一面 5：浏览器相关知识点与高频考题解析

Web 前端工程师写的页面要跑在浏览器里面，所以面试中也会出现很多跟浏览器相关的面试题目。

知识点梳理

- 浏览器加载页面和渲染过程
- 性能优化
- Web 安全

本小节会从浏览器的加载过程开始讲解，然后介绍如何进行性能优化，最后介绍下 Web 开发中常见的安全问题和预防。

加载页面和渲染过程

可将加载过程和渲染过程分开说。回答问题的时候，关键要抓住核心的要点，把要点说全面，稍加解析即可，简明扼要不拖沓。

> 题目：浏览器从加载页面到渲染页面的过程

加载过程

要点如下：

- 浏览器根据 DNS 服务器得到域名的 IP 地址
- 向这个 IP 的机器发送 HTTP 请求
- 服务器收到、处理并返回 HTTP 请求
- 浏览器得到返回内容

发送 HTTP 请求。

server 端接收到 HTTP 请求，然后经过计算（向不同的用户推送不同的内容），返回 HTTP 请求，返回的内容如下：

其实就是一堆 HMTL 格式的字符串，因为只有 HTML 格式浏览器才能正确解析，这是 W3C 标准的要求。接下来就是浏览器的渲染过程。

渲染过程

要点如下：

- 根据 HTML 结构生成 DOM 树
- 根据 CSS 生成 CSSOM
- 将 DOM 和 CSSOM 整合形成 RenderTree
- 根据 RenderTree 开始渲染和展示
- 遇到 `<script>` 时，会执行并阻塞渲染

上文中，浏览器已经拿到了 server 端返回的 HTML 内容，开始解析并渲染。最初拿到的内容就是一堆字符串，必须先结构化成计算机擅长处理的基本数据结构，因此要把 HTML 字符串转化成 DOM 树 —— 树是最基本的数据结构之一。

解析过程中，如果遇到 `<link href="...">` 和 `<script src="...">` 这种外链加载 CSS 和 JS 的标签，浏览器会异步下载，下载过程和上文中下载 HTML 的流程一样。只不过，这里下载下来的字符串是 CSS 或者 JS 格式的。

浏览器将 CSS 生成 CSSOM，再将 DOM 和 CSSOM 整合成 RenderTree ，然后针对 RenderTree 即可进行渲染了。大家可以想一下，有 DOM 结构、有样式，此时就能满足渲染的条件了。另外，这里也可以解释一个问题 —— **为何要将 CSS 放在 HTML 头部？** —— 这样会让浏览器尽早拿到 CSS 尽早生成 CSSOM，然后在解析 HTML 之后可一次性生成最终的 RenderTree，渲染一次即可。如果 CSS 放在 HTML 底部，会出现渲染卡顿的情况，影响性能和体验。

`<script>` 内容执行完之后，浏览器继续渲染。最后再思考一个问题 —— **为何要将 JS 放在 HTML 底部？** —— JS 放在底部可以保证让浏览器优先渲染完现有的 HTML 内容，让用户先看到内容，体验好。另外，JS 执行如果涉及 DOM 操作，得等待 DOM 解析完成才行，JS 放在底部执行时，HTML 肯定都解析成了 DOM 结构。JS 如果放在 HTML 顶部，JS 执行的时候 HTML 还没来得及转换为 DOM 结构，可能会报错。

关于浏览器整个流程，百度的多益大神有更加详细的文章，推荐阅读下：《从输入 URL 到页面加载完成的过程中都发生了什么事情？》。

性能优化

性能优化的题目也是面试常考的，这类题目有很大的扩展性，能够扩展出来很多小细节，而且对个人的技术视野和业务能力有很大的挑战。这部分笔者会重点讲下常用的性能优化方案。

> 题目：总结前端性能优化的解决方案

优化原则和方向

性能优化的原则是**以更好的用户体验为标准**，具体就是实现下面的目标：

1. 多使用内存、缓存或者其他方法
2. 减少 CPU 和GPU 计算，更快展现

优化的方向有两个：

- **减少页面体积，提升网络加载**
- **优化页面渲染**

减少页面体积，提升网络加载

- 静态资源的压缩合并（JS 代码压缩合并、CSS 代码压缩合并、雪碧图）
- 静态资源缓存（资源名称加 MD5 戳）
- 使用 CDN 让资源加载更快

- 懒加载（图片懒加载、下拉加载更多）
- 减少DOM 查询，对 DOM 查询做缓存
- 减少DOM 操作，多个操作尽量合并在一起执行（ `DocumentFragment` ）
- 事件节流
- 尽早执行操作（ `DOMContentLoaded` ）
- 使用 SSR 后端渲染，数据直接输出到 HTML 中，减少浏览器使用 JS 模板渲染页面 HTML 的时间

详细解释

静态资源的压缩合并

如果不合并，每个都会走一遍之前介绍的请求过程

```html
<script src="a.js"></script>
<script src="b.js"></script>
<script src="c.js"></script>
```

如果合并了，就只走一遍请求过程

```html
<script src="abc.js"></script>
```

静态资源缓存

通过链接名称控制缓存

```html
<script src="abc_1.js"></script>
```

只有内容改变的时候，链接名称才会改变

```html
<script src="abc_2.js"></script>
```

这个名称不用手动改，可通过前端构建工具根据文件内容，为文件名称添加 MD5 后缀。

CDN 会提供专业的加载优化方案，静态资源要尽量放在 CDN 上。例如：

```html
<script src="https://cdn.bootcss.com/zepto/1.0rc1/zepto.min.js"></script>
```

使用 SSR 后端渲染

可一次性输出 HTML 内容，不用在页面渲染完成之后，再通过 Ajax 加载数据、再渲染。例如使用 smarty、Vue SSR 等。

CSS 放前面，JS 放后面

上文讲述浏览器渲染过程时已经提过，不再赘述。

懒加载

一开始先给为 `src` 赋值成一个通用的预览图，下拉时候再动态赋值成正式的图片。如下， `preview.png` 是预览图片，比较小，加载很快，而且很多图片都共用这个 `preview.png` ，加载一次即可。待页面下拉，图片显示出来时，再去替换 `src` 为 `data-realsrc` 的值。

```html
<img src="preview.png" data-realsrc="abc.png"/>
```

另外，这里为何要用 `data-` 开头的属性值？—— 所有 HTML 中自定义的属性，都应该用 `data-` 开头，因为 `data-` 开头的属性浏览器渲染的时候会忽略掉，提高渲染性能。

DOM 查询做缓存

两段代码做一下对比：

```js
var pList = document.getElementsByTagName('p')  // 只查询一个 DOM ，缓存在 pList 中了
var i
for (i = 0; i < pList.length; i++) {
}
```

```
}
```

总结：DOM 操作，无论查询还是修改，都是非常耗费性能的，应尽量减少。

合并 DOM 插入

DOM 操作是非常耗费性能的，因此插入多个标签时，先插入 Fragment 然后再统一插入 DOM。

```js
var listNode = document.getElementById('list')
// 要插入 10 个 li 标签
var frag = document.createDocumentFragment();
var x, li;
for(x = 0; x < 10; x++) {
    li = document.createElement("li");
    li.innerHTML = "List item " + x;
    frag.appendChild(li);  // 先放在 frag 中，最后一次性插入到 DOM 结构中。
}
listNode.appendChild(frag);
```

事件节流

例如要在文字改变时触发一个 change 事件，通过 keyup 来监听。使用节流。

```js
var textarea = document.getElementById('text')
var timeoutId
textarea.addEventListener('keyup', function () {
    if (timeoutId) {
        clearTimeout(timeoutId)
    }
    timeoutId = setTimeout(function () {
        // 触发 change 事件
    }, 100)
})
```

尽早执行操作

```
})
document.addEventListener('DOMContentLoaded', function () {
    // DOM 渲染完即可执行，此时图片、视频还可能没有加载完
})
```

性能优化怎么做

上面提到的都是性能优化的单个点，性能优化项目具体实施起来，应该按照下面步骤推进：

1. 建立性能数据收集平台，摸底当前性能数据，通过性能打点，将上述整个页面打开过程消耗时间记录下来
2. 分析耗时较长时间段原因，寻找优化点，确定优化目标
3. 开始优化
4. 通过数据收集平台记录优化效果
5. 不断调整优化点和预期目标，循环2~4步骤

性能优化是个长期的事情，不是一蹴而就的，应该本着先摸底、再分析、后优化的原则逐步来做。

Web 安全

> 题目：前端常见的安全问题有哪些？

Web 前端的安全问题，能回答出下文的两个问题，这个题目就能基本过关了。开始之前，先说一个最简单的攻击方式 —— SQL 注入。

上学的时候就知道有一个「SQL注入」的攻击方式。例如做一个系统的登录界面，输入用户名和密码，提交之后，后端直接拿到数据就拼接 SQL 语句去查询数据库。如果在输入时进行了恶意的 SQL 拼装，那么最后生成的 SQL 就会有问题。但是现在稍微大型一点的系统，都不会这么做，从提交登录信息到最后拿到授权，要经过层层的验证。因此，SQL 注入都只出现在比较低端小型的系统上。

XSS（Cross Site Scripting，跨站脚本攻击）

举一个例子，我在一个博客网站正常发表一篇文章，输入汉字、英文和图片，完全没有问题。但是如果我写的是恶意的 JS 脚本，例如获取到 `document.cookie` 然后传输到自己的服务器上，那我这篇博客的每一次浏览都会执行这个脚本，都会把访客 cookie 中的信息偷偷传递到我的服务器上来。

其实原理上就是黑客通过某种方式（发布文章、发布评论等）将一段特定的 JS 代码隐蔽地输入进去。然后别人再看这篇文章或者评论时，之前注入的这段 JS 代码就执行了。**JS 代码一旦执行，那可就不受控制了，因为它跟网页原有的 JS 有同样的权限**，例如可以获取 server 端数据、可以获取 cookie 等。于是，攻击就这样发生了。

XSS的危害

XSS 的危害相当大，如果页面可以随意执行别人不安全的 JS 代码，轻则会让页面错乱、功能缺失，重则会造成用户的信息泄露。

比如早些年社交网站经常爆出 XSS 蠕虫，通过发布的文章内插入 JS，用户访问了感染不安全 JS 注入的文章，会自动重新发布新的文章，这样的文章会通过推荐系统进入到每个用户的文章列表面前，很快就会造成大规模的感染。

还有利用获取 cookie 的方式，将 cookie 传入入侵者的服务器上，入侵者就可以模拟 cookie 登录网站，对用户的信息进行篡改。

XSS的预防

那么如何预防 XSS 攻击呢？—— 最根本的方式，就是对用户输入的内容进行验证和替换，需要替换的字符有：

```css
& 替换为：&amp;
< 替换为：&lt;
> 替换为：&gt;
” 替换为：&quot;
‘ 替换为：&#x27;
/ 替换为：&#x2f;
```

替换了这些字符之后，黑客输入的攻击代码就会失效，XSS 攻击将不会轻易发生。

除此之外，还可以通过对 cookie 进行较强的控制，比如对敏感的 cookie 增加 `http-only` 限制，让 JS 获取不到 cookie 的内容。

CSRF 是借用了当前操作者的权限来偷偷地完成某个操作，而不是拿到用户的信息。

例如，一个支付类网站，给他人转账的接口是 `http://buy.com/pay?touid=999&money=100` ，而这个接口在使用时没有任何密码或者 token 的验证，只要打开访问就直接给他人转账。一个用户已经登录了 `http://buy.com` ，在选择商品时，突然收到一封邮件，而这封邮件正文有这么一行代码 `<img src="http://buy.com/pay?touid=999&money=100"/>` ，他访问了邮件之后，其实就已经完成了购买。

CSRF 的发生其实是借助了一个 cookie 的特性。我们知道，登录了 `http://buy.com` 之后，cookie 就会有登录过的标记了，此时请求 `http://buy.com/pay?touid=999&money=100` 是会带着 cookie 的，因此 server 端就知道已经登录了。而如果在 `http://buy.com` 去请求其他域名的 API 例如 `http://abc.com/api` 时，是不会带 cookie 的，这是浏览器的同源策略的限制。但是 —— **此时在其他域名的页面中，请求 `http://buy.com/pay?touid=999&money=100` ，会带着 `buy.com` 的 cookie ，这是发生 CSRF 攻击的理论基础。**

预防 CSRF 就是加入各个层级的权限验证，例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者指纹才行。除此之外，敏感的接口使用 `POST` 请求而不是 `GET` 也是很重要的。

小结

本小节总结了前端运行环境（即浏览器）的一些常考查知识点，包括页面加载过程、如何性能优化以及需要注意的安全问题。

留言

写下你的留言