浏览器输入 URL 后发生了什么?

这是一道非常经典的题目，相信你被面试或者面试别人有非常大的概率接触过，也可能只是其中某一部分进行提问。这道题涵盖的知识点非常多，考察得比较全面，网上一搜也有成百上千篇文章，不同的人有不同的见解，然而大部分都是千篇一律。如果你没有深入透彻系统性地研究过，光靠死记硬背，面试官稍稍针对某一点提问，或者换成另外一种方式提问，就有可能露出破绽。仔细想想，学习积累到了一定阶段，也该凭技术储备对知识体系进行一遍全面的梳理总结。

开放性的题目，没有固定的答案，涉及计算机图形学、操作系统、编译原理、计算机网络、通信原理、分布式系统、浏览器原理等多个不同的学科、领域。但无论从哪个领域入手，软件角度或硬件角度，铺开来讲都可以是长篇大论。如果你专精某个学科领域多年，那你在这一方面肯定比我有更深厚的经验、更独特的见解，欢迎指点。

从我的角度来看，在题意不够明确、缺少情景和限定条件的情况下，没法直接作答。在计算机越来越复杂的今天，任何一个条件的变化与组合，都有可能产生千千万万种可能，打破常规。对题目本身而言，就会包括但不仅限于以下几种条件：

- 请求资源类型
- 浏览器类型及版本
- 服务器类型及版本
- 网络协议类型及版本
- 网络链路状况
- 经过哪些中间设备
- 局域网类型及标准
- 物理媒介类型
- 运营商路线

如果请求的是静态资源，那么流量有可能到达 CDN 服务器；如果请求的是动态资源，那么情况更加复杂，流量可能依次经过代理/网关、Web 服务器、应用服务器、数据库。图 1 为阿里云 SLB（Server Load Balancer，负载均衡）高可用部署示意图，它不同于传统的主备切换模式过于依赖单机处理能力，来自公网的请求通过上层交换机的 ECMP（Equal-cost multi-path routing，等价多路径路由）将流量转发给 LVS 集群（四层 SLB），对于 TCP/UDP 请求，LVS 集群直接转发给后端 ECS 集群，对于 HTTP 请求，则转发给 Tengine 集群（七层 SLB），由 Tengine 集群再转发给后端 ECS 集群，集群之间通过实现会话同步、健康检查等机制来保证高可用。

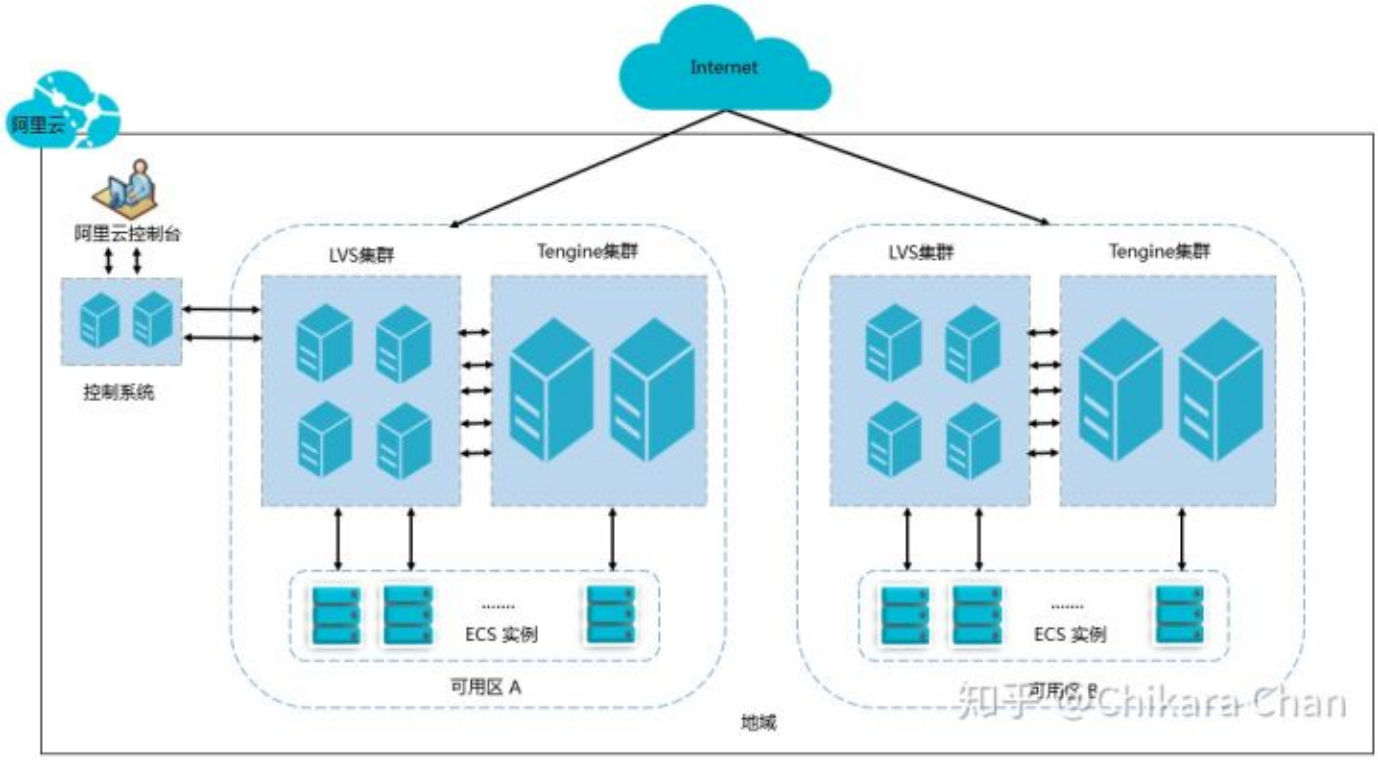

图 1：阿里云负载均衡服务

随着业务规模的不断扩大，为了承载千万级甚至亿级流量及海量存储，对系统的多机房容灾、自由扩容的要求越来越高，系统还可能演进为异地多活架构（图 2）。与传统的灾备设计不同的是，多个数据中心同时对外提供服务，同时保证异地单元间数据库数据的一致性和完整性（CAP 理论）。整个系统架构分为流量层、应用层、数据层，利用 DNS 技术实现 GSLB（Global Server Load Balance，全局负载均衡），实现用户就近访问。如果某个地域的系统发生整体故障，则把所有流量请求切换到另一个地域，来满足异地容灾，这也类似于饿了么现阶段整体架构方案。

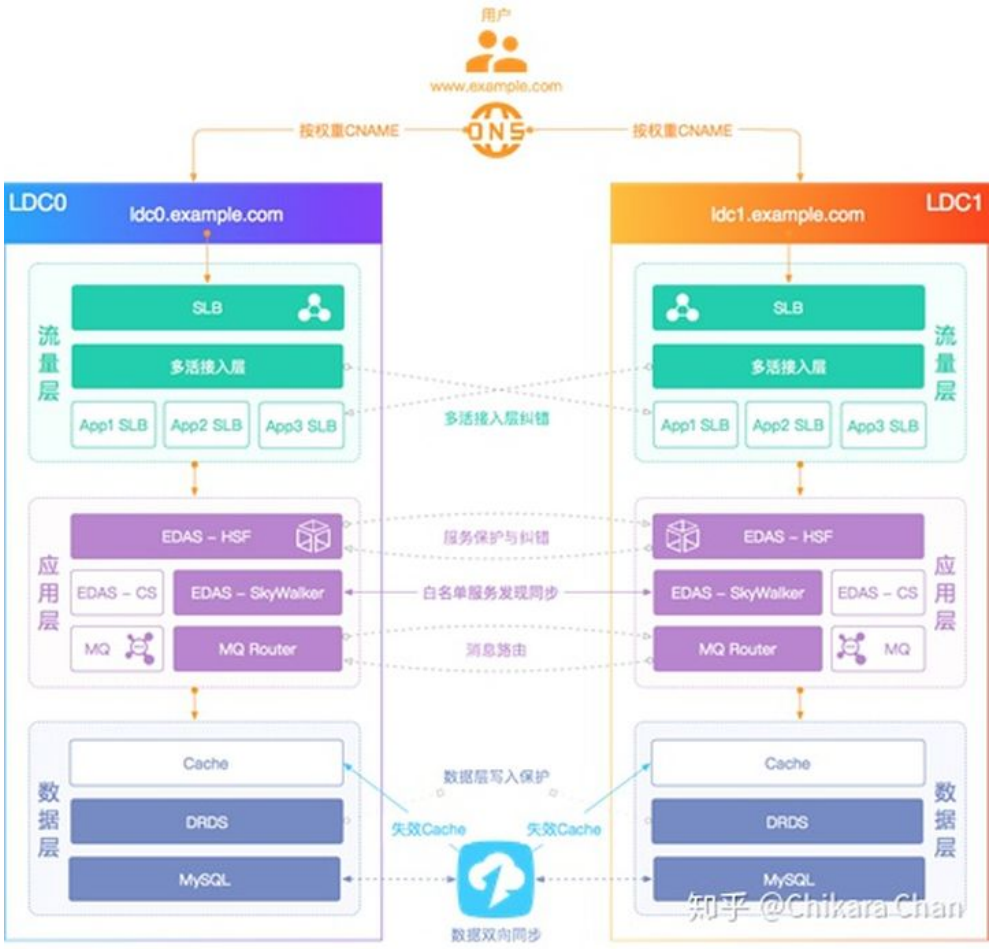

图 2：阿里云异地多活解决方案

但是，不同于动态资源，考虑部署成本和流量成本，静态资源一般是通过 CDN，利用中间服务器作缓存，如果没有命中缓存，再回源到 OSS（Object Storage Service，阿里云对象存储服务）或者私有服务器。

因此，现实世界的情况是千变万化的，你也很难想象一个 GET 请求甚至触发银行的转账操作，一切取决于人为实现。重新回到本文的主题，我们排除一切特殊条件，把问题简化一下，如果仅仅考虑：

- 一个 Chrome 浏览器
- 一台 Linux 服务器
- 发起 HTML 请求
- 不考虑任何缓存和优化机制
- 采用 HTTP/1.1 + TLS/1.2 + TCP 协议

这个过程如下：

DNS 解析过程

首先，浏览器向本地 DNS 服务器发起请求，由于本地 DNS 服务器没有缓存不能直接将域名转换为 IP 地址，需要采用递归或者迭代查询的方式（图 3）依次向根域名服务器、顶级域名服务器、权威域名服务器发起查询请求，直至找到一个或一组 IP 地址，返回给浏览器。一般本地 DNS 地址由 ISP（Internet Service Provider，互联网服务提供商）通过 DHCP 协议动态分配，我们仍可以手动把它修改为公共 DNS，比如 Google 提供的 8.8.8.8，国内的 114.114.114.114，它们分布在不同的地理位置上，借助 Anycast 技术，将请求路由到离用户最近的 DNS 服务器上。为了让 DNS 解析更加精确，客户端还需在请求包里带上自己的源 IP 地址，否则类似 GSLB 的 DNS 服务器不能够精准地匹配判断离用户最近的目标 IP 地址。

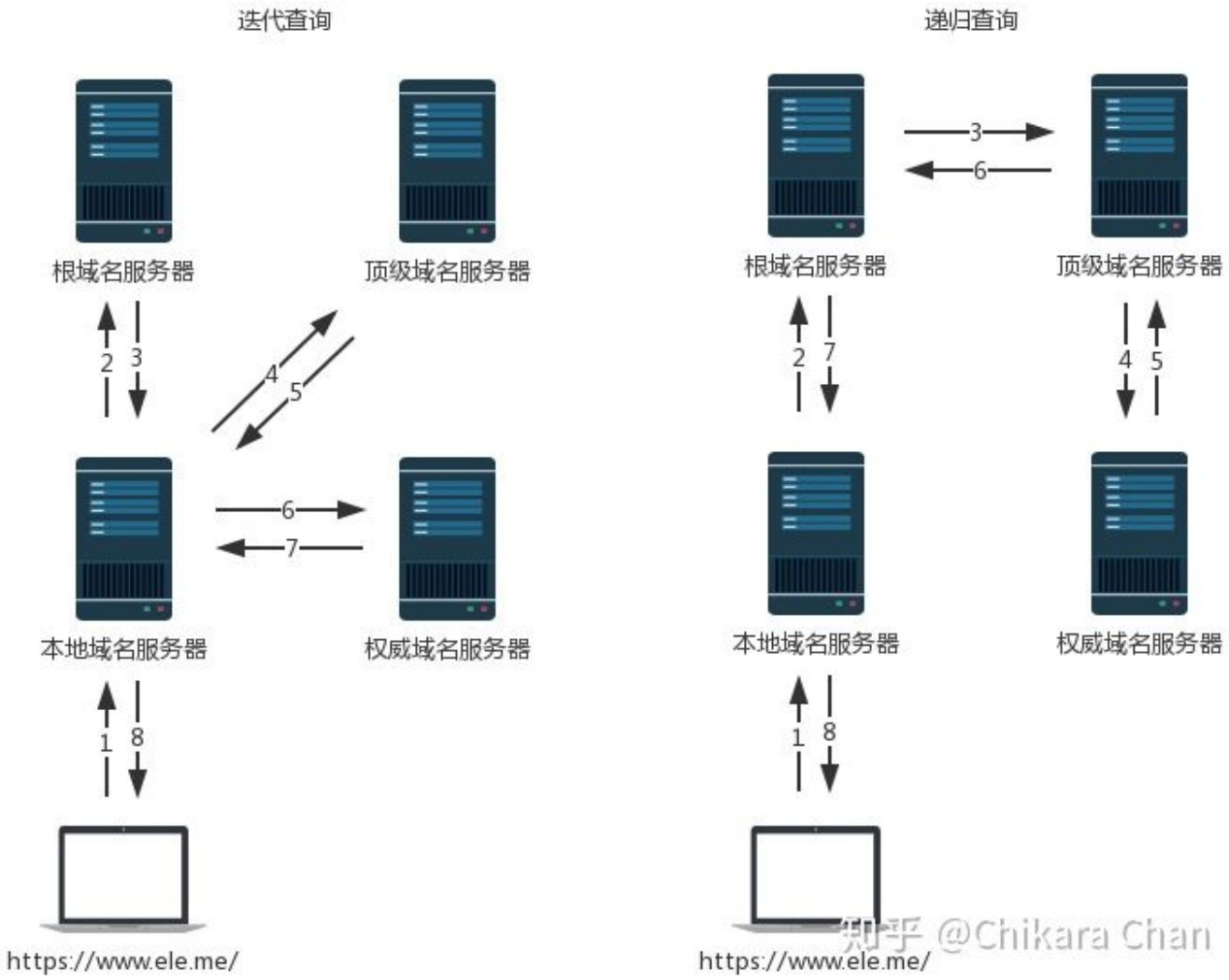

图 3：DNS 查询过程

HTTP 请求过程

由于 HTTP 是基于更易于阅读的文本格式，除了在浏览器直接发起 HTTP 之外，你也可以用命令行 `telnet` 来与服务器指定端口建立 TCP 连接，按照协议规定的格式，来发送请求头和请求实体。另外，如果想查看详细具体的封包内容，可以使用网络封包分析工具 Wireshark 或命令行 `tcpdump` ，来捕获某一块网卡上的数据包。在上一步我们通过 DNS 解析拿到服务器 IP 地址后，浏览器再通过系统调用 Socket 接口与服务器 443 端口进行通信，整个过程可以分解为建立连接、发送 HTTP 请求、返回 HTTP 响应、维持连接、释放连接五个部分（图 4）。图中所示箭头有可能代表一个 TCP 报文段，也有可能代表一个完整的应用层报文，在实际传输过程中，会被组合为一个或分片为多个 TCP 报文段。

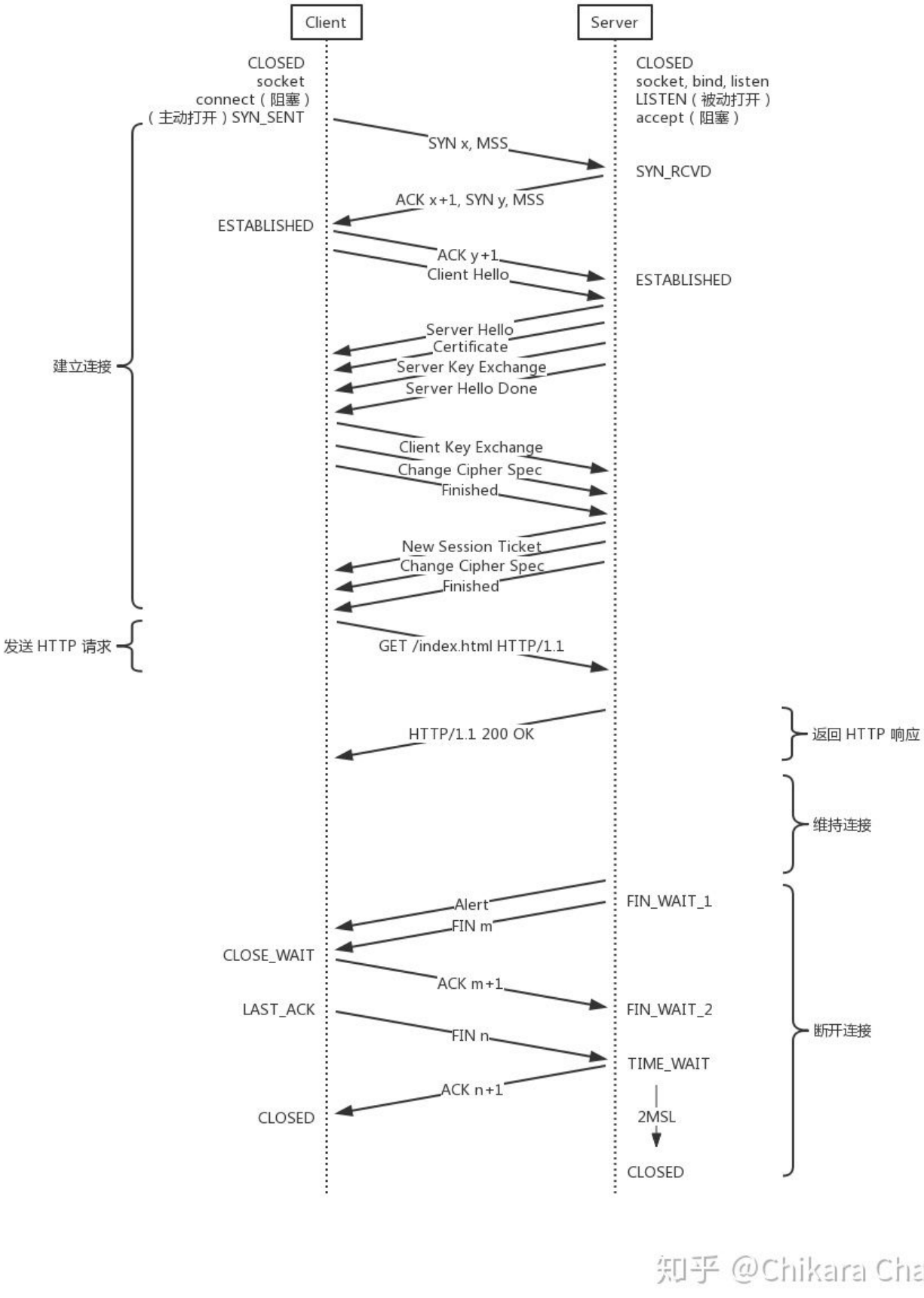

图 4：HTTP 请求过程

建立连接

1. 在连接建立之前，服务器必须做好接受连接的准备，通过调用 socket、bind、listen 和 accept 四个函数来完成绑定公网 IP、监听 443 端口和接受请求的任务。
2. 客户端通过 socket 和 connect 两个函数来主动打开连接，给服务器发送带有 SYN 标志位的

分组，随机生成一个初始序列号 x，以及附带 MSS（Maximum Segment Size，最大段大小）等额外信息。为了避免在网络层被 IP 协议分片使得出现丢失错误的概率增加，及达到最佳的传输效果，MSS 的值一般为以太网 MTU（Maximum Transmission Unit，最大传输单元）的值减去 IP 头部和 TCP 头部大小，等于 1460 字节。

3. 服务器必须确认收到客户端的分组，发送带有 SYN+ACK 标志位的分组，随机生成一个初始序列号 y，确认号为 x+1，以及附带 MSS 等额外信息。当一端收到另外一端的 MSS 值时，会根据两者的 MSS 取最小值来决定随后的 TCP 最大报文段大小。
4. 客户端确认收到服务器的分组，发送带有 ACK 标志位的分组，确认号为 y+1，从而建立 TCP 连接。
5. 如果客户端此前未与服务器建立会话，那么双方需要进行一次完整的 TLS 四次握手。客户端首先向服务器发送 Client Hello 报文，包含一个随机数、TLS 协议版本、按优先级排列的加密套件列表。
6. 服务器向客户端发送 Server Hello 报文，包含一个新的随机数、TLS 协议版本、经过选择后的一个加密套件。
7. 服务器向客户端发送 Certificate 报文，包含服务器 X.509 证书链，其中，第一个为主证书，中间证书按照顺序跟在主证书之后，而根 CA 证书通常内置在操作系统或浏览器中，无需服务器发送。
8. 如果密钥交换选择 DH 算法，服务器会向客户端发送 Server Key Exchange 报文，包含密钥交换所需的 DH 参数；如果密钥交换选择 RSA 算法，则跳过这一步。
9. 服务器向客户端发送 Server Hello Done 报文，表明已经发送完所有握手消息。
10. 客户端向服务器发送 Client Key Exchange 报文，如果密钥交换选择 RSA 算法，由客户端生成预主密钥，使用服务器证书中的公钥对其加密，包含在报文中，服务器只需使用自己的私钥解密就可以取出预主密钥；如果密钥交换选择 DH 算法，客户端会在报文中包含自己的 DH 参数，之后双方都根据 DH 算法计算出相同的预主密钥。需要注意的是，密钥交换的只是预主密钥，这个值还需进一步加工，结合客户端和服务器两个随机数种子，双方使用 PRF（pseudorandom function，伪随机函数）生成相同的主密钥。
11. 客户端向服务器发送 Change Cipher Spec 报文，表明已经生成主密钥，在随后的传输过程都使用这个主密钥对消息进行对称加密。
12. 客户端向服务器发送 Finished 报文，这条消息是经过加密的，因此在 Wireshark 中显示的是 Encrypted Handshake Message。如果服务器能解密出报文内容，则说明双方生成的主密钥是一致的。
13. 服务器向客户端发送 New Session Ticket 报文，而这个 Session Ticket 只有服务器才能解密，客户端把它保存下来，在以后的 TLS 重新握手过程中带上它进行快速会话恢复，减少往返延迟。
14. 服务器向客户端发送 Change Cipher Spec 报文，同样表明已经生成主密钥，在随后的传输过程都使用这个主密钥对消息进行对称加密。
15. 服务器向客户端发送 Finished 报文，如果客户端能解密出报文内容，则说明双方生成的主密钥是一致的。至此，完成所有握手协商。

发送 HTTP 请求

建立起安全的加密信道后，浏览器开始发送 HTTP 请求，一个请求报文由请求行、请求头、空行、实体（Get 请求没有）组成。请求头由通用首部、请求首部、实体首部、扩展首部组成。其中，通用首部表示无论是请求报文还是响应报文都可以使用，比如 Date；请求首部表示只有在请求报文

中才有意义，分为 Accept 首部、条件请求首部、安全请求首部和代理请求首部这四类；实体首部作用于实体内容，分为内容首部和缓存首部这两类；扩展首部表示用户自定义的首部，通过 `X-` 前缀来添加。另外需要注意的是，HTTP 请求头是不区分大小写的，它基于 ASCII 进行编码，而实体可以基于其它编码方式，由 `Content-Type` 决定。

返回 HTTP 响应

服务器接受并处理完请求，返回 HTTP 响应，一个响应报文格式基本等同于请求报文，由响应行、响应头、空行、实体组成。区别于请求头，响应头有自己的响应首部集，比如 Vary、Set-Cookie，其它的通用首部、实体首部、扩展首部则共用。此外，浏览器和服务器必须保证 HTTP 的传输顺序，各自维护的队列中请求/响应顺序必须一一对应，否则会出现乱序而出错的情况。

维持连接

完成一次 HTTP 请求后，服务器并不是马上断开与客户端的连接。在 HTTP/1.1 中， `Connection: keep-alive` 是默认启用的，表示持久连接，以便处理不久后到来的新请求，无需重新建立连接而增加慢启动开销，提高网络的吞吐能力。在反向代理软件 Nginx 中，持久连接超时时间默认值为 75 秒，如果 75 秒内没有新到达的请求，则断开与客户端的连接。同时，浏览器每隔 45 秒会向服务器发送 TCP keep-alive 探测包，来判断 TCP 连接状况，如果没有收到 ACK 应答，则主动断开与服务器的连接。注意，HTTP keep-alive 和 TCP keep-alive 虽然都是一种保活机制，但是它们完全不相同，一个作用于应用层，一个作用于传输层。

断开连接

1. 服务器向客户端发送 Alert 报文，类型为 Close Notify，通知客户端不再发送数据，即将关闭连接，同样，这条报文也是经过加密处理的。
2. 服务器通过调用 close 函数主动关闭连接，向客户端发送带有 FIN 标志位的分组，序列号为 m。
3. 客户端确认收到该分组，向服务器发送带有 ACK 标志位的分组，确认号为 m+1。
4. 客户端发送完所有数据后，向服务器发送带有 FIN 标志位的分组，序列号为 n。
5. 服务器确认收到该分组，向客户端发送带有 ACK 标志位的分组，序列号为 n+1。客户端收到确认分组后，立即进入 CLOSED 状态；同时，服务器等待 2 个 MSL(Maximum Segment Lifetime，最大报文生存时间) 的时间后，进入 CLOSED 状态。

浏览器解析过程

现代浏览器是一个及其庞大的大型软件，在某种程度上甚至不亚于一个操作系统，它由多媒体支持、图形显示、GPU 渲染、进程管理、内存管理、沙箱机制、存储系统、网络管理等大大小小数百个组件组成。虽然开发者在开发 Web 应用时，无需关心底层实现细节，只需将页面代码交付于浏览器计算，就可以展示出丰富的内容。但页面性能不仅仅关乎浏览器的实现方式，更取决于开发

者的水平，对工具的熟悉程度，代码优化是无止尽的。显然，了解浏览器的基本原理，了解 W3C 技术标准，了解网络协议，对设计、开发一个高性能 Web 应用帮助非常大。

当我们在使用 Chrome 浏览器时，其背后的引擎是 Google 开源的 Chromium 项目，而 Chromium 的内核则是渲染引擎 Blink（基于 Webkit）和 JavaScript 引擎 V8。在阐述浏览器解析 HTML 文件之前，先简单介绍一下 Chromium 的多进程多线程架构（图 5），它包括多个进程：

- 一个 Browser 进程
- 多个 Renderer 进程
- 一个 GPU 进程
- 多个 NPAPI Render 进程
- 多个 Pepper Plugin 进程

而每个进程包括若干个线程：

- 一个主线程
- 在 Browser 进程中：渲染更新界面
- 在 Renderer 进程中：使用持有的内核 Blink 实例解析渲染更新界面
- 一个 IO 线程
- 在 Browser 进程中：处理 IPC 通信和网络请求
- 在 Renderer 进程中：处理与 Browser 进程之间的 IPC 通信
- 一组专用线程
- 一个通用线程池

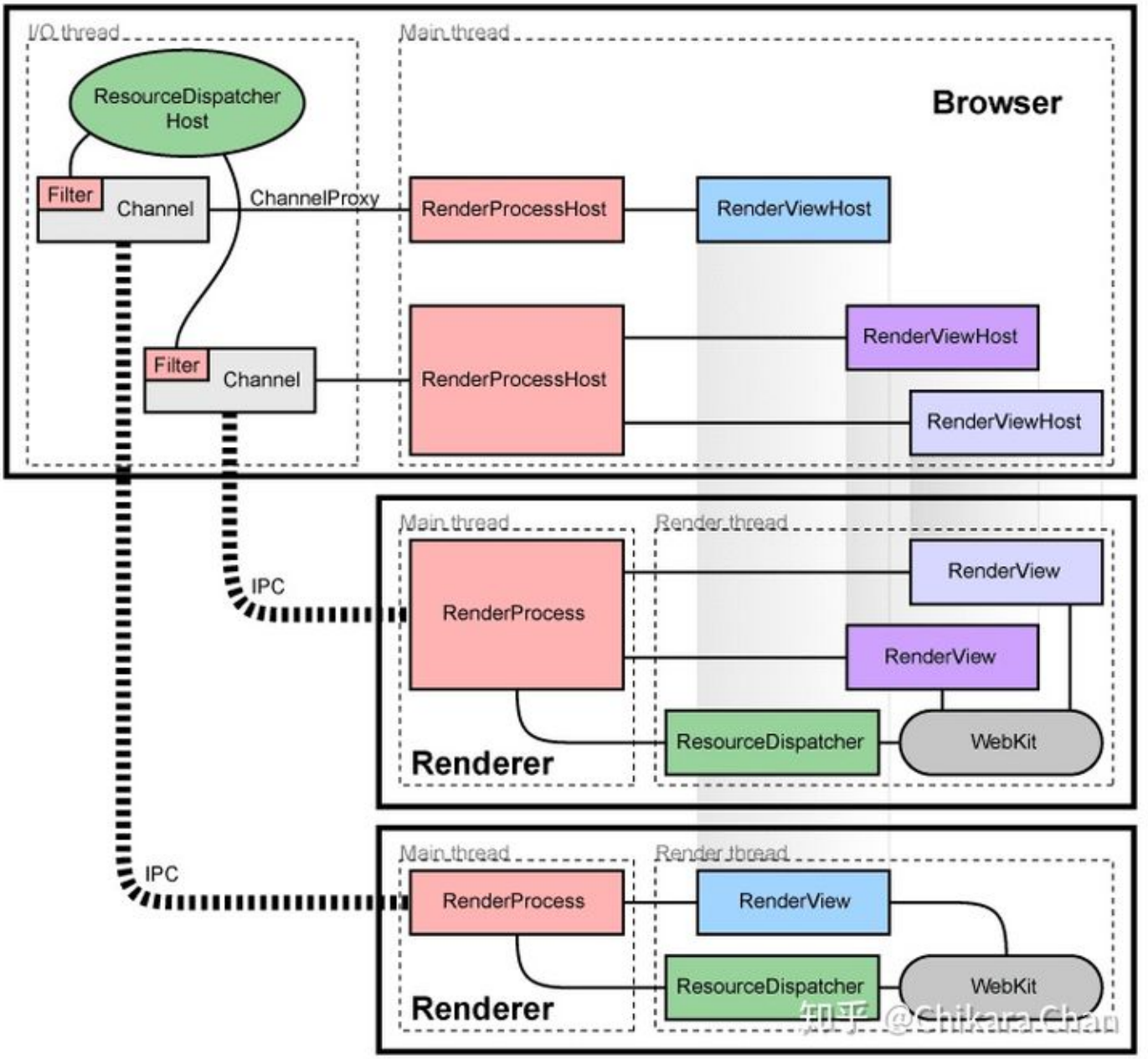

图 5：Chromium 多进程多线程架构

Chromium 支持多种不同的方式管理 Renderer 进程，不仅仅是每一个开启的 Tab 页面，iframe 页面也包括在内，每个 Renderer 进程是一个独立的沙箱，相互之间隔离不受影响。

- Process-per-site-instance：每个域名开启一个进程，并且从一个页面链接打开的新页面共享一个进程（noopener 属性除外），这是默认模式
- Process-per-site：每个域名开启一个进程
- Process-per-tab：每个 Tab 页面开启一个进程
- Single process：所有页面共享一个进程

当 Renderer 进程需要访问网络请求模块（XHR、Fetch），以及访问存储系统（同步 Local Storage、同步 Cookie、异步 Cookie Store）时，则调用 RenderProcess 全局对象通过 IO 线程与 Browser 进程中的 RenderProcessHost 对象建立 IPC 信道，底层通过 socketpair 来实现。正由于这种机制，Chromium 可以更好地统一管理资源、调度资源，有效地减少网络、性能开销。

主流程

页面的解析工作是在 Renderer 进程中进行的，Renderer 进程通过在主线程中持有的 Blink 实例边接收边解析 HTML 内容（图 6），每次从网络缓冲区中读取 8KB 以内的数据。浏览器自上而下逐行解析 HTML 内容，经过词法分析、语法分析，构建 DOM 树。当遇到外部 CSS 链接时，主线程调用网络请求模块异步获取资源，不阻塞而继续构建 DOM 树。当 CSS 下载完毕后，主线程在合适的时机解析 CSS 内容，经过词法分析、语法分析，构建 CSSOM 树。浏览器结合 DOM 树和 CSSOM 树构建 Render 树，并计算布局属性，每个 Node 的几何属性和在坐标系中的位置，最后进行绘制展示在屏幕上。当遇到外部 JS 链接时，主线程调用网络请求模块异步获取资源，由于 JS 可能会修改 DOM 树和 CSSOM 树而造成回流和重绘，此时 DOM 树的构建是处于阻塞状态的。但主线程并不会挂起，浏览器会使用一个轻量级的扫描器去发现后续需要下载的外部资源，提前发起网络请求，而脚本内部的资源不会识别，比如 `document.write` 。当 JS 下载完毕后，浏览器调用 V8 引擎在 Script Streamer 线程中解析、编译 JS 内容，并在主线程中执行（图 7）。

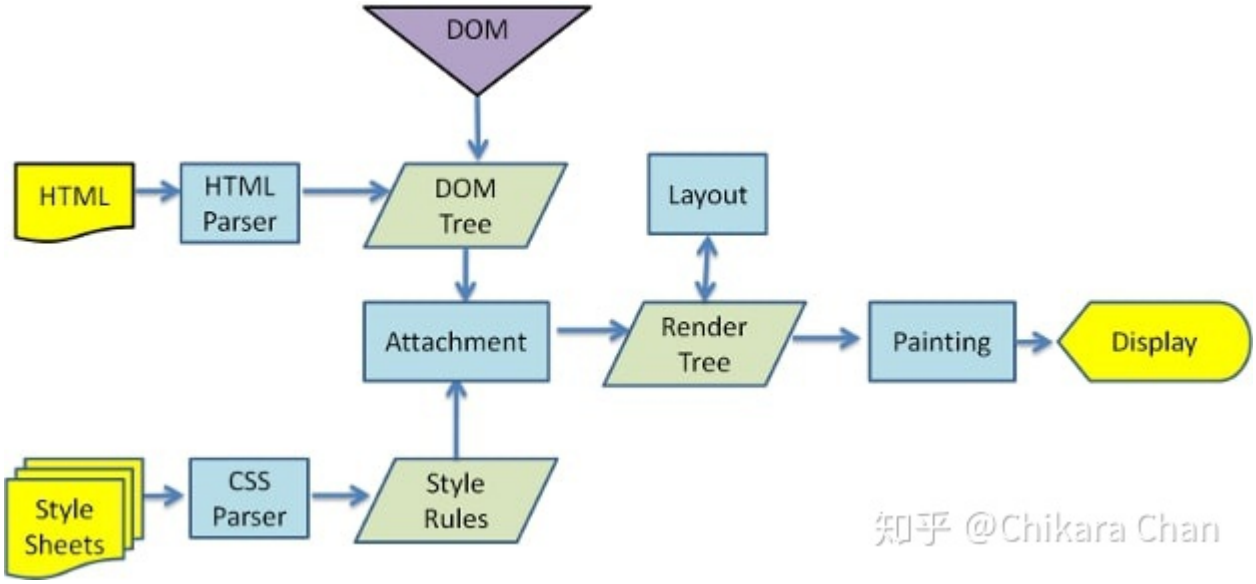

图 6：Webkit 主流程

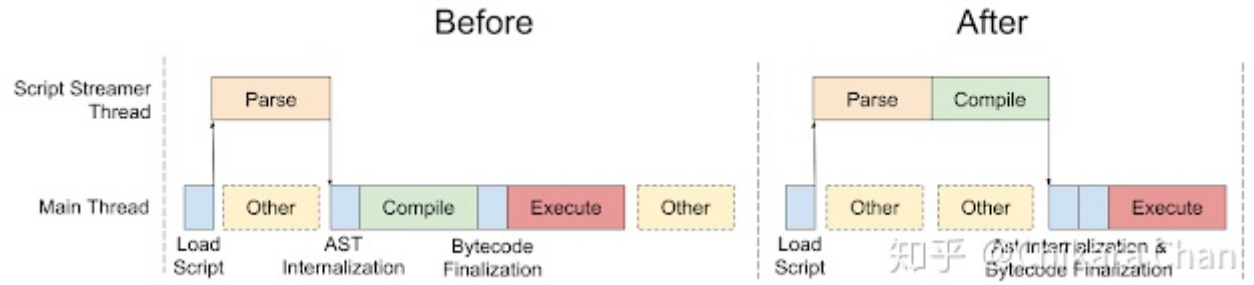

图 7：V8 解释流程，Chrome 66 以前对比 Chrome 66

渲染流程

当 DOM 树构建完毕后，还需经过好几次转换，它们有多种中间表示（图 8）。首先计算布局、绘图样式，转换为 RenderObject 树（也叫 Render 树）。再转换为 RenderLayer 树，当 RenderObject 拥有同一个坐标系（比如 canvas、absolute）时，它们会合并为一个 RenderLayer，这一步由 CPU 负责合成。接着转换为 GraphicsLayer 树，当 RenderLayer 满足合成层条件（比如 transform，熟知的硬件加速）时，会有自己的 GraphicsLayer，否则与父节点合并，这一步同样由 CPU 负责合成。最后，每个 GraphicsLayer 都有一个 GraphicsContext 对象，负责将层绘制成位图作为纹理上传给 GPU，由 GPU 负责合成多个纹理，最终显示在屏幕上。

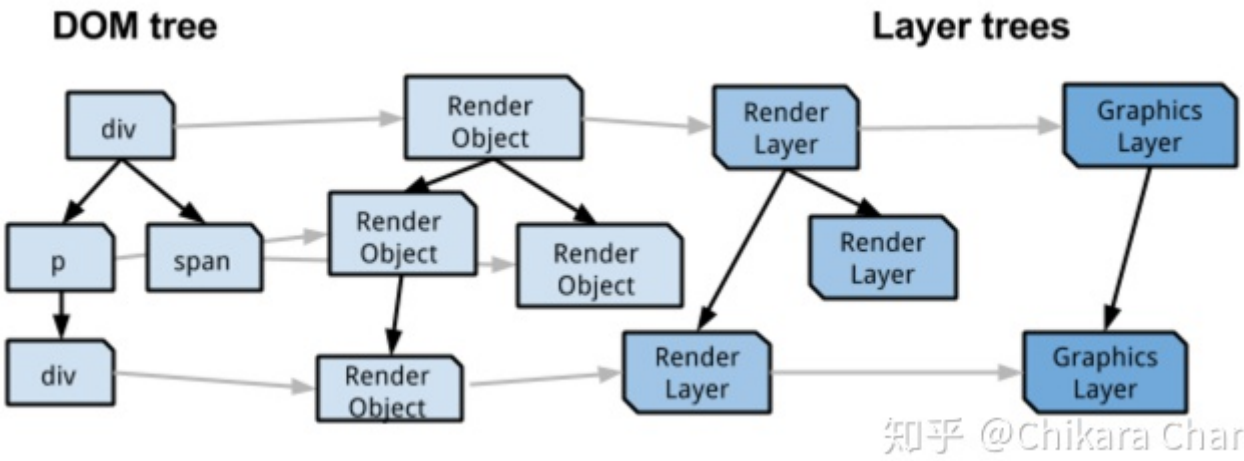

图 8：从 DOM 树到 GraphicsLayer 树的转换

另外，为了提升渲染性能效率，浏览器会有专用的 Compositor 线程来负责层合成（图 9），同时负责处理部分交互事件（比如滚动、触摸），直接响应 UI 更新而不阻塞主线程。主线程把 RenderLayer 树同步给 Compositor 线程，由它开启多个 Rasterizer 线程，进行光栅化处理，在可视区域以瓦片为单位把顶点数据转换为片元，最后交付给 GPU 进行最终合成渲染。

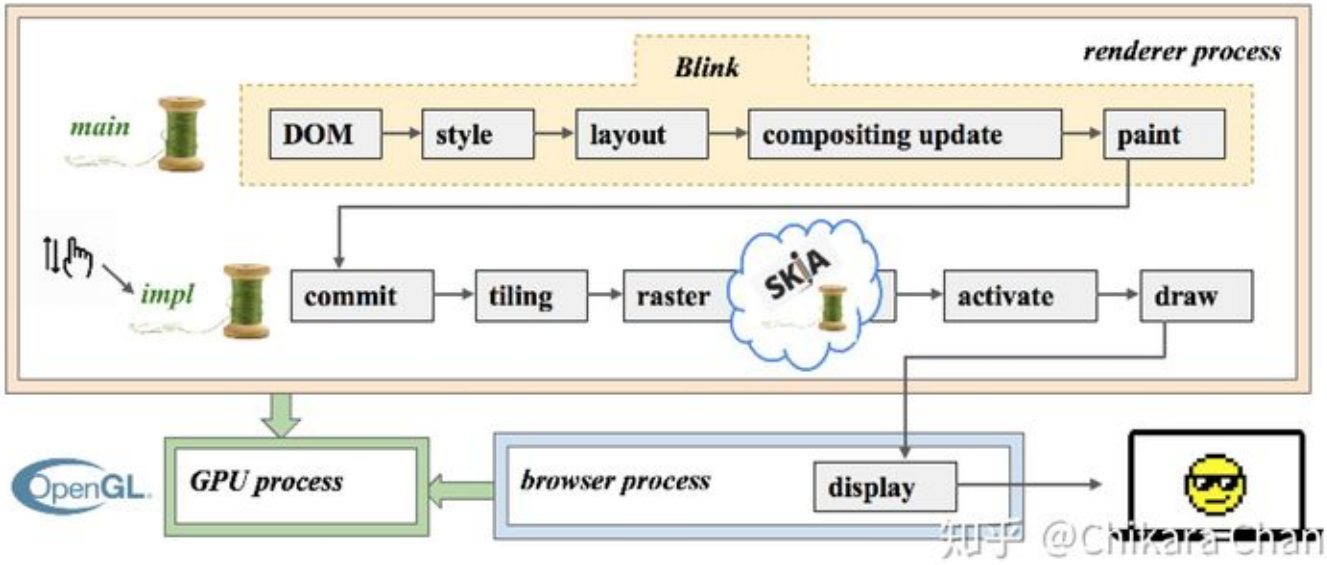

图 9：Chromium 多线程渲染

页面生命周期

页面从发起请求开始，结束于跳转、刷新或关闭，会经过多次状态变化和事件通知，因此了解整个过程的生命周期非常有必要。浏览器提供了 Navigation Timing 和 Resource Timing 两种 API 来记录每一个资源的事件发生时间点，你可以用它来收集 RUM（Real User Monitoring，真实用户监控）数据，发送给后端监控服务，综合分析页面性能来不断改善用户体验。图 10 表示 HTML 资源加载的事件记录全过程，而中间黄色部分表示其它资源（CSS、JS、IMG、XHR）加载事件记录过程，它们都可以通过调用 `window.performance.getEntries()` 来获取具体指标数据。

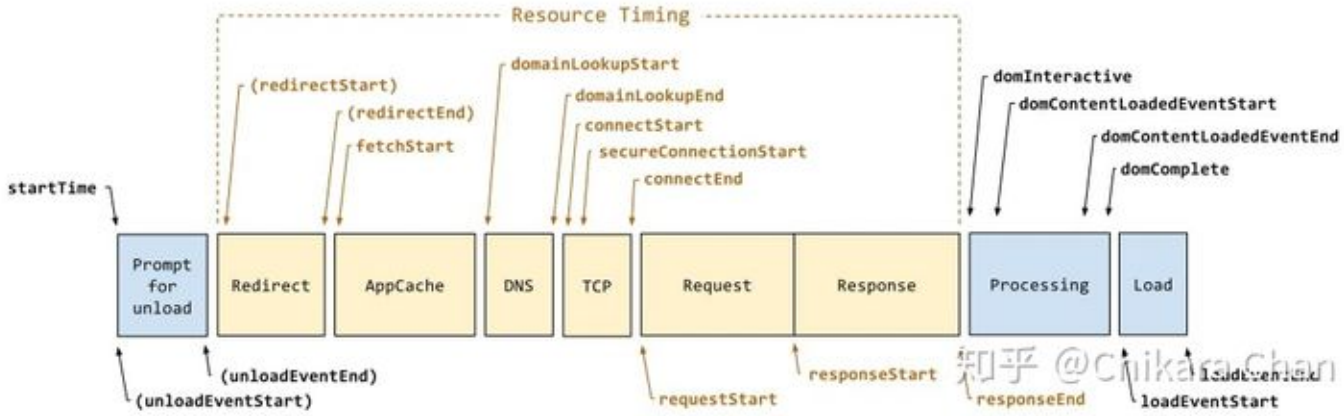

图 10：页面加载事件记录流程

衡量一个页面性能的方式有很多，但能给用户带来直接感受的是页面何时渲染完成、何时可交互、何时加载完成。其中，有两个非常重要的生命周期事件，DOMContentLoaded 事件表示 DOM 树构建完毕，可以安全地访问 DOM 树所有 Node 节点、绑定事件等等；load 事件表示所有资源都加载完毕，图片、背景、内容都已经完成渲染，页面处于可交互状态。但是迄今为止浏览器并不能像 Android 和 iOS app 一样完全掌控应用的状态，在前后台切换的时候，重新分配资源，合理地利用内存。实际上，现代浏览器都已经在做这方面的相关优化，并且自 Chrome 68 以后提供了 Page Lifecycle API，定义了全新的浏览器生命周期（图 11），让开发者可以构建更出色的应用。

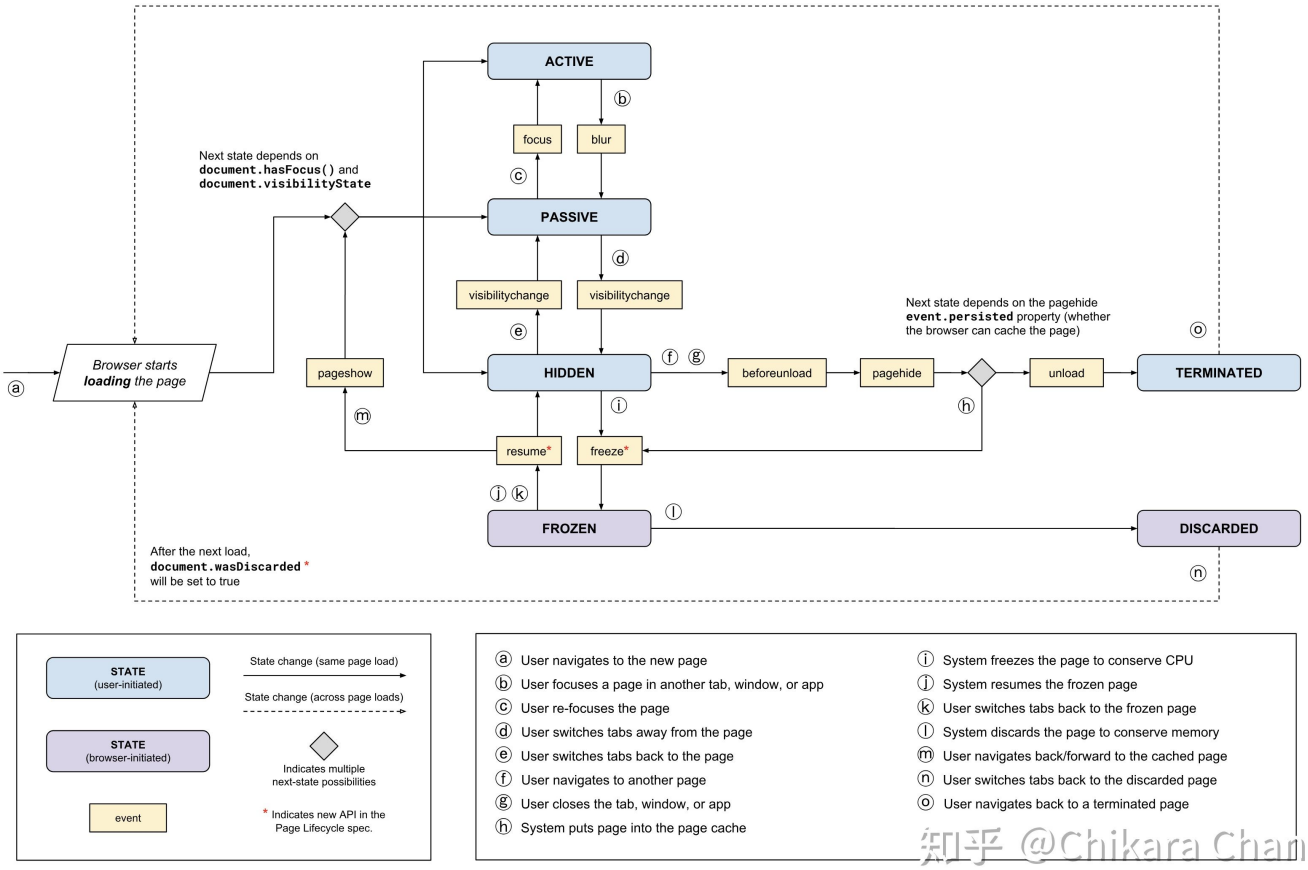

图 11：新版页面生命周期

现在，你可以通过给 `window` 和 `document` 绑定上所有生命周期监听事件（图 12），来监测页面切换、用户交互行为所触发的状态变化过程。不过，开发者只能感知事件在何时发生，不能直接获取某一刻的页面状态（图 11 中的 STATE）。即使如此，利用这个 API，也可以让脚本在合适的时机执行某项任务或进行界面 UI 反馈。

图 12：生命周期监听事件