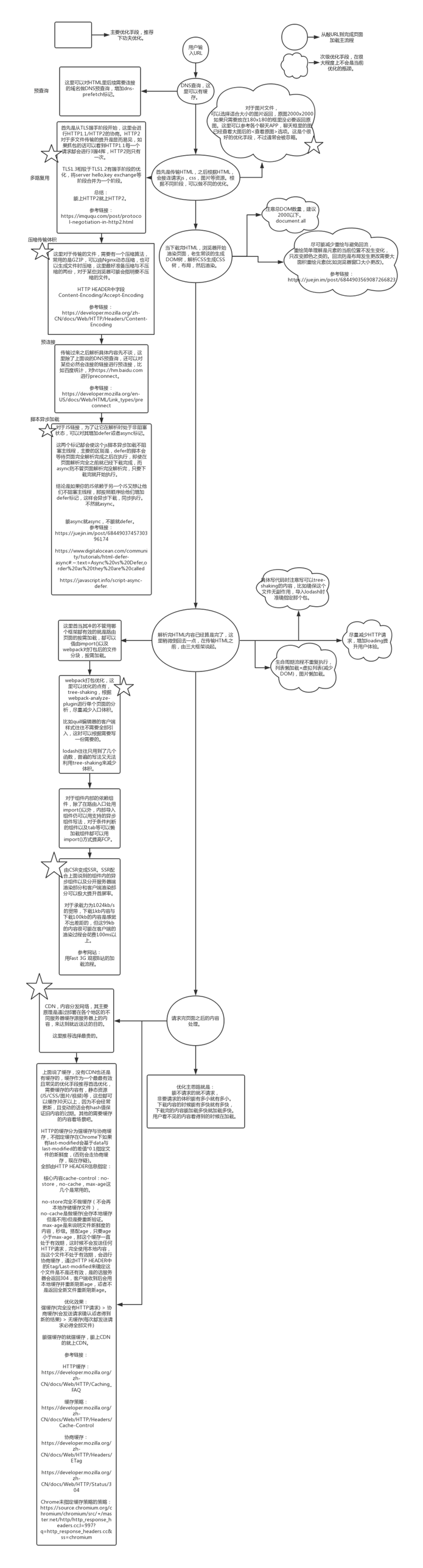
主要优化手段，推荐下边关优化。
用户输入URL
从献URL到完成页面加载主流程
次级优化手段，在很大程度上不会是当前优化的瓶颈。
DNS查询，这里可以有缓存。
预查询
这里可以对HTML里后续需要连接的域名做DNS预查询，增加dns-prefetch标记。
对于图片文件，可以选择适合大小的图片返回，原图2000x2000如果只需要放在180x180的框里没必要返回原图。这里可以参考各个智天APP，都天相当的现已经查看大图后的＜查看原图＞选项。这是个很好的优化手段，不过逼哥会被忽略。
多路复用
首先是从TLS端手的段开始，这里会进行HTTP1.1/HTTP2的协商，HTTP2对于多文件传输的提升是显而易见，如果抓包的话可以看到HTTP1.1每一个请求都会进行3端4挥，HTTP2则只有一次。
TLS1.3相较于TLS1.2有端手的段的优化，将server hello,key exchange等阶段合并为一个阶段。
总结：
能上HTTP2就上HTTP2。
参考链接：
https://imququ.com/post/protocol-negotiation-in-http2.html
首先是传输HTML，之后根据HTML，会触发请求js，css，图片等资源。根据不同阶段，可以做不同的优化。
注意总DOM数量，建议2000以下。document.all
压缩传输体积
这里对于传输的文件，需要有一个压缩算法，常用的是GZIP，可以由Nginx动态压缩，也可以生成文件时压缩，这里最好准备压缩与不压缩的两份，对于某些浏览器可能会报明要不压缩的文件。
HTTP HEADER中字段
Content-Encoding/Accept-Encoding
参考链接：
https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Encoding
当下载完HTML，浏览器开始渲染页面，也生常说的生成DOM树，解析CSS生成CSS树，布局，然后渲染。
尽可能减少重绘与避免回流，重绘简单理解是元素的当前位置不发生变化，只改变颜色之类的。回流则是布局发生更改需要大面积重绘元素(比如浏览器窗口大小更改)。
参考链接：
https://juejin.im/post/6844903569087266823
预连接
传输过来之后解析具体内容先不谈，这里除了上面说的DNS预查询，还可以对某些必然会连接的链接进行预连接，比如百度统计，对https://hm.baidu.com进行preconnect。
参考链接：
https://developer.mozilla.org/en-US/docs/Web/HTML/Link_types/preconnect
脚本异步加载
对于JS链接，为了让它在解析时处于非阻塞状态，可以对其增加defer或者async标记。
这两个标记都会使这个js脚本异步加载不阻塞主线程，主要的区别是，defer的脚本会等待页面完全解析完成之后在执行，即使在页面解析完全之前就已经下载完成，而async则不管页面解析完没解析完，只要下载完就开始执行。
结论是如果你的JS依赖于另一个JS又想让他们不阻塞主线程，那按照顺序给他们增加defer标记，这样会异步下载，同步执行。
不然就async。
能async就async，不能就defer。
参考链接：
https://juejin.im/post/6844903745730396174
https://www.digitalocean.com/community/tutorials/html-defer-async#:~:text=Async%20vs%20Defer,order%20as%20they%20are%20called
https://javascript.info/script-async-defer
具体写代码时注意写可以tree-shaking的内容，比如清楚这个文件无副作用，导入lodash时准确指定那个包。
解析完HTML内容已经算是完了，这里稍微回去一点，在传输HTML之前，由三大框架说起。
这里首当其冲的不管用哪个框架都有效的就是路由页面的按需加载，都可以借由import()以及webpack对打包后的文件分块，按需加载。
尽量减少HTTP请求，增加loading提升用户体验。
生命周期流程不重复执行，列表懒加载+虚拟列表(减少DOM)，图片懒加载。
webpack打包优化，这里可以优化的点有，tree-shaking，根据webpack-analyze-plugin进行单个页面的分析，尽量减少入口体积。
比如quill编辑器的客户端样式往往不需要全部引入，这时可以根据需要写一份需要的。
lodash往往只用到了几个函数，普遍的写法又无法利用tree-shaking来减少体积。
对于组件内部的依赖组件，除了在路由入口处用import()以外，内部导入组件仍可以用支持的异步组件写法，对于条件判断的组件以及tab等可以懒加载组件都可以用import()方式提高FCP。
由CSR变成SSR。SSR配合上面说到的组件内的异步组件以及分开服务器端渲染部分和客户端渲染部分可以极大提升首屏率。
对于承载力为1024kb/s的页面，下载1kb内容与下载100kb的内容是感觉不出差距的，但这99kb的内容很可能在客户端的渲染过程会花费100ms以上。
参考网站：
用Fast 3G 观察B站的加载流程。
请求完页面之后的内容处理。
CDN，内容分发网络，其主要原理是通过部署在各个地区的不同服务器缓存源服务器上的内容，来达到就近连达的目的。
这里推荐选择最贵的。
优化主思路就是：
能不请求的就不请求，
非要请求的体积能有多小就有多小。
下载内容的时候能有多快就有多快，
下载完的内容能加载多快就加载多快。
用户看不见的内容看得到的时候在加载。
上面说了缓存，没有CDN也还是有缓存的，缓存作为一个最最有效且常见的优化手段推荐首选优化，需要缓存的内容有，静态资源(JS/CSS/图片/视频)等，这些都可以缓存30天以上，因为不会经常更新，且实动的话会带hash值保证旧内容的访问。其他的需要缓存的内容看场景吧。
HTTP的缓存分为强缓存与协商缓存，不指定缓存在Chrome下如果有last-modified会基于data与last-modified的差值*0.1指定文件的新鲜度，(否则会走协商缓存，现在存疑)。
全部由HTTP HEADER信息指定：
核心内容cache-control：no-store，no-cache，max-age这几个是常用的。
no-store完全不做缓存(不会再本地存储缓存文件)，no-cache是做缓存(会存本地缓存但是不用)但是要重新验证。max-age是来说明文件新鲜度的内容，秒级。搭配age，只要age小于max-age，那这个缓存一直处于有效期，这时候不会发送任何HTTP请求，完全使用本地内容，当这个文件不处于有效期，会进行协商缓存，通过HTTP HEADER中的Etag/Last-modified来确定这个文件是不是还有效，是的话服务器会返回304，客户端收到后会用本地缓存并重新刷新age，或者不是返回全新文件重新刷新age。
优化效果：
强缓存(完全没有HTTP请求) > 协商缓存(会发送请求确认或者得到新的结果) > 无缓存(每次都发送请求必得全部文件)
能强缓存的就强缓存，能上CDN的就上CDN。
参考链接：
HTTP缓存：
https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Caching_FAQ
缓存策略：
https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Cache-Control
协商缓存：
https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/ETag
https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Status/304
Chrome未指定缓存策略的策略：
https://source.chromium.org/chromium/chromium/src/+/master:net/http/http_response_headers.cc;l=977?q=http_response_headers.cc&ss=chromium