 四川:电子科技大学,2023.

[11] 郭春霞. 基于 Spark 的 Web 应用前端性能监控系统的设计与实现[D].北京交通大学,2023.DOI:10.26944/d.cnki.gbfju.2022.000691.

[12] 崔娟,王伟民,冯继虎,等. 基于 Nginx 反向代理解决公网上服务跨域问题的研究[J]. 现代信息科技,2023,7(8):79-82,87. DOI:10.19850/j.cnki.2096-4706.2023.08.020.

[13] 王鑫琦,李昕.一种用于收集日志数据的消息系统的设计和实现[J]. 2017.

[14] 何伟,王浩,蔡忠平. 基于 Kafka 的公交 GPS 数据实时通讯系统的设计与应用[J]. 城市公共交通,2022,288(6):34-37,41. DOI:10.3969/j.issn.1009-1467.2022.06.014.

[15] 唐新,欧阳尔.基于 Java Web Token 的单点登录技术研究与实现[J].中文科技期刊数据库

（全文版）工程技术, 2021.

[16] 高春庚.基于 Cookie 的会话跟踪技术及应用[J].信息记录材料, 2021, 22(10):2.

[17] 李君元.结构可靠性计算方法性能比较研究[D].内蒙古工业大学[2024-05-24].DOI:CNKI:CDMD:2.1018.795385.

[18] 王洪哲,关锋,丁兆俊,et al.一种基于前后端分离架构的前端双随机多态混淆方法:CN202310238451.8[P].CN115935303A[2024-05-24].

[19] Herrero A , Avallone A .Sourcemap: a graphical representation to enhance the low frequency source radiation[J].Geophysical Journal International, 2022(3):3.DOI:10.1093/gji/ggac134.

[20] Rodwell M J , Hoskins B J .A Model of the Asian Summer Monsoon.Part II: Cross-Equatorial Flow and PV Behavior[J].Journal of the Atmospheric ences, 1995, 52(9):1341-1356.DOI:10.1175/1520-0469(1995)0522.0.CO;2.

[21] 丁晓娇.可保护数据隐私性和完整性的安全数据融合算法研究[D].河南大学[2024-05-24].DOI:CNKI:CDMD:2.1015.650856.

致谢

当我将最后一笔落在论文上，那一刻，我深刻地意识到，我的学生生涯即将画上句号。借此致谢，总结过去，向往未来。

教海如春风，师恩似海深。感谢在深技大遇到的每一位老师，他们的一言一行，都在无形中塑造着我，让我不断成长和变化。特别感谢我的指导老师郑俊虹，感谢您接受我的自命题毕设，感谢您在论文撰写期间给予我的帮助，在初稿修改过程中，不厌其烦地给我很多修改建议。愿老师们身体健康，工作顺利。

从大专到本科是自我改变的过程，自认为自己还算是努力，还算是走运。但巧妇难为无米之炊，感谢深技大提供了如此优越的学习和成长平台将我带到了一个全新的高度。在这里，我的视野得到了极大的开阔，我也改变了对问题的思考方式。

父母本是在世佛，何须千里拜灵山。得知我将要去北京实习，感谢父母给予我资金和精神上的支持。初到北京，夜深时分最为思家，但我明白这是离开校园，走向社会的的必经之路。独自一人处理各种事务，我深感父母早期培养我独立和生活技能的重要性。父母虽未受过高等教育，但也尽了最大的努力给我最好的，养育我并支持我的学业。我站在父母的肩上，看到了更广阔的世界。希望有一天父母能踩着我的肩膀，享受世间繁华。

秋招失利，春招勉强上岸。经历了找工作的挫败与焦灼后，我才真正明白：生活的路漫漫其修远兮，每一步都不会轻松；但我对未来充满了期待和勇气。我深信无论前路如何，只要我保持着学习和探索的心态，勇敢地迈出每一步，生活的美好总会在不经意间绽放。