|  |
| --- |
| **学士学位论文** |
| **前端性能监测系统设计与实现** |
| |  |  | | --- | --- | | 学 号： | 20141000313 | | 姓　 名： | 吴锋 | | 学 科 专 业： | 遥感科学与技术 | | 指 导 教 师： | 刘副江 副教授 | | 培 养 单 位： | 信息工程学院 | |
| 二○一八年六月 |
|  |

原创性声明

中国地质大学（武汉）学士学位论文原创性声明

本人郑重声明：本人所呈交的学士学位论文《前端性能监测系统的设计与实现》，是本人在指导老师的指导下，在中国地质大学（武汉）攻读学士学位期间独立进行研究工作所取得的成果。论文中除已注明部分外不包含他人已发表或撰写过的研究成果，对论文的完成提供过帮助的有关人员已在文中说明并致以谢意。

本人所呈交的学士学位论文没有违反学术道德和学术规范，没有侵权行为，并愿意承担由此而产生的法律责任和法律后果。

学位论文作者签名：

日 期： 年 月 日

**前端性能监测系统的设计与实现**

**摘要**

现代计算机技术和网络技术的快速发展促进了web应用的蓬勃发展，web应用在电子商务、娱乐、生活等方面起着越来越重要的作用。近年来Web应用数量激增，各大公司通过web应用这种媒介展示和传播自己的服务，由于市场需求，应用的用户规模和用户需求也日益庞大。在这种情况下，对前端的性能要求也越来越高，用户关注的不仅仅是网页本身所能呈现的内容，还对页面本身的加载速度有要求，漫长的网页加载等待很大程度上会导致用户失去耐心而直接关闭该网页，欠佳的用户体验对企业利益来说是极大的打击。所以，前端性能优化的工作势在必行。

本文首先研究了前端性能优化的相关理论和技术，包括浏览器工作原理、性能优化原理和性能优化方案。然后针对性能优化需求，提出前端性能检测系统Lighthouse的设计与实现方案。最后探究了Lighthouse的数据存储和数据可视化分析。

在相关理论与技术方面，主要探讨了浏览器是如何工作，并根据浏览器的工作原理分析性能优化原理。同时还探究了本系统将采集的性能数据指标，并如何应用在可视化表达中。

在Lighthouse系统的前端架构方面，研究并分析了浏览器API、pageSpeed API接口、基于react+redux的开发模式、数据可视化技术，并通过这些技术开发出一个用于监测网页性能指标，并给出相应优化方案的前端性能监测系统（Lighthouse）。

**关键词**：React Flux 性能优化 RESTful API 数据可视化

**ABSTRACT**

With the clipping development of Modern computer technology and network technology, web applications are playing an increasing important role in E-commerce, entertainment as well as our daily routine. In recent years, the number of web applications have surged, and many Intenet companies spread their products and services through the web applications. However, the increasing market demands as well as the growing users require better and faster performace for the front end applications. Because users focus not only on the content on the page but also how fast they can be delivered. To a larger extent, user may be wore out because of the slow progress, which can be a great blow to corperate interests. All in all, it is imperative to do the front-end performance optimization.

In this article, we studied related theories and techniques on front-end performance optimization, which included the principle of browser, performance optimization and solutions. Nextly, the paper analyzed the design and implementaton of front-end performance monitoring system. Finally, the article discussed some aspects of data storage and data visualization analysis.

In terns of performance optimization related theories and techniques, the paper mainly studied how the browser work, from which front-end performance optimization principle and method can be discovered. Also, explore a list of performace indicatiors data and analyzed how it can be applied in visual expressions.

In the aspect of the architecture of front-end performance monitoring system, the chapter analyzed the react-redux-based development mode, PageSpeed API, performance.timing API as well as data visualization techniques. Based on all the research above, a system that monitoring web page performance indicatiors and providing corresponding optimization solutions were developed.

**Key words:** React, Flux , performance optimization, RESTful API, data visualization

**目录**

[第一章 引言 1](#_Toc515223344)

[1.1 研究背景 1](#_Toc515223345)

[1.2 研究现状 1](#_Toc515223346)

[1.3 研究内容 2](#_Toc515223347)

[1.4 论文组织结构 3](#_Toc515223348)

[第二章 性能优化理论和技术分析 4](#_Toc515223349)

[2.1 浏览器工作原理 4](#_Toc515223350)

[2.1.1 渲染引擎 4](#_Toc515223351)

[2.1.2 解析和DOM树构建 5](#_Toc515223352)

[2.1.3 渲染树构建 5](#_Toc515223353)

[2.1.4 布局与绘制 6](#_Toc515223354)

[2.2 性能优化原理与方案 6](#_Toc515223355)

[2.2.1 优化CSS 6](#_Toc515223356)

[2.2.2 JavaScript优化 7](#_Toc515223357)

[2.2.3 HTTP缓存 8](#_Toc515223358)

[2.3 性能指标数据采集 9](#_Toc515223359)

[2.3.1 使用浏览器API采集性能数据 9](#_Toc515223360)

[2.3.2 使用PageSpeed API采集性能数据 9](#_Toc515223361)

[第三章 前端性能监测系统设计与实现 11](#_Toc515223362)

[3.1 Flux架构 11](#_Toc515223363)

[3.1.1 Flux和MVC模式 11](#_Toc515223364)

[3.1.2 Redux架构的应用 13](#_Toc515223365)

[3.2 React组件化开发 14](#_Toc515223366)

[3.2.1 组件化开发的优势 14](#_Toc515223367)

[3.2.2 容器和展示组件 15](#_Toc515223368)

[3.2.3 Ant Design组件库的使用 15](#_Toc515223369)

[3.2.4 React中的数据通信 16](#_Toc515223370)

[3.3 功能设计与实现 18](#_Toc515223371)

[3.3.1 整体设计 18](#_Toc515223372)

[3.3.2 路由设计 19](#_Toc515223373)

[3.3.3 主页设计 20](#_Toc515223374)

[3.3.4 分析页设计 21](#_Toc515223375)

[第四章 数据存储与可视化 25](#_Toc515223376)

[4.1 本地存储技术 25](#_Toc515223377)

[4.1.1 Cookie存储 25](#_Toc515223378)

[4.1.2 localStorage存储 25](#_Toc515223379)

[4.2 数据可视化研究 26](#_Toc515223380)

[4.2.1 均值的局限性 26](#_Toc515223381)

[4.2.2 箱型图分析 26](#_Toc515223382)

[4.2.2 数据可视化工具——Bizchart 29](#_Toc515223383)

[第五章 总结与展望 30](#_Toc515223384)

[4.1 总结 30](#_Toc515223385)

[4.2 展望 30](#_Toc515223386)

[致谢 32](#_Toc515223387)

[参考文献 33](#_Toc515223388)

**第一章 引言**

**1.1 研究背景**

近年来随着web技术的发展和用户数量的激增，web应用的性能成为各大企业关注的一个焦点，美观、流畅、交互性好的用户体验能帮助企业吸引更多的用户，带来更多的企业利润；然而卡顿、缓慢、过长等待的用户体验可能会造成用户量的流失，这对企业来讲是非常巨大的损失。

对于很多企业来讲，前端网页的性能在一定程度上与企业收益有很大的相关性。国外有相关的调研数据显示[1]：Google的加载延迟了400ms将会导致搜索量下降0.59%；Bing加载延迟2s将会导致收入下降4.3%；Yahoo延迟400ms，用户流量将会减少5-9%；Netflix开启Gzip压缩模式可将性能提升13.25%，带宽减少50%。美国研究生项目资讯网站的调查结果显示，网页的加载时间超过4秒将会导致四分之一的人放弃打开该网页[2]。调查机构KissMitrics研究发现：网页加载速度影响用户消费，如果电子商务每天收入为10万美元，那么1秒的延迟将会让该网站每年损失250万美元[3]。

国内很多公司也相应开发出一些针对自己网站的监控系统，如阿里巴巴、美团、腾讯、百度等互联网大公司，网站的用户体验对他们来说尤为重要。与此同时，国内的网速与许多发达国家相比慢很多，中国大陆的网速在世界排名第71，平均网速为1.774Mbps，这个数字远远低于世界平均水平，因此，在国内较低网速的环境下，对Web应用进行性能优化显得尤为重要[5]。

**1.2 研究现状**

Google公司作为一个队速度有着极致追求的互联网巨头，对网页的性能优化非常的重视，谷歌的Chrome浏览器自带的开发者工具本身就是一个网站性能监测分析工具。在网络面板中可以查看网络请求资源的实时信息，明确和定位哪些比预期加载更耗时的请求以便针对性的进行请求优化。在Timeline面板中可以整体上看到web页面加载和被使用过程中时间消耗在哪里，所有的时间从加载资源到解析JavaScript都会被标记在时间线上。本系统使用的PageSpeed API 也是Google提供的一套性能分析接口，该工具是从页面的加载时间来衡量网页性能。

雅虎作为过去的互联网巨头，是Web前端优化方面的先驱。雅虎提出了在前端优化方面非常著名的“雅虎军规34法则”[4]，这些法则到现在还影响着Web开发人员。另外，雅虎的首席性能工程师Steve Souders根据性能优化经验编写了《High Performance Web Sites》以及其他另外8位Web前端专家一起编写的《Even Faster Web Sites》提供了提升网站性能的最佳实践和使用建议[6]。

在网页性能分析工具方面，比较出名的有以下几个工具：PageSpeed, Yslow和SpeedTracer，它们主要对页面进行评分，然后会显示各项指标的修改意见；WebpageTest，一个非常权威的在线性能测试的网站，可以通过输入的URL生成一个URL加载的时间瀑布图，并针对所有资源（JS，CSS，Image等）列出优化清单。

国内许多互联网巨头在Web性能优化方面也非常关注，在工程实践中采取了许多有针对性的优化方案，如针对请求数量优化、请求带宽优化、浏览器缓存利用、页面结构优化等方案。

**1.3 研究内容**

本文首先对浏览器的工作机制进行探究，以此从渲染引擎，解析和DOM树的构建，渲染树构建和布局与绘制四个渐进的过程进行探讨，通过了解浏览器渲染原理，再进行性能优化原理的探讨，进而得出性能优化的方案，然后针对性能优化需求，明确Lighthouse系统需要采集并分析的性能指标数据。

然后本文重点研究了Flux架构、基于react的组件化开发模式和Lighthouse系统的功能设计与实现三大方面。

在Flux架构方面，研究了Flux架构和传统的MVC模式的异同点，并引出并探究Flux架构中最常用的redux框架，然后探究了如何在react应中使用react-redux进行数据的持久化。

在react组件化开发方面，本内容研究了组件化开发的起源及优势，接着提出容器和组件的概念，探究该如何对业务逻辑拆分成容器和组件。然后介绍了项目中使用到的蚂蚁金服的一套UI库Antd Design及其具体使用方法。最后探究了软件开发中最常见的数据通信问题，并详细探讨了本项目中使用的react中的数据通信。

在前端性能监测平台的功能设计与实现方面，研究了整体的设计，路由设计，入口主页设计，以及可视化分析页设计，其中这些设计包括页面的响应式布局设计，容器和组件的设计，数据存储和分发的设计。

最后，在数据存储与可视化方面，研究了本平台使用的Cookie存储和localStorage存储两种本地存储技术。在数据分析与可视化方面探究了数据的箱型图分析方法和平台使用的Bizchart可视化工具。

**1.4 论文组织结构**

全文总共分为五个章节，针对每章内容做如下研究：

第一章，引言。分别概述了本文的研究背景，前端性能优化的研究现状，具体的研究内容以及研究的结构和安排。

第二章，性能优化理论和技术分析。分别探究了浏览器工作原理，Web性能优化原理及方案，性能指标数据采集三个方面，为系统的设计和性能优化研究提供理论基础。

第三章，前端性能监测系统的设计与实现。分析Flux架构与传统MVC模式的异同及Flux架构在开发中的有点，接着探究本项目使用的react+redux开发模式，然后分析并探讨了基于react的组件化开发模式，并详细阐述了前端性能监测平台的功能设计与实现。

第四章，数据存储与可视化。研究平台使用的本地存储技术，然后探讨了对性能指标数据的分析方法和可视化方法。

第五章，总结与展望。本章对论文全篇所研究和实现的内容进行了总结，提出研究成果所存在的不足之处，并对未来性能优化方面的学习和研究进行展望。

**第二章 性能优化理论和技术分析**

这一章内容首先研究了浏览器的工作机制，然后基于浏览器的渲染原理，对前端性能优化原理进行探究并得出性能优化方案。通过以上的理论基础，探究本课题开发的项目所需采集的前端性能指标以及相关的采集方法。

**2.1 浏览器工作原理**

浏览器的功能主要是承载用户操作，将所有操作映射成发送到服务器端的请求，并将返回的资源呈现在浏览器窗口中，其中资源的地址由用户使用统一资源标示符（URL）来指定。理解浏览器工作原理可以通过以从浏览器的渲染流程进行分析：

**2.1.1 渲染引擎**

渲染引擎主要功能是将请求到的资源解析并组合最终显示在浏览器内容窗口上。市面上不同的浏览器有不同的渲染引擎，所以它们的渲染流程会有差异，本文研究中以最常用的Webkit渲染引擎作为研究对象。如下为Webkit渲染引擎的基本工作流程：

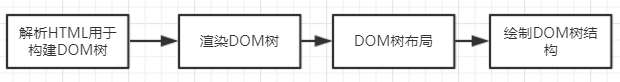


图2.1.1 渲染引擎的基本流程

根据上图所示，渲染引擎首先解析服务器返回的HTML文档资源，生成许多DOM元素对象节点，这些节点用于后期生成DOM树结构。与此同时，样式资源也被解析并生成样式规则表，然后与上一步骤中生成的DOM树上的节点进行匹配并生成渲染树。

在渲染树生成后，开始进入布局处理阶段，这个阶段是对DOM元素进行大小与位置的确定，例如根据CSS文件中的position、width、height等属性，然后根据这些CSS规则进行DOM树的布局。

需要提及的是，渲染引擎的工作是一个逐步完成的过程。为了得到更好的用户体验，浏览器的解决方案是让先解析完成的内容进行绘制和布局，而不必等到整个HTML文档解析完整后再去构建和布局渲染树。

如下为Webkit渲染引擎的主流程：

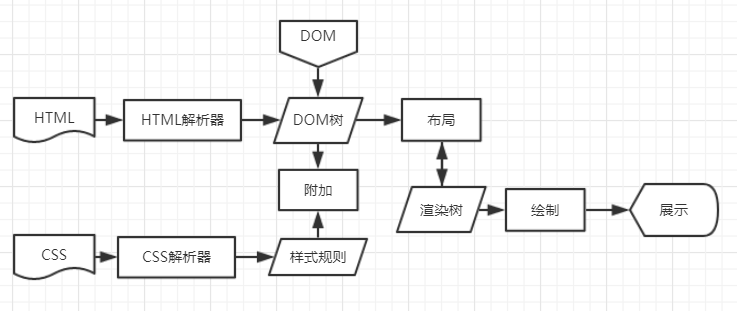


图2.1.2 Webkit引擎主流程

**2.1.2 解析和DOM树构建**

解析是渲染引擎中一个非常重要的过程，它将资源文档翻译成机器可以理解的内容。解析所得到的结果通常是代表了文档结构的节点树，也就是解析树或语法树。

解析的过程可以分成两个子过程：词法分析和语法分析，词法分析是将输入内容分割成标记的过程，标记就是构成内容的单位；语法分析是应用语言的语法规则的过程[9]。解析是一个迭代的过程，解析器会从分析器中请求一个符号，并将其与一条语法规则进行匹配，若匹配成功，则将该符号相对应的节点添加到解析树上，然后再进行相同的操作。若是没有匹配到规则，解析器会将符号存储起来，并请求下一个符号，知道找到所有的内部标记与其匹配的规则。

在生成解析器的同时，作为DOM树根节点的Document对象也会被创建。在后续的树构建过程中，DOM树会因为不断有新解析出的DOM节点加入而不断修改，直到最后一个DOM元素添加完毕，便形成了最终的DOM树结构。

**2.1.3 渲染树构建**

渲染树是在DOM树构建的同时，浏览器通过可视化元素按照其显示顺序而构建的文档的可视化表示，也就是我们所看到的页面真实内容。浏览器在构建渲染树过程中大致完成以下工作：

· 从DOM树的根节点开始遍历每个可见节点（不可见节点不会体现在渲染输出中）

· 为每个可见节点适配相应的规则，并应用它们。

· 构建可见节点，连同其内容和计算的样式。

* 最终输出的渲染树包含了屏幕上所有可见内容及其样式信息。

另外需要注意的是，渲染树和DOM元素是相对应的，但非一一对应的关系。非可视化的DOM元素不会插入呈现树中，例如将一个元素的display属性设置”none”时，该元素便脱离了文档流，渲染树便不会将其包含在树结构中[10]。

**2.1.4 布局与绘制**

布局过程是浏览器计算每个DOM元素最终在窗口显示的大小和位置。页面中每个元素的布局是相对的，所以在用户操作下使得一个元素的位置有了变动，会导致其他元素的布局产生变化，进而使得浏览器进行重新布局。但是如果每次微小的元素变动都导致整个布局重排的话，将会对浏览器造成很大的性能损耗。所以针对这种情况，浏览器使用的是“dirty”位系统作为解决方案，它能够有效降低多余的重排次数。

绘制是最后一个阶段，将渲染树中每个节点转换成屏幕上的肉眼可见的像素的过程。CSS2规范定义了绘制流程的顺序[11]，其顺序就是元素进入堆栈样式上下文的顺序。和布局类似，在用户的细小操作而导致小部分页面发生变化时，浏览器会花费最小代价对页面进行重绘。如果是页面发生较大的改变（例如HTML的结果发生变化），则会对整个渲染树进行重新布局和绘制；若只改变其中一个元素，则只对该元素进行布局和重绘。

**2.2 性能优化原理与方案**

浏览器在渲染出页面前会先生成DOM树和CSSOM树（叠层样式对象模型），这两棵树是通过HTML、CSS和JavaScript资源构建的。这三种资源都会对页面渲染产生阻塞影响，所以出去必要的HTML资源，可以对CSS和JavaScript两方面进行优化，尽可能减少阻塞的影响。

**2.2.1 优化CSS**

（1）使用媒体查询

媒体查询相当于条件表达式，可以让某一段CSS代码在特定条件下使用，这样只有在特定条件下才会让浏览器进行阻塞渲染然后构建CSSOM树，而不是一开始就进行CSSOM树的构建。

（2）使用CSS压缩

压缩可以分为常用的CSS压缩工具压缩和Gzip压缩。CSS压缩工具有如YUI Compressor、CSS Compressor、CSS driver等，其原理很简单，主要是将CSS文件中没用的空白符、注释等无效信息删除，以达到缩减字符个数的目的。Gzip压缩是一种流行的文件压缩算法，在现在的应用中十分流行，尤其是在Linux平台，这个方式不仅仅针对CSS资源，当使用Gzip压缩一个纯文本时，大约可以减少70%以上的文件大小（这也取决于文件的内容）[12]。

（3）将CSS文件放在头部中

将CSS放在页面顶部有利于减少页面的重绘和重排。当浏览器下载资源并构建DOM树结构时，会根据浏览器默认及现有样式规则生成渲染树来渲染页面，当遇到新解析出来的样式规则时将下载并合并现有CSS并重新生成渲染树，这种情况下就导致了多次渲染之前的内容。若是将所有CSS都放在页面顶部，这就避免了多次渲染的问题。

**2.2.2 JavaScript优化**

JavaScript实现了Web应用的网页逻辑，若根据该语言的特性并加以利用，将会降低JavaScript脚本运行时的内存消耗，从而提升页面的加载速度[13]。JavaScript优化方法非常多，范围也非常广，本节主要针对项目中使用到的方法进行研究。

（1）DOM事件绑定

很多情况下需要对一系列相同的DOM元素绑定相同的事件。比如一个无序列表内有多个li标签需要绑定相同的点击事件，这种情况下最好避免使用传统的使用一个循环的方式进行事件绑定，这会大大浏览器降低性能。可以通过JavaScript的事件冒泡机制，直接在父级元素中绑定需要的事件，当点击任意一个子元素时通过逐层冒泡并被父元素捕获。这样不仅可以减少代码量，而且在增加新的子元素时依旧可以捕获该事件[14]。

（2）防止JavaScript阻塞

浏览器在解析时是先对HTML进行解析，如果使用内联JavaScript（嵌套在HTML中的JavaScript），就会造成JavaScript阻塞渲染。为了防止这一问题出现，一种方法是将<script>标签标记为异步的。我们可以将那些在启动渲染时不必要的脚本标记为异步加载状态，而不阻碍页面结构和样式的渲染。

**2.2.3 HTTP缓存**

通过网络来获取资源是非常缓慢的，如果文件体积过于庞大，浏览器可能还行进行多次的往返传输才能完全下载该资源。因此，利用缓存机制来减少多次重复请求是性能优化的一个关键方面。

浏览器自带了HTTP缓存功能，只需确保每个服务器响应的头部包含以下属性：

（1）ETag：Etag是一个与web资源相关的记号标志，它对资源的更新进行检查，如果资源未发生变化时服务器就不会向客户端发送数据。Etag从服务端生成，在客户端中会判断所请求的资源是否被修改，当服务器将返回“304 Not Modified”的响应时，表明浏览器不必重新下载资源而是继续复用缓存。

（2）Cache-Control: Cache-Control定义了缓存的策略，它规定在什么条件下可以缓存响应以及可以缓存多久。可以设置它的属性以达到想要的缓存效果，例如，可以标记为public，则表明资源可以被任何一端进行缓存；也可以设置max-age属性为一个较长的时间，使资源在该时间内都不会失效。如下图为定义最佳Cache-Control策略流程图[16]：



图2.2.1 Cache-Control最佳策略（来源：https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching）

**2.3 性能指标数据采集**

**2.3.1 使用浏览器API采集性能数据**

本章主要探究的浏览器API是Performance API，它用于精确度量、控制、增强浏览器的表现。本文研究的性能数据来自于performance中的timing对象。timing对象包含了各种与浏览器性能相关的时间戳数据。我们可以通过这些时间数据计算出不同耗时指标来定量分析网页具体在哪块加载耗时长，以便有针对性的进行性能优化。获取该数据的方式非常简单，甚至可以直接在浏览器控制台中输入performance.timing，点击回车即可看到当前网页的时间数据，如下图为浏览器中获取百度（www.baidu.com）的时间加载数据：



图2.3.1 网页加载时间数据

**2.3.2 使用PageSpeed API采集性能数据**

PageSpeed API能够针对移动设备和桌面设备生产网页的实际性能报告，并能够提供关于如何改进网页的建议。本课题中API的数据以json格式返回，具体将其划分为以下几个方面内容：

（1）速度得分。PageSpeed会整合Chrome用户体验报告（CrUX）[17]中的数据，以显示网页的实际性能数据，并会报告两项指标，即首次内容绘制（FCP）和文档内容加载（DCL）的得分数据。

（2）优化程度得分。PageSpeed会评估相应网页的首屏加载时间和完整网页加载时间，计算出一个得分（分值介于0到100之间）来估算相应网页的性能优化空间。当得分为80分或更高时为良好，表面该网页采用了大多数性能优化最佳做法；得分在60到79之间为中等，该网页未采用某些常用的性能优化建议；得分在0和59之间为偏低，相应网页未经过优化，有较大优化空间。

（3）网页的速度统计信息。该部分显示网页使用阻碍呈现的资源而需要加载的往返次数、相应网页使用的总字节数，以及它们与数据集中往返次数和所用字节数中间值的对比情况，此部分表面，如果改变相应网页的外观和功能，其加载速度可能会变得更快[18]。

（4）优化建议。此部分数据是由PageSpeed API返回的针对测试网站可采用的性能优化最佳实践，由于网络连接性能有很大差异，优化建议针对的是网页性能中与网络无关的方面，例如服务器配置、网页的HTML结构，以及网页使用外部资源情况等。这些建议只是作为参考，网页的绝对性能仍受用户网络连接的影响。

**第三章 前端性能监测系统设计与实现**

本章将从前端性能监测系统所采用的Flux架构进行分析，探究为何采用此种架构模式以及相比其他架构的优势。然后探究了基于react的组件化开发，以及项目中如何对组件结构进行划分。最后详细探究整个系统的设计和功能实现。

**3.1 Flux架构**

**3.1.1 Flux和MVC模式**

（1）MVC模式

传统的MVC架构将程序分成三个角色，从上到下依次为：

·View层，用于承载用户操作、显示数据，是用户可见部分，

·Controller层，对用户的操作进行逻辑处理部分，

·Model层，是程序的数据中心，负责管理并定义数据的用途。

对于一个简单的MVC架构程序来讲，其工作流程如下[18]：



图 3.1.1 MVC工作流程（来源：https://developers.google.com/speed/docs/insights/about）

用户在View上进行操作，在View上触发的操作发送到Controller中，Controller根据其定义的数据行为在Modal中进行数据更新。此时Modal层向UI层的视图发送对应修改的通知，收到通知的视图会进行数据更新，展示新的内容给用户。

（2）Flux模式

Flux框架是Facebook推出的用来代替原有的MVC框架的一套解决方案。用户在View上的操作最终会映射成一类Action，Action传递给Dispatcher，再由Dispatcher执行注册在指定Action上的回调函数。最终完成对Store的操作。如果用户操作触发了Store中的数据变更，则描述该数据的事件变会被触发，View层监听这些事件进而使UI层发生变动。

基本的Flux流程[19]可由如下表示：



图3.1.2 Flux工作流程（来源：https://medium.com/hacking-and-gonzo/flux-vs-mvc-design-patterns）

从工作流程图可看出，Flux架构一共分为四个模块角色。

· View，是Modal的真实可见形式，表示当前显示的视图。不同于MVC的Controller直接调用模型层业务逻辑接口，Flux中来自用户的操作请求先会被转义成对应的Action，再交由Action进行下一步处理。

· Action，用于描述针对某一事件的操作，由键名和键值组成，比如：

{ 'actionType': 'delete', 'data': item}

· Dispatcher，属于整个应用的事件中心，负责分发事件，管理所有的数据流。在接收到Action传来的数据之后，调用Store注册在Action上的回调函数。

· Store，包含程序的数据与业务逻辑，所有的数据变换都发生在Store内部。Store对外只知道如何修改数据，但是不提供修改数据的接口，View层也不允许直接修改Store。

（3）MVC的局限

现实开发场景中通常都是比较复杂的，视图和模型之间可能存在多对多的关系，这种情况下对于MVC开发模式来说会显得比较复杂。单个视图的数据可以来自多个模型，单个模型更新是需要通知多个视图，用户在视图上的操作可以对多个模型造成影响，这种情况下，视图与模型之间的更新可能进入死循环（如图3.1.3）[20]。而Flux通过其单向数据流模式来解决这种复杂性（如图3.1.4）[20]，所以在大型的应用场景中，为了避免造成混乱的数据流和后期的难以维护，Flux架构是一个比较好的选择。



图 3.1.3 复杂MVC模型（来源：https://jaysoo.ca/2015/02/06/what-the-flux）



图 3.1.4 复杂Flux模型（来源：https://jaysoo.ca/2015/02/06/what-the-flux）

**3.1.2 Redux架构的应用**

Redux可以看作是Flux的一种实现，本项目中使用redux来进行数据的管理。Redux将各个Store整合成一个完整的Store，并且可以根据这个store推导出应用完整的state。Redux与Flux稍微有点不同的是它由Action、Reducer和Store三部分组成。

Action和Flux中的Action一样，主要用于传递操作State的信息，以JavaScript Plain Object的形式存在，如 {type: ’ADD\_ITEM’ , name: ’ITEM\_NAME’}。其中必须包含的是type，因为需要它来定义该action的类型。除此之外，action对象的结构完全可以自定义。

Reducer是根据action提供的信息做对应的操作，它是一个纯函数，通过传入旧的state和action来更新state，从而改变视图。如下为一个简单的reducer示例：

function filterReducer(state,action) {  
 switch (action.type) {

case ‘ADD\_ITEM:

return Object.assign({},state,{

visibilityFilter: action.filter

});

default:

return state;

}

}

Store在Redux中是唯一的，用于处理分发Action中的state数据并维护全局State。可以通过createStore()方法进行创建初始化的Store，其中有三个主要的方法:

·store.getState()用于获取当前state。

·store.subscribe(listener)相当于观察者模式中的监听，每当有state数据更新时，将会触发监听函数。

·store.dispatch(action)用于将用户触发事件对应的action分发给reducer进行处理，即用于处理用户操作。

Store对象使得我们可以通过store.dispatch(action)来减少对reducer的直接调用，并且能更好的对state进行统一管理。如果没有store，可能会造成reducer(currentState,action)这样频繁传入state参数的更新形式，对性能来讲是一个不小的损耗。

**3.2 React组件化开发**

**3.2.1 组件化开发的优势**

组件化在软件工程领域不是一个新概念，近年来随着前端工程化的迅猛发展，组件化的开发思想不多的得到完善和提升，相应的涌现了许多诸如React，Vue，Angular，Ember等组件化框架。它们有着一些几个优势：

（1）可扩展性。因为组件本身不会包含具体的业务逻辑，仅仅是针对某项具体功能而实现的，具有较高的抽象性。通过组件之间的组合搭配，可以构建出满足业务需求的新组件。

（2）可复用性。当一个应用需要大量使用到相同的一个功能时，可将该功能抽象成一个组件，在使用时只需传入必要参数和配置即可，这样避免了多次写相同的代码，极大的简化了程序的复杂性。

（3）高内聚低耦合性。在我们使用某一个组件时，无需关心该组件内部具体实现细节，我们只需关注它暴露出来的接口，对于使用者来讲完全就是一个开箱即用的“黑盒”。这对前端工程化及降低代码的维护来说是有很大好处的。

**3.2.2 容器和展示组件**

展示组件通常是一种只关注UI层面的组件。作为专门展示页面内容而不带任何业务逻辑的展示组件，一般情况下是没有状态的，只通过props来接受来自父组件的数据进行渲染和回调函数，本身不进行数据处理。同时，展示组件不使用任何Redux的API，如Redux的Action或Store。

容器是内部包含展示组件，作为关注应用如何运作并提供数据的一类组件。通常是与Redux层进行连接，将获得并处理的数据通过props的形式传递给展示组件。本项目中使用的容器组件通过react-redux的connect方法创建，这种方式能运行我们将Store中的数据作为props绑定到组件上，这是通过Redux管理数据的第一步。

**3.2.3 Ant Design组件库的使用**

目前开源社区有很多优秀的前端组件库，如Ant Design、Element UI、iView等。这些组件库更是将组件化开发形成了一整套完整的解决方案，使得开发一个前端页面变得非常的高效和便捷。由于Antd Design提供了完整的基础组件，其开发文档非常完善，开发社区活跃，所以本项目中使用了蚂蚁金服的Ant Design作为UI组件库。

Antd Design在加载组件的时采用的按需加载模式，这种加载模式避免了页面因加载过多无用的组件而造成页面阻塞，需要用到哪种具体的组件直接使用import方法载入该组件即可。同时Antd Design还提供了方便的样式配置，可以根据自己项目的场景对原有组件的样式进行覆盖，以实现更加符合业务的页面。

本项目的基本布局中使用到的sider、layout组件，充当承载页面内容的card组件，用于响应式的Grid组件以及button、input、select等基础组件，避免重复造轮子，极大的提高了系统的开发效率。

**3.2.4 React中的数据通信**

基于React的组件化开发使得业务模块得以更好的分离，做到高内聚低耦合。然而各个模块或组件间不可能各自独立的工作，它们之间往往会有数据交互，通过组件或模块间的数据通信使得应用才能正常运作。本课题针对前端性能监测系统的特点，研究并应用了四种组件间通信的方式。

（1）HTML5的postMessage方法

通常，对于两个不同页面的脚本，只有当执行它们的页面位于具有相同的协议（通常为https），端口号，以及主机时，这两个脚本才能进行通信[21]，若不满足上述三者中任何一个，浏览器的同源策略便会限制脚本间的信息传递。postMessage是HTML5中的一个方法，该方法可以安全地实现跨域通信[22]。

postMessage方法的语法如下：

otherWindow.postMessage( message, targetOrigin, [transfer] );

message参数是将要发送到其他窗口的数据，它在发送时会被结构化克隆算法序列化[23]。 targetOrigin参数是指定的目标地址。transfer是一个可选参数，表示源窗口将数据所有权转移到目标窗口，而发送方不再具有数据所有权。

（2）react中的state和porps

React推崇的是单向数据流，数据流向是从父组件传递到子组件，而不会逆向传递。由于组件化开发的特性，使用react开发应用时通常会将业务组件分成容器组件和展示组件，容器组件负责连接接口并处理数据，数据通常保存在组件的内部state中；展示组件通过props接收来自父组件的数据并渲染出来，不用考虑数据层面。如下为一个父子组件数据传递例子，父组件将内部state中的数据data传递给子组件Child，Child组件通过props的形式取得父组件的数据，并在div标签中渲染出来：

//父组件

class Parent extends React.component {

state = {

data: ‘我是数据’

}  
 render() {

return < Child childData = {this.state.data}/>  
}

}

//子组件

class Child extends React.component {

render() {

return <div>{this.props.childData}</div>

}

}

（3）react中的context

context的使用是为了解决props传递数据带来的缺点的。当有一个层级很深的组件，最内层的组件的行为将影响最外层的表现。如果是使用props的方式传递数据的话，做法是在最外层绑定回调函数，再一级一级地传入至最内层组件上，通过触发回调来进行上述行为。如果这个层级关系非常深的话很容易造成代码不简洁的情况，而且在后期代码维护上会加大维护难度。context的设计目的就是用于直接跨层级通信，以及非父子关系的组件间的通信，大大简化了嵌套的复杂度。使用时只要在外层组件上使用getChildContext方法声明要传递给子组件的context：

Class Parent extends React.component {

getChildContext() {

return {color: ‘red’ };

}

}

然后就可以在任意一级子组件上访问context里面的内容：

Class RandomChild extends React.Component {

render() {

return(

<p>{this.context.color}</p>

)

}

}

然而，过多的使用context也会带来问题，当过多的跨级传递数据时context的数据流同样也会非常杂乱，这样往往不容易跟踪数据的流向，给后期维护带来不便。所以本项目中在非必要的情况下尽量少用context作为数据传递渠道。

（4）观察者模式

观察者模式又被称作发布-订阅者模式或消息机制，定义了一种依赖关系，解决了主体对象与观察者之间功能的耦合[24]。本项目使用的Redux采用了观察者模式，Redux内部实现的观察者模式如下图：

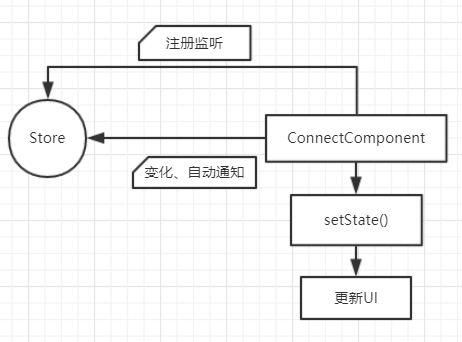


图 3.2.1 redux观察者模式实现

具体工作流程如下：

* store提供了subscribe(listener)方法用于注册监听。store内部用于存储所有通过store.subscrib注册的listener，store里state发生变化（即调用store.dispatch()）后，以此执行数组里的listener；最后store.subscribe返回unsubscribe方法，用于注销当前的listener.
* connect组件向store中注册监听方法。在构造方法中，先初始化订阅逻辑，将listener传递给Subscription.js；然后在componentDidMount生命周期中调用store.subscribe(listener)注册监听方法；最后在componentWillUnmount生命周期中注销订阅。

**3.3 功能设计与实现**

**3.3.1 整体设计**

前端性能监测系统按照功能的不同分为登录界面、主界面、性能分析界面三个部分。其中登录界面是系统的主入口，通过输入相应的用户名密码进入主界面，由于此系统不涉及数据库的增删改查功能，所以其中用户名和密码存储于内存中。主界面是一个用户操作界面，其功能是输入测评网站及相关的查询参数，如语言选择、设备类型。性能分析页面是显示具体网站性能监测指标的页面，其中包含页面加载分布、页面资源加载情况、优化建议及webpageTest分析页面四个模块。各界面之间关系图如下所示：

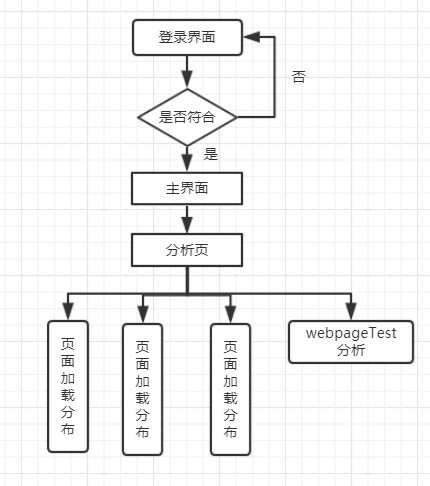


图 3.3.1 界面关系图

**3.3.2 路由设计**

使用React开发本系统是个单页应用，然而要实现各个模块和页面的跳转展示，同时又不希望在URL更改时重新刷新页面而造成很差的用户体验，所以需要用到路由器，本项目使用的是react-router作为项目路由。路由的本质是用于实现URL与UI界面的同步，当URL改变时相应渲染另一个页面。在react-router中，URL为浏览器的location对象所决定，而UI是由components渲染，这样就变成了location与react components之间的同步问题。可通过下图来描述：

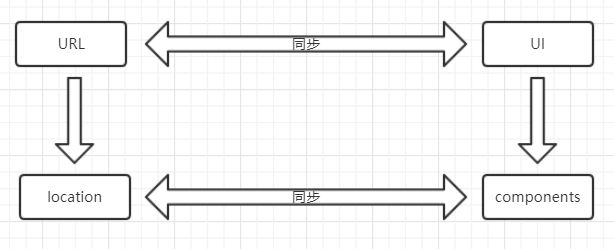


图 3.3.2 react-router基本原理

设计路由时，通常以一个主入口（如下图的index）作为根路径，进入各个主模块。如果用户输入不合法的URL时，通常可以使用<Redirect>组件进行重定向到统一的一个页面。如下表格为前端性能监控系统的路由配置：



图 3.3.3 路由地址与组件

**3.3.3 主页设计**

主页提供了用户输入测评网址和参数，并通过这些输入条件得到网站的具体性能分析结果。最顶端的选择Tab用于选择测评网址是在移动端还是在PC端的情况下进行测评；往下分别是测评网址输入栏和语言选择栏；再往下是网站测评概要信息，包括速度得分，性能优化得分和网站描述；最后点击详情按钮可以跳到分析页查看详细的性能分析结果。主页结构图如下：



图3.3.3 主页结构图

**3.3.4 分析页设计**

分析页布局采用的是常见的sider侧边栏、header标题栏、container容器三部分的布局方式。侧边栏为固定的导航栏，用于导航到不同的性能分析项目，点击不同的项目时，中间的container容器会渲染不同的内容；header标题栏为一个显示静态内容的容器，主要用于展示系统的概要信息和作为一个容器边界的作用；中间的容器承载的是可视化展示的性能数据，容器内容采用CSS3的flex弹性布局，可以自适应屏幕大小。整体的分析页结构如下图所示：

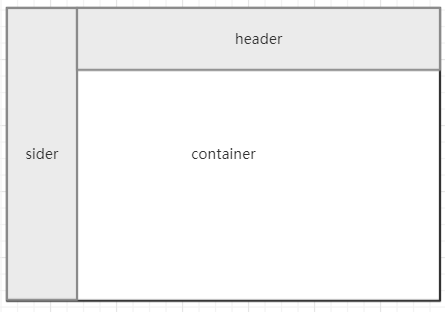


图 3.3.4 分析页结构图

其中分析页包括以下四个分析模块：

（1）页面加载分布。该模块统计了页面DOMContentLoaded（DOM内容加载，以下简称DCL）和First contentful paint（首次内容渲染，以下简称FCP）的加载分布情况，分别统计量快速、中速和慢速三个部分资源加载情况的百分比、分布情况，并以可视化的形式展现出来，如下图3.3.5所示。

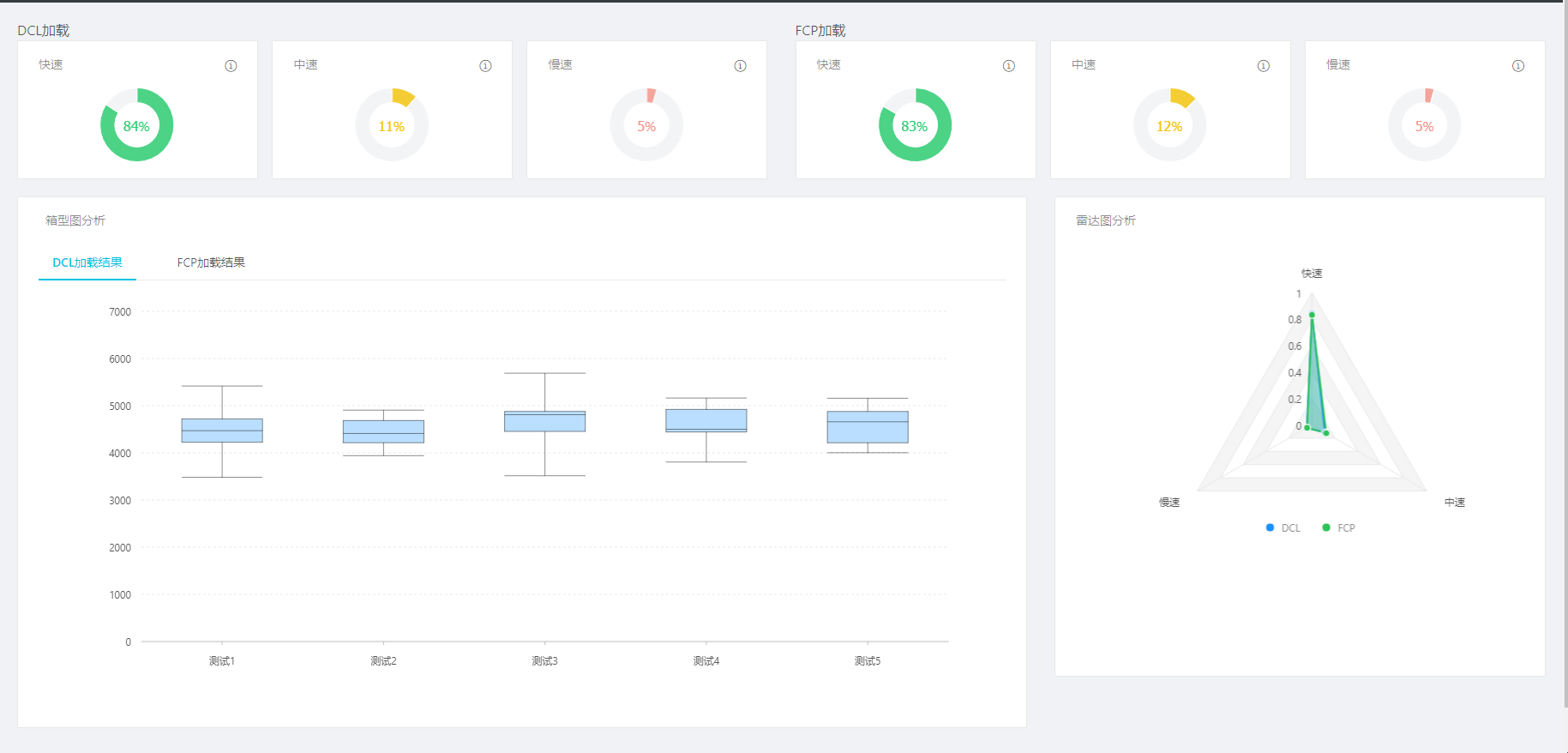


图3.3.5 页面加载分布

DCL是衡量网页速度的一个重要指标，当一个HTML文档被加载和解析完成，DCL事件变会被触发，而不需要等到像图片和脚本等会阻塞文档解析的资源加载再触发。在很多情况下，我们浏览网页时并不需要等待网页上所有的资源都加载完成，而只需把主体内容加载出来就可以开始正常使用。比如我们阅读一篇带有很多配图的博客，可能并不需要等所有图片都加载出来，而是看到博客正文便可正常阅读。所以，衡量一个网页加载速度的一个方法就是计算这个网页的首屏绘制时间，这就是使用DCL衡量网页速度的很重要一个原因。

FCP指标报告了浏览器首次呈现文本、图片、canvas画布或SVG的时间，和FP（First Paint首次绘制）不同，FCP能表示应用以及进入运行状态而不是处于卡机或者加载失败状态，对于用户体验具有更好的代表性[25]。

（2）页面资源加载耗时。该模块分为页面加载耗时指标和资源统计两部分内容。页面加载耗时指标通过window.performace.timing接口提供的数据，经过简单的计算得出一系列的网页性能数据，计算方式如下：

var t = performance.timing

t是performance.timing对象，通过获取对象里面的指标参数，这些指标的值是一个时间戳不具有使用意义，然而将其进行简单的减法运算便可得出各个性能的时间耗时，具体计算结果如下表：

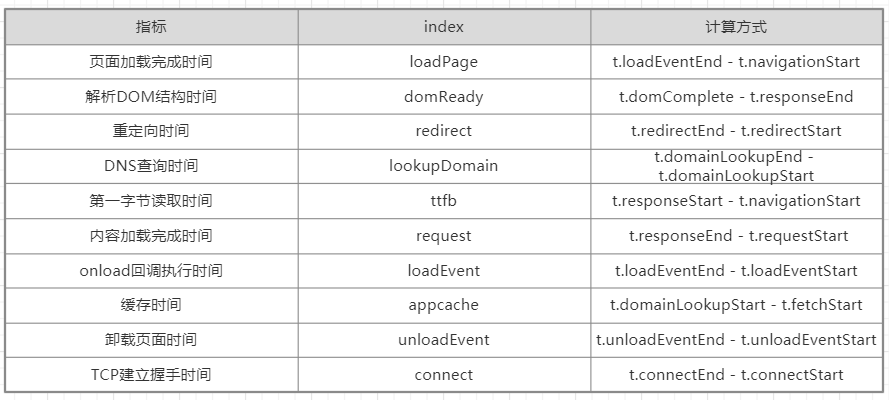


表 3.3.6 页面加载耗时指标

资源统计部分的数据是pageSpeed API返回的数据，包括请求的资源字节数、请求资源的个数和请求资源其他统计量三部分，具体字段如下表：



表 3.3.7 统计资源情况

可视化后的结果如下图所示：



图 3.3.8 资源加载耗时页

（3）优化建议。该模块是pageSpeed API返回针对所测评的网站而提出的性能优化建议数据，主要分成启用压缩、浏览器缓存、资源优化和首屏阻塞四个部分的优化建议。启用压缩、资源优化和首屏阻塞部分的建议会详细列出可以做相应改进的文件，并且预测使用该项改进措施后会减少多少空间，从性能上将会提升的百分比；浏览器缓存详细列举出可缓存的资源选项。这样用户便可以有针对性的根据性能优化建议对网站进行相应的调整。如下图为优化建议模块页面展示：

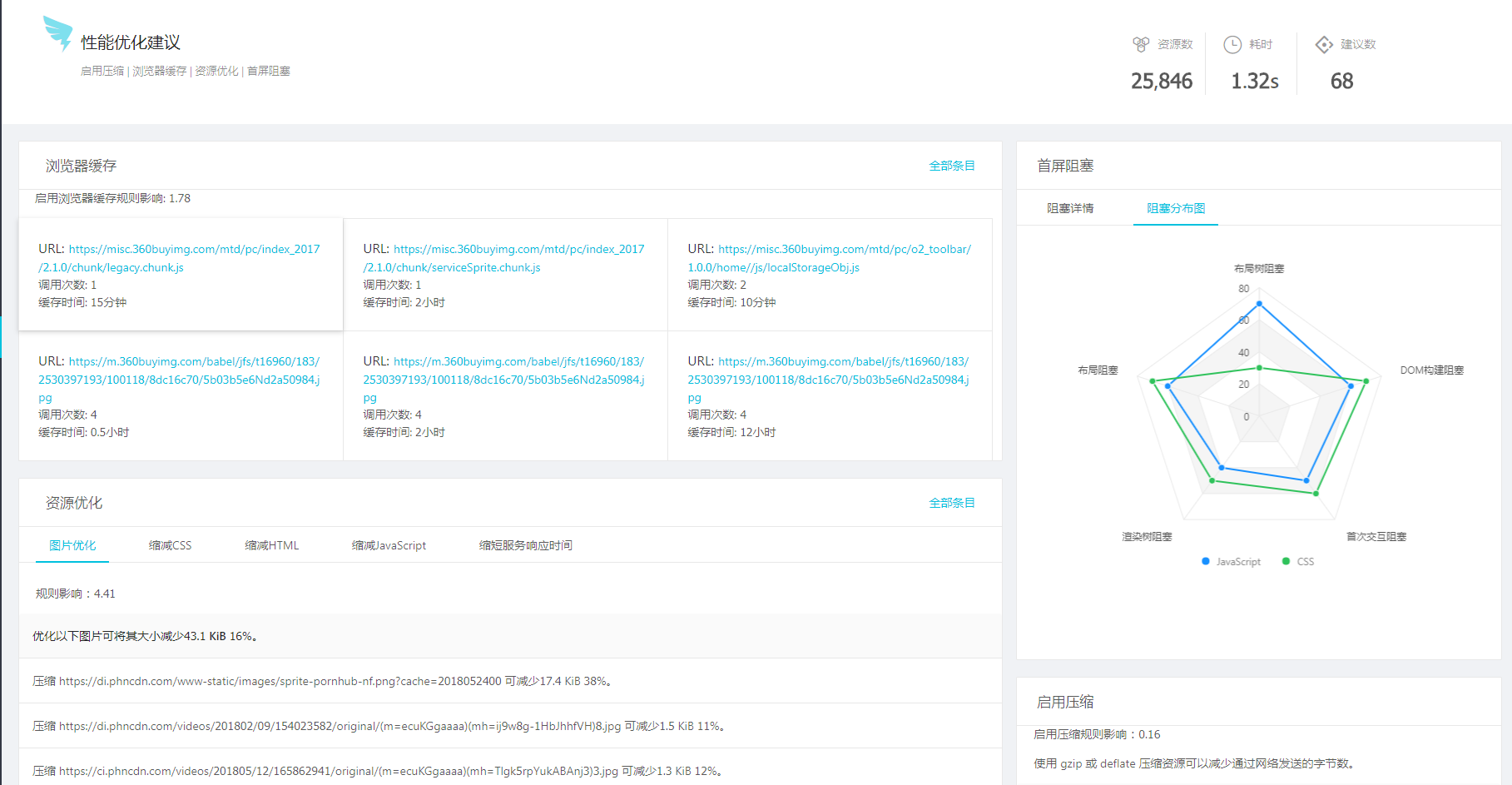


图 3.3.9 优化建议页面

（4）WebpageTest分析。该工具从全球不同地点访问网站，提供了友好的测试结果报表，其中最优特色的是资源加载的瀑布流图，可以清楚看出每个加载阶段加载的资源以及网页的显示状态。本文开发的前端性能监测系统直接使用iframe标签嵌入webpageTest网页对目标网站的测评结果，使用其测评结果的原因是它提供了非常详尽的瀑布流图，可以非常清晰看出每个资源加载的顺序和加载耗时，可以及时的发现阻塞了进程的资源以便进行相应的处理。如下如为使用WebpageTest测试京东商城（https://www.jd.com）所得到的瀑布流图:

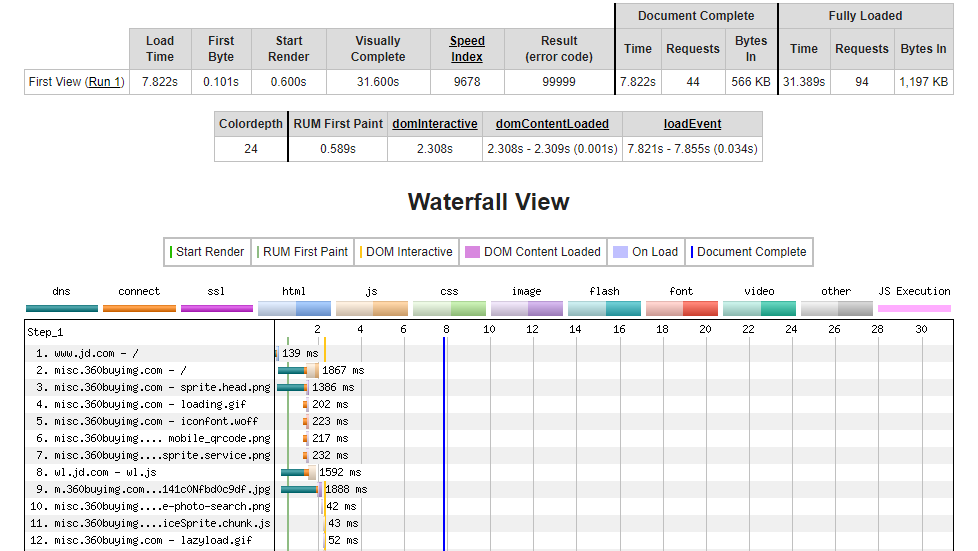


图3.3.9 webpageTest瀑布流图

**第四章 数据存储与可视化**

**4.1 本地存储技术**

当系统的使用量和访问量迅速增加，高并发会导致服务器响应速度变慢，从而降低网页的响应速度。然而通过适当的使用本地存储技术，可以有效的降低服务器的数据传输载荷，提高网页的响应速度。本章针对前端性能监测平台的应用特点，对所使用的两种本地存储技术进行了研究。

**4.1.1 Cookie存储**

在早期的时候，服务器很难知道不同的请求是否来自同一个浏览器，这就造成了状态管理上的困难，cookie的出现便是用于解决身份认证的问题。存储在cookie中的数据，每次都会在浏览器发送http请求时一同发送至服务器，这些数据通常是用于身份认证的数据，在每次向服务器请求时都会带上cookie数据用于服务器身份校验。

一般来讲，cookie就是存储在电脑上的一个纯文本文件，不包含任何代码。创建cookie的方法是通过发送一个http请求，请求中带有set-cookie消息头，其格式如下所示：

set-cookie:value[;expires=date][;domain=domain][;path=path][;secure]

设置的值中每一部分分别表示cookie的值、过期时间设置、作用域设置、请求资源指定路径和安全性设置。另外，也可以使用JavaScript代码来对cookie进行创建、维护和删除，主要是通过document.cookie进行操作，比如要使用和set-cookie相同格式的cookie：

document.cookie = “name=userLogin;domain=www.baidu.com;path=/”

要通过JavaScript读取cookie的值，同样直接使用document.cookie即可进行读取。

**4.1.2 localStorage存储**

localStorage是HTML5提出的Web Storage的一种，通过键值对的数据结构来保存数据，它是一种可靠度长期型本地存储，除非主动删除，否则数据将永久保留。localStorage可容纳的数据量比cookie大，通常为5M，与cookie不同的是，它仅在客户端中保存，不参与服务器的通信。这种情况下，它的数据读取速度是非常快的，非常适合用于存储页面首屏渲染时所需显示的数据。使用localStorage通常是通过setItem(key,value)和getItem(key)两种方法进行操作，setItem方法传入键名和键值来存储需要的数据，然后通过数据对应键名调用getItem方法即可获取数据。这两个方法在全局作用域有效，不需要担心数据跨域问题。

**4.2 数据可视化研究**

**4.2.1 均值的局限性**

通常而言，性能数据往往是一个波动的数据值，均值是数据处理中最常见的一种方式。因为它能直观的表示指标的趋势与分布状况，方便进行评估、瓶颈发现与告警。以下表格显示的是浏览器页面加载耗时的测试数据：

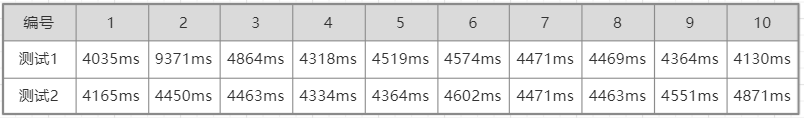


表 4.2.1 加载耗时数据

那么根据以上结果，可以得到：测试1的均值为4911.5ms，测试2的均值为4473.4ms。但是如果就这样粗略的就使用该均值作为评估结果的话，那就显得比较武断，因为进一步探究即可以发现：

* 测试结果没有剔除异常数据。测试1中编号2的数据明显是异常数据，应该剔除。
* 对测试数据描述不够完善。光看均值很难了解数据的波动情况
* 对部分数据的描述感觉不准确。若数据的分布相差较大，则对部分数据而言使用均值可能没有较大的意义。

**4.2.2 箱型图分析**

本文在研究数据可视化之初便是使用均值作为处理数据的方法，然而在后来的研究中发现均值具有其自身的局限性，并不能全面的展示性能数据。所以在本章研究中使用的是已有统计工具中的箱型图进行性能数据分析。首先要了解的是四分位数的概念：

四分位数（Quartile）在统计学中将所有数值由小到大排列并分成四等分，处于三个分割点位置的数值就是四分位数，分别是：

* 第一四分位数（Q1），为数值顺序排序后第25%的数字。
* 第二四分位数（Q2），为数值顺序排序后第50%的数字。
* 第三四分位数（Q3），为数值顺序排序后第75%的数字

其中，Q1和Q2的差距也被称为四分位距（InterQuartile Range,IQR）。其中异常值为在Q1-1.5IQR或Q3+1.5IQR范围之外的值，即箱型图上下边界之外的值。

箱型图（Box-plot）就是基于四分位数设计的，它能显示数据集的上边缘、下边缘、上四分位数、下四分位数及中位数，是一种用作显示一组数据分散情况资料的统计图[26]。

返回到上一节中讨论的加载耗时数据，可以将其整理成箱型图参数的形式：

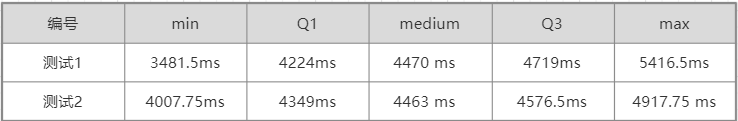


表4.2.2 箱型图参数结果

根据以上数据可以绘制出箱型图如下所示：

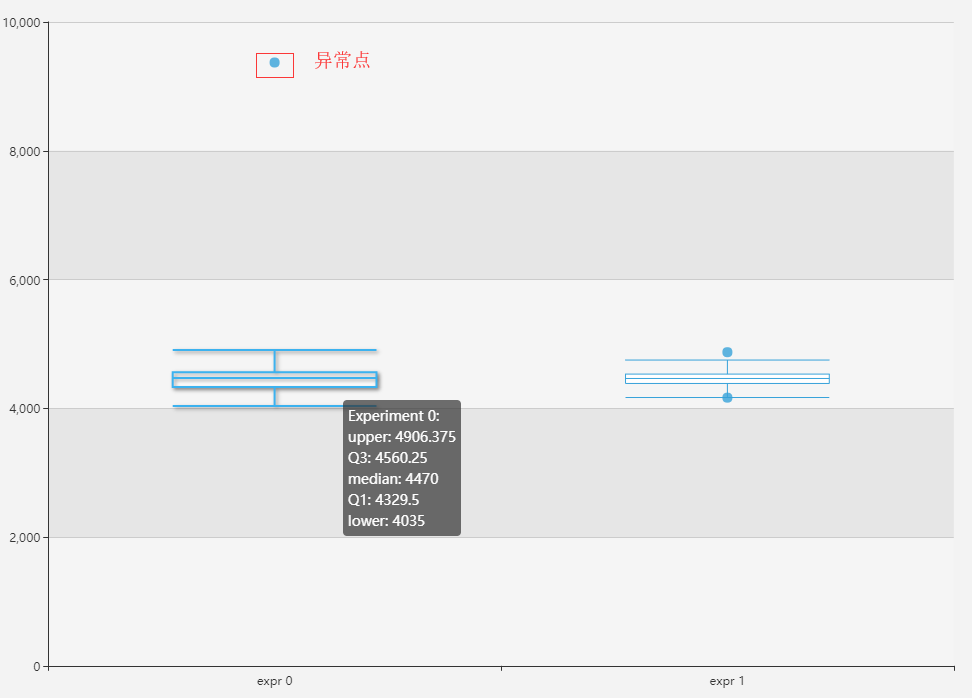


图 4.2.3 箱型图结果

通过对比均值和箱型图的结果，可以得到不同的结论：

* 均值：测试1的速度（4911.5ms）较慢与测试2的速度（4463ms）。
* 箱型图：

（1）测试1与测试2的速度基本相同（中位数分别为4470ms和4463ms）；

（2）测试1的速度变化大于测试2（四分位数分别为495ms和227.5ms）；

（3）测试1中有一个很明显需要剔除的异常数据（9371ms）；

使用箱型图有三个优势：可以识别数据异常值、快速对比不同数据集基本特征、对不严格服从正太分布的数据集耐抗性强[26]。箱型图的基本结构如下图：

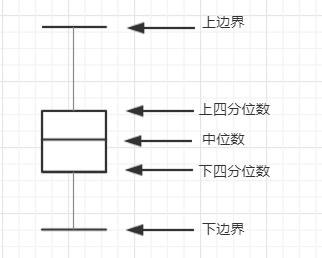


图 4.2.4 箱型图结构图

本文使用箱型图用于分析页面加载分布中的DCL(DOM Content Loaded)和FCP(First Contentful Paint)加载分布和页面加载耗时中的耗时数据。由于性能数据不是一个具体固定的值，所有本研究中采用多组多次取值的方法，具体工作内容如下：

* 在页面加载之初，即react生命周期中的componentWillMount阶段，使用phantomjs工具对服务器发送多次请求，将每次获取的数据标记并保存。
* 利用获取的计算出每组数据的下边界、中位数、上四分位数、下四分位数和下边界，并绘制相应的箱型图。
* 取最稳定那组的数据的平均数作为最终显示的数据指标值。

**4.2.2 数据可视化工具——Bizchart**

Bizchart是由阿里巴巴研发的基于G2 React的开源图表库。G2是一套提供强大语义化图表的可视化解决方案，用户可以通过简单的语法组合和配置即可将数据表达的栩栩如生，大大的提高了数据的解释能力，可让我们迅速发现其内在信息。而Bizchart是针对React而进行版本封装出的一套工具，其底层实现和G2相同，只是它的开发模式更加符合React技术栈的习惯，能让开发者使用React的开发思维使用该工具。如下图为一个图表示例：

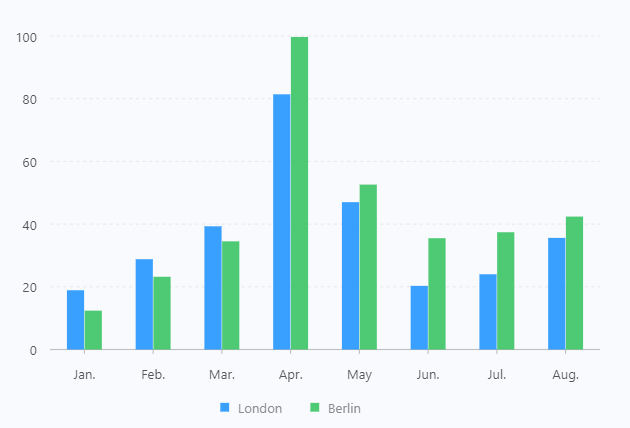


图 4.2.5 Bizchart柱状图示例

创建图表时可以像使用React组件一样进行搭建：

<Chart height={400} data={dv} forceFit>

<Axis name="月份" />

<Axis name="月均降雨量" />

<Legend />

<Tooltip crosshairs={{type : "y"}}/>

<Geom

type='interval'

position="月份\*月均降雨量"

color={'name'}

adjust={[{type: 'dodge',marginRatio: 1/32}]}

/>

</Chart>

**第五章 总结与展望**

**4.1 总结**

本文首先通过分析现代前端性能优化对企业利益的重要性，良好的用户体验和流畅的网页体验对世界各大公司的利益有着巨大的影响。接着探讨了国内外对前端性能的研究，包括前端性能分析工具和性能优化方案。通过研究了国内外已有的性能分析工具，提出并实现了一套基于现有开源工具开发的前端性能监测平台，具有精准，简洁、高效和数据表达直观的特点。本文就此前端性能监测平台对以下内容进行了研究：

（1）对浏览器工作原理进行剖析。从最基本的渲染引擎入手，了解到其如何解析并构建DOM树结构，并通过DOM树进行渲染、布局和绘制，最终呈现出我们在浏览器中所见的内容。浏览器的工作流程的每一阶段也就是可进行性能优化切入点，这里便引出了性能优化原理及其对应的解决方案。最后，基于开源的性能数据接口，探究了浏览器的performance.timing捕获网页性能指标和Google PageSpeed API提供性能数据两种采集数据方法。

（2）平台采用有别于传统MVC模式的Flux框架，分析了Flux架构的优势，并引出react-redux，一个能与react完美配合开发的一种Flux架构。

（3）探讨了react的组件化开发思想及优势，阐述如何对项目中的组件进行拆分成容器组件和展示组件，接着使用说明如何使用Ant Design作为本平台一个UI解决方案，然后详细讨论了平台中的组件间数据通信问题，最后详细的说明了整个平台的搭建。

（4）数据分析与可视化问题。从API中获取的原始数据为JSON格式对象形式，要从深层嵌套的数据格式中提取需要展示的内容，本文对原始数据进行加工并使用BizChart进行可视化处理，以更直观的形式展现数据。

**4.2 展望**

本文的不足之处有以下两点：

1. 前端性能监测系统本身的不足。Lighthouse虽然实现了对目标网站的性能进行监测并生成性能报告和优化建议，但是目前不支持监测本地和企业内网运行的应用，因为在使用PageSpeed API接口时是通过远程服务器对网站进行访问后才进行性能分析的，而内网和本地应用远程服务器是访问不到的。同时，系统本身对异常的处理还不够完善，目前的处理措施只是对遇到过的异常使用hack方式来解决，并没有一个完善的解决方案。

2. 数据分析方面的不足。本文研究的性能数据大多为数值型数据，使用箱型图分析也有其自身的局限性，比如在程序上实现时需要多次请求数据，会造成一定性能负担；同时可视化分析只是对单一的数据进行可视化展示，缺少多种类数据组合起来进行多维分析，未来的研究中可以将具有相似特征的数据组合起来进行多维可视化分析，以便能更好的发现性能优化的切入点。

**致谢**

时光如流，四年的本科生涯也即将告一段落。在中国地质大学读书的四年里，我得到的不光是专业知识和技能上的提升，更多的是对这个世界和自身有更深的认识，为此我对在求学路上帮助过我的亲人、同学、老师表达衷心的谢意。

感谢我的导师刘福江老师，作为我的班主任兼导师，在我的生活和学习过程中给予了我很大的帮助。我本身是学遥感的，但是对前端开发有着较浓厚的兴趣，刘副江老师鼓励我在毕业设计的选题上针对自己的兴趣来进行研究。感谢老师在研究过程中对我的悉心指导，并提出宝贵有针对性的意见。

感谢京东给了我实习和工作的机会，在实习过程中，我的开发动手能力有了很大的提升，对前端技术的掌握也有了较深的了解。在企业做项目时意识到前端性能对企业网站的重要性，这也萌发了本文关于前端性能优化的研究内容。感谢一起工作的同事们，在我遇到技术问题时耐心的和我讨论并给予我指导，能够使我更好的进行毕设的研究。

感谢大学四年来帮助过我的同学、老师、家人和朋友，是他们每个人对我的鼓励和支持，让我更加的对未来有信心和目标。同时感谢自己在大学四年中一直坚持对自我的探索，在失落迷茫时不放弃，能从困境中尽快走出，朝着自己既定的目标努力，在最后毕业之际给自己一个较为满意的结果。

最后感谢审阅人周琪老师和各位评委老师，感谢你们的宝贵时间和宝贵意见。

**参考文献**

[1] Impact of Web Latency on Conversion Rates [EB/OL],https://www.slideshare.net/bitcurrent/ impact-of-web-latency-on-conversion-rates

[2] Onlinegraduateprograms. Instant America Network search [EB/OL], http://www.onlinegraduateprograms.com/instant-america

[3] KissMetrics. How Loading Time Affects Your Bottom Line [EB /OL], http://blog.Kissmetrics.com/loading-time/?wide=1

[4] Best Practices for Speeding Up Your Web Site [EB/OL], http://developer.yahoo.com/performance/rules.html

[5] 操秀英，唐婷. 中国互联网为何“跑”不出世界的网速[N].科技日报 2010,10(3):1.

[6] 王成，李少元. web前端性能实现方案与实践[J]. 计算机应用与软件 2014 （12）95-101

[7] Steve Sounder, et al. Even Faster Web Sites: Performance Best Practices for Web Developers[M]. O’Reily Media, 2009: 1-250.

[8] 常见的浏览器内核 [EB/OL]. https://www.jianshu.com/p/6efcccb5ed43

[9] How Browsers Work:Behind the scenes of modern web browsers [EB/OL], https://www.html5rocks.com/en/tutorials/internals/howbrowserswork

[10] 渲染树构建、布局及绘制[EB/OL], https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction

[11] Appendix E.Elaborate description of Stacking Contexts [EB/OL], https://www.w3.org/TR/CSS21/zindex.html

[12] 宋刚，蒋梦奇，张云泉，等. 基于共享存储和Gzip的并行压缩算法研究[J]. 计算机工程与设计,2009(4):781-784

[13] 曹海歌. 基于改进的Diff算法的Web前端性能优化及应用 [D]. 武汉：华中师范大学，2016：16

[14] 莫里森，深入浅出JavaScript[M],第3版,中国电力出版社,2007年10月。

[15] Nicholas C.Zakas，JavaScript高级程序设计[M],第三版，人民邮电出版社，2017年5月。

[16] HTTP缓存 [EB/OL], https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=zh-cn

[17] Chrome User Experience Report [EB/OL], https://developers.google.com/web/tools/chrome-user-experience-report

[18] PageSpeed Insights [EB/OL], https://developers.google.com/speed/docs/insights/about

[18] Flux vs. MVC (Design Pattern) [EB/OL], https://medium.com/hacking-and-gonzo/flux-vs-mvc-design-patterns-57b28c0f71b7#.2uqupqv13

[19] What is Flux [EB.OL], http://fluxxor.com/what-is-flux.html

[20] What the Flux? (On Flux,DDD,and CQRS) [EB/OL], https://jaysoo.ca/2015/02/06/what-the-flux/

[21] MDN web docs,window.postMessage [EB/OL], https://developer.mozilla.org/zhCN/docs/Web/API/Window/postMessage

[22] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In NDSS, 2013.

[23] The structured clone algorithm [EB/OL], https://developer.mozilla.org/enUS/docs/Web/API/Web\_Workers\_API/Structured\_clone\_algorithm

[24] 张容铭，JavaScript设计模式[M]，第一版，人民邮电出版社，2015年8月。

[25] First Contentful Paint Explained [EB/OL], https://gtmetrix.com/blog/first-contentful-paint-explained

[26] D.F. Williamson, R.A. Parker, J.S. Kendrick, The box plot: a simple visual method to interpret data, Ann. Intern. Med. 110(11) (1989)