性能优化

> https://mp.weixin.qq.com/s/o6iM32E_ggYSDBMjTdN_Rg

提起`性能优化`很多人眼前浮现的面试经验是不是历历在目呢？反正，性能优化在我看来他永远是前端领域的`热度之王`。

最近维护的项目恰巧在这个方向下了很大功夫，一些经验之谈奉上，希望对大家有些许帮助！

性能优化标准

既然说性能优化，那他总得有一个公认的标准，这就是我们很多次听到的`Lighthouse`

在很多单位，都有着自己的性能监控平台，我们只需要引入相应的sdk，那么在平台上就能分析出你页面的存在的性能问题，大家是不是学的很神奇！

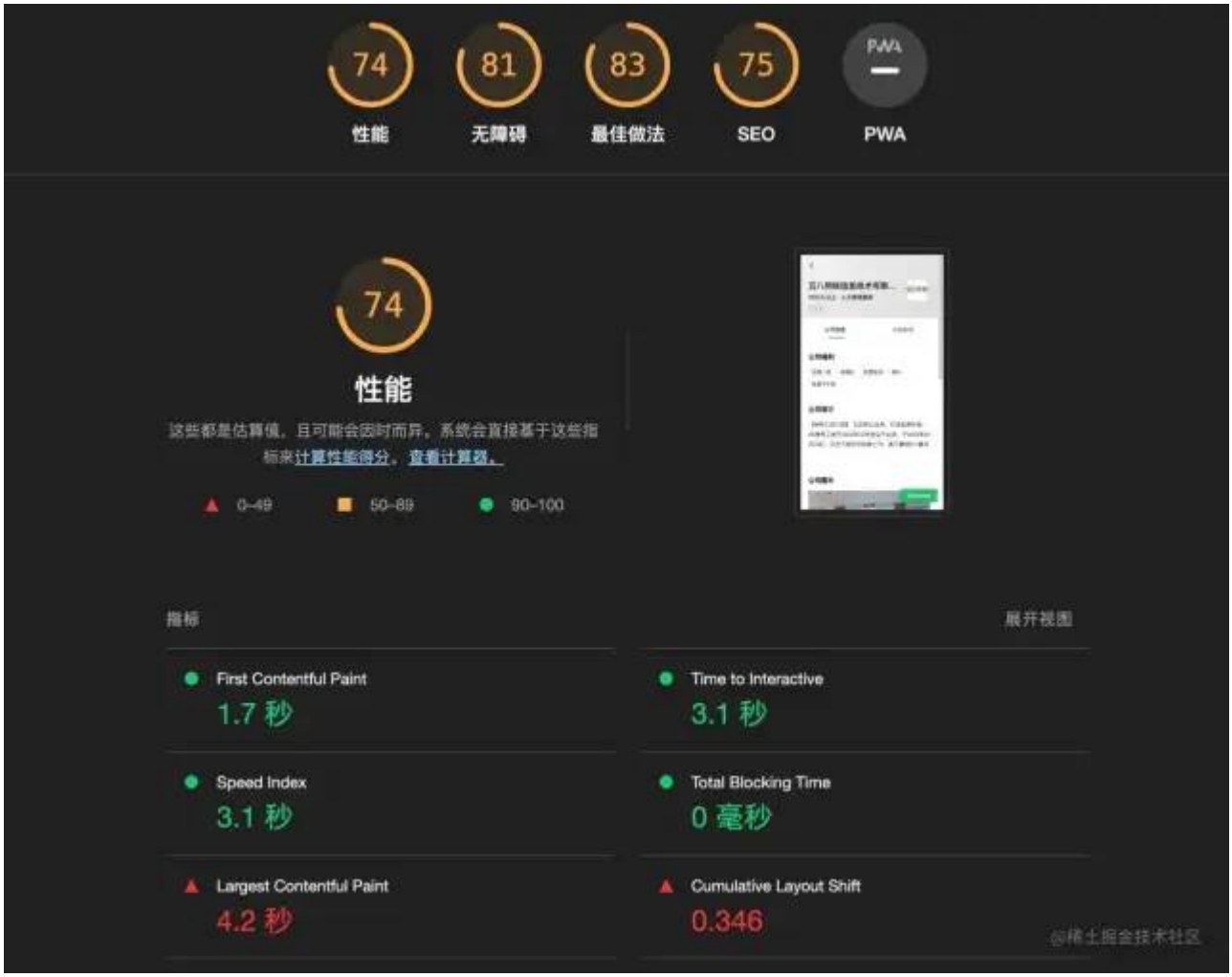
其实除了苛刻的业务，需要`特殊的定制`，大多数的情况下我们单位的性能优化平台本质上其实就是利用无头浏览器（Puppeteer）跑`Lighthouse`。

理解了我们单位的性能监控平台的原理之后，我们就能针对性的做性能优化，也就是面向`Lighthouse`编程

Lighthouse

lighthouse[1] 是 Google Chrome 推出的一款开源自动化工具，它可以搜集多个现代网页性能指标，分析 Web 应用的性能并生成报告，为开发人员进行性能优化的提供了参考方向。

说起`Lighthouse`在现代的谷歌浏览器中业已经集成



他可以分析出我们的页面性能，通过几个指标

Lighthouse 会衡量以下性能指标项：

- 首次内容绘制[2]（First Contentful Paint）。即浏览器首次将任意内容（如文字、图像、canvas 等）绘制到屏幕上的时间点。
- 可交互时间[3]（Time to Interactive）。指的是所有的页面内容都已经成功加载，且能够快速地对用户的操作做出反应的时间点。
- 速度指标[4]（Speed Index）。衡量了首屏可见内容绘制在屏幕上的速度。在首次加载页面的过程中尽量展现更多的内容，往往能给用户带来更好的体验，所以速度指标的值约小越好。
- 总阻塞时间[5]（Total Blocking Time）。指First Contentful Paint 首次内容绘制 (FCP)与Time to Interactive 可交互时间 (TTI)之间的总时间
- 最大内容绘制[6]（Largest Contentful Paint）。度量标准报告视口内可见的最大图像或文本块的呈现时间
- 累积布局偏移[7]（# Cumulative Layout Shift）。衡量的是页面整个生命周期中每次元素发生的非预期布局偏移得分的总和。每次可视元素在两次渲染帧中的起始位置不同时，就说是发生了LS（Layout Shift）。

在一般情况下，据我的经验，由于性能监控平台的和本地平台的`差异`，本地可能要达到**70分**，线上才有可能达到及格的状态,如果有性能优化的需求时，大家酌情处理即可（不过本人觉得，及格即可， 毕竟`大学考试有曰：60分万岁，61分浪费`，传承不能丢，咱们要把更多的时间，放到更重要的事情上来!）

通用常规优化手段

`lighthouse`的的牛x之处就是它能找出你页面中的一些常规的性能瓶颈，并提出优化建议，比如：

于是针对这些优化建议，我们需要做一些常规的优化：

1. 减少未使用的javascript
2. 移出阻塞渲染的资源
3. 图片质量压缩
4. 限制使用字体数量，尽可能少使用变体
5. 优化关键渲染路径：只加载当前页面渲染所需的必要资源，将次要资源放在页面渲染完成后加载

通用性能优化分析

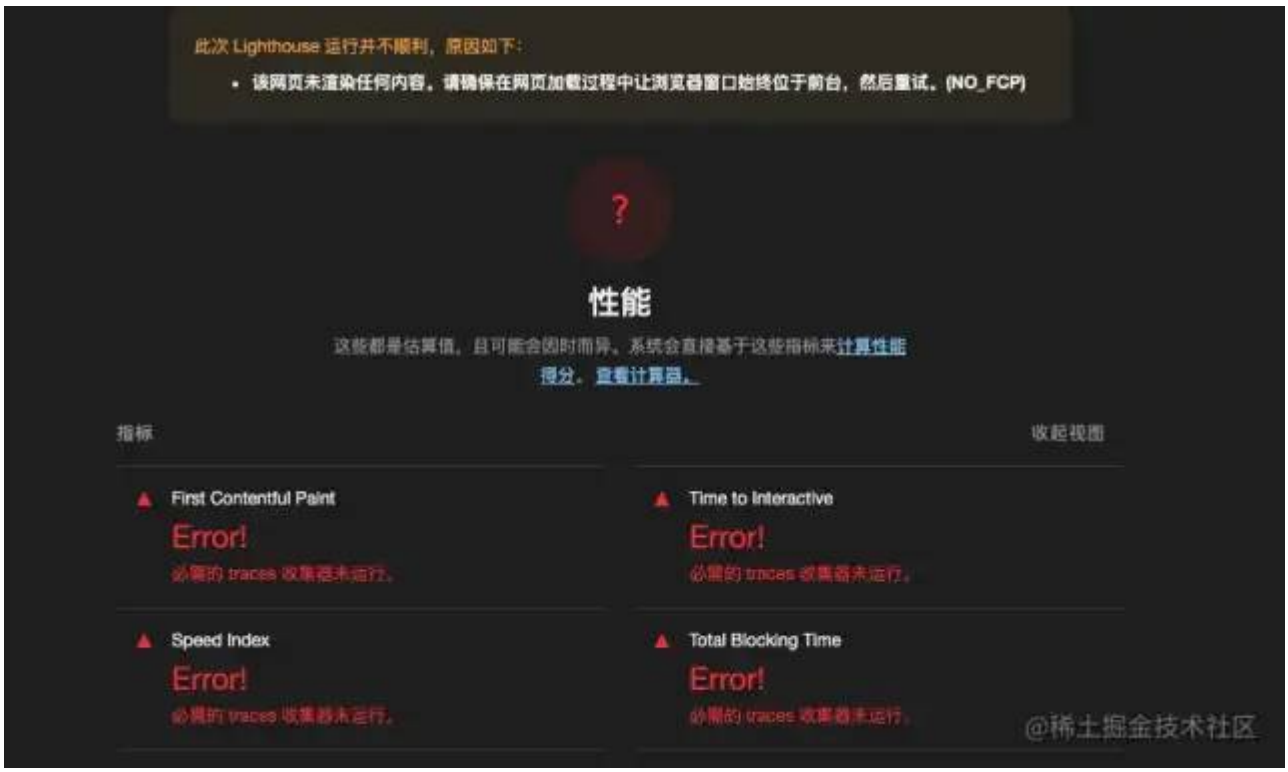
我们知道lighthouse 中有六个性能指标，而在这六个指标中，`LCP、 FCP、speed index`、 这三个指数尤为重要，因为在一般情况下 这个三个指标会影响 `TTI、TBT、CLS` 的分数

所以在我们在优化时， 需要提高LCP、 FCP和speedIndex 的分数，经过测试， 即使是空页面也会有时间上的损耗， 初始分数基本都是`0.8`秒

注意：需要值得大家注意的是，我们当前所有测试全部建立在，`移动端`（之所以用移动端，是由于pc 的强大算力，很少有性能瓶颈）的基础上,并且页面上必须有一下内容，才能得出分数，内容必须包括一下的一种或者多种

- 内嵌在svg元素内的image元素
- video元素（使用封面图像）
- 通过url()[8]函数（而非使用CSS 渐变[9]）加载的带有背景图像的元素
- 包含文本节点或其他行内级文本元素子元素的块级元素[10]。

否则就会有如下错误



接下来我们就从LCP、 FCP和speedIndex 这三个指标入手

FCP（First Contentful Paint)

顾名思义就是`首次内容绘制`，也就是页面最开始绘制内容的时间，但是由于我们现在开发的页面都是spa应用，所以，框架层面的初始化是一定会有一定的`性能损耗`的，以vue-cli 搭建的脚手架为例，当我初始化空的脚手架，打包后上传cdn部署，FCP 就会从0.8s提上到1.5秒，由此可见vue 的diff 也不是`免费`的他也会有性能上的损耗

在优化页面的内容之前我们声明三个前提

1. 提高FCP的时间其实就是在优化关键渲染路径[11]
2. 如果它是一个样式文件（CSS文件），浏览器就必须在渲染页面之前完全解析它（这就是为什么说CSS具有渲染阻碍性）
3. 如果它是一个脚本文件（JavaScript文件），浏览器必须：停止解析，下载脚本，并运行它。只有在这之后，它才能继续解析，因为 JavaScript 脚本可以改变页面内容（特别是HTML）。（这就是为什么说JavaScript阻塞解析）

针对以上的用例测试，我们发现，无论我们怎么优化，框架本身的性能损耗是无法抹除的，我们唯一能做的就是让框架更早的去执行初始化，并且初始化更少的内容，可做的优化手段如下：

1. 所有初始化用不到的js 文件全部走异步加载，也就是加上`defer`或者`asnyc`，并且一些需要走cdn的第三方插件需要放在页面底部（因为放在顶部，他的解析会阻止html 的解析，从而影响css 等文件的下载，这也是`雅虎军规`的一条）
2. js 文件拆包，以vue-cli 为例，一般情况下我们可以通过cli的配置 splitChunks 做代码分割，将一些第三方的包走cdn，或者拆包。如果有路由的情况下将路由做拆包处理，保证每个路由只`加载当前路由对应的js代码`
3. 优化文件大小 减少字体包、css文件、以及js文件的大小（当然这些 脚手架默认都已经做了）
4. 优化项目结构，每个组件的初始化都是有`性能损耗`的，在在保证`可维护性`的基础上，尽量减少初始化组件的加载数量
5. 网络协议层面的优化，这个优化手段需要服务端配合纯前端已经无法达到，在现在`云服务器`盛行的时代,自家单位一般都会默认在云服务器中开启这些优化手段，比如开启`gzip`，使用`cdn`等等

其实说来说去，提高FCP 的核心只有理念之后两个`减少初始化视图内容`和`减少初始化下载资源大小`

LCP(Largest Contentful Paint)

顾名思义就是`最大内容绘制`， 何时报告LCP,官方是这样说的

为了应对这种潜在的变化，浏览器会在绘制第一帧后立即分发一个`largest-contentful-paint`类型的PerformanceEntry[12]，用于识别最大内容元素。但是，在渲染后续帧之后，浏览器会在最大内容元素发生变化时分发另一个`Perf``ormanceEntry`。

例如，在一个带有文本和首图的网页上，浏览器最初可能只渲染文本部分，并在此期间分发一个`largest-contentful-paint`条目，其`element`属性通常会引用一个`<p>`或`<h1>`。随后，一旦首图完成加载，浏览器就会分发第二个`largest-contentful-paint`条目，其`element`属性将引用`<img>`。

需要注意的是，一个元素只有在渲染完成并且对用户可见后才能被视为最大内容元素。尚未加载的图像不会被视为"渲染完成"。在字体阻塞期[13]使用网页字体的文本节点亦是如此。在这种情况下，较小的元素可能会被报告为最大内容元素，但一旦更大的元素完成渲染，就会通过另一个`PerformanceEntry`对象进行报告。

其实用大白话解释就是，通常情况下，`图片、视频以及大量文本绘制完成后`就会报告LCP

理解了这一点，的优化手段就明确了,尽量减少这些资源的大小就可以了，经过测试，减少首屏渲染的图片以及视频内容大小后，整体分数显著提高，提供一些优化方法：

1. 本地图片可以使用在线压缩工具自己压缩 推荐tinypng.com[14]
2. 接口中附带图片，一般情况下单位中都有对应的oss或者cdn传参配置通过地址栏传参方式控制图片质量
3. 图片懒加载

SpeedIndex（速度指数）

`Speed Index`采用可视页面加载的视觉进度，计算内容绘制速度的总分。为此，首先需要能够计算在页面加载期间，各个时间点“完成”了多少部分。

在WebPagetest中，通过捕获在浏览器中加载页面的视频并检查每个视频帧（在启用视频捕获的测试中，每秒10帧）来完成的，这个算法在下面有描述，但现在假设我们可以为每个视频帧分配一个完整的百分比（在每个帧下显示的数字）

以上是官方解释的计算方式，其实通俗的将，所谓速度指数就是衡量页面内容填充的速度

一图胜千言

经过测试，跟LCP相同，图片以及视频内容对于SpeedIndex的影响巨大，所有优化方向，通之前一致，总的来说，只要提高LCP 以及FCP 的时间SpeedIndex 的时间就会有显著提高

不过需要注意的是，接口的速度也会影响SpeedIndex的时间，由于AJAX流行的今天，我们大多数的数据都是使用接口拉取。如果接口速度过慢，他就会影响你页面的初始渲染， 导致性能问题，所以，在做性能优化的同时，`请求后端伙伴协助，也是性能优化的一个方案`

排查性能瓶颈

上述分析，根据三个指标提供了一些常规的优化手段，那么在这些优化手段中，有的你可以立马排查到，并且优化例如：

1. 优化图像，优化字体大小
2. 跟服务端配合利用浏览器缓存机制.启用cdn、启用gzip等
3. 减少网络协议过程中的消耗，减少http 请求、减少dns查询、避免重定向
4. 优化关键渲染路径，异步加载js等

但是有的优化手段我们不容易排查，因为他是打在包里面的，这个js 文件包含了很多逻辑怎么办，这里我有两个手段或许能够帮助排查出性能瓶颈发生在哪里：

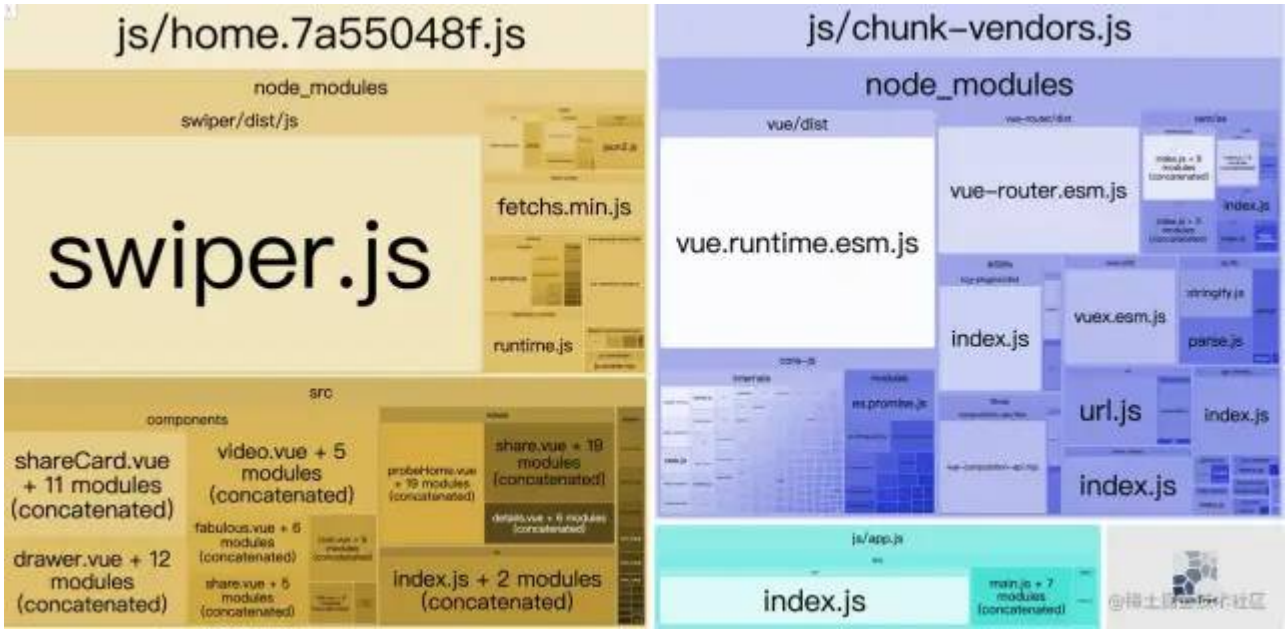
分析包内容

在通常情况下，我们无法判断的优化点，都是在打包后，我们无法分析出，那些东西不是我们在首屏必须需要的，从而不能做出针对新的优化，为了解决当前问题，各大bundle厂商也都有各自的分析包的方案

以vue-cli 为例

```
"report": "vue-cli-service build --report"
```

我们只需要在脚手架中提供以上命令，就能在打包时生成，整个包的分析文件



如上图所示 在打包后就能分析出打包后的js 文件他包含什么组件，如此以来，我们就能知道那些文件是没必要同步加载的，或者走cdn的，通过配置将他单独的隔离开来，从而找出性能的问题

利用chorme devtool 的代码覆盖率

如下图所示，

利用 devtool的代码覆盖率检查就能知道那些js 或者css 文件的代码没有被使用过，结合包内容的分析，我们就能大概的猜出性能的瓶颈在哪里从而做相应的特殊处理

针对vue 的特殊优化

以上内容都是通用的一些优化手段，您在哪都能查到，只是我表达了一下做这些常规优化的深层原因。能让您更清楚的了解这些原因之后，`在性能瓶颈的时候能游刃有余，而不是为了面试死记硬背，一到用的时候就不灵`

然后我司是vue啊，咱得上得vue 的手段

图片懒加载

所谓图片懒加载，就是页面只渲染当前可视区域内的图片，如此一来，减少了其他图片渲染数量，能大大提高 `SpeedIndex` 和 `LCP` 的时间，从而提高分数

在vue中提起图片懒加载插件，首推vue-lazyload[15]

使用方式简单，功能丰富

虚拟滚动

在一含有长列表页面中,你有没有发现你是往下越滑越卡，此时虚拟滚动就排上用场了， 他的基本原理就是只渲染可视区域内的几条数据，但是模拟出正常滑动的效果，因为每次只渲染可是剧域内的数据,在滑动的时候他的性能就会有飞速提升

在vue中比较好用的插件有两个vue-virtual-scroller[16]和vue-virtual-scroll-list[17]

目前我司统一用的vue-virtual-scroll-list 他下拉的时候到了分页的地方能加些loading提示

vue 中的函数式组件

在vue中我们知道组件的初始化是比较损耗性能的，大家可以去试一下，使用vue 直接渲染一个文字内容，和直接渲染一个app.vue 组件他的分数是略有不同的。

但是当有了函数式组件，这个问题就迎刃而解了

因为函数是组件顾名思义他就是个函数，说白了就是个`render函数`，他少了组件初始化的过程，省去了很多初始化过程的`开销`

什么时候用函数式组件呢?

当你的组件中没有业务逻辑只展示内容时，这时候函数式组件就派上用场了

利用v-show 、KeepAlive 复用dom

我们知道v-show是通过display 控制dom的展示隐藏，他并不会删除dom 而我们在切换v-show的时候其实是减少了diff的对比，而KeepAlive 则是直接复用dom，连diff 的过程都没了，并且他们俩的合理使用还不会影响到初始化渲染。如此一来减少了js 的执行开销，但是值得注意的是，`他并不能优化你初始化的性能，而是操作中的性能`

分批渲染组件

在前面我们提到过SpeedIndex 的渐进渲染是提高SpeedIndex的关键，有了这个前提，我们就可以分批异步渲染组件。先看到内容，然后在渲染其他内容

举个例子:

```
<template>
    <div>
        {{ data1 }}
    </div>
    <div v-if="data1">
        {{ data2 }}
    </div>
</template>
<script>
import { ref } from 'vue'
export default {
    setup() {
        let data1 = ref('')
        let data2 = ref('')
        // 假设 这是从后端取到的数据
        const data = {
            data1: '这是渲染内容1',
```

```
            data2: '这是渲染内容2'
        }
        data1.value = data.data1
        //利用requestAnimationFrame 在空闲的时候当前渲染之后在渲染剩余内容
        requestIdleCallback(() => {
            data2.value = data.data2
        })
        return {
            data1,
            data2
        }
    },
}
</script>
```

上述例子比较简单可能描述的不太贴切，在这里特此说明一下，当前方法适用于组件内容较多，每次render 时间过长，导致白屏时间过长，比如，`一次拉取用户列表`，那么分批渲染就非常合适，先展示一部分用户信息，最后直到慢慢将所有内容渲染完毕。如此对浏览器的SpeedIndex 也非常友好

最后

性能优化一直是一个很火的话题， 不管从面试以及工作中都非常重要，有了这些优化的点，你在写代码或者优化老项目时都能游刃有余，能提前考虑到其中的一些坑，并且规避。

但是大家需要明白的是，不要为了性能优化而性能优化，我们在要`因地制宜`，在不破坏项目可维护性的基础上去优化，千万不要你优化个项目性能是好了，但是大家都看不懂了，这就有点得不偿失了，还是那句话，`60分万岁61份浪费，差不多得了`，把经历留着去干更重要的事情!