# 一、乱七八糟

首屏的显示就涉及了 DNS、HTTP、DOM 解析、CSS 阻塞、JavaScript 阻塞等技术因素，其中一项没处理好就可能导致整个页面的延时。

Node.js 是基于 Chrome 的 JavaScript 引擎 V8 来实现的，它的特点是可以脱离浏览器环境来执行 JavaScript，于是大家惊讶地发现，原来也可以使用 JavaScript 写服务器程序呀！

打开一个新页面采用的渲染进程策略：

通常情况下，打开新的页面都会使用单独的渲染进程；

如果从 A 页面打开 B 页面，且 A 和 B 都属于同一站点的话，那么 B 页面复用 A 页面的渲染进程；如果是其他情况，浏览器进程则会为 B 创建一个新的渲染进程。

process-per-site-instance：如果从一个页面打开了另一个新页面，而新页面和当前页面属于同一站点的话，那么新页面会复用父页面的渲染进程

变量提升：是指在 JavaScript 代码执行过程中，JavaScript 引擎把变量的声明部分和函数的声明部分提升到代码开头的“行为”。变量被提升后，会给变量设置默认值，这个默认值就是我们熟悉的 undefined。

# 二、并行处理，进程、线程，单线程、多线程

并行处理：同一时间，处理多个任务。

进程：一个进程就是一个程序的运行实例。详细解释就是，当一个程序开始运行时，它就是一个进程。操作系统会为该程序创建一块内存，用来存放代码、运行中的数据和一个执行任务的主线程。一个进程可由多个线程组成。

线程：程序中的一个执行流，按照某个顺序执行任务。每个线程都有自己的专有寄存器(栈指针、程序计数器等)，但代码区是共享的，即不同的线程可以执行同样的函数。

单线程：程序中只包含一个执行流。

多线程：程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。多线程可以并行处理任务。

线程是依附于进程的，由进程来启动和管理的。进程中使用多线程并行处理能提升运算效率。进程中的任意一线程执行出错，都会导致整个进程的崩溃。线程之间共享进程中的数据。当一个进程关闭之后，操作系统会回收进程所占用的内存。进程之间的内容相互隔离。

# 三、chrome浏览器

Chrome浏浏览器基于多进程进行架构。包括：1 个浏览器（Browser）主进程、1 个 GPU 进程、1 个网络（NetWork）进程、多个渲染进程和多个插件进程。

浏览器进程。主要负责界面显示、用户交互、子进程管理，同时提供存储等功能。

渲染进程。核心任务是将 HTML、CSS 和 JavaScript 转换为用户可以与之交互的网页，排版引擎 Blink 和 JavaScript 引擎 V8 都是运行在该进程中，默认情况下，Chrome 会为每个 Tab 标签创建一个渲染进程。出于安全考虑，渲染进程都是运行在沙箱模式下。 GPU 进程。其实，Chrome 刚开始发布的时候是没有 GPU 进程的。而 GPU 的使用初衷是为了实现 3D CSS 的效果，只是随后网页、Chrome 的 UI 界面都选择采用 GPU 来绘制，这使得 GPU 成为浏览器普遍的需求。最后，Chrome 在其多进程架构上也引入了 GPU 进程。

网络进程。主要负责页面的网络资源加载，之前是作为一个模块运行在浏览器进程里面的，直至最近才独立出来，成为一个单独的进程。插件进程。主要是负责插件的运行，因插件易崩溃，所以需要通过插件进程来隔离，以保证插件进程崩溃不会对浏览器和页面造成影响。

所以chrome浏览器仅仅打开了 1 个页面，为什么有 4 个进程？因为打开 1 个页面至少需要 1 个网络进程、1 个浏览器进程、1 个 GPU 进程以及 1 个渲染进程，共 4 个；如果打开的页面有运行插件的话，还需要再加上 1 个插件进程。

Chrome 有个机制，同一个域名同时最多只能建立 6 个 TCP 连接，如果在同一个域名下同时有 10 个请求发生，那么其中 4 个请求会进入排队等待状态，直至进行中的请求完成。

# 四、如何保证文件能被完整的送到浏览器

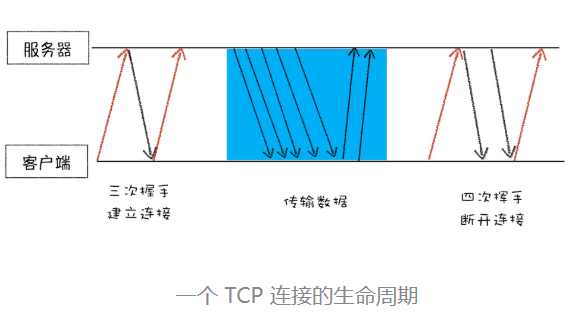
文件以数据包的形式传输。

IP ：网际协议（Internet Protocol），把数据包送达目的主机。 通过 IP 地址信息把数据包发送给指定的电脑。

UDP：用户数据包协议（User Datagram Protocol），把数据包送达应用程序。 通过端口号把数据包分发给正确的程序。UDP 不能保证数据可靠性，但是传输速度却非常快。缺点：容易丢包，不提供重传机智，大包被拆为小包，不提供包排序机制，不知道如何组装包。优点：传输速度快。

TCP：传输控制协议（Transmission Control Protocol，传输控制协议）。把数据完整地送达应用程序，是一种面向连接的、可靠的、基于字节流的传输层通信协议。是UDP的升级，解决了UDP的缺点。

一个完整的 TCP 连接的生命周期包括了“建立连接”“传输数据”和“断开连接”三个阶段。



首先，建立连接阶段。这个阶段是通过“三次握手”来建立客户端和服务器之间的连接。TCP 提供面向连接的通信传输。面向连接是指在数据通信开始之前先做好两端之间的准备工作。所谓三次握手，是指在建立一个 TCP 连接时，客户端和服务器总共要发送三个数据包以确认连接的建立。

其次，传输数据阶段。在该阶段，接收端需要对每个数据包进行确认操作，也就是接收端在接收到数据包之后，需要发送确认数据包给发送端。所以当发送端发送了一个数据包之后，在规定时间内没有接收到接收端反馈的确认消息，则判断为数据包丢失，并触发发送端的重发机制。同样，一个大的文件在传输过程中会被拆分成很多小的数据包，这些数据包到达接收端后，接收端会按照 TCP 头中的序号为其排序，从而保证组成完整的数据。

最后，断开连接阶段。数据传输完毕之后，就要终止连接了，涉及到最后一个阶段“四次挥手”来保证双方都能断开连接。

HTTP 协议是基于 TCP 协议的。HTTP协议和TCP协议都是TCP/IP协议簇的子集。HTTP协议属于应用层，TCP协议属于传输层，HTTP协议位于TCP协议的上层。请求方要发送的数据包，在应用层加上HTTP头以后会交给传输层的TCP协议处理，应答方接收到的数据包，在传输层拆掉TCP头以后交给应用层的HTTP协议处理。建立 TCP 连接后会顺序收发数据，请求方和应答方都必须依据 HTTP 规范构建和解析HTTP报文。

# 五、浏览器缓存

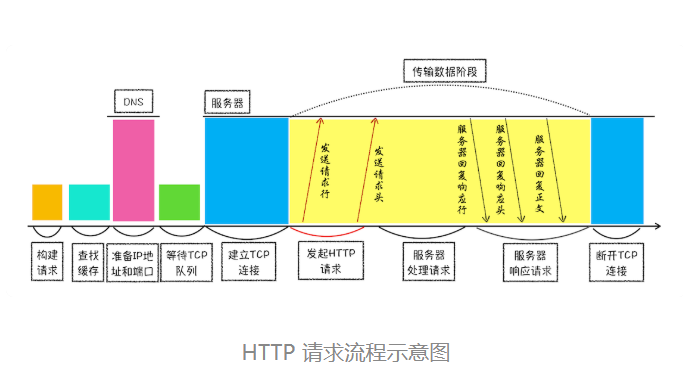
第二次打开页面很快，主要原因是因为第一次加载的过程中，缓存了一些耗时数据。

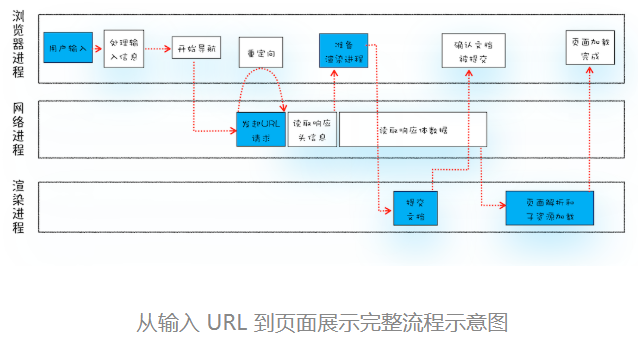
缓存了哪些数据？DNS缓存和页面资源缓存。

网页第一次请求之后，浏览器把资源缓存在本地，下次请求的时候查看是否在在缓存有效期内，在：直接在本地拿，关闭请求，不在去服务器拿，请求头加If-None-Match:"4f80f-13c-3a1xb12a"，服务器对比资源是否有更新，无更新，返回304，在本地拿，有更新，直接返回资源，再缓存。

通过响应头Cache-Control:Max-age=2000查看是否缓存，和缓存有效期

# 六、浏览器输入网址，发生了什么





导航:用户发出 URL 请求到页面开始解析的这个过程。

1. 用户输入URL，浏览器会根据用户输入的信息判断是搜索还是网址，如果是搜索内容，就将搜索内容+默认搜索引擎合成新的URL；如果用户输入的内容符合URL规则，浏览器就会根据URL协议，在这段内容上加上协议合成合法的URL  
2. 用户输入完内容，按下回车键，浏览器导航栏显示loading状态，但是页面还是呈现前一个页面，这是因为新页面的响应数据还没有获得  
3. 浏览器进程浏览器构建请求行信息，会通过进程间通信（IPC）将URL请求发送给网络进程  
GET /index.html HTTP1.1  
4. 网络进程获取到URL，先去本地缓存中查找是否有缓存文件，如果有，拦截请求，直接200返回；否则，进入网络请求过程  
5. 网络进程请求DNS返回域名对应的IP和端口号，如果之前DNS数据缓存服务缓存过当前域名信息，就会直接返回缓存信息；否则，发起请求获取根据域名解析出来的IP和端口号，如果没有端口号，http默认80，https默认443。如果是https请求，还需要建立TLS连接。  
6. Chrome 有个机制，同一个域名同时最多只能建立 6 个TCP 连接，如果在同一个域名下同时有 10 个请求发生，那么其中 4 个请求会进入排队等待状态，直至进行中的请求完成。如果当前请求数量少于6个，会直接建立TCP连接。  
7. TCP三次握手建立连接，http请求加上TCP头部——包括源端口号、目的程序端口号和用于校验数据完整性的序号，向下传输  
8. 网络层在数据包上加上IP头部——包括源IP地址和目的IP地址，继续向下传输到底层  
9. 底层通过物理网络传输给目的服务器主机  
10. 目的服务器主机网络层接收到数据包，解析出IP头部，识别出数据部分，将解开的数据包向上传输到传输层  
11. 目的服务器主机传输层获取到数据包，解析出TCP头部，识别端口，将解开的数据包向上传输到应用层  
12. 应用层HTTP解析请求头和请求体，如果需要重定向，HTTP直接返回HTTP响应数据的状态code301或者302，同时在请求头的Location字段中附上重定向地址，浏览器会根据code和Location进行重定向操作；如果不是重定向，首先服务器会根据 请求头中的If-None-Match 的值来判断请求的资源是否被更新，如果没有更新，就返回304状态码，相当于告诉浏览器之前的缓存还可以使用，就不返回新数据了；否则，返回新数据，200的状态码，并且如果想要浏览器缓存数据的话，就在相应头中加入字段：  
Cache-Control:Max-age=2000  
响应数据又顺着应用层——传输层——网络层——网络层——传输层——应用层的顺序返回到网络进程  
13. 数据传输完成，TCP四次挥手断开连接。如果，浏览器或者服务器在HTTP头部加上如下信息，TCP就一直保持连接。保持TCP连接可以省下下次需要建立连接的时间，提示资源加载速度  
Connection:Keep-Alive  
14. 网络进程将获取到的数据包进行解析，根据响应头中的Content-type来判断响应数据的类型，如果是字节流类型，就将该请求交给下载管理器，该导航流程结束，不再进行；如果是text/html类型，就通知浏览器进程获取到文档准备渲染  
15. 浏览器进程获取到通知，根据当前页面B是否是从页面A打开的并且和页面A是否是同一个站点（根域名和协议一样就被认为是同一个站点），如果满足上述条件，就复用之前网页的进程，否则，新创建一个单独的渲染进程  
16. 浏览器会发出“提交文档”的消息给渲染进程，渲染进程收到消息后，会和网络进程建立传输数据的“管道”，文档数据传输完成后，渲染进程会返回“确认提交”的消息给浏览器进程  
17. 浏览器收到“确认提交”的消息后，会更新浏览器的页面状态，包括了安全状态、地址栏的 URL、前进后退的历史状态，并更新web页面，此时的web页面是空白页  
18. 渲染进程对文档进行页面解析和子资源加载，HTML 通过HTM 解析器转成DOM Tree（二叉树类似结构的东西），CSS按照CSS 规则和CSS解释器转成CSSOM TREE，两个tree结合，形成render tree（不包含HTML的具体元素和元素要画的具体位置），通过Layout可以计算出每个元素具体的宽高颜色位置，结合起来，开始绘制，最后显示在屏幕中新页面显示出来

# 七、浏览器渲染流程

相关概念：

1. 构建 DOM 树：这是因为浏览器无法直接理解和使用 HTML，所以需要将 HTML 转换为浏览器能够理解的结构。树状结构。

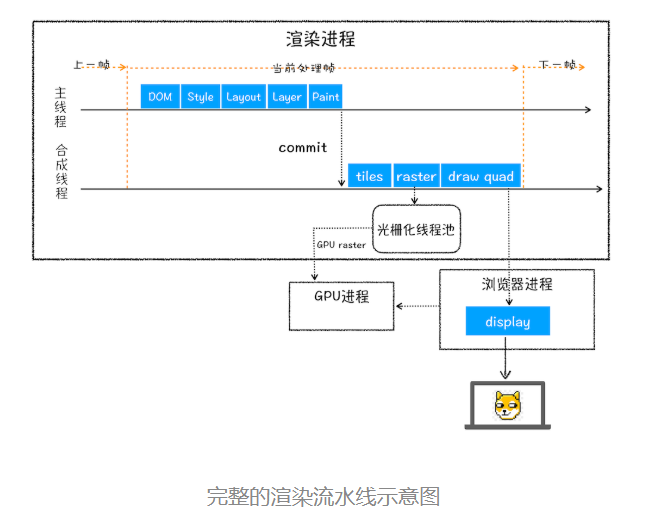
2. 样式计算（Recalculate Style）：样式计算的目的是为了计算出 DOM 节点中每个元素的具体样式，这个阶段大体可分为三步来完成：把 CSS 转换为浏览器能够理解的结构，styleSheets；转换样式表中的属性值，使其标准化；计算出 DOM 树中每个节点的具体样式。

3. 布局阶段：那么接下来就需要计算出 DOM 树中可见元素的几何位置，我们把这个计算过程叫做布局。布局阶段需要完成两个任务：创建布局树和布局计算。在显示之前，我们还要额外地构建一棵只包含可见元素布局树。有了一棵完整的布局树，就会计算布局树节点的坐标位置了。

4. 分层：因为页面中有很多复杂的效果，如一些复杂的 3D 变换、页面滚动，或者使用 z-indexing 做 z 轴排序等，为了更加方便地实现这些效果，渲染引擎还需要为特定的节点生成专用的图层，并生成一棵对应的图层树（LayerTree）。

5. 图层绘制:

6. 栅格化（raster）操作: 所谓栅格化，是指将图块转换为位图。而图块是栅格化执行的最小单位。



结合上图，一个完整的渲染流程大致可总结为如下：

1.渲染进程将 HTML 内容转换为能够读懂的 DOM 树结构。

2.渲染引擎将 CSS 样式表转化为浏览器可以理解的 styleSheets，计算出 DOM 节点的样式。

3.创建布局树，并计算元素的布局信息。

4.对布局树进行分层，并生成分层树。为每个图层生成绘制列表，并将其提交到合成线程。

5.合成线程将图层分成图块，并在光栅化线程池中将图块转换成位图。

6.合成线程发送绘制图块命令 DrawQuad 给浏览器进程。

7.浏览器进程根据 DrawQuad 消息生成页面，并显示到显示器上。

重排: 如果你通过 JavaScript 或者 CSS 修改元素的几何位置属性，例如改变元素的宽度、高度等，那么浏览器会触发重新布局，解析之后的一系列子阶段，这个过程就叫重排。无疑，重排需要更新完整的渲染流水线，所以开销也是最大的。

重绘: 比如通过 JavaScript 更改某些元素的背景颜色。如果修改了元素的背景颜色，那么布局阶段将不会被执行，因为并没有引起几何位置的变换，所以就直接进入了绘制阶段，然后执行之后的一系列子阶段，这个过程就叫重绘。相较于重排操作，重绘省去了布局和分层阶段，所以执行效率会比重排操作要高一些。

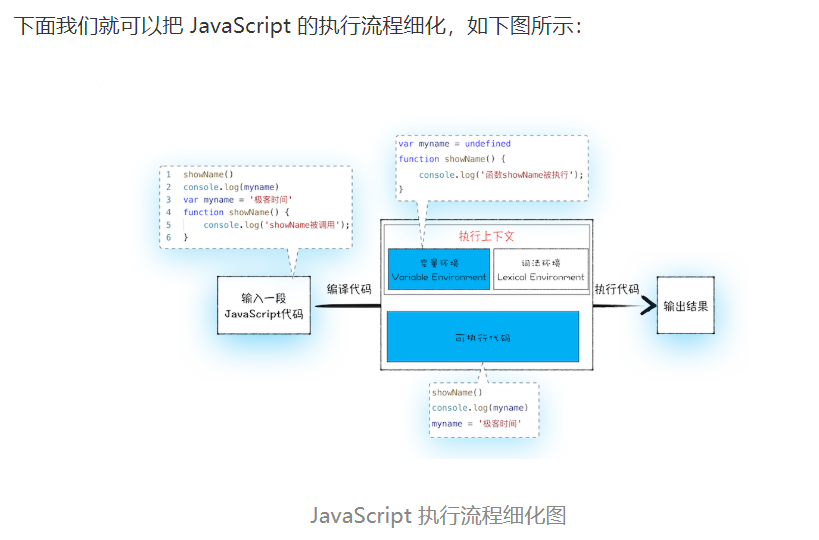
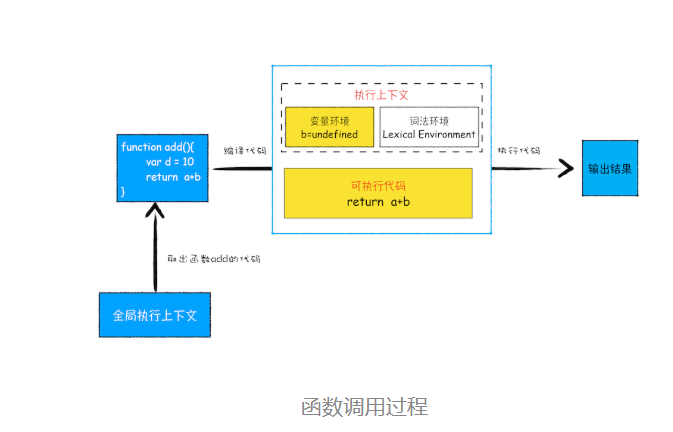
合成：更改一个既不要布局也不要绘制的属。渲染引擎将跳过布局和绘制，只执行后续的合成操作，我们把这个过程叫做合成。使用了 CSS 的 transform 来实现动画效果。

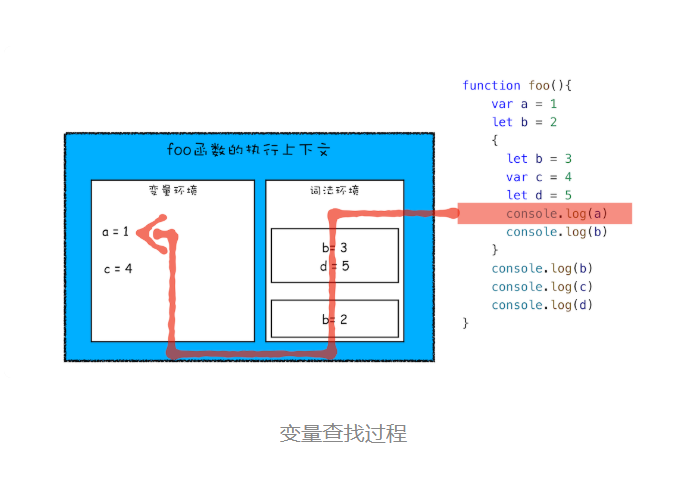
# 八、js执行机制

变量提升：是指在 JavaScript 代码执行过程中，JavaScript 引擎把变量的声明部分和函数的声明部分提升到代码开头的“行为”。变量被提升后，会给变量设置默认值，这个默认值就是我们熟悉的 undefined。

执行上下文：js编译后生成的一部分内容，是 JavaScript 执行一段代码时的运行环境，比如调用一个函数，就会进入这个函数的执行上下文，确定该函数在执行期间用到的诸如 this、变量、对象以及函数等。在执行上下文中存在一个变量环境的对象（Viriable Environment），该对象中保存了变量提升的内容。

JavaScript 的执行机制：先编译，再执行。JavaScript 代码执行过程中，需要先做变量提升，而之所以需要实现变量提升，是因为 JavaScript 代码在执行之前需要先编译。



var 声明的放在变量环境中，let，const声明的放在词法环境中，他们都属于执行上下文。

ES6 是通过变量环境和词法环境来同时支持变量提升和块级作用域。

·调用栈(执行上下文栈)：就是用来管理函数调用关系的一种数据结构。

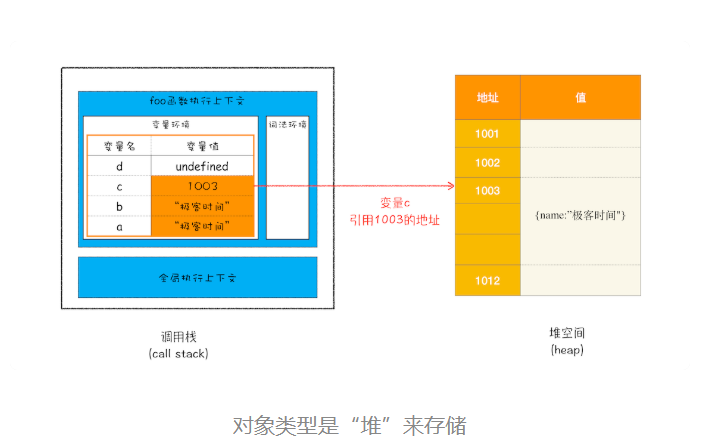
var的创建和初始化被提升，赋值不会被提升。  
let的创建被提升，初始化和赋值不会被提升，报错:。Cannot access xxxbefore initialization所以会形成一个暂时性死区。  
function的创建、初始化和赋值均会被提升。

JavaScript 语言的作用域链是由词法作用域决定的，而词法作用域是由代码结构来确定的。

this 是和执行上下文绑定的。

因为箭头函数没有自己的执行上下文，所以它会继承调用函数中的 this。

原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的。



# 九、消息队列与事假循环机制

消息队列：是一种数据结构，可以存放要执行的任务。它符合队列“先进先出”的特点，也就是说要添加任务的话，添加到队列的尾部；要取出任务的话，从队列头部去取。

通常我们把消息队列中的任务称为宏任务，每个宏任务中都包含了一个微任务队列，在执行宏任务的过程中，如果 DOM 有变化，那么就会将该变化添加到微任务列表中，这样就不会影响到宏任务的继续执行，因此也就解决了执行效率的问题。等宏任务中的主要功能都直接完成之后，这时候，渲染引擎并不着急去执行下一个宏任务，而是执行当前宏任务中的微任务，因为 DOM 变化的事件都保存在这些微任务队列中，这样也就解决了实时性问题。