**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。

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，并如何应用与可视化表达中。

在Lighthouse系统的前端架构方面，研究并分析了基于react+redux的开发模式、pageSpeed API接口、浏览器API、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

**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，性能在一定程度上与利益直接相关。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[2]，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。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]。

由于网站的性能对企业的利益起着至关重要的作用，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[5]。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。

**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Web前端的性能优化非常的重视，谷歌的Chrome浏览器的Chrome DevTools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。在多年以前便提出来Web性能优化的黄金法则：先优化前端程序的性能，因为这是80%或以上的最终用户响应时间的花费所在。同时雅虎还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》[6]以及其他另外8位Web前端专家一起编写的《Even Faster Web Sites》[7]提供了提升网站性能的最佳实践和使用建议。

在网页性能分析工具方面，比较出名的有以下几个工具：PageSpeed和SpeedTracer，主要对页面进行评分，然后会显示各项指标的修改意见；Yslow，和PageSpeed类似，对各种影响网站性能的因素进行评分；WebpageTest，一个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HTTP协议、浏览器工作方式、浏览器缓存机制、页面结构及Ajax请求等方案。

**1.3 研究内容**

本文首先研究了浏览器的工作原理，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，然后结合react和react路由，提出了应用与本系统的react+redux+react-router开发模式。

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。最后介绍了项目中使用到的蚂蚁金服的一套UI库antd及其具体使用方法。

在Lighthouse功能设计与实现方面，研究了整体的设计，入口页设计，可视化分析页设计以及路由设计，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分析方面，研究了项目使用的本地存储技术，包括localStorage存储和sessionStorage；组件间数据传输方式，包括react特有的传递数据方式时HTML5下的传递方式；数据可视化分析。

**1.4 论文组织结构**

全文总共分为五个章节，每个章节的具体内容如下：

第一章，引言。分别概述了本课题的研究背景，研究现状，具体的研究内容以及研究的结构和安排。

第二章，相关理论和技术分析。分别探究了浏览器工作原理，Web性能优化原理，性能优化方案以及性能指标数据四个方面，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，探究react+redux的开发模式，然后设计并实现前端性能监测系统。

第四章，数据的存储与可视化研究。分析项目中主要应用到的本地存储技术及其类型，探究组件间如何通信的问题，探究对数据可视化分析。

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 相关理论和技术分析**

本章首先研究了浏览器的工作原理，然后基于浏览器的渲染原理，对Web性能优化原理进行探究，进而有针对性的研究了性能优化方案。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**

浏览器的主要功能是向服务器发出请求，在浏览器窗口中呈现相应的HTML文档、PDF、图片或其他类型的网络资源。资源的位置由用户使用URL（统一资源标示符）指定。

浏览器的主要组件为：

1. 用户界面：主要包括地址栏、前进/后退功能按钮、书签栏等。除了浏览器主窗口显示的请求资源外，其他显示的各个部分都属于用户界面。

2. 浏览器引擎：用于在用户界面和呈现引擎之间传送指令。

3. 渲染引擎：根据用户请求的内容进行解析相应的资源，并通过一定的规则显示在屏幕上。常见的渲染引擎[8]有Firefox的Gecko引擎，IE使用的Trident引擎和Chrome和Safari使用的Webkit引擎[8]。

4. 网络：用于网络调用，比如HTTP请求。其接口与平台无关，并为所有平台提供底层实现。

5. 用户界面后端：用户绘制基本的窗口部件，比如组合框和窗口。其公开了与平台无关的通用接口，而在底层使用操作系统的用户界面方法。

6. JavaScript引擎：用于解析和执行JavaScript代码。

7. 数据存储：这部分是持久层。浏览器需要在硬盘上保持各种数据，例如常见的Cookie和日志。新的HTML规范（HTML5）定义了“网络数据库”，这是一个完整的浏览器内数据库。

浏览器结构如图[9]：



图2.1 浏览器构造图

**2.1.1 渲染引擎**

渲染引擎主要功能是在浏览器中呈现请求的内容。默认的情况下，渲染引擎可显示HTML和XML文档与图片。通过插件（或浏览器扩展程序），还可以显示其他类型的内容，例如，使用PDF查看器插件就能显示PDF文档。

渲染引擎一开始会从网络层获取请求文档的内容，内容的大小一般限制在8000个块以内。如下为渲染引擎的基本工作流程：

解析HTML用于构建DOM树

渲染DOM树

DOM树布局

绘制DOM树结构

图2.1.1 渲染引擎的基本流程

根据上图，渲染引擎将开始解析HTML文档，并将各标记逐渐转化成“内容树”上的DOM节点。同时也会解析外部样式文件以及样式文件中的样式数据。文档中带有视觉指令的样式信息将用于创建另一种树结构，呈现树。该树包含了多个带有视觉属性（如颜色和尺寸大小）的矩形，这些矩形的排列顺序就是将在屏幕上渲染的顺序。

在呈现树构建完毕后，开始进入布局处理阶段，这个阶段是为每个节点分计算其在屏幕上的确切坐标，然后在绘制阶段由用户界面后端层将每个节点绘制出来。

需要提及的是，渲染引擎工作的这个过程是一个渐进的过程。为了达到更好的用户体验，渲染引擎会力求尽快将内容渲染到屏幕上，它不必等到整个HTML文档解析完整后再进行布局和渲染，在不断接收和处理来自网络传输的其余内容的同时，渲染引擎先将部分内容优先解析并显示出来。

如下为两种渲染引擎的主流程：

HTML 解析器

DOM树

附加

渲染树

布局

绘制

展示

HTML

样式规则

CSS解析器

CSS

DOM

图2.1.2 Webkit引擎主流程

DOM

解析器

HTML

重排

框架树

绘制

展示

样式规则

CSS解析器

CSS

内容槽

框架构建器

内容模型

图2.1.3 Gecko引擎主流程

从上面两图可看出，Gecko引擎将视觉格式化元素组成的树成为“框架树”，即每个元素都是一个框架。Webkit引擎使用的术语是“呈现树”，由“呈现对象”组成。对于元素的定位，Webkit使用的术语是布局而Gecko称之为“重排”。二者还有一个差别就是Gecko在HTML和DOM树之间还有一个称之为“内容槽”的层，用于生成DOM元素。虽然二者在相关的术语上有些不同，但是整体流程是基本相同的。

**2.1.2 解析和DOM树构建**

解析文档是指将文档转化为可以让代码理解和使用的结构，解析所得到的结构通常是代表了文档结构的节点树，也称之为解析树或语法树。

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大量标记的过程，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。解析是一个迭代的过程，通常解析器会向词法分析其请求一个新标记，并尝试将该标记与某条语法规则进行匹配，若发现了匹配规则，解析器会将一个对应于该标记的节点添加到解析树中，然后继续请求下一个标记。如果是没有可用匹配的规则，解析器就会将标记存储到内部，并继续请求标记，直到找到可与所有内部存储的标记匹配的规则。如果找不到任何匹配规则，解析器就会触发一个异常，该异常意味着文档无效。

在创建解析器的同时，也会创建用于构建DOM树的Document对象。在树构建阶段，以Document为根节点的DOM数也会不断进行修改，向其中添加各种元素，标记生成器发送的每个节点都会由树构建器进行处理。W3C规范中定义了每个标记所对应的DOM元素，这些元素会在接收到相应的标记时创建，这样通过元素的一个个创建，便形成了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。

在处理html和body标记时就会构建渲染树根节点，这个根节点称之为最上层的区块（block），该区块包含了其他所有区块。它的尺寸就是视口，即浏览器窗口显示区域的尺寸。为构建渲染树，浏览器大致完成以下工作[10]：

1. 从DOM树的根节点开始遍历每个可见节点

2. 对于每个可见节点，为其找到适配的规则并应用它们。

3. 构建可见节点，连同其内容和计算的样式。

4. 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。

**2.1.4 布局与绘制**

布局的过程是输出一个“盒模型”，它精确地捕获每个元素在视口内的确切位置和尺寸，所有相对测量值都转换为屏幕上的绝对元素。HTML是基于流的布局模型，这意味着大多数情况下只要遍历一次就能计算出几何信息，处于流靠后位置的元素通常不会影响靠前位置元素的几何特征，因此布局可以按照从左至右、从上到下的顺序遍历文档。布局同样是一个递归的过程，从根呈现器（比如HTML文档中的<html>元素）开始，然后递归遍历部分或所有的框架层次结构，为每一个需要计算的呈现器计算几何信息。同时，为了避免由于细小的更改而进行整体计算布局，浏览器采用了一种“dirty位”系统进行标记更改的元素，减少不必要的布局计算以提高浏览器渲染速度。

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实际像素。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**

浏览器想要渲染页面必须首先构建出DOM树和CSSOM（叠层样式对象模型）树，如果HTML与CSS文件结构非常庞大，这显然会给页面加载速度带来严重影响。HTML、CSS、JavaScript都是会对页面渲染产生阻塞的资源，除去必须的HTML，可以对CSS和JavaScript着手进行优化，尽可能减少阻塞的影响。

**2.2.1 优化CSS**

（1）使用媒体查询

媒体查询由媒体类型以及零个或多个检查特定媒体特征状况的表达式组成，可以让CSS资源在特定条件下使用，这样这些资源就可以在首次加载时先不进行构建CSSOM树，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。

（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[12]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）。

（3）CSS放在head中，减少repaint和reflow

将CSS放在页面顶部有利于页面优化。当浏览器从上到下下载HTML并生产DOM树结构，然后根据浏览器默认及现有CSS生成渲染树来渲染页面，当遇到新的CSS时下载并合并现有CSS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）优化循环[15]

循环在代码中非常常见。优化循环的步骤如下：

1）减值迭代。在很多情况下，从最大值开始，循环中不断减值的迭代器更加高效。

2）简化终止条件。由于每次循环过程中都会计算终止条件，所有必须保证它尽可能快，也就是说避免属性查找或其他O(n)的操作。

3）简化循环体。需要确保循环体被最大限度优化，确保没有某些可以被很容易移出循环的密集计算。

4）使用后测试循环。最常用的for循环和while都是前测试循环，而如do-while这种属于后测试循环，可以避免最初终止条件的计算。

**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，浏览器还需要与服务器之间进行多次往返才能获得完整的资源。因此，利用缓存复用之前获取的资源成为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是一个传递验证令牌，它对资源的更新进行检查，如果资源未发生变化时不会传递任何数据。当浏览器发送一个请求，会把ETag一起发送到服务器，服务器根据当前资源核对令牌，如果资源发送变化，服务器将返回“304 Not Modified”响应，这是浏览器不必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，则即使它有关联的HTTP身份验证，设置响应状态码通常无法缓存的情况下都可以是浏览器进行缓存；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。如下图为定义最近Cache-Control策略流程图[16]：



图2.2.1 Cache-Control最佳策略

**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数据，提供浏览器处理网页各个阶段的耗时，我们可以通过这些时间数据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，如下图为获取百度的时间加载数据：



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。此部分会列出可应用于相应网页性能优化的最佳做法，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视图，用户数据显示，同时接受用户输入，

·Controller，相应用户的输入，对数据进行操作，

·Model，负责管理程序需要的数据，并且定义了操作数据的行为。

对于一个简单的MVC架构程序来讲，其工作流程[18]如下：



图 3.1.1 MVC工作流程

从最右边的View开始，当用户在UI上进行操作之后，用户的操作被转发到Controller上，Controller根据用户的操作对数据进行更新，数据更新之后View所展现的内容也会相应的更新。Model层此时可以向所有关联的视图发送通知，收到通知的视图重新获取更新的数据。

（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，在由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Store中的数据发生了改变，则触发数据变更事件，View监听这些事件并做出相应的反映。

一个简单的Flux流程[19]如下：



图3.1.2 Flux工作流程

从上图可以看出，Flux架构一共分为四个模块角色。

·View，可以理解为View与Controller的结合，它既展现数据，同时也处理用户的交互请求。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最终会映射成对应的Action，并交由Action进行下一步处理。

·Action，用于描述组件触发的操作，由键名和键值组成，比如：

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

·Dispatcher，属于Flux的中央枢纽，所有的Action都会交由Dispatcher进行处理。在接收到Action之后，调用Store注册在Action上的回调函数。与MVC中Controller不同的是，Dispatcher不包含业务逻辑，可以被程序复用也可以被其他Dispatcher替换。

·Store，包含程序的数据与业务逻辑。与MVC的Model比较，MVC中每一个model即对应中一个领域模型；而Flux中的一个Store可能包含多个模型，并且只有Store知道如何修改数据，它并不对外直接提供操作数据的接口，Action和Dispatcher都无法操作Store。

（3）MVC的局限

显示开发场景中通常都是多视图、多模型的，更严重的是视图和模型之间还可以是多对多的关系。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



图 3.1.4 复杂Flux模型

**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type属性是必要的，除了type字段外，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中是单一的，维护着全局的State，并且根据Action来进行事件分发处理State，是一个将Action和Reducer结合起来的对象。Redux提供了createStore()方法来生产Store，并提供了三个API:

·store.getState()用了获取state数据,

·store.subscribe(listener)用于注册监听函数，每当state数据更新时，将会触发监听函数,

·store.dispatch(action)用于将一个action对象发送给reducer进行处理。

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**

**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] Steve Sounder. High Performance Web Sites[M]. 2007:1-170．

[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/>