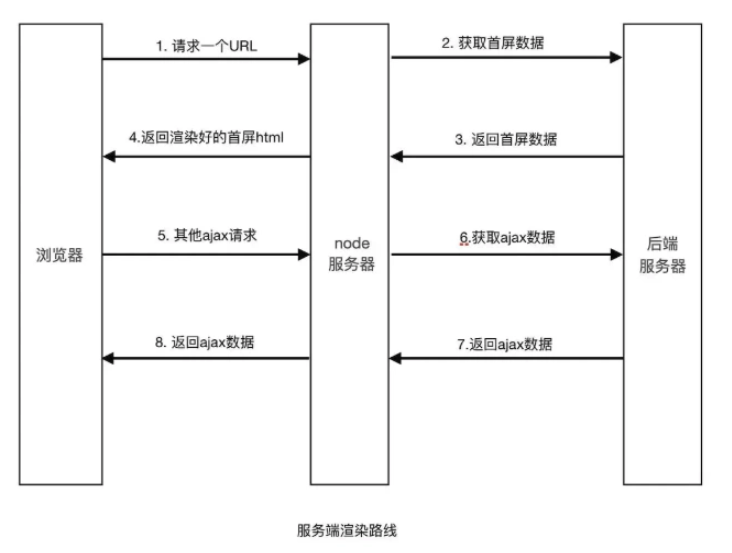
Vue-SSR

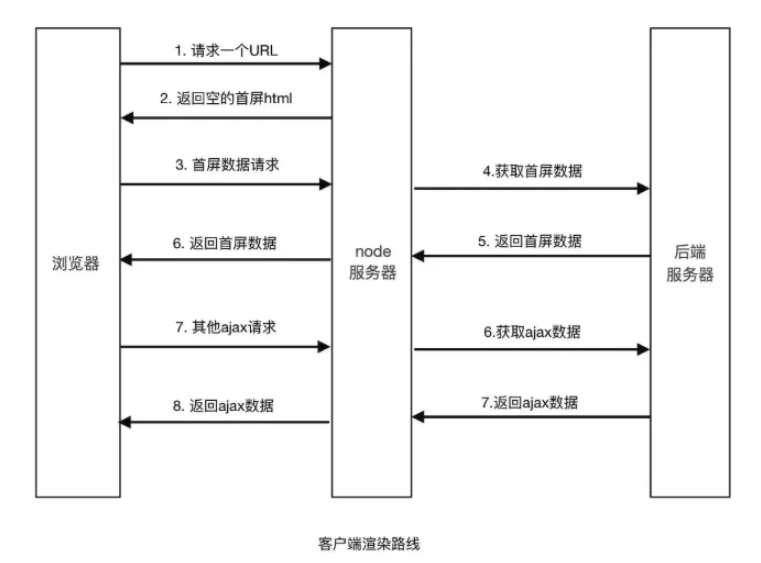
相信大家都不陌生，它与传统 SPA 相比，服务器端渲染 (SSR) 能够具备更好的SEO，方便搜索引擎爬虫，抓取工具可以直接查看完全渲染的页面，除此之外，SSR能够在更短的时间内渲染出页面内容，通过在服务端填充数据吐出到客户端的方式，让用户有更好的用户体验。

首先我们来说一下SSR和CSR的区别

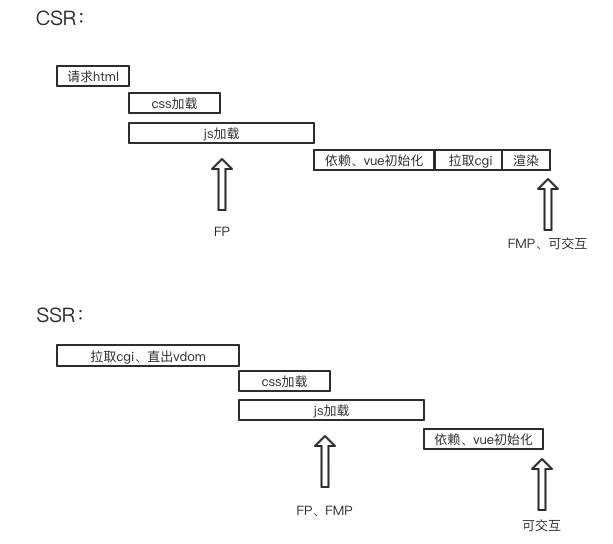
CSR一般由静态资源服务器（CDN）等直接返回HTML资源，之后浏览器解析HTML加载CSS、JS资源，CSS加载结束后页面会尽快进行首屏渲染，也就是FP，JS依赖加载结束后，Vue实例初始化，拉取页面数据，页面渲染（FMP）。

SSR由nodejs服务器来直出页面，请求到达后端后，后端拉取cgi接口数据，根据直出bundle生成render对象，render对象将执行客户端代码构建VDOM，生成HTML string，填充进模板HTML，返回HTML资源，浏览器解析后加载CSS、JS资源，在CSS加载结束后触发FP和FMP，Vue实例初始化，接管后端直出的HTML，页面可响应。





**时序图：**



SSR存在的缺点：

1、对服务器提出更高的要求，生成虚拟DOM如果相对较长的运行和计算耗时；

2、由于cgi拉取和vdom直出后才吐出HTML页面，FMP虽然提前了，但是FP相对延迟了；

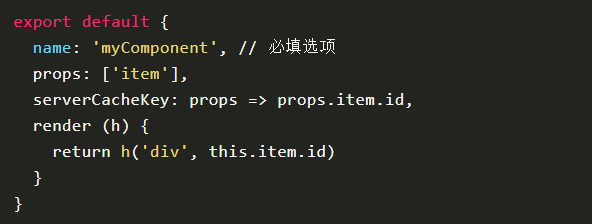
3、相比CSR，SSR渲染后，由于仍然需要进行依赖、vue初始化，页面可交互时间并没有较大改善。

那么，常见优化有哪些呢？

一、缓存优化

1、页面级别缓存：vuessr官网给我们提供了一种方法，如果页面并非千人千面，总是为所有用户渲染相同的内容，我们可以利用名为 micro-caching 的缓存策略，来大幅度提高应用程序处理高流量的能力。这通常在 Nginx 层完成，也可以在 Node.js 中实现。

2、组件级别缓存：通过对组件设置serverCacheKey的方式，如果组件serverCacheKey相同，将复用之前渲染的组件产物，不需要重新渲染。



3、cgi接口缓存：如果部分cgi接口返回的数据是固定的， 我们可以在node后端拉取cgi的时候，设置cgi缓存，缓存至memcache或其他轻量存储服务，当然，你也需要设置好缓存更新策略。

二、代码实现优化

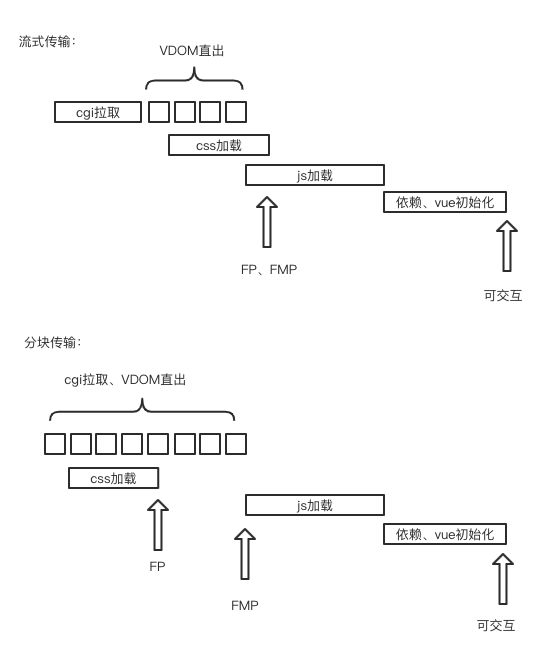
1、减少组件嵌套层次，优化HTML结构：由于组件最初需要在node后端进行VDOM计算和渲染，优化组件层次结构，减少过深曾经的DOM嵌套，可以减少VDOM计算耗时。

2、减少首页渲染数据量：根据业务调整用户首屏可见的所需渲染的数据，其他数据懒加载或异步加载。

三、资源加载

1、流式传输：vuessr官网给我们介绍了一种方法，render对象会暴露renderToStream方法，把原有的直出结果以流的形式输出，让我们可以更快的响应数据到客户端，能减少首屏渲染时间，更早开始加载页面资源。（流式传输需要在asyncData执行结束后开始，否则没有数据，这意味着流式传输受限于cgi拉取耗时）

2、分块传输：由模板的语法树， 分析代码的上下文，分析数据和模板间的依赖，用异步数据分割模板，分块逐步输出。（相比流式传输，前置位的cgi数据一旦ready，就会渲染输出，而不需要等待所有的cgi拉取到后才开始渲染输出，但是该方案改造成本较大）



四、改造SSR算法

SSR算法改造：就是改造直出算法，不用vue-loader而用自研的loader，将vdom渲染转换为字符串模板，具有更高的渲染性能。

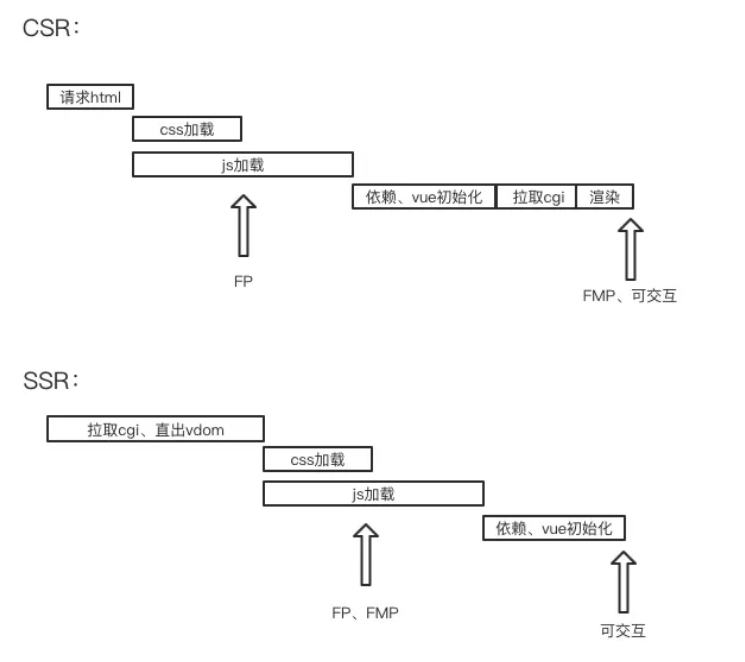
性能提高的同时，由于没有完整的组件运行环境，也带来了部分语法上的约束，同时，也不支持vuex。

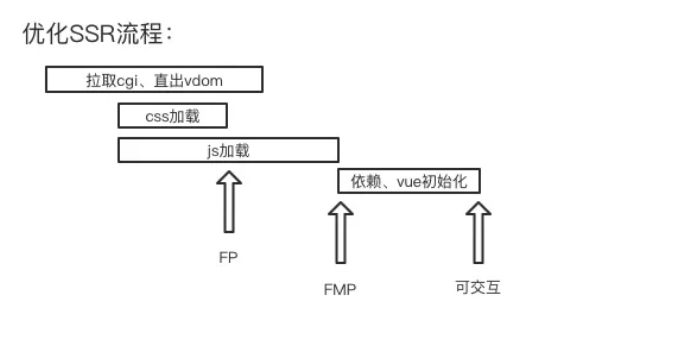
通过上期视频，大家应该对SSR了如指掌且熟悉了常见的优化方法，但是我们再回过头来思考一下，Vue-SSR的优化无非是在 cgi拉取 和 VDOM直出渲染 上下功夫，因为这两者就是node后端最耗时的步骤，其次，由于这种耗时会同步阻塞页面的FP，所以更进一步的方法是流式输出或分块，减少首屏渲染时间。

然而，但是并不是所有的cgi都能缓存，类似拉取用户个人信息的cgi就无法缓存，SSR算法改造成本大，约束也大。再看看流式传输和分块传输，两者虽然都对FP时间优化了，但流式传输受限于cgi拉取时间，分块传输改造成本大。而且两者存在的一个共性问题，那就是可交互时间仍然没有优化。

当然，这里并不是要否定所有的优化方法，而是方法各有优劣，比较优缺点大家才能根据自己的业务需求和优化场景选取合适的优化方法。受流式传输和分块传输的启发，我们能不能在这上面下功夫？在请求到来时，先返回一份完整的HTML空页面，让客户端更快的FP，其次，后端拉取cgi和渲染VDOM 与 前端拉取CSS、JS资源 两者同步进行，之后再吐出直出的HTML string 与 页面store，再次渲染页面，这样的话FP提前了，和CSR的FP时间一毛一样，其次，FMP相比CSR大大提高，更重要的是，由于JS资源的加载让Vue初始化触发的更早，意味着页面可响应时间也会提高。

为了阐明这种区别，我们看一下流程图：





那么就是，先吐空页面，之后再吐直出后的数据，但是关键是怎么让直出后的数据再渲染上去，同时不要让JS先执行了，导致页面直接变成CSR了。

思路是这样的：不要让JS执行，等直出数据回来了再执行，这可咋办，我们最初想实现一个JS加载控制器，不通过script来引入js，而是自己去拉取js代码，eval函数执行，这样js的执行控制权就在自己手上了，但是有几个问题，eval函数解析只是把字符串当js来执行，那错误上报就会出问题，接了sentry错误上报是基于js文件、错误行列来定位的，除此之外，ajax来拉取js代码会不会存在性能问题，和浏览器加载js资源速度上是否存在差异？还有第三方js不能直接ajax拉取，需要设置跨域请求头。那么我们再换一种思路，能不能给每个js文件包裹一层函数，通过setTimeout(fn,0)的方式来延迟调用，但是这又有问题，有多个js文件且文件已经是打包好了的，改了js文件，map映射不就乱了吗？错误上报不就乱了吗？

既然源码在自己手里，为啥不直接在源码上提供一个调用入口，来触发js执行，最后直出的时候吐出<script>window.render()<script>来控制js执行不就可以了吗？

【具体代码在下面】

到这儿，我们思路就分析清楚了，具体实现代码，小伙伴们可以下载文档进行阅览~看到这儿，相信你对Vue-SSR有了更加深刻的认识和了解，这两期视频比较了CSR和SSR，并总结归纳了Vue-SSR的常见方法，最后又尝试了新的方法，达到了一定程度上的优化。优化方案各有优劣，也有成本开销，根据自己业务需求来选择合适的优化方法，才是最有效的。希望本视频能给你带来帮助~

### 开始改造

**客户端改造：**

原有的直出存在entry-client和entry-server两个js文件，分开两个入口各自打包，我们需要改造的是entry-client，让其可控制，开头笔者只是对new App()和mount 包裹了一层函数，但是后来发现，第三方js依赖执行了，其实如果你明白webpack的打包原理，那么require的时候就会触发相应的依赖执行，我们要在entry-client之外再包一层来控制。新增entry-runner文件：

window.\_\_GLOBAL\_RENDER\_\_ = function() {  
 require("./entry-client.js");  
 console.log("\_\_GLOBAL\_RENDER\_\_ ENDING"); // eslint-disable-line  
};

以这个为打包入口，加载完js后就不会执行了（当然，还会执行webpack的运行时代码，控制chunk执行的，忽略不计）

改造服务端直出代码：

module.exports = function handle(req,res){  
 res.writeHead(200, {  
 'Content-Type': 'text/html',  
 });  
 res.write(FP\_html.replace(/<\/body>[\s\S]\*<\/html>/g,''));  
 // ...other code  
}

FP\_html是客户端打包的时候生成的index.html，里面已经插入好了css、js依赖，你只需要把尾部body和html的结束标签去掉。

接下来是在直出后吐出直出数据。

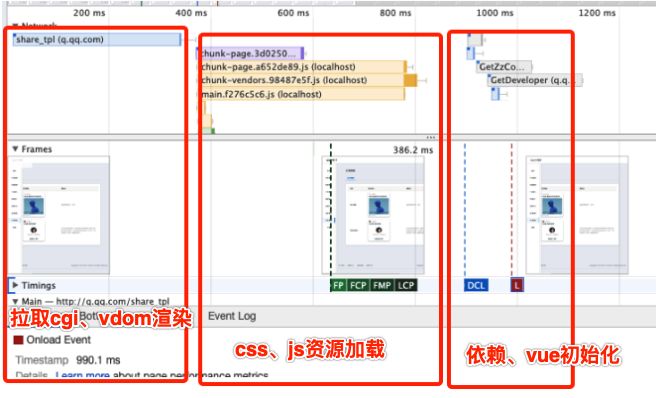
const context = {  
 url: req.REQUEST.pathname  
};  
const html = await renderToString(context);  
const html\_render = html.match(/(<div data-server-rendered[\s\S]\*<\/div>)[\s\S]\*(<script>window.\_\_INITIAL\_STATE\_\_[\s\S]\*injected -->)/g);  
const innerHTML = RegExp.$1;  
const state = RegExp.$2;  
res.write(`  
 <script>  
 document.body.innerHTML = \`${innerHTML}\`  
 </script>  
 ${state}  
 <script>  
 setTimeout(()=>{  
 window.\_\_GLOBAL\_RENDER\_\_ && window.\_\_GLOBAL\_RENDER\_\_()  
 },0)  
 </script>  
 </body>  
 </html>  
 `);  
res.end();

通过正则提取出渲染结果html以及store，之后write吐会给前端，innerHTML会覆盖第一次吐给前端的页面中的div#app，接下来state即全局的store初始化，最后setTimeout控制window.\_\_GLOBAL\_RENDER\_\_执行即可，因为vue判断是否直出是根据div以及全局的store是否初始化来判断的，所以我们这样做没有问题。其次，为了优先触发一次FMP，我们需要通过setTimeout的方式调用全局渲染方法。

接下来我们来比较一下CSR、SSR以及改造后的效果：

CSR:



SSR：

优化后的SSR：

各项数据对比：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型\指标 | FP | FMP | 可交互时间 |
| CSR | 0.4s | 1.1s | 1.2s |
| SSR | 0.7s | 0.7s | 1s |
| 优化SSR | 0.4s | 0.6-0.7s | 0.8s |

可以明显的看到，优化后的SSR，FP时间跟CSR一样，让我们的首屏渲染更快了（可优先渲染页面骨架图），其次，FMP时间跟SSR相差不大，最后是可交互时间，由于JS依赖较早开始加载，所以页面直出结束后可马上执行vue初始化逻辑，所以可交互时间缩短到0.8s。

我们找到了一种成本不是很高，不仅优化了FP、FMP时间还优化了可交互时间的方法！

提问：这种优化方案是否会影响SEO？