极客时间

# 前端全链路性能优化实战

极客时间

扫码试看/订阅

《前端全链路性能优化实战》视频课程

# 第三章：页面渲染架构设计和性能优化

极客时间

- 3.1 浏览器渲染过程
- 3.2 页面渲染技术架构和方案总览
- 3.3 后端同步渲染
- 3.4 静态化技术方案和实现思路
- 3.5 前后端分离技术与实现
- 3.6 单页面应用技术方案
- 3.7 BigPipe 简介和工作模式

# 第三章：页面渲染架构设计和性能优化

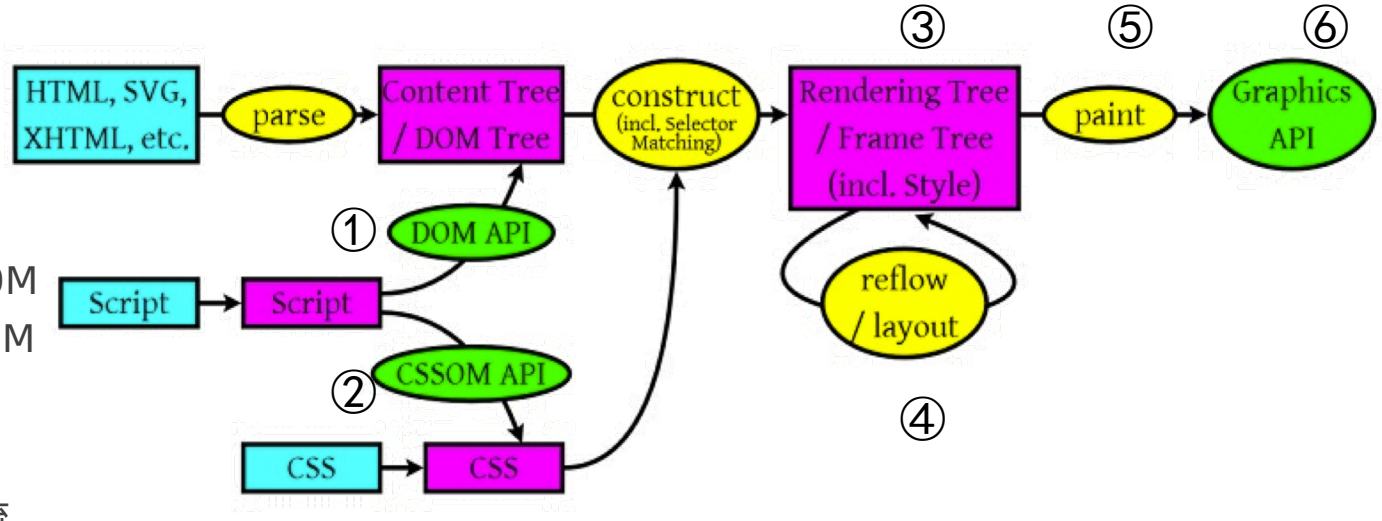
极客时间

- 3.8 同构直出技术方案
- 3.9 PWA 技术方案和实现思路
- 3.10 页面渲染技术选型的合理化建议
- 3.11 页面加载策略优化
- 3.12 接口服务调用优化
- 3.13 接口缓存策略优化

# 3.1 浏览器渲染过程

# 浏览器渲染过程

1. 浏览器解析 HTML，生成 DOM Tree（Parse HTML）。
2. 浏览器解析 CSS，生成 CSSOM（CSS Object Model）Tree。
3. JavaScript 会通过 DOM API 和 CSSOM API 来操作 DOM Tree 和 CSS Rule Tree，浏览器将 DOM Tree 和 CSSOM Tree 合成渲染树（Render Tree）。
4. 布局（Layout）：根据生成的 Render Tree，进行回流，以计算每个节点的几何信息（位置、大小、字体样式等等）。
5. 绘制（Painting）：根据渲染树和回流得到的几何信息，得到每个节点的绝对像素。
6. 展示（Display）：将像素发送给图形处理器（GPU），展示在页面上。



# 3.2 页面渲染技术架构和方案总览

# 页面渲染技术架构和方案总览

- 服务端渲染
  - 后端同步渲染、同构直出、BigPipe
- 客户端渲染
  - JavaScript 渲染：静态化、前后端分离、单页面应用
  - Web App：Angular、React、Vue 等，PWA
  - 原生 App：iOS 、Android
  - Hybrid App：PhoneGap 、AppCan 等
  - 跨平台开发：RN 、Flutter 、小程序等

# 3.3 后端同步渲染

# 后端同步渲染

极客时间

- 同步渲染步骤（以 JSP 为例）
  - 步骤1：JSP Servlet 映射以 .jsp 结尾的 URL，当 .jsp 文件请求时，servlet 容器知道要调用哪个 Servlet。
  - 步骤2：Servlet 容器检查 Servlet 是否已被编译。
  - 步骤3：如果未在步骤2中进行编译，则 Servlet 容器会将 JSP 转换为 Servlet 代码，并进行编译。
  - 步骤4：Servlet 容器将 JSP 请求转发到编译 JSP Servlet 类。
  - 步骤5：JSP Servlet 类返回并发送给客户端浏览器 HTML。

# 后端同步渲染



- 技术优点
  - 快速实现业务开发并上线。
- 技术不足
  - 需要先实现业务逻辑和功能，再输出 HTML 内容给浏览器，用户需等待 HTML 页面完全加载之后才能看到页面内容；代码耦合严重，不太好业务逻辑和页面模板；需求变更上线周期相对较长。
- 协作方式
  - 前端出静态页面，后端套页面。

# 后端同步渲染

- 选型建议
  - 如果是创业初期验证阶段，或者 20 个研发人员以内的项目，为了满足业务快速验证并一天内多次上线，技术选型上使用服务端同步渲染没有任何问题，一人字就是“快”。
- 协作利器
  - jSmart
  - Velocity.js

# 3.4 静态化技术方案和实现思路

# 静态化

- 定义
  - 静态化是使动态化的网站生成静态 HTML 页面以供用户更好访问的技术，一般分为纯动态化和伪动态化。
- 技术优势
  - 提高了页面访问速度，降低了服务器的负担，因为访问页面时不需要每次去访问数据库。
  - 提高网站内容被搜索引擎搜索到的几率，因为搜索引擎更喜欢静态页面。
  - 网站更稳定，如果后端程序、数据库出现问题，会直接影响网站的正常访问，而静态化页面有缓存，更不容易出现问题。

# 静态化

- 技术不足
  - 服务器存储占用问题，因为页面量级在增加，要占用大量硬盘空间。
  - 静态页面中的链接更新问题会有死链或者错误链接问题。
- 技术实现
  - 跑定时任务，将已有的动态内容进行重定，生成静态的 HTML 页面。
  - 利用模板技术，将模板引擎中模板字符替换为从数据库字段中取出来的值， 同时生成 HTML 文件。

# 静态化

- 协作方式
  - 前端统一写好带有交互的完整静态页面。
  - 后端拆分出静态页面文件，并嵌套在后端模板文件中。

- 选型建议
  - 后端研发人员充分，又需要考虑用户体验、服务器负载的业务。

# 3.5 前后端分离技术与实现

# 前后端分离

- 定义
  - 前后端分离是指研发人员分离、业务代码分离、后端实现业务接口，前端渲染页面。
- 技术实现
  - 后端只负责功能接口实现，提供按照约定的数据格式并封装好的 API 接口。
  - 前端负责业务具体实现，获取到 API 接口数据后，进行页面模板拼接和渲染，独立上线。
- 协作方式
  - 前端负责实现页面前端交互，根据后端 API 接口拼装前端模板。
  - 后端专注于业务功能实现和 API 接口封装。

# 前后端分离

- 技术优势
  - 团队更加专注
  - 提升了开发效率
  - 增加代码可维护性
- 技术架构
  - 后端架构：Java、PHP + Nginx，使用微服务（比如 Dubbo 等）等实现业务的解耦，所有的服务使用某种协议提供不同的服务（比如 JSF 等）。
  - 前端架构：使用Angular、React、Vue 前端框架并部署页面至 CDN 。
  - 前端架构2：使用Angular、React、Vue 前端框架并部署在 Node Server 。

# 前后端分离

- 技术不足
  - 因为前端需要负责一大部分业务逻辑实现，和服务端同步、静态化，需要前端人力非常多。
  - 页面数据异步渲染，不利于 SEO，搜索引擎更喜欢纯静态页面。

- 选型建议
  - 这是大型互联网公司正在采用的开发模式，一句话，如果考虑用户体验，以及前端人力够用，就可以积极采用。

# 3.6 单页面应用技术方案

# 单页面应用

- 定义
  - 单页应用（single-page application，缩写 SPA），通过动态重写当前页面，来与用户交互，而非传统的从服务器重新加载整个新页面。这种方法在使用过程中不需要重新加载页面，避免了页面之间切换打断用户体验，使应用程序更像一个桌面应用程序。
- 技术优点
  - 不错的加载速度：用户往往感觉页面加载非常快，因为一进入页面就能看到页面元素；
  - 良好的交互体验：进行局部渲染，避免不必要的页面间跳转和重复渲染；
  - 前后端职责分离：前端进行页面交互逻辑，后端负责业务逻辑；
  - 减轻服务器负载：服务器只处理数据接口输出，不用考虑页面模板渲染和 HTML 展示。

# 单页面应用



- 技术缺点
  - 开发成本相对较高
  - 首次页面加载时间过多
  - SEO 难度比较大
- 技术实现
  - 使用 Angular、React、Vue 框架可以很好的

# 单页面应用

- 选型建议
  - 重交互页面业务
  - 核心链路场景业务

# 3.7 BigPipe 简介和工作模式

# BigPipe

- 定义
  - BigPipe 通过将页面加载到称为 Pagelet 的小部件中，来加快页面渲染速度，并允许浏览器在 PHB 服务器呈现页面的同时，一直请求页面不同区块的结构，类似一个“流”传输管道。

# BigPipe



- 技术实现

1. 浏览器从服务器请求页面。

2. Server 迅速呈现一个包含 <head> 标记的页面框架，以及一个包含空 div 元素的主体，这些元素充当 Pagelet 的容器。由于该页面尚未完成，因此与浏览器的 HTTP 连接保持打开状态。

3. 浏览器将开始下载 bigpipe.js 文件，然后它将开始呈现页面。

4. PHP 服务器进程仍在执行，并且一次构建每个 Pagelet 。Pagelet 完成后，其结果将在 <script> BigPipe.onArrive（...）</ script> 标记内发送到浏览器。

# BigPipe



- 技术实现

5. 浏览器将收到的 html 代码注入正确的位置。如果小页面需要任何 CSS 资源，则也将下载这些 CSS 资源。

6. 接收完所有的页面集之后，浏览器将开始加载那些页面集所需的所有外部 JavaScript 文件。

7. 下载 JavaScript 后，浏览器将执行所有内联 JavaScript 。

# 3.8 同构直出技术方案

# 同构直出

- 定义
  - 一套代码既可以在服务端运行又可以在客户端运行，这就是同构（Universal）。
- 技术优势
  - 性能: 降低首屏渲染时间
  - SEO: 服务端渲染对搜索引擎的爬取有着天然的优势
  - 兼容性: 有效规避客户端兼容性问题，比如白屏
  - 代码同构：直接上线两个版本，利于灾备

# 同构直出

- 技术实现
  - next.js：服务器端渲染 React 组件框架（参考查看：https://nextjs.org/），React 采用 ReactDOMServer 调用 renderToString() 方法。
  - gatsbyjs：服务端 React 渲染框架（参考查看： https://www.gatsbyjs.org/）。
  - nuxt.js：服务器端渲染 Vue 组件框架（参考查看：https://nuxtjs.org/），Vue 采用 vue-server-renderer 调用 renderToString() 方法。

# 同构直出

- 协作方式
  - 后端专注于业务功能实现和 API 接口封装。
  - 前端负责实现页面前端交互，根据后端 API 接口拼装前端模板，页面渲染，以及服务器维护。

- 选型建议
  - 前端要处理 Node server 的机器环境、代码部署、日志、容灾、监控等以往后端人员需要具备运维知识，前端人员的综合能力要求会比以往要高。
  - 前端项目开发周期变长了，需要事先和产品、运营沟通排期问题。
  - 总之一名话：“前端人力和能力够，直接上马，向全栈工程师迈进”。

# 3.9 PWA 技术方案和实现思路

# PWA

- 定义
  - Progressive Web App，简称 PWA，PWA 应用是使用特定技术和标准模式来开发的 Web 应用，这将同时赋予它们 Web 应用和原生应用的特性。

# PWA

- 技术优势
  - 用户可以用手机屏幕启动应用，即使在离线状态或者弱网下，通过事先缓存的资源，也可正常加载运行当前应用，可以完全消除对网络的依赖，从而给用户非常可靠的体验。
  - 因为预先缓存了资源，部分资源无须经过网络，即秒开页面。
  - 和移动设备上的原生应用一样，具有沉浸式的用户体验。
  - 内容可以被搜索引擎收录。
  - 可以给用户发送离线推送消息。

# PWA

- 技术实现
  - 全站改造成 HTTPS，没有 HTTPS 就没有 Service Worker。
  - 应用 Service Worker 技术提升性能，离线提供静态资源文件，提升首屏用户体验。
  - 使用 App Manifest。
  - 最后可以考虑离线消息推送等功能。
- 浏览器兼容性
  - ServiceWorkerGlobalScope API 88%
  - Web App Manifest 83%

# 3.10 页面渲染技术选型的合理化建议

# 页面渲染技术选型的合理化建议



- 存在即合理
- 依赖业务形式
- 依赖团队规模
- 依赖技术水平
- 没有银弹

# 3.11 页面加载策略优化

# 懒加载

- 定义：
  - 懒加载也叫延迟加载，指的是长网页中延迟加载特定元素（可以是图片，也可以是 JS/CSS 文件，当然也可以是 JavaScript 的特定函数和方法，以下简称“懒加载元素”）。
- 好处：
  - 可以减少当前屏无效资源的加载。
- 实际方式和示例：
  - 把页面上“懒加载元素”src 属性设置为空字符，把真实的 src 属性写在 data-lazy 属性中，当页面滚动的时候监听 scroll 事件，如果“懒加载元素”在可视区域内，就把图片的 src 属性或者文件 URL 路径设置成 data-lazy 属性值。

# 预加载

- 定义：
  - 可以使用预加载让浏览器来预先加载某些资源（比如图片、JS/CSS/模板），而这些资源是在将来才会被使用到的。简单来说，就是将所需资源提前加载到浏览器本地，这样后面在需要使用的时候就可以直接从浏览器缓存中取了，而不用再重新开始加载。
- 好处：
  - 减少用户后续加载资源等待的时间。

# 预加载

- 实现方式：

1. HTML 标签

```
<img
src="https://img10.360buyimg.com/n4/g7/M03/08/0D/rBEHZlBzwZwIAAA
AAAI4sOvIiLkAABpMQDf8E4AAjjI749.jpg" style="display:none"/>
```

2. 使用 Image 对象

```
var image= new Image();

image.src=https://img10.360buyimg.com/n4/g7/M03/08/0D/rBEHZlBzwZwIAA
AAAAI4sOvIiLkAABpMQDf8E4AAjjI749.jpg
```

# 预加载

- 实现方式：

3. 使用 preload，prefetch 和 preconnect

```
<link rel=“preload” href=“src/style.css” as=“style” >
<link rel="prefetch" href="scr/image.png">
<link rel="dns-prefetch" href="https://my.com">
<link rel="preconnect" href="https://my.com" crossorigin>
```

# 预渲染

- 好处：

  1. 对于大型项目，在懒加载组件被加载之前，组件可能还会有其它懒加载组件的代码或数据，所以用户还是需要时间等待组件加载完成。

  2. 那另外一种预加载组件的方式就是提前渲染它，在页面中渲染组件，但是并不在页面中展示，也就是渲染好后先隐藏起来，用的时候再直接展示。

- 实现方式：

```
<link rel="prerender" href="https://my.com">
```

# 按需加载

- 常规按需加载（如 JS 原生、jQuery）
- 不同 App 按需加载（如 JS-SDK 脚本文件）
- 不同设备按需加载（如 PC 端和 HTML5 端样式文件）
- 不同分辨率按需加载（CSS Media Query）

# 按需加载

- React 异步载入

```
const componentA = (location, callback) => {
    require.ensure([], require => {
        callback(null, require('modules/componentA'))
    }, 'componentA')
}

const componentB = (location, callback) => {
    require.ensure([], require => {
        callback(null, require('modules/componentB'))
    }, 'componentB')
}

<Router history={history}>
  <Route path="/" component={App}>
    <Route path="componentA" getComponent={componentA}></Route>
    <Route path="componentB" getComponent={componentB}></Route>
  </Route>
</Router>
```

# 按需加载

- Vue 异步载入

```
import Vue from 'vue';
import App from './App.vue';
import VueRouter from 'vue-router';
Vue.use(VueRouter);

const componentA = resolve => require(['src/a.vue' ], resolve);
const componentB = resolve => require(['src/b.vue' ], resolve);

const router = new VueRouter({
  routes: [{path:"a”  ,name:"/a”  ,component:componentA},
        {path:"b”  ,name:"/b”  ,component:componentB}]
  })

new Vue({
  el: '#app',
  router: router,
  render: h => h(App)
})
```

# 楼层式加载

- 定义：
  - 楼层数据异步加载和本地缓存相结合的方式
- 实现方式：

```
<div class="lazy-fn" data-title="服饰" id="lazy-clothes" data-
path="floor1-floor_index.js" data-
time="01d15d664a61ff8f11cf6321f5b7a503"></div>
```

# 3.12 接口服务调用优化

# 3.12 接口服务调用优化

1. 接口合并

   - 这个是指一个页面的众多的业务接口和依赖的第三方接口统一使一个部署在集群的接口统一调用，以减少页面接口请求数。

2. 接口上 CDN

   - 主要基于接口性能考虑，我们可以把不需要实时更新的接口同步至 CDN，等此接口内容变更之后自动同步至 CDN 集群上。如果一定时间内未请求到数据，会用源站接口再次请求。

3. 接口域名上 CDN

   - 增强可用性、稳定性。

# 3.12 接口服务调用优化

4. 接口降级

- 这个基于大促备战考虑，核心接口进行降级用基础接口进行业务实现，比如千人千面的推荐接口，在大促时间点可以直接运营编辑的数据。另外接口万一无法访问，使用预设好的垫底备份数据。

5. 接口监控

- 监控接口成功率，不是常说的 TP99，而是和用户实际情况一致的成功和失败监控，包括比如弱网、超时、网络异常、网络切换等情况。排查出来问题需要联合后端、运维、网络岗位人员一并解决。

# 3.13 接口缓存策略优化

# 3.13 接口缓存策略优化

1. Ajax/fetch 缓存

   - 前端请求时候带上 cache，依赖浏览器本身缓存机制。

2. 本地缓存

   - 异步接口数据优先使用本地 localStorage 中的缓存数据。

3. 多次请求

   - 接口数据本地无 localStorage 缓存数据，重新再次发出 ajax 请求。

极客时间

扫码试看/订阅

《前端全链路性能优化实战》视频课程