[译]Google Chrome中的高性能网络

【译注】这部分不再详细翻译，只列出核心意思。

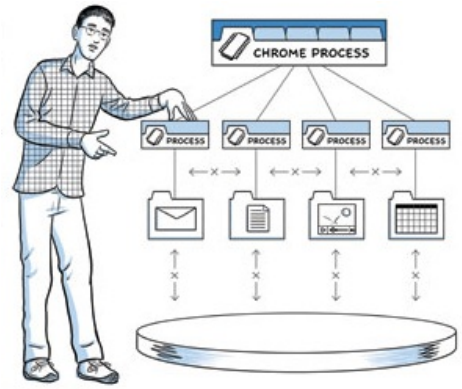
驱动Chrome继续前进的核心原则包括:

- Speed: 做最快的(fastest)的浏览器。
- Security:为用户提供最为安全的(most secure)的上网环境。
- Stability: 提供一个健壮且稳定的(resilient and stable)的Web应用平台。
- Simplicity: 以简练的用户体验(simple user experience)封装精益求精的技术(sophisticated technology)。

本文关将注于第一点，速度。

关于性能的方方面面

一个现代浏览器就是一个和操作系统一样的平台。在Chrome之前的浏览器都是单进程的应用，所有页面共享相同的地址空间和资源。引入多进程架构这是Chrome最为著名的改进【译注:省略一些反复谈论的细节】。



一个进程内，Web应用主要需要执行三个任务:获取资源，页面 排版及渲染，和运行JavaScript。渲染和脚本都是在运行中交替以单线程的方式运行的，其原因是为了保持DOM的一致性，而JavaScript本 身也是一个单线程的语言。所以优化渲染和脚本运行无论对于页面开发者还是浏览器开发者都是极为重要的。

Chrome的渲染引擎是WebKit, JavaScript Engine则使用深入优论的V8 (“V8” JavaScript runtime)。**但是，如果网络不畅，无论优化V8的JavaScript执行，还是优化WebKit的解析和渲染，作用其实很有限。巧妇难为无米之炊，数据没来就得等着!**

相对于用户体验，作用最为明显的就是如何优化网络资源的加载顺 序、优先级及每一个资源的延迟时间(latency)。也许你察觉不到，Chrome网络模块每天都在进步，逐步降低每个资源的加载成本：向DNS lookups学习，记住页面拓扑结构(topology of the web), 预先连接可能的目标网址，等等，还有很多。从外面来看就是一个简单的资源加载的机制，但在内部却是一个精彩的世界。

关于Web应用

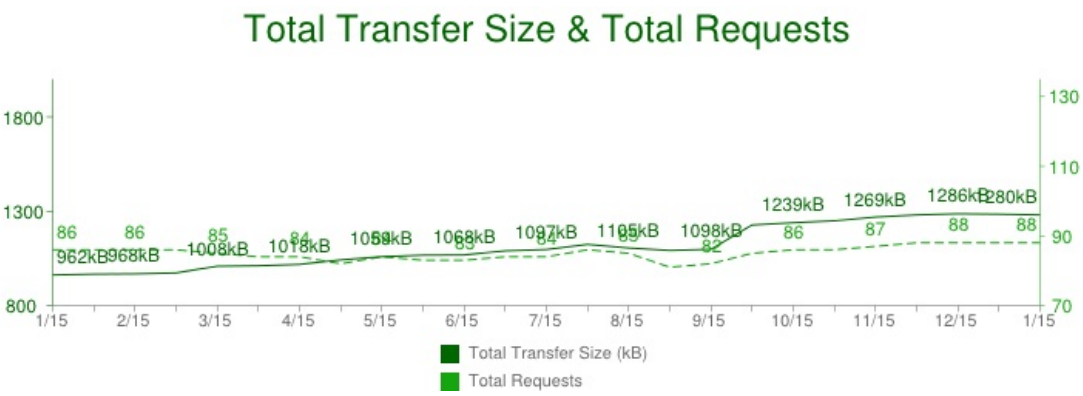
开始正题前，还是先来了解一下现在网页或者Web应用在网络上的需求。

建议继续学习

- 浏览器的工作原理：新式网络浏览器幕后揭秘
- 浏览器的工作原理：新式网络浏览器幕后揭秘
- 浅析http协议、cookies和session机制、浏览器缓存
- 从输入 URL 到页面加载完成的过程中都发生了什么事情?
- Chrome和goagent的配置方法，你懂的
- 图说浏览器战争：火狐、微软、谷歌那些事
- 程序员眼里IE浏览器是什么样的
- 浏览器的渲染原理简介
- 各种浏览器审查、监听http头工具介绍
- 警惕 Chrome 的查看源代码 (View Page Source) 功能
- 浏览器缓存机制

[625] 在vim保存时获得sudo权限
[579] java中文乱码解决之道（六）—-javaW
[417] 我希望我知道的七个JavaScript技巧
[404] Linux系统的CPU使用率和Load
[383] Java跨语言调用实现方案
[370] 小tip:纯CSS让overflow:aut
[324] 关于http代理

HTTP Archive 项目一直在追踪网页构建。除了页面内容外，它还会分析流行页面使用的资源数量，类型，头信息以及不同目标地址的元数据(metadata)。下面是2013年1月的统计资料，由300，000目标页面得出的平均数据:

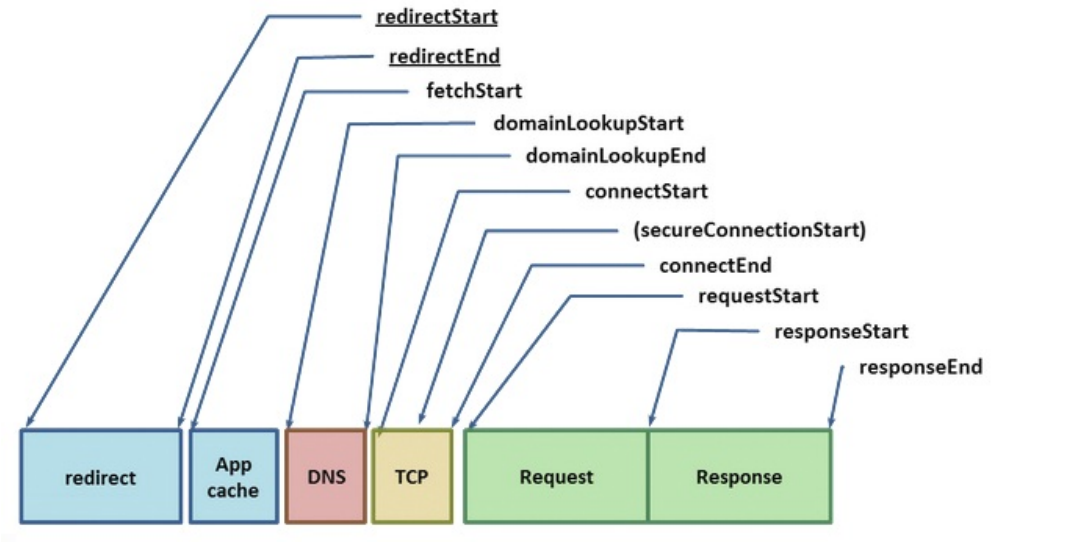


- 1280 KB
- 包含88个资源(Images,JavaScript,CSS …)
- 连接15个以上的不同主机(distinct hosts)

这些数字在过去几年中一直持续增长(steadily increasing)，没有停下的迹象。这说明我们正不断地建构一个更加庞大的、野心勃勃的网络应用。还要注意，平均来看每个资源不过12KB, 表明绝大多数的网络传输都是短促(short and bursty)的。这和TCP针对大数据、流式(streaming)下载的方向不一致，正因为如此，而引入了一些并发症。下面就用一个例子来抽丝剥茧，一窥究竟……

一个Resource Request的一生

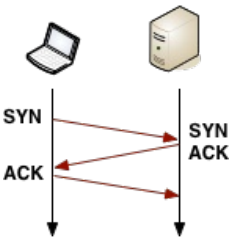
W3C的Navigation Timing specification定义了一组API，可以观察到浏览器的每一个请求(request)的时序和性能数据。下面了解一些细节:



给定一个网页资源地址后，浏览器就会检查本地缓存和应用缓存。如果之前获取过并且有相应的缓存信息(appropriate cache headers)(如Expires, Cache-Control, etc.), 就会用缓存数据填充这个请求，毕竟最快的请求就是没有请求(**the fastest request is a request not made**)。否则，我们重新验证资源，如果已经失效(expired),或者根本就没见过，一个耗费网络的请求就无法避免地发送了。

给定了一个主机名和资源路径后，Chrome先是检查现有已建立的连接(existing open connections)是否可以复用, 即sockets指定了以(scheme、host和port)定义的连接池(pool)。但如果配置了一个代理，或者指定了proxy auto-config(PAC)脚本，Chrome就会检查与proxy的连接。PAC脚本基于URL提供不同的代理，或者为此指定了特定 的规则。与每一个代理间都可以有自己的socket pool。最后，上述情况都不存在，这个请求就会从DNS查询(DNS lookup)开始了，以便获得它的IP地址。

幸运的话，这个主机名已经被缓存过。否则，必须先发起一个 DNS Query。这个过程所需的时间和ISP,页面的知名度，主机名在中间缓存(intermediate caches)的可能性，以及authoritative servers的响应时间这些因素有关。也就是说这里变量很多，不过一般还不致于到几百毫秒那么夸张。



拿到解析出的IP后,Chrome就会在目标地址间打开一个新TCP连接，我们就要执行一个3度握手(“three-way handshake”): SYN > SYN-ACK > ACK。这个操作每个新的TCP连接都必须完成，没有捷径。根据远近，路由路径的选择，这个过程可能要耗时几百毫秒，甚至几秒。而到现在，我们连一个有效的字节都还没收到。

当TCP握手完成了，如果我们连接的是一个HTTPS地址，还有一个SSL握手过程，同时又要增加最多两轮的延迟等待。如果SSL会话被缓存了，就只需一次。

最后，Chrome终于要发送HTTP请求了 (如上面图示中的requestStart)。 服务器收到请求后，就会传送响应数据(response data)回到客户端。这里包含最少的往返延迟和服务的处理时间。然后一个请求就完成了。但是，如果是一个HTTP重定向(redirect)的话？我们 又要从头开始这个过程。如果你的页面里有些冗余的重定向，最好三思一下！

你得出所有的延迟时间了吗? 我们假设一个典型的宽带环境：没有本地缓存，相对较快的DNS lookup(50ms), TCP握手，SSL协商，以及一个较快服务器响应时间(100ms)和一次延迟(80ms,在美国国内的平均值):

- 50ms for DNS
- 80ms for TCP handshake (one RTT)
- 160ms for SSL handshake (two RTT’s)
- 40ms （发送请求到服务器）
- 100ms (服务器处理)
- 40ms (服务器回传响应数据)

一个请求花了470毫秒, 其中80%的时间被网络延迟占去了。看到了吧，我们真得有很多事情要做！事实上,470毫秒已经很乐观了:

- 如果服务器没有达到到初始TCP的拥塞窗口(congestion window)，即4-15KB，就会引入更多的往返延迟。
- SSL延迟也可能变得更糟。如果需要获取一个没有的认证(certificate)或者执行online certificate status check(OCSP), 都会让我们需要一个新的TCP连接，又增加了数百至上千毫秒的延迟。

怎样才算”够快”?

前面可以看到服务器响应时间仅是总延迟时间的20%，其它都被DNS，握手等操作占用了。过去用户体验研究(user experience research)表明用户对延迟时间的不同反应：

- 0 - 100ms 迅速
- 100 - 300ms 有点慢
- 300 - 1000ms 机器还在运行
- 1s+ 想想别的事……
- 10s+ 我一会再来看看吧……

上表同样适用于页面的性能表现: 渲染页面，至少要在250ms内给个回应来吸引住用户。这就是简单地针对速度。从Google, Amazon, Microsoft,以及其它数千个站点来看，额外的延迟直接影响页面表现：流畅的页面会吸引更多的浏览、以及更强的用户吸引力(engagement) 和页面转换率(conversion rates).

现在我们知道了理想的延迟时间是250ms，而前面的示例告诉我们，DNS Lookup, TCP和SSL握手，以及request的准备时间花去了370ms, 即便不考虑服务器处理时间，我们也超出了50%。

对于绝大多数的用户和网页开发者来说，DNS, TCP，以及SSL延迟都是透明，很少有人会想到它。这也就是为什么Chrome的网络模块那么的复杂。

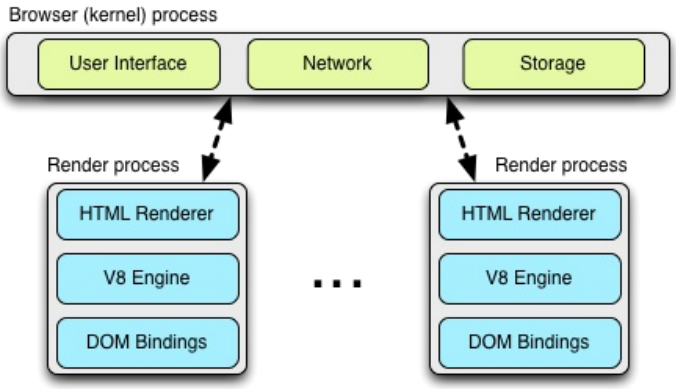
我们已经识别出了问题，下面让我们深入一下实现的细节…

深入Chrome的网络模块

多进程架构

Chrome的多进程架构为浏览器的网络请求处理带来了重要意义，它目前支持四种不同的执行模式(four different execution models)。

默认情况下，桌面的Chrome浏览器使用process-per-site模式, 将不同的网站页面隔离起来, 相同网站的页面组织在一起。举个简单的例子: 每个tab独立一个进程。从网络性能的角度上说,并没什么本质上的不同，只是process-per- tabl模式更易于理解。

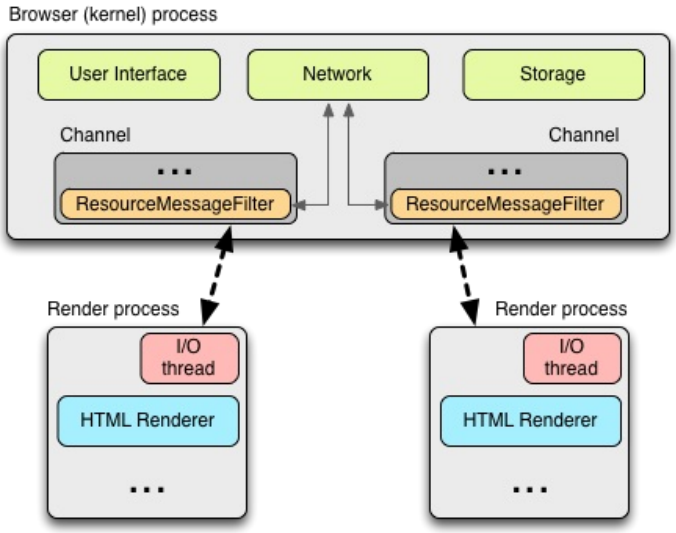


每一个tab有一个渲染进程(render process)，其中包括了用于解析页面(interpreting)和排版(layout out)的WebKit的排版引擎(layout engine), 即上图中的HTML Render。还有V8引擎和两者之间的DOM Bindings，如果你对这部分很好奇，可以看这里(great introduction to the plumbing)。

每一个这样的渲染进程被运行在一个沙箱环境中，只会对用户的电 脑环境做极有限的访问-包括网络。而使用这些资源，每一个渲染进程必须和浏览内核进程(browser[kernel] process)沟通，以管理每个渲染进程的安全性和访问策略(access policies)。

进程间通讯(IPC)和多进程资源加载

渲染进程和内核进程之间的通讯是通过IPC完成的。在Linux和 Mac OS上，使用了一个提供异步命名管道通讯方式的socketpair()。每一个渲染进程的消息会被序列化地到一个专用的I/O线程中，然后再由它发到内 核进程。在接收端，内核进程提供一个过滤接口(filter interface)用于解析资源相关的IPC请求(ResourceMessageFilter), 这部分就是网络模块负责的。



这样做其中一个好处是所有的资源请求都由I/O进程处理，无论是UI产生的活动，或者网络事件触发的交互。在内核进程(browser/kernel process)的I/O线程解析资源请求消息，将转发到一个ResourceDispatcherHost的单例(singleton)对象中处理。

这个单例接口允许浏览器控制每个渲染进程对网络的访问，也能达到有效和一致的资源共享：

- **Socket pool 和 connection limits**: 浏览器可以限定每一个profile打开256个sockets, 每个proxy打开32个sockets, 而每一组{scheme, host, port}可以打开6个。注意同时针对一组{host,port}最多允计打开6个HTTP和6个HTTPS连接。
- **Socket reuse**: 在Socket Pool中提供持久可用的TCP connections，以供复用。这样可以为新的连接避免额外建立DNS、TCP和SSL(如果需要的话)所花费的时间。
- **Socket late-binding(延迟绑定)**: 网络请求总是当Scoket准备好发送数据时才与一个TCP连接关连起来，所以首

先有机会做到对请求有效分级(prioritization)，比如，在 socket连接过程中可能会到达到一个更高优先级的请求。同时也可以有更好的吞吐率(throughput),比如，在连接打开过程中，去复用一个刚好 可用的socket, 就可以使用到一个完全可用的TCP连接。其实传统的TCP pre-connect(预连接)及其它大量的优化方法也是这个效果。

- **Consistent session state(一致的会话状态)**: 授权、cookies及缓存数据会在所有渲染进程间共享。
- **Global resource and network optimizations(全局资源和网络优化)**: 浏览器能够在所有渲染进程和未处理的请求间做更优的决策。比如给当前tab对应的请求以更好的优先级。
- **Predictive optimizations（预测优化）**: 通过监控网络活动，Chrome会建立并持续改善预测模型来提升性能。
- … 项目还在增加中。

单就一个渲染进程而言, 透过IPC发送资源请求很容易，只要告诉浏览器内核进程一个唯一ID, 后面就交给内核进程处理了。

跨平台的资源加载

跨平台也是Chrome网络模块的一个主要考量，包括Linux, Windows, OS X, Chrome OS, Android, 和iOS。 为此，网络模块尽量实现成了单进程模式（只分出了独立的cache和proxy进程）的跨平台函数库, 这样就可以在平台间共用基础组件(infrastructure)并分享相同的性能优化，更有机会做到同时为所有平台进行优化。

相关的代码可以在这里找到the “src/net” subdirectory)。本文不会详细展开每个组件，不过了解一下代码结构可以帮助我们理解它的能力结构。 比如:

- net/android 绑定到Android 运行时(runtime) [译注(Horky):运行时真是一个很烂的术语，翻和没翻一样。]
- net/base 公共的网络工具函数。比如,主机解析， cookies, 网络转换侦测(network change detection),以及SSL认证管理
- net/cookies 实现了Cookie的存储、管理及获取
- net/disk_cache 磁盘和内存缓存的实现
- net/dns 实现了一个异步的DNS解析器(DNS resolver)
- net/http 实现了HTTP协议
- net/proxy 代理(SOCKS 和 HTTP)配置、解析(resolution) 、脚本抓取(script fetching), …
- net/socket TCP sockets,SSL streams和socket pools的跨平台实现
- net/spdy 实现了SPDY协议
- net/url_request URLRequest, URLRequestContext和URLRequestJob的实现
- net/websockets 实现了WebSockets协议

上面每一项都值得好好读读，代码组织的很好，你还会发现大量的单元测试。

Mobile平台上的架构和性能

移动浏览器正在大发展，Chrome团队也视优化移动端的体验为最高优先级。先要说明的是移动版的Chrome的并不是其桌面版本的直接移植，因为那样根本不会带来好的用户体验。移动端的先天特性就决定了它是一个资源严重受限的环境，在运行参数有一些基本的不同:

- 桌面用户使用鼠标操作，可以有重叠的窗口，大的屏幕，也不用担心电池。网络也非常稳定，有大量的存储空间和内存。
- 移动端的用户则是触摸和手势操作，屏幕小，电池电量有限，通过只能用龟速且昂贵的网络，存储空间和内存也是相当受限。

再者，不但没有典型的样板移动设备，反而是有一大批各色硬件的设备。Chrome要做的，只能是设法兼容这些设备。好在Chrome有不同的运行模式(execution models)，面对这些问题，游刃有余！

在Android版本上，Chrome同样运用了桌面版本的多进程架构。一个浏览器内核进程，以及一个或多个渲染进程。但因为内存的限制，移动版的Chrome无法为每一个tabl运行一个特定的渲染进程，而是根据内存情况等条件决定一个最佳的渲染进程个数，然后就会在多个tab间共享这些渲染进程。

如果内存实在不足，或其它原因导致Chrome无法运行多进程，它就会切到单进程、多线程的模式。比如在iOS设备上，因为其沙箱机制的限制，Chrome只能运行在这种模式下。

关于网络性能，首先Chrome在Android和iOS使用的是 各其它平台相同的网络模块。这可以做到跨平台的网络优化，这也是Chrome明显领先的优势之一。所不同的是需要经常根据网络情况和设备能力进行些调整， 包括推测优化(speculative

optimization)的优先级、socket的超时设置和管理逻辑、缓存大小等。

比如，为了延长电池寿命，移动端的Chrome会倾向于延迟关闭空 闲的sockets (lazy closing of idle sockets), 通常是为了减少信号(radio)的使用而在打开新的socket时关闭旧的。另外因为预渲染(pre-rendering,稍后会介绍)会使用一定的网 络和处理资源，它通常只在WiFi才会使用。

关于移动浏览体验会独立一章，也许就在POSA系列的下一期。

Chrome Predictor的预测功能优化

Chrome会随着使用变得更快。它这个特性是通过一个单例对象Predictor来实现的。这个对象在浏览器内核进程(Browser Kernel Process)中实例化，它唯一的职责就是观察和学习当前网络活动方式，提前预估用户下一步的操作。下面是一个示例：

- 用户将鼠标停留在一个链接上，就预示着一个用户的偏好以及下一步的浏览行为。这时Chrome就可以提前进行DNS Lookup及TCP握手。用户的点击操作平均需要将近200ms,在这个时间就可能处理完DNS和TCP相关的操作,也就是省去几百毫秒的延迟时间。
- 当在地址栏(Omnibox/URL bar) 触发高可能性选项时，就同样会触发一个DNS lookup和TCP预连接(pre-connect)，甚至在一个不可见的页签中进行预渲染(pre-render)!
- 我们每个人都一串天天会访问的网站, Chrome会研究在这些页面上的子资源, 并且尝试进行预解析(pre-resolve),甚至可能会进行预加载(pre-fetch)以优化浏览体验。

除了上面三项，还有很多..

Chrome会在你使用过程中学习Web的拓扑结构，而不单单是你的浏览模式。理想的话，它将为你省去数百毫秒的延迟,更接近于即时页面加载的状态. 正是为了这个目标,Chrome投入了以下的核心优化技术:

- DNS预解析(pre-resolve)：提前解析主机地址,以减少DNS延迟
- TCP预连接(pre-connect)：提前连接到目标服务器,以减少TCP握手延迟
- 资源预加载(prefetching)：提前加载页面的核心资源,以加载页面显示
- 页面预渲染(prerendering)：提前获取整个页面和相关子资源,这样可以做到及时显示

每一个决策都包含着一个或多个的优化, 用来克服大量的限制因素. 不过毕竟都只是预测性的优化策略，如果效果不理想，就会引入多余的处理和网络传输。甚至可能会带来一些加载时间上的负体验。

Chrome如何处理这些问题呢? Predictor会尽量收集各种信息，诸如用户操作，历史浏览数据，以及来自渲染引擎(render)和网络模块自身的信息。

和Chrome中负责网络事务调度的ResourceDispatcherHost不同，Predictor对象会针对用户和网络事务创建一组过滤器(filter):

- IPC channel filter用来监控来自render进程的事务。
- 每个请求上都会加一个ConnectInterceptor 对象，这样就可以跟踪网络传输的模式以及每一个请求的度量数据。

渲染进程(render process)会在一系列的事件下发送消息到浏览器进程(browser process), 这些事件被定义在一个枚举(ResolutionMotivation)中以便于使用 (url_info.h):

```
enumResolutionMotivation{
   MOUSE_OVER_MOTIVATED,      // 鼠标悬停.
   OMNIBOX_MOTIVATED,         // Omni-box建议进行解析.
   STARTUP_LIST_MOTIVATED,    // 这是在前10个启动项中的资源.
   EARLY_LOAD_MOTIVATED,      // 有时需要使用prefetched来提前建立连接.
   // 下面定义了预加载评估的方式，会由一个navigation变量指定.
   // referring_url_也需要同时指定.
   STATIC_REFERAL_MOTIVATED,  // 外部数据库(External Database)建议进行解析。
   LEARNED_REFERAL_MOTIVATED,// 前一次浏览(prior navigation建议进行解析.
   SELF_REFERAL_MOTIVATED,    // 猜测下一个连接是不是需要进行解析.
   // <略> ...
};
```

通过这些给定的事 件，Predictor的目标就可以评估它成功的可能性, 然后再适时触发操作。每一项事件都有其成功的机率、优先级以及时间戳，这些可以在内部维护一个用优先级管理的队列，也是优化的一个手段。最终，对于这个队 列中

发出的每一个请求的成功率，都可以被Predictor追踪到。基于这些数据，Predictor就可以进一步优化它的决策。

Chrome网络架构小结

- Chrome使用多进程架构，将渲染进程同浏览器进程隔离开来。
- Chrome维护着一个资源分发器的实例(a single instance of the resource dispatcher), 它运行在浏览器内核进程，并在各个渲染进程间共享。
- 网络层是跨平台的,大部分情况下以单进程库存在。
- 网络层使用非阻塞式(no-blocking)操作来管理所有网络任务。
- 共享的网络层支持有效的资源排序、复用、并为浏览器提供在多进程间进行全局优化的能力。
- 每一个渲染进程通过IPC和资源分发器(resource dispatcher)通讯。
- 资源分发器(Resource dispatcher)通过自定义的IPC Filter解析资源请求。
- Predictor在解析资源请求和响应网络事务中学习，并对后续的网络请求进行优化。
- Predictor会根据学习到的网络事务模式预测性的进行DNS解析, TCP握手，甚至是资源请求，为用户实际操作时节省数百毫秒的时间。

了解晦涩的内部细节后，让我们来看一下用户可以感受到的优化。一切从全新的Chrome开始。

优化冷启动(Cold-Boot)体验

第一次启动浏览器，它当然不可能了解你的使用习惯和喜欢的页面。但事实上，我们大多数会在浏览器的冷启动后做些类似的事情，比如到电子邮箱查看邮件，加一些新闻页面、社交页面及内部 页面到我的最爱，诸如此类。这些页面各有不同，但它们仍然具有一些相似性，所以Predictor仍然可以对这个过程提速。

Chrome记下了用户在全新启动浏览器时最常用的10个域名。当浏览器启动时，Chrome会提前对这些域名进行DNS预解析。你可以在Chrome中使用chrome://dns查看到这个列表。在打开页面的最上面的表格中会列出启动时的备选域名列表。

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

通过Omnibox优化与用户的交互

引入Omnibox是Chrome的一项创新, 并不是简单地处理目标的URL。除了记录之前访问过的页面URL，它还与搜索引擎的整合，并且支持在历史记录中进行全文搜索(比如，直接输入页面名称)。

当用户输入时，Omnibox自动发起一个行为，要么查找浏览记录中的URL, 要么进行一次搜索。每一次发起的操作都会被加以评分，以统计它的性能。你可以在Chrome输入chrome://predictors来观察这些数据。

☑ Filter zero confidences

Entries: 125

User Text	URL	Hit Count	Miss Count	Confidence
g	http://gmail.com/	594	186	0.7615384615384615
gi	http://githubarchive.org/	25	55	0.3125
gi	https://gist.github.com/	16	49	0.24615384615384617
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	0.9975728155339806

Chrome维护着一个历史记录，内容包括用户输入的前置文字，采用的行为，以命中的资数。在上面的列表，你可以看到，当输入g时，有76%的机会尝试打开Gmail. 如果再补充一个m (就是gm), 打开Gmail的可能性增加到99.8%。

那么网络模块会做什么呢？上 表中的黄色和绿色对于ResourceDispatcher非常重要。如果有一个一般可能性的页面(黄色), Chrome就是发起DNS预解析。如果有一个高可能性的页面(绿色)，Chrome还会在DNS解析后发起TCP预连接。如果这两项都完成了，用户仍然 继续录入，Chrome就会在一个隐藏的页签进行预渲染(pre-render)。

相对的，如果输入的前置文字找不到合适的匹配项目，Chrome会向搜索引擎服务者发起DNS预解析和TCP预连，以获取相似的搜索结果。

平均而言用户从填写查询内容到评估给出的建议需要花费数百毫秒。此时Chrome可以在后台进行预解析，预连接，甚至进行预渲染。再当用户准备按下回车键时，大量的网络延迟已经被提前处理掉了。

优化缓存性能

最快的请求就是没有请求。 无论何时讨论性能，都不能不谈缓存。相信你已经为页面上所有资源的都提供了Expires, ETag, Last-Modified和Cache-Control这些响应头信息(response headers)。什么? 还没有？那你还是先处理好再来看吧!

Chrome有两种不同的内部缓存的实现：一种备份于本地磁盘(local disk)，另一种则存储于内存(in-memory)中。内存模式(in-memory)主要应用于无痕浏览模式(**Incognito browsing mode**),并在关闭窗口清除掉。 两种方式使用了相同的内部接口(disk_cache::Backend, 和disk_cache::Entry)，大大简化了系统架构。如果你想实现一个自己的缓存算法，可以很容易地实现进去。

在内部，磁盘缓存(disk cache)实现了它自己的一组数据结构, 它们被存储在一个单独的缓存目录里。其中有索引文件(在浏览器启动时加载到内存中)，数据文件(存储着实际数据，以及HTTP头以及其它信息)。比较有趣 的是，16KB以下的文件存储于共同的数据块文件中(data block-files,即小文件集中存储于一个大文件中)，其它较大的文件才会存储到自己专属的文件中。最后，磁盘缓存的淘汰策略是维护一个LRU，通 过比如访问频率和资源的使用时间(age)的度量进行管理。

在Chrome开个页签，输入chrome://net-internals/#httpCache。 如果你要看到实际的HTTP数据和缓存的响应处理，可以打开chrome://cache, 里面会列出所有缓存中可用的资源。打开每一项，还可以看到详细的数据头等信息。

优化DNS预连接

前面已经多次提到了DNS预解析，在深入实现之前，先汇总一下DNS预解析的场景和理由:

- 运行在渲染进程中的WebKit文档解析器(document parser), 会为当前页面上所有的链接提供一个主机名(hostname)列表，Chrome可以选择是否提前解析。
- 当用户要打开页面时，渲染进程先会触发一个鼠标悬停(hover)或按键(button down)事件。

- Omnibox可能会针对一个高可能性的建议页面发起解析请求。
- Chrome Predictor会基于过往浏览记录和资源请求数据发起主机解析请求。(下面会详细解释。)
- 页面本身会显式地要求Chrome对某些主机名称进行预解析。

上述各项对于Chrome都只是一个线索。Chrome并不保证预解析一定会被执行，所有的线索会由Predictor进行评估，以决定后续的操作。最坏的情况下，可能无法及时解析主机名，用户就必须等待一个 DNS解析时间，然后是TCP连接时间，最后是资源加载时间。Predictor会记下这个场景，在未来决策时相应地加以参考。总之，一定是越用越快。

之前提过到Chrome可以 记住每个页面的拓扑(topology),并可以基于这个信息进行加速。还记得吧，平均每个页面带有88个资源，分别来自于30多个独立的主机。每打开这 个页面，Chrome会记下资源中比较常用的主机名，在后续的浏览过程中，Chrome就会发起针对某些主机或者全部主机的DNS解析，甚至是TCP预连接!

Host for Page	Page Load Count	Subresource Navigations	Subresource PreConnects	Subresource PreResolves	Expected Connects	Subresource Spec
https://plus.google.com/	688	6	4	17	0.013	https://apis.google.com/
		2	3	8	0.065	https://csi.gstatic.com/
		152	27	33	0.194	https://lh3.googleusercontent.com/
		2	3	1	0.509	https://lh6.googleusercontent.com/
		896	296	386	1.853	https://plus.google.com/
		79	22	18	0.194	https://ssl.gstatic.com/

使用chrome://dns 就可以观察到上面的数据(Google+页面), 其中有6个子资源对应的主机名，并记录了DNS预解析发生的次数，TCP预连接发生的次数，以及到每个主机的请求次数。这些数据就可以让Chrome Predictor执行相应的优化决策。

除了内部事件通知外，页面设计者可以在页面中嵌入如下的语句请求浏览器进行预解析：

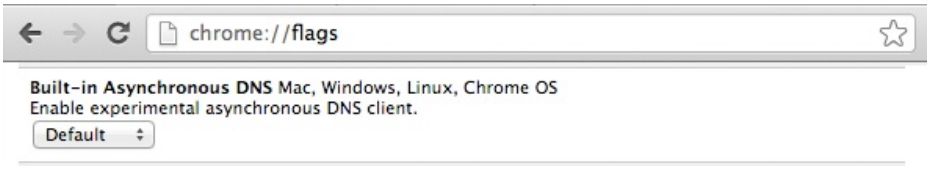
```
<link rel="dns-prefetch"href="//host_name_to_prefetch.com">
```

之所以有这个需求，一个典型的例子是重定向(redirects). Chrome本身没办法判断出这种模式，通过这种方式则可以让浏览器提前进行解析。

具体的实现也是因版本而有所差异，总体而言，Chrome中的DNS处理有两个主要的实现：1.基于历史数据(historically), 通过调用平台无关的getaddrinfo()系统函数实现。2.代理操作系统的DNS处理方法，这种方法正在被Chrome自行实现的一套异步DNS解析机制(asynchronous DNS resolver)所取代。

依赖于系统的实现，代码少而 且简单，但是getaddrInfo()是一个阻塞式的系统调用，无法有效地并行多个查询操作。经验数据还显示，并行请求过多甚至会超出路由器的负额。 Chrome为此设计了一个复杂的机制。对于其中带有worker-pool的预解析, Chrome只是简单的发送getaddrinfo() 调用, 同时阻塞worker thread直到收到响应数据。因为系统有DNS缓存的原因，针对解析过的主机，解析操作会立即返回。 这个过程简单，有效。

但还不够！ getaddrinfo()隐藏了太多有用的信息，比如Time-to-live(TTL)时间戳, DNS缓存的状态等。于是Chrome决定自己实现一套跨平台的异步DNS解析器。



这个新技术可以支持以下优化:

- 更好地控制重转的时机，有能力并行执行多个查询操作。 清晰地记录TTLs。
- 更好地处理IPv4和IPv6的兼容。
- 基于RTT和其它事件，针对不同服务器进行错误处理(failover)

Chrome仍然进行着持续的优化. 通过chrome://histograms/DNS可以观察到DNS度量数据：

```
Histogram: DNS.PrefetchResolution recorded 2621 samples, average = 126.6 (flags = 0x1)
0   0                                                                          (0 = 0.0%)
15 ------------------------------------------------------------------------O (366 = 14.0%) {0.0%}
17 ----------------------------------------------------------O                (292 = 11.1%) {14.0%}
19 ----------------------------------------------------------O                (289 = 11.0%) {25.1%}
21 ---------------------------------------O                                  (195 = 7.4%) {36.1%}
23 ------------------------O                                                 (187 = 7.1%) {43.6%}
26 ---------------O                                                          (112 = 4.3%) {50.7%}
29 -----------O                                                              (85 = 3.2%) {55.0%}
32 --------O                                                                 (79 = 3.0%) {58.2%}
36 ------O                                                                   (59 = 2.3%) {61.2%}
40 ----O                                                                     (51 = 1.9%) {63.5%}
45 ----O                                                                     (47 = 1.8%) {65.4%}
50 ---O                                                                      (42 = 1.6%) {67.2%}
```

上面的柱状图展示了 DNS预解析延迟时间的分布：比如将近50%(最右侧)的查询在20ms内完成。这些数据基于最近的浏览操作(采样9869次)，用户可以选择是否报告这 些使用数据，然后这些数据会以匿名的形式交由工程团队加以分析，这样就可以了解到功能的性能，以及未来如何进一步调整。周而复始，不断优化。

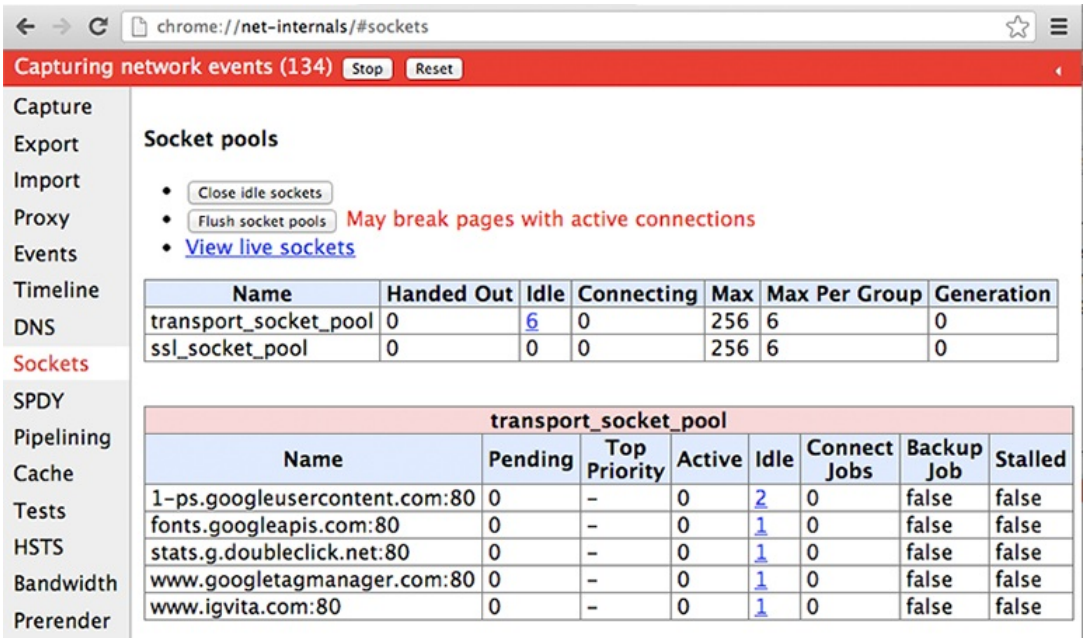
使用预连接优化了TCP连接管理

已经预解析到了主机名，也有了 由OmniBox和Chrome Predictor提供信号，预示着用户未来的操作。为什么再进一步连接到目标主机，在用户真正发起请求前完成TCP握手呢？这样就可省掉了另一个往返的 延迟，轻易地就能为用户节省到上百毫秒。其实，这就是TCP预连接的工作。 通过访问chrome://dns 就可以看到TCP预连接的使用情况。

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

首先, Chrome检查它的socket pool里有没有目标主机可以复用的socket， 这些sockets会在socket pool里保留一段时间，以节省TCP握手时间及启动延时(slow-start penalty)。如果没有可用的socket, 就需要发起TCP握手，然后放到socket pool中。这样当用户发起请求时，就可以用这个socket立即发起HTTP请求。

打开 chrome://net-internals#sockets 就可以看到当前的sockets的状态:



你可以看到每一个socket的时间轴:连接和代理的时间，每个封包到达的时间，以及其它一些信息。你也可以把这些数据导出，以方便进一步分析或者报告问题。**有好的测试数据是优化的基础, chrome://net-internals就是Chrome网络的信息中心**。

使用预加载优化资源加载

Chrome支持在页面的HTML标签中加入的两个线索来优化资源加载:

```
<link rel="subresource"href="/javascript/myapp.js">
<link rel="prefetch"    href="/images/big.jpeg">

```

在rel中指定的 subresource(子资源)和prefetch(预加载)非常相似。不同的是，如果一个link指定rel(relation)为prefetch后，就是告诉浏览器这个资源是稍后的页面中要用到的。而指定为subresource则表示在本页中就会用到,期望能在使用前

加载。两者不同的语义让 resource loader有不同的行为。prefetch的优先级较低，一般只会在页面加载完成后才会开始。而subresource则会在解析出来时就被尝试优先执行。

还要注意，prefetch是HTML5的一部分，Firefox和Chrome都可以支持。但subresource还只能用在Chrome中。

应用Browser Prefreshing优化资源加载

不过，并不是所有的Web开发者会愿意加入上面所述的subresource relation, 就算加了，也要等收到主文档并解析出这些内容才行，这段时间开销依赖于服务器的响应时间和客户端与服务器间的延迟时间，甚至要耗去上千毫秒。

和前面的预解析,预连接一样，这里还有一个prefreshing::

- 用户初始化一个目标页面的请求。
- Chrome查询Predictor之前针对目标页面的子资源加载，初始化一组DNS预解析，TCP预连接及资源prefreshing。
- 如是在缓存中发现之前记录的子资源，由从磁盘中加载到内存中。
- 如果没有或者已经过期了，就是发送网络请求。

直到在2013年初, prefreshing还是处于早期的讨论阶段。如果通过数据结果分析，这个功能最终上线了，我们就可以稍晚些时候使用到它了。

使用预渲染优化页面浏览

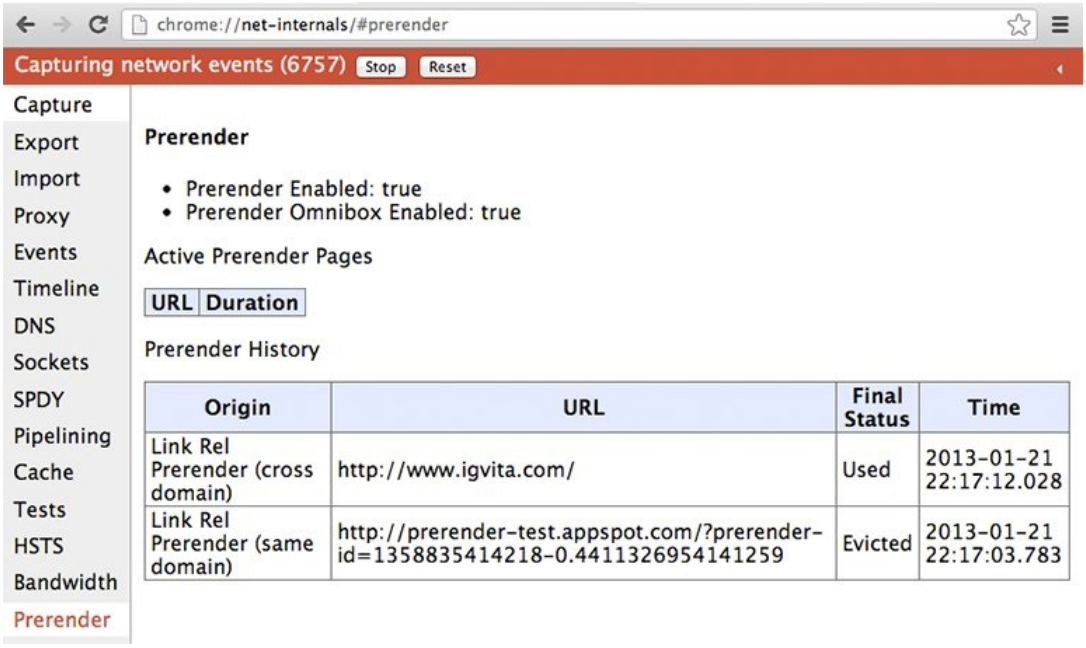
前面讨论的每个优化都用来帮助减少用户发起请求到看到页面内容的延迟时间。多快才能带来即时呈现的体验呢？基于用户体验数据，这个时间是100毫秒，根本没给网络延迟留什么空间。而在100毫秒内，又怎样渲染页面呢?

大家可能都有这样的体验: 同时开多个页签时会明显快于在一个页签中等待。浏览器为此提供了一个实现方式:

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

这就是Chrome的预渲染(prerendering in Chrome)! 相对于只下载一个资源的“prefetch”, “prerender”会让Chrome在一个不可见的页签中渲染一个页面,包含了它所有的子资源。当用户要浏览它时，这个页签被切到前台，做到了即时的体验。

可以浏览prerender-test.appspot.com来体验一下效果，再通过chrome://net-internals/#prerender查看下历史记录和预连接页面的状态。



因为预渲染会同时消耗CPU和网络资源，因些一定要在确信预渲染页面会被使用到情况下才用。Google Search就在它的搜索结果里加入prerender, 因为第一个搜索结果很可能就是下一个页面(也叫作Google Instant Pages)

你可以使用预渲染特性，但以下限制项一定要牢记:

- 所有的进程中最多只能有一个预渲染页。
- HTTPS和带有HTTP认证的页面不可以预渲染。
- 如果请求资源需要发起非幂等（non-idempotent,idempotent request的意义为发起多次，不会带来服务的负面响应的请求)的请求(只有GET请求)时，预渲染也不可用。
- 所有的资源的优先级都很低。
- 页面渲染进程的使用最低的CPU优先级。
- 如果需要超过100MB的内存，将无法使用预渲染。
- 不支持HTML5多媒体元素。

预渲染只能应用于确信安全的页面。另外JavaScript也最好在运行时使用Page Visibility API来判断一下当前页是否可见(参考 you should be doing anyway) !

最后，总之，Chrome正逐步优化网络延迟和用户体验，让它随着用户的使用越来越快!

Ilya Grigorik is a web performance engineer and developer advocate on the Make The Web Fast team at Google, where he spends his days and nights on making the web fast and driving adoption of performance best practices.
Follow @igrigorik

建议继续学习：

1. 浏览器的工作原理：新式网络浏览器幕后揭秘　(阅读：13655)
2. 浅析http协议、cookies和session机制、浏览器缓存　(阅读：11867)
3. 从输入 URL 到页面加载完成的过程中都发生了什么事情？　(阅读：11302)
4. Chrome和goagent的配置方法，你懂的　(阅读：9810)
5. 图说浏览器战争：火狐、微软、谷歌那些事　(阅读：5268)
6. 程序员眼里IE浏览器是什么样的　(阅读：5053)
7. 浏览器的渲染原理简介　(阅读：4986)
8. 各种浏览器审查、监听http头工具介绍　(阅读：4877)
9. 警惕 Chrome 的查看源代码 (View Page Source) 功能　(阅读：4592)
10. 浏览器缓存机制　(阅读：4528)

前一篇： 战斗HTTP　　**后一篇：** NAS解决方案实现多媒体文件共享播放

文章信息

- 作者：Horky　来源：UC技术博客
- 标签：Chrome 浏览器
- 发布时间：2014-12-02 23:41:12

评论

登录 | 注册

还可以输入140字

表情 同步到微博 评论

10条评论

Yiyang34472 1分钟前

@保存到为知笔记

回复

方垣之城 1月15日 10:18

@我的印象笔记

回复

Halloween92 1月14日 01:06

@我的印象笔记

回复

Topppy 1月4日 10:37

@我的印象笔记

回复

kenical 1月4日 08:12

@我的印象笔记

回复

看黛玉葬花 1月3日 12:08

没VPN的话不能更新。有什么办法解决？一直重新下载安装么？

回复

司青玄 1月3日 11:49

谷歌浏览器是最好用的浏览器，当之无愧

回复

Becominger 1月3日 11:41

@我的印象笔记

回复

若-尘-无-悔 1月3日 11:40

@保存到为知笔记

回复

获得微博评论箱