如何回答出让面试官满意的前端性能优化?

引言

相信大家在平常的面试过程中或多或少都被问到过前端性能优化如何实现，但是大部分同学的回答可能说一说自己做了aa、bb、cc...配置；亦或是说了这些配置之后，没有量化的结果去衡量自己的优化到底有没有用。

这时候面试官可能就会想：“这位同学的性能优化知识，不成体系，想到哪儿说到哪儿，并且他没有的量化指标，谁知道他是不是从网上背的配置呢。”

本篇文章不会提及过多的具体实现，只会提方向，如何回答问题，具体的解决方案可以大家自行查阅。

面试官想考察的是什么呢?

首先，在我看来，问性能优化问题的面试官大多都是处于以下几点目的：

1. 面试者是否有做过前端性能优化方面的实践。
2. 面试者是否有系统的性能优化知识。（不一定强要求）

我们应该如何回答?

一般面试官的问题大致分为两种情况：

1. 你在XX项目中**做过**哪些性能优化？（**具体实现**）
2. 你如何对一个项目进行性能优化，或者说说你对前端性能优化的看法？（**整体理解，系统优化**）

> 性能指标的知识会在**整体理解**篇里具体说明

（一）做过的具体实现：

回答的要点：

1. 首先，我们一定要选自己**做过**的性能优化的点来说。
2. 最好要有**性能指标**来**量化**衡量。
3. 回答的时候一定要**自信**，你要坚信在你做过的地方没人比你更懂，**不要吞吞吐吐**，最好的面试都是**交流**而不是他问你答。

回答的结构：

1. **性能问题的出现**：在今年的xx月，测试同事发现在这个项目的xx页面加载的时候出现**卡顿**。（这点其实能编，注意 对于产品、测试、用户 而言，能直观感受到的就是卡顿、慢）

2. **问题复现**：随后我打开页面，通过**工具测试**发现（这里的工具可以是`performance`、`lighthouse`、`前端埋点SDK`亦或者`其他第三方的监测工具`，你要说你直接调用浏览器的`performance API`估计也能行但不推荐，容易被面试官反问为什么不封装个性能检测工具...orz）**几个性能指标存在问题**：`FCP`、`TTI`这两个性能指标都过长，`FCP`达到了3.x秒，`TTI`更是长达5.x秒（不要选太多性能指标，很多性能指标可以不纳入你们公司的衡量范围，or你编的衡量范围~~）。
3. **问题分析**：（分析过程相信大家都有，这段大家可以自己想想，在此我以`FCP`为例）我发现在xx页面加载的时候会先获取几张比较大的图片，导致`FCP`指标过长。
4. **优化方案**：采取了图片优化策略xxx执行优化。（下文提及哪些优化策略）
5. **量化优化效果**：在经过上述的优化方案后，我们最终将`FCP`优化到了1.8秒，`TTI`优化到了3.8秒。（量化你的优化成果）
6. （非必要）**优化是否达标**：如果同学们的公司对性能指标的数据有强要求，比如`FCP`必须在2秒以内诸如此类...，可以提一下，可以代表你在之前的公司是有完善的性能优化流程的。

这就是一套**比较科学**的回答结构，大家可以参考或者在评论区补充。

（二）性能优化系统方案（整体理解）

回答的要点：

1. 突出自己性能优化知识的**系统性**。
2. 结合**性能指标**，笔者认为没有量化的性能优化都是耍流氓。
3. 回答的时候尽量不要卡壳，想不起来可以暂时跳过，面试官觉得缺了一点的时候，你可以经过他的提示再补充。

由于此处不涉及具体问题，在此我直接给出一份我认为比较系统的前端性能优化方案及结构，并圈出要点。

第一步，如何对项目进行性能分析

通常来说，在大公司里面会采用前端埋点SDK，而中小型公司可能会直接使用浏览器的扩展工具：`performance`和`lighthouse`这里不对工具做具体的使用分析，直接说我们**需要得到什么：性能指标！**

指标是衡量我们项目的性能最最重要的东西，**笔者始终认为，没有指标的性能优化都是耍流氓**。

我们常见的指标有以下这些：

- **load**（Onload Event），它代表页面中依赖的所有资源加载完的事件。
- **DCL**（DOMContentLoaded），DOM解析完毕。
- **FP**（First Paint），表示渲染出第一个像素点。FP一般在HTML解析完成或者解析一部分时候触发。

- **FCP**（First Contentful Paint），表示渲染出**第一个内容**，这里的“内容”可以是文本、图片、canvas。
- **FMP**（First Meaningful Paint），首次渲染有意义的内容的时间，“有意义”没有一个标准的定义，FMP的计算方法也很复杂（建议不使用，或者结合产品经理讨论使用）。
- **LCP**（largest contentful Paint），最大内容渲染时间。

这里不再对性能指标进行详细解析和分析其计算方式，网上有大量的文章解析，主要还是回到我们回答问题上。

> **第一句话：我会通过`performance`工具对项目页面性能进行分析（有埋点SDK最好），从中筛选出部分指标作为本项目的性能衡量指标（指标一般不同公司有不同标准）。**

第二步，系统的优化方案

在本篇中会系统性结构化的说明前端常见的性能优化方案，可能由于笔者的习惯，会于主流的结构划分有点差异，但归根结底，这一步主要是**体现自己系统化的知识**。

开发时性能优化

其实在很多文章中，这一步也被划分到了**加载性能**或者**网络层面**中，但笔者还是觉得这个东西对于用户来说是没有感知的，只是会影响我们日常开发或者说打包的速度，应该单独提出来说。

> 这一步其实主要是构建打包方面的问题，其实具体的操作与`Vite`和`Webpack`等工具是强耦合的，在此我还是只提方向举一小部分例，具体实现还需要各位针对不同工具去实践

- **缩小加载范围：配置include/exclude缩小Loader对文件的搜索范围**，好处是`避免不必要的转译`。不然所有`node_modules`都跑一边那不是卡死了。
- **打包缓存**：很多工具都可以**开启打包的缓存**，这一步能大大减少构建时间。如`Umi`的`MFSU`或者`hardsource-webpack-plugin`等实现缓存效果的工具都是笔者见过效果立竿见影的。
- **提前构建：配置DllPlugin将第三方依赖提前打包**，好处是`将DLL与业务代码完全分离且每次只构建业务代码`。（这个玩意儿非常老了，并且我在三年前的实践中就感觉他速度提升不是很明显，**可以不用提及**）
- **并行构建**：释放CPU多核并发的优势。诸如`happyPack`、`thread-loader`等工具都可以在不同阶段开启CPU多核进行并行构建，大大提升开发时效率。
- **可视化分析**：对打包后的文件大小进行可视化分析，能够更好的分析哪些包比较大，或者小的进行合并。如`Vite`的`rollup-plugin-visualizer`、和`Webpack`的`webpack-bundle-analyzer`。

生产时性能优化

大家可以想一道非常常见的面试题：**从浏览器输入URL到页面渲染完成经历了哪些过程?**

这其实就是我们整个生产过程中需要优化的地方

这一步是最能体现我们系统性能优化知识的地方，笔者看了诸多文章，总结了一下，分为两个模块：

加载层面，顾名思义，就是项目文件的网络加载过程。

渲染层面，也是顾名思义，就是我们拿到文件后页面开始渲染并且交互的过程。

接下来我们按照这个结构，系统的讲讲应该做些什么？

加载层面

首先，加载层面就是网络在加载文件，核心要点就是**快**，怎么快呢：

1. 文件小
2. 网络快
3. 缓存

核心策略：

我们依次再说具体一点的方案：

- **构建策略：减小文件体积：**
  - **代码分割：** `Split Chunk`
  - `Tree Shaking`：其实大部分工具已经自带了
  - **动态垫片：通过垫片服务根据UserAgent返回当前浏览器代码垫片**，好处是 `无需将繁重的代码垫片打包进去` 。每次构建都配置 `@babel/preset-env` 和 `core-js` 根据某些需求将 `Polyfill` 打包进来，这无疑又为代码体积增加了贡献

    - **按需加载**：使用的时候才加载
    - **压缩资源**：**压缩HTML/CSS/JS代码，压缩字体/图像/音频/视频**，好处是`更有效减少打包体积`。
    - **图像处理**：在此单独提出图像的压缩处理，是因为大多数情况下，对图片进行优化的成效往往是巨大的，可能远程你分包，修改代码的优化程度，而且花费时间甚少。图像的选型往往也可以在不同场景下提供不同的效果。这里不再展开。
- **网络策略：CDN**：`CDN`即时内容分发网络。使用`CDN`可降低网络拥塞，提高用户访问响应速度和命中率。其核心特征是`缓存`和`回源`，缓存是把资源复制到`CDN服务器`里，回源是`资源过期/不存在`就向上层服务器请求并复制到`CDN服务器`里。（此种方式虽然成本较高，但是中大型的公司一般都会有购买`CDN`）
- **缓存策略：强缓存、协商缓存**：这也是非常常见的浏览器缓存方案，这里不对其原理和配置进行描述。应用场景都可根据项目需求制定。

渲染层面

其次，便是我们的渲染过程，上面的优化策略大多是配置方面，渲染层面的配置就落实到我们前端的具体技术细节上了。

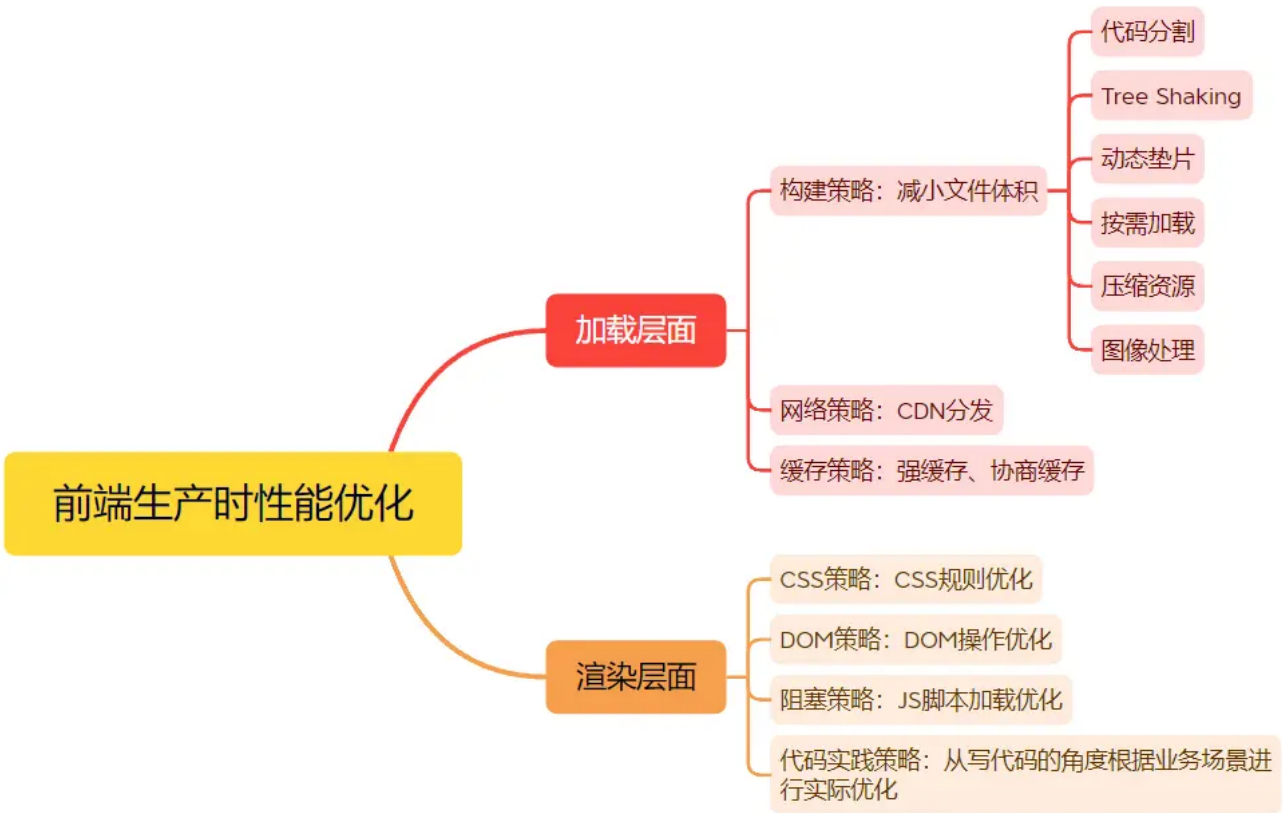
核心策略：

我们依次再说具体一点的方案：

- **CSS策略：**
    - 避免出现多层的`嵌套规则`
    - 避免为`ID选择器`添加多余选择器
    - 避免使用`通配选择器`，只对目标节点声明规则
    - 避免重复匹配重复定义，关注`可继承属性`（这点不强要求）
- **DOM策略：（回流重绘）**
    - 缓存`DOM计算属性`
    - 避免过多`DOM操作`

- 使用 `DOMFragment` 缓存批量化 `DOM操作`
- 使用 `display` 控制 `DOM显隐` ，将 `DOM离线化`
- 在 `异步任务` 中修改 `DOM` 时把其包装成 `微任务`

- **阻塞策略：**
  - 脚本与 `DOM/其它脚本` 的依赖关系很强：对 `<script>` 设置 `defer`
  - 脚本与 `DOM/其它脚本` 的依赖关系不强：对 `<script>` 设置 `async`
- **代码实践策略：**
  - 防抖、节流
  - 懒加载
  - 绘图时可开启GPU加速
  - 时间分片、 `Web Worker` 处理大、长逻辑
  - ……



以上，就是详细系统的优化策略

> 说完第一句后，紧接着说：在前端优化方面，我通常将其分为**两个部分**：
> **开发时性能和生产时性能**。
> 开发时性能主要是提升我们前端开发打包效率.....

> 生产时性能我将其分为**两个层面。第一，加载层面。第二，渲染层面**。（抛出你对性能优化的整体结构划分），加载层面我将其分为以下几个策略：构建、网络、缓存....，渲染层面我将其分为以下几个策略：CSS、DOM、阻塞、代码实践部分.....（同上）
> （如果面试官感兴趣，再细说具体的操作包括可以详细到`Webpack`、`Vite`等工具的使用，以及具体的代码实现）

第三步，系统优化之后如何衡量（非必须）

我们在经过上述系统性优化项目之后，如何检验我们的性能是否达到标准呢？

前文提到过，可能部分同学的公司会对性能指标有很详细的要求，要让面试官感受到你完整的性能优化流程，还需要提一提你们的标准（不用全提）。这里由于项目场景、复杂度、技术栈等不同，肯定指标也不能一概而论。在此我给出一个性能表供大家参考，基本和谷歌对其：

Metric Name	Good(ms)	Needs Improvement(ms)	Poor(ms)
FP	0-1000	1000-2500	Over 2500
FCP	0-1800	1800-3000	Over 3000
LCP	0-2500	2500-4000	Over 4000
TTI	0-3800	3800-7300	Over 7300
FID	0-100	100-300	Over 300

最后一句，优化完之后，我会根据查看项目的性能指标与标准指标进行对比，从而确定是否达标或者进行再优化。