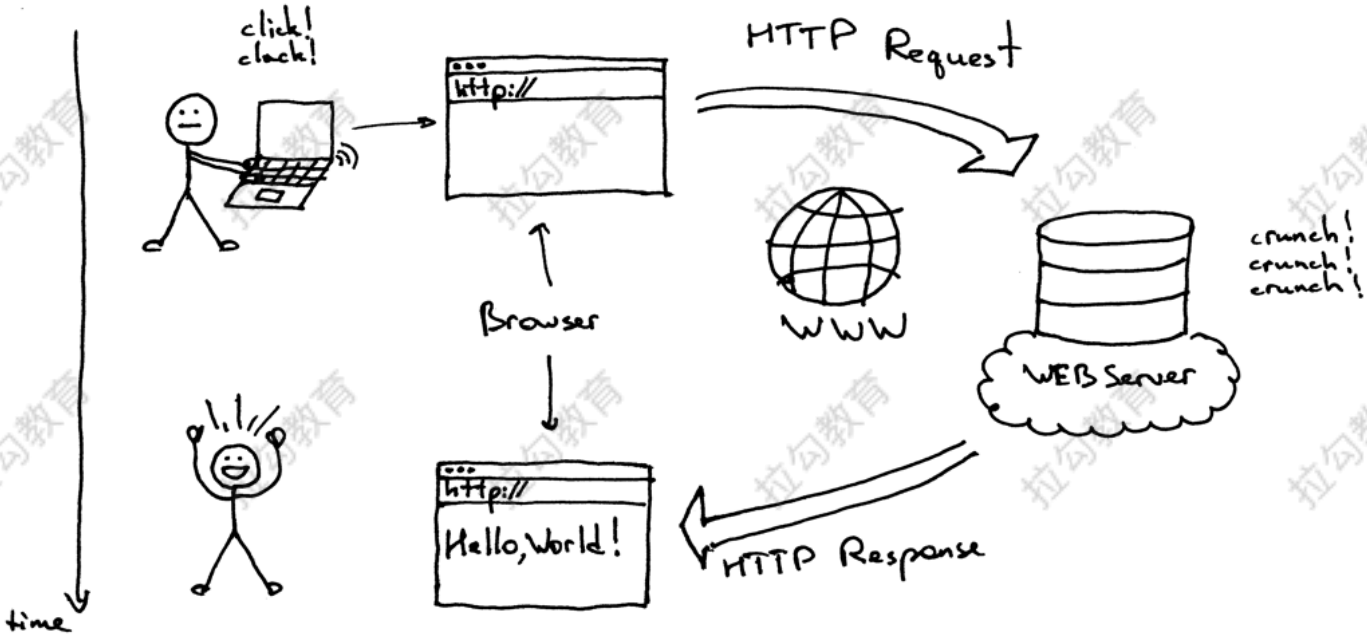
前端页面的生命周期

性能问题呈现给用户的感受往往是简单而直接的：加载资源缓慢、运行过程卡顿或响应交互迟缓等，当把这些问题呈现到前端工程师面前时，却是另一种系统级别复杂的图景。

从域名解析、TCP 建立连接到 HTTP 的请求与响应，以及从资源请求、文件解析到关键渲染路径等，每一个环节都有可能因为设计不当、考虑不周、运行出错而产生性能不佳的体验。作为前端工程师，为了能在遇到性能问题时快速而准确地定位问题所在，并设计可行的优化方案，熟悉前端页面的生命周期是一堂必修课。

本章就从一道常见的前端面试题开始，通过对此问题的解答，来分析前端页面生命周期的各个环节，并着重分析其中关键渲染路径的具体过程和优化实践，希望以此为基础帮读者建构一套完整知识框架的图谱，而后续章节的专题性优化，也都是对此生命周期中某个局部过程的优化分析。

一道前端面试题



大家在进行前端面试时，经常问这样一个问题：**从浏览器地址栏输入URL后，到页面渲染出来，整个过程都发生了什么？**这个问题不仅能很好地分辨出面试候选人对前端知识的掌握程度，能够考查其知识体系的完整性，更重要的是，能够考查面试者在前端性能优化方面理解和掌握此过程的深入程度，与快速定位性能瓶颈及高效权衡出恰当的性能优化解决方案是正相关的。

根据笔者面试和工作的经验，笔者将工程师的能力由低到高划分了若干等级：不堪一击、初窥门径、略有小成、驾轻就熟、融会贯通……

如果面试者的回答是：首先浏览器发起请求，然后服务器返回数据，最后脚本执行和页面渲染，那么这种程度大概在不堪一击与初窥门径之间，属于刚入门前端，对性能优化还没什么概念。

如果知道在浏览器输入 URL 后会建立 TCP 连接，并在此之上有 HTTP 的请求与响应，在浏览器接收到数据后，了解 HTML 与 CSS 文件如何构成渲染树，以及 JavaScript 引擎解析和执行的基本流程，这种程度基本算是初窥门径，在面对网站较差的性能表现时，能够尝试从网络连接、关键渲染路径及 JS 执行

过程等角度去分析和找寻可能存在的问题。本课程的目标便是带领各位同学从初窥门径的程度向更高的级别提升。

其实这个问题的回答可以非常细致，能从信号与系统、计算机原理、操作系统聊到网络通信、浏览器内核，再到 DNS 解析、负载均衡、页面渲染等，但这门课程主要关注前端方面的相关内容，为了后文表述更清楚，这里首先将整个过程划分为以下几个阶段。

（1）浏览器接收到 URL，到网络请求线程的开启。
（2）一个完整的 HTTP 请求并的发出。
（3）服务器接收到请求并转到具体的处理后台。
（4）前后台之间的 HTTP 交互和涉及的缓存机制。
（5）浏览器接收到数据包后的关键渲染路径。
（6）JS 引擎的解析过程。

本章接下来的部分将对以上各阶段进行介绍，由于其中涉及一些知识点，笔者认为这些知识点对理解性能问题和实施优化十分重要，需要更多的篇幅才能表述清楚，所以本章仅对其讲明原理，而后续章节将会单独详述，比如发起完整 HTTP 请求阶段的 DNS 域名解析，前后台 HTTP 交互阶段的数据压缩与缓存等。

网络请求线程开启

浏览器接收到我们输入的 URL 到开启网络请求线程，这个阶段是在浏览器内部完成的，需要先来了解这里面涉及的一些概念。

首先是对 URL 的解析，它的各部分的含义如下图所示。

标题	名称	备注
Protocol	协议头	说明浏览器如何处理要打开的文件，常见的有 HTTP、FTP、Telnet 等。
Host	主机域名 / IP 地址	所访问资源在互联网上的地址，主机域名或经过 DNS 解析为 IP 地址。
Port	端口号	请求程序和响应程序之间连接用的标识
Path	目录路径	请求的目录或文件名
Query	查询参数	请求所传递的参数

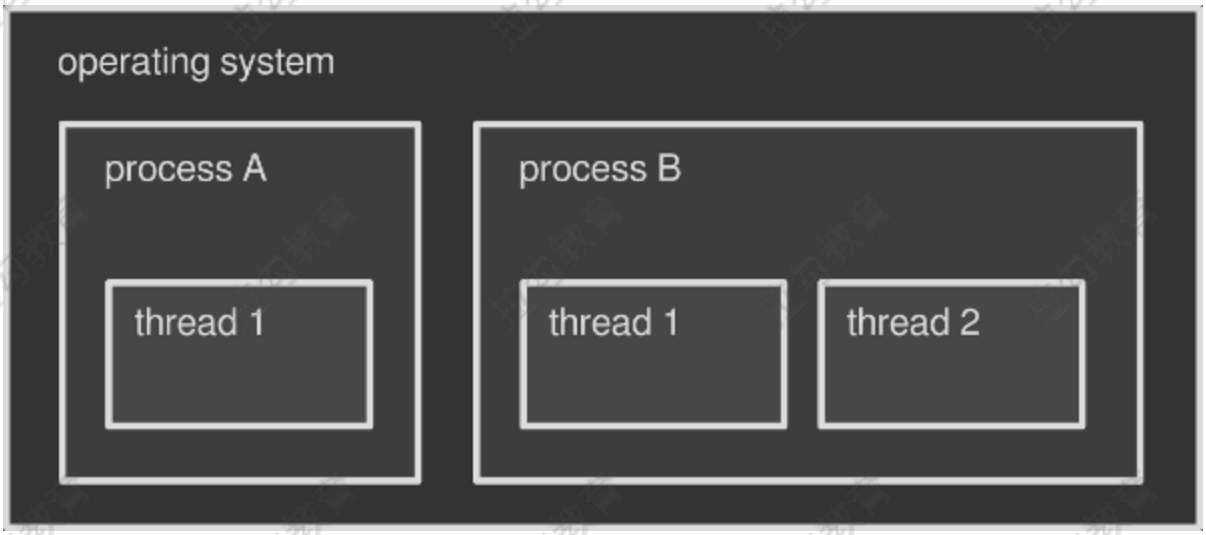
Fragment	片段	次级资源信息，通常作为前端路由或锚点

> URL结构：`Protocol://Host:Port/Path?Query#Fragment`，例如 `http://example.com/users/1?foo=bar#abc`

解析 URL 后，如果是 HTTP 协议，则浏览器会新建一个网络请求线程去下载所需的资源，要明白这个过程需要先了解进程和线程之间的区别，以及目前主流的多进程浏览器结构。

进程与线程

简单来说，进程就是一个程序运行的实例，操作系统会为进程创建独立的内存，用来存放运行所需的代码和数据；而线程是进程的组成部分，每个进程至少有一个主线程及可能的若干子线程，这些线程由所属的进程进行启动和管理。由于多个线程可以共享操作系统为其所属的同一个进程所分配的资源，所以多线程的并行处理能有效提高程序的运行效率。



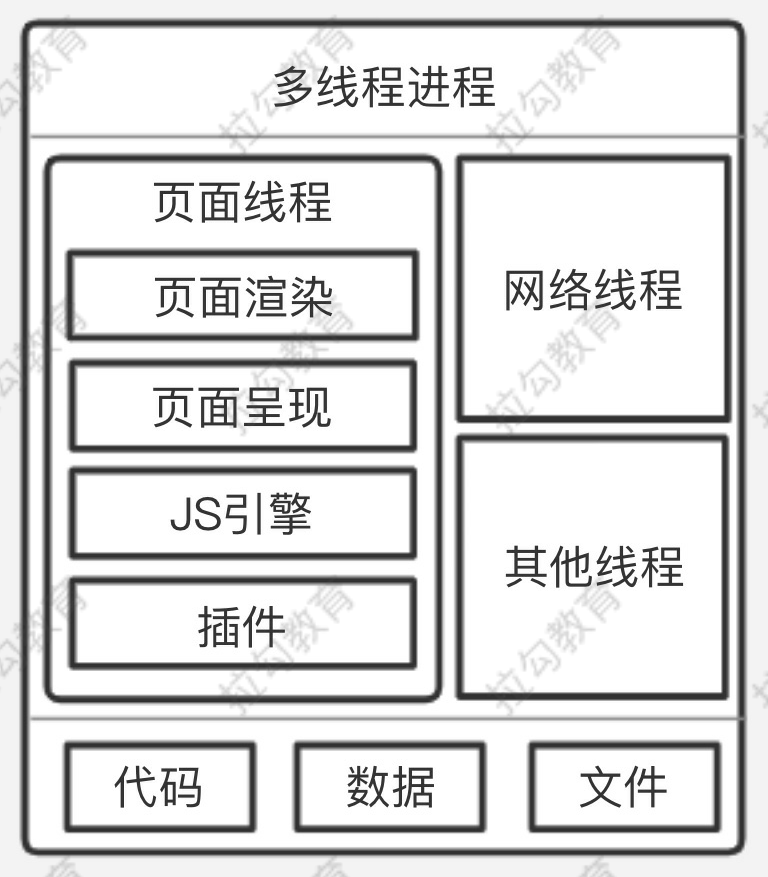
进程和线程之间的区别：

（1）只要某个线程执行出错，将会导致整个进程崩溃。
（2）进程与进程之间相互隔离。这保证了当一个进程挂起或崩溃的情况发生时，并不会影响其他进程的正常运行，虽然每个进程只能访问系统分配给自己的资源，但可以通过 IPC 机制进行进程间通信。
（3）进程所占用的资源会在其关闭后由操作系统回收。即使进程中存在某个线程产生的内存泄漏，当进程退出时，相关的内存资源也会被回收。
（4）线程之间可以共享所属进程的数据。

单进程浏览器

在熟悉了进程和线程之间的区别后，我们在此基础上通过了解浏览器架构模型的演变，来看看网络请求线程的开启处在怎样的位置。

说到底浏览器也只是一个运行在操作系统上的程序，那么它的运行单位就是进程，而早在 2008 年谷歌发布 Chrome 多进程浏览器之前，市面上几乎所有浏览器都是单进程的，它们将所有功能模块都运行在同一个进程中，其架构示意图如下图所示。

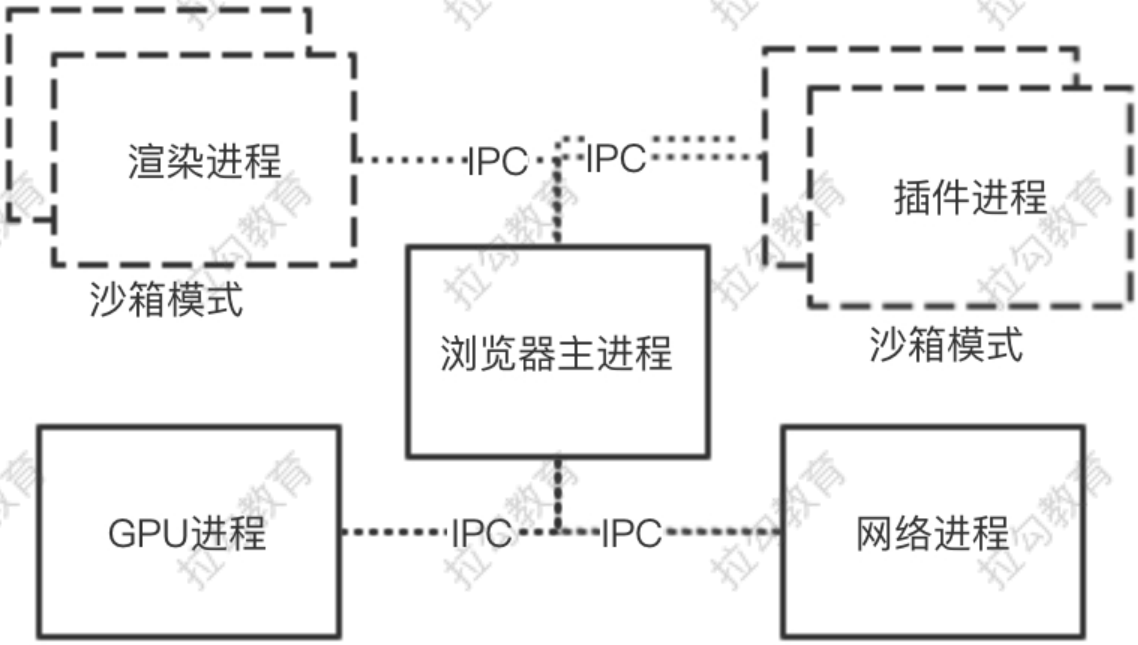


单进程浏览器在以下方面有着较为明显的隐患。

- 流畅性：首先是页面内存泄漏，浏览器内核通常非常复杂，单进程浏览器打开再关闭一个页面的操作，通常会有一些内存不能完全回收，这样随着使用时间延长，占用的内存会越来越多，从而引起浏览器运行变慢；其次由于很多模块运行在同一个线程中，如JS引擎、页面渲染及插件等，那么执行某个循环任务的模块就会阻塞其他模块的任务执行，这样难免会有卡顿的现象发生。
- 安全性：由于插件的存在，不免其中有些恶意脚本会利用浏览器漏洞来获取系统权限，进行引发安全问题的行为。
- 稳定性：由于所有模块都运行在同一个进程中，对于稍复杂的JS代码，如果页面渲染引擎崩溃，就会导致整个浏览器崩溃。同样，各种不稳定的第三方插件，也是导致浏览器崩溃的隐患。

多进程浏览器

出于对单进程浏览器存在问题的优化，Chrome 推出了多进程浏览器架构，浏览器把原先单进程内功能相对独立的模块抽离为单个进程处理对应的任务，主要分为以下几种进程。



（1）浏览器主进程：一个浏览器只有一个主进程，负责如菜单栏、标题栏等界面显示，文件访问，前进后退，以及子进程管理等。

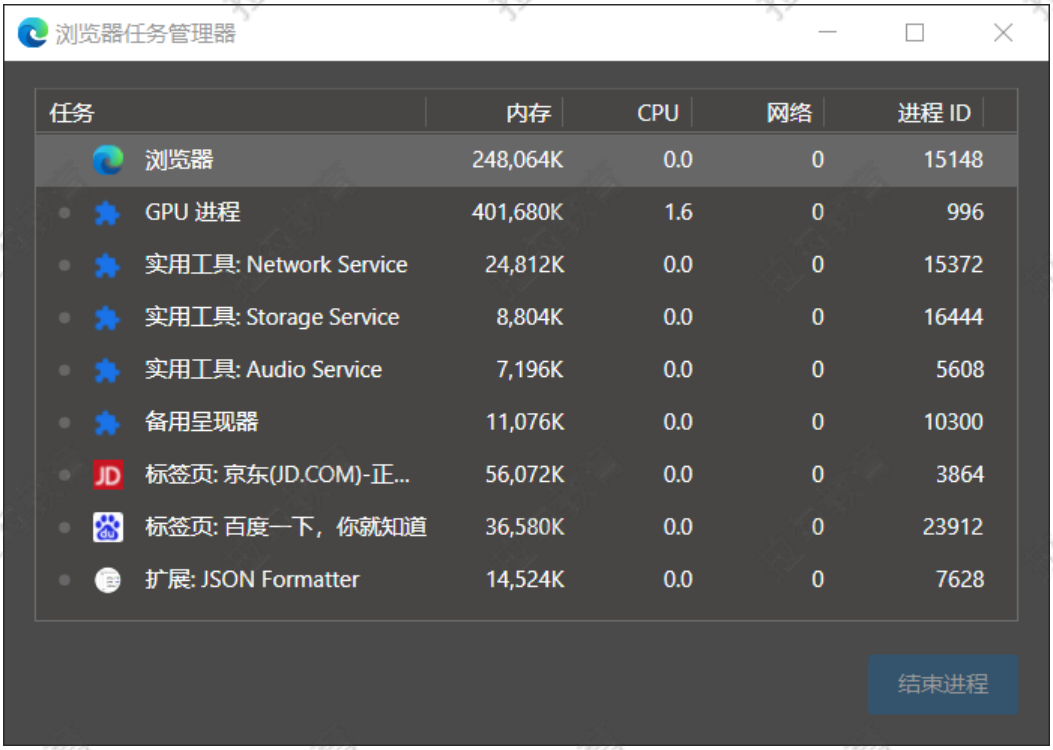
（2）GPU 进程：GPU（图形处理单元）最初是为了实现 3D 的 CSS 效果而引入的，后来随着网页及浏览器在界面中的使用需求越来越普遍，Chrome 便在架构中加入了 GPU 进程。

（3）插件进程：主进程会为每个加入浏览器的插件开辟独立的子进程，由于进程间所分配的运行资源相对独立，所以即便某个插件进程意外崩溃，也不至于对浏览器和页面造成影响。另外，出于对安全因素的考虑，这里采用了沙箱模式（即上图中虚线所标出的进程），在沙箱中运行的程序受到一些限制：不能读取敏感位置的数据，也不能在硬盘上写入数据。这样即使插件运行了恶意脚本，也无法获取系统权限。

（4）网络进程：负责页面的网络资源加载，之前属于浏览器主进程中的一个模块，最近才独立出来。

（5）渲染进程：也称为浏览器内核，其默认会为**每个标签窗口页开辟一个独立的渲染进程**，负责将 HTML、CSS 和 JavaScript 等资源转为可交互的页面，其中包含多个子线程，即 JS 引擎线程、GUI 渲染线程、事件触发线程、定时触发器线程、异步 HTTP 请求线程等。当打开一个标签页输入 URL 后，所发起的网络请求就是从这个进程开始的。另外，出于对安全性的考虑，渲染进程也被放入沙箱中。

打开 Chrome 任务管理器，可以从中查看到当前浏览器都启动了哪些进程，如下图所示。



此时仅打开了两个标签页，除了笔者浏览器添加插件所开辟的进程，还可以看到浏览器进程、GPU 进程、网络进程，以及最近新抽离出来的一个音频服务进程等。

建立 HTTP 请求

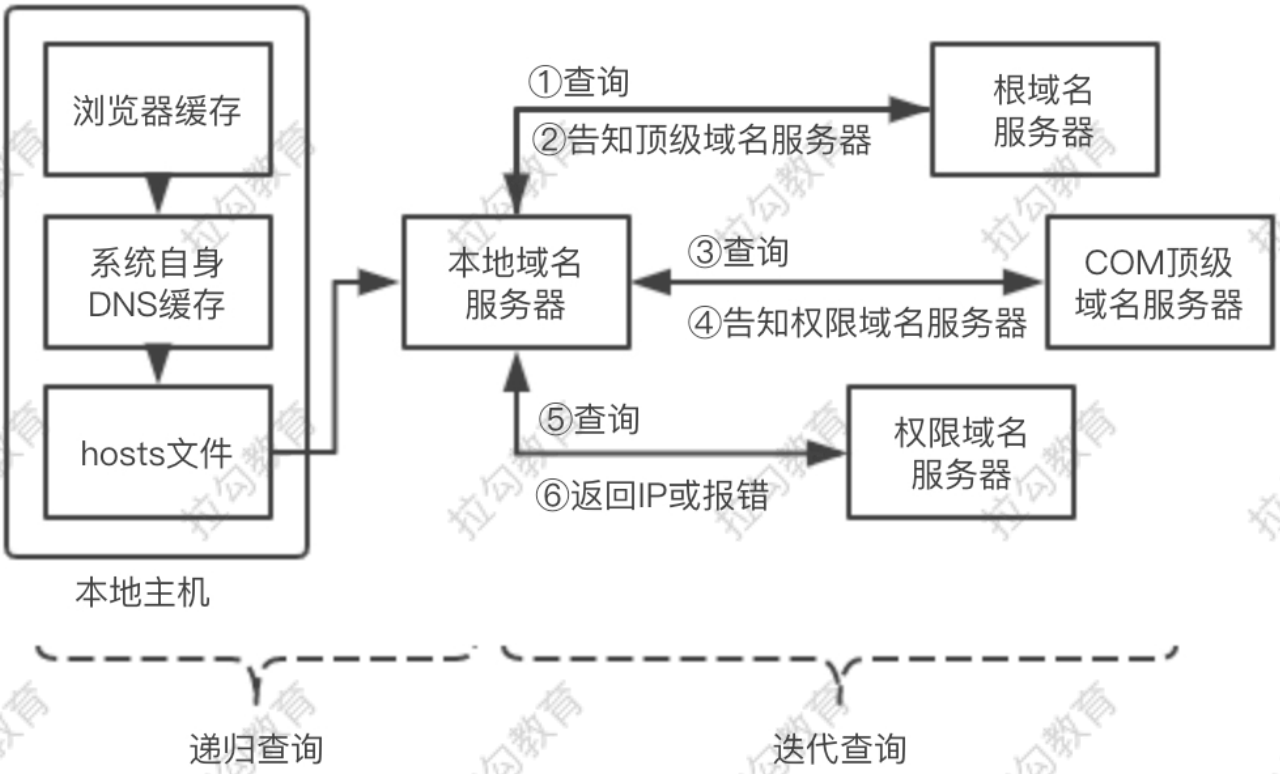
这个阶段的主要工作分为两部分：DNS 解析和通信链路的建立。

简单说就是：

- 首先发起请求的客户端浏览器要明确知道所要访问的服务器地址
- 然后建立通往该服务器地址的路径

DNS 解析

在前面章节讲到的 URL 解析，其实仅将 URL 拆分为代表具体含义的字段，然后以参数的形式传入网络请求线程进行进一步处理，首先第一步便是这里讲到的 DNS 解析。其主要目的便是通过查询将 URL 中的 Host 字段转化为网络中具体的 IP 地址，因为域名只是为了方便帮助记忆的，IP 地址才是所访问服务器在网络中的“门牌号”。



首先查询浏览器自身的 DNS 缓存，如果查到 IP 地址就结束解析，由于缓存时间限制比较大，一般只有1分钟，同时缓存容量也有限制，所以在浏览器缓存中没找到 IP 地址时，就会搜索系统自身的 DNS 缓存；如果还未找到，接着就会尝试从系统的 hosts 文件中查找。

在本地主机进行的查询若都没获取到，接下来便会在本地域名服务器上查询。如果本地域名服务器没有直接的目标 IP 地址可供返回，则本地域名服务器便会采取迭代的方式去依次查询根域名服务器、COM 顶级域名服务器和权限域名服务器等，最终将所要访问的目标服务器 IP 地址返回本地主机，若查询不到，则返回报错信息。

由此可以看出 DNS 解析是个很耗时的过程，若解析的域名过多，势必会延缓首屏的加载时间。本节仅对 DNS 解析过程进行简要的概述，而关于原理及优化方式等更为详细的介绍会在后续课程中单独展开介绍。

网络模型

在通过 DNS 解析获取到目标服务器 IP 地址后，就可以建立网络连接进行资源的请求与响应了。但在此之前，我们需要对网络架构模型有一些基本的认识，在互联网发展初期，为了使网络通信能够更加灵活、稳定及可互操作，国际标准化组织提出了一些网络架构模型：OSI 模型、TCP/IP 模型，二者的网络模型图示如下图所示。

OSI（开放系统互连）模型将网络从底层的物理层到顶层浏览器的应用层一共划分了 7 层，OSI 各层的具体作用如下表所示。

应用层	负责给应用程序提供接口，使其可以使用网络服务，HTTP 协议就位于该层
表示层	负责数据的编码与解码，加密和解密，压缩和解压缩
会话层	负责协调系统之间的通信过程
传输层	负责端到端连接的建立，使报文能在端到端之间进行传输。TCP/UDP 协议位于该层
网络层	为网络设备提供逻辑地址，使位于不同地理位置的主机之间拥有可访问的连接和路径
数据链路层	在不可靠的物理链路上，提供可靠的数据传输服务。包括组帧、物理编址、流量控制、差错控制、接入控制等
物理层	主要功能包括：定义网络的物理拓扑，定义物理设备的标准（如介质传出速率、网线或光钎的接口模型等），定义比特的表示和信号的传输模式

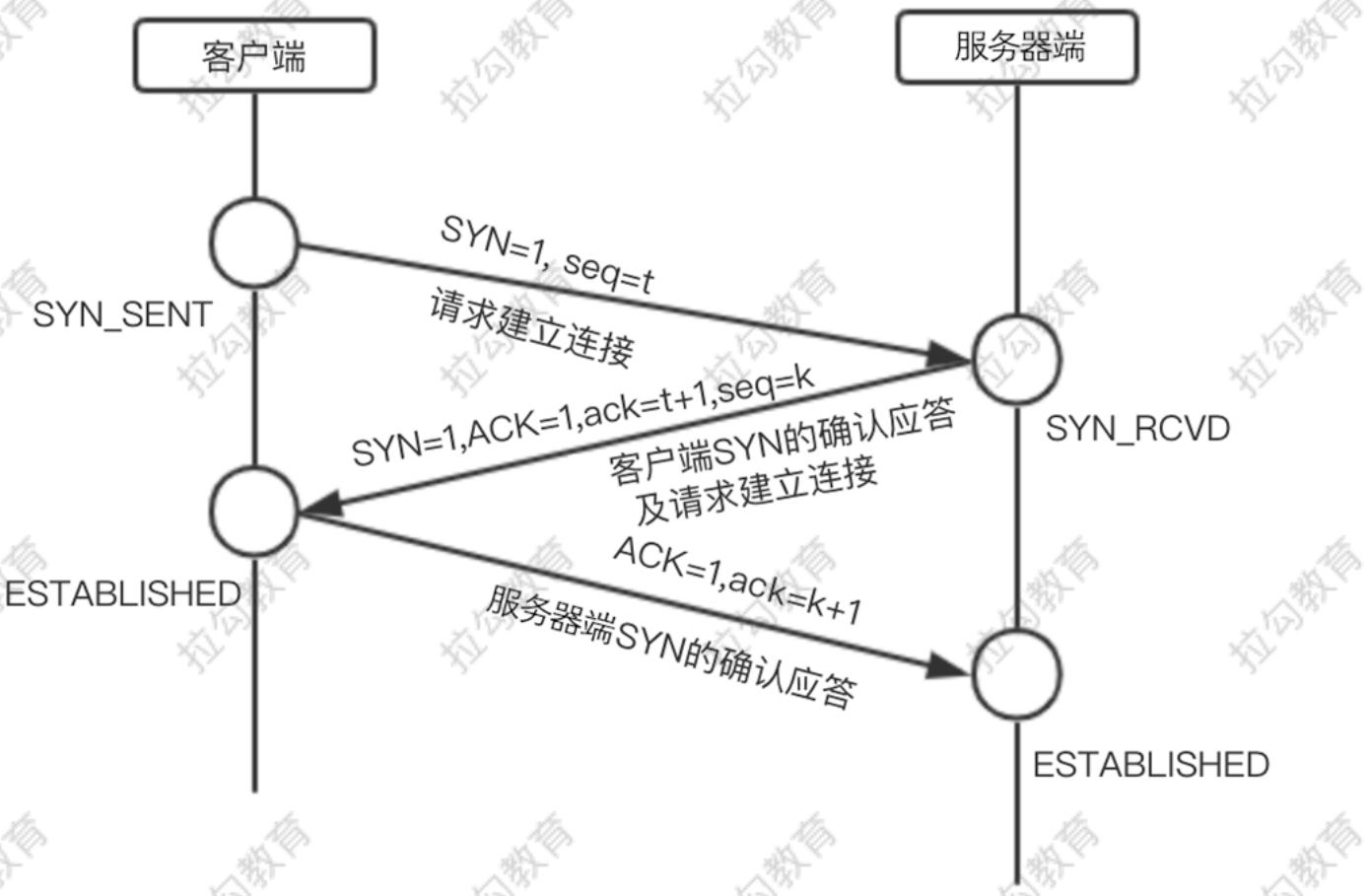
OSI 是一种理论下的模型，它先规划了模型再填入协议，先制定了标准再推行实践，TCP/IP 充分借鉴了 OSI 引入的服务、接口、协议及分层等概念，建立了 TCP/IP 模型并广泛使用，成为目前互联网事实上的标准。

TCP 连接

根据对网络模型的介绍，当使用本地主机连上网线接入互联网后，数据链路层和网络层就已经打通了，而要向目标主机发起 HTTP 请求，就需要通过传输层建立端到端的连接。

传输层常见的协议有 TCP 协议和 UDP 协议，由于本章关注的重点是前端页面的资源请求，这需要面向连接、丢包重发及对数据传输的各种控制，所以接下来仅详细介绍 TCP 协议的“三次握手”和“四次挥手”。

由于 TCP 是面向有连接的通信协议，所以在数据传输之前需要建立好客户端与服务器端之间的连接，即通常所说的“三次握手”，具体过程分为如下步骤。



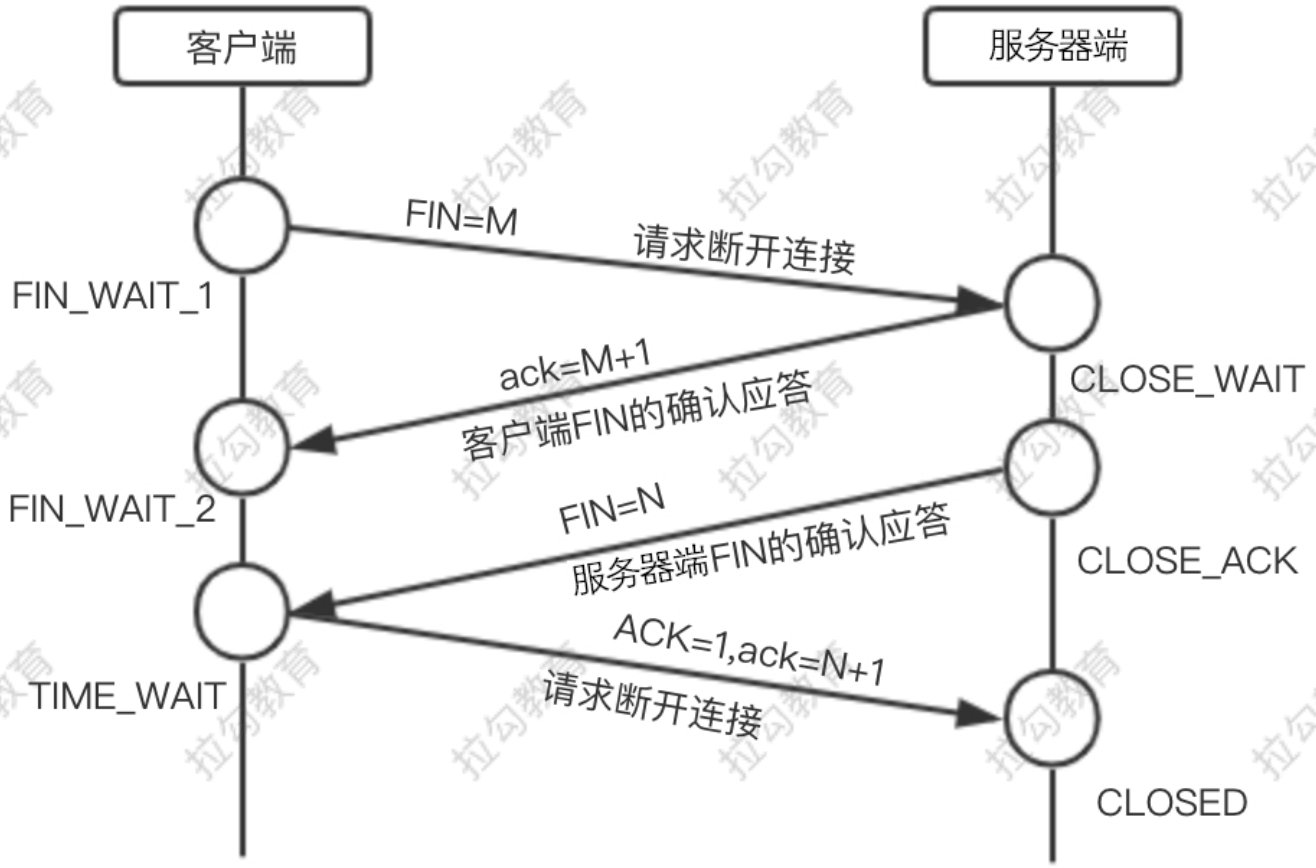
“3次握手”的作用就是双方都能明确自己和对方的收、发能力是正常的。

（1）第1次握手：客户端生成一个随机数 `seq`，假设其值为 `t`，并将标志位 `SYN` 设为 `1`，将这些数据打包发给服务器端后，客户端进入等待服务器端确认的状态。

（2）第2次握手：服务器端收到客户端发来的 `SYN=1` 的数据包后，知道这是在请求建立连接，于是服务器端将 `SYN` 与 `ACK` 均置为`1`，并将请求包中客户端发来的随机数t加 1 后赋值给 `ack`，然后生成一个服务器端的随机数 `seq=k`，完成这些操作后，服务器端将这些数据打包再发回给客户端，作为对客户端建立连接请求的确认应答。

（3）第三次握手：客户端收到服务器端的确认应答后，检查数据包中 `ack` 的字段值是否为 `t+1`，ACK 是否等于 1，若都正确就将服务器端发来的随机数加 `1（ack=k+1）`，将 `ACK=1` 的数据包再发送给服务器端以确认服务器端的应答，服务器端收到应答包后通过检查 `ack` 是否等于 `k+1` 来确认连接是否建立成功。连接建立的关系图如下图所示。

当用户关闭标签页或请求完成后，TCP 连接会进行“四次挥手”断开连接，具体过程如下。



（1） 由客户端先向服务器端发送 `FIN=M` 的指令，随后进入完成等待状态 `FIN_WAIT_1`，表明客户端已经没有再向服务器端发送的数据，但若服务器端此时还有未完成的数据传递，可继续传递数据。

（2） 当服务器端收到客户端的FIN报文后，会先发送 `ack=M+1` 的确认，告知客户端关闭请求已收到，但可能由于服务器端还有未完成的数据传递，所以请客户端继续等待。

（3） 当服务器端确认已完成所有数据传递后，便发送带有 `FIN=N` 的报文给客户端，准备关闭连接。

（4） 客户端收到 `FIN=N` 的报文后可进行关闭操作，但为保证数据正确性，会回传给服务器端一个确认报文 `ack=N+1`，同时服务器端也在等待客户端的最终确认，如果服务器端没有收到报文则会进行重传，只有收到报文后才会真正断开连接。而客户端在发送了确认报文一段时间后，没有收到服务器端任何信息则认为服务器端连接已关闭，也可关闭客户端信息。

只有连接建立成功后才可开始进行数据的传递，由于浏览器对同一域名下并发的 TCP 连接有限制，以及在 1.0 版本的 HTTP 协议中，一个资源的下载需对应一个 TCP 的请求，这样的并发限制又会涉及许多优化方案，我们将在后续章节中进行进一步介绍。

这里较为详细地介绍了 TCP 连接建立和断开的过程，首先让读者有一个网络架构分层的概念，虽然前端工作基本围绕在应用层，但有一个全局的网络视角后，能帮助我们在定位性能瓶颈时更加准确；其次也为了说明影响前端性能体验的因素，不仅是日常编写的代码和使用的资源，网络通信中每个环节的优劣缓急都值得关注。

QA：为什么三次握手建立连接的 TCP 客户端最后还要发送一次确认呢？

> 一句话，主要防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。
>
> 如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请求连接并且没有丢失，只是因为在网络节点中滞留的时间太长了，由于 TCP 的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。
>
> 如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

QA：为什么建立连接是三次握手，关闭连接确是四次挥手呢？

> 建立连接的时候， 服务器在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。
> 而关闭连接时，服务器收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送，从而导致多了一次。

QA：如果已经建立了连接，但是客户端突然出现故障了怎么办？

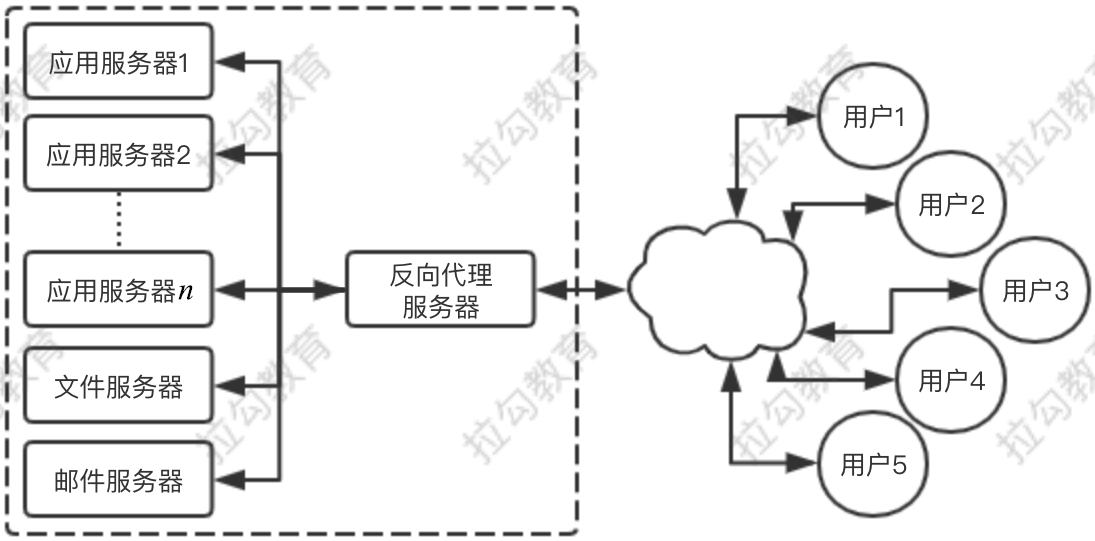
> TCP 还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

前后端的交互

当 TCP 连接建立好之后，便可通过 HTTP 等协议进行前后端的通信，但在实际的网络访问中，并非浏览器与确定 IP 地址的服务器之间直接通信，往往会在中间加入反向代理服务器。

反向代理服务器

对需要提供复杂功能的网站来说，通常单一的服务器资源是很难满足期望的。一般采用的方式是将多个应用服务器组成的集群由反向代理服务器提供给客户端用户使用，这些功能服务器可能具有不同类型，比如文件服务器、邮件服务器及 Web 应用服务器，同时也可能是相同的Web服务部署到多个服务器上，以实现负载均衡的效果，反向代理服务器的作用如图所示。



反向代理服务器根据客户的请求，从后端服务器上获取资源后提供给客户端。反向代理服务器通常的作用如下：

- 负载均衡。
- 安全防火墙。
- 加密及SSL加速。
- 数据压缩。
- 解决跨域。
- 对静态资源缓存。

常用作反向代理服务器的有 Nginx、IIS、Apache，我们会在后面针对 Nginx 深入介绍一些可用于性能优化的配置。

后端处理流程

经反向代理收到请求后，具体的服务器后台处理流程大致如下。

（1）首先会有一层统一的验证环节，如跨域验证、安全校验拦截等。如果发现是不符合规则的请求，则直接返回相应的拒绝报文。
（2）通过验证后才会进入具体的后台程序代码执行阶段，如具体的计算、数据库查询等。
（3）完成计算后，后台会以一个HTTP响应数据包的形式发送回请求的前端，结束本次请求。

HTTP 相关协议特性

HTTP 是建立在传输层 TCP 协议之上的应用层协议，在 TCP 层面上存在长连接和短连接的区别。所谓长连接，就是在客户端与服务器端建立的 TCP 连接上，可以连续发送多个数据包，但需要双方发送心跳检查包来维持这个连接。

短连接就是当客户端需要向服务器端发送请求时，会在网络层 IP 协议之上建立一个 TCP 连接，当请求发送并收到响应后，则断开此连接。根据前面关于 TCP 连接建立过程的描述，我们知道如果这个过程频繁发生，就是个很大的性能耗费，所以从 HTTP 的 1.0 版本开始对于连接的优化一直在进行。

在 HTTP 1.0 时，默认使用短连接，浏览器的每一次 HTTP 操作就会建立一个连接，任务结束则断开连接。

在 HTTP 1.1 时，默认使用长连接，在此情况下，当一个网页的打开操作完成时，其中所建立用于传输 HTTP 的 TCP 连接并不会断开关闭，客户端后续的请求操作便会继续使用这个已经建立的连接。如果我们对浏览器的开发者工具留心，在查看请求头时会发现一行 `Connection: keep-alive`。长连接并非永久保持，它有一个持续时间，可在服务器中进行配置。

而在 HTTP 2.0 到来之前，每一个资源的请求都需要开启一个 TCP 连接，由于 TCP 本身有并发数的限制，这样的结果就是，当请求的资源变多时，速度性能就会明显下降。为此，经常会采用的优化策略包括，将静态资源的请求进行多域名拆分，对于小图标或图片使用雪碧图等。

在 HTTP 2.0 之后，便可以在一个 TCP 连接上请求多个资源，分割成更小的帧请求，其速度性能便会明显上升，所以之前针对 HTTP 1.1 限制的优化方案也就不再需要了。

HTTP 2.0 除了一个连接可请求多个资源这种多路复用的特性，还有如下一些新特性。

（1）二进制分帧：在应用层和传输层之间，新加入了一个二进制分帧层，以实现低延迟和高吞吐量。

（2）服务器端推送：以往是一个请求带来一个响应，现在服务器可以向客户端的一个请求发出多个响应，这样便可以实现服务器端主动向客户端推送的功能。

（3）设置请求优先级：服务器会根据请求所设置的优先级，来决定需要多少资源处理该请求。

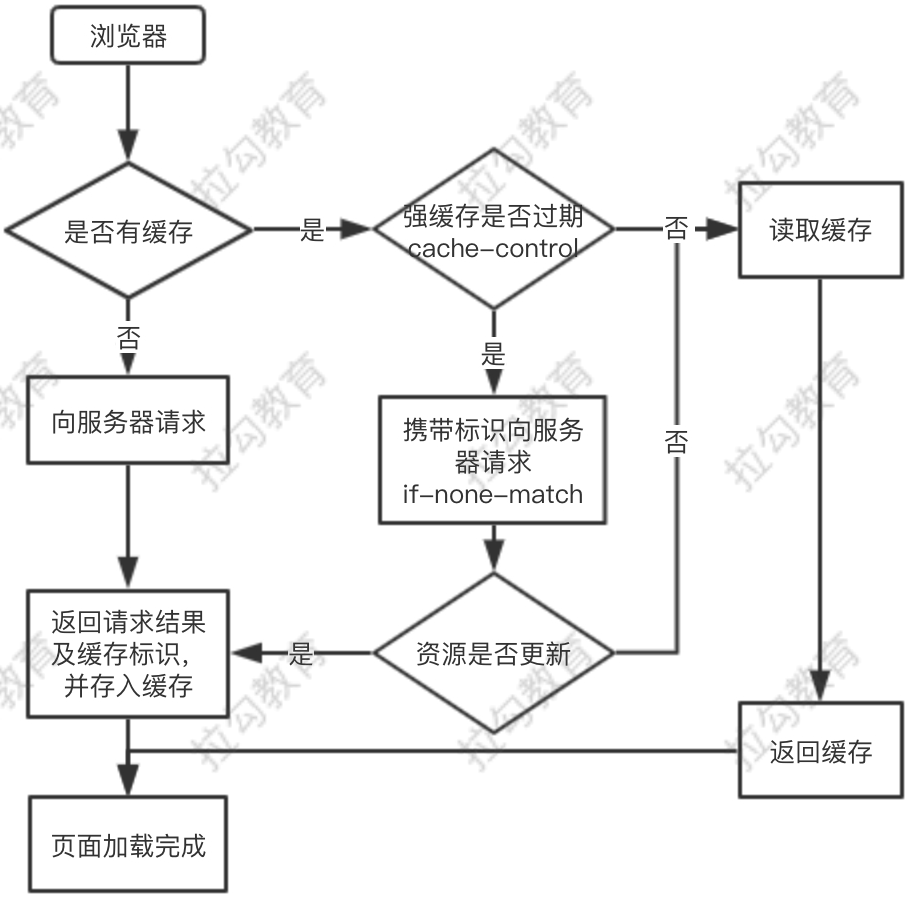
（4）HTTP头部压缩：减少报文传输体积。

浏览器缓存

在基于 HTTP 的前后端交互过程中，使用缓存可以使性能得到显著提升。具体的缓存策略分为两种：强缓存和协商缓存。

强缓存就是当浏览器判断出本地缓存未过期时，直接读取本地缓存，无须发起 HTTP 请求，此时状态为：`200 from cache`。在HTTP 1.1版本后通过头部的 `cache-control` 字段的 `max-age` 属性值规定的过期时长来判断缓存是否过期失效，这比之前使用 `expires` 标识的服务器过期时间更准确而且安全。

协商缓存则需要浏览器向服务器发起 HTTP 请求，来判断浏览器本地缓存的文件是否仍未修改，若未修改则从缓存中读取，此时的状态码为：`304`。具体过程是判断浏览器头部 `if-none-match` 与服务器短的 `e-tag` 是否匹配，来判断所访问的数据是否发生更改。这相比 HTTP 1.0 版本通过 `last-modified` 判断上次文件修改时间来说也更加准确。具体的浏览器缓存触发逻辑如图所示。



在浏览器缓存中，强缓存优于协商缓存，若强缓存生效则直接使用强缓存，若不生效则再进行协商缓存的请求，由服务器来判断是否使用缓存，如果都失效则重新向服务器发起请求获取资源。本节仅简要说明浏览器缓存的触发过程，由于这部分对性能优化来说比较重要，所以在后续章节也会详细介绍。

关键渲染路径（CRP）

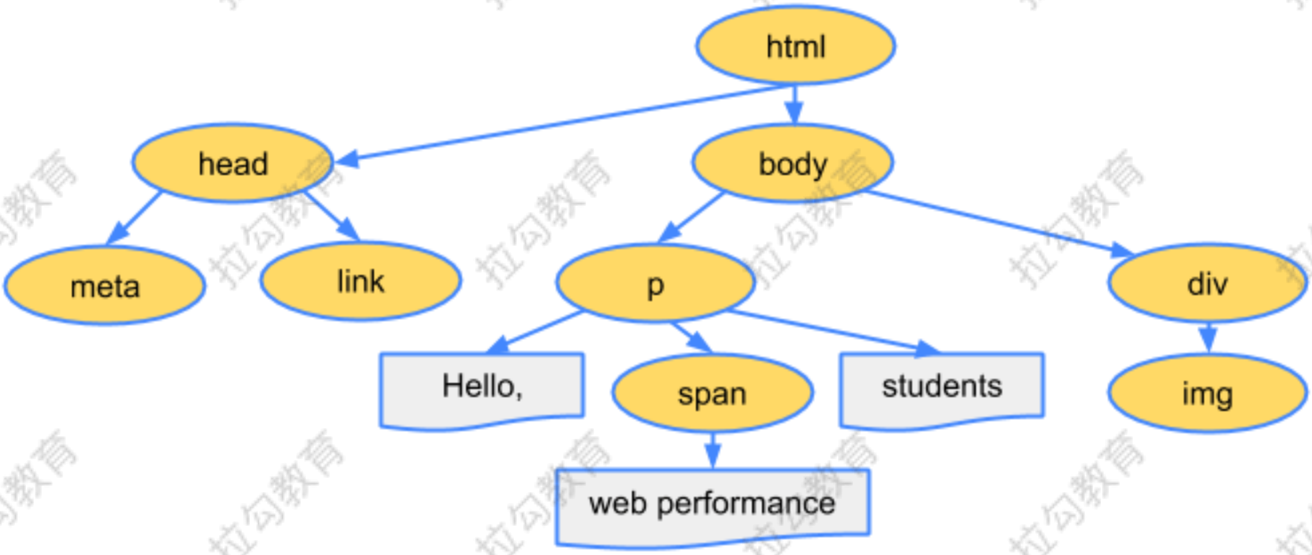
当我们经历了网络请求过程，从服务器获取到了所访问的页面文件后，浏览器如何将这些 HTML、CSS 及 JS 文件组织在一起渲染出来呢？

构建对象模型

首先浏览器会通过解析 HTML 和 CSS 文件，来构建 DOM（文档对象模型）和 CSSOM（层叠样式表对象模型），为便于理解，我们以如下 HTML 内容文件为例，来观察文档对象模型的构建过程。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scal
e=1.0">
  <title>关键路径渲染</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <p>你好<span>性能优化</span></p>
  <div>
    <img src="avatar.jpg">
  </div>
</body>
</html>
```

浏览器接收读取到的 HTML 文件，其实是文件根据指定编码（UTF-8）的原始字节，形如 `3C 62 6F 79 3E 65 6C 6F 2C 20 73 70…`。首先需要将字节转换成字符，即原本的代码字符串，接着再将字符串转化为 W3C 标准规定的令牌结构，所谓令牌就是 HTML 中不同标签代表不同含义的一组规则结构。然后经过词法分析将令牌转化成定义了属性和规则值的对象，最后将这些标签节点根据 HTML 表示的父子关系，连接成树形结构，如下图所示。



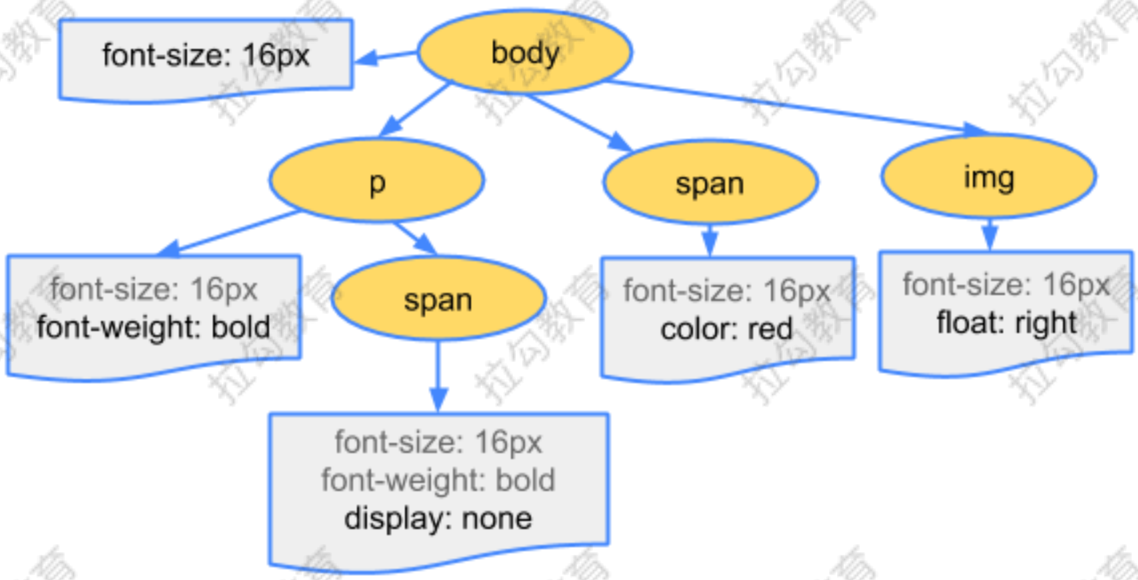
构建 CSSOM

DOM 树表示文档标记的属性和关系，但未包含其中各元素经过渲染后的外观呈现，这便是接下来 CSSOM 的职责了，与将 HTML 文件解析为文档对象模型的过程类似，CSS 文件也会首先经历从字节到字符串，然后令牌化及词法分析后构建为层叠样式表对象模型。假设 CSS 文件内容如下：

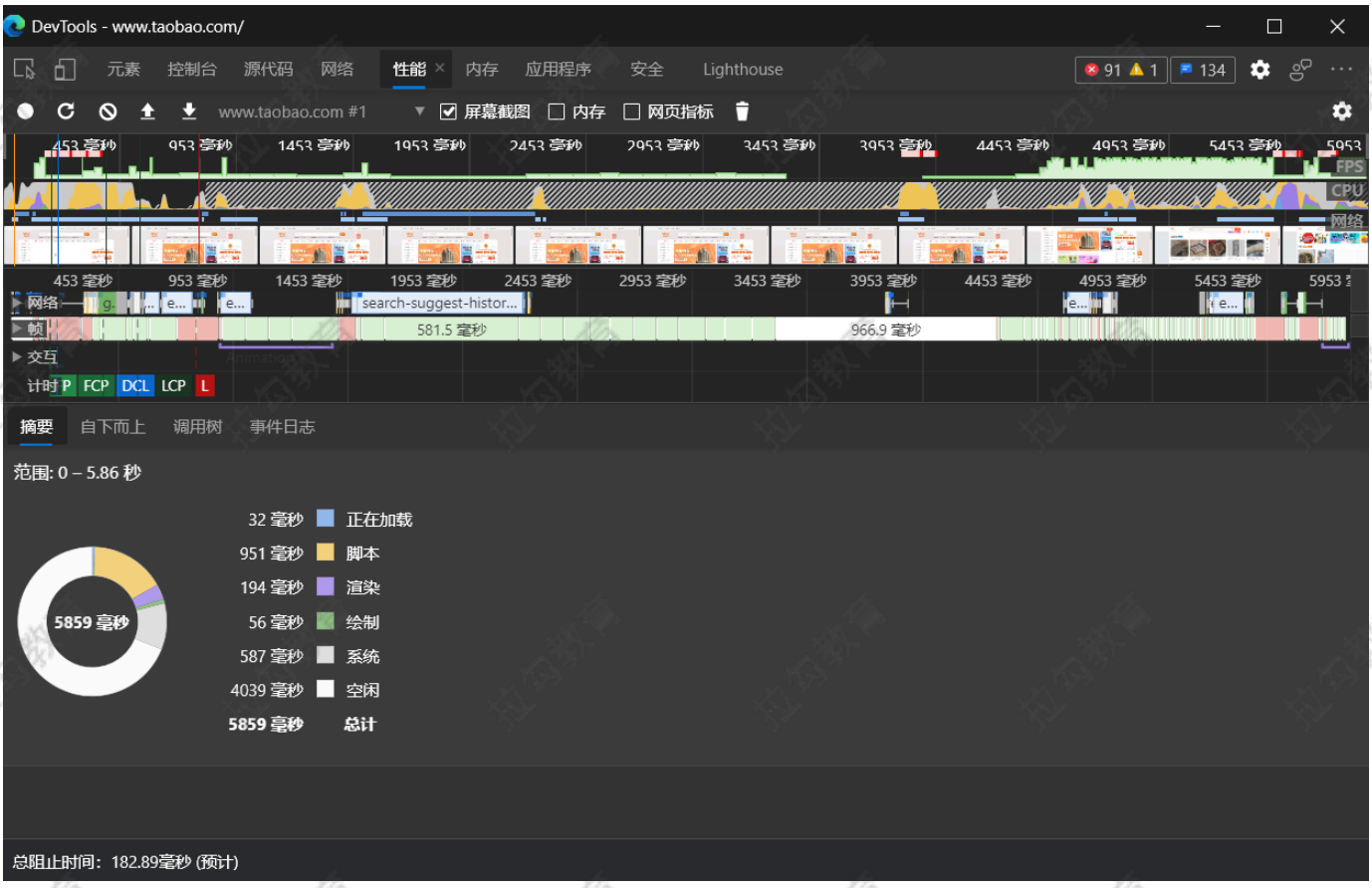
```
body {
  font-size: 16px;
}

p {
  font-weight: bold;
}

span {
  color: red;
}

p span {
  display: none;
}

img {
  float: right;
```

```
19 }
```

最后构建的 CSSOM 树如图所示。

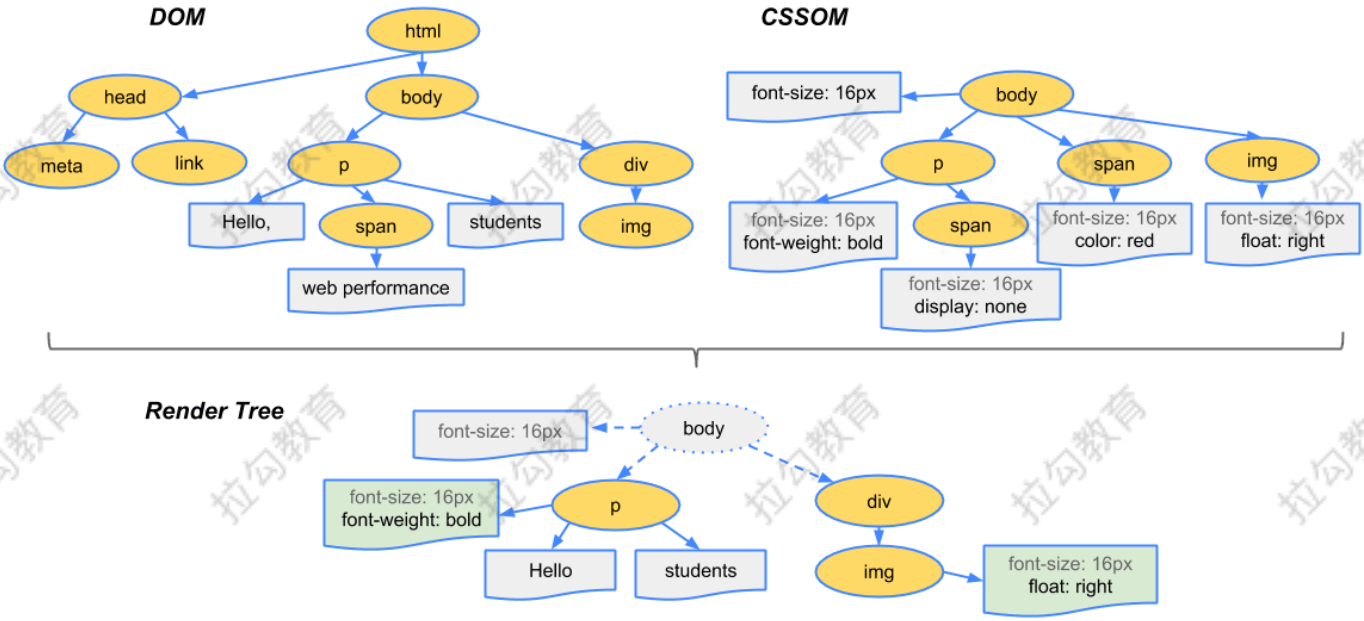


这两个对象模型的构建过程是会花费时间的，可以通过打开 Chrome 浏览器的开发者工具的性能选项卡，查看到对应过程的耗时情况，如图所示。



渲染绘制

当完成文档对象模型和层叠样式表对象模型的构建后，所得到的其实是描述最终渲染页面两个不同方面信息的对象：一个是展示的文档内容，另一个是文档对象对应的样式规则，接下来就需要将两个对象模型合并为渲染树，渲染树中只包含渲染可见的节点，该 HTML 文档最终生成的渲染树如图所示。



渲染绘制的步骤大致如下。

（1）从所生成 DOM 树的根节点开始向下遍历每个子节点，忽略所有不可见的节点（脚本标记不可见、CSS 隐藏不可见），因为不可见的节点不会出现在渲染树中。
（2）在 CSSOM 中为每个可见的子节点找到对应的规则并应用。
（3）布局阶段，根据所得到的渲染树，计算它们在设备视图中的具体位置和大小，这一步输出的是一个“盒模型”。
（4）绘制阶段，将每个节点的具体绘制方式转化为屏幕上的实际像素。

此处所举的例子较为简单，读者要明白执行构建渲染树、布局及绘制过程所需要的时间取决于实际文档的大小。文档越大，浏览器需要处理的任务就越多，样式也复杂，绘制需要的时间就越长，所以关键渲染路径执行快慢，将直接影响首屏加载时间的性能指标。

当首屏渲染完成后，用户在和网站的交互过程中，有可能通过 JavaScript 代码提供的用户操作接口更改渲染树的结构，一旦 DOM 结构发生改变，这个渲染过程就会重新执行一遍。可见对于关键渲染路径的优化影响的不仅是首屏性能，还有交互性能。

本节仅对首屏渲染过程进行了简要描述，其中细节性的优化方案，将会在后续章节中展开介绍。

小结

本章通过一道前端工程师常见的面试题，较为详细地描述了当用户从浏览器的地址栏输入 URL 后，到页面渲染出来的整个过程。其实不难理解当某个较差的性能体验发生时，很有可能是这个过程中的某个环节出现了过多的性能损耗，后续我们会介绍一些辅助的性能分析工具来帮助定位具体的性能瓶颈，其实它们也是以页面加载生命周期为“纲”进行逐步分析的，所以我们理解并掌握了这个过程，对具体的优化手段可以做到心中有数。

后续的章节安排，就是选取本章介绍的页面生命周期的某个局部环节进行优化，以及某些具体的优化技巧和实用工具。如果说这些是前端性能优化的“术”，那么理解页面生命周期就是“道”。

参考链接

- https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work
- https://zhuanlan.zhihu.com/p/80551769
- https://docs.microsoft.com/zh-cn/microsoft-edge/extensions-chromium/getting-started/
- https://blog.csdn.net/qzcsu/article/details/72861891
- https://zhuanlan.zhihu.com/p/53374516