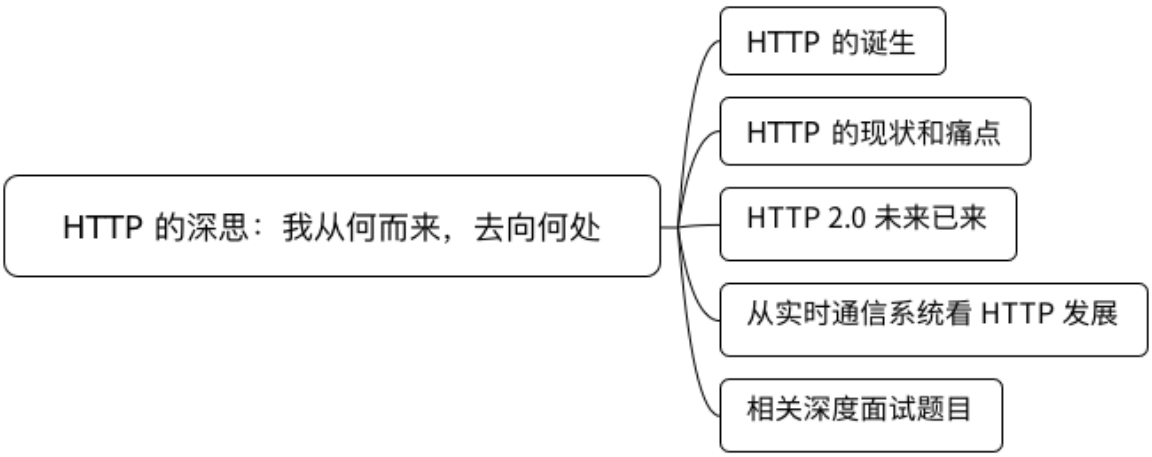
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

HTTP 的深思：我从何而来，去向何处

在前面三讲中，分别了解了网络的基本内容、HTTP 的特性，尤其是缓存特性。这一讲让我们从历史的角度来审视 HTTP，探究 HTTP 的演进是为了更好的应用，更理解网络这个宏大概念中的一环。

主要内容如下：



HTTP 的诞生

HTTP 从何处来?

HTTP 协议诞生自 1989 年（可能比很多开发者年纪要大），第一版本是 HTTP 0.9，但 HTTP 0.9 并不是一个正式标准；直到 1996 年，根据 RFC 1945，HTTP 1.0 成为 IEFT 标准，1999 年，在 RFC 2616 中发布了 HTTP 1.1。

版本路线如下：

HTTP/0.9

HTTP/1.0

HTTP/1.1

HTTP/2

但是需要注意的是，HTTP 和 JavaScript 一样，说到底还是需要浏览器的支持和实现。到那时每个浏览器或服务器对于该协议的每个方面并不能完全一致实现，因此还是有着细微用户体验与标准规范不一致的情况。

这些「陈年旧事」我们不再过多回顾，下面来看一下 HTTP 的现状和发展痛点。

HTTP 的现状和痛点

HTTP 2.0 于 2015 年发布，考虑到发布后的落地情况，以及现在各大厂商的应用情况，我们认可 HTTP 1.1 作为现状分析。

HTTP 1.1 是划时代的，它解决了 HTTP 1.0 时代最重要的两个大问题：

TCP 连接无法复用，每次请求都需要重新建立 TCP 通道，也就要重复三次握手和四次挥手的情况；就是说*每个 TCP 连接只能发送一个请求*。

对头阻塞，每个请求都要过「独木桥」，桥宽为一个请求的宽度；也就是说，即使多个请求并行发出，也只能一个接一个地进行请求排队。

HTTP 1.1 的改进点「对症下药」，它引入了：

长连接：HTTP 1.1 支持长连接（Persistent Connection），且默认就开启了 Connection：keep-alive，这样在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。业界的成熟方案，如 Google 的 protobuf。

管线化：在长连接的基础上，管线化（HTTP Pipelining）使得多个请求使用同一个 tcp 连接使请求并按照并行方式成为可能：多个请求同时发起，无需等待上一个请求的回包。但是需要注意，管线化只是让请求并行，但并没有

从根本上解决对头阻塞问题，因为响应仍然要遵循先进先出的原则，第一个请求的回包发出之后，才会响应第二个请求。同时，浏览器供应商很难实现管道，而且大多数浏览器默认禁用该特性，有的甚至完全删除了它。

除此以外，HTTP 1.1 还有一些创造性的改进，比如：

缓存处理

带宽优化及网络连接的使用，比如，range 头，支持断点续传功能，这个内容我们已经在前面的内容中进行了介绍

错误通知的增强，响应码的增强

Host 头处理，请求消息中如果没有 Host 头域会报告一个错误

基于 HTTP 1.1 的变革，一些成熟的方案也应运而出，比如：

http long-polling

http streaming

websocket

这些内容我们会在本专栏「从实时通信系统看 HTTP 发展」部分进行介绍。

这么看来，HTTP 1.1 简直不要太完美！当然他还是有一些缺陷和痛点的。比如：

队头堵塞问题没有真正解决

明文传输，安全性有隐患

header 携带内容过多，增加了传输成本

默认开启 keep-alive 可能会给服务端造成更大的性能压力，比如对于一次性的请求（图片 CDN 服务），在文件被请求之后还保持了不必要的连接很长时

间

HTTP 2.0 未来已来

说起 HTTP 2.0，不得不提一下 SPDY 协议。2009 年，谷歌针对 HTTP 1.1 的一些问题，发布了 SPDY 协议。这个协议在 Chrome 浏览器上进行应用，并证明可行后，就成为了 HTTP 2.0 的基础，主要特性都在 HTTP 2.0 之中得到继承。但作为推动时代发展的产出，SPDY 说到底不会主宰时代而流行，我们暂不介绍更多，而把主要精力放在 HTTP 2.0 上。

HTTP 2.0 目标是显著改善性能，同时做到迁移透明。我们先来理解几个 HTTP 2.0 的相关前置基础概念：

帧：HTTP 2.0 中，客户端与服务器通过交换帧来通信，帧是基于这个新协议通信的最小单位。

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。

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标识符

最主要的特性如下。

二进制分帧

HTTP 2.0 的协议解析决定采用二进制格式，而非 HTTP 1.x 的文本格式，二进制协议解析起来更高效，这可以说是性能增强的焦点。新协议称为二进制分帧层（Binary Framing Layer），每一个请求都有这些公共字段：

Type：帧的类型，标识帧的用途

Length：整个帧的开始到结束大小

Flags：指定帧的状态信息

Steam Identifier：用于流控制，可以跟踪逻辑流的帧成员关系

Frame payload：r 请求正文

这些内容相对底层和细致，这里只需要大家明白：二进制协议将通信分解为帧的方式，这些帧交织在客户端与服务器之间的双向逻辑流中，这样就使得所有通信都在单个 TCP 连接上执行，而且该连接在整个对话期间一直处于打开状态。

请求/响应复用

上面提到，为每帧分配一个流标识符，这就可以在一个 TCP 连接上独立发送它们。此技术实现了完全双向的请求和响应消息复用，解决了队头阻塞的问题。换句话说：一个请求对应一个 stream 并分配一个 id，这样一个连接上可以有多个 stream，每个 stream 的 frame 可以混杂在一起，接收方可以根据 stream id 将 frame 再归属到各自不同的请求里面。

总结一下：所有的相同域名请求都通过同一个 TCP 连接并发完成。同一 TCP 中可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。这是真正意义上的多路复用。

报头压缩

报头压缩的实现方式是，要求客户端和服务器都维护之前看见的标头字段的列表。在第一个请求后，它仅需发送与前一个标头的不同之处，其他相同之处，服务器可以从标头的列表中恢复。

流优先化

消息帧通过流进行发送。我们提到每个流都分配了一个 id，同时也可以分配优先级。这样一来，服务端可以根据优先级确定它的处理顺序。

服务器推送

当一个客户端主动请求资源 K，如果这时候服务器知道它很可能也需要资源 M，那么服务器可以主动将资源 M 推送给客户端。当客户端真的请求 M 时，便可以从缓存中读取。

这里有一个问题是：如何管理让服务器推送资源而不会让客户端过载?

事实上，针对服务度希望发送的每个资源，服务端会发送一个 PUSH_PROMISE 帧，但客户端可通过发送 RST_STREAM 帧作为响应来拒绝推送。

流控制

流控制允许接收者主动示意停止或减少发送的数据量。比如一个视频应用，在观看一个视频流时，服务器会同时向客户端发送数据。如果视频暂停，客户端会通知服务器停止发送视频数据，以避免耗尽它的缓存。

从实时通信系统看 HTTP 发展

从上面的知识我们看出，传统的浏览器和 HTTP 协议，早期只能通过客户端主动发送请求，服务端应答回复请求来实现数据交互。但是在一些监控、Web 在线通讯、即时报价系、在线游戏等场景中，都需要将后台发生的变化主动地、实时地传送到浏览器端，而不需要用户手动地刷新页面。为了达到这个目的，一些 hack 方法或「官方」方法便应用而生。

Ajax polling

轮询是最简单无脑的方案：客户端通过定期发送 ajax 请求，服务端受理请求立刻返回数据。这种方式保证了数据的相对实时性，具有很好的浏览器兼容性和简单性。但是缺点也明显：数据延迟取决于轮询频率，如果频率过高，就会产生大部分无效请求；反转频率过低，数据的实时性较差。同时，服务端的压力较大，也浪费了带宽流量。

Ajax long polling

在短轮询的基础上，长轮询的实现思路是：客户端通过 ajax 发起请求，服务器在接到请求后不马上返回，而是保持住这个连接，等待数据的更新。当有数据要推送给客户端时，才发送目标数据给客户端，请求返回。客户端收到响应之后，马上再发起一个新的请求给服务器，周而复始。

这样的长轮询能够有效减少轮询次数，而且延迟大大降低，但服务端需要保持大量连接，也是一种消耗。

Comet streaming

Comet streaming 技术又被称为 Forever iframe，这种技术听上去更加 hack：需要我们动态载入一个隐藏的 iframe 标签，iframe 的 src 指向请求的服务器地址。同时客户端准备好一个处理数据的函数，在服务端通过 iframe 和客户端通信时，服务端返回类似 script 标签的文本，客户端解析为 JavaScript 脚本，并调用用预先准备好的函数，将数据传递给 parent window，类似 Jsonp 的实现原理一样：

这样的实现也不算复杂，但说到底也是一种奇怪的 hack。

Ajax multipart streaming

这种方式就用到了 HTTP 1.1 的 multipart 特性：客户端发送请求，服务端保持住这个连接，利用 HTTP 1.1 的 chunked encoding 机制（分块传输编码），将数据传递给客户端，直到 timeout 或者客户端手动断开。

这种方法至少属于官方规范，但是就像前面知识所介绍的那样，HTTP 1.1 的 multipart 特性并没有被更广泛的浏览器接受并实现，它需要浏览器支持 multipart 特性。

Websocket

WebSocket 是 HTML5 开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。依靠这种技术可以实现客户端和服务器端的长连接，双向实时通信。

我们该如何理解 Websocket 和 HTTP 协议呢?

HTTP 和 WebSocket 都是应用层协议，都是基于 TCP 协议来传输数据的。WebSocket 依赖一种升级的 HTTP 协议进行一次握手，握手成功后，数据就直接从 TCP 通道传输。

这样一来，连接的发起端还是客户端，但是一旦 Websocket 连接建立，客户端和服务端任何一方都可以向对方发送数据。

Webscoket 无疑是强大的，但是它也错过了浏览器为 HTTP 提供的一些服务，需要开发中在使用时自己实现。因此 WebSocket 并不能取代 HTTP。

由此看出，HTTP 的发展不是封闭的，而是吸取了「民间方案」和各种应用技术所长。尤其是 HTTP 2.0 更是一个极大的补充和优化。在下一部分，我们对 HTTP 和 Tcp. 以及相关内容以面试题的方式在进行巩固。

相关深度面试题目

> 题目一：「HTTP 连接分为长连接和短连接，而我们现在常用的都是 HTTP 1.1，因此我们用的都是长连接。」这种说法正确吗?

其实这句话只对了后半句：我们现在大多应用 HTTP 1.1，因此用的都是长连接，这种说法勉强算对，因为 HTTP 1.1 默认 Connection 为 keep-alive。但是 HTTP 协议并没有长连接、短连接之分，所谓的长短连接都是在说 TCP 连接，TCP 连接是一个双向的通道，它是可以保持一段时间不关闭的，因此 TCP 连接才有真正的长连接和短连接这一说。

这个可以回到网络分层的话题上，HTTP 协议说到底是应用层的协议，而 TCP 才是真正的传输层协议，只有负责传输的这一层才需要建立连接。

> 题目二：长连接是一种永久连接吗?

事实上，长连接并不是永久连接的，在长连接建立以后，如果一段时间内没有 HTTP 请求发出，这个长连接就会断掉。这个超时的时间可以在 header 中进行设置。

> 题目三：现代浏览器在与服务器建立了一个 TCP 连接后是否会在一个 HTTP 请求完成后断开？什么情况下会断开?

在 HTTP 1.0 中，一个服务器在发送完一个 HTTP 响应后，会断开 TCP 链接。但 HTTP 1.1 中，默认开启 Connection：keep-alive，浏览器和服务器之间是会

维持一段时间的 TCP 连接，不会一个请求结束就断掉。除非显式声明：Connection: close。

> 题目四：一个 TCP 连接可以对应几个 HTTP 请求，这些 HTTP 请求发送是否可以一起发送?

不管是 HTTP 1.0 还是 HTTP 1.0，单个 TCP 连接在同一时刻只能处理一个请求，意思是说：两个请求的生命周期不能重叠，任意两个 HTTP 请求从开始到结束的时间在同一个 TCP 连接里不能重叠。也就是上面说的「队头阻塞」。

虽然 HTTP 1.1 规范中规定了 Pipelining 来试图解决这个问题，但是这个功能在浏览器中默认是关闭的。

因此，在 HTTP 1.1 中，一个支持持久连接的客户端可以在一个连接中发送多个请求（不需要等待任意请求的响应），收到请求的服务器必须按照请求收到的顺序发送响应。HTTP 2.0 中，由于 Multiplexing 特点的存在，多个 HTTP 请求可以在同一个 TCP 连接中并行进行。

总结

这一讲中，我们从 HTTP 的发展角度，解析了当前 HTTP 协议的现状和痛点，并详细介绍了 HTTP 2.0 相关内容。

后面部分，从实时通信系统网络协议层面的解析，又一次巩固了相关知识。

到此，我们在理论层面已经有了必要的知识储备。**在整个课程完结时，感兴趣的同学可以跟我亲自动手，实践一下 HTTP 2.0，让我们从实战角度，探究一下「HTTP 2.0」的众多特性到底能不能优化应用。**

点击查看下一节

不可忽视的前端安全 - 单页应用鉴权设计