

 skyline75489 / [what-happens-when-zh_CN](#)What-happens-when 的中文翻译, 原仓库 <https://github.com/alex/what-happens-when>

☆ 6.4k stars 🍴 733 forks

 Star Watch ▾

Code

Issues 1

Pull requests 1

Actions

Projects

Wiki

Security

Insights

 master ▾

...



skyline75489 ...

on 18 Jan 2018

[View code](#)

当...时发生了什么?

这个仓库试图回答一个古老的面试问题: 当你在浏览器中输入 google.com 并且按下回车之后发生了什么?

不过我们不再局限于平常的回答, 而是想办法回答地尽可能具体, 不遗漏任何细节。

这将是一个协作的过程, 所以深入挖掘吧, 并且帮助我们一起完善它。仍然有大量的细节等待着你来添加, 欢迎向我们发送 Pull Request!

这些内容使用 [Creative Commons Zero](#) 协议发布。

目录

- [按下"g"键](#)
- [回车键按下](#)
- [产生中断\[非USB键盘\]](#)
- [\(Windows\)一个 WM_KEYDOWN 消息被发往应用程序](#)
- [\(Mac OS X\)一个 KeyDown NSEvent被发往应用程序](#)
- [\(GNU/Linux\)Xorg 服务器监听键码值](#)
- [解析URL](#)
- [输入的是 URL 还是搜索的关键字?](#)

- [转换非 ASCII 的 Unicode 字符](#)
- [检查 HSTS 列表](#)
- [DNS 查询](#)
- [ARP 过程](#)
- [使用套接字](#)
- [TLS 握手](#)
- [HTTP 协议](#)
- [HTTP 服务器请求处理](#)
- [浏览器背后的故事](#)
- [浏览器](#)
- [HTML 解析](#)
- [CSS 解析](#)
- [页面渲染](#)
- [GPU 渲染](#)
- [Window Server](#)
- [后期渲染与用户引发的处理](#)

按下"g"键

接下来的内容介绍了物理键盘和系统中断的工作原理，但是有一部分内容却没有涉及。当你按下“g”键，浏览器接收到这个消息之后，会触发自动完成机制。浏览器根据自己的算法，以及你是否处于隐私浏览模式，会在浏览器的地址框下方给出输入建议。大部分算法会优先考虑根据你的搜索历史和书签等内容给出建议。你打算输入 "google.com"，因此给出的建议并不匹配。但是输入过程中仍然有大量的代码在后台运行，你的每一次按键都会使得给出的建议更加准确。甚至有可能在你输入之前，浏览器就将 "google.com" 建议给你。

回车键按下

为了从零开始，我们选择键盘上的回车键被按到最低处作为起点。在这个时刻，一个专用于回车键的电流回路被直接地或者通过电容器间接地闭合了，使得少量的电流进入了键盘的逻辑电路系统。这个系统会扫描每个键的状态，对于按键开关的电位弹跳变化进行噪音消除 (debounce)，并将其转化为键盘码值。在这里，回车的码值是13。键盘控制器在得到码值之后，将其编码，用于之后的传输。现在这个传输过程几乎都是通过通用串行总线(USB)或者蓝牙(Bluetooth)来进行的，以前是通过PS/2或者ADB连接进行。

USB键盘：

- 键盘的USB元件通过计算机上的USB接口与USB控制器相连接，USB接口中的第一号针为它提供了5V的电压
- 键码值存储在键盘内部电路一个叫做"endpoint"的寄存器内
- USB控制器大概每隔10ms便查询一次"endpoint"以得到存储的键码值数据，这个最短时间间隔由键盘提供

- 键值码值通过USB串行接口引擎被转换成一个或者多个遵循低层USB协议的USB数据包
- 这些数据包通过D+针或者D-针(中间的两个针), 以最高1.5Mb/s的速度从键盘传输至计算机。速度限制是因为人机交互设备总是被声明成"低速设备" (USB 2.0 compliance)
- 这个串行信号在计算机的USB控制器处被解码, 然后被人机交互设备通用键盘驱动进行进一步解释。之后按键的码值被传输到操作系统的硬件抽象层

虚拟键盘(触屏设备):

- 在现代电容屏上, 当用户把手指放在屏幕上时, 一小部分电流从传导层的静电域经过手指传导, 形成了一个回路, 使得屏幕上触控的那一点电压下降, 屏幕控制器产生一个中断, 报告这次"点击"的坐标
- 然后移动操作系统通知当前活跃的应用, 有一个点击事件发生在它的某个GUI部件上了, 现在这个部件是虚拟键盘的按钮
- 虚拟键盘引发一个软中断, 返回给OS一个"按键按下"消息
- 这个消息又返回来向当前活跃的应用通知一个"按键按下"事件

产生中断[非USB键盘]

键盘在它的中断请求线(IRQ)上发送信号, 信号会被中断控制器映射到一个中断向量, 实际上就是一个整型数。CPU使用中断描述符表(IDT)把中断向量映射到对应函数, 这些函数被称为中断处理器, 它们由操作系统内核提供。当一个中断到达时, CPU根据IDT和中断向量索引到对应的中断处理器, 然后操作系统内核出场了。

(Windows)一个 WM_KEYDOWN 消息被发往应用程序

HID把键盘按下的事件传送给 KBDHID.sys 驱动, 把HID的信号转换成一个扫描码(Scancode), 这里回车的扫描码是 VK_RETURN(0x0d)。KBDHID.sys 驱动和 KBDCLASS.sys (键盘类驱动, keyboard class driver)进行交互, 这个驱动负责安全地处理所有键盘和小键盘的输入事件。之后它又去调用 Win32K.sys, 在这之前有可能把消息传递给安装的第三方键盘过滤器。这些都是发生在内核模式。

Win32K.sys 通过 GetForegroundWindow() API函数找到当前哪个窗口是活跃的。这个API函数提供了当前浏览器的地址栏的句柄。Windows系统的"message pump"机制调用 SendMessage(hWnd, WM_KEYDOWN, VK_RETURN, lParam) 函数, lParam 是一个用来指示这个按键的更多信息的掩码, 这些信息包括按键重复次数(这里是0), 实际扫描码(可能依赖于OEM厂商, 不过通常不会是 VK_RETURN), 功能键(alt, shift, ctrl)是否被按下(在这里没有), 以及一些其他状态。

Windows的 SendMessage API直接将消息添加到特定窗口句柄 hWnd 的消息队列中, 之后赋给 hWnd 的主要消息处理函数 WindowProc 将会被调用, 用于处理队列中的消息。

当前活跃的句柄 hWnd 实际上是一个edit control控件, 这种情况下, WindowProc 有一个用于处理 WM_KEYDOWN 消息的处理器, 这段代码会查看 SendMessage 传入的第三个参数 wParam, 因为这个参数是 VK_RETURN, 于是它知道用户按下了回车键。

(Mac OS X)一个 KeyDown NSEvent被发往应用程序

中断信号引发了I/O Kit Kext键盘驱动的中断处理事件，驱动把信号翻译成键码值，然后传给 OS X的 WindowServer 进程。然后， WindowServer 将这个事件通过Mach端口分发给合适的（活跃的，或者正在监听的）应用程序，这个信号会被放到应用程序的消息队列里。队列中的消息可以被拥有足够高权限的线程使用 `mach_ipc_dispatch` 函数读取到。这个过程通常是由 NSApplication 主事件循环产生并且处理的，通过 NSEventType 为 KeyDown 的 NSEvent。

(GNU/Linux)Xorg 服务器监听键码值

☰ README.rst

码。当这个映射过程完成之后， X Server 把这个按键字符发送给窗口管理器(DWM, metacity, i3等等)，窗口管理器再把字符发送给当前窗口。当前窗口使用有关图形API把文字打印在输入框内。

解析URL

- 浏览器通过 URL 能够知道下面的信息：

- *Protocol* "**http**"
使用HTTP协议
- *Resource* **"/**"
请求的资源是主页(index)

输入的是 URL 还是搜索的关键字？

当协议或主机名不合法时，浏览器会将地址栏中输入的文字传给默认的搜索引擎。大部分情况下，在把文字传递给搜索引擎的时候，URL会带有特定的一串字符，用来告诉搜索引擎这次搜索来自这个特定浏览器。

转换非 ASCII 的 Unicode 字符

- 浏览器检查输入是否含有不是 `a-z` , `A-Z` , `0-9` , `-` 或者 `.` 的字符
- 这里主机名是 `google.com` , 所以没有非ASCII的字符；如果有的话，浏览器会对主机名部分使用 [Punycode](#) 编码

检查 HSTS 列表

- 浏览器检查自带的“预加载 HSTS（HTTP严格传输安全）”列表，这个列表里包含了那些请求浏览器只使用HTTPS进行连接的网站
- 如果网站在这个列表里，浏览器会使用 HTTPS 而不是 HTTP 协议，否则，最初的请求会使用HTTP协议发送

- 注意，一个网站哪怕不在 HSTS 列表里，也可以要求浏览器对自己使用 HSTS 政策进行访问。浏览器向网站发出第一个 HTTP 请求之后，网站会返回浏览器一个响应，请求浏览器只使用 HTTPS 发送请求。然而，就是这第一个 HTTP 请求，却可能会使用户受到 [downgrade attack](#) 的威胁，这也是为什么现代浏览器都预置了 HSTS 列表。

DNS 查询

- 浏览器检查域名是否在缓存当中（要查看 Chrome 当中的缓存，打开 `chrome://net-internals/#dns`）。
- 如果缓存中没有，就去调用 `gethostbyname` 库函数（操作系统不同函数也不同）进行查询。
- `gethostbyname` 函数在试图进行 DNS 解析之前首先检查域名是否在本地 Hosts 里，Hosts 的位置 [不同的操作系统有所不同](#)
- 如果 `gethostbyname` 没有这个域名的缓存记录，也没有在 `hosts` 里找到，它将会向 DNS 服务器发送一条 DNS 查询请求。DNS 服务器是由网络通信栈提供的，通常是本地路由器或者 ISP 的缓存 DNS 服务器。
- 查询本地 DNS 服务器
- 如果 DNS 服务器和我们的主机在同一个子网内，系统会按照下面的 ARP 过程对 DNS 服务器进行 ARP 查询
- 如果 DNS 服务器和我们的主机在不同的子网，系统会按照下面的 ARP 过程对默认网关进行查询

ARP 过程

要想发送 ARP（地址解析协议）广播，我们需要有一个目标 IP 地址，同时还需要知道用于发送 ARP 广播的接口的 MAC 地址。

- 首先查询 ARP 缓存，如果缓存命中，我们返回结果：目标 IP = MAC

如果缓存没有命中：

- 查看路由表，看看目标 IP 地址是不是在本地路由表中的某个子网内。是的话，使用跟那个子网相连的接口，否则使用与默认网关相连的接口。
- 查询选择的网络接口的 MAC 地址
- 我们发送一个二层（[OSI 模型](#) 中的数据链路层）ARP 请求：

ARP Request：

```
Sender MAC: interface:mac:address:here
Sender IP: interface.ip.goes.here
Target MAC: FF:FF:FF:FF:FF:FF (Broadcast)
Target IP: target.ip.goes.here
```

根据连接主机和路由器的硬件类型不同，可以分为以下几种情况：

直连:

- 如果我们和路由器是直接连接的, 路由器会返回一个 ARP Reply (见下面)。

集线器:

- 如果我们连接到一个集线器, 集线器会把 ARP 请求向所有其它端口广播, 如果路由器也“连接”在其中, 它会返回一个 ARP Reply。

交换机:

- 如果我们连接到了一个交换机, 交换机会检查本地 CAM/MAC 表, 看看哪个端口有我们要找的那个 MAC 地址, 如果没有找到, 交换机会向所有其它端口广播这个 ARP 请求。
- 如果交换机的 MAC/CAM 表中有对应的条目, 交换机会向有我们想要查询的 MAC 地址的那个端口发送 ARP 请求
- 如果路由器也“连接”在其中, 它会返回一个 ARP Reply

ARP Reply :

```
Sender MAC: target:mac:address:here
Sender IP: target.ip.goes.here
Target MAC: interface:mac:address:here
Target IP: interface.ip.goes.here
```

现在我们有了 DNS 服务器或者默认网关的 IP 地址, 我们可以继续 DNS 请求了:

- 使用 53 端口向 DNS 服务器发送 UDP 请求包, 如果响应包太大, 会使用 TCP 协议
- 如果本地/ISP DNS 服务器没有找到结果, 它会发送一个递归查询请求, 一层一层向高层 DNS 服务器做查询, 直到查询到起始授权机构, 如果找到会把结果返回

使用套接字

当浏览器得到了目标服务器的 IP 地址, 以及 URL 中给出来端口号 (http 协议默认端口号是 80, https 默认端口号是 443), 它会调用系统库函数 `socket`, 请求一个 TCP 流套接字, 对应的参数是 `AF_INET/AF_INET6` 和 `SOCK_STREAM`。

- 这个请求首先被交给传输层, 在传输层请求被封装成 TCP segment。目标端口会被加入头部, 源端口会在系统内核的动态端口范围内选取 (Linux下是 `ip_local_port_range`)
- TCP segment 被送往网络层, 网络层会在其中再加入一个 IP 头部, 里面包含了目标服务器的 IP 地址以及本机的 IP 地址, 把它封装成一个 IP packet。
- 这个 TCP packet 接下来会进入链路层, 链路层会在封包中加入 frame 头部, 里面包含了本地内置网卡的 MAC 地址以及网关 (本地路由器) 的 MAC 地址。像前面说的一样, 如果内核不知道网关的 MAC 地址, 它必须进行 ARP 广播来查询其地址。

到了现在, TCP 封包已经准备好了, 可以使用下面的方式进行传输:

- 以太网
- WiFi
- 蜂窝数据网络

对于大部分家庭网络和小型企业网络来说, 封包会从本地计算机出发, 经过本地网络, 再通过调制解调器把数字信号转换成模拟信号, 使其适于在电话线路, 有线电视光缆和无线电话线路上传输。在传输线路的另一端, 是另外一个调制解调器, 它把模拟信号转换回数字信号, 交由下一个 **网络节点** 处理。节点的目标地址和源地址将在后面讨论。

大型企业和比较新的住宅通常使用光纤或直接以太网连接, 这种情况下信号一直是数字的, 会被直接传到下一个 **网络节点** 进行处理。

最终封包会到达管理本地子网的路由器。在那里出发, 它会继续经过自治区域(autonomous system, 缩写 AS)的边界路由器, 其他自治区域, 最终到达目标服务器。一路上经过的这些路由器会从IP数据报头部里提取出目标地址, 并将封包正确地路由到下一个目的地。IP数据报头部 time to live (TTL) 域的值每经过一个路由器就减1, 如果封包的TTL变为0, 或者路由器由于网络拥堵等原因封包队列满了, 那么这个包会被路由器丢弃。

上面的发送和接受过程在 TCP 连接期间会发生很多次:

- 客户端选择一个初始序列号(ISN), 将设置了 SYN 位的封包发送给服务器端, 表明自己要建立连接并设置了初始序列号
- **服务器端接收到 SYN 包, 如果它可以建立连接:**
 - 服务器端选择它自己的初始序列号
 - 服务器端设置 SYN 位, 表明自己选择了一个初始序列号
 - 服务器端把 (客户端ISN + 1) 复制到 ACK 域, 并且设置 ACK 位, 表明自己接收到了客户端的第一个封包
- **客户端通过发送下面一个封包来确认这次连接:**
 - 自己的序列号+1
 - 接收端 ACK+1
 - 设置 ACK 位
- **数据通过下面的方式传输:**
 - 当一方发送了N个 Bytes 的数据之后, 将自己的 SEQ 序列号也增加N
 - 另一方确认接收到这个数据包 (或者一系列数据包) 之后, 它发送一个 ACK 包, ACK 的值设置为接收到的数据包的最后一个序列号
- **关闭连接时:**
 - 要关闭连接的一方发送一个 FIN 包
 - 另一方确认这个 FIN 包, 并且发送自己的 FIN 包
 - 要关闭的一方使用 ACK 包来确认接收到了 FIN

TLS 握手

- 客户端发送一个 `ClientHello` 消息到服务器端, 消息中同时包含了它的 Transport Layer Security (TLS) 版本, 可用的加密算法和压缩算法。
- 服务器端向客户端返回一个 `ServerHello` 消息, 消息中包含了服务器端的 TLS 版本, 服务器所选择的加密和压缩算法, 以及数字证书认证机构 (Certificate Authority, 缩写 CA) 签发的服务器公开证书, 证书中包含了公钥。客户端会使用这个公钥加密接下来的握手过程, 直到协商生成一个新的对称密钥
- 客户端根据自己的信任 CA 列表, 验证服务器端的证书是否可信。如果认为可信, 客户端会生成一串伪随机数, 使用服务器的公钥加密它。这串随机数会被用于生成新的对称密钥
- 服务器端使用自己的私钥解密上面提到的随机数, 然后使用这串随机数生成自己的对称主密钥
- 客户端发送一个 `Finished` 消息给服务器端, 使用对称密钥加密这次通讯的一个散列值
- 服务器端生成自己的 hash 值, 然后解密客户端发送来的信息, 检查这两个值是否对应。如果对应, 就向客户端发送一个 `Finished` 消息, 也使用协商好的对称密钥加密
- 从现在开始, 接下来整个 TLS 会话都使用对称密钥进行加密, 传输应用层 (HTTP) 内容

HTTP 协议

如果浏览器是 Google 出品的, 它不会使用 HTTP 协议来获取页面信息, 而是会与服务器端发送请求, 商讨使用 SPDY 协议。

如果浏览器使用 HTTP 协议而不支持 SPDY 协议, 它会向服务器发送这样的一个请求:

```
GET / HTTP/1.1
Host: google.com
Connection: close
[其他头部]
```

“其他头部”包含了一系列的由冒号分割开的键值对, 它们的格式符合 HTTP 协议标准, 它们之间由一个换行符分割开来。(这里我们假设浏览器没有违反 HTTP 协议标准的 bug, 同时假设浏览器使用 HTTP/1.1 协议, 不然的话头部可能不包含 `Host` 字段, 同时 `GET` 请求中的版本号会变成 `HTTP/1.0` 或者 `HTTP/0.9`。)

HTTP/1.1 定义了“关闭连接”的选项 `"close"`, 发送者使用这个选项指示这次连接在响应结束之后会断开。例如:

```
Connection:close
```

不支持持久连接的 HTTP/1.1 应用必须在每条消息中都包含 `"close"` 选项。

在发送完这些请求和头部之后, 浏览器发送一个换行符, 表示要发送的内容已经结束了。

服务器端返回一个响应码, 指示这次请求的状态, 响应的形式是这样的:

200 OK

[响应头部]

然后是一个换行，接下来有效载荷(payload)，也就是 `www.google.com` 的HTML内容。服务器下面可能会关闭连接，如果客户端请求保持连接的话，服务器端会保持连接打开，以供之后的请求重用。

如果浏览器发送的HTTP头部包含了足够多的信息（例如包含了 Etag 头部），以至于服务器可以判断出，浏览器缓存的文件版本自从上次获取之后没有再更改过，服务器可能会返回这样的响应：

304 Not Modified

[响应头部]

这个响应没有有效载荷，浏览器会从自己的缓存中取出想要的内容。

在解析完 HTML 之后，浏览器和客户端会重复上面的过程，直到HTML页面引入的所有资源（图片，CSS，favicon.ico等等）全部都获取完毕，区别只是头部的 `GET / HTTP/1.1` 会变成 `GET /$(相对www.google.com的URL) HTTP/1.1`。

如果HTML引入了 `www.google.com` 域名之外的资源，浏览器会回到上面解析域名那一步，按照下面的步骤往下一步一步执行，请求中的 `Host` 头部会变成另外的域名。

HTTP 服务器请求处理

HTTPD(HTTP Daemon)在服务器端处理请求/响应。最常见的 HTTPD 有 Linux 上常用的 Apache 和 nginx，以及 Windows 上的 IIS。

- HTTPD 接收请求
- **服务器把请求拆分为以下几个参数：**
 - HTTP 请求方法(`GET` , `POST` , `HEAD` , `PUT` , `DELETE` , `CONNECT` , `OPTIONS` , 或者 `TRACE`)。直接在地址栏中输入 URL 这种情况下，使用的是 `GET` 方法
 - 域名： `google.com`
 - 请求路径/页面： `/` (我们没有请求 `google.com` 下的指定的页面，因此 `/` 是默认的路径)
- 服务器验证其上已经配置了 `google.com` 的虚拟主机
- 服务器验证 `google.com` 接受 `GET` 方法
- 服务器验证该用户可以使用 `GET` 方法(根据 IP 地址，身份信息等)
- 如果服务器安装了 URL 重写模块（例如 Apache 的 `mod_rewrite` 和 IIS 的 `URL Rewrite`），服务器会尝试匹配重写规则，如果匹配上的话，服务器会按照规则重写这个请求

- 服务器根据请求信息获取相应的响应内容，这种情况下由于访问路径是 "/" ,会访问首页文件（你可以重写这个规则，但是这个是最常用的）。
- 服务器会使用指定的处理程序分析处理这个文件，假如 Google 使用 PHP，服务器会使用 PHP 解析 index 文件，并捕获输出，把 PHP 的输出结果返回给请求者

浏览器背后的故事

当服务器提供了资源之后（HTML，CSS，JS，图片等），浏览器会执行下面的操作：

- 解析 —— HTML，CSS，JS
- 渲染 —— 构建 DOM 树 -> 渲染 -> 布局 -> 绘制

浏览器

浏览器的功能是从服务器上取回你想要的资源，然后展示在浏览器窗口当中。资源通常是 HTML 文件，也可能是 PDF，图片，或者其他类型的内容。资源的位置通过用户提供的 URI(Uniform Resource Identifier) 来确定。

浏览器解释和展示 HTML 文件的方法，在 HTML 和 CSS 的标准中有详细介绍。这些标准由 Web 标准组织 W3C(World Wide Web Consortium) 维护。

不同浏览器的用户界面大都十分接近，有很多共同的 UI 元素：

- 一个地址栏
- 后退和前进按钮
- 书签选项
- 刷新和停止按钮
- 主页按钮

浏览器高层架构

组成浏览器的组件有：

- **用户界面** 用户界面包含了地址栏，前进后退按钮，书签菜单等等，除了请求页面之外所有你看到的内容都是用户界面的一部分
- **浏览器引擎** 浏览器引擎负责让 UI 和渲染引擎协调工作
- **渲染引擎** 渲染引擎负责展示请求内容。如果请求的内容是 HTML，渲染引擎会解析 HTML 和 CSS，然后将内容展示在屏幕上
- **网络组件** 网络组件负责网络调用，例如 HTTP 请求等，使用一个平台无关接口，下层是针对不同平台的具体实现
- **UI后端** UI 后端用于绘制基本 UI 组件，例如下拉列表框和窗口。UI 后端暴露一个统一的平台无关的接口，下层使用操作系统的 UI 方法实现
- **Javascript 引擎** Javascript 引擎用于解析和执行 Javascript 代码

- **数据存储** 数据存储组件是一个持久层。浏览器可能需要在本地存储各种各样的数据，例如 Cookie 等。浏览器也需要支持诸如 localStorage, IndexedDB, WebSQL 和 FileSystem 之类的存储机制

HTML 解析

浏览器渲染引擎从网络层取得请求的文档，一般情况下文档会分成8kB大小的分块传输。

HTML 解析器的主要工作是对 HTML 文档进行解析，生成解析树。

解析树是以 DOM 元素以及属性为节点的树。DOM是文档对象模型(Document Object Model)的缩写，它是 HTML 文档的对象表示，同时也是 HTML 元素面向外部(如Javascript)的接口。树的根部是"Document"对象。整个 DOM 和 HTML 文档几乎是一一对应的关系。

解析算法

HTML不能使用常见的自顶向下或自底向上方法来进行分析。主要原因有以下几点:

- 语言本身的“宽容”特性
- HTML 本身可能是残缺的，对于常见的残缺，浏览器需要有传统的容错机制来支持它们
- 解析过程需要反复。对于其他语言来说，源码不会在解析过程中发生变化，但是对于 HTML 来说，动态代码，例如脚本元素中包含的 document.write() 方法会在源码中添加内容，也就是说，解析过程实际上会改变输入的内容

由于不能使用常用的解析技术，浏览器创造了专门用于解析 HTML 的解析器。解析算法在 HTML5 标准规范中有详细介绍，算法主要包含了两个阶段：标记化（tokenization）和树的构建。

解析结束之后

浏览器开始加载网页的外部资源（CSS，图像，Javascript 文件等）。

此时浏览器把文档标记为可交互的（interactive），浏览器开始解析处于“推迟（deferred）”模式的脚本，也就是那些需要在文档解析完毕之后再执行的脚本。之后文档的状态会变为“完成（complete）”，浏览器会触发“加载（load）”事件。

注意解析 HTML 网页时永远不会出现“无效语法（Invalid Syntax）”错误，浏览器会修复所有错误内容，然后继续解析。

CSS 解析

- 根据 [CSS词法和句法](#) 分析CSS文件和 <style> 标签包含的内容以及 style 属性的值
- 每个CSS文件都被解析成一个样式表对象（StyleSheet object），这个对象里包含了带有选择器的CSS规则，和对应CSS语法的对象
- CSS解析器可能是自顶向下的，也可能是使用解析器生成器生成的自底向上的解析器

页面渲染

- 通过遍历DOM节点树创建一个“Frame 树”或“渲染树”，并计算每个节点的各个CSS样式值
- 通过累加子节点的宽度，该节点的水平内边距(padding)、边框(border)和外边距(margin)，自底向上的计算“Frame 树”中每个节点的首选(preferred)宽度
- 通过自顶向下的给每个节点的子节点分配可行宽度，计算每个节点的实际宽度
- 通过应用文字折行、累加子节点的高度和此节点的内边距(padding)、边框(border)和外边距(margin)，自底向上的计算每个节点的高度
- 使用上面的计算结果构建每个节点的坐标
- 当存在元素使用 floated，位置有 absolutely 或 relatively 属性时，会有更多复杂的计算，详见<http://dev.w3.org/csswg/css2/> 和 <http://www.w3.org/Style/CSS/current-work>
- 创建layer(层)来表示页面中的哪些部分可以成组的被绘制，而不用被重新栅格化处理。每个帧对象都被分配给一个层
- 页面上的每个层都被分配了纹理(?)
- 每个层的帧对象都会被遍历，计算机执行绘图命令绘制各个层，此过程可能由CPU执行栅格化处理，或者直接通过D2D/SkiaGL在GPU上绘制
- 上面所有步骤都可能利用到最近一次页面渲染时计算出来的各个值，这样可以减少不少计算量
- 计算出各个层的最终位置，一组命令由 Direct3D/OpenGL发出，GPU命令缓冲区清空，命令传至GPU并异步渲染，帧被送到Window Server。

GPU 渲染

- 在渲染过程中，图形处理层可能使用通用用途的 CPU，也可能使用图形处理器 GPU
- 当使用 GPU 用于图形渲染时，图形驱动软件会把任务分成多个部分，这样可以充分利用 GPU 强大的并行计算能力，用于在渲染过程中进行大量的浮点计算。

Window Server

后期渲染与用户引发的处理

渲染结束后，浏览器根据某些时间机制运行JavaScript代码(比如Google Doodle动画)或与用户交互(在搜索栏输入关键字获得搜索建议)。类似Flash和Java的插件也会运行，尽管Google主页里没有。这些脚本可以触发网络请求，也可能改变网页的内容和布局，产生新一轮渲染与绘制。

Releases

No releases published

Packages

No packages published

Contributors 10

