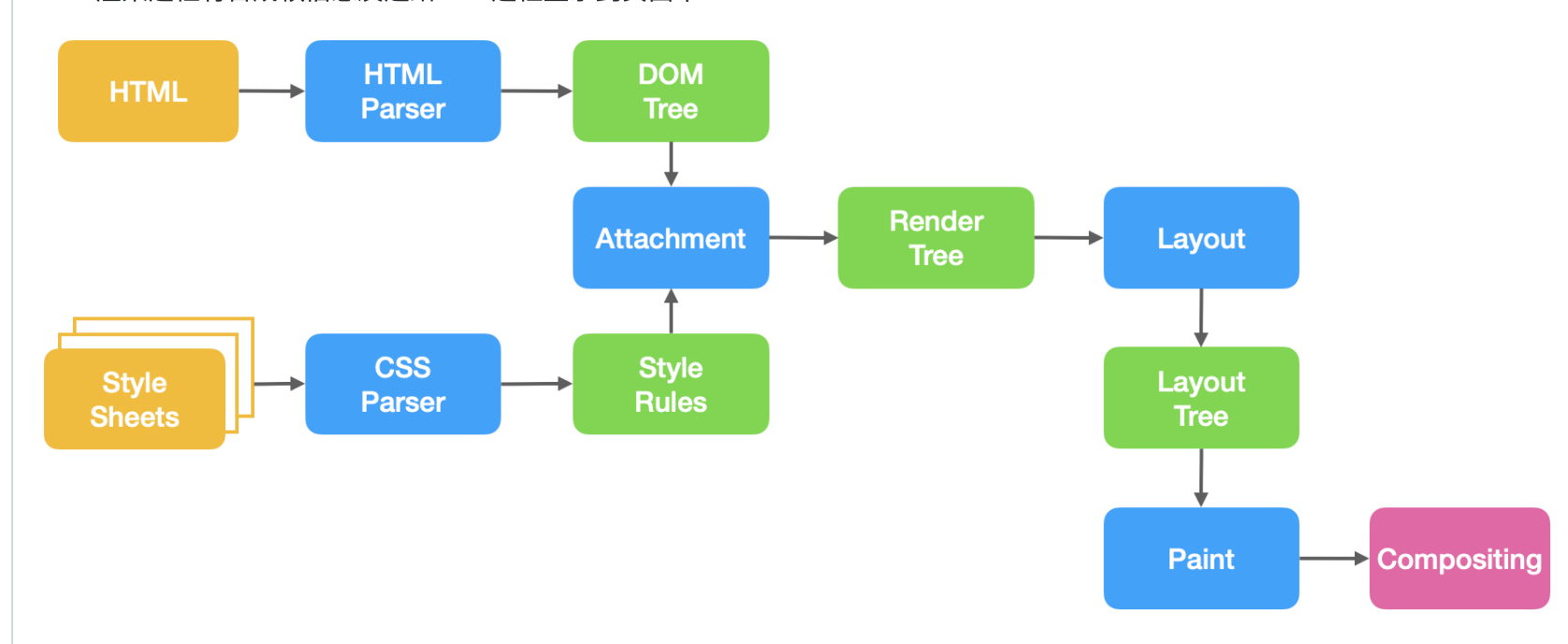
landing-page

1、cdn原理

3、router

# **csr和ssr两者的渲染流程**

1. 浏览器请求得到html文本
2. 渲染进程解析html构建dom树，解析遇到内敛css或者样式脚本，则下载并构建样式规则，遇到js脚本也会下载并执行脚本
3. 当样式规则和dom树都构建完成以后，结合成渲染树
4. 渲染进程对渲染树进行布局处理，生成布局树
5. 渲染引擎将布局树绘制合成帧，最后通过gpu进程展示到页面上



客户端渲染过程：浏览器请求html，首先请求返回一个空的的html，接着浏览器开始解析html，遇到js文件就下载并执行，以及请求后台数据，js渲染页面，最终生成页面.且对于复杂页面，js文件就更多更大，加载执行的时间就更长，白屏时间也更长

服务端渲染：浏览器请求html时，服务端返回一个组装好的带有数据的html，浏览器不用再去执行额外js，直接进行解析html页面构建dom树就可以了.

# **csr和ssr优缺点：**

1. 客户端：

* 返回的首字节更快
* 降低服务端成本
* 白屏时间更长

1. 服务端

* 白屏时间更短
* 利于搜索引擎检索，保证搜索结果可以排到更高位
* 更多工作从前端移交到服务端，增加服务端负载，原本只需要返回静态资源文件的服务器，现在增加获取数据的io，以及渲染html的cpu占用，如果出现流量激增，很有可能会down机，所以需要做好负载均衡
* 代码复杂度增加了，原本只用在客户端运行的代码，需要做兼容逻辑，保证能够在服务端渲染

# **直出到二次渲染、同构**

### 1、打包

webpack针对客户端和node端将资源打包成两份，分别兼容不同环境，其中webpack针对动态加载的模块会单独打成一个包，index-[nodex/we].js + app.js作为主控打包成main包，可以路由加载对应的对应资源包。

landing-page中各个页面通过动态加载的方式引入的，所以会打成一个个单独chunk，之所以没有直接使用常规的动态引入方式而使用了loadable，是因为webpack4之前服务端渲染不支持react自带的懒加载。

以及node\_modules的内容单独达成一个lib的包

从而生成的资源文件主要包含main、lib、以及每个页面单独的chunk，以及一个写着各chunk映射关系的json文件

### 2、直出和二次渲染

server端首次收到请求，走普通渲染，并且缓存拿到的资源（pagename作为key）。

拿到服务端环境可执行的资源包，组装好带数据的html，放入模板文件

同时拿到客户端可执行的资源包，以标签形式注入模板文件等待被加载执行。

模板文件中把该页面渲染需要的状态和数据等注入全局；

模板文件生成的html开始渲染，先解析构建组装好的html，绘制上屏，此时的dom还没有状态以及事件绑定（直出），接着开始下载执行客户端环境的资源文件，拿到注入放在全局的状态和数据，进行状态处理和事件绑定（二次渲染）。

**如何实现同构直出的？**

**整个项目主要在承接各种活动页面，所以做了路由、分包、同构直出的工作，这些工作中，主要衔接客户端和服务端之间的页面流程**

1. 首先是渲染层面上，node调用react ssr api（renderToString）进行组件渲染并注入到html返回，在浏览器第一次渲染的时候调用ReactDOM.hydrate进行注水，得到一个可交互页面

2.数据层面上，保证数据一致性，react项目我们需要进行状态管理，服务端得到的数据，在返回的时候注入script标签，在浏览器初始化全局store的时候使用该数据，完整前后端渲染数据的共享。

3. 项目使用react router管理多页面，所以需要保证路由一致性，在node端和浏览器端分别使用了staticRouter和broswerRouter进行路由管理。

4. 针对执行环境（node和浏览器）的不同，利用打包工具分别构建对应环境的资源bundle。针对像window之类的变量和api在node端并没有情况，代码里也针对执行环境不同进行了不同的处理，比如自定义方法判断环境等。

# **图片懒加载**

图片懒加载其实就是图片的链接放在其他属性中，因为放在src就会立马发起请求。然后通过位置计算，判断元素的位置，决定是否需要展示图片，再将图片链接换到src中，此处位置的计算推荐使用intersectionObserver，首先拿到所有的图片标签，用intersectionObserver监听，判断图片与视窗之间的位置关系，当图片在视窗时，触发该方法的回调，在回调中给img的src中放链接。

如果此时出现图片的新增或切换新突破，该如何触发？推荐使用api mutationObserver，监听整个html的属性变动、子节点变动，当监听到这些表动时，把该标签加入监听队列中

# **曝光**

为需要曝光的元素增加classname,比如 report。queryselectall所有带有report的标签，构建一颗曝光树。

旧思路：当页面滚动，遍历曝光树，调用方法boundingclientrect计算元素在视窗的位置，如果在视窗内，则进行曝光上报，并标记为已曝光

新思路：调用intersectionObserver监听曝光树所有元素可见性的变化，当可见时进行曝光上报，并标记为已曝光

# **渲染异常监控**

由于线上的事故的发现往往依赖于内部反馈或外部反馈，这样导致影响面大，并且响应不够快，过于被动。所以期望在发生事故的时候能快速感知并且做降级处理。但是当前的事故监控存在两个痛点：

一个是异常渲染并不能被完全捕获，而且一旦异常，整个页面就会挂掉。

二是异常catch过于依赖开发人员自驱

首先针对第一个问题，当时调研了一下市面上的一些异常监控的方案，其中包括用window.onerror来捕获异常，由于onerror捕获异常的思路简单来说就是查看最外层节点下是否还有子节点，那这种方式其实并不能覆盖所有的异常场景、并且当某个子节点渲染异常会导致整个页面崩溃。以及监听dom变化的过程，这个也不能够很好的定位异常。包括轮询的方式查看html长度的变化，也不能定位异常。

还有一种方式就是react错误边界，它是react的一种组件，当监听生命周期getDriverStateFromError或者ComponentDidCatched中的一个，子组件下的任何位置的渲染异常都会被捕获到。其中getDriverStateFromError主要是用来捕获异常并返回降级ui，componentDidCatched主要用来打印异常。

但是我们不可能重复为每个组件加上这个两个生命周期并作相应的异常处理，那么对于这种横切关注点的问题就自定义了高阶组件来做这个异常捕获的事情，这样只用给需要的组件外面包一层，就可以进行异常监控。但是当时我们的业务组件已经非常多了，大约有500+的组件，如果每个都去手动包高阶组件，也是一个很繁琐的事情

所以当时针对这种情况，就自定义了babel插件，在babel对已经解析的抽象语法树做转换的阶段，对符合要求的组件包上错误边界高阶组件。这样只用开发人员在babel的配置文件中配置这个插件就能自动为组件加上错误边界高阶组件。

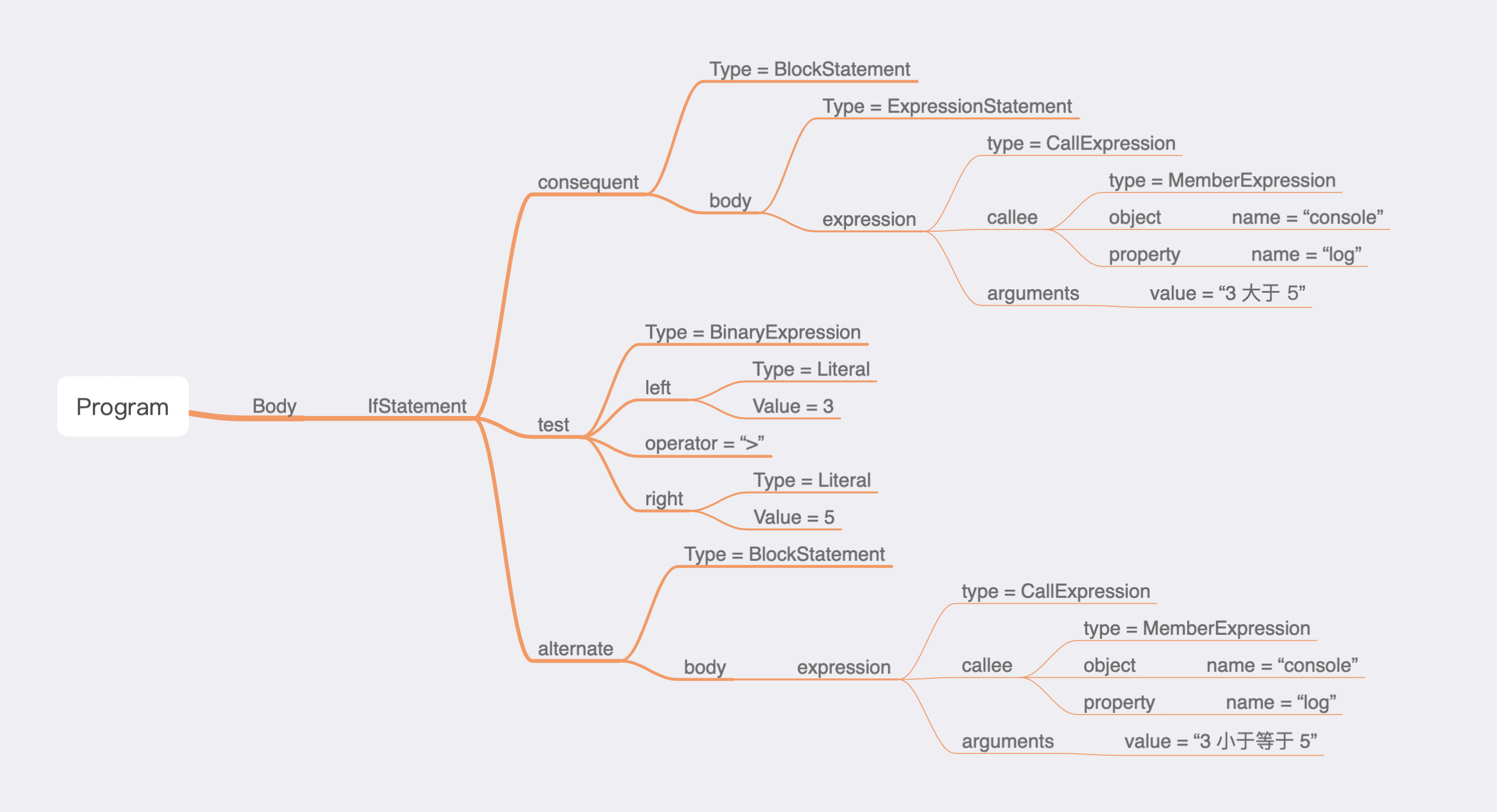
### 1、babel（js编译器）的工作过程：

1. 解析【@babel/parse】：接收代码并转换成ast

* 解析词法解析 （代码转换成令牌流：一个扁平的语法数组）
* 语法解析（令牌流转换成ast）

1. 转换【@babel/traverse @babel/types】：遍历ast，对树进行增删改查，是plugin作用的阶段
2. 生成【@babel/genertor】：进行深度遍历，将ast转换成字符串形式的代码，并且生成sourcemap

ps：ast就是一个对象，定义了获取树状结构中节点的方法



### 2、错误边界

1）定义：错误边界是一种react组件，用于捕获子组件树任何位置的渲染错误，并做组件的降级处理以及打印这些错误。

2）比较：相较window.onerror，好处有两个

* 覆盖场景更多：因为onerror的原理是判断最外层节点下是否有子节点，但是有些场景下存在多个外层节点，这种情况就不太适用
* 能将影响范围缩小：当某个组件挂了，onerror无法缩小组件的影响范围，会导致整个页面崩溃，但是错误边界可以将范围缩小到这一个组件

3）应用：在类组件中定义生命周期getDriverStateFromError或componentDidCatch就可以捕获错误，其中getDriverStateFromError主要用于渲染一个备用组件，componentDidCatch用于打印错误信息；

### 3、高阶组件

定义：组件是将props转ui，高阶组件是将一个组件转换为另一个组件。主要解决横切关注点的问题

mixin：mixin也是用来解决横切关注点的问题，mixin就是用赋值的方式将mixin对象中的方法都挂载到目标对象上，来实现混入。存在的问题

* 复杂度滚雪球，多个mixin混入，导致越来越难维护
* 导致命名冲突，不同mixin中的命名不可知，非常可能冲突
* 破坏原有的封装，带来一些新的状态需要维护

# **性能优化**

其中一个增长页承接的业务场景是浏览器热搜榜，当时热榜的性能不佳，导致存在比较大的用户漏斗，从用户点击热榜到热榜可交互的过程中，用户漏斗达到了26%左右。首屏耗时高达1600多毫秒

针对这个问题，首先第一期的常规手段的优化，比如基于路由的代码拆分和按需加载，tree shaking，关键的css内联，cdn缓存，懒加载等。以及node端的同构直出

那第一期优化下来，首屏耗时减少了300多毫秒，可交互时间减少了1100ms，用户漏斗直接降了16%多。

那之后就对落地页各个环节的耗时进行了打点分析，发现从接收首字节到真正开始渲染，中间有600ms的gap。正常情况下，浏览器响应首字节以后就会开始解析资源。所以二期优化首先就要排查这个未知的高耗时问题。

当时进行问题排查的时候，第一个考虑到的是定制webview的问题， 因为热搜榜场景的特殊性，当时热榜场景的webview和普通h5的webview有所不同，后来测试发现，定制webview对于这种未知高耗时的复现频率确实略高。同时由于热榜的页面入口在浏览器首页，经过验证，浏览器首页的复现频率确实比普通入口更高

所以当是基于这两个变量就做了一个小流量的abtest，分别是页面是否由定制webview承接以及页面入口是否在浏览器首页，一共四种场景，当时针对这四种场景进行了渠道投放，最后测试下来的数据发现，未知高耗时和定制webview没有必然关系，主要取决于入口是否来自于qb首页。

之后再通过chrome tarcing工具开始调试发现，从发起请求到开始首字渲染，一共经历了四次跨进程的通信，其中高耗时的两处都与浏览器进程的mian线程繁忙有关。

第一次是网络进程通知浏览器进程开始接受数据，这里大约存在300ms左右的延迟，浏览器进程的mian线程才开始响应处理

第二次是渲染进程的渲染线程去调用mian线程的字节码缓存的时候，也存在300ms左右的等待。

那与端同学沟通下来，发现造成mian线程无法及时处理消息的主要原因有两个：一个是页面在浏览器首页，可能存在打开首页的时候浏览器进程还没有初始化完成。二是qb首页任务比较多，cpu的负载高

所以当是的第一反应是做内核层面的优化，也就是减少线程间的通信，但是内核架构复杂，改造难度太大，不太现实

再者就是考虑做终端层面的优化，因为qb首页任务多，导致线程繁忙，那是否可以减少首页任务呢，但这也是行不通的，因为涉及的业务团队太多，并且还是端版本影响

所以说主要思考的还是能否在前端做一些优化，考虑到线程繁忙的时间不变，那是否可以提前开始线程通信，从而提前首字渲染。那主要做法就是说缩短ttfb首字节响应，或者说尽可能更快的返回资源。

为了更快的返回资源，当时就采用了分块返回和流式渲染的方案。分块就是把服务端生成的html分成几块，比如项目中分成三块，头部、尾部、和中间数据拼接部分。当node端接收到请求的时候，先将不依赖后台数据的chunk从缓存中拿取并马上返回，等后台返回数据以后，再进行流式渲染，返回其余chunk。通过这样的方式，首字渲染成功提速了130ms

通过

1. 路由
2. js在渲染进程下的执行过程

要运行的JavaScript脚本会从网络或缓存中被**加载。** 通过对JavaScript脚本文本的分析可以生成用于描述源代码结构化的数据，**抽象语法树（AST）。** 接下来**Ignition解释器**会将AST转化成生成体积更小的**字节码**，字节码中的每行指令代表着对寄存器的操作，当字节码生后以后AST将会被废弃以节省空间，后续的执行和优化都基于字节码。 在解释器执行字节码时，**Object Shapes**会试图将代码中对象的类型缓存下来生成**Type Feedback**，当访问这些对象时会尝试从缓存中获取，如果找不到再动态查找并更新缓存。 **TurboFan**是V8中的代码优化编译器，它会评估函数是否需要被进一步优化成机器码以提高性能，需要被优化的函数被编译成**Optimized Code**。 但当编译后的函数被发现函数中变量的数据类型与之前缓存的类型不同时，则需要放弃优化的代码回到字节码重新解释执行

<https://zhuanlan.zhihu.com/p/279920830>

1. chrome启动时会开启的进程：<http://t.zoukankan.com/ly1368489670-p-12834431.html>