HTTP 1.1

HTTP 1.0 的优化策略非常简单，就一句话：升级到 HTTP 1.1。完了！

• 持久化连接以支持连接重用；

• 分块传输编码以支持流式响应；

• 请求管道以支持并行请求处理；

• 字节服务以支持基于范围的资源请求； • 改进的更好的缓存机制。

对网络优化：

• 减少DNS查询

每次域名解析都需要一次网络往返，增加请求的延迟，在查询期间会阻塞请求。

• 减少HTTP请求

任何请求都不如没有请求更快，因此要去掉页面上没有必要的资源。

• 使用CDN

从地理上把数据放到接近客户端的地方，可以显著减少每次 TCP 连接的网络延迟，增加吞吐量。

• 添加Expires首部并配置ETag标签

相关资源应该缓存，以避免重复请求每个页面中相同的资源。 Expires 首部可用于指定缓存时间，在这个时间内可以直接从缓存取得资源，完全避免 HTTP 请求。 ETag 及 Last-Modified 首部提供了一个与缓存相关的机制，相当于最后一次更新的指纹或时间戳。

• Gzip资源

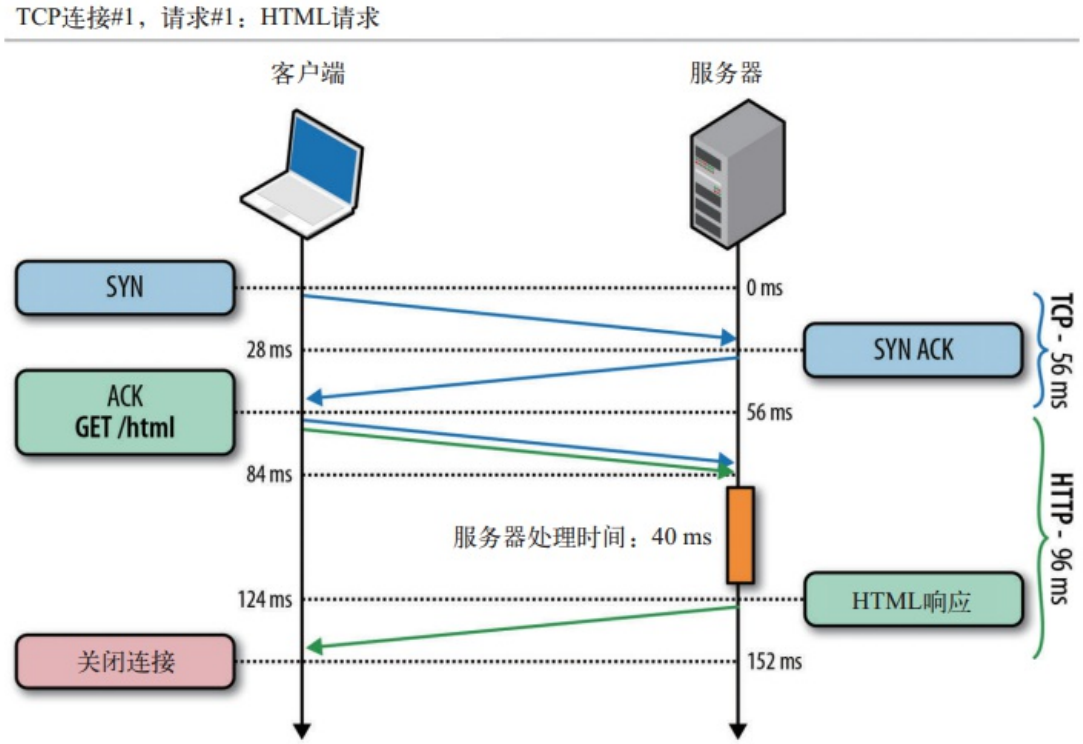
所有文本资源都应该使用 Gzip 压缩，然后再在客户端与服务器间传输。一般来说， Gzip 可以减少 60%~80% 的文件大小，也是一个相对简单（只要在服务器上配置一个选项），但优化效果较好的举措。

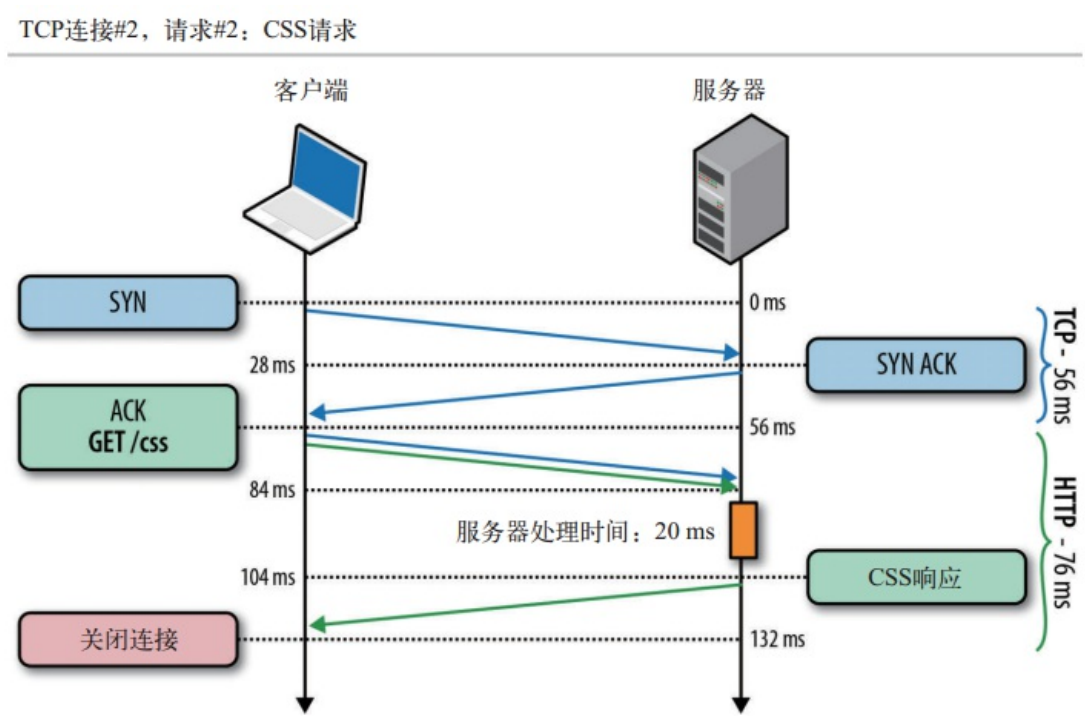
• 避免HTTP重定向

HTTP 重定向极其耗时，特别是把客户端定向到一个完全不同的域名的情况下，还会导致额外的 DNS 查询、 TCP 连接延迟，等等。

> 持久连接的优点

每个 TCP 连接开始都有三次握手，要经历一次客户端与服务器间完整的往返。最少等于两次网络往返的时间：**一次用于握手，一次用于请求和响应**。这是所有非持久 HTTP 会话都要付出的固定时间成本.



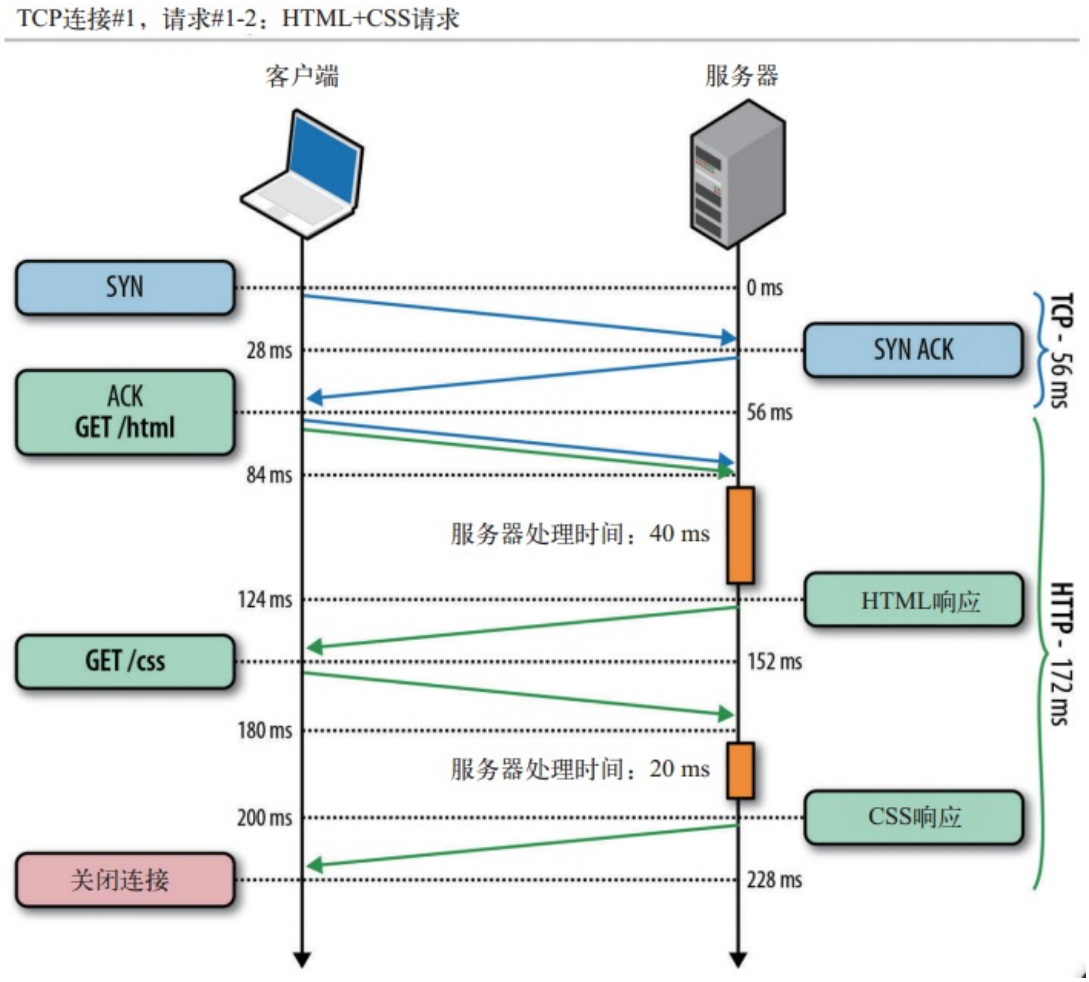


实际上，这时候最简单的优化就是重用底层的连接！添加对 HTTP 持久连接的支持，就可以避免第二次 TCP 连接时的三次握手、消除另一次 TCP 慢启动的往返，节约整整一次网络延迟。

在我们两个请求的例子中，总共只节约了一次往返时间。但是，更常见的情况是一次 TCP 连接要发送 N 次 HTTP 请求，这时：

• 没有持久连接，每次请求都会导致两次往返延迟；

• 有持久连接，只有第一次请求会导致两次往返延迟，后续请求只会导致一次往返延迟。

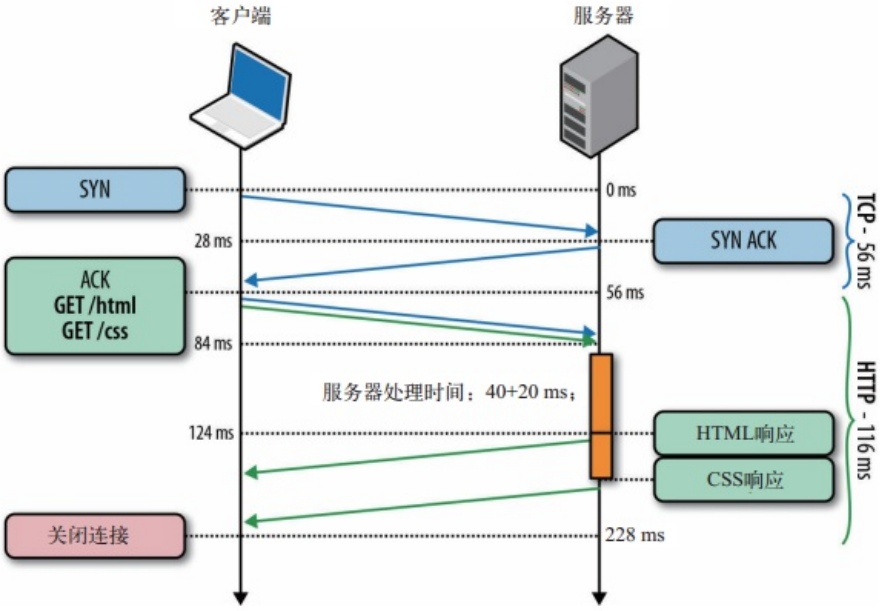


持久 HTTP 可以让我们重用已有的连接来完成多次应用请求，但多次请求必须严格满足先进先出（ FIFO）的队列顺序：发送请求，等待响应完成，再发送客户端队列中的下一个请求。 HTTP 管道是一个很小但对上述工作流却非常重要的一次优化。管道可以让我们把 FIFO 队列从客户端（请求队列）迁移到服务器（响应队列）。

HTTP管道

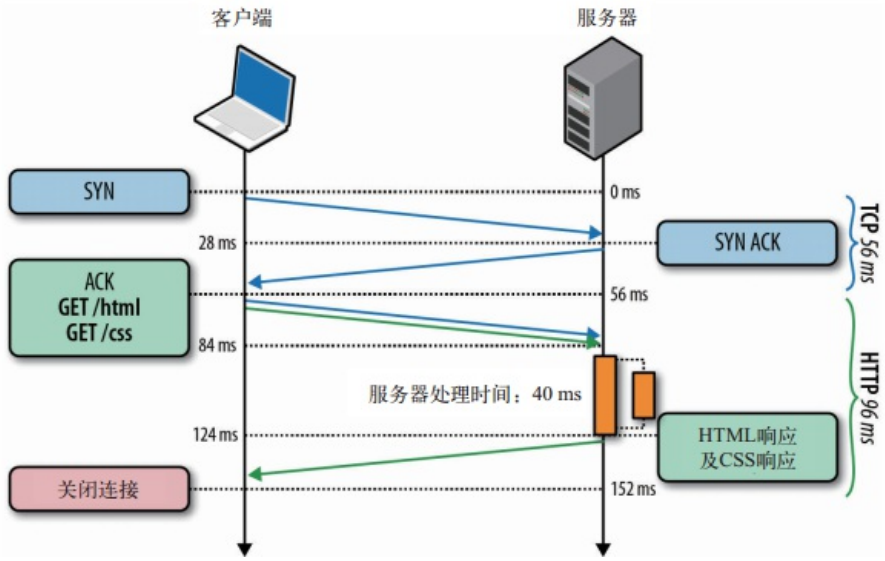
首先，服务器处理完第一次请求后，会发生了一次完整的往返：先是响应回传，接着是第二次请求。在此期间服务器空闲。如果服务器能在处理完第一次请求后，立即开始处理第二次请求呢？甚至，如果服务器可以并行或在多线程上或者使用多个工作进程，同时处理两个请求呢？

通过尽早分派请求，不被每次响应阻塞，可以再次消除额外的网络往返。这样，就从非持久连接状态下的每个请求两次往返，变成了整个请求队列只需要两次网络往返.



现在我们暂停一会，回顾一下在性能优化方面的收获。一开始，每个请求要用两个TCP 连接在使用持久连接后，避免了一次握手往返，最后，通过使用 HTTP 管道，又减少了两次请求之间的一次往返.

可惜的是，当我们想要采取这个优化措施时，发现了 HTTP 1.x 协议的一些局限性。HTTP 1.x 只能严格串行地返回响应。特别是， HTTP 1.x 不允许一个连接上的多个响应数据交错到达（多路复用），因而一个响应必须完全返回后，下一个响应才会开始传输。为说明这一点，我们可以看看服务器并行处理请求的情况



• HTML 和 CSS 请求同时到达，但先处理的是 HTML 请求；

• 服务器并行处理两个请求，其中处理 HTML 用时 40 ms，处理 CSS 用时 20 ms；

• CSS 请求先处理完成，但被缓冲起来以等候发送 HTML 响应；

• 发送完 HTML 响应后，再发送服务器缓冲中的 CSS 响应。

即使客户端同时发送了两个请求，而且 CSS 资源先准备就绪，服务器也会先发送HTML 响应，然后再交付 CSS。这种情况通常被称作**队首阻塞**，并经常导致次优化交付：不能充分利用网络连接，造成服务器缓冲开销，最终导致无法预测的客户端延迟。假如第一个请求无限期挂起，或者要花很长时间才能处理完，怎么办呢？在 HTTP 1.1 中，所有后续的请求都将被阻塞，等待它完成.

使用多个TCP连接

现实中，大多数现代浏览器，包括桌面和移动浏览器，都支持每个主机打开 6 个连接。 大家可以做个试验，在一个主机上同时打开 6 个并行下载，然

后再打开第 7 个下载请求，这个请求会挂 起，直到前面的请求完成才会执行。

域名分区

HTTP 1.x 协议的一项空白强迫浏览器开发商引入并维护着连接池，每个主机最多 6个 TCP 流。好的一方面是对这些连接的管理工作都由浏览器来处理。作为应用开发者，你根本不必修改自己的应用。不好的一方面呢，就是 6 个并行的连接对你的应用来说可能仍然不够用。

当然，天下没有免费的午餐，域名分区也不例外：每个新主机名都要求有一次额外的 DNS 查询，每多一个套接字都会多消耗两端的一些资源，而更糟糕的是，站点作者必须手工分离这些资源，并分别把它们托管到多个主机上。

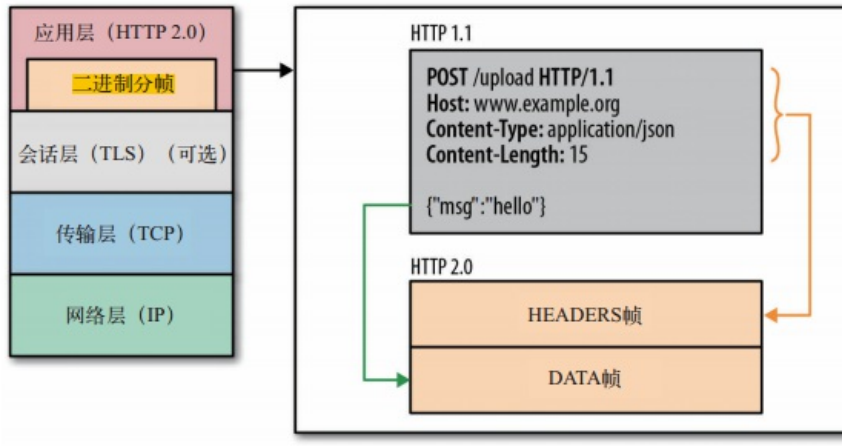
HTTP 2.0

HTTP 2.0 可以让我们的应用更快、更简单、更健壮!

HTTP 2.0 把很多以前我们针对 HTTP 1.1 想出来的“歪招儿”一笔勾销，把解决那些问题的方案内置在了传输层中。不仅如此， HTTP 2.0 还为我们进一步优化应用、改进性能，提供了全新的机会！

HTTP 2.0 的目的就是通过支持请求与响应的多路复用来减少延迟，通过压缩 HTTP首部字段将协议开销降至最低，同时增加对请求优先级和服务器端推送的支持。

HTTP 2.0 不会改动 HTTP 的语义。 HTTP 方法、状态码、 URI 及首部字段，等等这些核心概念一如往常。但是， HTTP 2.0 修改了格式化数据（分帧）的方式，以及客户端与服务器间传输这些数据的方式。HTTP 2.0 增加了新的**二进制分帧数据层**



简言之， HTTP 2.0 致力于突破上一代标准众所周知的性能限制，但它也是对之前1.x 标准的扩展，而非替代。

为了说明这个过程，我们需要了解 HTTP 2.0 的两个新概念。

• 流

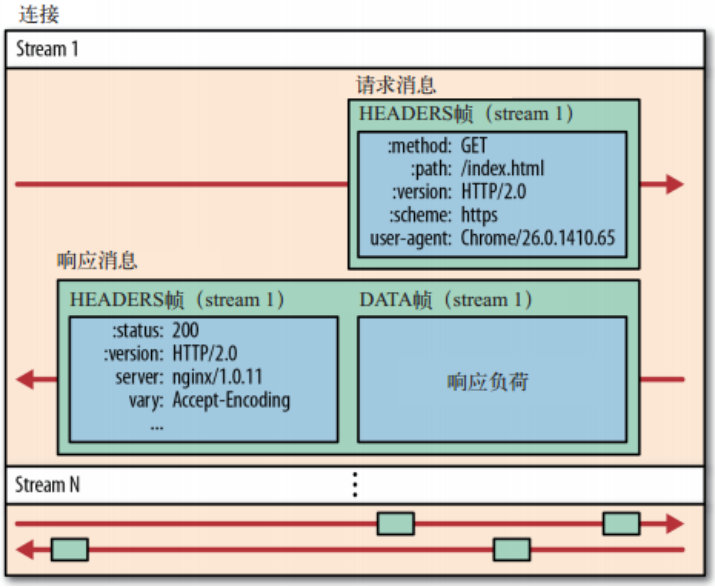
已建立的连接上的双向字节流。 流是连接中的一个虚拟信道，可以承载双向的消息；每个流都有一个唯一的整数标识符（ 1、 2...N）。

• 消息

与逻辑消息对应的完整的一系列数据帧。消息是指逻辑上的 HTTP 消息，比如请求、响应等，由一或多个帧组成。

• 帧

HTTP 2.0 通信的最小单位，每个帧包含帧首部，至少也会标识出当前帧所属的流。承载着特定类型的数据，如 HTTP 首部、负荷，等等。



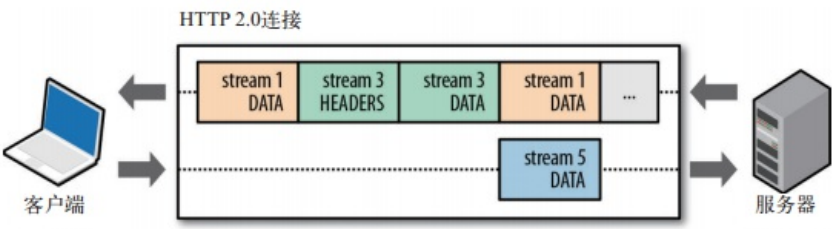
HTTP 1.x 以换行符作为纯文本的分隔符，而 HTTP2.0 将所有传输的信息分割为更小的消息和帧，并对它们采用二进制格式的编码.

所有 HTTP 2.0 通信都在一个连接上完成，这个连接可以承载任意数量的双向数据流。相应地，每个数据流以消息的形式发送，而消息由一或多个帧组成，这些帧可以乱序发送，然后再根据每个帧首部的流标识符重新组装。

多向请求与响应

这是 HTTP 1.x 交付模型的直接结果，该模型会保证每个连接每次只交付一个响应（多个响应必须排队）。更糟糕的是，这种模型也会导致队首阻塞，从而造成底层 TCP 连接的效率低下。

HTTP 2.0 中新的二进制分帧层突破了这些限制，实现了多向请求和响应：客户端和服务器可以把 HTTP 消息分解为互不依赖的帧，然后乱序发送，最后再在另一端把它们重新组合起来。



把 HTTP 消息分解为独立的帧，交错发送，然后在另一端重新组装是 HTTP 2.0 最重要的一项增强。事实上，这个机制会在整个 Web 技术栈中引发一系列连锁反应，从而带来巨大的性能提升，因为：

- 可以并行交错地发送请求，请求之间互不影响；
- 可以并行交错地发送响应，响应之间互不干扰；
- 只使用一个连接即可并行发送多个请求和响应；
- 消除不必要的延迟，从而减少页面加载的时间；
- 不必再为绕过 HTTP 1.x 限制而多做很多工作；

总之， HTTP 2.0 的二进制分帧机制解决了 HTTP 1.x 中存在的队首阻塞问题，也消除了并行处理和发送请求及响应时对多个连接的依赖。结果，就是应用速度更快、开发更简单、部署成本更低。支持多向请求与响应，可以省掉针对 HTTP 1.x 限制所费的那些脑筋和工作，比如拼接文件、图片精灵、域名分区.

请求优先级

把 HTTP 消息分解为很多独立的帧之后，就可以通过优化这些帧的交错和传输顺序，进一步提升性能。为了做到这一点，每个流都可以带有一个 31 比特的优先值：

• 0 表示最高优先级； • 2^31-1 表示最低优先级。

浏览器在渲染页面时，并非所有资源都具有相同的优先级： HTML 文档本身对构建 DOM 不可或缺， CSS 对构建 CSSOM 不可或缺，而 DOM 和 CSSOM 的构建都可能受到 JavaScript 资源的阻塞，其他资源（如图片）的优先级都可以降低

服务器推送

HTTP 2.0 新增的一个强大的新功能，就是服务器可以对一个客户端请求发送多个响应。换句话说，除了对最初请求的响应外，服务器还可以额外向客户端推送资源，而无需客户端明确地请求。因此，客户端可以限定推送流的数量，或者通过把这个值设置为 0 而完全禁用服务器推送。

为什么需要这样一个机制呢？通常的 Web 应用都由几十个资源组成，客户端需要分析服务器提供的文档才能逐个找到它们。那为什么不让服务器提前就把这些资源推送给客户端，从而减少额外的时间延迟呢？服务器已经知道客户端下一步要请求什么资源了，这时候服务器推送即可派上用场。事实上，如果你在网页里嵌入过 CSS、JavaScript，或者通过数据 URI 嵌入过其他资源（参见 11.7 节“嵌入资源”），那你就已经亲身体验过服务器推送了.

• 客户端可以缓存推送过来的资源；

• 客户端可以拒绝推送过来的资源；

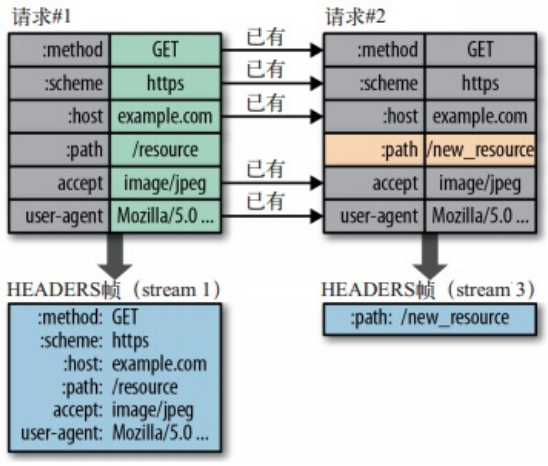
• 推送资源可以由不同的页面共享；

• 服务器可以按照优先级推送资源。

所有推送的资源都遵守同源策略。换句话说，服务器不能随便将第三方资源推送给客户端，而必须是经过双方确认才行。

首部压缩

TTP 2.0 连接的两端都知道已经发送了哪些首部，这些首部的值是什么，从而可以针对之前的数据只编码发送差异数据.



第二个请求只需要发送变化了的路径首部（ :path），如果请求中不包含首部（例如对同一资源的轮询请求），那么首部开销就是零字节。此时所有首部都自动使用之前请求发送的首部！