Chrome 的历史和指导原则

Chrome 在 2008 年的下半年发布了第一个运行在 Windows 环境的 Beta 版本，如今它已成为最受欢迎的浏览器，占据 35%的市场份额。

Chrome 开发过程中的指导原则：

1. Speed

   让浏览器更快

1. Security

   给用户提供最安全的环境

1. Stability

   提供一个弹性的、稳定的 Web 应用平台

1. Simplicity

   把精细的技术包装在简单的用户体验中

性能的多个层面

当今的浏览器是一个平台，如同你的操作系统，在运行着各种应用。Chrome 正是这样设计的。

在 Chrome 之前，大多数浏览器建立在单个进程之上，所有标签共享内存空间并且彼此之间进行资源的竞争。当某个标签或者是浏览器出现 bug 时，可能整个浏览器就奔溃了。

与此相反，Chrome 运行在多进程模式，每个标签运行在一个轻量级的沙盒中，在进程上和内存上都是彼此隔离的。在飞速发展的多核时代，这个架构带来性能的显著提升。

一个 Web 程序主要面临三个问题：“获取资源和页面布局”，“渲染”和“执行 JavaScript”。

渲染和脚本遵循单线程，交错执行的模型，因为不可能并发地修改 DOM。其中的部分原因是 JavaScript 本身是一门单线程的语言。

因此，如何让渲染和脚本的执行一起进行，对于应用开发者和浏览器开发者来说都极其重要。

在渲染方面，Chrome 使用 Blink——一个快速、开源、标准兼容的布局引擎；

在 JavaScript 方面，Chrome 自己优化了 V8 Javascript runtime，并且发布为一个独立的的开源项目。

然而这两方面的优化在当浏览器在网络上发生阻塞，等待资源到达时就表现不出效果。

一个浏览器的优化网络资源的的顺序、优先级和延迟的能力是决定用户体验的最重要的因素之一。

你可能没有意识到，但是 Chrome 的网络协议栈确确实实在一天比一天变得更加聪明，尽力去减少请求资源的延迟。

网络协议栈的学习方式就像 DNS 查询，它记住网络的拓扑，提前连接到可能要访问的目标。

从外部看来，它给自己提供了一个简单的资源获取机制；从内部看来，它是一个精细的、吸引人的案例，研究如何优化网络性能和带给用户最好的体验。

当今的 Web 应用是什么？

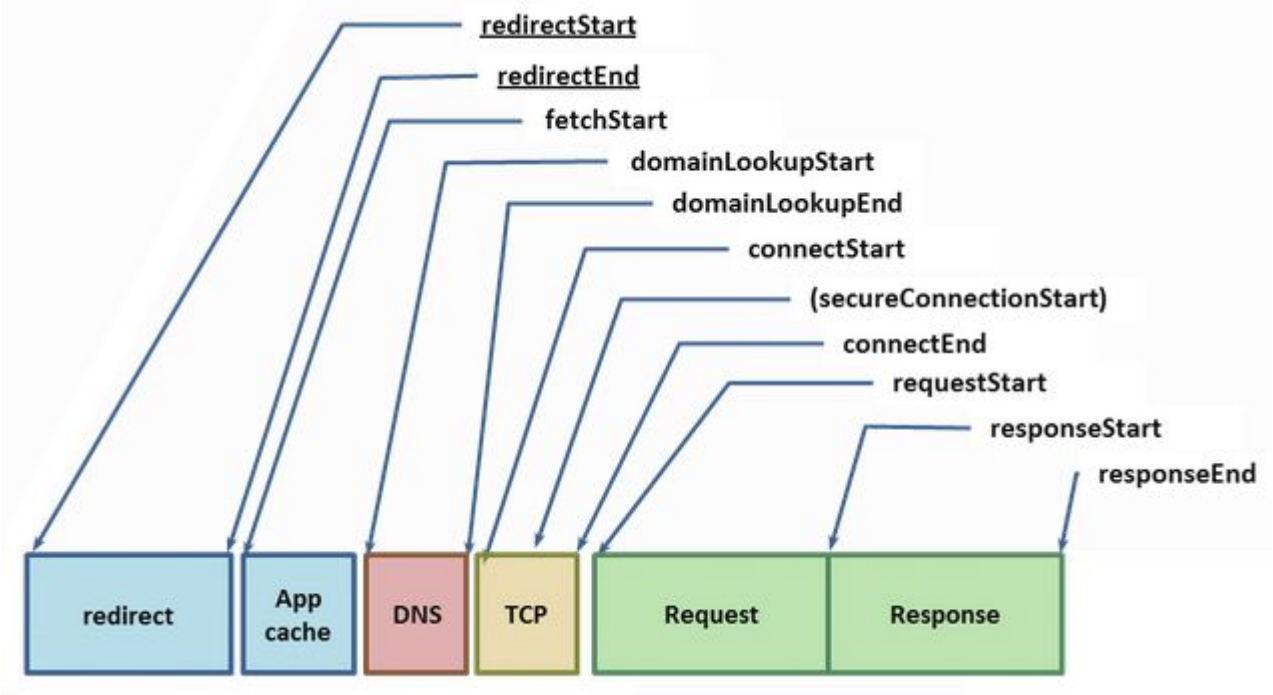
HTTP Archive 这个项目定期地爬取知名的网站的每个网址来记录和总体分析它们使用的资源，content type，headers 和其他元数据。

2013 年一月的对 300，000 个网址的统计结果如下：

- 平均大小为 1280 KB
- 平均由 88 个资源组成
- 平均连接 15 个以上的主机

我们可以计算出每个资源的平均大小为 15 KB，这意味着浏览器中大多数的网络传输是又短又急。这体现了其自身的复杂性，因为 TCP 擅长量大、流式的下载。

一次资源请求的过程



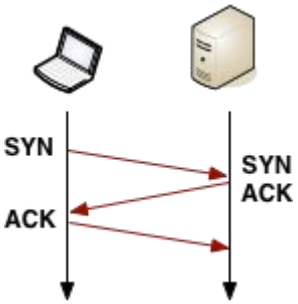
给出一个网络中资源的 URL 后，浏览器从检查本地缓存和应用缓存开始。如果你之前已经获取过这个资源并且给出了合适的资源缓存头部（例如 Expires 和 Cache-Control），我们可能可以用本地的副本来填充这次请求（最快的请求就是不进行请求）。或者，我们不得不检验一下资源，如果它过期了，或者之前没有获取过，那么必须发起一个网络请求来获取资源。

给出一个主机名和资源路径，Chrome 首先检查是否存在{schema，host，port}一致的可以复用的连接。或者，你配置了一个代理，或给定了一个 PAC（proxy auto-config）脚本，Chrome 会检查是否存在可以复用的，经过正确代理的连接。PAC 脚本允许基于 URL 的使用不同的代理，或者其他给定的规则，这两种情况都能有各自的连接池。如果上述的条件都不匹配，那么请求必须从 DNS 解析开始。

如果我们很幸运，主机名的解析结果已经在缓存中，这时我们只需要进行一个系统调用。否则，在其他工作开始之前我们必须发起一个 DNS 请求。

DNS 解析的时常与“你所使用的网络提供商”、“网站的热门程度”、”主机名存在于中间缓存的可能性”和"权威服务器的响应时间"相关。

一个 DNS 解析耗费几百毫秒是很寻常的事情。



拿到解析出来的 IP 地址后，Chrome 终于能够打开一个 TCP 连接到目的地址，这意味着我们必须进行三次握手：“SYN > SYN ACK > ACK”。这个过程可能会在应用数据开始传输之前，造成几十、几百甚至几千毫秒的延迟。延迟的大小由服务端和客户端的距离、和所选择的路由路径决定。

一旦三次握手完成，如果我们连接到一个使用 HTTPS 的地址，我们必须进行 SSL 握手。这时会增加两个来回的延迟。如果 SSL 回话被缓存起来，我们可以只增加一个来回的延迟。

终于，Chrome 能够开始发送 HTTP 请求了。服务器接收到请求后，进行处理，并把返回数据给客户端。然后，我们完成了这次请求，除非这时一个 redirect 响应，这时我们必须重复一次整个流程。

为了说明问题，我们设想一个最坏的情况，本地缓存命中失败，进行 DNS 查询，进行 TCP 握手，进行 SSL 握手，客户端发出请求到服务器，服务器处理请求，服务器返回结果：

- 50ms，用于 DNS 查询
- 80ms，用于 TCP 握手（1 RTT）
- 160ms，用于 SSL 握手（2 RTTs）
- 40ms，用于向服务器发出请求
- 100ms，用于服务器处理请求
- 40ms，用于用于向客户端返回结果

总计 470ms，这意味着一个请求中超过 80%的时间开销并非用于请求的处理。由此看来，我们有很多工作要做。

事实上，470ms 是一个乐观的估算结果：

- 如果服务器的响应结果比初始的 TCP 拥塞窗口（4KB~15KB）大，会引入额外一个或多个来回的开销。
- SSL 延迟可能更严重。如果我们需要去获取一个缺少的证书或进行一次在线证书状态检查（OCSP），那么我们需要另一个完整的 TCP 连接，这可能会带来几百或几千毫秒的额外开销。

怎么样才足够快？

DNS、握手和来回的时间开销在我们之前的案例中决定着总的请求时间。这些延迟的影响真的很大吗？是的。

用户体验研究 依据对用户对应用（包括实时和离线）延迟的感受的研究，总结出一张表格：

Table 1.1 – User perception of latency

Delay	User Reaction
0 - 100 ms	Instant
100 - 300 ms	Small perceptible delay
300 - 1000 ms	Machine is working
1 s+	Mental context switch
10 s+	I’ ll come back later…

Table1.1 解析了一个在网络性能社区中的非官方的规则 :在 250ms 的可见延迟内渲染出你的页面，从而保持用户的吸引力。通过对 Google，Amazon 和众多其他的网站的研究可以看出，额外的延迟会直接地影响你的网站的 bottom line（底线？？），更快的网站带来更多的页面访问，更强的用户吸引力和更高的转化率。

对于大多数的用户甚至是网络应用开发者来说，DNS，TCP，SSL 所带来的延迟是发生在网络层的，透明的。然而，这些步骤对用户体验的影响却很严重，因为每一个额外的网络请求可以增加几十或几百毫秒的延迟。这也正是 Chrome 的网络协议栈如此重要的原因，让我们来深入它的实现细节。

Chrome's Network Stack from 10,000 Feet

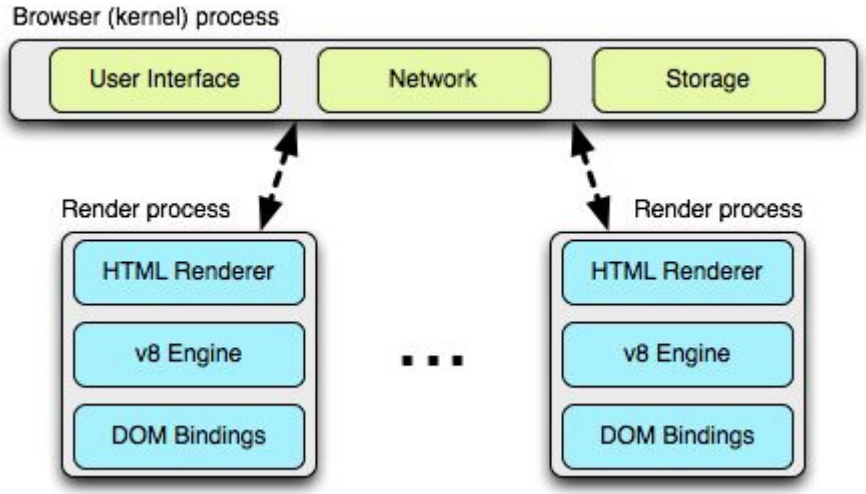
多进程架构

Chrome 的多进程架构对每个网络请求在浏览器中如何处理来说有着重要的意义。

Chrome 支持四种不同的执行模型来决定进程分配是如何进行的。

默认情况下，桌面版 Chrome 使用“一个网站一个进程”的模型，这个模型隔离不同的站点，但是把相同站点的的实例聚集在一个进程中。

为了使事情简单，我们想象一个最简单的场景：一个标签一个进程的情况。在网络性能方面，这两个模型的区别并不明显，但是“一个标签一个进程”模型更容易理解。



如图若示，每个渲染进程包含 Blink 布局引擎，V8 JavaScript 引擎和连接这两个组件的代码。

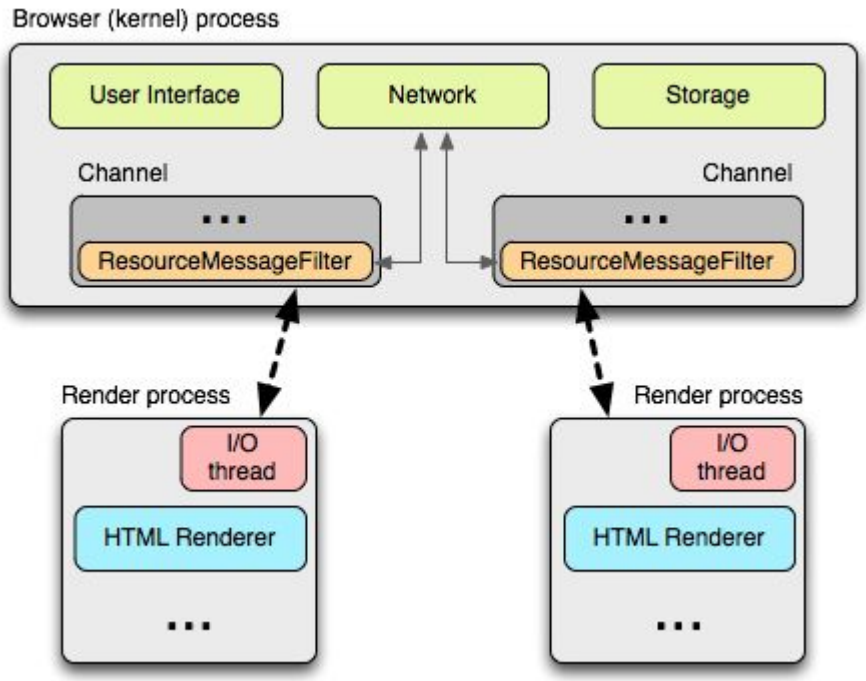
每一个渲染进程运行在一个沙盒环境里，对用户计算机和网络的访问权很有限。

为了获得这些资源，每个渲染进程需要和浏览器进程通信，这就能对渲染进程施加安全策略。

进程间通信（IPC）和多进程资源加载

在 Chrome 中，浏览器进程和渲染进程的通信由 IPC 完成。在 Linux 和 OS X 上，socketpair( ) 用来提供一个命名管道来传输异步通信的消息。

从渲染进程来的每一个消息都被序列化后传递给 I/O 现成，I/O 线程把消息发送给浏览器进程。在接收端，浏览器进程提供一个过滤接口，允许 Chrome 拦截需要交给网络协议栈处理的资源 IPC 请求。



这个架构的一个优点是，所有的资源请求在 I/O 线程处理，UI 的生成和网络事件的处理不会相互影响。资源过滤器在浏览器进程的 I/O 线程运行，拦截资源请求消息，转发给浏览器进程中的 ResourceDispatcherHost 单例。

使用单例允许浏览器控制每个渲染进程对网络的访问，使资源能够高效、一致地共享。一些例子：

- socket 池和连接限制：浏览器能够在每一个 profile（？？）、代理、{schema，host，port}的组合上的 socket 打开数施加限制。请注意，这样允许在一个{host，port}组合上最多有 6 个 http 连接和 6 个 https 连接。（why？？？？）

- socket 重用：持续的 TCP 连接被保留在 socket 池中，为了能够被重用，避免额外的在 DNS，TCP 和 SSL 上的开销。
- socket 延迟绑定：当 socket 准备发送应用的请求时，才让请求和一个 TCP 连接关联起来。
- 一致的会话状态：身份验证、cookie 和缓存数据被多个缓存进程共享。
- 全局资源和网路优化：浏览器能够在渲染进程中作出决定，让一些请求拥有更高的优先级。例如给当前标签的请求更高的网络优先级。
- 预测优化：通过监控所有的网络流量，Chrome 能够建立和提炼出一个预测模型来提高性能。

对于渲染进程来说，它只要简单地通过 IPC 发送一个资源请求消息给浏览器进程，这个消息被标记上一个唯一的 ID，然后浏览器进程就会接管剩下的工作。

跨平台资源获取

Chrome 网络协议栈实现的一个主要的担忧是如何实现在不同平台上的可移植性。

为了应对这一挑战，网络协议栈作为一个几乎是单线程的跨平台库来实现，这样就允许 Chrome 在不同平台上复用相同的基础设施，提供相同的性能优化和更大的优化可能性。

所有的网络协议栈代码，是开源的，并且能在 src/net（https://code.google.com/p/chromium/codesearch#chromium/src/net/&ct=rc&cd=1&q=src.net&sq=package:chromium）目录下找到。我们不会详述各个组件，但是代码的组织方式会告诉你它的功能和结构，其中一部分列举在表 1.2 中：

Table 1.2 - Components of Chrome

Component	Description
net/android	Bindings to the Android runtime
net/base	Common net utilities, such as host resolution, cookies, network change detection, and SSL certificate management
net/cookies	Implementation of storage, management, and retrieval of HTTP cookies
net/disk_cache	Disk and memory cache implementation for web resources
net/dns	Implementation of an asynchronous DNS resolver
net/http	HTTP protocol implementation
net/proxy	Proxy (SOCKS and HTTP) configuration, resolution, script fetching, etc.
net/socket	Cross-platform implementations of TCP sockets, SSL streams, and socket pools
net/spdy	SPDY protocol implementation
net/url_request	URLRequest, URLRequestContext, and URLRequestJob implementations
net/websockets	WebSockets protocol implementation

在手机平台上的架构和性能

与桌面环境相比，手机环境有更多的限制，和很多基本的操作上的差异：

- 桌面用户使用鼠标来导航，可能有重叠的窗口，大屏幕，几乎没有电量限制，通常有稳定的网络资源和更多的存储空间和内存。
- 手机用户使用触摸和和手势来导航，小屏幕，电池和电量的限制，通常对网络进行计量，拥有有限的存储空间和内存。

在未来，不会有“典型的手机设备”这一回事。取而代之的是拥有不同硬件能力的设备。为了提供最好的性能，Chrome 必须适应每一种和每一个设备的限制。

在 Android 设备，Chrome 使用与桌面版本相同的多进程架构。不同的是，由于手机设备内存的限制，Chrome 可能不能为每一个标签运行一个专用的渲染进程。Chrome 依据可用的内存和其他设备上的限制来决定最佳的渲染进程数量，让多个标签共享一个渲染进程。

万一只有很少的资源可用，或者 Chrome 没有办法运行多个进程，它能切换到单进程，多线程处理模式。事实上，在 IOS 设备，由于该平台上运行沙盒的限制，Chrome 就是使用这种模式运行的。

网络性能又如何呢？首先，在 Android 和 IOS 上，Chrome 使用与其他版本相同的网络协议栈。这使得所有的网络优化策略能在不同平台上使用，这是 Chrome 一个重要的性能优势。不同的是，预测优化技术的优先级、socket 超时时长、管理逻辑、缓存大小等会因为所使用的设备和网络的性能而进行调整。

举例来说，为了保持电量，手机的 Chrome 可能会选择延迟关闭空闲的 socket。仅当要打开新的 socket 时才去关闭空闲的 socket，从而最小化 radio（？？？）的使用。类似地，预渲染会要求网络和处理资源，这个功能通常是在 Wifi 情况下才可用。

Chrome Predictor 之预测优化

Chrome 会随着你的使用而变得越来越快。这归功于与浏览器进程一起初始化的单例的 Predictor 实例，它负责观察网络模式，并且学习和预测用户未来的动作。举一些例子：

- 用户把鼠标放在链接上面预示着可能进行点击，这时 Chrome 就能通过提前对域名进行进行 DNS 查询和进行 TCP 握手来加速。当用户点击时，平均花了 200ms 的时间，这个时候我们已经完成了 DNS 查询和 TCP 握手的工作，为导航事件减少了数百毫秒的延迟。
- 当用户地址栏输入内容并触发推荐高相似的链接的时候，Chrome 可能会开始进行 DNS 查询，TCP 预连接，甚至在一个隐藏的标签开始渲染。
- 我们每个人都有自己非常喜欢并且每天都要访问的网站。Chrome 会学习这些网站包含的资源，预测性地进行提前解析，甚至可能通过提前加载资源来加快浏览体验。
- Chrome 会学习网络的拓扑，和你自己的浏览模式。如果这项工作做的好，它可以为每次导航减少数百毫秒的延迟，让用户体检到页面几乎是立即加载完毕的。为了达到这个目标，Chrome 利用 表 1.3 中列出的四项核心优化技术

Table 1.3 - Network optimization techniques used by Chrome

Technique	Description

Table 1.3 - *Network optimization techniques used by Chrome*

Technique	Description
DNS pre-resolve	Resolve hostnames ahead of time, to avoid DNS latency
TCP pre-connect	Connect to destination server ahead of time, to avoid TCP handshake latency
Resource prefetching	Fetch critical resources on the page ahead of time, to accelerate rendering of the page
Page prerendering	Fetch the entire page with all of its resources ahead of time, to enable instant navigation when triggered by the user

每一个利用这一个或多个这些技术的决定都会面临极大的限制。毕竟 ,每一项都是预测性的优化，这意味着如果预测得不好，可能会触发无意义的工作和浪费网络流量。更糟糕的是，这可能会带来更长的加载时间，给用户带来更糟糕的用户体验。

Chrome 是如何解决这个问题的呢？Predictor 消费尽可能多的信号，包括用户产生的动作，历史浏览数据和从渲染器和网络协议栈本身来的信号。

与在 Chrome 中负责协调网络活动的 ResourceDispatcherHost 没有什么不同的是 ,Predictor 在用户和网络产生的活动上创建了许多过滤器：

- IPC channel filter，用来监控渲染进程产生的信号。

- ConnectIntercepter 对象被添加到每个请求中，用来观察每个请求的网络模式和记录成功指标。

作为一个深入的例子，渲染进程能够向浏览器进程触发如下暗示动作的任意一种消息，这些消息定义在 ResourceMotivation（url_info.h）：

```
enum ResolutionMotivation {
  MOUSE_OVER_MOTIVATED,     // Mouse-over initiated by the user.
  OMNIBOX_MOTIVATED,        // Omnibox suggested resolving this.
  STARTUP_LIST_MOTIVATED,   // This resource is on the top 10 startup list.
  EARLY_LOAD_MOTIVATED,     // In some cases we use the prefetcher to warm up
                            // the connection in advance of issuing the real
                            // request.

  // The following involve predictive prefetching, triggered by a navigation.
  // The referring_url_ is also set when these are used.
  STATIC_REFERAL_MOTIVATED,  // External database suggested this resolution.
  LEARNED_REFERAL_MOTIVATED, // Prior navigation taught us this resolution.
  SELF_REFERAL_MOTIVATED,    // Guess about need for a second connection.

  // <snip> ...
};
```

给出这样的一个信号后，Predictor 的目标就是提高预测成功的可能性，然后在资源可用的情况下触发相应的活动。每一个暗示动作都会有一个成功率，优先级，和过期的时间戳，这些数据组合在一起用来维护一个用于预测优化的内部优先级队列。最后，每一个从这个队列发出的请求，Predictor 能够追踪它的成功率，用来优化未来的决策。

Chrome 网络架构总结：

- 使用多进程架构，将渲染进程和浏览器进程隔离。
- 维护一个单实例的资源分发器，运行在浏览器进程，被渲染进程共享使用。
- 网络协议栈是一个跨平台，几乎单线程的库。
- 网络协议栈使用非阻塞的操作来处理所有的网络操作。
- 使用共享的网络协议栈能够带来高效的资源优先级，资源重用，并且使浏览器能在多个进程中实现全局优化。
- 每个渲染进程通过 IPC 与资源分发器通信。
- 资源分发器通过一个定制的 IPC 过滤器来拦截资源请求。
- Predictor 通过拦截资源的请求和响应来学习，并优化未来的资源请求。
- Predictor 基于流量模型的学习，通过预测来规划 DNS，TCP，甚至是资源的请求，这节省了用户导航时数百毫秒的时间。

浏览器会话的生命周期

在了解 Chorme 的网络协议栈之后，让我们深入了解一下在面向用户方面的优化。假设我们刚刚创建了一个新的 Chrome 配置文件并且准备开始浏览网页。

优化冷启动的体验

第一次启动你的浏览器的时候，Chrome 几乎不知道你喜欢哪些网站和你的浏览模式。然而，大多数人会浏览自己的邮箱、最喜欢的新闻网站、社交网站、门户网站等。具体访问哪个网站因人而异，但是这些会话的相似性让 Chrome 的 Predictor 能够加速你的冷启动体验。

Chrome 记住用户打开浏览器时最有可能访问的前十个网站（这十个网站并不是全球最热门的前十个网站）。当 Chrome 加载的时候，它可以触发对这些网站提前进行 DNS 解析。如果你很感兴趣，你可以通过打开一个新的标签，浏览“chrome://dns"来查看启动加载域名列表。

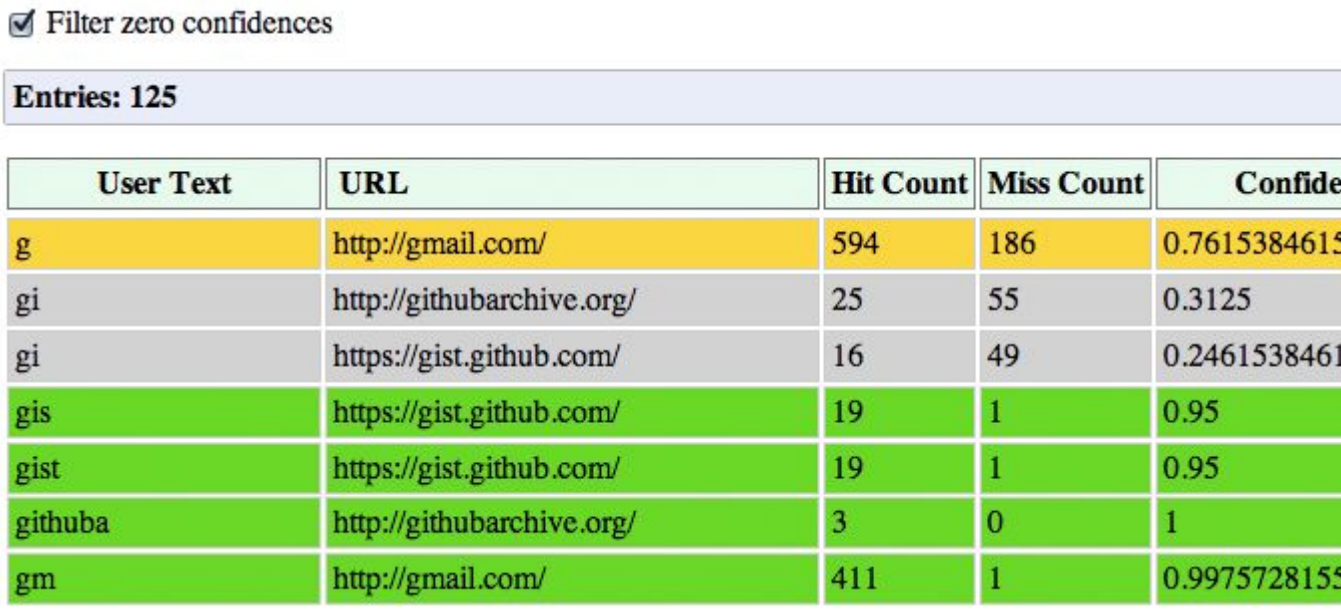
Future startups will prefetch DNS records for 10 hostnames

Host name	How long ago (HH:MM:SS)	Motivation
http://www.google-analytics.com/	15:31:33	n/a
https://a248.e.akamai.net/	15:31:30	n/a
https://csi.gstatic.com/	15:31:16	n/a
https://docs.google.com/	15:31:18	n/a
https://gist.github.com/	15:31:34	n/a
https://lh6.googleusercontent.com/	15:31:16	n/a
https://secure.gravatar.com/	15:31:29	n/a
https://ssl.google-analytics.com/	15:31:29	n/a
https://ssl.gstatic.com/	15:31:16	n/a
https://www.google.com/	15:31:16	n/a

优化地址栏的交互

地址栏的引入是 Chorme 的一项创新，不像它的前身不仅仅处理 URL 地址。除了记住你过去浏览的页面的 URL 地址，地址栏还提供历史浏览记录的全文搜索，以及轻量集成你所选择的搜索引擎的搜索结果。

当用户输入的时候，地址栏会自动提示，这个提示要么是一个基于你浏览历史的 URL，要么是一个搜索词条。在底层，每一个提示都会根据过去的表现来得分。事实上，Chrome 允许我们通过访问“Chrome://predictor”来查看这些数据。



☑ Filter zero confidences

Entries: 125

User Text	URL	Hit Count	Miss Count	Confide
g	http://gmail.com/	594	186	0.761538461
gi	http://githubarchive.org/	25	55	0.3125
gi	https://gist.github.com/	16	49	0.246153846
gis	https://gist.github.com/	19	1	0.95
gist	https://gist.github.com/	19	1	0.95
githuba	http://githubarchive.org/	3	0	1
gm	http://gmail.com/	411	1	0.997572815

Chrome 会维护一份用户输入前缀、提示的动作和命中率的历史纪录。以我的个人数据来说，当我在地址栏输入“g”时，我有 76%的概率会前往 Gmail。当我继续输入“m”时，置信度会达到 99.8%。事实上，在 412 次访问记录中，我只有一次在输入“gm”后没有前往 Gmail。

这跟网络协议栈有什么关系呢？黄色和绿色的可能的候选网址对 ResourceDispatcher 来说是一非常重要的信号。当我可能前往一个黄色的候选网址时，Chrome 可能会预先触发一个 DNS 查询。如果我可能前往一个绿色的候选网址，Chrome 可能还会在域名解析完成后提前打开一

个 TCP 连接。最后，如果两个动作都完成了而用户还在犹豫的时候，Chrome 甚至可能会在一个隐藏的标签渲染整个页面。

也许，基于你的搜索记录并没有找到一个结果与输入前缀匹配，这时 Chrome 可能会向你的搜索引擎发起 DNS 查询和 TCP 连接，来期望用户进行搜索请求。

平均起来，用户会花数百毫秒的时间来输入完整个查询和评估自动补全的建议是否满意。在后台，Chrome 能够趁这个时间预先进行 DNS 解析，打开 TCP 连接，甚至会预先渲染页面，因此当用户准备按下回车键时，大部分网路延迟已经消除了。

优化缓存性能

最好、最快的请求，就是一个不用执行的请求。当谈论性能的时候，我们就一定要谈到缓存。你有在你网页的所有资源都提供 Expires，ETag，Last-Modified 和 Cache-Control 这些头部吧？如果没有，快点修复这个问题。

Chrome 的内部缓存有两种不同的实现：一种是备份在本地磁盘中，另一种保存在内存中。基于内存的实现方式是用在匿名浏览模式中，并且当你关掉窗口的时候会被清空。两种方式都实现了相同的内部接口（“disk_cache::Backend”和“disk_cache::Entry”），极大地简化了架构，并让你能够简单地试验你自己的缓存实现方式。

在内部，基于磁盘的缓存实现了它自己的一套数据结构，所有数据都保存在一个单一的缓存文件夹里。在文件夹里，有在浏览器启动时进行内存映射的索引文件，和保存实际数据、HTTP 头部和其他统计信息。

如果你对 Chrome 的缓存状态感兴趣，你可以打开一个标签并浏览“chrome://net-internals/#httpCache”。或者，你想看确切的 HTTP 元数据和缓存的响应，你还可以访问"chrome://cache"，它会列举所有的现在能够访问的缓存资源。你可以从那个页面搜索一个你在寻找的资源并点击 URL 去查看实际被缓存的头部和响应的字节数据。

通过预获取来优化 DNS

我们已经在几个场景提到 DNS 预解析。所以在我们深入了解实现细节之前，让我们回顾一下哪些情况下会被触发和为什么触发：

- 运行在渲染进行测文档解释器——Blink，会提供整个页面的所有链接的域名列表，让 Chrome 来选择去提前解析。
- 当用户把鼠标放在链接上面或者按下的时候，渲染进程会触发一个事件表示用户可能会访问这个地址。
- 地址栏会基于高可能性的推荐来触发一个解析。
- Predictor 会基于过去的浏览和资源请求记录来请求提前解析域名。
- 网页的拥有者可能会明确地向 Chrome 生命哪些域名需要提前解析。

在所有的情况中，DNS 预解析都被当作一个暗示。Chrome 并没有保证预解析会发生，而是用它自己的 poredictor 来评估这些信号并决定行动的方向。在最好情况下，如果 Chrome 不能及时准时地解析完域名，用户必须等待 DNS 解析，紧接着一个 TCP 连接和实际的资源的获取。然而，当这些发生后，predictor 会做记录并调整未来的决策。因此，Chrome 会随着你的使用越快，越智能。

我们之前没有提到的一种优化是 Chrome 学习每个网站的拓扑并利用这些信息来加速之后的访问的能力。特别地，再提一次，每个网页平均包含 88 个资源，15 个不同的域名。每次你浏览的时候，Chrome 为页面上的热门资源记录下域名，在之后访问时，它可能会选择去为某些或全部域名触发 DNS 预解析和预先打开 TCP 连接。

为了查看 Chrome 保存的子资源域名，可以访问“chrome://nds”并搜索一个网址的热门资源域名。在上面的例子中，你可能会看 Chrome 为 Google+记住了 6 条子资源域名和统计 DNS 预解析或 TCP 提前连接被触发的次数，和每次会有多少个请求用到这个域名。

除了所有的内部信号 ,网站的拥有着还能够在页面中红嵌入额外的标记来请求浏览器提前对域名进行解析。

```
<link rel="dns-prefetch" href="//host_name_to_prefetch.com">
```

为什么要这么做呢？有些情况下，你可能想要提前解析一个在页面中没有提及的域名。重定向就是一个标准的例子：一个链接可能会指向一个域名（可能是一个分析追踪服务），然后把用户重

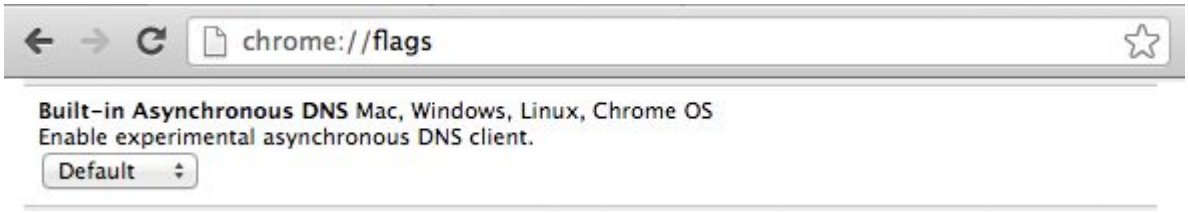
定向到确切的目的地。Chrome 自己不能推断出这样的模式，但是你可以通过提供一个人工的暗示来帮助浏览器提前解析最终目的地址的域名。

在底层这是怎么实现的呢？问题的答案，就像其他我们所提到的优化一样，要看你使用哪个版本的 Chrome，因为 Chrome 团队一直在研究新的更好的方法来提升性能。然而，广泛地说，在 Chrome 中的 DNS 基础设施主要有实现方式。历史上，Chrome 曾经依靠平台独立的系统调用——“getaddrinfo（）”，并委托操作系统来负责 DNS 查询。然而，这个逐步地被 Chrome 自己实现的异步 DNS 解析器替代。

在早先的依赖操作系统的实现中，有它的好处：极少，极简的代码，并且能够极大的利用操作系统的 DNS 缓存。然而，“getaddrinfo（）”是一个阻塞的系统调用，这意味着 Chrome 必须创建和维护一个专用的工作线程池来允许并行地执行多个查询。这个县城持有 6 个工作线程，这是一个基于普遍的硬件最低标准的经验主义的数字。更高的数字被证明会使用户的路由器超负荷。

对于使用工作现成的预解析来说，Chrome 简单地发起“getaddrinfo（）”的调用，阻塞工作线程直到结果返回，这时它只是丢弃返回的结果并开始执行下一个预解析的请求。解析的结果被缓存在操作系统的 DNS 缓存中，在之后的 getaddrinfo（）查询时会马上返回结果。这实践中既简单，高效，又表现梁好。

这虽然高效，但还没有足够好。“getaddrinfo（）”调用隐藏了很多有用的信息，例如 TTL，记录的时间戳和 DNS 缓存的状态。为了提升性能，Chrome 团队决定自己实现一个跨平台，的异步 DNS 解析器。



通过把 DNS 解析搬到 Chrome，新的异步解析器使很多优化变成可能：

- 更好地控制传输时间和能同时并行地执行多个请求
- 知道 TTL，让 Chrome 能提前刷新热记录。
- 在双协议栈(Ipv4 Ipv6)的实现上有更好的表现
- 基于 RTT 和其他信号，在需要时将解析请求转移到到别的服务器

Chrome 在不断试验和实现上述想法，这带一个明显的问题：我们怎么知道和衡量这些想法的影响呢？

简单的方式是：Chrome 为每一种配置统计网络性能并绘制成直方图。

要查看收集的 DNS 指标，打开一个新标签，访问“chrome://histograms/DNS”

```
Histogram: DNS.PrefetchResolution recorded 2621 samples, average = 126.6 (flags = 0x
0  O                                                                          (0 = 0.
15 ---------------------------------------------------------------------------O (366 =
17 ----------------------------------------------------------------O          (292 =
19 ----------------------------------------------------------------O          (289 =
21 ------------------------------------O                                     (195 =
23 -------------------------O                                               (187 =
26 ---------------O                                                          (112 =
29 -----------O                                                              (85 = 3
32 --------O                                                                 (79 = 3
36 ------O                                                                   (59 = 2
40 ----O                                                                     (51 = 1
45 ----O                                                                     (47 = 1
50 ---O                                                                      (42 = 1
```

上面的直方图展示了 DNS 预解析请求延迟的分布规律：粗劣地说，50%（最右列）的预先请求在 20ms（最左列）内完成。需要注意的是这些数据是基于近期浏览器会话的，对用户来说是私有的。如果用户选择向 Chrome 报告这些数据，这些数据会被总结后以匿名的方式定期传回工程师团队。Chrome 团队能依据这些数据知道试验的影响并作出调整。

用 TCP 提前连接来优化 TCP 连接

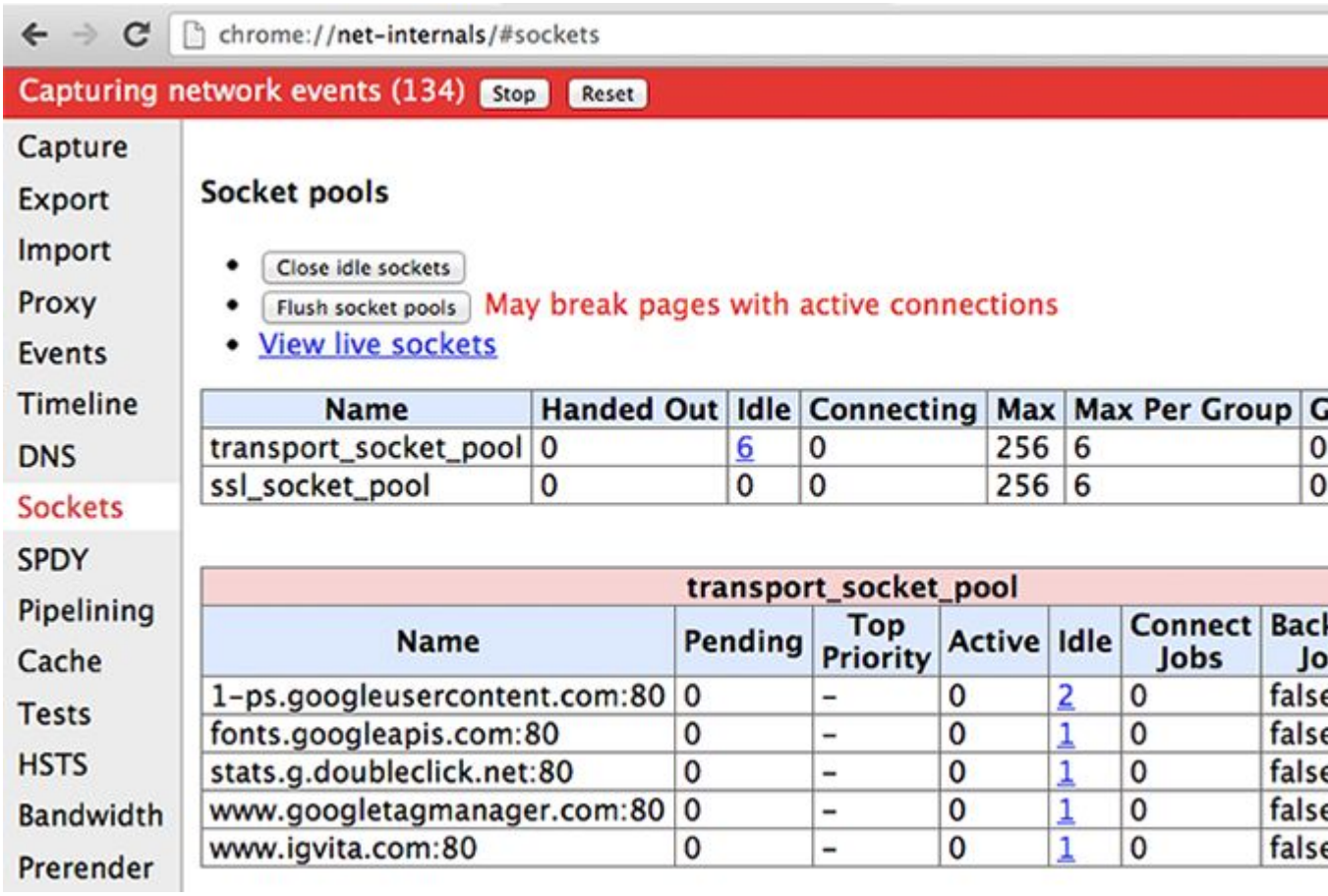
在我们提前解析了域名之后，并且地址栏或 Predictor 预测到有一个极高可能的浏览事件将要发生，为什么不更进一步，提前打开连接到该域名，并在用户发出请求前完成三次握手呢？这样做的话， 我们能够消除另外一个来回的延迟，轻而易举地给用户节省数百毫秒的时间。

要查已经触发提前 TCP 连接的域名，访问 “ chrome://dns”

Host for Page	Page Load Count	Subresource Navigations	Subresource PreConnects	Subresource PreResolves	Expected Connects	Subresource
https://plusone.google.com/	51	36	23	18	1.215	https://plusone

首先，Chrome 检查它的 socket 池，看看是否已经存一个对应域名的 socket，若有的话，Chrome 让这个 socket 在池中保留一段时间，以避免多余的 TCP 握手。如果没有一个 socket 可用，那么它会开始进行 TCP 握手，并把 socket 放在池中。然后，当用户开始浏览的时候，HTTP 请求可以被立刻发起。

对于那些好奇的人，Chrome 在“`chrome://net-internals#sockets`”提供了一个工具来查看已打开的 socket 的状态。



你还可以钻进每个 socket 里面查看时间轴：连接和代理的时间每个包到达的时间等等。最后但并不是不重要的，你还可以为将来的分析或 BUG 报告导出这些数据。有好的分析工具是任何性能优化的关键，“`chrome://net-internals`”集合里 Chrome 中的所有工具。

用预取提示来优化资源加载

有时网页的作者会基于他们网站的结构提供一些额外的导航，或者页面上下文，并帮助浏览器优化用户的体验。Chrome 支持两种这样的提示，可以嵌入页面中。

```
<link rel="subresource" href="/javascript/myapp.js">

<link rel="prefetch"    href="/images/big.jpeg">
```

“subresource”和“prefetch”看起来非常相似，但是有非常不同的语义，当一个链接资源申明它的关系为“prefetch”，这暗示浏览器之后浏览可能会用到这个资源。换句话说，实际上是一个跨页面的提示。因此这个提示可能想要在遇到这个资源之前提前请求。

如你所愿，两种提示不同的语义导致了资源加载器非常不同的行为。标记有“prefetch”的资源被认为是低优先级的，并且仅单当前页面加载完毕后，浏览器彩绘去加载这些资源。标记有“subresource”的资源拥有高优先级，浏览器一遇到就会马上去获取，并且和当前页面中剩下的资源进行竞争。

这两种提示如果用的好，可以在你的网站中极大地提升用户体验。最后，需要注意的是，“prefetch”是 HTML5 规范的一部分，并被 Firefox 和 Chrome 支持，然而“subresource”只有被 Chrome 支持。

通过预先刷新来优化资源加载

不幸的是，并不是所有的网站能够或者希望在他们网站的标签中为浏览器提供“ subresource ”的提示。更进一步，即使他们做到了，在我们能够解析提示并开始获取必要的“subresource ”之前必须等待 HTML 文档的到达。这依赖服务器响应时间，和客户端与服务器之间的延迟。这会花费几百甚至几千毫秒。

然而，正如我们之前看到的，Chrome 已经学习了热门资源的域名来提前进行 DNS 解析和打开 TCP 连接。那么，不顺便预测性地提前获取一些资源呢？这正是预刷新能做的事情：

- 用户向一个 URL 发出请求。
- Chrome 向 Predictor 查询目标 URL 相关的所学习到的“subresource”，并开始进行 DNS 预解析，TCP 预连接和资源预获取的一系列动作。
- 如果所学习到的“subresource”在缓存中，那么它会被从磁盘加载到内存。
- 如果所学习到的“subresource”不在缓存中或者过期了，那么会发起一个网络请求。

资源预取是 Chrome 的每个实验性的优化理论的一个极好的工作流程的例子，它会带来更好的性能，但同时也有些代价。只有一个办法能够可靠地决定它是否能达到标准并加入到 Chrome 中：实现它并在一些预览版的 Chrome 中对真实用户进行 A/B 测试。

自 2013 年初，是 Chrome 团队讨论这个实现的早期。如果它达到标准，我们可能回在几年之内看到预刷新。提高 Chrome 网络性能的过程从未停止，Chrome 团队一直在实验新的方法、想法和技术。

通过预渲染来优化导航

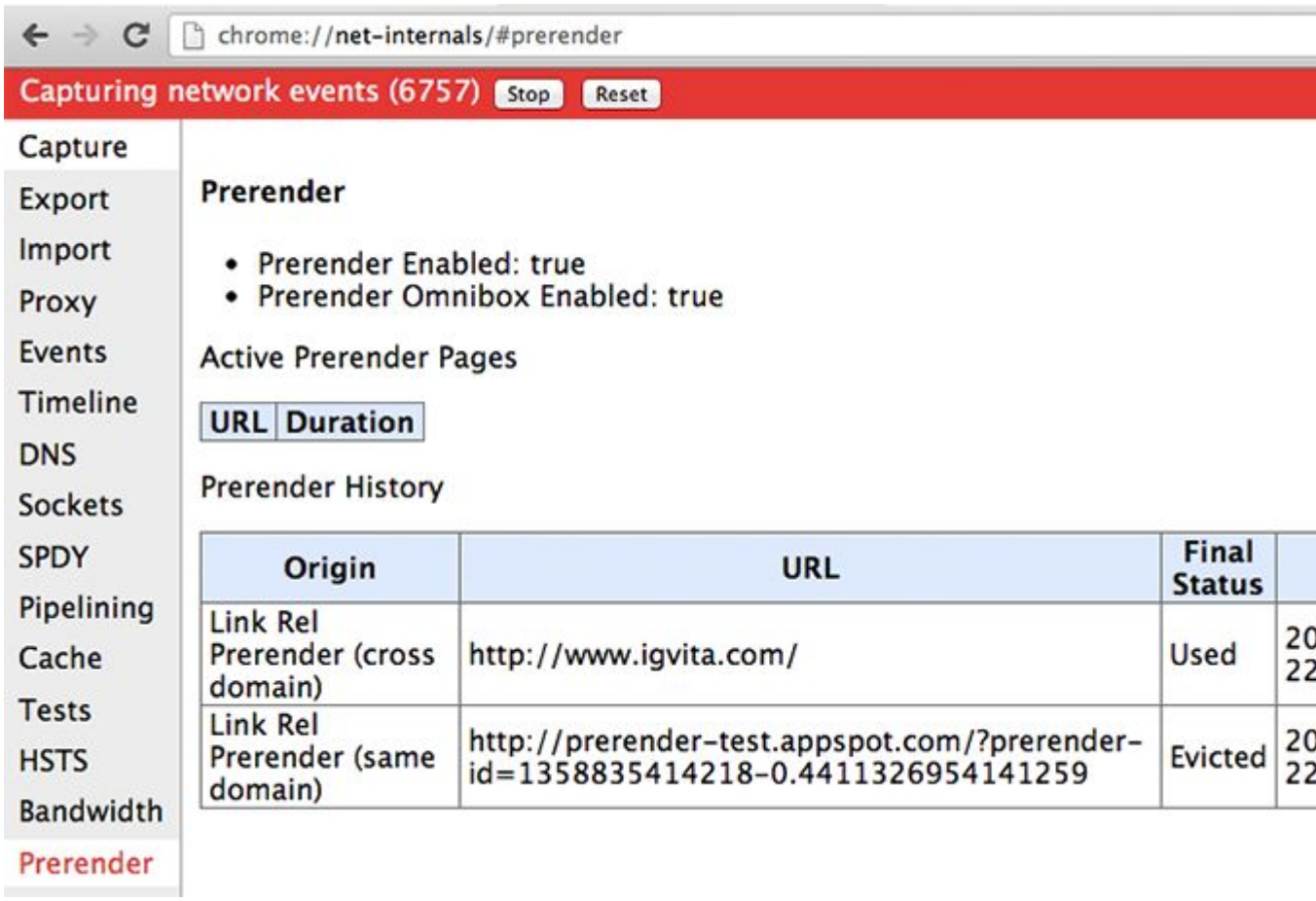
至此，每一个我们提到的优化策略帮助我们减少用户请求一个页面到在它们的标签中渲染出整个页面之间的延迟。然而，怎么洋财能有一个真实的即时体验呢？基于我们之前看过的 UX 的数据，交互必须发生在 100ms 以内，这没有给网络延迟留太多空间。我们怎么样才能在 100ms 内给出一个渲染完的页面呢？

当然，你已经知道答案，因为这对用户来说是一个非常普遍的浏览方式：如果你打开多个标签，你在多个标签之间的切换时即时的。这很明显比你在当前的标签等待导航要快的多。如果浏览器提供这样一个 API 会怎么样？

```
<link rel="prerender" href="http://example.org/index.html">
```

你猜到了，那就是 Chrome 的预渲染功能。“prerender”提示除了做了与“prefetch”相同的事情外，它还会向 Chrome 申明应该在一个隐藏的标签提前渲染这个页面和它的“subresource”。这个隐藏的标签对用户来说是不可见的，但是当用户触发导航的时候，这个标签会被置换到前面，带来一种即时体验。

你很好奇想要试试看嘛？你可以访问 prerender-test.appspot.com 亲身体验一下。你还能通过 `chrome://net-internals/#prerender` 查看已提前渲染的页面的历史和状态。



如你所愿，在一个隐藏的标签渲染整个页面会消耗很大的资源，包括 CPU 和网路。因此，仅当我们很有自信这个隐藏的标签会被使用的情况下才使用。例如，当你在使用地址栏的时候，如果一个交易的置信度很高，那么就会触发预渲染。谷歌搜索如果预测到第一个搜索结果极有可能是用户的目的地，那么它会在标签中加入“prerender”（也被称为谷歌即时页面）。

你可以在你的网站中加入“prerender”提示。在此之前，你需要知道预渲染有很多限制：

- 所有进程只能最多拥有一个一个预渲染标签。
- HTTPS 和带有身份验证的 HTTP 页面，那么不允许。
- 页面所请求的资源或者子资源需要进行一个非幂等的请求，那么不允许。

- 所有资源以最低优先级进行获取。
- 页面以最低 CPU 优先级渲染。
- 如果页面需要超过 100MB 的内存，那么页面会被放弃。
- 插件初始化会被推迟，如果有 HTML5 媒体标签，那么页面预渲染会被放弃。

换句话说，预渲染不能保证发生，并且适用于安全的页面。另外，因为 JavaScript 和其他逻辑可能会在隐藏标签执行，所以利用 *Page Visibility API* 去检测页面是否可见是一个最佳事实践，

Chrome 会随着你的使用越来越快

chrome 的网络协议栈不仅仅是一个 socket 管理器是不言而喻的。当你浏览网页的时候，我们在背后透明地进行了多方面的潜在优化。Chrome 对你的网络拓扑和浏览模式知道得越多，它就能运行地更好。如同魔术一般，Chrome 会越用越快。

最后，我们需要注意的是，Chrome 团队还在继续迭代和实验各种新的想法来提高性能。在达到每个页面的加载时间在 100ms 之前，Chrome 团队会继续努力，不断地进行开发、测试、发布新的优化版本。最后，为了回收资源，磁盘缓存会维护一个 LRU 缓存来考虑等级指标，例如访问频率和资源年龄。